EU TRUST SERVICE LOCAL

SYSTEM DESIGN

PREMESSA

Lo scopo di questo file è fornire la documentazione progettuale utilizzata per realizzare l'applicativo "*EU Trust Local*", assegnatoci dal docente **Boldrin Luca**, per la verifica dell'apprendimento del corso di **Elementi di Ingegneria del Software 2021/22**.

INTRODUZIONE

Ci viene richiesto di creare un applicativo locale che funzioni in maniera analoga al seguente sito internet, ricavando i dati dalla opportuna API pubblica, dalle seguenti funzionalità minime :

- Presentare la lista dei paesi EU
- Presentare la lista dei "Tipi di Trust Service"
- Selezionare / Deselezionare uno o più paesi (da una lista)
- Selezionare / Deselezionare uno o più provider di servizi (da una lista)
- Selezionare / Deselezionare uno o più tipi di servizio (da una lista)
- Selezionare / Deselezionare uno o più stati del servizio (da una lista)
- Una volta effettuata una selezione, le selezioni successive devono presentare nella lista delle selezioni solo quelle effettivamente disponibili.
- Fare una query basata sugli oggetti correntemente selezionati – la query ritorna una lista di servizi

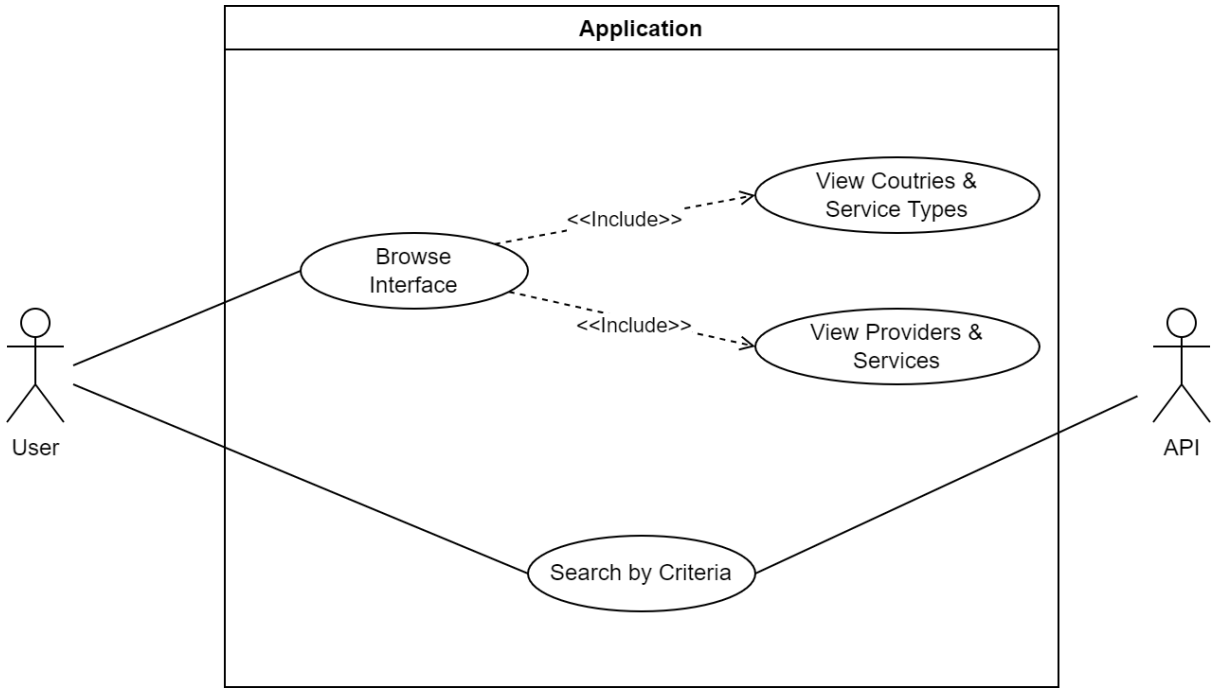
Inoltre ci viene vivamente consigliato di utilizzare una libreria grafica di Java (JavaFX) per fornire un'interfaccia utente adatta al programma.

USE CASES

Con il termine use cases definiamo dei tools atti a **descrivere cosa l'utente può effettuare** con il programma. Gli use cases derivano in maniera diretta dai **requisiti minimi** di sistema. In questo caso abbiamo raggruppato gli otto requisiti minimi in due **use cases principali**:

- **Browse**, la capacità di un utente di esplorare l'interfaccia grafica e interagire con essa per visualizzare i dati che vuole ottenere.
- **Search**, la capacità di un utente di cercare esplicitamente dei dati, attraverso dei criteri e parametri di ricerca dinamici.

Inizialmente abbiamo costruito la **rappresentazione UML** degli use cases:



Application è il nome che abbiamo voluto associare al nostro sistema (rappresentato sempre da un rettangolo contenitivo). Gli **attori** sono due: User e API.
User lo definiamo come **attore primario** ed essendo colui che inizia ad interagire con il sistema è alla sinistra di esso.
API invece è un **attore secondario** ed, in quanto un attore che *"reagisce"* agli use case del sistema, si trova a destra nella rappresentazione.
Nelle fasi successive risulterà evidente la distinzione tra attore primario e secondario.
Le ellissi all'interno del sistema sono i veri e propri **use cases**, le linee presenti sul diagramma sono le **relazioni**. In questo diagramma abbiamo due tipi di relazioni:

- **Associative:**rappresentate con linee continue, indicano un'interazione standard con il sistema;
- **Inclusive:** rappresentate invece con linee tratteggiate indicano una dipendenza, in quanto lo use case include gli use cases che utilizza per funzionare correttamente. Nel nostro caso *Browse Interface* include view Countries e view Providers.

Per quanto il diagramma UML risulti intuitivo, risulta altrettanto generico e poco descrittivo. Per completare il nostro lavoro dunque, abbiamo utilizzato un template personalizzato per descrivere ogni use case, indicando solamente i parametri che donassero una maggiore comprensione degli stessi, così da ottenere un buon rapporto **informazioni/leggibilità**.

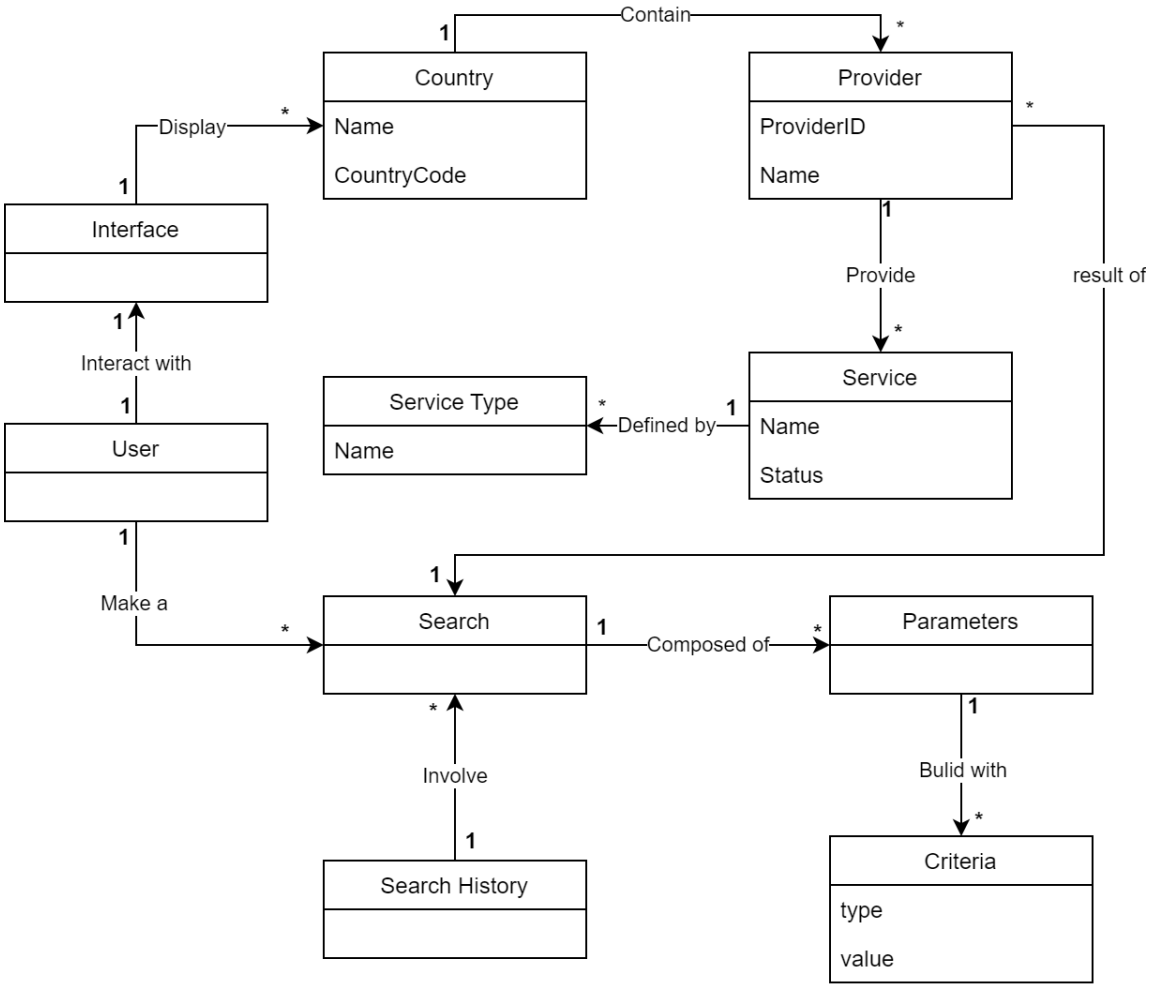
Title	Browse the Interface
Description	User's ability to explore all the information in our application
Actors	User
Actions	• user open program • user views countries, service types, trust services and services
Precondition	• Connection to Internet • System contains countries • System contains service types
Postcondition	• ordinated view of all data (countries, providers, services) • countries saved in the system
System Responses	• display countries with flags • display providers & services • **error message:** internet is not available

Title	Search by Criteria
Description	• User's ability to search specific information in our application by parameters • API's ability to provide specific informations
Actors	User, API
Actions	• user click search button to start a search • user chooses multiple criteria • when user completes the query, press "go forward"
Precondition	• Connection to Internet • System contains countries • System contains services
Postcondition	• user sees list of services with selected criteria • system saves query criteria
System Responses	• display list of services with specific criteria • **error message:** internet is not available

DOMAIN MODEL

Il domain model è definito come un **modello concettuale** delle varie classi di un sistema, classi che non rappresentano assolutamente oggetti del software. Questo modello potrebbe ingannare facendolo sembrare molto simile al class diagram (che visualizzeremo nel dettaglio in seguito), ma ricordiamo che questi oggetti rappresentati fanno parte del mondo reale. Questo modello, secondo **Craig Larman**, aiuta la comprensione di un modello di business di cui per esperienza non sappiamo nulla, definendo un *"dizionario"* di termini noti in relazione tra loro. Molti oggetti che verranno rappresentati nel domain model verranno anche ripresi nel class diagram, perché in questo modo abbiamo molta familiarità nelle informazioni che contiene e delle responsabilità che detiene.

nota: abbiamo usato le linee guida contenute nel libro di **Craig Larman** (Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development 2004), ci sono sembrate le migliori e le più complete.



note: Interface è da interpretare come l'oggetto reale che ti porta alla visualizzazione dei dati, quindi esso non ha nessuna relazione con l'oggetto software (interface come oggetto software potrebbe essere la struttura e la programmazione delle interfacce, mentre l'oggetto fisico potrebbe essere il monitor).

DESIGN MODEL

Il design model consiste nella *"rappresentazione in UML della realizzazione degli use cases"*. Da questa definizione si può intuire quanto sia più **vicino all'implementazione** rispetto a tutti i modelli precedentemente presi in analisi. Il vantaggio di questo strumento è sicuramente la capacità di avere tutte le componenti del nostro sistema a portata di mano e capire quali sono le loro interazioni. Questi punti possono aiutarci notevolmente quando parliamo di progettazione, per esempio nel nostro caso il vantaggio più grande è stato comprendere i **sottosistemi,** soprattutto per capire come migliorare le relazioni in modo che entrassero e uscissero da una sola classe del sottosistema.
Nel nostro progetto abbiamo provveduto alla creazione di quattro sottosistemi:

- Creation Subsystem
- Controller Subsystem
- Query Subsystem
- View Subsystem

In questa sezione andremo a vedere inizialmente tutti i sottosistemi separati, successivamente osservemo il diagramma completo. Prima di iniziare però dobbiamo porre attenzione inizialmente sugli oggetti ricavabili dal servizio API.

Data Structures

Per ottenere gli oggetti generici ricavabili dall'API abbiamo dovuto consultare il file json che ci viene restituito dopo una richiesta. Il formato Json è estremamente intuitivo per identificare oggetti, quindi non è stato difficile ricavarne delle classi Java.

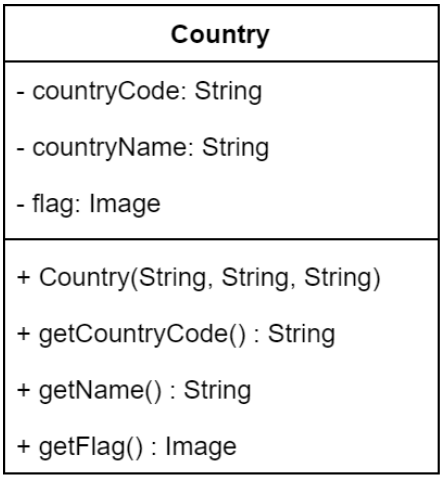
Country

Json representation:

```
{
   "countryCode" : "string",
   "countryName" : "string",
}
```

note:

- Flag è una immagine ottenuta da un link appartenente ad un API chiamato https://flagsapi.com/
- i valori una volta inseriti sono di sola lettura, per questo motivo non esistono setter



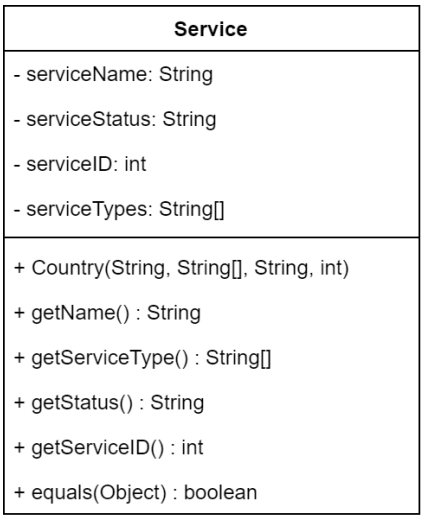
Service

Json representation:

```
{
    “countryCode” : “string”,
    “currentStatus” : “string”,
    “qServiceTypes” : [“string”],
    “serviceId” : 0,
    “serviceName”: “string”,
    “tob”: “string”,
    “tspId” : 0,
    “type” : “string”
}
```

note:

- i parametri del json non inseriti all’interno delle classi mancano perché non risultavano utili alla nostra applicazione
- i valori una volta inseriti sono di sola lettura, per questo motivo non esistono setter



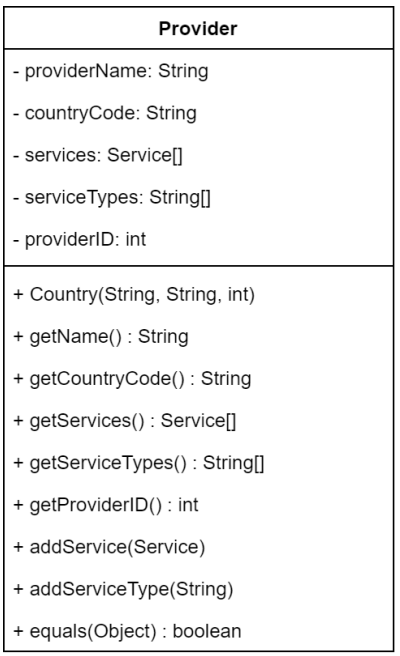
Provider

Json representation:

```
{
    “countryCode” : “string”,
    “name” : “string”,
    “services” : [ service ],
    “trustmark” : “string”,
    “tspId” : 0
}
```

note:

- le funzioni add sono state inserite perché in fase di creazione il package json ci faceva prima ottenere i dati del provider e successivamente dei servizi.

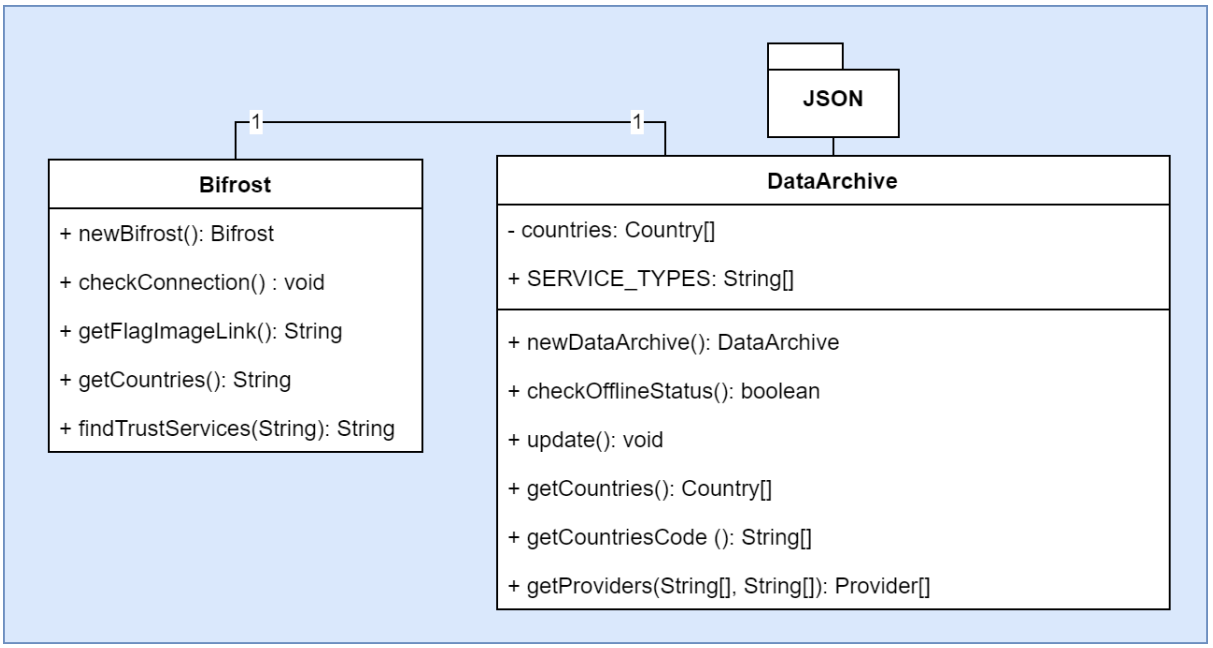


All Structures with Relations

Potrebbe sembrare che la mancata relazione di country con provider sia un errore ma in realtà è una scelta tecnica ragionata (argomenteremo questa scelta nel documento “codice”). Quello che occorre sapere è che non esiste nessuna connessione, dal punto di vista del codice, tra country e provider.

Creation Subsystem

Questo sottosistema si occupa di comunicare con il server e fornire a tutti gli altri sistemi gli oggetti java che rappresentano i dati ricevuti.



Questo sottosistema è composto da due oggetti, **Bifrost** e **DataArchive**, e di un package **JSON** (un pacchetto esterno per la manipolazione di file json).

Bifrost

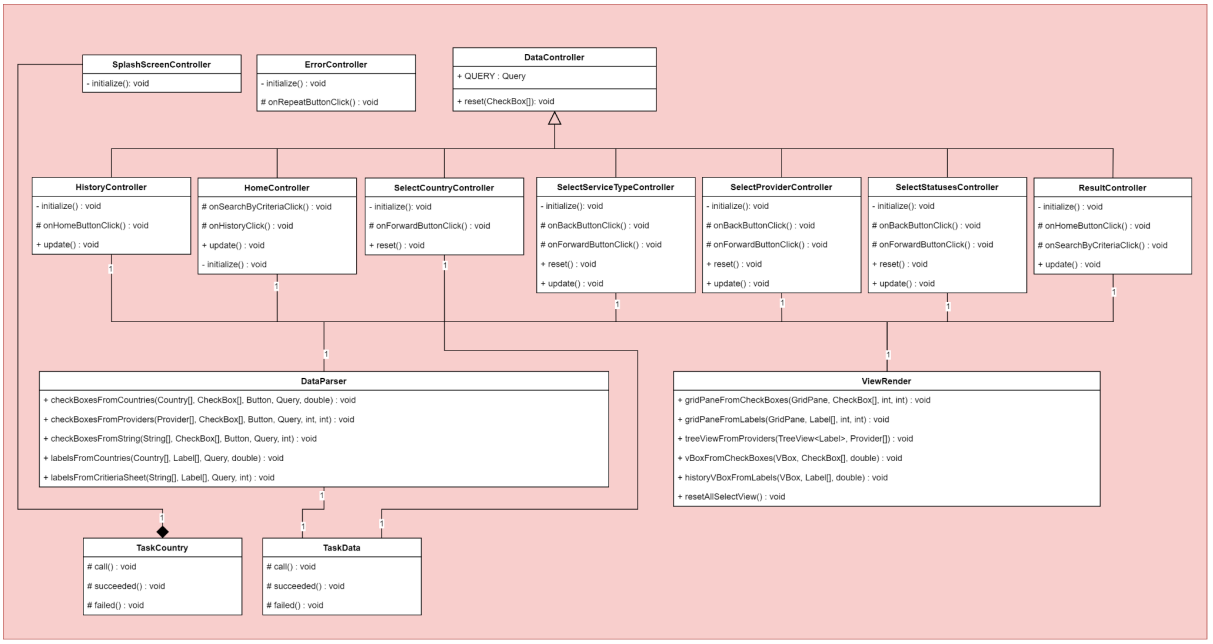
Il bifrost è il componente che si occupa del richiedere dati all'API e ricevere le relative risposte, il suo nome deriva dalla mitologia norrena *"il ponte dell'arcobaleno, che unisce la terra alla dimora degli dei"* *(Wikipedia)* una rappresentazione fantasiosa ma azzeccata del funzionamento di questo componente. Dalla prima funzione scritta nell'UML possiamo notare che il Bifrost sfrutta il design pattern **Singleton**, un famoso pattern della GoF, che permette una singola istanza di quell'oggetto in tutto il programma, ci sembrava azzeccata questa implementazione, in parte perché è logico che colui che richiede al server sia uno e unico, in secondo luogo perché i componenti interni (per esempio HttpClient che nel modello non vediamo) sfruttano essi stessi il pattern. **getCountries()** e **findTrustServices(String)** sono le rispettive HTTP request di nostro interesse elencate nella sezione api-search-controller della documentazione dell'API, la prima funzione si occupa di ricevere la lista di country da inserire all'interno della nostra applicazione, mentre la seconda fornisce una lista di provider e servizi in base al country e ai tipi di servizio. Interessante **getFlagImageURL(String CountryCode)**, una funzione che sfrutta una seconda API (non effettuando una vera e propria richiesta ad un server) che fornisce le immagini delle bandiere grazie al CountryCode, rispettando l'ISO 3166. **checkConnection()** controlla, attraverso un ping al server google, la connessione ad internet, esso è stato aggiunto per ricevere un feedback quasi istantaneo dello stato di connettività.

Data Archive

Il dataArchive come fa presagire il nome, è il componente che si occupa della gestione e mantenimento dei dati ottenuti dal server in oggetti.
Partendo dai suoi parametri, il primo che troviamo è **countries**, un array di Country privato, che viene popolato alla creazione del DataArchive o quando richiesto esplicitamente dal sistema (grazie al metodo update). Il secondo parametro è un array di Stringhe chiamato **SERVICE_TYPE**, questo array costante contiene tutti i tipi di servizi, abbiamo scelto di salvarli manualmente, in primo luogo perché il recupero era inutilmente complicato e dispendioso (approfondimenti riguardo l'argomento vengono trattati nel documento code), in secondo luogo perché non ci sembra che la lista subirà notevoli e rapidi cambiamenti nel tempo. Da **newDataArchive()** notiamo che anche questo oggetto sfrutta il design pattern **Singleton**. Il metodo **update()** serve per aggiornare la lista di country e la connessione al server, quando ne abbiamo bisogno; abbiamo aggiunto questa funzione per non dover fare la richiesta ogni volta che si tornasse alla pagina principale ma solo quando viene esplicitamente richiesto dal sistema o dall'utente. **getCountries()** restituisce una copia dell'array countries, mantenendo così l'incapsulamento. **getCountryCodes()** è leggermente diverso perché restituisce solo gli identificativi delle countries salvate (**countryCode**). L'ultima funzione è **getProviders(String[], String[])** che richiede due array di stringhe, il primo contiene i countryCode delle country su cui vogliamo effettuare la ricerca, il secondo contiene invece i **serviceType** che deve avere il provider, quando e se trova dei valori, restituisce un array di Provider.

Controller Subsystem

Questo sottosistema ha il compito di gestire la logica delle pagine dell' applicazione (inizializzazione, aggiornamenti, eventi sulle pagine). Pertanto, ad ogni View è associato uno specifico Controller, caricato in esecuzione da JavaFX.



Data Controller

La superclasse DataController contiene un' unica istanza di Query, **QUERY**, per poter eseguire il caching dei dati ogni qualvolta viene fatta una nuova richiesta alla REST API, mediante Bifrost. Dunque, questa classe viene estesa da tutti i Controller che necessitano dei dati contenuti in Query. Inoltre, è presente un metodo **reset(CheckBox[])** che permette di resettare le selezioni effettuate in una specifica View.

DataParser & ViewRender

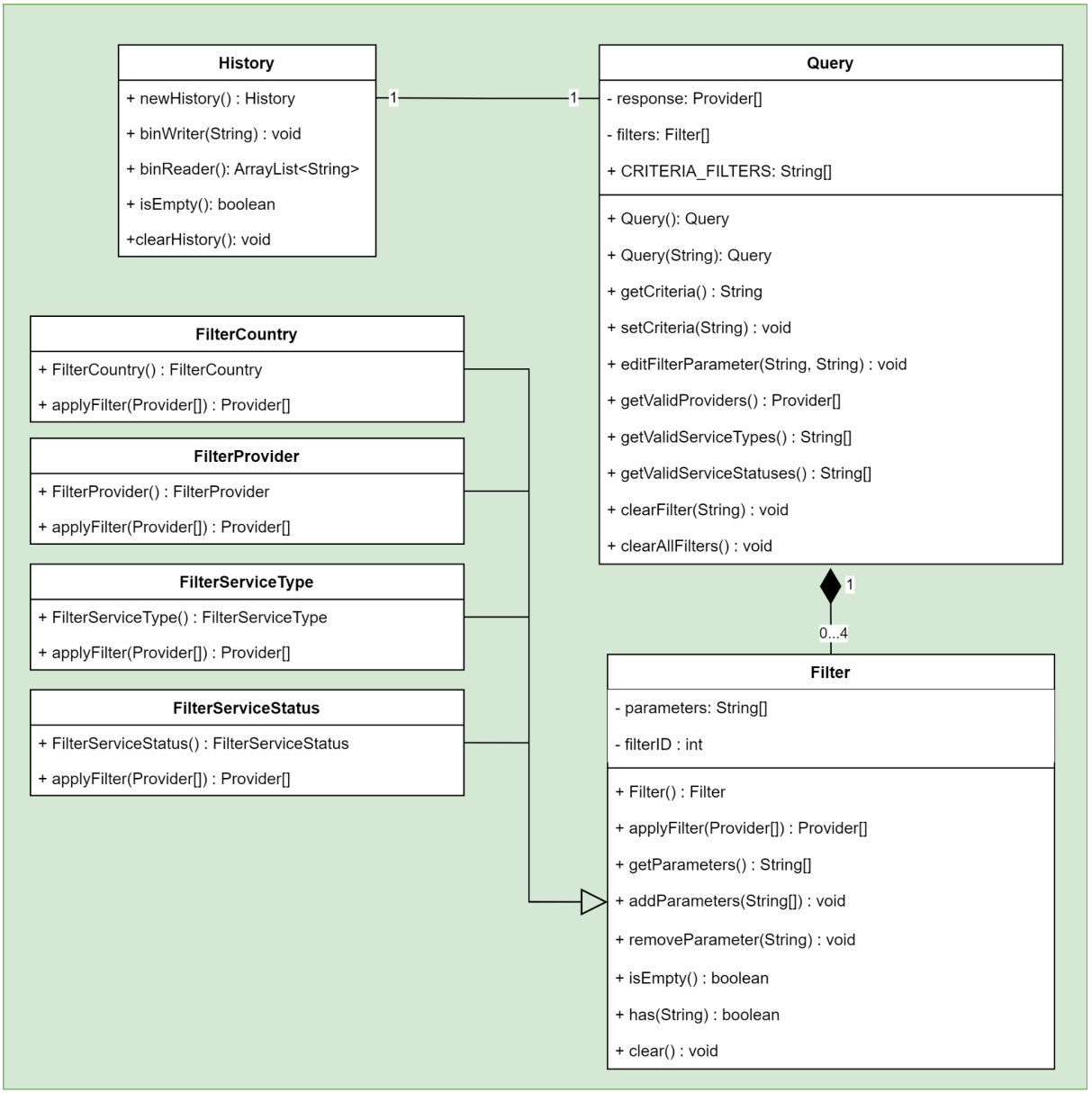
Le classi DataParser e ViewRender sono classi ausiliarie con l'unico scopo di rendere più leggibile il codice contenuto nei Controller che le utilizzano.
DataParser è un oggetto che si occupa di convertire i dati provenienti da Query o Data Archive in oggetti che possano essere caricati nelle varie View.
ViewRender, infatti, si occupa di collegare gli oggetti creati da DataParser alle varie View.

TaskCountry & TaskData

TaskCountry e TaskData sono dei task fondamentali per il funzionamento fluido e consistente dell' applicazione: hanno il compito di eseguire operazioni in maniera concorrente al main thread, evitando crash o freeze del programma.
TaskCountry si occupa di richiedere gli oggetti Country alla classe **Data Archive**. L'esecuzione di questo task è parallela al main thread e viene effettuata all'avvio, nella schermata di caricamento. È necessario che non interrompa il main thread, per una questione puramente estetica.
TaskData si occupa di richiedere gli oggetti contenuti in **Query**. L'esecuzione di TaskData interrompe il main thread per evitare che l'utente possa eseguire operazioni non consentite o

vedere informazioni non corrette. Questo task viene invocato ed eseguito nel **SelectCountryController** e nei metodi della classe **DataParser**.

Query Subsystem



Query

Per quanto riguarda la Query, si tratta della componente che fa da intermediario tra l'interfaccia grafica e il DataArchive, con lo scopo di salvare le richieste dell'utente nei relativi filtri, per poi effettuare in maniera ottimizzata una chiamata al server tramite il Bifrost o filtrare i dati già scaricati per ottenere una risposta adeguata.
Tra i parametri presenti, si ha il **CRITERIA_FILTERS**, un array di Stringhe pubblico utilizzato per indicare agli altri sistemi di che parametri hanno bisogno i metodi della Query, tra cui **editFilterParameter(String, String)** o **clearFilter(String)**.
Sono presenti due ArrayList di Provider per salvare la risposta del server ottenuta dal Bifrost, **response** e **filteredResponse**; uno contenente la risposta non modificata e il secondo

contenente la risposta filtrata; questa aggiunta si è resa necessaria per ottimizzare il numero di chiamate al server nel caso in cui i dati fossero già stati scaricati, ed avessero solo bisogno di un diverso filtraggio.
Sono presenti due costruttori della Query: uno standard che crea una Query vuota e uno che ha bisogno di una stringa chiamata **criteria**, in un formato concepito da noi, contenente i parametri dei vari filtri di una Query già esistente.
La stringa criteria può essere ottenuta dal metodo **getCriteria(),** che la costruisce inserendo i parametri dei filtri della Query.
Per aggiungere o rimuovere i parametri ai filtri della Query, serve usare il metodo **editFilterParameter(String, String)**, usando come parametri il nome del filtro da modificare (ottenuto da CRITERIA_FILTERS) e il parametro da aggiungere al filtro scelto.
Si hanno poi i metodi per accedere al contenuto filtrato della Query, **getValidProviders(), getValidServiceTypes()** e **getValidServiceStatuses(),** che restituiscono rispettivamente i providers, i service type e i service status ottenuti dalla ricerca effettuata.
Gli ultimi metodi da segnalare sono **clear()**, usato per pulire le response e altri archivi interni, **clearFilter(String)** e **clearAllFilters()**, utilizzati per pulire i filtri della Query.

Filter

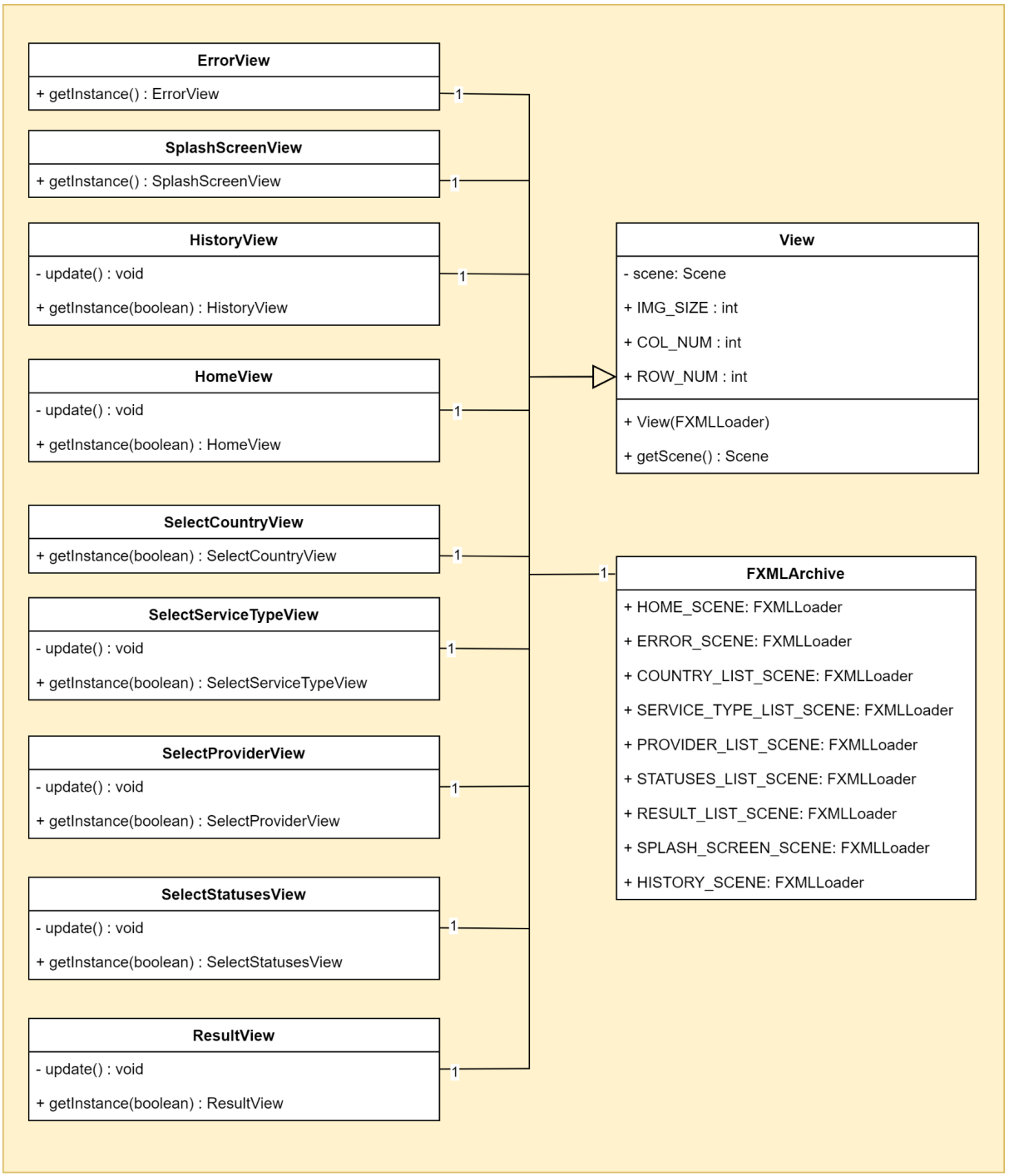
Questa è una superclasse, che verrà poi estesa nei quattro diversi tipi di filtri: **filterCountry, filterProvider, filterServiceType** e **filterServiceStatus.**
Dentro ogni filtro è presente un ArrayList di stringhe, **parameters**, contenente i parametri del filtro.
I metodi esistenti ed eventualmente sovrascritti dagli altri filtri, sono i metodi di scrittura e lettura dei parametri **addParameters(String[]), removeParameter(String)** e il metodo **applyFilter(ArrayList<Provider>)** che prende la lista di provider e ne restituisce una versione filtrata.

History

Abbiamo pensato infine di implementare una classe History, che rendesse disponibile all'utente una funzionalità aggiuntiva, ovvero la possibilità di consultare e ripetere le ultime 30 ricerche effettuate.
Il salvataggio di ogni ricerca avviene una volta che essa viene completata, per mezzo del metodo **binWriter(String)** passando come parametro esplicito una stringa di criteri della query, ottenuta dal metodo getCriteria() della classe Query.
Il metodo binWriter andrà, non a creare un semplice file di testo, bensì a creare(se non presente) un file .bin, popolandolo con le ricerche convertiti in codice binario.
Per la consultazione dei dati del file .bin, abbiamo implementato il metodo **binReader()** che si occupa della lettura del file, della conversione del codice binario in caratteri alfanumerici , ed infine di scrivere gli elementi in un array history, che andrà a restituire.
Se il numero di ricerche è maggiore di 30, il metodo restituirà solamente le 30 più recenti.
Questa scelta di utilizzare un file .bin per il salvataggio delle ricerche e non un semplice file .txt, nasce da una necessità di proteggerlo da eventuali modifiche volontarie da parte di terzi.
Nella classe sono presenti anche il metodo **isEmpty()**, che permette di verificare se il file History.bin è vuoto, ed il metodo **clearHistory()** che svuota il contenuto del file History.bin.

View Subsystem

Il sottosistema View ha il compito di caricare e salvare i file FXML delle varie schermate in oggetti Scene. Ogni sottoclasse di View, è un Singleton con lo scopo di contenere una sola scena per specifica schermata. Questo permette una sorta di caching dei dati e delle operazioni effettuate sulle View.



View

La superclasse View ha lo scopo di caricare mediante il costruttore **View(FXMLLoader)** le Scene, interpretando i file FXML che definiscono l'interfaccia grafica. In questa fase, alla schermata viene associato il relativo Controller.

FXMLArchive

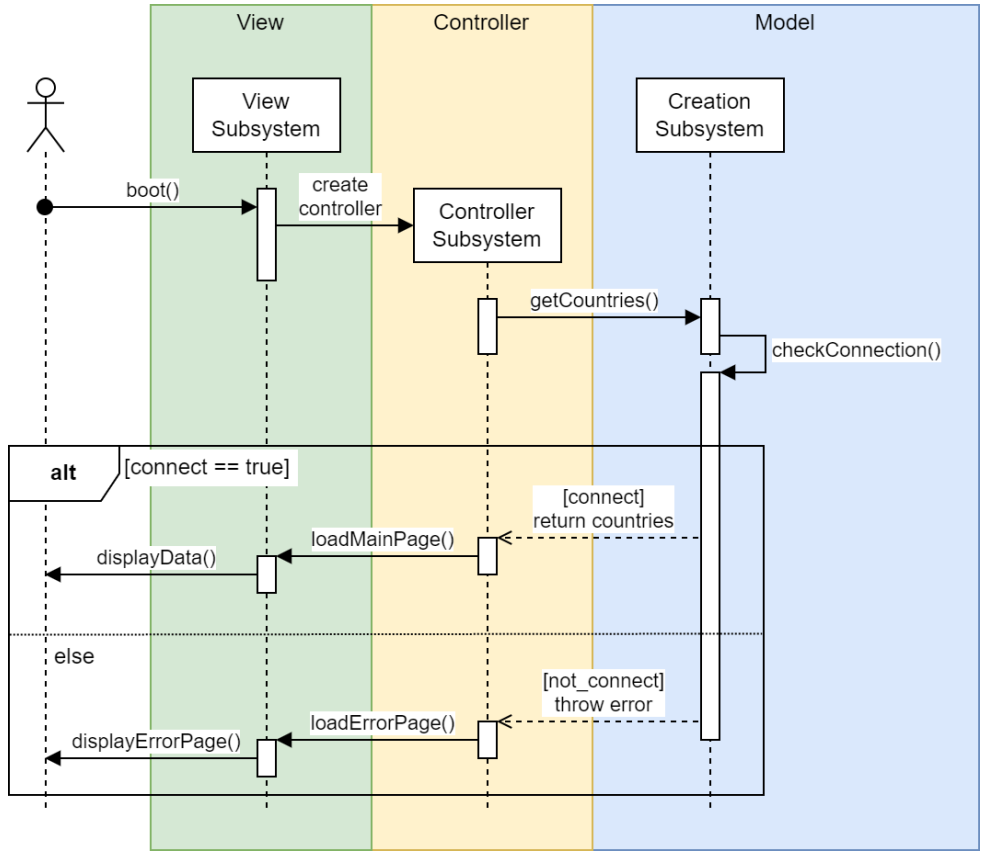
La classe FXMLArchive si occupa di raggruppare in maniera statica e convertire i file FXML in oggetti FXMLLoader, interpretabili dalle View.

SEQUENCE DIAGRAM

Il sequence diagram occorre per comprendere il percorso e le chiamate delle funzioni di ciascun sottosistema durante uno use case, questo tool viene rappresentato attraverso diagramma UML e sezione testuale descrittiva.

Boot Program

La fase di boot è quando l'utente avvia il programma ed esso inizia a partire.



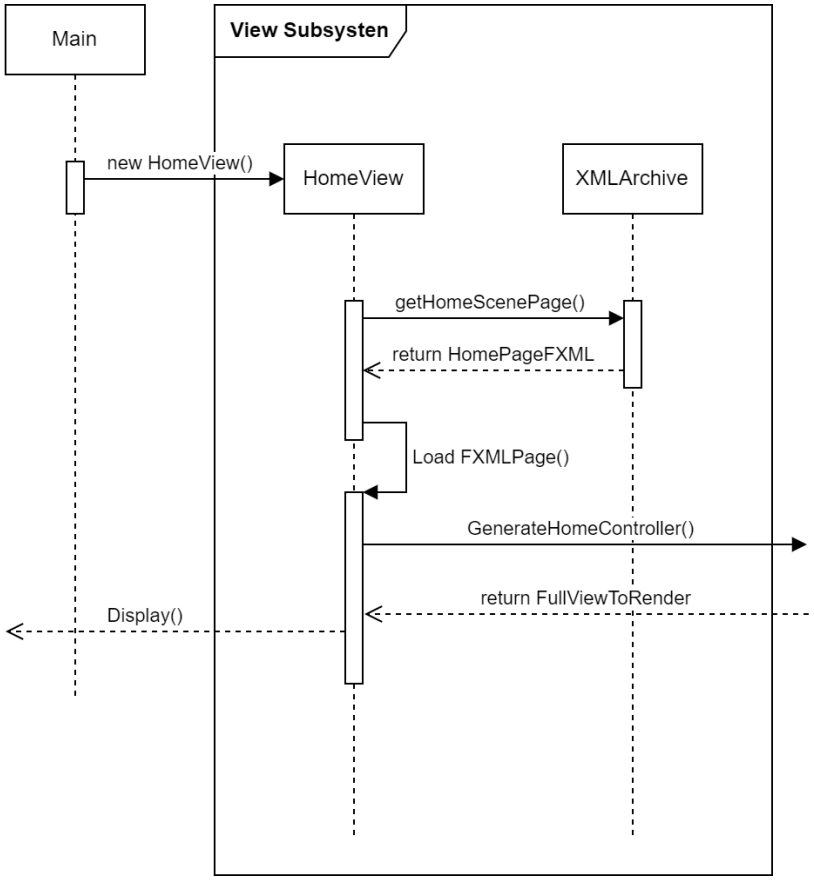
Quindi i passaggi sono:

1. L'utente lancia l'applicazione e si avvia il sottosistema delle schermate
2. il View Subsystem ricava il proprio controller creandone un'istanza presa da Controller Subsystem.
3. Il controller corretto (alla pagina Home, sarà corrisposto un HomeController) richiede di ricavare le countries dal Creation Subsystem.
4. Il Creation Subsystem controlla che sia disponibile la connessione ad internet prima di effettuare la richiesta.
5. **SE LA CONNESSIONE È PRESENTE :** Creation Subsystem Invia le countries come classi java al controller chiamante
    a. il controller compone i dati dinamici e li manda alla Pagina chiamante
    b. La pagina chiamante viene renderizzati con i valori dinamici trovati in precedenza.

6. **SE LA CONNESSIONE NON È PRESENTE:** Creation Subsystem lancia una eccezione Bad Response
    a. Il controller impone il caricamento della pagina ERROR
    b. Il view Subsystem cambia la pagina da quella chiamante a quella ERROR e la renderizza

View Subsystem

Il view subsystem è quel sottosistema che contiene tutte le pagine di visualizzazione che vengono create in base alle necessità



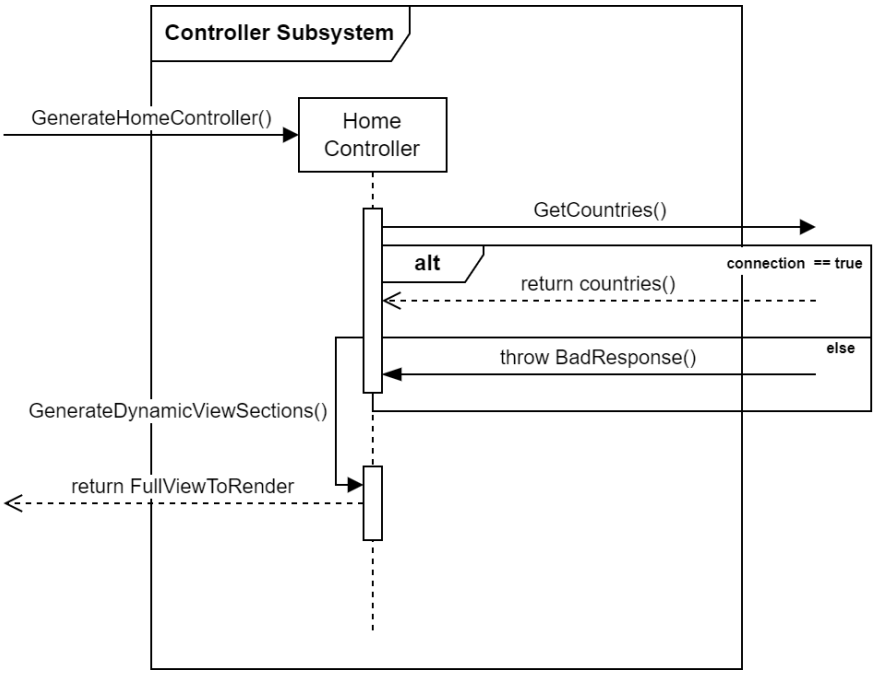
Passaggi:

1. il main crea una un nuova HomeView
2. La HomeView va a cercare il relativo file FXML in XML archive
3. HomeView Grazie a JavaFx cerca di caricare il file FXML
4. Il file FXML richiama HomeController che deve essere opportunamente generato
5. Il sottosistema del controller restituirà la pagina da renderizzare a schermo
6. JavaFX renderizza la pagina (in questo caso ho scritto Display ma non esiste una vera e propria funzione Display)

nota: il main è la classe principale di java, quindi anche se non è scritta nel Sequence Diagram completo abbiamo comunque pensato fosse giusto riproporla nella parte specifica alla view

Controller Subsystem

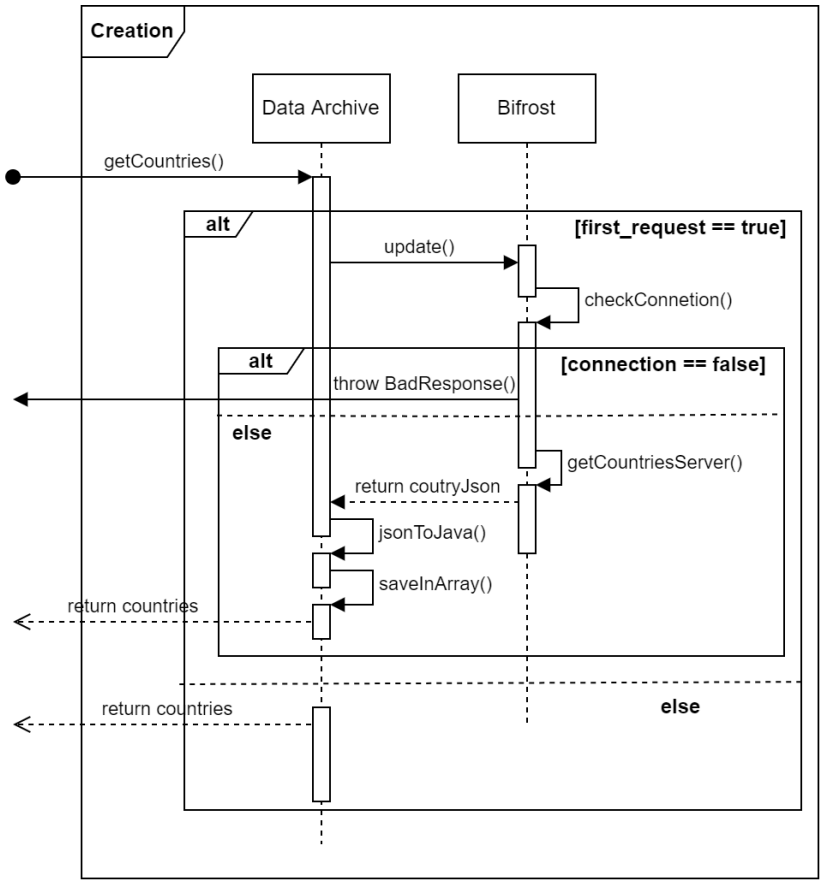
Il controller subsystem contiene i controller di ogni pagina, tali classi sono costruite per rendere dinamiche e interattive le pagine FXML



Passaggi:

1. Viene creato HomeController dal View Subsystem
2. l'home controller richiede le countries al creation Subsystem
3. **se l'applicativo è connesso ad internet** home controller riceve le countries dal creation subsystem
    a. HomeController inserisce i field dinamici all'interno della pagina
    b. HomeController restituisce la pagina renderizzata
4. **se l'applicativo non è connesso ad internet** home controller riceve l'errore lanciato dal creation subsystem
    a. HomeController cattura l'eccezione e carica la pagina "ErrorView" (in questo caso generalizzato a Generate Dynamic View Sections)
    b. HomeController restituisce la pagina di errore renderizzata

Creation Subsystem



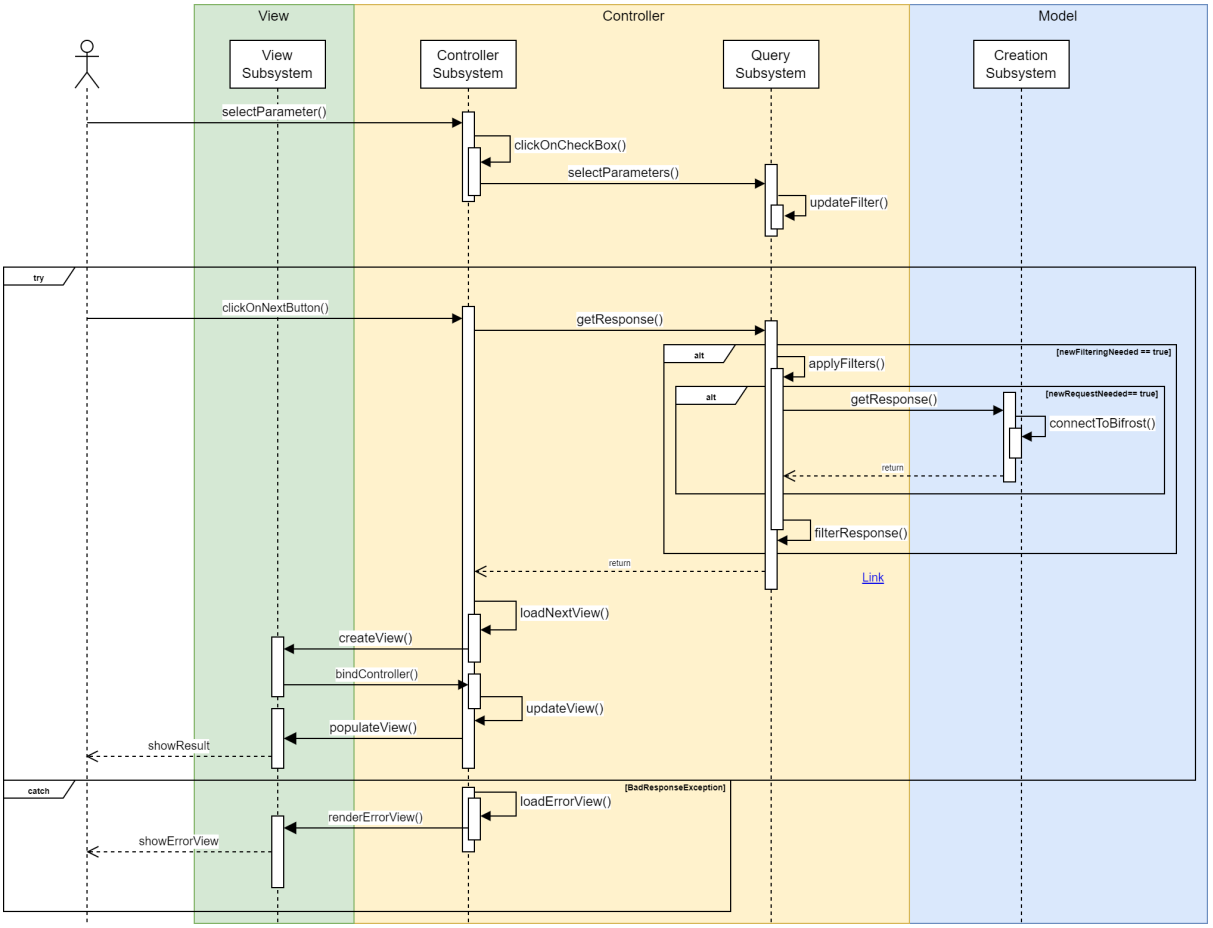
Passaggi:

1. Il dataArchive riceve la richiesta delle countries dal controller
2. **se è la prima richiesta** dataArchive richiede a Bifrost di ricavare i dati dal server.
3. Bifrost controlla la connessione
    a. **Se non sei connesso ad internet** Bifrost lancia BadResponseException
    b. **Se sei connesso ad internet** Bifrost effettua l'httpRequest e consegna i risultati a Data Archive
    c. Data Archive converte la risposta Json in oggetti Java
    d. gli oggetti Java vengono raggruppati in in un array e salvati nel DataArchive
    e. Viene restituito l'array di countries
4. **se non è la prima richiesta** restituisce l'array precedentemente salvato.

Search

Per semplicità di lettura, questo sequence diagram è una generalizzazione di 1 dei 4 passaggi che vengono effettuati dall'utente per completare una ricerca:

- **Selezione dei Country**
- **Selezione dei Service**
- **Selezione dei Provider**
- **Selezione degli stati del Service**



I passaggi sono:

1. L'utente seleziona i parametri cliccando sulle CheckBox contenute in una delle schermate.
2. Ad ogni click sulle CheckBox, il Controller dedicato alla specifica View, comunica l'aggiornamento dei filtri al Query Subsystem, che aggiornerà i suoi criteri di ricerca.
3. Al click sul bottone Next, il Controller chiederà al Query Subsystem di eseguire una query con i parametri precedentemente modificati.
4. **Se la query ha bisogno di filtrare i dati**
    a. **Se la query ha bisogno di connettersi all'API**
        i. La query richiede i dati necessari al Creation Subsystem, che tramite il bifrost comunicherà con l'API
    b. I dati salvati nella query vengono filtrati in base ai parametri scelti dall'utente
5. I dati filtrati vengono restituiti al Controller

6. **Se l’applicazione è connessa ad internet**
    a. I dati ricevuti dal Query Subsystem vengono elaborati dal Controller Subsystem e, contemporaneamente, viene richiesta la View successiva al View Subsystem.
    b. Il controller corrispondente alla View chiamata, aggiorna la View con i dati che il Controller Subsystem ha ottenuto in precedenza dal Query Subsystem.
    c. Una volta aggiornata la View, il View Subsystem restituisce all’utente la schermata richiesta.
7. **Se l’applicazione non è connessa ad internet**
    a. Il Controller Subsystem richiede al View Subsystem la ErrorView
    b. Il View Subsystem restituisce all’utente la schermata di errore.