Progetto APL
Dai, Ricicla e Sostieni

Studenti:

Beatrice Gagliano 1000000319

Cristina Gagliano 1000000320

Sommario

Obiettivi del progetto

Si vuole realizzare un'applicazione per la gestione dei rifiuti con l'obiettivo di eseguire un corretto smaltimento di essi, che deriva da una buona raccolta differenziata. In particolare, l'applicazione deve consentire agli enti di controllo di tenere traccia della spazzatura degli utenti che giunge al centro di rifiuto in modo da segnalare l'utente in caso di scorretto smaltimento. Ogniqualvolta l'utente deve “buttare" la spazzatura, si recherà presso i punti di raccolta dove gli verrà richiesto attraverso un sistema automatizzato di inserire una card a barre con Id utente (username e password) e la scelta del tipo di rifiuto (organico, carta, indifferenziata, plastica… etc.). Dopodiché, il sistema convalida la richiesta dell’utente solo quando verificherà l’identità dell’utente e i suoi privilegi. In seguito, è previsto il rilascio di un’etichetta che viene assegnata al sacco di spazzatura, e potrà il rifiuto essere lasciato al punto di raccolta. L'applicazione prevede diverse funzionalità: consente la registrazione e autenticazione di un utente, consente ad un utente di inserire un nuovo rifiuto sul punto di raccolta dopo aver verificato l'identità, prevede la possibilità da parte di un addetto al controllo (admin) di effettuare una segnalazione nell'apposito punto di controllo all’utente che ha eseguito uno scorretto smaltimento, infine prevede una procedura per vedere i dettagli dell’utente (dati del cliente, quantità di spazzatura, tipo di rifiuto, segnalazioni).

Casi d’uso

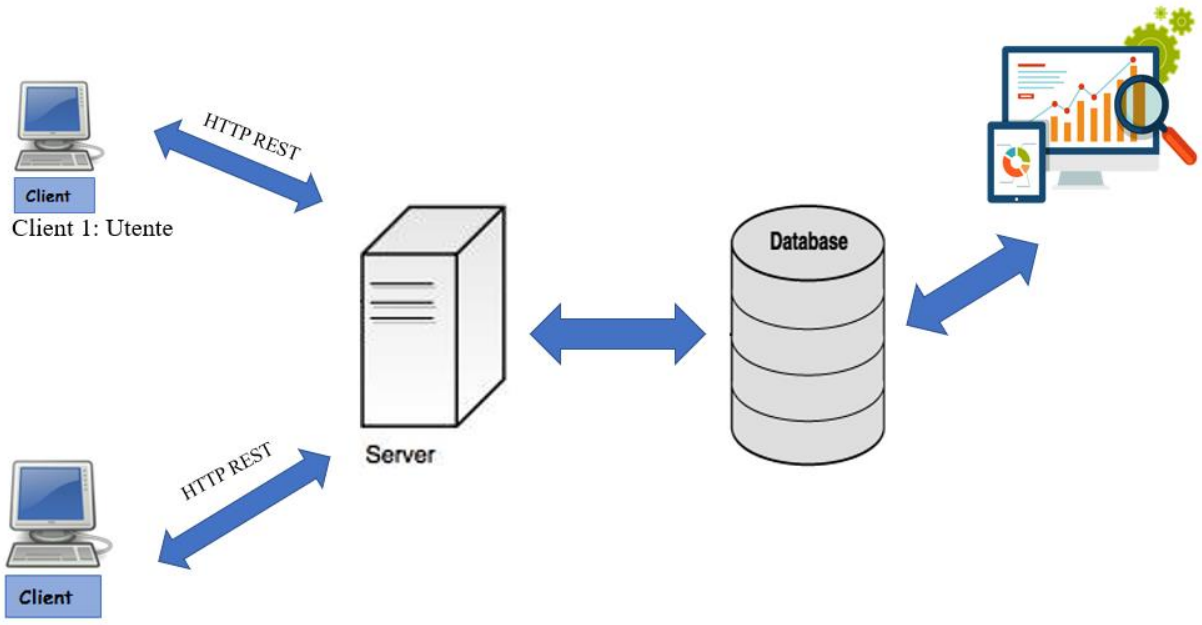
Dai, Ricicla e Sostieni consente agli enti di controllo di migliorare la gestione dei rifiuti.

Nello specifico:

- UC1: Registrazione di ogni utente e di ogni addetto al controllo (admin).
- UC2: Autenticazione (username, password) di ogni utente e addetto al controllo.
- UC3: L’utente deve poter inserire un nuovo rifiuto.
- UC4: L’utente deve poter visualizzare i rifiuti inseriti.
- UC5: L’addetto al controllo(admin) deve poter controllare e verificare ciascun rifiuto.
- UC6: L’addetto al controllo deve poter inserire le segnalazioni per errato smaltimento.
- UC7: L’addetto al controllo deve poter visualizzare le segnalazioni effettuate.
- UC8: L’utente deve poter visualizzare le segnalazioni ai rifiuti.

Per ognuno dei casi d’uso verrà fornita una spiegazione teorica seguita da una spiegazione tecnica.

Architettura dell'applicazione



L'architettura prevede la presenza di due tipologie di Client: l'utente e l'addetto al controllo. L'utente è colui che dopo essersi autenticato, inserisce il rifiuto sul punto di raccolta, l'addetto al controllo è colui che effettua il controllo dei rifiuti e crea le segnalazioni in caso di scorretto smaltimento.

Linguaggi di programmazione

I linguaggi di programmazione che si intendono usare sono **C#** per i moduli Client dell'applicazione, **Python** per il modulo Server ed **R** per il modulo Client relativo alle statistiche.

AppClientFinal (C#)

E' stata realizzata un'App Web ASP.NET Core (Model-View-Controller). L'architettura del modulo è divisa in tre parti: Models, Views, Controllers.

DrsProject (Python)

E' stato realizzato un Django Server implementato attraverso Django Rest framework che permette di creare le API.

Steps:

1. Creazione progetto "*DrsProject*".
2. Creazione ambiente virtuale per isolare localmente le dipendenze dei pacchetti:
   py – m venv venv
   venv\Scripts\activate.bat
3. Installazione Django e Django Rest framework nell'ambiente virtuale:
   pip install django
   pip install django rest framework
4. Inizializzazione progetto e applicazione
   django-admin startproject drs .
   django-admin startapp app
5. Migrazione
   python manage.py migrate

DataR (R)

Il modulo R contiene degli script che si connettono al database SQLite ed eseguono delle statistiche a partire dai dati presenti sul db.

Database (SQLite)

E' stato utilizzato SQLite che consente una facile integrazione con il Server in Python e con il modulo R.

Descrizione casi d'uso

UC1: Registrazione di ogni utente e addetto al controllo

Registrazione (C#)

1. Allo Startup, si apre la pagina relativa alla registrazione di un utente così come indicato nella classe '*Startup.cs'*.
2. Viene invocato il metodo '*Signin'* all'interno del '*RegistrationController.cs'* che ritorna la View '*Signin.cshtml'* collegata al Models '*Utente*'. Questo consente di visualizzare la pagina relativa alla registrazione.
3. Premuto il bottone '*Register*', viene invocato il metodo *public async Task<ActionResult> Signin (Utente credentials)* che consente la registrazione di un utente. Viene fatta una HttpPost all'endpoint `http://localhost:8000/utente/` in cui è in ascolto il Python Server.
4. Viene fatto un controllo su eventuali campi vuoti poiché tutti sono obbligatori. Nel caso di campi vuoti, verrà invocato il metodo '*Error*' che ritorna la View '*Error.cshtml'* dove l'utente verrà indirizzato e visualizzerà il messaggio di errore: "An error has occured, please try again".
5. Viene settata la variabile di sessione allo username dell'utente che si registra.
6. Viene effettuato un controllo sul campo isAdmin, se esso è True, significa che l'utente che si registra è un admin e dunque verrà indirizzato nella Control Section (Signal), altrimenti, l'utente sarà indirizzato alla pagina '*Home*'.

Registrazione (Python)

Il Django Server è in ascolto sulla porta 8080.

1. E' stato creato il modello *Utente* dentro il file '*models.py'* della nostra applicazione '*app'*. E' stato eseguito il comando '*python manage.py makemigrations app*' per generare la migrazione per tali modifiche e poi '*python manage.py migrate*' per applicare le modifiche al db.
2. Per serializzare e deserializzare le istanze in formato JSON è stato creata la classe *'UtenteSerializers'* dentro il file '*serializers.py'* in esso importiamo il modello creato e definiamo i campi. E' stato utilizzato il sistema di relazioni basato su collegamenti ipertestuali – url – con il metodo *HyperlinkedModelSerializer* fornito dal modulo serializers.
3. E' stata creata la classe *'UtenteViewSet'* dentro il file *'views.py'* dove viene usato il modulo *viewsets* fornito da rest_framework utile per la creazione delle URL.
4. E' stato creato il file '*urls.py*' dove diciamo all'applicazione quali sono le URL di accesso, dato che è stato usato il modulo *viewsets*, possiamo generare automaticamente le configurazioni URL per la nostra API, semplicemente registrando le nostre viste al modulo routers fornito da rest_framework.
   Nel caso dell'utente: `router. register(r'utente', views.UtenteViewSet)`

UC2: Autenticazione (username, password) di ogni utente e addetto al controllo.

Autenticazione (C#)

1. Viene invocato il metodo '*Login'* all'interno del '*RegistrationController.cs'* che ritorna la View '*Login.cshtml'* collegata al Models '*UtenteLogin*'. Questo consente di visualizzare la pagina relativa al login.
2. Premuto il bottone '*Login*', viene invocato il metodo *public async Task<ActionResult> Login (UtenteLogin credentials)* che consente l'autenticazione di un utente. Viene fatta una HttpGet all'endpoint `http://localhost:8000/utente/` in cui è in ascolto il Python Server.
3. Viene effettuato un controllo sull'username e password inseriti dall'utente, in particolare si verifica che lo username e la password inseriti siano tra quelli restituiti dalla HttpGet. Se il controllo va a buon fine l'utente viene indirizzato alla pagina *'Home'*.
4. Viene fatto un controllo su eventuali campi vuoti poiché tutti sono obbligatori. Nel caso di campi vuoti, verrà invocato il metodo '*Error*' che ritorna la View '*Error.cshtml'* dove l'utente verrà indirizzato e visualizzerà il messaggio di errore: "An error has occured, please try again".
5. Viene settata la variabile di sessione allo username dell'utente che si logga.
6. Viene effettuato un controllo sul campo isAdmin, se esso è True, significa che l'utente che si logga è un admin e dunque verrà indirizzato nella Control Section (Signal), altrimenti, l'utente sarà indirizzato alla pagina '*Home*'.

Autenticazione (Python)

Si veda il paragrafo Registrazione (Python).

UC3: Inserimento rifiuto

1. L'utente si reca presso lo sportello del punto di raccolta.
2. L'utente inserisce la propria tessera in un apposito lettore, e digita username e password.
3. Il Sistema effettua il riconoscimento dell'identità e verifica la correttezza dei dati immessi. Se i dati sono corretti, effettua l'accesso al Sistema Dai, Ricicla e Sostieni.
4. L'utente viene reindirizzato nella "**Home**" dove effettua l'inserimento del rifiuto. Il Sistema verifica che l'utente possa effettuare l'operazione.
5. L'utente seleziona il "tipo di rifiuto".
6. L'utente per ogni rifiuto genera un codice univoco e lo stampa su un'etichetta adesiva. L'utente provvede al fissaggio dell'etichetta sul rifiuto, e allo smaltimento.

I passi 5-6 verranno ripetuti finché serve.

7. Il Sistema genera una ricevuta al termine dell'azione.
8. L'utente ritira la ricevuta, estrae la tessera e va via.

Inserimento Rifiuto (C#)

1. Viene invocato il metodo '*Index'* all'interno di *'HomeController.cs'* che ritorna la View '*Index.cshtml'* collegata al Models '*Rifiuto*'. Questo consente di visualizzare la pagina di Home dove è possibile inserire il rifiuto.
2. L'utente seleziona il punto di raccolta ed il tipo di rifiuto dalle tabelle 'Collection Point' e 'Waste Type'.
3. Al click del bottone *'Confirm'* viene invocato il metodo *public async Task<ActionResult> Index (Rifiuto rifiuto)* che consente l'inserimento di un rifiuto dopo aver recuperato lo username dell'utente conservato nella variabile di sessione. Viene fatta una HttpPost all'endpoint http://localhost:8000/rifiuto/ in cui è in ascolto il Python Server.
4. Viene fatto un controllo su eventuali campi vuoti poiché tutti sono obbligatori. Nel caso di campi vuoti, verrà invocato il metodo '*Error*' che ritorna la View '*Error.cshtml'* dove l'utente verrà indirizzato e visualizzerà il messaggio di errore: "An error has occured, please try again".
5. Se tutto va a buon fine, l'utente rimane nella pagina Home dove potrà proseguire se vuole con l'inserimento di un nuovo rifiuto.

Inserimento Rifiuto (Python)

Il Django Server è in ascolto sulla porta 8080.

1. E' stato creato il modello *Rifiuto* dentro il file '*models.py'* della nostra applicazione '*app'*. Il modello Rifiuto mantiene mediante foreign key una relazione many-to-one con 3 modelli: Utente, Punto Raccolta, Tipo Rifiuto.
2. E' stato eseguito il comando '*python manage.py makemigrations app*' per generare la migrazione per tali modifiche e poi '*python manage.py migrate*' per applicare le modifiche al db.
3. Per serializzare e deserializzare le istanze in formato JSON è stata creata una classe *'RifiutoSerializers'* dentro il file '*serializers.py'* in esso importiamo il modello creato e definiamo i campi.
4. E' stata creata la classe *'RifiutoViewSet'* dove viene usato il modulo *viewsets* fornito da rest_framework utile per la creazione delle URL.
5. Viene registrata la vista al modulo routers fornito da rest_framework.

UC4: L'utente deve poter visualizzare i rifiuti inseriti.

WasteList (C#)

1. Viene invocato il metodo '*RifiutibyUser'* all'interno di *'RifiutoController.cs'* che ritorna la View '*RifiutibyUser.cshtml'* collegata al Models '*Rifiuto*' di tipo IEnumerable.
2. Alla chiamata del metodo viene innanzitutto recuperato lo username dell'utente dalla variabile di sessione.
3. Viene effettuata una HttpGet all'endpoint `http://localhost:8000/rifiuti/{id_utente}` in cui è in ascolto il Python Server.
4. Viene deserializzata la json string alla lista di Rifiuti tramite il metodo '*JsonConvert.DeserializeObject'.*
5. Viene restituita la View che prende come parametro il contenuto deserializzato.

WasteList (Python)

Il Django Server è in ascolto sulla porta 8080.

1. E' stata creata la classe *'RifiutiList'* dentro il file *views.py* dove viene usato il modulo *generics* fornito da rest_framework che restituisce l'intero set di query per un gestore di modelli.
2. Attraverso l'override del metodo *get_queryset()* è possibile filtrare il set di query di qualsiasi view che include sottoclassi *GenericAPIView*.
3. E' stato effettuato un filtraggio in base all'URL, creando una vista che restituisce un set di query filtrato dallo username dell'URL.
4. Viene registrata la vista all'interno degli *urlpatterns*:
```
url('^rifiuti/(?P<id_utente>.+)/$', RifiutiList.as_view()).
```

UC5: Controllo rifiuto

1. L'Addetto inserisce l'username in una apposita macchina del punto di Controllo e digita la propria password.
2. Il Sistema effettua il riconoscimento dell'identità, e verifica la correttezza dei dati immessi. Se i dati sono corretti, effettua l'accesso al Sistema.
3. L'Addetto al controllo sceglie la funzionalità "**WasteList**". Il Sistema verifica che l'addetto possa eseguire il controllo.
4. L'Addetto al controllo preleva un rifiuto; identifica con l'apposito lettore l'etichetta e verifica il corretto smaltimento. Il Sistema registra l'avvenuto smaltimento del rifiuto.

Il passo 4 si ripete finché serve.

WasteList (C#)

La funzionalità WasteList dell'admin è diversa da quella vista nell'UC4 poiché prevede la possibilità di visualizzare tutti i rifiuti inseriti da tutti gli utenti in modo da controllarli e nel caso procedere con una eventuale segnalazione di essi.

1. Viene invocato il metodo '*Index'* all'interno di *'SegnalazioneController.cs'* che ritorna la View '*Index.cshtml'* collegata al Models '*Rifiuto*' di tipo IEnumerable.
2. Viene effettuata una HttpGet all'endpoint `http://localhost:8000/rifiuto/` in cui è in ascolto il Python Server che restituisce un elenco di tutti i rifiuti inseriti da tutti gli utenti.
3. Viene deserializzata la json string alla lista di tipo Rifiuto tramite il metodo '*JsonConvert.DeserializeObject'*.
4. Viene restituita la View che prende come parametro il contenuto deserializzato.
5. La View permette di visualizzare una tabella con tutti i rifiuti inseriti da tutti gli utenti del sistema DRS e contiene un collegamento alla Control Section. In questo modo, l'admin recupera il codice del rifiuto e poi procede con la funzionalità '*Signal*'.

WasteList (Python)

Si veda il paragrafo Inserimento Rifiuto (Python).

UC6: Inserimento Segnalazione

1. L'Addetto inserisce l'username in una apposita macchina del punto di Controllo e digita la propria password.
2. Il Sistema effettua il riconoscimento dell'identità, e verifica la correttezza dei dati immessi. Se i dati sono corretti, effettua l'accesso al Sistema.
3. L'Addetto al controllo sceglie l'attività "**Signal**". Il Sistema verifica che l'addetto possa eseguire l'azione.
4. L'Addetto al controllo deve definire i campi della segnalazione a cui fa riferimento l'utente segnalato: Titolo segnalazione, descrizione e importo della sanzione.
5. L'addetto provvede ad associare la segnalazione al rifiuto e al punto di controllo.
6. L'addetto conferma la segnalazione ed il sistema la registra.

Inserimento Segnalazione(C#)

1. Viene invocato il metodo '*Signal'* all'interno di *'SegnalazioneController.cs'* che ritorna la View '*Signal.cshtml'* collegata al Models '*Segnalazione'*. Questo consente di visualizzare la Control Section dove è possibile inserire la segnalazione.
2. L'admin seleziona il punto di controllo dalla tabella 'Control Point', inserisce il titolo, la descrizione, l'importo della segnalazione e il codice del rifiuto da segnalare.
3. Al click del bottone *'Signal'* viene invocato il metodo *public async Task<ActionResult> Signal (Segnalazione segnalazione)* che consente l'inserimento di una segnalazione dopo aver recuperato lo username dell'admin conservato nella variabile di sessione. Viene fatta una HttpPost all'endpoint `http://localhost:8000/segnalazione/` in cui è in ascolto il Python Server.
4. Viene fatto un controllo su eventuali campi vuoti poiché tutti sono obbligatori. Nel caso di campi vuoti, verrà invocato il metodo '*Error*' che ritorna la View '*Error.cshtml'* dove l'utente verrà indirizzato e visualizzerà il messaggio di errore: "An error has occured, please try again".
5. Se tutto va a buon fine, l'utente rimane nella pagina Signal dove potrà proseguire se vuole con la segnalazione di un nuovo rifiuto.

Inserimento Segnalazione (Python)

Il Django Server è in ascolto sulla porta 8080.

1. E' stato creato il modello *Segnalazione* dentro il file '*models.py'* della nostra applicazione '*app'*. Il modello Segnalazione mantiene mediante foreign key una relazione con 3 modelli: Rifiuto, Punto Controllo, Addetto.
2. E' stato eseguito il comando '*python manage.py makemigrations app*' per generare la migrazione per tali modifiche e poi '*python manage.py migrate*' per applicare le modifiche al db.
3. Per serializzare e deserializzare le istanze in formato JSON è stata creata una classe *'SegnalazioneSerializers'* dentro il file '*serializers.py'* in esso importiamo il modello creato e definiamo i campi.
4. E' stata creata la classe *'SegnalazioneViewSet'* dove viene usato il modulo *viewsets* fornito da rest_framework utile per la creazione delle URL.
5. Viene registrata la vista al modulo routers fornito da rest_framework.

UC7: L'admin deve poter visualizzare le segnalazioni effettuate.

SignalList (C#)

6. Viene invocato il metodo '*SignalList'* all'interno di *'SegnalazioneController.cs'* che ritorna la View '*SignalList.cshtml'* collegata al Models '*Segnalazione*' di tipo IEnumerable.
7. Alla chiamata del metodo viene innanzitutto recuperato lo username dell'admin dalla variabile di sessione.
8. Viene effettuata una HttpGet all'endpoint `http://localhost:8000/signal/{id_addetto}` in cui è in ascolto il Python Server.
9. Viene deserializzata la json string alla lista di tipo Segnalazione tramite il metodo '*JsonConvert.DeserializeObject'*.
10. Viene restituita la View che prende come parametro il contenuto deserializzato.

SignalList (Python)

Il Django Server è in ascolto sulla porta 8080.

1. E' stata creata la classe *'SegnalazioniList1'* dentro il file *views.py* dove viene usato il modulo *generics* fornito da rest_framework che restituisce l'intero set di query per un gestore di modelli.
2. Attraverso l'override del metodo *get_queryset()* è possibile filtrare il set di query di qualsiasi view che include sottoclassi *GenericAPIView*.
3. E' stato effettuato un filtraggio in base all'URL, creando una vista che restituisce un set di query filtrato dallo username dell'URL.
4. Viene registrata la vista all'interno degli *urlpatterns*:

```
url('^signal/(?P<id_addetto>.+)/$', SegnalazioniList1.as_view()).
```

UC8: L'utente deve poter visualizzare le segnalazioni.

SignalAll (C#)

La funzionalità SignalAll dell'utente prevede la possibilità di visualizzare tutte le segnalazioni eseguite dagli admin e conoscere eventuali suoi rifiuti segnalati mediante il codice rifiuto.

1. Viene invocato il metodo '*SignalAll'* all'interno di *'RifiutoController.cs'* che ritorna la View '*SignalAll.cshtml'* collegata al Models '*Segnalazione*' di tipo IEnumerable.
2. Viene effettuata una HttpGet all'endpoint `http://localhost:8000/segnalazione/` in cui è in ascolto il Python Server che restituisce un elenco di tutte le segnalazioni.
3. Viene deserializzata la json string alla lista di tipo Segnalazione tramite il metodo '*JsonConvert.DeserializeObject'*.
4. Viene restituita la View che prende come parametro il contenuto deserializzato.
5. La View permette di visualizzare una tabella con tutte le segnalazioni inserite da tutti gli admin del sistema DRS.

SignalAll (Python)

Si veda il paragrafo Inserimento Segnalazione(Python).

Data R

Sono state eseguite delle statistiche a partire dai dati presenti sul db. In particolare, viene visualizzato:

1. Un andamento sulla tipologia del rifiuto maggiormente smaltito tra carta, organico, plastica, vetro ed indifferenziata.
2. Un andamento sull'importo delle segnalazioni effettuate dagli admin.

Connessione al db SQLite

R può connettersi a quasi tutti i tipi di database esistenti. I tipi di database più comuni hanno pacchetti R che consentono di connettersi ad essi.

RSQLite e DBI sono stati quelli usati per la connessione al db SQLite chiamato *'db.sqlite3'*.

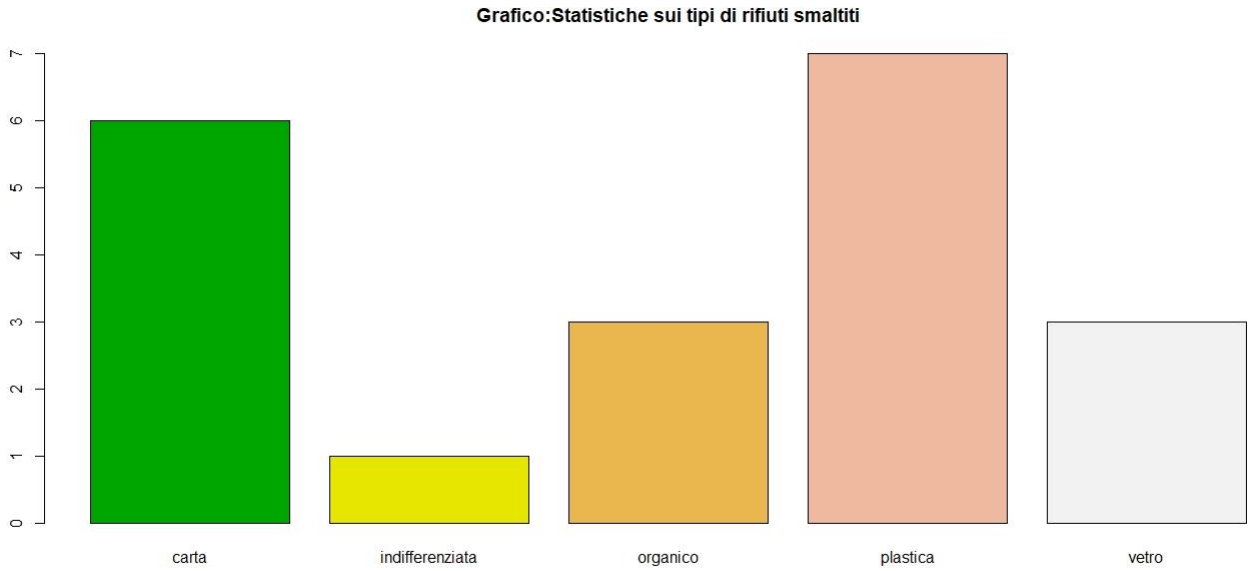
Steps:

1. Download dei package **RSQLite** e **DBI**
2. Connessione al database impiegando il seguente comando che prende come parametro il path della directory in cui si trova il db:
   db=dbConnect(SQLite(),dbname="C:/Users/gagli/sqlite/db.sqlite3")
3. Verifica del successo della connessione con il seguente comando che consente di elencare le tabelle del db:
   dbListTables(db)

Caso 1 (StatisticheTipoRifiuti.R)

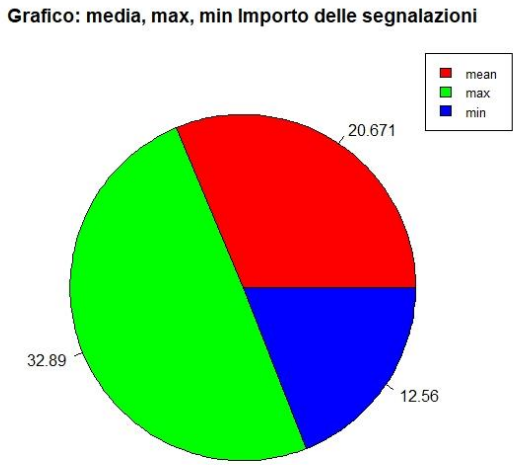
1. Dopo aver recuperato i tipi di rifiuto dalla tabella *'app_rifiuto'*, essi vengono salvati nella variabile '*TipiRifiuti*'.
2. Viene usata la funzione **str**(TipiRifiuti) per verificare il contenuto e la struttura del **data frame**.
3. Le colonne del data frame appartengono alla classe **character** per cui possono assumere un numero limitato di valori detti **livelli**. Per lavorare con questi dati, si utilizza la classe **Factor.** Viene dunque eseguita la conversione, utilizzando la funzione **factor()**.
4. Verifica del successo della conversione tramite l'utilizzo della funzione **summary()** che produce una tabella con i conteggi per ogni livello di fattore.
5. Plot del numero di rifiuti per ogni tipologia.

In figura, vengono mostrati i risultati ottenuti:



Caso 2 (StatisticheImporti_Max_Min_Mean.R)

1. Vengono recuperati gli importi delle segnalazioni eseguite dagli admin dalla tabella *'app_segnalazione'* e salvati nella variabile *'importi'*.
2. Calcolo della media, del massimo e del minimo degli importi utilizzando rispettivamente le funzioni: **mean(),max(),min().**
3. Creazione del data frame che prende come parametri i valori calcolati al punto 2.
4. Pie dei valori ottenuti.



Gestione dello stato della sessione in ASP.NET Core

I dati della sessione sono supportati da una cache. Per abilitare il middleware della sessione, Startup deve contenere:

- Qualsiasi cache **IDistributedCache** di memoria. L'implementazione IDistributedCache viene usata come archivio di backup per la sessione.
- Una chiamata a **AddSession()** in ConfigureServices .
- Una chiamata a **UseSession()** in Configure.

Lo stato della sessione è accessibile con **HttpContext.Session** che è disponibile dopo che è stato configurato lo stato della sessione.
HttpContext.Session è un implementazione di **ISession.** L'implementazione **ISession** fornisce i vari metodi per settare e recuperare valori interi e stringhe come **GetString() e SetString().**
I dati della sessione vengono eliminati quando viene chiamata l'implementazione **ISession.Clear(),** funzionalità implementata all'interno del metodo **Logout()** di *'HomeController.cs'*.

Il metodo Main in ASP.NET Core

Il metodo Main della classe Program è l'entry point della ASP.NET Core application. Crea il Web Host all'avvio:

1. **CreateWebHostBuilder** crea l'host e gli fornisce la configurazione.
2. Il **buildmethod** lo costruisce utilizzando la configurazione fornita
3. Il **Runmethod** lo esegue e quindi ascolta le richieste HTTP.

Glossario

● **Segnalazione:** provvedimento di tipo economico nei confronti degli utenti che non hanno rispettato le regole di raccolta differenziata.

● **Addetto al punto di controllo:** utente che si occupa non solo di controllare e verificare il rifiuto ma anche di gestire le segnalazioni.

● **Punto di raccolta:** centro per la raccolta e lo smistamento dei rifiuti.

● **Punto di controllo**: centro per il controllo e la verifica dei rifiuti.