Progetto WebApplication (sessione estiva 2021)

Overview

Le specifiche del progetto si trovano in `/static/doc/Progetto Esame WebApplication - Giugno 2021.pdf`.

Frameworks

Per questo progetto ho scelto di utilizzare:

- **Django 3.2.4** come **framework lato server**. La scelta è caduta su questo framework perché:
    - ha una **struttura semplice e ordinata**. Ho apprezzato molto il fatto che ogni sezione della webapp principale possa essere trattata come una webapp più piccola, con le sezioni (quasi) totalmente indipendenti tra loro. Cioè permette una **migliore organizzazione del lavoro**. Inoltre il framework si compone di **pochi file**, il che rende molto più **agevole lo sviluppo** delle applicazioni.
    - ha una **curva di apprendimeno abbastanza rapida**, anche se si parte da zero.
    - c'è abbondanza di **documentazione** e supporto in Rete. Molto utile, soprattutto per chi inizia, la documentazione presente sulla MDN Web Docs di Mozilla.
    - grazie a **Django REST Framework**, installato come plugin, si riescono a scrivere **API RESTful in maniera rapida ed efficace**.
    - utilizza **Python** come linguaggio. Nonostante la mia esperienza in materia di sviluppatore Web sia basata interamente su Php, ho scelto un framework in Python (pur non avendo esperienza con questo linguaggio) per una questione di **continuità** con altri progetti di altri corsi, progetti sviluppati (per scelta più o meno libera) appunto in Python. In questo modo acquisisco **ulteriore esperienza sul linguaggio**, oltre a **imparare un metodo di lavoro differente** rispetto al quale ero abituato.
    - utilizza un **sistema di autenticazione integrato**. Pur non essendo obbligatorio il suo utilizzo, è molto comodo e permette anche di inserire gli utenti in gruppi e assegnare loro **permissions specifiche** sulla webapp. Nel progetto in questione, ho utilizzato questo sistema per gestire l'autenticazione degli utenti e la loro appartenenza ai gruppi.
    - c'è la possibilità di **modificare la struttura della base di dati** operando direttamente sui model, grazie all'ORM (non serve più entrare nel prompt del DBRM e dare i comandi SQL a mano) e al sistema delle **migrations**.

  Questi sono i **pro** della mia scelta. C'è però anche qualche **contro**, che ho potuto verificare durante lo sviluppo. I principali riguardano l'**ORM** e sono:

    - la **minor potenza rispetto al raw SQL nelle query al database**. Pur essendo ORM obiettivamente più facile e immediato di SQL quando si ha a che fare con interrogazioni semplici, le stesse query in ORM diventano di **difficile gestione** quando queste si complicano (ad esempio quando si deve effettuare una o più *join* tra tabelle diverse). Per quel che è la mia esperienza pregressa e per quel che è stata l'esperienza con

Django, trovo ancora più potente (ed efficiente) effettuare query direttamente in SQL. Ma con una maggiore esperienza sull'ORM di Django probabilmente il mio giudizio sarebbe stato differente.

    - la **gestione di database separati**, soprattutto se il database dei dati non è quello di default. Pur potendo specificare la base di dati da utilizzare direttamente *inline* nelle query in ORM, nelle chiamate alle API in scrittura queste si sono rivelate spesso inutili (ottenevo puntualmente *relation nome_tabella> not found* pur avendo specificato correttamente dove volevo scrivere). L'unica soluzione al problema si è rivelata il **routing dei database**, soluzione non sempre elementare e che comporta modifiche alla configurazione del sistema.
    - il **caching delle query**. Mi è capitato di dover modificare alcuni dati all'interno della database della parte dati direttamente dal prompt di PostgreSQL a causa della presenza di spazi bianchi alla fine delle stringhe, questo dopo aver fatto svariate prove per testare il funzionamento corretto della webapp in quanto questi spazi causavano talvolta problemi. Nonostante le modifiche ai dati, rieseguendo la query dalla webapp continuavo ad ottenere il dato non aggiornato, appunto per via del meccanismo di caching delle query che Django effettua. Ho dovuto risolvere il problema inserendo all'interno del codice i vari metodi di *trim* delle stringhe, sia lato Django, sia lato Javascript.

- **Django REST Framework 3.12.4** (https://www.django-rest-framework.org/) per lo **sviluppo delle API RESTful**, come già accennato al punto precedente, con i seguenti **pro**:
    - **creazione rapida** delle API e supporto per autenticazione e *permissions*
    - **integrazione con l'ORM** di Django tramite i *serializers*

  e i seguenti **contro**:
    - **non molto intuitivo l'aspetto dei serializers**, almeno per quella che è stata la mia esperienza, soprattutto nella gestione delle **foreign keys**. Una volta compreso il meccanismo, però, il tutto diventa relativamente agevole da utilizzare
- **jQuery 3.6.0** come **framework lato client**. La scelta è caduta su jQuery invece che su altri framework perché:
    - **ho già esperienza** con questo framework e lo ritengo ancora il più adatto a me e all'uso che di solito faccio di javascript all'interno di una webapp
    - **semplifica** molto l'utilizzo di **javascript**
    - **facile da capire** e da imparare (soprattutto se con Javascript "puro" si ha qualche difficoltà...)

  Di **contro**, jQuery ha forse accusato un po' il passo dopo l'avvento di framework lato client più recenti e basati su template come Angular, VueJS e React ed è ora meno utilizzato rispetto a qualche anno fa risultando un po' **obsoleto**. Resta però, a mio avviso, ancora uno strumento molto valido, soprattutto per progetti medio-piccoli come questo.
- **Bootstrap 5.0** per garantire alla webapp un'**interfaccia grafica il più possibile gradevole** e, soprattutto, la sua **responsiveness**. Non penso di dover giustificare pro e contro di questa scelta, perchè mi pare ovvia la necessità di non dover "reinventare la ruota" perdendo molto tempo nella scrittura di CSS. Se servono dei CSS particolari per un particolare tipo di design, quasi sempre è sufficiente fare l'**overloading** di una classe CSS di Bootstrap già esistente o scrivere classi CSS di piccola entità, molto più facilmente gestibili.

Databases

Come da specifica, la base di dati non deve essere unica, bensì una per gestire gli account utenti e un'altra per gestire i dati dell'applicazione. Il DBMS utilizzato è **PostgreSQL 13** (a fronte di una richiesta minima che prevedeva la versione 12). Per verificare di avere installata la versione corretta sulla macchina su cui girerà la webapp, dare semplicemente

```
marcuzzo@LoHacker:~$ psql --version
psql (PostgreSQL) 13.2 (Debian 13.2-1)`
```

Le basi di dati utilizzate (e da creare se il progetto viene clonato da qualche parte) sono **progetto_default** (che gestirà l'autenticazione utente) e **progetto_dati** (che gestirà la parte dati dell'applicazione) e a cui avrà accesso l'utente PostgreSQL *progetto* (password: *progetto*). I dati da importare in *progetto_dati* si trovano in `/static/db_dump_dati.sql`, mentre per quanto riguarda *progetto_default* è necessario dare `$ python3 manage.py migrate` per creare le tabelle relative e procedere poi con la creazione degli utenti dall'interfaccia di admin di Django (può essere necessario dare prima il comando `$ python3 manage.py makemigrations`).

> **ATTENZIONE:** modificare le impostazioni dei PostgreSQL riguardanti il **formato della data**: di defaul è m/d/Y, ma a noi server **d/m/Y**! Inoltre, ricordarsi di rimuovere gli spazi bianchi a fine stringa presenti in tutte le tabelle di progetto_dati perchè possono creare problemi.

La **gestione della lettura/scrittura** sulle basi di dati avviene attraverso il **routing dei database**: lo si può verificare nel file **dbRoute.py** e relativa configurazione in *progetto/settings.py*.

Utenti

Gli utenti, come detto, sono creati dall'interfaccia di admin di Django (non dalla webapp in sè) che ne gestisce anche tutti i vari aspetti (gestione password, sessioni, ...).

Gli utenti sono divisi in tre gruppi: **customers**, **agents** e **managers**. Per ogni gruppo, questi sono i relativi utenti (se non è presente il dump, crearli manualmente):

- **customers**: username da **C00001** a **C00025** (tutti con password *clientecliente*)
- **agents**: username da **A001** a **A012** (tutti con password *agenteagente*)
- **managers**: username da **M001** a **M003** (tutti con password *managermanager*)

API

Orders

Creating

```
POST /api/orders/new/
```

Inserisce un nuovo ordine all'interno del database.

Nota: l'id dell'ordine nonché chiave primaria, *ord_num*, penso sia da intendere come autoincrementale: in questo caso l'API (o meglio, il relativo serializer) *non* si preoccupa di generare il suddetto id, ma deve essere esplicitamente passato assieme agli altri dati!

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `CanInsertModifyDeleteOrders`: l'utente deve far parte dei gruppi predefiniti *agents* oppure *managers* per poter effettuare l'operazione di inserimento.

Reading (orders list)

```
GET /api/orders/[?sort_by={[-]column}]
```

Ritorna la lista degli ordini. L'output differisce in base all'utente loggato sul sistema:

- se l'utente è di tipo **customer**, viene ritornato l'elenco di tutti gli ordini da lui effettuati, con l'indicazione dell'agent che ha gestito l'ordine;
- se l'utente è di tipo **agent**, viene ritornato l'elenco di tutti gli ordini dei customer da lui gestiti, con l'indicazione dei clienti relativa a ogni ordine;
- se l'utente è di tipo **manager**, viene ritornato l'intero elenco degli ordini presenti, con indicazione di agent e customer per ogni ordine.

Query String Parameters

`{[-]columm}`: indica la colonna per la quale si desidera l'ordinamento dei dati. In accordo con la sintassi di Django, il segno `-` davanti al nome della colonna indica un **ordinamento discendente**, mentre il semplice nome indica un **ordinamento ascendente**.

Returnable Fields

Per ogni ordine, l'API ritorna i seguenti campi:

- `ord_num` (id dell'ordine)
- `ord_amount`
- `advance_amount`
- `ord_date`
- `cust_code` (codice del customer che ha effettuato l'ordine)
- `agent_code` (codice dell'agent che gestisce l'ordine)
- `cust_name` (nome del cliente)
- `agent_name` (nome dell'agente)
- `ord_description`

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `CanView`: l'utente deve far parte di uno dei tre gruppi predefiniti (*customers*, *agents*, *managers*) per poter effettuare l'operazione di visualizzazione.

Reading (single order)

```
GET /api/orders/{ord_num}/
```

Ritorna il singolo ordine, indicato per numero d'ordine.
`TODO`*: migliorare la query affinché un customer non possa visualizzare altri ordini oltre ai propri*

Returnable Fields

- `ord_num` (id dell'ordine)
- `ord_amount`
- `advance_amount`
- `ord_date`
- `cust_code` (codice del customer che ha effettuato l'ordine)
- `agent_code` (codice dell'agent che gestisce l'ordine)
- `cust_name` (nome del cliente)
- `agent_name` (nome dell'agente)
- `ord_description`

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `CanView`: l'utente deve far parte di uno dei tre gruppi predefiniti (*customer*. *agents*, *managers*) per poter effettuare l'operazione di visualizzazione.

Updating

```
PUT /api/orders/update/{ord_num}/
```

Aggiorna un ordine, specificato da `{ord_num}`.

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `CanInsertModifyDeleteOrders`: l'utente deve far parte dei gruppi predefiniti *agents* oppure *managers* per poter effettuare l'operazione di aggiornamento.

Deleting

```
DELETE /api/orders/delete/{ord_num}/
```

Elimina un ordine, specificato da `{ord_num}`.

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `CanInsertModifyDeleteOrders`: l'utente deve far parte dei gruppi predefiniti *agents* oppure *managers* per poter effettuare l'operazione di aggiornamento.

Customers

Creating

Questa operazione non è disponibile.

#Reading (customers list)

```
GET /api/customers/[?sort_by={[-]column}]
```

Ritorna la lista degli ordini. L'output differisce in base all'utente loggato sul sistema:

- se l'utente è di tipo **customer**, non può visualizzare l'elenco dei clienti
- se l'utente è di tipo **agent**, viene ritornato l'elenco di tutti i customer da lui gestiti
  -- se l'utente è di tipo **manager**, viene ritornato l'intero elenco dei customer, con indicazione dell'agent associato

Query String Parameters

`{[-]columm}`: indica la colonna per la quale si desidera l'ordinamento dei dati. In accordo con la sintassi di Django, il segno `-` davanti al nome della colonna indica un **ordinamento discendente**, mentre il semplice nome indica un **ordinamento ascendente**.

Returnable Fields

Per ogni customer, l'API ritorna i seguenti campi:

- `cust_code` (id del customer)
- `cust_name`
- `cust_city`
- `working_area`
- `cust_country`
- `grade`
- `opening_amt`
- `receive_amt`
- `payment_amt`
- `outstanding_amt`
- `phone_no`
- `agent_code`
- `agent_name` (nome dell'agent associato al customer)

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `IsAgentOrManager`: l'utente deve far parte del gruppo predefinito *agents* oppure *managers* per poter effettuare l'operazione di visualizzazione.

Reading (single customer)

```
GET /api/customers/{cust_code}/
```

Ritorna il singolo customer, indicato per codice.

Returnable Fields

- `cust_code` (id del customer)
- `cust_name`
- `cust_city`
- `working_area`
- `cust_country`
- `grade`
- `opening_amt`
- `receive_amt`
- `payment_amt`
- `outstanding_amt`
- `phone_no`
- `agent_code`
- `agent_name` (nome dell'agent associato al customer)

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `IsAgentOrManager`: l'utente deve far parte del gruppo predefinito *agents* oppure *managers* per poter effettuare l'operazione di visualizzazione.

Updating

Questa operazione non è disponibile.

Deleting

Questa operazione non è disponibile.

Agents

Creating

Questa operazione non è disponibile.

Reading (agents list)

```
GET /api/agents/[?sort_by={[-]column}]
```

Ritorna la lista degli agenti. Solo gli utenti di tipo **managers** possono visualizzare la lista completa degli agenti

Query String Parameters

`{[-]columm}`: indica la colonna per la quale si desidera l'ordinamento dei dati. In accordo con la sintassi di Django, il segno `-` davanti al nome della colonna indica un **ordinamento discendente**, mentre il semplice nome indica un **ordinamento ascendente**.

Returnable Fields

Per ogni agent, l'API ritorna i seguenti campi:

- `agent_code` (id dell'agent)
- `agent_name`
- `working_area`
- `commission`
- `phone_no`
- `country`

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `IsManager`: l'utente deve far parte del gruppo predefinito *managers* per poter effettuare l'operazione di visualizzazione.

Reading (single agent)

```
GET /api/agents/{agent_code}/
```

Ritorna il singolo customer, indicato per codice.

Returnable Fields

Per ogni agent, l'API ritorna i seguenti campi:

- `agent_code` (id dell'agent)
- `agent_name`
- `working_area`
- `commission`
- `phone_no`
- `country`

Permissions

- `IsAuthenticated`: l'utente deve aver effettuato il login per accedere alla risorsa
- `CanView`: l'utente deve far parte di uno dei tre gruppi predefiniti (*customer*. *agents*, *managers*) per poter effettuare l'operazione di visualizzazione.

Updating

Questa operazione non è disponibile.

Deleting

Questa operazione non è disponibile.

API Testing

Per il **testing delle API** ho utilizzato **Postman**: https://www.postman.com/downloads/ .

Questo tool si è dimostrato molto potente ed efficiente, nonché facile da utilizzare. Di contro, necessita di un account (gratuito) e non zippato pesa oltre 400 MB

WCAG 2.1 AA Testing

Per i **test di accessibilità WCAG 2.1** ho utilizzato **a11y-sitechecker** (https://github.com/forsti0506/a11y-sitechecker).

Dopo l'installazione, configurare il tool editando `~/node_modules/a11y-sitechecker/lib/utils/setup-config.js`. La configurazione non è esattamente delle più intuitive, ad ogni modo quella usata per testare il progetto è la seguente, all'interno della funzione `setupConfig()` :

```
const config = {
    json: true,
    resultsPath: '/home/marcuzzo/UniVr/WebApp/progetto/WCAG21aa-test/results',
    resultsPathPerUrl: '',
    axeConfig: {},
    threshold: 1000,
    imagesPath: '/home/marcuzzo/UniVr/WebApp/progetto/WCAG21aa-
test/results/images',
    timeout: 30000,
    debugMode: false,
    viewports: [
        {
            width: 1920,
            height: 1080,
        },
    ],
    resultTypes: ['violations', 'incomplete'],
    runOnly: ['wcag2a', 'wcag2aa', 'wcag21a', 'wcag21aa', 'best-practice',
'ACT'],
    crawl: false,
    name: '',
    urlsToAnalyze: ['http://3.143.240.119:8000/ordini/',
'http://3.143.240.119:8000/ordini/200123/',
'http://3.143.240.119:8000/clienti/C00022/'],
    login: {
        url: 'http://3.143.240.119:8000/auth/login/',
        steps: [
            {
                input: [
                    {
                        selector: '#id_username',
                        value: "C00022"
                    },
                    {
                        selector: '#id_password',
                        value: 'clientecliente'
                    }
                ],
                submit: '#sendlogin'
            }
        ]
      }
    };
```

Come si nota dalla configurazione, dovendo garantire il **livello AA** per le pagine accessibili ai clienti si sono testate:

- la **pagina principale** della sezione **ordini**
- la pagina che mostra **i dettagli di un singolo ordine** (appartenente al cliente)
- la pagina che mostra **i dati del cliente stesso**, accessibile dalla navbar in alto a destra

I **test effettuati** sono specificati dalla direttiva `runOnly`, mentre alla direttiva `resultTypes` sono specificati i tipi di risultato che verranno restituiti.

Dato che la webapp richiede l'**accesso dell'utente**, alla direttiva `login` ho impostato le credenziali di uno dei clienti (ne ho preso uno con più di un ordine effettuato, così da avere più elementi visualizzati sulla pagina e quindi un testing più accurato).

Una volta terminata la configurazione, **lanciare il sitechecker** spostandosi in `~/node_modules/a11y-sitechecker/bin/` e dando il comando `./a11y-sitechecker.js -T=1000`

I risultati verranno salvati in **formato JSON** all'interno della directory *WCAG21aa-test/results* per i risultati testuali e *WCAG21aa-test/results/images* per le immagini (ricordarsi di creare le directory prima, altrimenti i dati non verranno salvati).

Per riguarda il progetto in questione, **tutti i test** relativi alle linee guida *WCAG 2.1* inerenti al livello *AA* **sono stati superati**. L'unica *violation* riguarda una *best practice*, precisamente relativa all'*heading*: tale violazione, però, non inficia sull'accessibilità della webapp sviluppata.

Recap opzioni *manage.py*

- **python3 manage.py inspectdb [--database nome_database > myapp/models.py]**

Se ho già una base di dati, il comando la ispeziona e ne ricava un model (che potrebbe essere anche da sistemare, nel caso qualcosa non combaciasse). Il risultato lo salvo nel *models.py* dell'app voluta

- **python3 manager.py shell**

Questo torna comodo: apre una shell Python, per dare i comandi "on the fly"

- **python3 manage.py runserver [ip_address[:port]]**

Lancia il webserver interno. Di default resta in ascolto su *localhost* alla porta 8000

- **python3 manage.py makemigrations [--database nome_database]**

  Se sono state fatte delle modifiche ai model, prepara il necessario per l'esportazione delle modifiche alla base di dati reale

- **python3 manage.py migrate [--database nome_database]**

  Riporta le modifiche ai model sulla base di dati reale