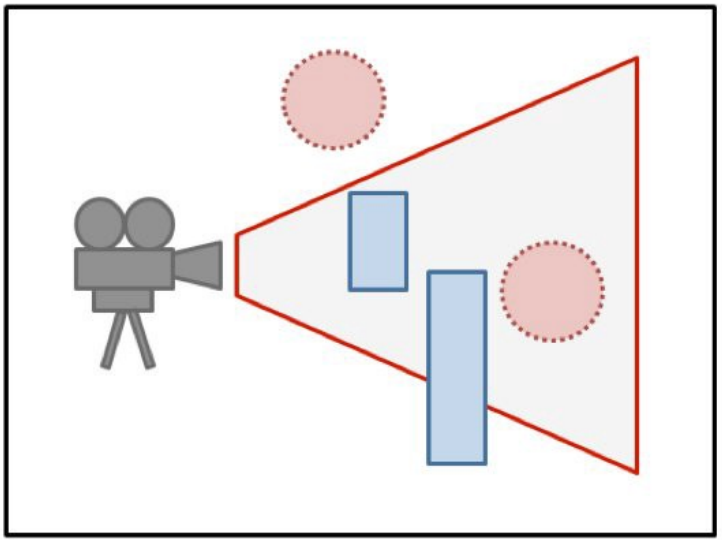
EU TRUST SERVICE LOCAL

CODE

Come il game development ha cambiato il nostro modo di vedere i dati

Nello sviluppo videoludico sussiste una tecnica di ottimizzazione chiamata **occlusion culling**. Essendo un videogioco un insieme di molti elementi complessi nello stesso ambiente, i dati che dovrebbe elaborare un computer sono molti e dispendiosi.



In questo caso la filosofia per alleggerire il peso computazionale risulta *"elabora solo quello che puoi vedere"* (nell'immagine sopra, i due cerchi non vengono "elaborati" perché non visibili). Seppur nel nostro caso la mole di dati sia esponenzialmente meno complessa, seguire una filosofia di "risparmio" ci ha permesso, in un certo senso, di agire con premura in situazioni e legami illogici che ci avrebbero fatto consumare un quantitativo notevole di memoria.

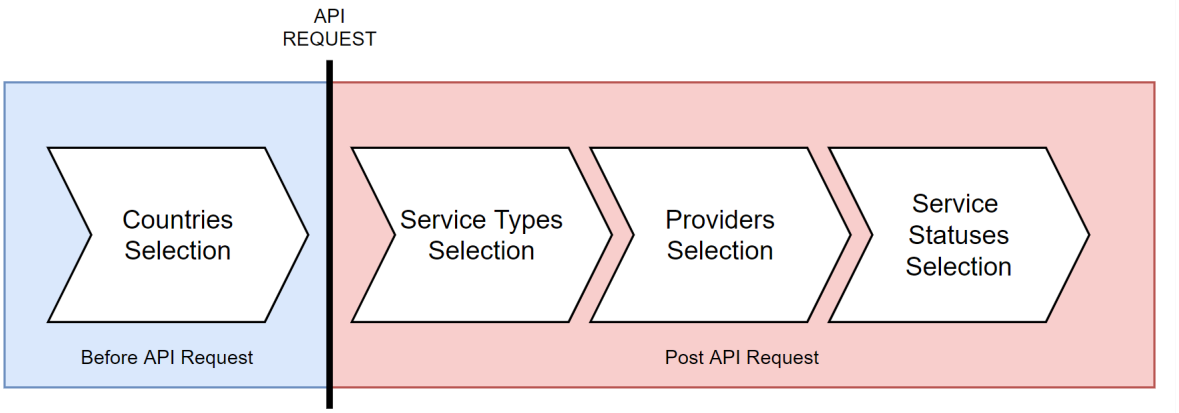
Collegare Provider e Country

Inizialmente il collegamento tra provider e country risulta logico, *"una country contiene più provider"*. Anche se sul piano razionale non sussiste alcun errore, i requisiti di sistema ci permettono di spezzare le catene di questa relazione, semplificando e permettendo al garbage collector di agire più liberamente. Essendo **ogni provider provvisto di countryCode della country a cui appartiene** abbiamo deciso di creare l'illusione di questo legame attraverso la rappresentazione visiva delle bandiere, in questo modo sembrerà che ci sia una relazione che in realtà non è strettamente collegata al codice.

Dati a Blocchi

Basando la nostra architettura sull' *"elaborare solo il necessario"* non può certamente esistere un approccio di caching del server (scaricare tutto il contenuto del server durante l'avvio del sistema). Nonostante anche questo approccio abbia i suoi difetti, secondo i nostri pareri risulta il più performante nella maggior parte delle situazioni, **ma come funziona questo sistema?** Innanzitutto, all'avvio del programma **vengono salvate in memoria solo le country, con i rispettivi dati,** e vengono mantenute per tutto il periodo di esecuzione del programma, questo perché, in parte sembra difficile che vengano aggiornate le countries durante l'utilizzo del software e in parte la mole di dati salvata è abbastanza esigua da permetterci di salvarli senza riserve. La ricerca invece, risulta un pochino più elaborata. Se in primo luogo la selezione delle country viene fatta con i dati in locale, per rispettare il requisito

minimo *“le selezioni successive devono presentare nella lista delle selezioni solo quelle effettivamente disponibili”* dobbiamo già effettuare una richiesta al server per avere i service types effettivamente disponibili. Per gli altri due parametri di ricerca, non è disponibile una richiesta all’API, quindi dovremmo comunque agire in locale. Nel pratico quindi la nostra ricerca è formata nel seguente modo:



Il vantaggio di questo approccio risulta il seguente: nonostante il tempo per scaricare i dati richiesti, nel filtro locale, anche se concettualmente della stessa complessità temporale di un software che esegue il caching del server, avrà a che fare, nella maggior parte delle volte, con una mole di dati minore, aumentando le performance. Per rispondere alla critica “così però non puoi utilizzare il tuo applicativo offline”, noi ci limitiamo a dire che, essendo la versione desktop di un servizio online, intrinsecamente la connessione a internet è un prerequisito, mentre non lo è occupare eccessiva memoria o velocità di calcolo.

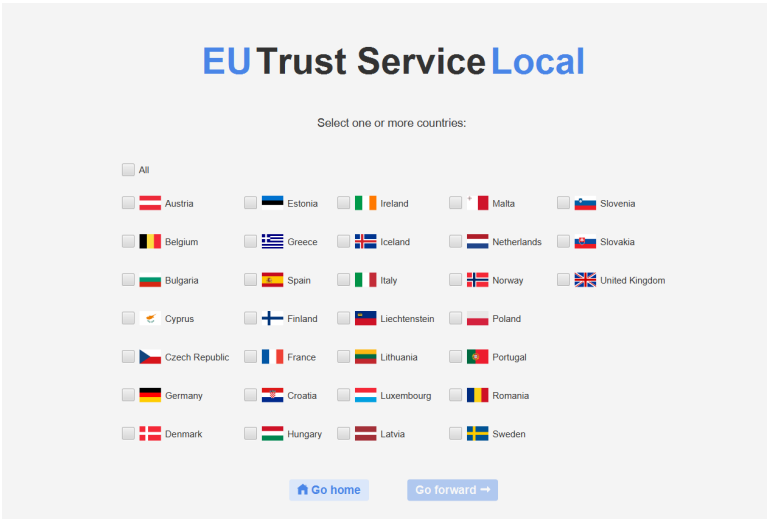
Ottimizzazione della Query

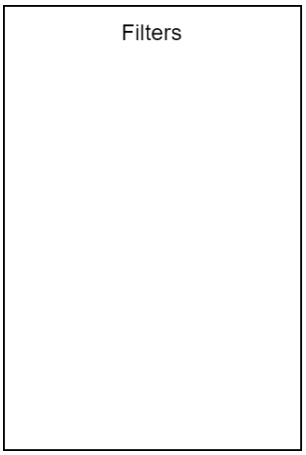
Comunicare con l’API e richiedere i dati ad ogni ricerca è ovviamente la soluzione più semplice; tuttavia se voglio eseguire una ricerca simile ad un’altra eseguita in precedenza, parte dei dati da elaborare sono già presenti in locale, quindi scaricarli di nuovo sarebbe inutile (inutile dire che rallenterebbe anche l’intera applicazione).
Per permetterci di avere sempre dati aggiornati, tuttavia, questa ottimizzazione vale solo tra ricerche effettuate durante lo stesso runtime dell’applicazione: se salvassimo i dati in locale man mano che vengono effettuate ricerche, arriveremo ad un punto in cui tutti i dati sono salvati sul dispositivo, non facendo più chiamate all’API, rischiando di perdere eventuali aggiornamenti dei dati.
Siccome l’API request viene effettuata conoscendo solo i paesi voluti, per ottimizzare le chiamate ci basterà tenere traccia dei paesi selezionati.
Tenendo conto dei paesi su cui è già stata fatta una chiamata all’API, è facile capire quali sono i dati che abbiamo già scaricato e quali sono ancora da scaricare; inoltre se si vuole eseguire una ricerca con una parte di dati già scaricata e una parte ancora da scaricare, la Query richiede all’API solo la parte ancora da scaricare, per effettuare richieste meno pesanti.
Ovviamente, questo significa che, andando avanti con le ricerche, aumenta la mole di dati salvati in locale: abbiamo notato tuttavia, che lo spazio occupato dai dati scaricati in locale non è troppo, grazie anche alla selezione di dati effettivamente importanti effettuata nel Creation Subsystem.

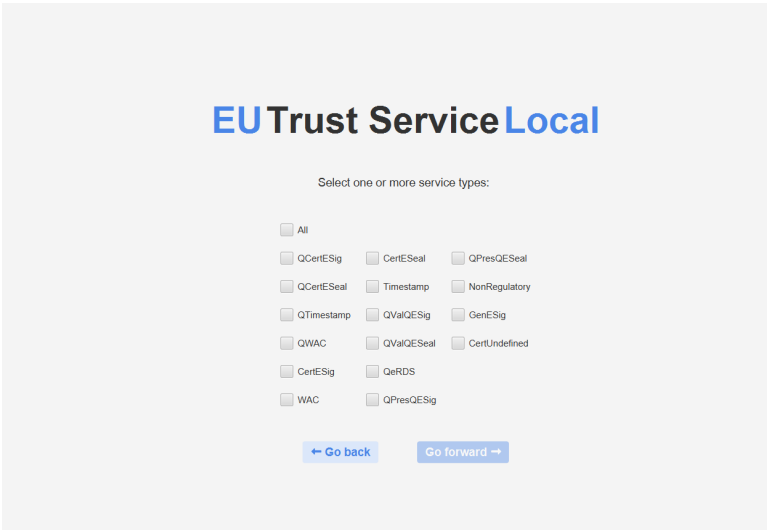
In media quindi, siccome vengono richiesti ad ogni ricerca solo i dati non ancora in nostro possesso, il tempo che ci impieghiamo ad eseguire una ricerca è inferiore al tempo necessario per scaricare tutti i dati in locale una volta sola.

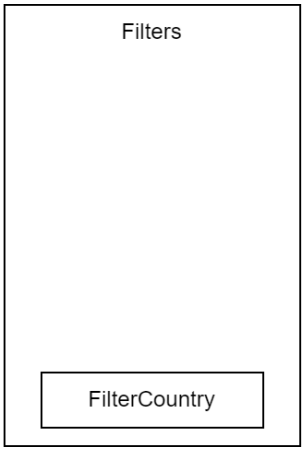
Oggetti Filter e la vicinanza al Filter Pattern

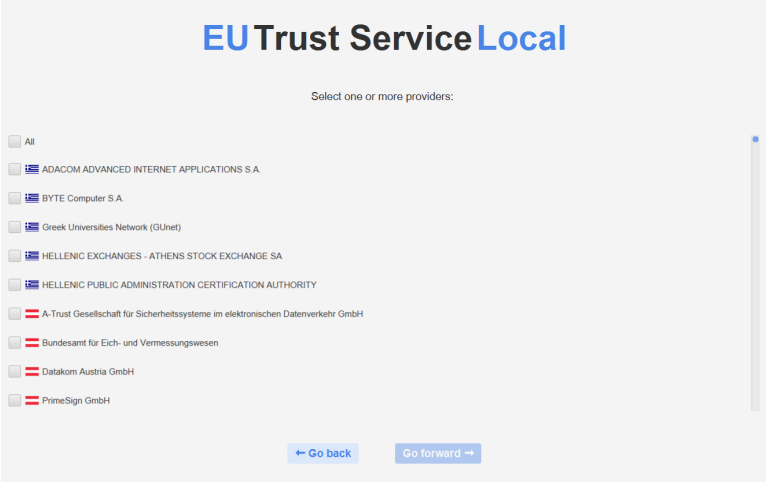
Seppur non in maniera rigorosa per costruire la Query ci siamo ispirati al **Filter Pattern**. Una delle ispirazioni più evidenti è la definizione di "oggetti" chiamati filtri che effettuano le operazioni sulla lista. Sembra banale ma questo approccio permette una buona scalabilità. Nel contesto attuale avviene la seguente situazione (a sinistra vediamo le schermate, a destra l'array Filters dentro Query prima di cliccare "go forward"):

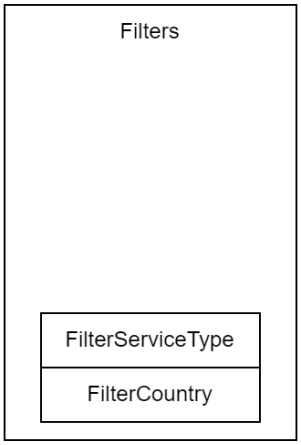


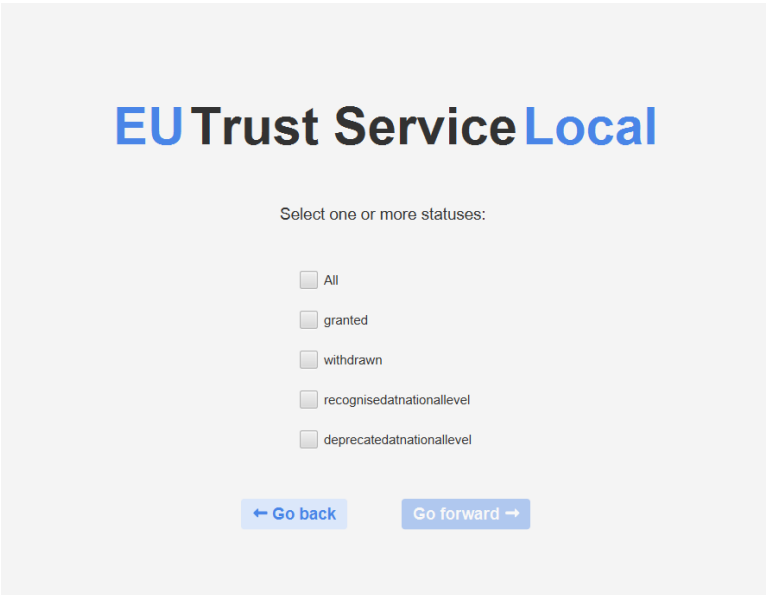


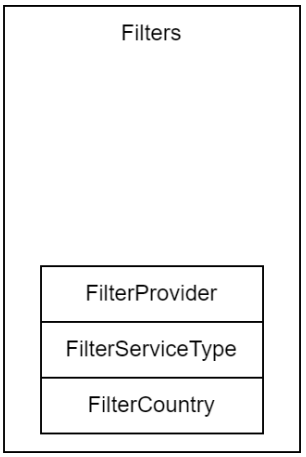


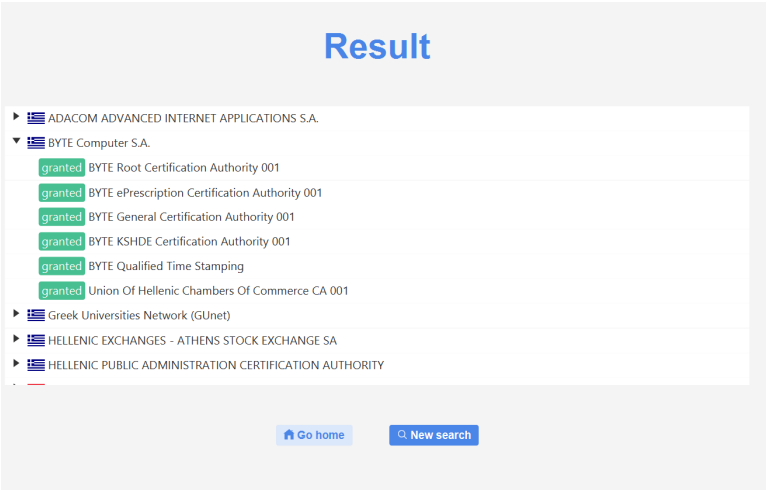


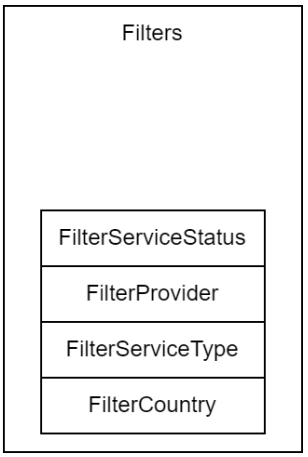












Quindi, dopo ogni selezione, viene invocato l'oggetto addetto a quel tipo di filtro e viene modificato. Non appena sono necessari i dati filtrati, la query passa al setaccio i dati in suo possesso, tramite i filtri, e restituisce il risultato.

Con questa architettura, innanzitutto i filtri sono intercambiabili a piacimento, senza variare i risultati, successivamente possiamo creare altri oggetti che estendono Filter, in modo tale da filtrare per altri parametri a piacimento.

Impegno nel rispetto dei primo principio S.O.L.I.D

Per scrivere buon codice occorre seguire delle filosofie o linee guida che ci aiutino a sviluppare il nostro applicativo, noi abbiamo improntato l'architettura e successivamente l'implementazione sui principi SOLID (a causa della dimensione ridotta del progetto siamo riusciti ad applicare solo il primo principio nel pratico):

- **S**ingle Responsibility Principle
- **O**pen/Close Principle
- **L**iskov's Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency inversion principle

Single Responsibility Principle

Questo principio è molto semplice ed intuitivo, come in una catena di montaggio ogni dipendente ha un'unica mansione, nel programma, ogni classe deve adempiere ad un solo scopo. Per esempio, un programmatore crea una classe A che :

- Si connetta all'api
- Converte i dati in oggetti java

Per rispetto a questo principio sopra descritto questa classe A dovrebbe essere scomposta in due. Un esempio di valenza di questo principio nel nostro progetto è proprio il **Creation Subsystem**, che contiene una classe di comunicazione con il server e una di creazione di oggetti java dai dati ottenuti.

Da una web application abbiamo preso l'MVC come pattern Architetturale

Il pattern architetturale MVC è uno dei più usati quando si parla di web application, MVC sta per:

- **Model:** fornisce i dati dell'applicazione
- **View:** visualizza graficamente il contenuto del Model
- **Controller:** manipola model e view attraverso comandi utente

Nel nostro caso, ogni punto rappresenta un nostro sottosistema, dove il **creation subsystem** svolge la funzione di **Model**, il **Query Subsystem** e il **Controller Subsystem** quella di **Controller**, e infine, ovviamente, il View Subsystem come **View**.

La questione API, tra standard non rispettati e feature mancanti

Quando si ha un'idea chiara di cosa si vuole fare nel proprio sistema è sempre complicato interfacciarsi o subire i sistemi esterni e anche in questo caso abbiamo avuto complicazioni riguardo le APIs.

ISO 3166

L'ISO 3166 è lo standard internazionale per i country codes, sussiste in una tabella di tutti gli stati del mondo e del loro codice a 2 o 3 cifre. Nell'API delle flag l'ISO sembra rispettato, al contrario nell'API europea ci sono due variazioni:

- L'inghilterra, che secondo l'ISO il country code risulta **GB** nell'api è **UK**
- La grecia, che secondo l'ISO il country code risulta **GR** nell'api è **EL**

Mancanza della request: search/service_types

Per noi sarebbe stato comodo avere una richiesta all'interno dell'API che restituisse solo i service types, però, per un motivo a noi ignoto, questa feature manca completamente. L'unico modo per ottenerla è ricavarla da "*search/tsp_list"* filtrando tutti i provider di tutte le nazioni. Risulta evidente che questo tipo di metodo è dispendioso e inutile, quindi con dispiacere siamo stati obbligati a salvarci staticamente i types. In questo caso abbiamo dovuto decidere se andare in contrasto con la nostra filosofia "carica solo il necessario" oppure incappare in istruzioni inutili e pesanti, ovviamente bisogna sì mantenere la coerenza con la propria filosofia ma essere flessibili nei momenti opportuni.

Valorizzare l'esperienza utente

La nostra grafica risulta, secondo la nostra opinione, piacevole ed intuitiva, adatta a tutti, rispettando alcuni criteri e caratteristiche tipiche del material design, come la reattività delle interazioni, la coerenza dei materiali, le regole di alto contrasto e l'utilizzo di icone.