



SIRIUS

SEQUENZIATORE

Definizione di prodotto

Versione 2.0.0

Ingegneria Del Software AA 2013-2014

Informazioni documento

Titolo documento:	Definizione Di Prodotto
Data creazione:	2014-04-18
Versione attuale:	2.0.0
Utilizzo:	Esterno
Nome file:	<i>DefinizioneDiProdotto_v2.0.0.pdf</i>
Redazione:	Vanni Giachin
Approvazione:	Santangelo Davide
Distribuito da:	Sirius
Destinato a:	Prof. Vardanega Tullio Prof. Cardin Riccardo Zucchetti S.p.A.

Sommario

Tale documento andrà a trattare in modo approfondito le componenti e la struttura del prodotto il *Sequenziatore* trattate nel documento *SpecificaTecnica_v3.0.0.pdf*

Diario delle modifiche

Versione	Data	Autore	Ruolo	Descrizione
1.0.0	2014-06-27	Santangelo Davide	Responsabile	Approvazione documento
0.1.4	2014-06-26	Seresin Davide	Verificatore	Verifica del documento
0.1.3	2014-06-24	Botter Marco	Progettista	Aggiunta metodi a classi di server.presenter
0.1.2	2014-06-20	Quaglio Davide	Progettista	Aggiunta metodi e modifica tipo parametri per client.presenter
0.1.1	2014-06-13	Vanni Giachin	Progettista	Modifica dei nomi per rispondere alle norme di progetto
0.1.0	2014-06-07	Santangelo Davide	Verificatore	Verifica documento
0.0.7	2014-06-05	Seresin Davide	Progettista	Aggiornamento metodi classi server.
0.0.6	2014-06-03	Seresin Davide	Progettista	Aggiunta classi server.model
0.0.5	2014-06-01	Quaglio Davide	Progettista	Aggiornamento classi server.presenter, aggiornati i nomi
0.0.4	2014-05-30	Quaglio Davide	Progettista	Definizione classi server.presenter
0.0.3	2014-05-26	Botter Marco	Progettista	Definizione classi client.presenter e client.model
0.0.2	2014-05-23	Giachin Vanni	Progettista	Definizione classi client.view
0.0.1	2014-05-15	Giachin Vanni	Progettista	Stesura scheletro

Indice

1	Introduzione	2
1.1	Scopo del documento	2
1.2	Scopo del Prodotto	2
1.3	Glossario	2
1.4	Riferimenti	2
1.4.1	Normativi	2
1.4.2	Informativi	2
2	Standard di progetto	4
2.1	Standard di progettazione architettuale	4
2.2	Standard di documentazione del codice	4
2.3	Standard di denominazione di entità e relazioni	4
2.4	Standard di programmazione	4
2.5	Strumenti di lavoro	4
3	Specifica della componente view	5
3.1	Package com.sirius.sequenziatore.client.view	6
3.2	Package com.sirius.sequenziatore.client.view.user	6
3.3	Package com.sirius.sequenziatore.client.view.processowner	7
4	Specifica della componente presenter	9
4.1	Client	9
4.1.1	Package com.sirius.sequenziatore.client.presenter	10
4.1.2	Package com.sirius.sequenziatore.client.presenter.user	13
4.1.3	Package com.sirius.sequenziatore.client.presenter.processowner	22
5	Specifica della componente Controller	38
5.0.4	Package com.sirius.sequenziatore.server.controller.common	38
5.0.5	Package com.sirius.sequenziatore.server.presenter.processowner	42
5.0.6	Package com.sirius.sequenziatore.server.presenter.user	46
6	Specifica della componente Service	49
6.1	Package com.sirius.sequenziatore.server.service	50
7	Specifica della componente model	58
7.1	Client	58
7.1.1	Package com.sirius.sequenziatore.client.model	58
7.1.2	Package com.sirius.sequenziatore.client.model.collection	62
7.2	Server	66
7.2.1	Package com.sirius.sequenziatore.client.model	66

8	Diagrammi di sequenza	86
8.1	Creazione di un processo	86
8.2	Approvazione di un passo	87
8.3	Registrazione	88

Elenco delle tabelle

Elenco delle figure

1	Diagramma classe <i>Router</i>	10
2	Diagramma classe <i>Login</i>	12
3	Diagramma classe <i>EventDispatcher</i>	13
4	Diagramma classe <i>MainUser</i>	14
5	Diagramma classe <i>Register</i>	15
6	Diagramma classe <i>OpenProcess</i>	17
7	Diagramma classe <i>ManagementProcess</i>	18
8	Diagramma classe <i>PrintReport</i>	19
9	Diagramma classe <i>SendData</i>	20
10	Diagramma classe <i>EventDispatcher</i>	22
11	Diagramma classe <i>MainProcessOwner</i>	23
12	Diagramma classe <i>OpenProcess</i>	24
13	Diagramma classe <i>NewProcess</i>	26
14	Diagramma classe <i>AddStep</i>	29
15	Diagramma classe <i>ManageProcess</i>	32
16	Diagramma classe <i>CheckStep</i>	34
17	Diagramma classe <i>ProcessData</i>	36
18	Diagramma package - <i>com.sirius.sequenziatore.server.controller.common</i> 38	
19	Diagramma classe - <i>SignUpController</i>	39
20	Diagramma classe - <i>LoginController</i>	40
21	Diagramma classe - <i>StepInfoController</i>	40
22	Diagramma classe - <i>ProcessInfoController</i>	41
23	Diagramma package - <i>com.sirius.sequenziatore.server.presenter.processowner</i> 42	
24	Diagramma classe - <i>StepController</i>	43
25	Diagramma classe - <i>ProcessController</i>	44
26	Diagramma classe - <i>ApproveStepController</i>	45
27	Diagramma package - <i>com.sirius.sequenziatore.server.controller.user</i> 46	
28	Diagramma classe - <i>UserStepController</i>	46
29	Diagramma classe - <i>UserProcessController</i>	47
30	Diagramma classe - <i>ReportController</i>	48
31	Diagramma package - <i>com.sirius.sequenziatore.server.service</i>	50
32	Diagramma classe - <i>SignUpService</i>	50
33	Diagramma classe - <i>ApproveStepService</i>	51
34	Diagramma classe - <i>ProcessInfoService</i>	52
35	Diagramma classe - <i>StepInfoService</i>	52

36	Diagramma classe - <code>LoginService</code>	53
37	Diagramma classe - <code>ProcessService</code>	54
38	Diagramma classe - <code>UserProcessService</code>	55
39	Diagramma classe - <code>UserStepService</code>	56
40	Diagramma classe - <code>StepService</code>	56
41	Diagramma classe - <code>ReportService</code>	57
42	Diagramma classe <i>UserModel</i>	58
43	Diagramma classe <i>ProcessModel</i>	59
44	Diagramma classe <i>ProcessDataModel</i>	60
45	Diagramma classe <i>StepModel</i>	61
46	Diagramma classe <i>ProcessCollection</i>	62
47	Diagramma classe <i>ProcessDataCollection</i>	63
48	Diagramma classe <i>StepCollection</i>	64
49	Diagramma interfaccia <code>IDataAccessObject</code>	66
50	Diagramma classe <code>UserDao</code>	67
51	Diagramma classe <code>ProcessDao</code>	68
52	Diagramma classe <code>ProcessOwnerDao</code>	69
53	Diagramma classe <code>StepDao</code>	70
54	Diagramma classe <code>User</code>	72
55	Diagramma classe <code>Process</code>	74
56	Diagramma classe <code>Step</code>	76
57	Diagramma classe <code>DataSent</code>	78
58	Diagramma interfaccia <code>IDataValue</code>	79
59	Diagramma classe <code>TextualValue</code>	80
60	Diagramma classe <code>NumericValue</code>	81
61	Diagramma classe <code>ImageValue</code>	82
62	Diagramma classe <code>GeographicValue</code>	83
63	Diagramma classe <code>UserStep</code>	84
64	Diagramma classe <code>ProcessOwner</code>	85

1 Introduzione

1.1 Scopo del documento

In questo documento si prefigge come obiettivo la definizione in modo approfondito della struttura e delle relazioni tra le componenti del prodotto *software Sequenziatore*, approfondendo quanto riportato nel documento *SpecificaTecnica_v3.0.0.pdf*.

1.2 Scopo del Prodotto

Lo scopo del progetto *Sequenziatore*, è di fornire un servizio di gestione di processi definiti da una serie di passi da eseguirsi in sequenza o senza un ordine predefinito, utilizzabile da dispositivi mobili di tipo *smartphone* o *tablet*.

1.3 Glossario

Al fine di rendere più leggibili e comprensibili i documenti, i termini tecnici, di dominio, gli acronimi e le parole che necessitano di essere chiarite, sono riportate nel documento *Glossario_v4.0.0.pdf*.

Ciascuna occorrenza dei vocaboli presenti nel *Glossario* è seguita da una “G” maiuscola in pedice.

1.4 Riferimenti

1.4.1 Normativi

- Norme di Progetto: *NormeDiProgetto_v4.0.0.pdf*;
- Analisi dei Requisiti: *AnalisiDeiRequisiti_v3.0.0.pdf*;
- Specifica tecnica: *SpecificaTecnica_v3.0.0.pdf*.

1.4.2 Informativi

- Developing Backbone.js Applications, Addy Osmani
<http://addyosmani.github.io/backbone-fundamentals>;
- BackboneJS
<http://backbonejs.org/>;
- Documentazione Spring.io
<http://spring.io/docs>;
- Regolamento dei documenti, prof. Vardanega Tullio:
<http://www.math.unipd.it/~tullio/IS-1/2013/>;

- Dispense di ingegneria del software:
 - Programmazione: criteri e strategie, prof. Vardanega Tullio:
<http://www.math.unipd.it/~rcardin/pdf/B02.pdf>;
 - Diagrammi delle classi e degli oggetti, prof. Cardin Riccardo:
<http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/E02a.pdf>;
 - Diagrammi di sequenza, prof. Cardin Riccardo:
<http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/E03a.pdf>;
 - Diagrammi dei package, prof. Cardin Riccardo:
<http://www.math.unipd.it/~tullio/IS-1/2013/Dispense/E05.pdf>;

2 Standard di progetto

2.1 Standard di progettazione architettuale

Gli standard di progettazione architettuale sono definiti nel documento *Specifica Tecnica_v3.0.0.pdf*.

2.2 Standard di documentazione del codice

Gli standard di documentazione del codice sono definiti nel documento *NormeDiProgetto_v4.0.0.pdf*.

2.3 Standard di denominazione di entità e relazioni

Gli standard di denominazione di dei *package*, delle classi, degli attributi e dei metodi, sono definiti nel documento *NormeDiProgetto_v4.0.0.pdf*.

2.4 Standard di programmazione

Gli standard di programmazione sono definiti nel documento *NormeDiProgetto_v4.0.0.pdf*.

2.5 Strumenti di lavoro

Gli strumenti da utilizzare e le procedure da seguire durante lo sviluppo del prodotto *software Sequenziatore*, sono definiti nel documento *NormeDiProgetto_v4.0.0.pdf*.

3 Specifica della componente view

La componente *view* è formata da *template HTML_G* che possono contenere codice *javascript_G* che, utilizzati dalle componenti del *presenter*, consentono di renderizzare l'interfaccia grafica dell'applicazione.

Le componenti del *presenter*, si interfacciano con la *view* utilizzando il metodo `template` della libreria *underscoreJS*, che consente di generare codice *HTML_G* a seconda dei parametri del metodo. Per questo motivo, le interfacce presenti nel *package* `com.sirius.sequenziatore.client.view` definite nel documento *SpecificaTecnica_v3.0.0.pdf*, non verranno né implementate né descritte nel presente documento.

La componente *view* è composta dai seguenti *template*:

- `com.sirius.sequenziatore.client.view.Login`;
- `com.sirius.sequenziatore.client.view.user.MainUser`;
- `com.sirius.sequenziatore.client.view.user.Register`;
- `com.sirius.sequenziatore.client.view.user.UserData`;
- `com.sirius.sequenziatore.client.view.user.OpenProcess`;
- `com.sirius.sequenziatore.client.view.user.ManagementProcess`;
- `com.sirius.sequenziatore.client.view.user.SendData`;
- `com.sirius.sequenziatore.client.view.user.SendText`;
- `com.sirius.sequenziatore.client.view.user.SendNumb`;
- `com.sirius.sequenziatore.client.view.user.SendPosition`;
- `com.sirius.sequenziatore.client.view.user.SendImage`;
- `com.sirius.sequenziatore.client.view.user.PrintProcess`;
- `com.sirius.sequenziatore.client.view.processowner.MainProcessOwner`;
- `com.sirius.sequenziatore.client.view.processowner.NewProcess`;
- `com.sirius.sequenziatore.client.view.processowner.AddStep`;
- `com.sirius.sequenziatore.client.view.processowner.OpenProcess`;
- `com.sirius.sequenziatore.client.view.processowner.ManageProcess`;
- `com.sirius.sequenziatore.client.view.processowner.CheckStep`;

3.1 Package `com.sirius.sequenziatore.client.view`

3.1.0.1 Login

- **Descrizione:** *Template HTML* che permette di gestire l'interfaccia grafica relativa alle richieste di autenticazione al sistema.

3.2 Package `com.sirius.sequenziatore.client.view.user`

3.2.0.2 MainUser

- **Descrizione:** Classe che permette la gestione delle principali componenti dell'interfaccia grafica dell'utente.

3.2.0.3 Register

- **Descrizione:** *Template HTML* che permette di gestire dell'interfaccia grafica relativa alle richieste di registrazione da parte dell'utente.

3.2.0.4 UserData

- **Descrizione:** *Template HTML* che permette la realizzazione dei *widget* che consentono visualizzazione e modifica dei dati dell'utente.

3.2.0.5 OpenProcess

- **Descrizione:** *Template HTML* che permette di realizzare i *widget* per consentire l'apertura di un processo tramite ricerca o selezionandolo da una lista.

3.2.0.6 ManagementProcess

- **Descrizione:** *Template HTML* che permette di realizzare i *widget* per consentire la visualizzazione dello stato del processo selezionato e i vincoli per concludere il passo in corso.

3.2.0.7 SendData

- **Descrizione:** *Template HTML* che permette di realizzare i *widget* per consentire l'invio dei dati richiesti per la conclusione del passo in esecuzione.

3.2.0.8 SendText

- **Descrizione:** *Template HTML* che permette di realizzare i *widget* che consentono di inserire il testo da inviare per concludere il passo in esecuzione.

3.2.0.9 SendNumb

- **Descrizione:** *Template HTML* che permette agli oggetti che la implementano di realizzare i *widget* che consentono di inserire i dati numerici da inviare per concludere il passo in esecuzione.

3.2.0.10 SendPosition

- **Descrizione:** *Template HTML* che permette di realizzare i *widget* che consentono di inviare la posizione geografica richiesta per la conclusione del passo in esecuzione.

3.2.0.11 SendImage

- **Descrizione:** *Template HTML* che permette di realizzare i *widget* che consentono di inserire le immagini richieste per concludere il passo in esecuzione.

3.2.0.12 PrintProcess

- **Descrizione:** *Template HTML* che permette di realizzare i *widget* che consentono il salvataggio dei *report* sull'esecuzione del processo.

3.3 Package com.sirius.sequenziatore.client.view.processowner

3.3.0.13 MainProcessOwner

- **Descrizione:** Componente che permette la gestione delle principali componenti dell'interfaccia grafica dell'utente *process owner*.

3.3.0.14 NewProcess

- **Descrizione:** *Template HTML* che permette di gestire l'interfaccia grafica che consente di creare nuovi processi.

3.3.0.15 AddStep

- **Descrizione:** *Template HTML* che permette di gestire l'interfaccia grafica che consente di definire un nuovo passo del processo in creazione.

3.3.0.16 OpenProcess

- **Descrizione:** *Template HTML* che permette di realizzare i *widget* che consentono di aprire un processo tramite ricerca o selezionandolo da una lista.

3.3.0.17 ManageProcess

- **Descrizione:** *Template HTML* che permette di realizzare *iwidget* che consentono di gestire l'accesso ai dati inviati al *server_G* dagli utenti.

3.3.0.18 CheckStep

- **Descrizione:** *Template HTML* che permette di realizzare *iwidget* che consentono di gestire l'approvazione dei passi che richiedono intervento umano.

4 Specifica della componente presenter

Questa componente consente la gestione della logica principale dell'applicazione *Sequenziatore* e viene suddivisa in due parti: *client* e *server*.

4.1 Client

Il *presenter* lato *client* consente di gestire la logica delle pagine dell'applicazione. La inizializzazione delle classi e la gestione degli eventi di cambio pagina, avviene tramite la classe principale **Router**, che estende la classe **Backbone.Router** fornita dal *framework_G Backbone*. Le altre classi della componente, consentono di renderizzare le viste utilizzando i *template* della componente *view*, di gestire gli eventi generati dagli utenti, e di gestire la comunicazione con il server tramite le classi della componente *model*. Ogni classe presenter, tranne le classi derivate da *BaseDispatcher*, è figlia di *BasePresenter*.

La componente è formata dalle seguenti *classi*:

- `com.sirius.sequenziatore.client.presenter.Router`;
- `com.sirius.sequenziatore.client.presenter.Login`;
- `com.sirius.sequenziatore.client.presenter.user.MainUser`;
- `com.sirius.sequenziatore.client.presenter.user.Register`;
- `com.sirius.sequenziatore.client.presenter.user.OpenProcess`;
- `com.sirius.sequenziatore.client.presenter.user.ManageProcess`;
- `com.sirius.sequenziatore.client.presenter.user.EventDispatcher`;
- `com.sirius.sequenziatore.client.presenter.user.SendData`;
- `com.sirius.sequenziatore.client.presenter.user.PrintReport`;
- `com.sirius.sequenziatore.client.presenter.processowner.MainProcessOwner`;
- `com.sirius.sequenziatore.client.presenter.processowner.NewProcess`;
- `com.sirius.sequenziatore.client.presenter.processowner.AddStep`;
- `com.sirius.sequenziatore.client.presenter.processowner.OpenProcess`;
- `com.sirius.sequenziatore.client.presenter.processowner.ManageProcess`;
- `com.sirius.sequenziatore.client.presenter.processowner.EventDispatcher`;
- `com.sirius.sequenziatore.client.presenter.processowner.CheckStep`;
- `com.sirius.sequenziatore.client.presenter.processowner.ProcessData`.

4.1.1 Package com.sirius.sequenziatore.client.presenter

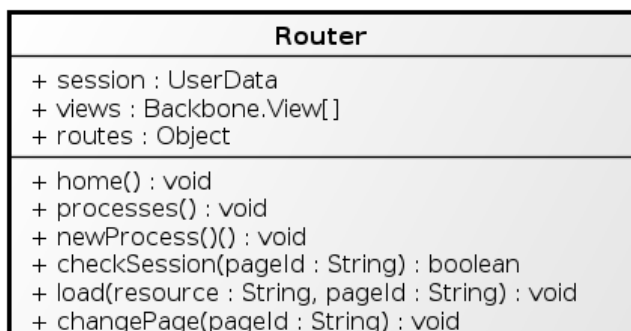


Figura 1: Diagramma classe *Router*

4.1.1.1 Router

- **Descrizione:** Classe che permette di coordinare l'inizializzazione e la renderizzazione delle pagine, gestendo gli eventi e le azioni di cambio pagina;
- **Relazioni con altri componenti:**

La classe reperisce le informazioni di sessione dalla classe `com.sirius.sequenziatore.client.model::UserModel` e comunica con le seguenti classi se l'utente dispone dei diritti d'accesso necessari:

- `com.sirius.sequenziatore.client.presenter.Login;`
- `com.sirius.sequenziatore.client.presenter.user.Register;`
- `com.sirius.sequenziatore.client.presenter.user.MainUser;`
- `com.sirius.sequenziatore.client.presenter.user.UserData;`
- `com.sirius.sequenziatore.client.presenter.user.OpenProcessgic;`
- `com.sirius.sequenziatore.client.presenter.user.ManagmentProcess;`
- `com.sirius.sequenziatore.client.presenter.processowner.MainProcessOwner;`
- `com.sirius.sequenziatore.client.presenter.processowner.OpenProcess;`
- `com.sirius.sequenziatore.client.presenter.processowner.NewProcess;`
- `com.sirius.sequenziatore.client.presenter.processowner.CheckStep;`

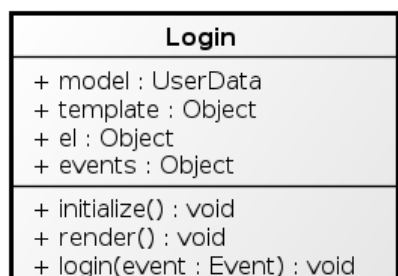
- `com.sirius.sequenziatore.client.presenter.processowner.ManageProcess;`

- **Attributi:**

- `+ UserData session:`
oggetto di tipo `com.sirius.sequenziatore.client.model.UserData`, che consente di gestire la sessione dell'utente;
- `+ Backbone.View[] views:`
array che contiene le classi del presenter in esecuzione;
- `+ Object routes:`
oggetto ridefinito da `Backbone.Router` che associa ad ogni evento di `routingG`, un metodo della classe;

- **Metodi:**

- `+ void home():`
gestisce l'evento di `routingG home`;
- `+ void processes():`
gestisce l'evento di `routingG processes`;
- `+ void newProcess():`
gestisce l'evento di `routingG newProcess`;
- `+ void checkStep():`
gestisce l'evento di `routingG checkStep`;
- `+ void process():`
gestisce l'evento di `routingG process`;
- `+ void register():`
gestisce l'evento di `routingG register`;
- `+ void user():`
gestisce l'evento di `routingG user`;
- `+ bool checkSession(String pageId):`
ritorna `true` solo se l'utente è autenticato; in caso contrario crea e renderizza la pagina di `login`;
- `+ void load(String resource, String pageId):`
crea e aggiunge una vista di tipo `resource` al campo dati `this.views`, all'indice `pageId`;
- `+ void changePage(String pageId):`
imposta la pagina con id `pageId` come attiva, ed esegue la transizione di cambio pagina.


Figura 2: Diagramma classe *Login*

4.1.1.2 Login

- **Descrizione:** Classe che ha il compito di gestire le richieste di autenticazione al sistema;

- **Relazioni con altri componenti:**

La classe gestisce i dati di sessione comunicando con la classe `com.sirius.sequenziatore.client.model.UserModel` e realizza l'interfaccia grafica utilizzando il *template* `com.sirius.sequenziatore.client.viewLogin`.

- **Attributi:**

- + `UserDataModel model`:
campo dati di tipo `com.sirius.sequenziatore.client.model.UserModel` che contiene i dati di sessione dell'utente;
- + `Object template`:
oggetto ridefinito da `Backbone.View`, che contiene il *template* `HTMLG` associato alla classe;
- + `Object el`:
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento `HTMLG` entro cui la classe ascolta eventi generati dagli utenti;
- + `Object events`:
oggetto ridefinito da `Backbone.View` che associa ad ogni evento generato dagli utenti nella pagina `HTMLG`, un metodo della classe;

- **Metodi:**

- + `void initialize()`:
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina `HTMLG` associata al componente;

- + void render():
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina $HTML_G$ il *template* campo dati della classe;
- + void login(Event event):
effettua una richiesta di *login*, utilizzando il campo dati `com.sirius.sequenziatore.client.model` per comunicare con il *server_G*.

4.1.2 Package `com.sirius.sequenziatore.client.presenter.user`

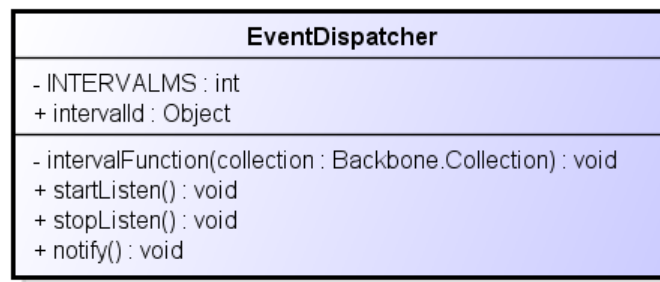


Figura 3: Diagramma classe *EventDispatcher*

4.1.2.1 EventDispatcher

- **Descrizione:** Classe per la gestione della notifica di presenza di passi che richiedono approvazione.

- **Relazioni con altri componenti:**

La classe estende la classe
`com.sirius.sequenziatore.client.presenter.BaseDispatcher`.

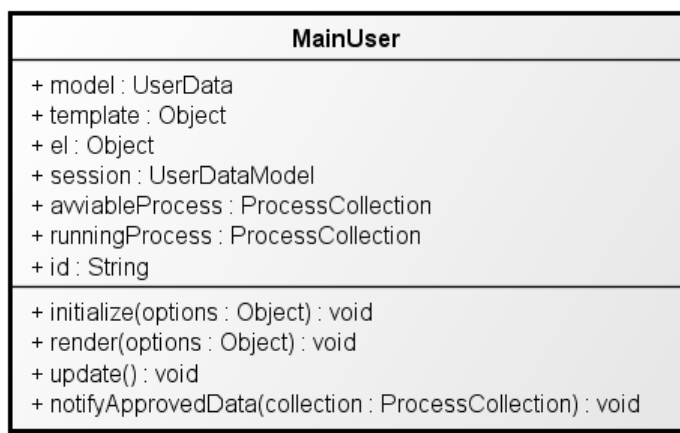
- **Attributi:**

- - int INTERVALMS : indica l'intervallo di pooling;
- +intervalId : indica l'id dell'intervallo;

- **Metodi:**

- - intervalFunction(collection : Backbone.Collection) : void:
invoca il metodo notify se la collezione contiene dei nuovi passi che richiedono approvazione;
- + startListen() : void :
inizia a monitorare eventuali variazioni nella collezione di passi che richiedono approvazione;

- + `stopListen() : void :`
interrompe l'esecuzione della funzione periodica con id `intervalId`;
- + `notify() : void:`
notifica gli observer.


Figura 4: Diagramma classe *MainUser*

4.1.2.2 MainUser

- **Descrizione:** Classe che ha il compito della gestione generale della logica delle funzionalità utente;

- **Relazioni con altri componenti:**

La classe comunica con l'interfaccia `com.sirius.sequenziatore.client.view.user.IMainUser` per la realizzazione dell'interfaccia grafica, e `ProcessCollection` per il recupero della lista dei processi attivi e da iscrivere.

- **Attributi:**

- + `UserDataModel model:`
campo dati di tipo
`com.sirius.sequenziatore.client.model.UserModel` che contiene i dati di sessione dell'utente;
- + `ProcessCollection avviableProcess:`
campo dati contenente i processi iscrivibili dall'utente;
- + `ProcessCollection runningProcess:`
campo dati contenente i processi iscritti dall'utente;

- + **Object** *template*:
oggetto ridefinito da **Backbone.View**, che contiene il *template HTML_G* associato alla classe;
- + **Object** *el*:
oggetto ridefinito da **Backbone.View** che rappresenta l'elemento *HTML_G* entro cui la classe ascolta eventi generati dagli utenti;
- + **String** *id*:
campo dati ridefinito da **Backbone.View** contenente l'id della classe;

• Metodi:

- + **initialize**(options : **Object**) : **void**:
metodo ridefinito da **Backbone.View**, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + **render**(options : **Object**) : **void**:
metodo ridefinito da **Backbone.View**, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;
- + **update**() : **void** : **void**:
aggiorna le collezioni di processi recuperando i dati dal server;
- + **notifyApprovedData**(collection : **ProcessCollection**) : **void**:
metodo per la gestione della notifica della presenta di un nuovo passo approvato/respinto.

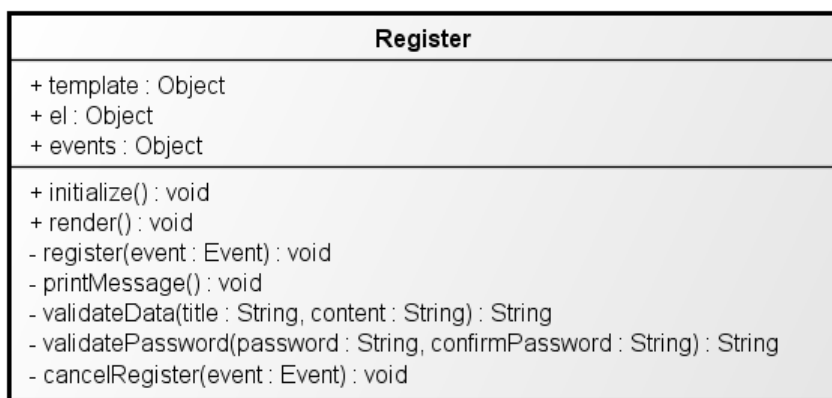


Figura 5: Diagramma classe *Register*

4.1.2.3 Register

- **Descrizione:** Classe che ha il compito di gestire le richieste di registrazione da parte dell'utente;

- **Relazioni con altri componenti:**

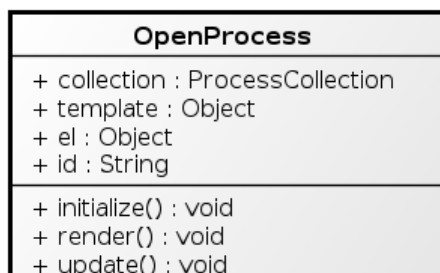
La classe comunica con l'interfaccia `com.sirius.sequenziatore.client.view.user.IRegister` per la realizzazione dei *widget* per la registrazione.

- **Attributi:**

- + **Object template:**
oggetto ridefinito da `Backbone.View`, che contiene il *template HTML_G* associato alla classe;
- + **Object el:**
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento *HTML_G* entro cui la classe ascolta eventi generati dagli utenti;
- + **Object events:**
oggetto ridefinito da `Backbone.View` che associa ad ogni evento generato dagli utenti nella pagina *HTML_G*, un metodo della classe;

- **Metodi:**

- + **void initialize():**
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + **void render():**
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;
- - **void register(Event event):**
effettua una richiesta di registrazione, utilizzando il campo dati `com.sirius.sequenziatore.client.model` per comunicare con il *server_G*;
- - **printMessage() : void :**
apre un popup con titolo `title` e contenuto `content`;
- - **validateData(title : String, content : String) : String :**
valida la data;
- - **validatePassword(password : String, confirmPassword : String) : String:**
valida la password;
- - **cancelRegister(event : Event) : void :**
gestisce l'evento per eliminare la registrazione.


Figura 6: Diagramma classe *OpenProcess*

4.1.2.4 OpenProcess

- **Descrizione:** Classe che ha il compito di selezionare, ricercare e aprire un processo fra quelli eseguibili;

- **Relazioni con altri componenti:**

La classe realizza e modifica l'opportuno *widget* mediante l'interfaccia `com.sirius.sequenziatore.client.view.user.IOpenProcess` e utilizza la classe

`com.sirius.sequenziatore.client.model.collection.ProcessCollection` per gestire e ottenere i dati dal *server_G*.

- **Attributi:**

- + `ProcessCollection` `collection`:
campo dati di tipo `com.sirius.sequenziatore.client.model.collection.ProcessCollection` che contiene la lista dei processi non terminati o non ancora eliminati dall'utente;
- + `Object` `template`:
oggetto ridefinito da `Backbone.View`, che contiene il *template HTML_G* associato alla classe;
- + `Object` `el`:
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento *HTML_G* entro cui la classe ascolta eventi generati dagli utenti;
- + `String` `id`:
campo dati ridefinito da `Backbone.View` contenente l'id della classe;

- **Metodi:**

- + `void initialize()`:
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;

- + void render():
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;
- + void update():
aggiorna il campo dati `collection` comunicando con il *server_G*.

4.1.2.5 ManagementProcess

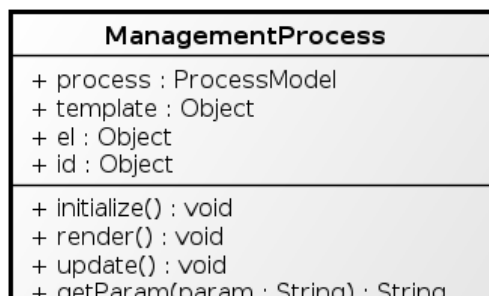


Figura 7: Diagramma classe *ManagementProcess*

- **Descrizione:** Classe che ha il compito di gestire e accedere alle informazioni relative allo stato del processo selezionato.;
- **Relazioni con altri componenti:**

La classe comunica con l'interfaccia `com.sirius.sequenziatore.client.view.user.IManagmentProcess` per realizzare il *widget* che permette la gestione del processo selezionato, utilizza la classe `com.sirius.sequenziatore.client.model.ProcessModel` per gestire e ottenere i dati dal *server_G*, e provvede ad invocare le seguenti classi in base alle decisioni dell'utente:

- `com.sirius.sequenziatore.client.presenter.user.PrintReport;`
- `com.sirius.sequenziatore.client.presenter.user.SendData.`

- **Attributi:**

- + `ProcessModel process:`
campo dati di tipo `com.sirius.sequenziatore.client.model.ProcessModel` che contiene i dati del processo in gestione;
- + `Object template:`
oggetto ridefinito da `Backbone.View`, che contiene il *template HTML_G* associato alla classe;

- + `Object el`:
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento *HTML_G* entro cui la classe ascolta eventi generati dagli utenti;
- + `String id`:
campo dati ridefinito da `Backbone.View` contenente l'id della classe;

• Metodi:

- + `void initialize()`:
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + `void render()`:
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;
- + `void update()`:
aggiorna i campi dati `process` e `processData` comunicando con il *server_G*;
- + `String getParam(String param)`:
ritorna il valore del parametro *param* se presente nella *URL_G*.

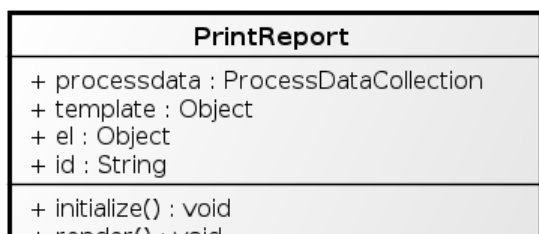


Figura 8: Diagramma classe *PrintReport*

4.1.2.6 PrintReport

- **Descrizione:** Classe che ha il compito di gestire la creazione del report di fine processo;
- **Relazioni con altri componenti:**

La classe comunica con l'interfaccia `com.sirius.sequenziatore.client.view.user.IPrintReport` per realizzare il *widget* per creare il report di fine processo, e utilizza la classe `com.sirius.sequenziatore.client.model.collection.ProcessDataCollection` per gestire e ottenere i dati dal *server_G*.

• Attributi:

- + `ProcessDataCollection processdata`:
campo dati di tipo `com.sirius.sequenziatore.client.model.collection.ProcessDataCollection` che contiene i dati inviati dall'utente relativi al processo in gestione;
- + `Object template`:
oggetto ridefinito da `Backbone.View`, che contiene il *template* `HTMLG` associato alla classe;
- + `Object el`:
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento `HTMLG` entro cui la classe ascolta eventi generati dagli utenti;
- + `String id`:
campo dati ridefinito da `Backbone.View` contenente l'id della classe;

• Metodi:

- + `void initialize()`:
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina `HTMLG` associata al componente;
- + `void render()`:
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina `HTMLG` il *template* campo dati della classe.

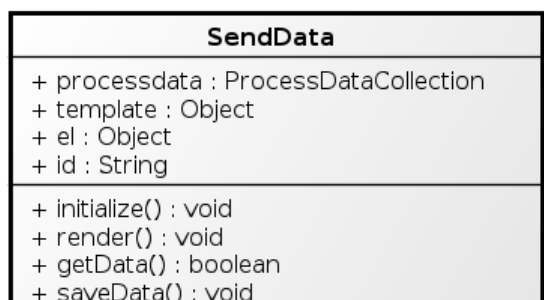


Figura 9: Diagramma classe *SendData*

4.1.2.7 SendData

- **Descrizione:** Classe che ha il compito di gestire l'inserimento e l'invio di dati da parte degli utenti, per completare il passo corrente;

- **Relazioni con altri componenti:**

La classe comunica con l'interfaccia `com.sirius.sequenziatore.client.view.user.ISendData` per creare il *widget* che consente di inviare i dati, utilizza la classe `com.sirius.sequenziatore.client.model.collection.ProcessDataCollection` per gestire e ottenere i dati dal *server_G*, e infine invoca le seguenti classi che gestiscono l'invio di un tipo di dato specifico:

- `com.sirius.sequenziatore.client.presenter.user.SendText;`
- `com.sirius.sequenziatore.client.presenter.user.SendNumb;`
- `com.sirius.sequenziatore.client.presenter.user.SendImage;`
- `com.sirius.sequenziatore.client.presenter.user.SendPosition.`

- **Attributi:**

- + `ProcessDataCollection processdata:`
campo dati di tipo `com.sirius.sequenziatore.client.model.collection.ProcessDataCollection` che consente di interagire con la lista dei dati inviati dall'utente relativa al processo in gestione presente nel *server_G*;
- + `Object template:`
oggetto ridefinito da `Backbone.View`, che contiene il *template HTML_G* associato alla classe;
- + `Object el:`
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento *HTML_G* entro cui la classe ascolta eventi generati dagli utenti;
- + `String id:`
campo dati ridefinito da `Backbone.View` contenente l'id della classe;

- **Metodi:**

- + `void initialize():`
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + `void render():`
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe. Utilizza le classi `com.sirius.sequenziatore.client.presenter.user.SendText`, `com.sirius.sequenziatore.client.presenter.user.SendNumb`, `com.sirius.sequenziatore.client.presenter.user.SendImage` e

`com.sirius.sequenziatore.client.presenter.user.SendPosition` per renderizzare l'interfaccia relativa all'inserimento dei diversi tipi di dato;

- `+ bool getData()`:
controlla se i dati inseriti dall'utente sono corretti: se lo sono ritorna `true` e li aggiunge alla collezione `processData`, altrimenti ritorna `false`;
- `+ bool saveData()`:
utilizza metodi del campo dati `processData`, per inviare i dati raccolti al `serverG`.

4.1.3 Package `com.sirius.sequenziatore.client.presenter.processowner`

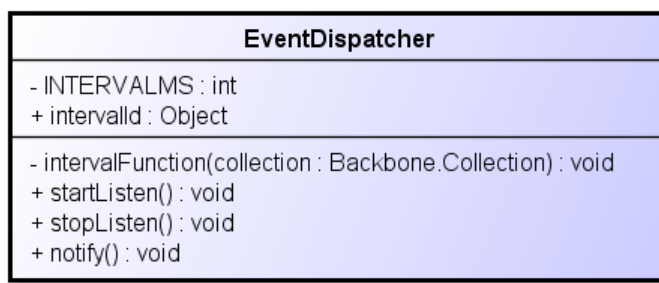


Figura 10: Diagramma classe *EventDispatcher*

4.1.3.1 EventDispatcher

- **Descrizione:** Classe per la gestione della notifica di presenza di passi che richiedono approvazione.

- **Relazioni con altri componenti:**

La classe estende la classe

`com.sirius.sequenziatore.client.presenter.BaseDispatcher`.

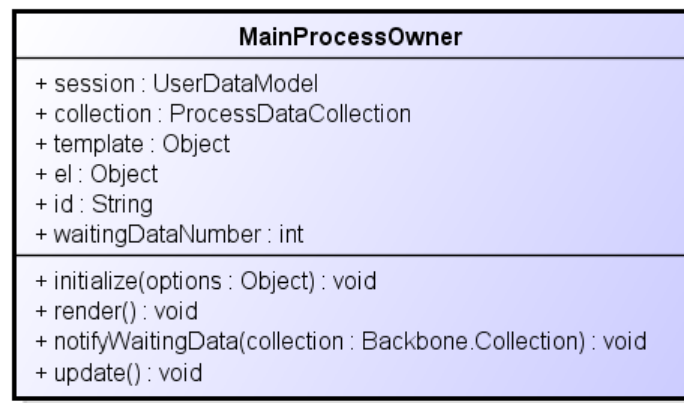
- **Attributi:**

- `- int INTERVALMS` : indica l'intervallo di pooling;
- `+intervalId` : indica l'id dell'intervallo;

- **Metodi:**

- `- intervalFunction(collection : Backbone.Collection) : void`:
invoca il metodo `notify` se la collezione contiene dei nuovi passi che richiedono approvazione;

- + `startListen()` : `void` :
inizia a monitorare eventuali variazioni nella collezione di passi che richiedono approvazione;
- + `stopListen()` : `void` :
interrompe l'esecuzione della funzione periodica con id `intervalId`;
- + `notify()` : `void`:
notifica gli observer.


Figura 11: Diagramma classe *MainProcessOwner*

4.1.3.2 MainProcessOwner

- **Descrizione:** Classe che ha il compito della gestione generale della logica delle funzionalità *Process Owner_G*;

- **Relazioni con altri componenti:**

La classe comunica con il *template*

`com.sirius.sequenziatore.client.view.processowner.IMainProcessOwner` per la realizzazione dell'interfaccia grafica, con

`com.sirius.sequenziatore.client.model.UserDataModel` per la gestione della sessione e con `com.sirius.sequenziatore.client.model.processowner.collection.ProcessDataCollection`

- **Attributi:**

- + `UserDataModel session` : variabile necessaria per la gestione della sessione;
- + `int waitingDataNumber` : numero di dati attesi;
- + `ProcessDataCollection collection` : variabile necessaria per la gestione dei dati ricevuti dagli utenti riguardanti un processo

- + **Object** *template*:
oggetto ridefinito da **Backbone.View**, che contiene il *template HTML_G* associato alla classe;
- + **Object** *el*:
oggetto ridefinito da **Backbone.View** che rappresenta l'elemento *HTML_G* entro cui la classe ascolta eventi generati dagli utenti;
- + **String** *id*:
campo dati ridefinito da **Backbone.View** contenente l'id della classe;

• Metodi:

- + **initialize**(options : **Object**) : **void**
metodo ridefinito da **Backbone.View**, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + **render**() : **void**
metodo ridefinito da **Backbone.View**, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;
- + **notifyWaitingData**(collection : **Backbone.Collection**) : **void**:
permette la gestione della notifica dell'evento: un passo richiede approvazione;
- + **update**() : **void**:
aggiorna il numero di passi che richiedono approvazione.

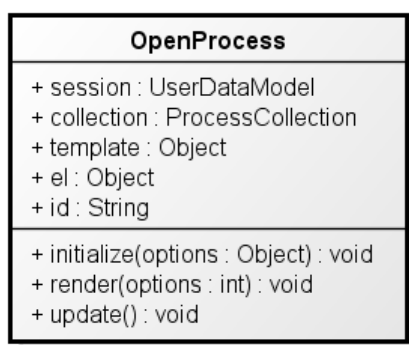


Figura 12: Diagramma classe *OpenProcess*

4.1.3.3 OpenProcess

- **Descrizione:** Classe che ha il compito di gestire la ricerca e la selezione di un processo;

- **Relazioni con altri componenti:**

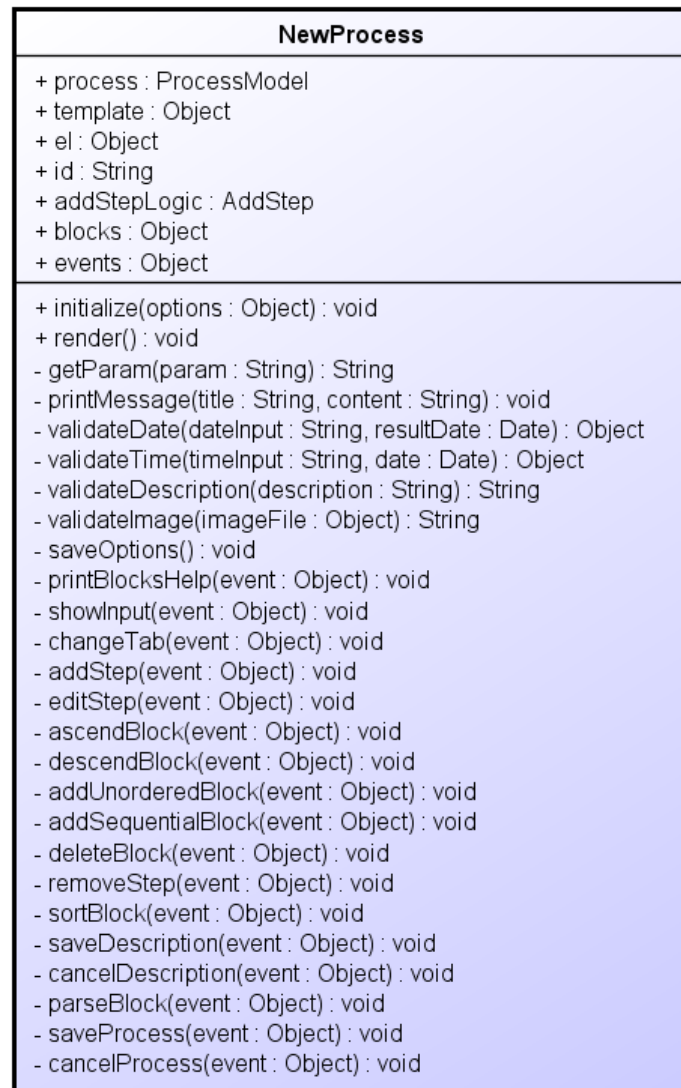
La classe comunica con il *template* `com.sirius.sequenziatore.client.view.processowner.IOpenProcess` per la realizzazione dell'interfaccia grafica, e con la classe `com.sirius.sequenziatore.client.model.collectionProcessCollection` per gestire e ottenere i dati dal *server_G*.

- **Attributi:**

- + `UserDataModel session:`
campo dati di tipo `com.sirius.sequenziatore.client.presenter.UserDataModel` utile per la gestione e recupero dati della sessione;
- + `ProcessCollection collection:`
campo dati di tipo `com.sirius.sequenziatore.client.model.processowner.collection.ProcessCollection` che contiene la lista dei processi non eliminati dal *process owner_G*;
- + `Object template:`
oggetto ridefinito da `Backbone.View`, che contiene il *template HTML_G* associato alla classe;
- + `Object el:`
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento *HTML_G* entro cui la classe ascolta eventi generati dagli utenti;
- + `String id:`
campo dati ridefinito da `Backbone.View` contenente l'id della classe;

- **Metodi:**

- + `initialize(options : Object) : void:`
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + `render(options : int) : void:`
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;
- + `void update():`
aggiorna il campo dati `collection` comunicando con il *server_G*.


Figura 13: Diagramma classe *NewProcess*

4.1.3.4 NewProcess

- **Descrizione:** Classe che ha il compito di gestire la logica della definizione di un nuovo processo;
- **Relazioni con altri componenti:**
La classe comunica con il *template* `com.sirius.sequenziatore.client.view.processowner.INewprocess` per la realizzazione dell'interfaccia grafica e con la classe `com.sirius.sequenziatore.client.presenter.processowner.AddStep`;
- **Attributi:**

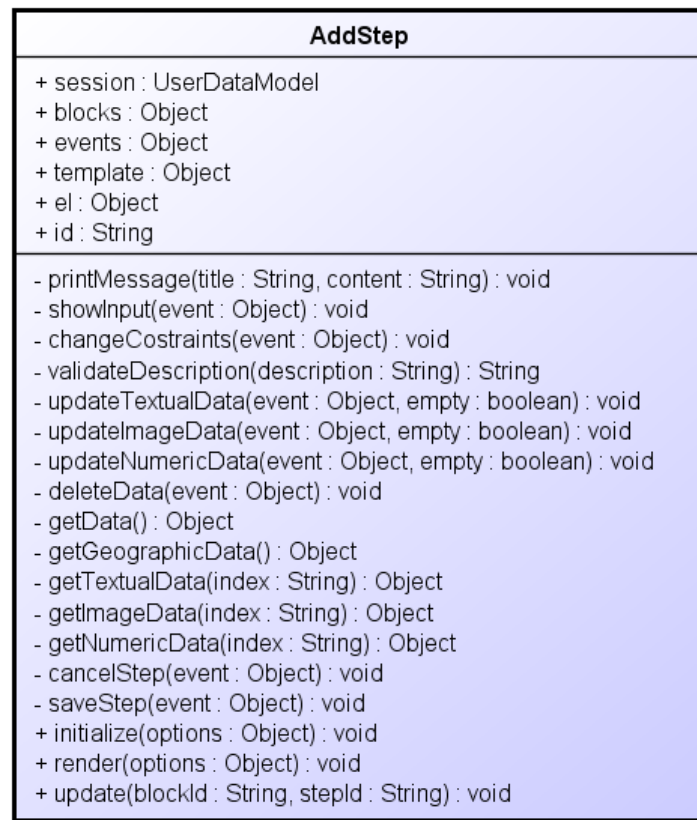
- + `ProcessModel process`:
campo dati di tipo
`com.sirius.sequenziatore.client.model.ProcessModel` che contiene i
dati del processo in definizione;
- + `Object template`:
oggetto ridefinito da `Backbone.View`, che contiene il *template* `HTMLG`
associato alla classe;
- + `Object el`:
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento `HTMLG`
entro cui la classe ascolta eventi generati dagli utenti;
- + `String id`:
campo dati ridefinito da `Backbone.View` contenente l'id della classe;
- + `AddStep addStepLogic`: campo istanza di
`com.sirius.sequenziatore.client.presenter.processowner.AddStep`,
necessario per l'aggiunta di passi in `NewProcess`;
- + `Object blocks` : oggetto contenente i blocchi del processo in creazione.
Un blocco è un'entità per raggruppare logicamente un insieme di passi, e
può essere sequenziale o non ordinato;

- **Metodi:**

- - `getParam(param : String) : String` :
ritorna il parametro get con nome `param` se presente nella url, altrimenti
ritorna false;
- - `printMessage(title : String, content : String) : void`:
apre un popup con titolo `title` e contenuto `content`;
- - `validateDate(dateInput : String, resultDate : Date) : Object`:
controlla la data e ritorna true o un eventuale stringa di descrizione
dell'errore;
- - `validateTime(timeInput : String, date : Date) : Object`:
controlla l'ora e ritorna true o un eventuale stringa di descrizione
dell'errore;
- - `validateDescription(description : String) : String` :
controlla il testo in input e ritorna true o un eventuale stringa di
descrizione dell'errore;
- - `validateImage(imageFile : Object) : String` :
controlla l'immagine in input e ritorna true o un eventuale stringa di
descrizione dell'errore;

- - `saveOptions() : void :`
salva le opzioni sui blocchi impostate dall'utente;
- - `printBlocksHelp(event : Object) : void :`
visualizza il pannello di help relativo all'aggiunta di blocchi;
- - `showInput(event : Object) : void :`
mostra e rende obbligatori i campi dati selezionati;
- - `changeTab(event : Object) : void :`
gestione dell'evento di cambio tab;
- - `addStep(event : Object) : void :`
delega la gestione della creazione di un nuovo passo alla classe `AddStep`;
- - `editStep(event : Object) : void :`
delega la gestione della modifica di un passo alla classe `AddStep`;
- - `ascendBlock(event : Object) : void :`
gestisce lo spostamento di un blocco ad un livello superiore;
- - `descendBlock(event : Object) : void:`
gestisce lo spostamento di un blocco ad un livello inferiore;
- - `addUnorderedBlock(event : Object) : void :`
aggiunge un blocco non ordinato;
- - `addSequentialBlock(event : Object) : void :`
aggiunge un blocco sequenziale;
- - `deleteBlock(event : Object) : void :`
rimuove il blocco selezionato;
- - `removeStep(event : Object) : void :`
rimuove il passo selezionato;
- - `sortBlock(event : Object) : void :`
gestione del cambio dell'ordine dei passi di un blocco sequenziale;
- - `saveDescription(event : Object) : void :`
salva la descrizione del processo in creazione;
- - `cancelDescription(event : Object) : void:`
annulla le modifiche alla descrizione del processo in creazione;
- - `parseBlock(event : Object) : void :`
rimuove i dati temporanei del blocco e imposta i valori di default;
- - `saveProcess(event : Object) : void :`
salva il processo creato;
- - `cancelProcess(event : Object) : void:`
cancella il processo creato;

- + `initialize(options : Object) : void`:
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + `render() : void` :
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;


Figura 14: Diagramma classe *AddStep*

4.1.3.5 AddStep

- **Descrizione:** Classe che ha il compito di gestire la logica di definizione dei passi di un processo;
- **Relazioni con altri componenti:**

La classe comunica con il *template*

`com.sirius.sequenziatore.client.view.processowner.IAddStep` per la realizzazione dell'interfaccia grafica e utilizza la classe

`com.sirius.sequenziatore.client.model.user.StepModel` per salvare i dati del passo in creazione.

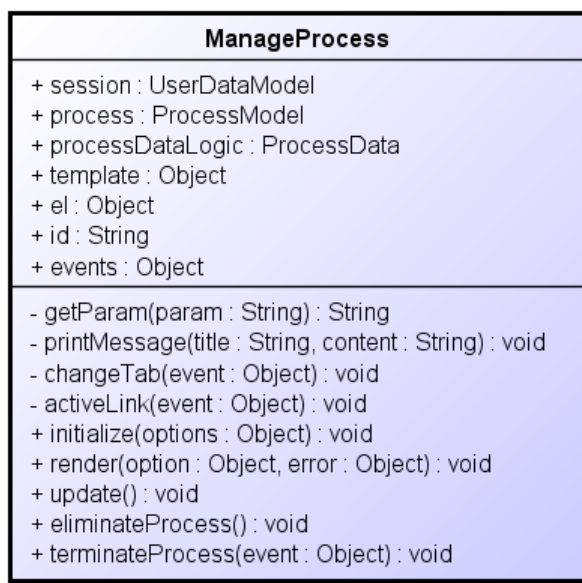
- **Attributi:**

- + `UserDataModel session` :
campo dati di tipo
`com.sirius.sequenziatore.client.model.UserModel` che contiene i dati della sessione;
- + `Object template`:
oggetto ridefinito da `Backbone.View`, che contiene il *template* `HTMLG` associato alla classe;
- + `Object el`:
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento `HTMLG` entro cui la classe ascolta eventi generati dagli utenti;
- + `String id`:
campo dati ridefinito da `Backbone.View` contenente l'id della classe;
- + `Object events`:
oggetto ridefinito da `Backbone.View` che associa ad ogni evento generato dagli utenti nella pagina `HTMLG`, un metodo della classe;
- + `Object blocks`:
array contenente i blocchi del processo in creazione. Un blocco è un'entità per raggruppare logicamente un insieme di passi, e può essere sequenziale o non ordinato;

- **Metodi:**

- – `printMessage(title : String, content : String): void` :
metodo invocato per l'apertura di un popup con titolo `title` e contenuto `content`;
- – `showInput(event : Object):void` :
metodo per visualizzare e rendere obbligatori i campi dati selezionati;
- – `changeConstraints(event : Object):void` :
cambia il vincolo di obbligatorietà dei dati geografici;
- – `validateDescription(description : String): String` :
controlla la validità della descrizione `description` e ritorna una stringa in caso di descrizione non valida;
- – `updateTextualData(event : Object, empty : boolean):void` :
aggiorna la lista dei dati testuali;

- - `updateImageData(event : Object, empty : boolean):void` :
aggiorna la lista dei dati di tipo immagine;
- - `updateNumericData(event : Object, empty : boolean):void` :
aggiorna la lista dei dati numerici;
- - `deleteData(event : Object):void` :
rimuove il dato selezionato;
- - `getData() : Object`:
restituisce i dati e i vincoli inseriti dall'utente relativi al passo in creazione;
- - `getGeographicData() : Object`:
restituisce vincoli geografici inseriti dall'utente;
- - `getTextualData(index : String):Object` :
restituisce la lista dei dati testuali inseriti dall'utente;
- - `getImageData(index : String):Object` :
restituisce la lista dei dati di tipo immagine inseriti dall'utente;
- - `getNumericData(index : String):Object` :
restituisce la lista dei dati numerici inseriti dall'utente;
- - `cancelStep(event : Object):void` :
annulla la modifica/creazione del passo;
- - `saveStep(event : Object):void` :
salva il passo se i dati inseriti dall'utente rispettano i vincoli;
- + `initialize(options : Object):void` :
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + `render(options : Object):void` :
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;
- + `update(blockId : String, stepId : String):void` :
metodo che gestisce la richiesta di creazione e/o modifica di un passo.


Figura 15: Diagramma classe *ManageProcess*

4.1.3.6 ManageProcess

- **Descrizione:** Classe che ha il compito di gestire e accedere alle informazioni relative allo stato dei processi e ai dati inviati dagli utenti. Le operazioni di gestione dello stato comprendono la terminazione e l'eliminazione di un processo;

- **Relazioni con altri componenti:**

La classe comunica con il *template*

`com.sirius.sequenziatore.client.view.processowner.IManageProcess` per la realizzazione dell'interfaccia grafica, e con le classi `ProcessData` e `com.sirius.sequenziatore.client.model.process_owner.ProcessModel` per gestire e ottenere i dati dal *server_G*.

- **Attributi:**

- + `UserDataModel session`:
campo dati di tipo `com.sirius.sequenziatore.client.model.UserDataModel` che permette la gestione della sessione;
- + `ProcessModel process`:
campo dati di tipo `com.sirius.sequenziatore.client.model.process_owner.ProcessModel` che contiene i dati del processo in gestione;
- + `ProcessData processdata`:

campo dati di tipo `ProcessData` necessario per invocare l'omonimo `widgetG`;

– + `Object` `template`:

oggetto ridefinito da `Backbone.View`, che contiene il *template* `HTMLG` associato alla classe;

– + `Object` `el`:

oggetto ridefinito da `Backbone.View` che rappresenta l'elemento `HTMLG` entro cui la classe ascolta eventi generati dagli utenti;

– + `String` `id`:

campo dati ridefinito da `Backbone.View` contenente l'id della classe;

– + `Object` `events`:

oggetto che contiene tutti gli eventi input che vengono gestiti dalla suddetta classe;

• **Metodi:**

– + `initialize(options : Object) : void`:

metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina `HTMLG` associata al componente;

– + `render(option : Object, error : Object) : void`:

metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina `HTMLG` il *template* campo dati della classe;

– + `void` `update()`:

aggiorna i dati della pagina recuperandoli dal server;

– + `eliminateProcess()` : `void` :

permette la gestione della richiesta di eliminazione di un processo terminato dalla lista dei processi gestibili dal process owner;

– + `terminateProcess(event : Object) : void` :

permette gestione della richiesta di terminazione di un processo.

– - `getParam(param : String) : String` :

ritorna il paramentro get con nome param se presente nella url, altrimenti ritorna false;

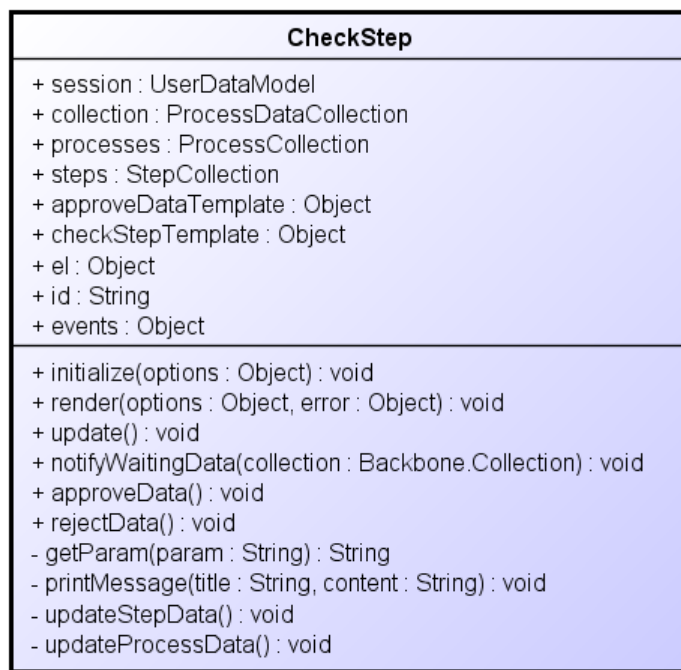
– - `printMessage(title : String, content : String) : void`

apre un popup con titolo title e contenuto content;

– - `changeTab(event : Object) : void` :

gestione dell'evento di cambio tab;

- - `activeLink(event : Object) : void`:
gestione della navigazione tra pagine tramite link contenuti all'interno di un tab;


Figura 16: Diagramma classe *CheckStep*

4.1.3.7 CheckStep

- **Descrizione:** Classe che ha il compito di definire la logica del controllo di un passo che richiede intervento umano per essere approvato;

- **Relazioni con altri componenti:**

La classe comunica con il *template* `com.sirius.sequenziatore.client.view.processowner.ICheckStep` per la realizzazione dell'interfaccia grafica, e con le classi `com.sirius.sequenziatore.client.model.processowner.collection.ProcessDataCollection` e `com.sirius.sequenziatore.client.model.process-owner.ProcessModel` per gestire e ottenere i dati dal *server_G*.

- **Attributi:**

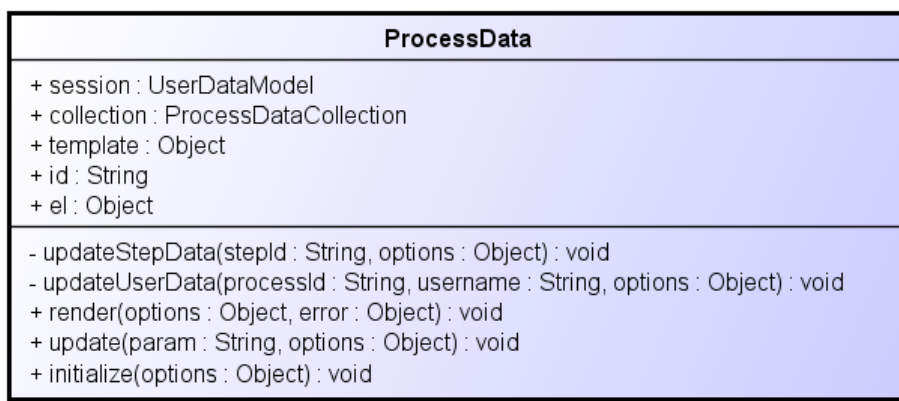
- + `UserDataModel session` : variabile di tipo `com.sirius.sequenziatore.client.model.UserDataModel` necessaria per la gestione della sessione;

- + `ProcessCollection processes` : `com.sirius.sequenziatore.client.model.processowner.collection.ProcessCollection` necessaria per la gestione dei dati riguardanti la collezione di processi accessibili all'utente `process owner`;
- + `StepCollection steps` : `com.sirius.sequenziatore.client.model.processowner.collection.StepCollection` necessari per la gestione dei dati riguardanti la collezione dei passi di un processo
- + `Object approveDataTemplate` : template per l'approvazione dei dati;
- + `Object checkStepTemplate` : template per il controllo dei passi;
- + `ProcessDataCollection collection`:
campo dati di tipo `com.sirius.sequenziatore.client.model.processowner.collection.ProcessDataCollection` che contiene i dati inviati dagli utenti in attesa di approvazione;
- + `Object el`:
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento *HTML_G* entro cui la classe ascolta eventi generati dagli utenti;
- + `String id`:
campo dati ridefinito da `Backbone.View` contenente l'id della classe;

● **Metodi:**

- - `getParam(param : String)` :
ritorna il valore del parametro *param* se presente nella *URL_G*;
- - `printMessage(title : String, content : String)` :
apre un popup con titolo *title* e contenuto *content*;
- - `updateStepData()` :
recupera le informazioni sui passi relativi ai dati che richiedono intervento umano;
- - `updateProcessData()` :
recupera le informazioni sui processi relativi ai dati che richiedono intervento umano;
- + `initialize(options : Object)` :
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTML_G* associata al componente;
- + `render(options : Object, error : Object) : void`
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTML_G* il *template* campo dati della classe;

- `update()` :
aggiorna la collezione dei dati che richiedono approvazione;
- `+ notifyWaitingData(collection : Backbone.Collection):`
permette la gestione dell'evento: `waitingDataNumber` passi richiedono approvazione;
- `+ approveData()` :
permette la gestione della richiesta di approvazione dei dati di un passo;
- `+ rejectData()` :
permette la gestione della richiesta di disapprovazione dei dati di un passo;


Figura 17: Diagramma classe *ProcessData*

4.1.3.8 ProcessData

- **Descrizione:** Classe che ha il compito di gestione della visualizzazione dei dati inviati dagli utenti riguardanti un processo creato

- **Relazioni con altri componenti:**

La classe comunica con il *template* `com.sirius.sequenziatore.client.view.processowner.ICheckStep` per la realizzazione dell'interfaccia grafica, e con le classi `com.sirius.sequenziatore.client.model.processowner.collection.ProcessDataCollection` e `com.sirius.sequenziatore.client.model.process_owner.ProcessModel` per gestire e ottenere i dati dal *server_G*.

- **Attributi:**

- `+ UserDataModel session` : variabile di tipo `com.sirius.sequenziatore.client.model.UserDataModel` necessaria per la gestione della sessione;

- + `Object approveDataTemplate` : template per la renderizzazione del widget;
- + `ProcessDataCollection collection`:
campo dati di tipo `com.sirius.sequenziatore.client.model.process-owner.collection.ProcessDataCollection` che contiene i dati inviati dagli utenti in attesa di approvazione;
- + `Object el`:
oggetto ridefinito da `Backbone.View` che rappresenta l'elemento *HTMLG* entro cui la classe ascolta eventi generati dagli utenti;
- + `String id`:
campo dati ridefinito da `Backbone.View` contenente l'id della classe;

- **Metodi:**

- - `updateStepData(stepId : String, options : Object) : void`
:
aggiorna la pagina con i dati inviati dagli utenti riguardanti il passo `stepId`;
- - `updateUserData(processId : String, username : String, options : Object) : void` :
aggiorna la pagina con i dati inviati dall'utente `username` riguardanti il processo `processId`;
- + `initialize(options : Object) : void` :
metodo ridefinito da `Backbone.View`, invocato alla costruzione di ciascun oggetto della classe, che consente di aggiungere una pagina *HTMLG* associata al componente;
- + `render(options : Object, error : Object) : void`
metodo ridefinito da `Backbone.View`, che consente di aggiungere alla pagina *HTMLG* il *template* campo dati della classe;
- + `update(param : String, options : Object) : void` :
aggiorna i dati della collezione `collection` recuperandoli dal server; di un passo;

5 Specifica della componente Controller

Questa componente è incaricata di gestire la comunicazione con il client restituendo i dati richiesti o lanciando delle opportune eccezioni. Tale componente è composta dalle classi:

- `com.sirius.sequenziatore.server.controller.common.SignUpController`
- `com.sirius.sequenziatore.server.controller.common.LoginController`
- `com.sirius.sequenziatore.server.controller.common.StepInfoController`
- `com.sirius.sequenziatore.server.controller.common.ProcessInfoController`
- `com.sirius.sequenziatore.server.controller.processowner.StepController`
- `com.sirius.sequenziatore.server.controller.processowner.ProcessController`
- `com.sirius.sequenziatore.server.controller.processowner.ApproveStepController`
- `com.sirius.sequenziatore.server.controller.user.UserStepController`
- `com.sirius.sequenziatore.server.controller.user.UserProcessController`
- `com.sirius.sequenziatore.server.controller.user.ReportController`

Nella prossime sezioni verranno trattate in dettaglio le seguenti classi dividendo l'esposizione per *package*, si evidenzia come la voce mappatura base sia l'estensione della mappatura su cui si programma il sistema che sarà `localhost:8080/sequenziatore/`, quindi tutte le mappature base saranno da considerarsi come aggiunte a seguito di `/sequenziatore/` e successivamente le varie varianti dei metodi. Tutte le classi *controller* dovranno essere marcate come `@Controller` per essere riconosciute in modo corretto da *Spring*.

5.0.4 Package `com.sirius.sequenziatore.server.controller.common`

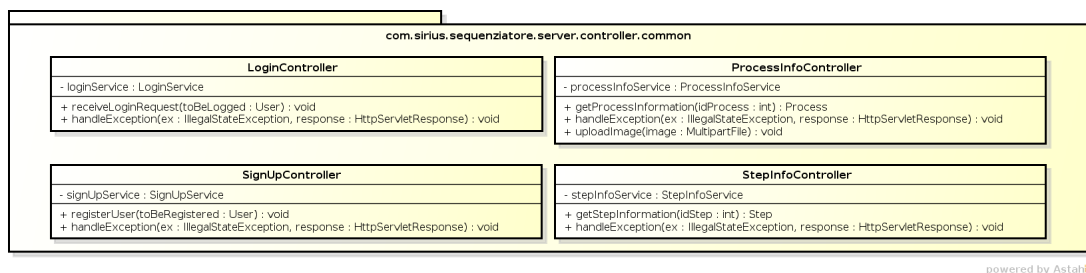


Figura 18: Diagramma package - `com.sirius.sequenziatore.server.controller.common`

All' interno di questa sezione verranno trattate tutte le classi contenute nel package *common*.

5.0.4.1 SignUpController

SignUpController
- signUpService : SignUpService
+ registerUser(toBeRegistered : User) : void + handleException(ex : IllegalStateException, response : HttpServletResponse) : void

Figura 19: Diagramma classe - SignUpController

- **Descrizione:** Questa classe dovrà gestire tutte le richieste di registrazione al sistema, sarà incaricata di inserire i dati nel database e di avvertire il client della riuscita della registrazione.
- **Mappatura base:** */signup*
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.User;`
 - `com.sirius.sequenziatore.server.service.SignUpService;`
- **Attributi:**
 - `-SignUpService signUpService:`
oggetto di tipo `com.sirius.sequenziatore.server.service.SignUpService` a cui viene affidata l' elaborazione della registrazione di un utente;
- **Metodi:**
 - `+void registerUser(User toBeRegistered):`
questo metodo gestirà una richiesta di tipo **POST** e dovrà lanciare un' eccezione di tipo `HttpError` qual' ora ci siano stati problemi nella registrazione;
 - `+void handleException(IllegalStateException,HttpServletResponse response):`
questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 409.

5.0.4.2 LoginController

LoginController
- loginService : LoginService
+ receiveLoginRequest(toBeLogged : User) : void + handleException(ex : IllegalStateException, response : HttpServletResponse) : void

Figura 20: Diagramma classe - LoginController

- **Descrizione:** Questa classe gestirà le richieste di *log in*, delegando l'elaborazione al *service* e poi avvisare il *client* se l'utente è un *process owner*, un utente normale o ci sono stati degli errori, in quest'ultimo caso dovrà lanciare un'eccezione;
- **Mappatura base:** */login*
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.User`;
 - `com.sirius.sequenziatore.server.service.LoginService`
- **Attributi:**
 - `LoginService loginService`:
oggetto di tipo `com.sirius.sequenziatore.server.service.LoginService` a cui viene affidata l'elaborazione della login;
- **Metodi:**
 - `+String receiveLoginRequest(User toBeLogged)`:
questo metodo gestirà un metodo di tipo **POST**, controllerà le credenziali di accesso e dovrà lanciare un'eccezione di tipo `HttpError` qualora ci siano stati problemi nella login;
 - `+void handleException(IllegalStateException,HttpServletResponse response)`:
questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 422.

StepInfoController
- stepInfoService : StepInfoService
+ getStepInformation(idStep : int) : Step + handleException(ex : IllegalStateException, response : HttpServletResponse) : void

Figura 21: Diagramma classe - StepInfoController

5.0.4.3 StepInfoController

- **Descrizione:** Questa classe restituirà lo scheletro, quindi la composizione del passo richiesto;
- **Mappatura base:** `/step/{id}`
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.Step;`
 - `com.sirius.sequenziatore.server.service.StepInfoService;`
- **Metodi:**
 - `+Step getStepInformation(int idStep):`
il metodo gestisce una richiesta di tipo **GET** restituendo la struttura del passo con id uguale all' id fornito dopo averla richiesta al service;
 - `+void handleException(IllegalStateException,HttpServletResponse response):`
questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 404.

ProcessInfoController
- processInfoService : ProcessInfoService
+ getProcessInformation(idProcess : int) : Process + handleException(ex : IllegalStateException, response : HttpServletResponse) : void + uploadImage(image : MultipartFile) : void

Figura 22: Diagramma classe - ProcessInfoController

5.0.4.4 ProcessInfoController

- **Descrizione:** Questa classe dovrà restituire a chi lo richiede un processo dato l' *id* con i suoi dati;
- **Mappatura base:** `/process/{id}`
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.Process;`
 - `com.sirius.sequenziatore.server.service.ProcessInfoService;`
- **Metodi:**

- `+Process getProcessInformation(int idProcess):`
il metodo gestisce una richiesta di tipo **GET** e restituisce la struttura di un processo con l' id processo richiesto;
- `+void handleException(IllegalStateException,HttpServletResponse response):`
questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 422.
- `+boolean uploadImage(MultipartFile image):`
il metodo gestisce una richiesta di tipo **POST** in `/process/{id}/saveimage` e affida al service l' incarico di salvare l' immagine.

5.0.5 Package `com.sirius.sequenziatore.server.presenter.processowner`

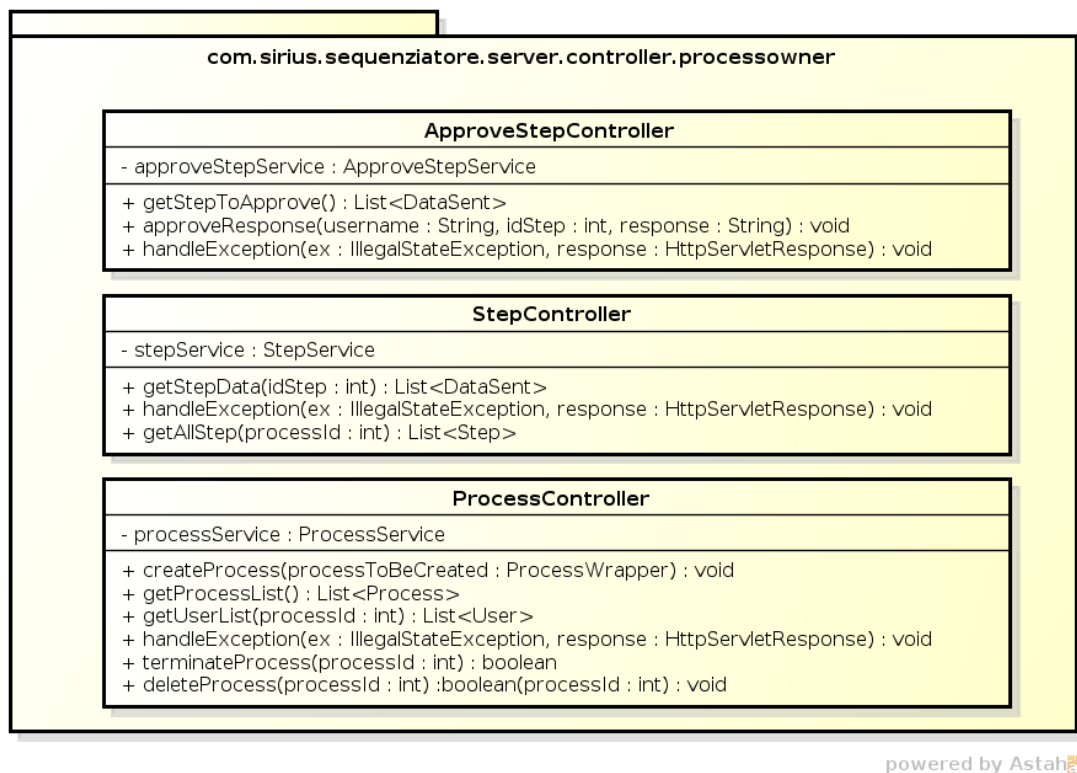


Figura 23: Diagramma package - `com.sirius.sequenziatore.server.presenter.processowner`

5.0.5.1 StepController

StepController
- stepService : StepService
+ getStepData(idStep : int) : List<DataSent> + handleException(ex : IllegalStateException, response : HttpServletResponse) : void

Figura 24: Diagramma classe - StepController

- **Descrizione:** Questa classe dovrà fornire al *process owner* tutti i dati inseriti dagli utenti per un dato passo, quindi dovrà restituire una collezione di dati al process owner il quale potrà visionarli;
- **Mappatura base:** */stepdata/{idstep}/processowner*
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.DataSent`;
 - `com.sirius.sequenziatore.server.model.Step`;
 - `com.sirius.sequenziatore.server.service.StepService`;
- **Metodi:**
 - `+List<DataSent> getStepData(int idStep):`
questo metodo gestisce una richiesta di tipo **GET** che fornisce al *process owner* tutti i dati inviati dagli utenti per un certo passo dopo averli richiesti al service e in caso di errore lancia un' eccezione;
 - `+List<Step> getAllStep(int processId):`
questo metodo riceve una richiesta di tipo **GET**, e ritorna al process owner una lista contenente tutti i passi di un dato processo, in caso di errore lancia un' eccezione.
 - `+void handleException(IllegalStateException,HttpServletResponse response):`
questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 422.

5.0.5.2 ProcessController

ProcessController
- processService : ProcessService
+ createProcess(processToBeCreated : ProcessWrapper) : void + getProcessList() : List<Process> + handleException(ex : IllegalStateException, response : HttpServletResponse) : void

Figura 25: Diagramma classe - ProcessController

- **Descrizione:** Questa classe permetterà la creazione di un processo da parte del *process owner* e sarà adibita a fornire la lista di tutti i processi esistenti nel sistema;
- **Mappatura base:** */process/processowner*
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.Process;`
 - `com.sirius.sequenziatore.server.model.User;`
 - `com.sirius.sequenziatore.server.service.ProcessService;`
 - `com.sirius.sequenziatore.server.controller.utilities.ProcessWrapper;`
- **Metodi:**
 - `+void createProcess(Process processToBeCreated):`
questo metodo gestisce una richiesta di tipo **POST** e incarica il service dell'inserimento del nuovo processo nel database, in caso di errori lancia un'eccezione;
 - `+List<Process> getProcessList():`
questo metodo gestisce una richiesta di tipo **GET** e restituisce al *process owner* una lista di processi che può visualizzare o in caso di errori lancia un'eccezione;
 - `+List<User> getUserList(int processId):`
metodo che riceve una richiesta di tipo **GET** da parte del process owner e restituisce una lista contenente tutto gli utenti che stanno eseguendo un processo.
 - `+boolean terminateProcess(int processId):`
metodo che gestisce una richiesta di tipo **POST** e che affida al service l'incarico di terminare il processo, in caso di errore lancia un'eccezione.
 - `+boolean deleteProcess(int processId):`
metodo che gestisce una richiesta di tipo **POST** e che affida al service l'incarico di eliminare il processo, in caso di errore lancia un'eccezione.

```
- +void handleException(IllegalStateException,HttpServletResponse
response):
```

questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 500.

5.0.5.3 ApproveStepController

ApproveStepController
- approveStepService : ApproveStepService
+ getStepToApprove() : List<DataSent>
+ approveResponse(username : String, idStep : int, response : String) : void
+ handleException(ex : IllegalStateException, response : HttpServletResponse) : void

Figura 26: Diagramma classe - ApproveStepController

- **Descrizione:** Questa classe serve per fornire al *process owner* i dati da approvare e per gestire quali passi siano stati approvati quali no, qualora un passo non venga approvato, verrà rimosso dal *database*;
- **Mappatura base:** */approvedata*
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.DataSent`;
 - `com.sirius.sequenziatore.server.service.ApproveStepService`;
- **Metodi:**
 - `+List<DataSent> getStepToApprove() :`
il metodo gestisce una richiesta di tipo *GET*, e restituirà un oggetto di tipo `List<DataSent>`, contenente tutti i dati che richiedono approvazione, in caso di errore lancia un' eccezione;
 - `+void approveResponse(String username,int idStep,String response) :`
il metodo gestisce una richiesta di tipo *POST*, riceve i dati di un passo che ha subito la moderazione del *process owner* e ne affida al service l' elaborazione, in caso di errore lancia un' eccezione;
 - `+void handleException(IllegalStateException,HttpServletResponse response):`
questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 422.

5.0.6 Package com.sirius.sequenziatore.server.presenter.user

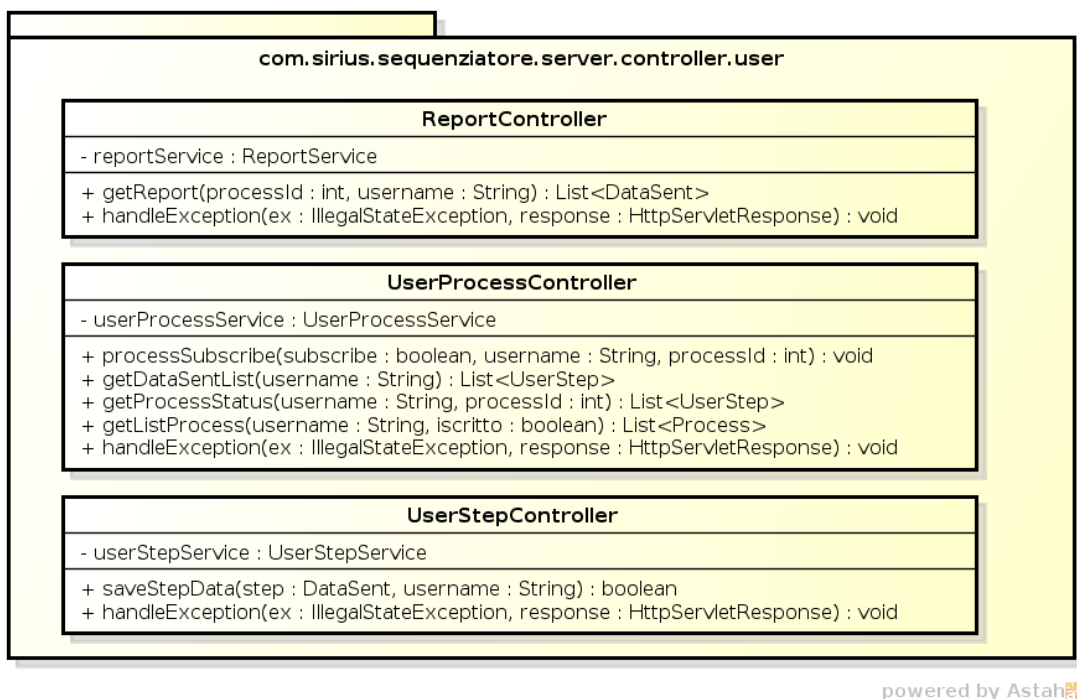


Figura 27: Diagramma package - com.sirius.sequenziatore.server.controller.user

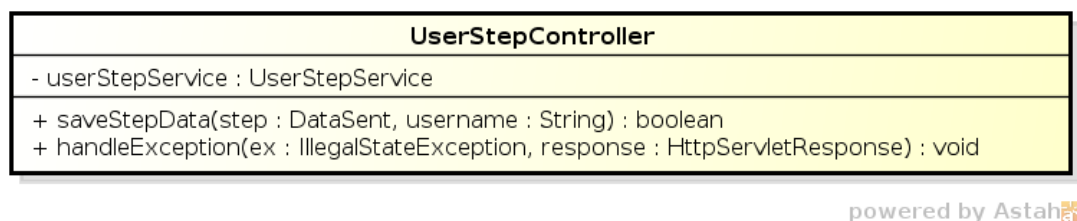


Figura 28: Diagramma classe - UserStepController

5.0.6.1 UserStepController

- **Descrizione:** Questa classe gestisce la ricezione dei dati di un passo inviati da un utente tramite una richiesta di tipo *POST*, tale passo dovrà essere inserito nel database, ponendo attenzione se è un passo che richiede approvazione o meno;
- **Mappatura base:** */stepdata/user*
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - com.sirius.sequenziatore.server.model.DataSent;
 - com.sirius.sequenziatore.server.service.UserStepService;

- **Attributi:**

- `-UserStepService userStepService.`

- **Metodi:**

- `+boolean saveStepData(DataSent step,String username):`
questo metodo gestisce una richiesta **POST** da un utente, riceve i dati inerenti a un passo e affida al service l'incarico di salvare tali dati, in caso di errore lancia un'eccezione;
 - `+void handleException(IllegalStateException,HttpServletResponse response):`
questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 409.

5.0.6.2 UserProcessController

UserProcessController
- userProcessService : UserProcessService
+ processSubscribe(subscribe : boolean, username : String, processId : int) : void + getDataSentList(username : String) : List<UserStep> + getProcessStatus(username : String, processId : int) : List<UserStep> + getListProcess(username : String, iscritto : boolean) : List<Process> + handleException(ex : IllegalStateException, response : HttpServletResponse) : void

Figura 29: Diagramma classe - UserProcessController

- **Descrizione:** Questa classe permette all'utente varie operazioni, innanzitutto l'iscrizione ad un processo, poi restituisce il passo a cui è arrivato e il suo stato per tale processo e infine fornisce una lista di processi con tutti i processi a cui si può iscrivere e i processi per i quali può chiedere di fare il *report*;
- **Mappatura base:** `/user/{username}`
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.Process;`
 - `com.sirius.sequenziatore.server.model.UserStep;`
 - `com.sirius.sequenziatore.server.model.ProcessDao;`
 - `com.sirius.sequenziatore.server.model.StepDao;`
 - `com.sirius.sequenziatore.server.service.UserProcessService;`
- **Attributi:**

– `-UserService userService;`

• Metodi:

- `+boolean processSubscribe(boolean subscribe,String username,int processId):`
questo metodo mappa su `/subscribe/{processid}` e gestisce una richiesta di tipo **POST** incaricando il service di iscrivere l'utente al processo voluto;
- `+List<UserStep> getProcessStatus(String username,int processId):`
questo metodo mappa su `/subscribe/{processid}` e gestisce una richiesta **GET** che restituisce all'utente il proprio status per tale processo, restituendo il passo o i passi che può eseguire e quanti passi ha completato del processo;
- `+List<Process> getListProcess(String username,boolean iscritto):`
questo metodo mappa su `/processlist` e gestisce una richiesta di tipo **GET** andando e restituire una lista di processi che contiene tutti i processi a cui è iscritto e quelli a cui si può iscrivere;
- `+List<UserStep> getDataSentList(String username):`
questo metodo gestisce una richiesta di tipo **GET** e restituisce all'utente il proprio status per tale processo.
- `+void handleException(IllegalStateException,HttpServletResponse response):`
questo metodo è un gestore delle eccezioni e sarà incaricato di lanciare al client un errore 409.

5.0.6.3 ReportController

ReportController
- reportService : ReportService
+ getReport(processId : int, username : String) : List<DataSent> + handleException(ex : IllegalStateException, response : HttpServletResponse) : void

Figura 30: Diagramma classe - ReportController

- **Descrizione:** Questa classe fornirà al client tutti i dati necessari per creare il report di un utente per un certo processo;
- **Mappatura base:** `/report/{username}/{processid}`
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:

- `com.sirius.sequenziatore.server.model.DataSent;`
- `com.sirius.sequenziatore.server.service.ReportService;`

- **Attributi:**

- `-ReportService reportService.`

- **Metodi:**

- `+List<DataSent> getReportData(DataSent step,String username):`
questo metodo gestisce una richiesta di tipo **GET** e fornirà tutti i dati inseriti da un utente per un certo processo;

6 Specifica della componente Service

Questa componente è incaricata elaborare le richieste arrivate dai controller restituendo i dati richiesti e quando necessario interroga la componente model per ottenere i dati dal database. Tale componente è composta dalle classi:

- `com.sirius.sequenziatore.server.service.SignUpService`
- `com.sirius.sequenziatore.server.service.LoginService`
- `com.sirius.sequenziatore.server.service.StepInfoService`
- `com.sirius.sequenziatore.server.service.ProcessInfoService`
- `com.sirius.sequenziatore.server.service.StepService`
- `com.sirius.sequenziatore.server.service.ProcessService`
- `com.sirius.sequenziatore.server.service.ApproveStepService`
- `com.sirius.sequenziatore.server.service.UserStepService`
- `com.sirius.sequenziatore.server.service.UserProcessService`
- `com.sirius.sequenziatore.server.service.ReportService`

Qui di seguito verranno ora analizzate le singole classi di tale componente, tali classi sono annotate con `@Service`, in modo da essere riconosciute in modo corretto da Spring.

6.1 Package com.sirius.sequenziatore.server.service

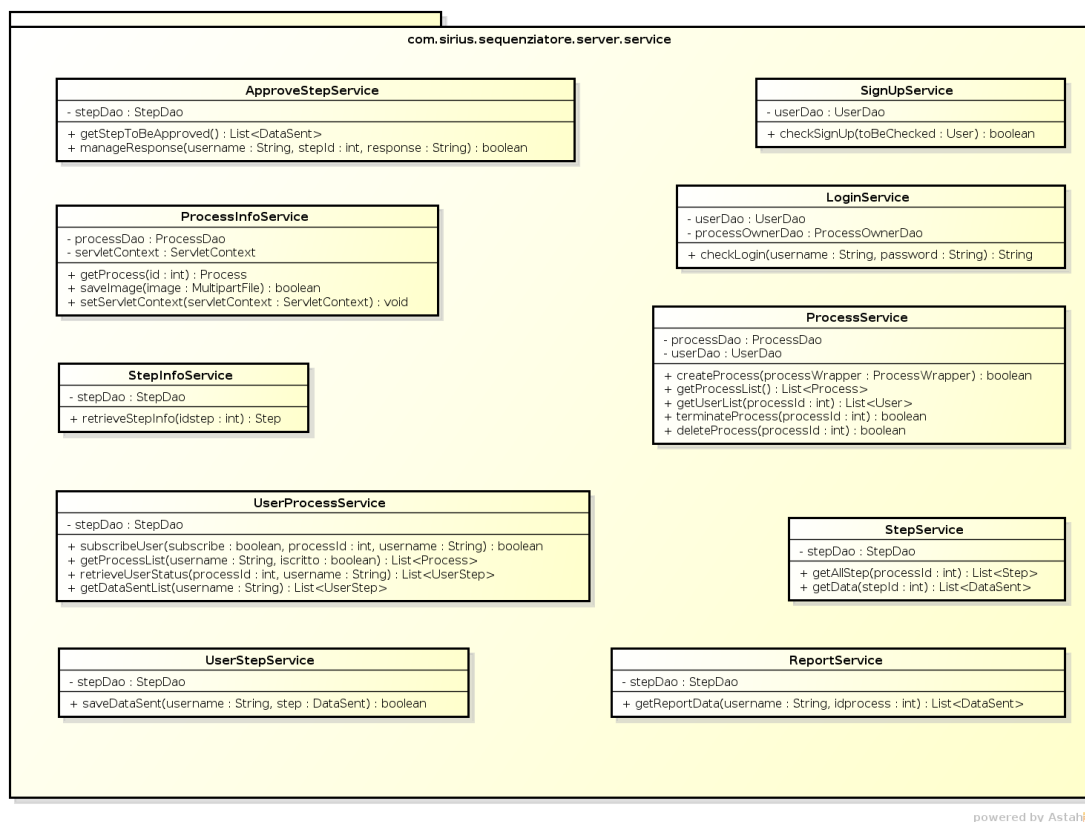


Figura 31: Diagramma package - com.sirius.sequenziatore.server.service

6.1.0.4 SignUpService

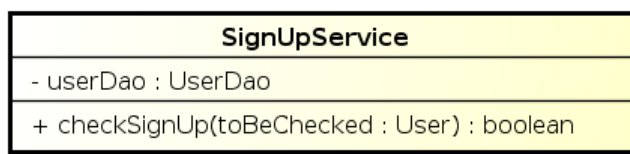


Figura 32: Diagramma classe - SignUpService

- **Descrizione:** Questa classe dovrà elaborare tutte le richieste di registrazione al sistema ricevute dal client, sarà incaricata di inserire i dati nel database.
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - com.sirius.sequenziatore.server.model.UserDao;
 - com.sirius.sequenziatore.server.model.User;
- **Attributi:**

- UserDao userDao:
oggetto usato per inserire il nuovo utente nel database;

- **Metodi:**

- +boolean checkSignUp(User toBeChecked):
questo metodo deve inserire nel database il nuovo utente ricevuto, e ritornare l'esito di tale operazione al controller;

6.1.0.5 ApproveStepService

ApproveStepService
- stepDao : StepDao
+ getStepToBeApproved() : List<DataSent> + manageResponse(username : String, stepId : int, response : String) : boolean

Figura 33: Diagramma classe - ApproveStepService

- **Descrizione:** Questa classe dovrà elaborare tutti gli esiti della moderazione dei passi del processowner quindi accettare o rifiutare i passi in attesa di approvazione.

- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:

- com.sirius.sequenziatore.server.model.StepDao;
- com.sirius.sequenziatore.server.model.DataSent;

- **Attributi:**

- StepDao stepDao:
oggetto usato per modificare i passi nel database secondo l'esito del process owner;

- **Metodi:**

- +List<DataSent> getStepToBeApproved():
questo metodo ritorna una lista di passi che sono in attesa di approvazione, dopo averla ottenuta dal database tramite stepDao;
- +boolean manageResponse(String username,int stepId,String response):
questo metodo permette di gestire l'esito della moderazione del process owner per un dato passo di un dato utente, in **response** sarà contenuto tale esito che sarà *APPROVED* o *REJECTED* e poi andrà a modificare il passo nel database tramite stepDao;

6.1.0.6 ProcessInfoService

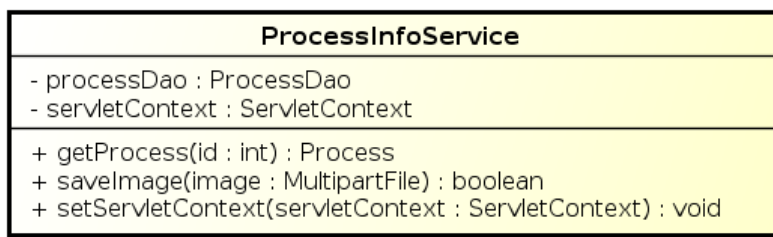


Figura 34: Diagramma classe - ProcessInfoService

- Descrizione:** Questa classe dovrà elaborare le varie richieste di tutti gli utenti per quanto riguarda le informazioni generali di un processo, quindi la sua struttura e permette il salvataggio di immagini;
- Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.ProcessDao;`
 - `com.sirius.sequenziatore.server.model.Process;`
- Attributi:**
 - `-ProcessDao processDao:`
oggetto usato per modificare i passi nel database secondo l'esito del process owner;
- Metodi:**
 - `+Process getProcess(int id):`
questo metodo ritorna la struttura del processo richiesto;
 - `+boolean saveImage(MultipartFile image):`
questo metodo permette il salvataggio delle immagini del processo;
 - `+void setServletContext(ServletContext servletContext):` questo metodo serve per settare la servletContext;

6.1.0.7 StepInfoService



Figura 35: Diagramma classe - StepInfoService

- **Descrizione:** Questa classe dovrà elaborare le varie richieste di tutti gli utenti per quanto riguarda le informazioni generali di un passo, quindi la sua struttura;
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.StepDao;`
 - `com.sirius.sequenziatore.server.model.Step;`
- **Attributi:**
 - `-StepDao stepDao:`
oggetto usato per ottenere i dati di un passo;
- **Metodi:**
 - `+Step retrieveStep(int idStep):`
questo metodo ritorna la struttura del passo richiesto;

6.1.0.8 LoginService



Figura 36: Diagramma classe - LoginService

- **Descrizione:** Questa classe elaborerà le richieste di login
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.UserDao;`
 - `com.sirius.sequenziatore.server.model.ProcessOwnerDao;`
- **Attributi:**
 - `-UserDao userDao:`
oggetto usato per ottenere i dati di un utente;
 - `-ProcessOwnerDao processOwnerDao:`
oggetto usato per ottenere i dati di un processowner;
- **Metodi:**
 - `+String checkLogin(String username,String password):`
questo metodo controlla i dati di login e ritorna l' esito al controller;

6.1.0.9 ProcessService

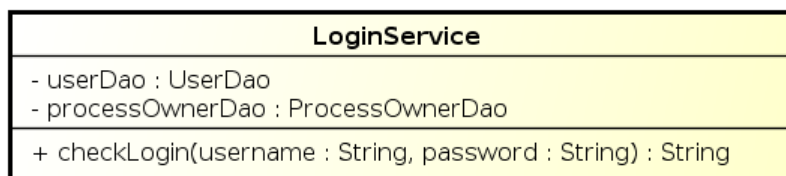


Figura 37: Diagramma classe - ProcessService

- **Descrizione:** Questa classe elaborerà le richieste riguardanti i processi derivanti dal processowner;
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.processDao;`
 - `com.sirius.sequenziatore.server.model.UserDao;`
 - `com.sirius.sequenziatore.server.model.Process;`
 - `com.sirius.sequenziatore.server.model.User;`
- **Attributi:**
 - `-UserDao userDao:`
oggetto usato per ottenere i dati di un utente;
 - `-ProcessDao processDao:`
oggetto usato per ottenere i dati di un processo;
- **Metodi:**
 - `+boolean createProcess(Process process):`
questo metodo permette il salvataggio nel nuovo processo creato;
 - `+List<Process> getProcessList():`
questo metodo permette di recuperare la lista di tutti i processi non eliminati del sistema
 - `+List<User> getUserList(int processId):`
questo metodo ritorna la lista di utenti iscritta al processo;
 - `+boolean terminateProcess(int processId):`
questo metodo permette la terminazione di un processo;
 - `+boolean deleteProcess(int processId):`
questo metodo permette l'eliminazione di un processo;

6.1.0.10 UserProcessService

UserProcessService
- stepDao : StepDao
+ subscribeUser(subscribe : boolean, processId : int, username : String) : boolean + getProcessList(username : String, iscritto : boolean) : List<Process> + retrieveUserStatus(processId : int, username : String) : List<UserStep> + getDataSentList(username : String) : List<UserStep>

Figura 38: Diagramma classe - UserProcessService

- **Descrizione:** Questa classe elaborerà le richieste degli utenti per quanto riguarda i processi come iscrizione o ottenere i dati o lo status di un processo;

- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:

- com.sirius.sequenziatore.server.model.StepDao;
- com.sirius.sequenziatore.server.model.Process;
- com.sirius.sequenziatore.server.model.UserStep;

- **Attributi:**

- -StepDao stepDao:
oggetto usato per ottenere i dati di un passo;

- **Metodi:**

- +boolean subscribeUser(boolean subscribe,int processId,String username):
questo metodo permette ad un utente di iscriversi a un processo;
- +List<Process> getProcessList(String username,boolean iscritto):
questo metodo ritorna la lista di processi utile all' utente;
- +List<UserStep> retrieveUserStatus(int processId,String username):
questo metodo ritorna la lista di di UserStep quindi lo status dell utente per il processo richiesto;
- +List<UserStep> getDataSentList(String username): questo metodo ritorna la lista di di UserStep quindi lo status dell utente per i vari processi a cui è iscritto serve ad identificare i processi a cui è iscritto;

6.1.0.11 UserStepService

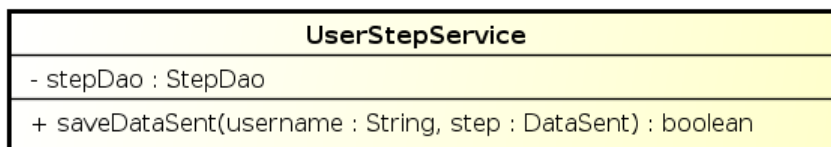


Figura 39: Diagramma classe - UserStepService

- **Descrizione:** Questa classe elaborerà le richieste degli utenti per quanto riguarda il salvataggio di un dato inviato;
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.StepDao`;
 - `com.sirius.sequenziatore.server.model.DataSent`;
- **Attributi:**
 - `-StepDao stepDao`:
oggetto usato per ottenere i dati di un passo;
- **Metodi:**
 - `+boolean saveDataSent(String username,DataSent step)`:
questo metodo permette ad un utente di salvare i dati di un dato passo;

6.1.0.12 StepService

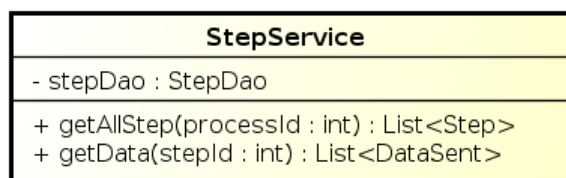


Figura 40: Diagramma classe - StepService

- **Descrizione:** Questa classe elaborerà le richieste del processowner per ottenere i dati dei passi o tutti i passi di un processo;
- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:
 - `com.sirius.sequenziatore.server.model.StepDao`;
 - `com.sirius.sequenziatore.server.model.DataSent`;
 - `com.sirius.sequenziatore.server.model.Step`;
- **Attributi:**

- `-StepDao stepDao:`
oggetto usato per ottenere i dati di un passo;

- **Metodi:**

- `+List<Step> getAllStep(int processId):`
questo metodo ritorna la lista di tutti i passi di un processo;
- `+List<DataSent> getData(int stepId):`
questo metodo ritorna la lista tutti i dati inviati dagli utenti per un passo;

6.1.0.13 ReportService



Figura 41: Diagramma classe - ReportService

- **Descrizione:** Questa classe elaborerà le richieste degli utenti per ottenere tutti i dati inviati di un dato processo;

- **Relazioni con altri componenti:** La classe utilizzerà le seguenti classi:

- `com.sirius.sequenziatore.server.model.StepDao;`
- `com.sirius.sequenziatore.server.model.DataSent;`

- **Attributi:**

- `-StepDao stepDao:`
oggetto usato per ottenere i dati di un passo;

- **Metodi:**

- `+List<DataSent> getReportData(String username,int idprocess):`
questo metodo ritorna la lista di tutti i dati inviati da un utente per un processo;

7 Specifica della componente model

Questa componente consente di rappresentare i dati e gestire la loro persistenza, e viene suddivisa in due parti: *client* e *server*.

7.1 Client

Il *model* lato *client* consente di gestire i dati dell'applicazione e la comunicazione con il *server_G*.

La componente è formata dalle seguenti *classi*:

- [com.sirius.sequenziatore.client.model.UserDataModel](#);
- [com.sirius.sequenziatore.client.model.ProcessModel](#);
- [com.sirius.sequenziatore.client.model.ProcessDataModel](#);
- [com.sirius.sequenziatore.client.model.StepModel](#);
- [com.sirius.sequenziatore.client.model.collection.ProcessCollection](#);
- [com.sirius.sequenziatore.client.model.ProcessDataCollection](#);
- [com.sirius.sequenziatore.client.model.collection.StepCollection](#).

7.1.1 Package com.sirius.sequenziatore.client.model

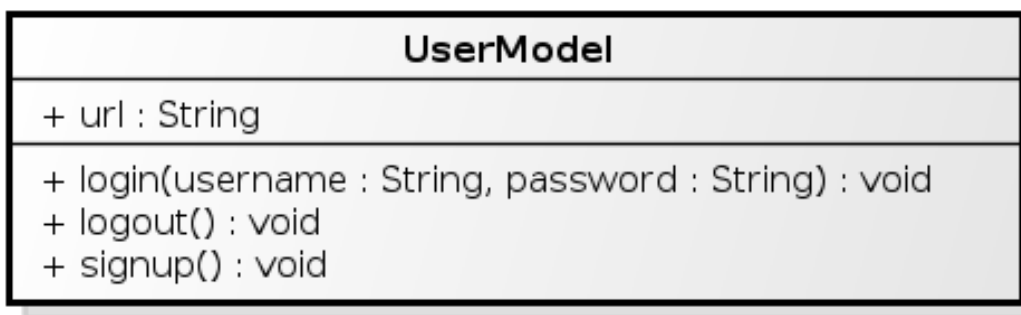


Figura 42: Diagramma classe *UserModel*

7.1.1.1 UserModel

- **Descrizione:** Classe che permette di gestire i dati di una sessione di un utente autenticato o di un *Process Owner_G*;
- **Attributi:**

- + `String url`:
campo dati di ridefinito da `Backbone.Model` che contiene l'indirizzo `urlG` per comunicare con il `serverG`;

- **Metodi:**

- + `void login(String username, String password)`:
delega al server il controllo delle credenziali e, al completamento della richiesta, salva i dati di sessione in caso di successo;
- + `void logout()`:
cancella di dati di sessione dell'utente;
- + `void signup()`:
effettua una richiesta di registrazione al `serverG` inviando i dati della classe.

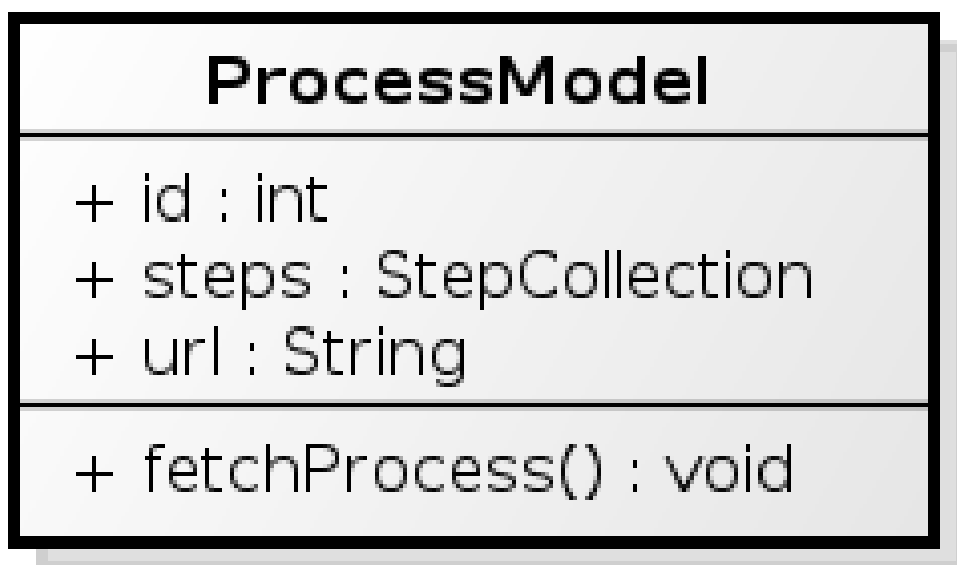


Figura 43: Diagramma classe *ProcessModel*

7.1.1.2 ProcessModel

- **Descrizione:** Classe che permette di gestire i dati di un processo, e di salvarli o recuperarli dal `serverG`;
- **Relazioni con altri componenti:**
La classe contiene un oggetto di tipo `com.sirius.sequenziatore.client.model.collection.StepCollection`.

- **Attributi:**

- + `int id`:
campo dati ridefinito da `Backbone.model` che rappresenta l'identificatore del processo;
- + `StepCollection steps`:
campo dati di tipo `com.sirius.sequenziatore.client.model.collection.StepCollection` che contiene la collezione dei passi del processo;
- + `String url`:
campo dati di ridefinito da `Backbone.Model` che contiene l'indirizzo `urlG` per comunicare con il `serverG`;

- **Metodi:**

- + `void fetchProcess()`:
recupera dal `serverG` i dati del processo, e i dati dei passi che assegna alla collezione `steps`, sincronizzando le operazioni.

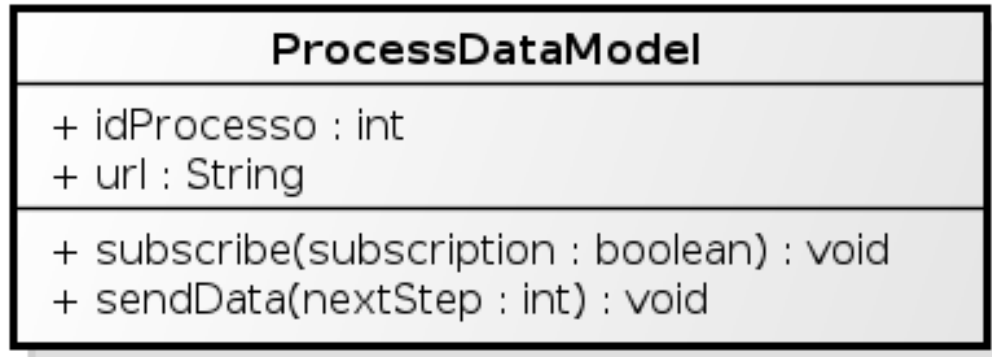


Figura 44: Diagramma classe *ProcessDataModel*

7.1.1.3 ProcessDataModel

- **Descrizione:** Classe che permette di gestire i dati inviati da un utente relativi ad un processo, e di salvarli o recuperarli dal `serverG`;
- **Attributi:**
 - + `int idProcesso`:
rappresenta l'identificatore del processo a cui i dati si riferiscono;

- + `String url`:
campo dati di ridefinito da `Backbone.Model` che contiene l'indirizzo `urlG` per comunicare con il `serverG`;

- **Metodi:**

- + `void subscribe(bool subscription)`:
effettua una richiesta di iscrizione o disiscrizione al `serverG` a seconda del valore del parametro `subscription`, riguardante il processo con id `idProcesso`;
- + `void sendData(int nextStep)`:
invia al `serverG` i dati della classe e l'id del prossimo passo da eseguire, che identifica una condizione del processo con id `idProcesso`.

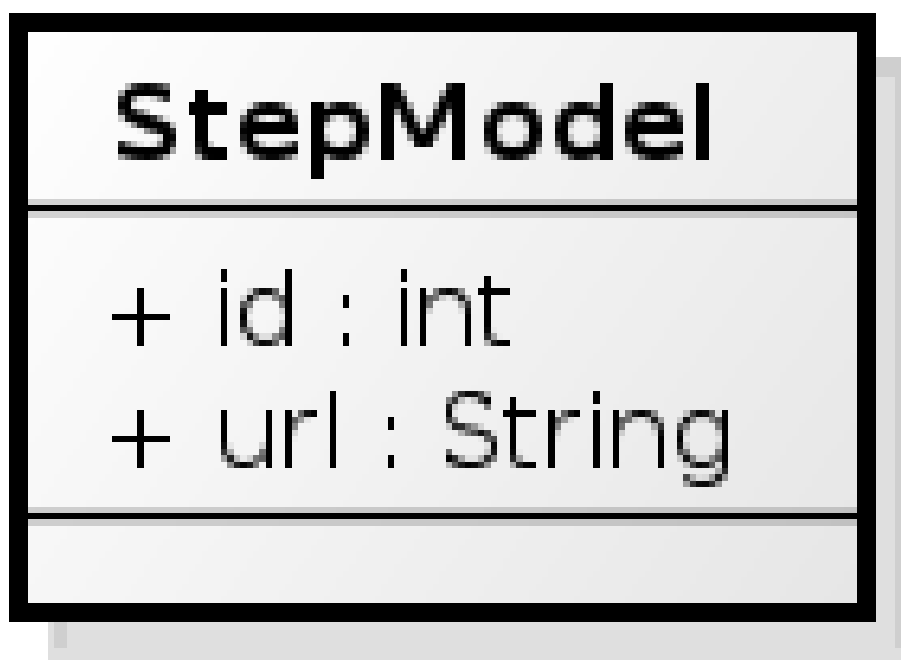


Figura 45: Diagramma classe *StepModel*

7.1.1.4 StepModel

- **Descrizione:** Classe che permette di gestire i dati di un passo di un processo, e di salvarli o recuperarli dal `serverG`;

- **Attributi:**

- + `int id`:
campo dati ridefinito da `Backbone.model` che rappresenta l'identificatore del passo;
- + `String url`:
campo dati di ridefinito da `Backbone.Model` che contiene l'indirizzo `urlG` per comunicare con il `serverG`;

7.1.2 Package `com.sirius.sequenziatore.client.model.collection`

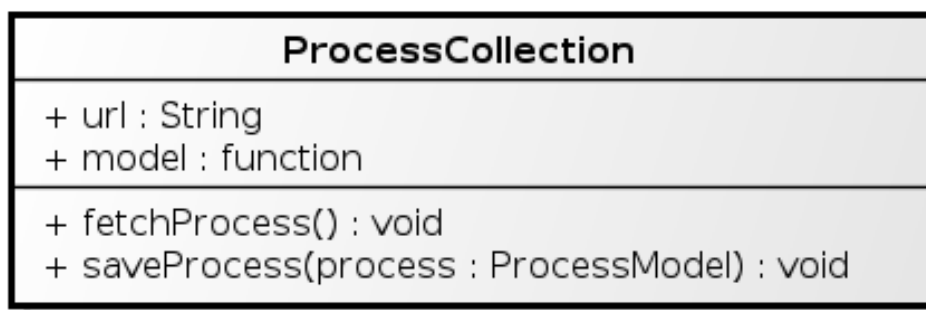


Figura 46: Diagramma classe *ProcessCollection*

7.1.2.1 ProcessCollection

- **Descrizione:** Classe che permette di gestire un insieme di dati inviati da un utente relativi ad un processo;

- **Relazioni con altri componenti:**

La classe definisce una collezione di
`com.sirius.sequenziatore.client.model.ProcessDataModel`.

- **Attributi:**

- + `String url`:
campo dati di ridefinito da `Backbone.Collection` che contiene l'indirizzo `urlG` per comunicare con il `serverG`;
- + `function model`:
campo dati di ridefinito da `Backbone.Collection` che contiene la definizione della classe
`com.sirius.sequenziatore.client.model.ProcessDataModel`;

- **Metodi:**

- + void `fetchProcesses()`:
richiede al server la lista dei processi a cui l'utente identificato dai dati di sessione può accedere;
- + void `saveProcess(ProcessModel process)`:
aggiunge il processo `process` alla collezione dei processi nel `serveG`.

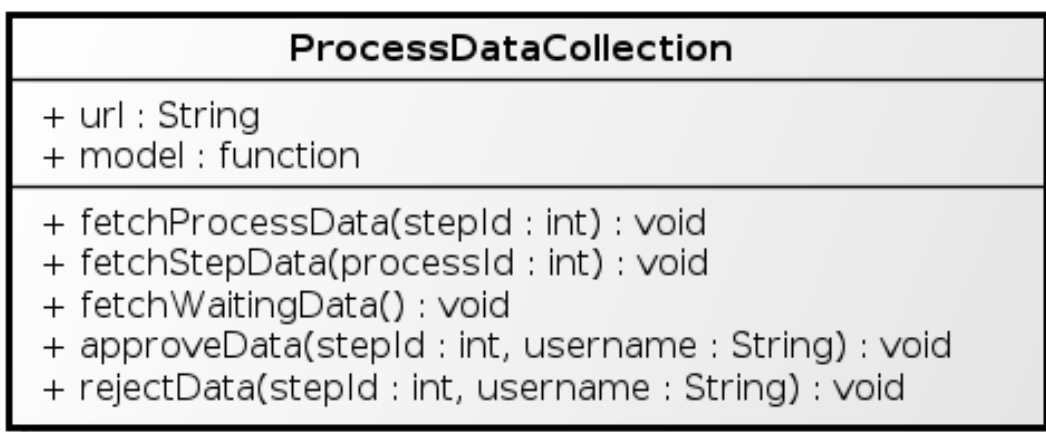


Figura 47: Diagramma classe *ProcessDataCollection*

7.1.2.2 ProcessDataCollection

- **Descrizione:** Classe che permette di gestire un insieme di dati inviati dagli utenti;

- **Relazioni con altri componenti:**

La classe definisce una collezione di
`com.sirius.sequenziatore.client.model.ProcessDataModel`.

- **Attributi:**

- + String `url`:
campo dati di ridefinito da `Backbone.Collection` che contiene l'indirizzo `urlG` per comunicare con il `serverG`;
- + function `model`:
campo dati di ridefinito da `Backbone.Collection` che contiene la definizione della classe
`com.sirius.sequenziatore.client.model.ProcessDataModel`;

- **Metodi:**

- + void `fetchProcessData(int stepId)`:
richiede al *server_G* la lista dei dati inviati riguardanti il passo con id `stepId`, ai quali l'utente identificato dai dati di sessione può accedere;
- + void `fetchStepData(int processId)`:
richiede al *server_G* la lista dei dati inviati riguardanti il processo con id `processId`, ai quali l'utente identificato dai dati di sessione può accedere;
- + void `fetchWaitingData()`:
richiede al *server_G* la lista dei dati inviati che richiedono controllo umano;
- + void `approveData(int stepId, String username)`:
invia al *server_G* la richiesta di approvazione dei dati riguardanti il passo con id `stepId` e l'utente con username `username`.
- + void `rejectData(int stepId, String username)`:
invia al *server_G* l'esito negativo del controllo dei dati riguardanti il passo con id `stepId` e l'utente con username `username`.

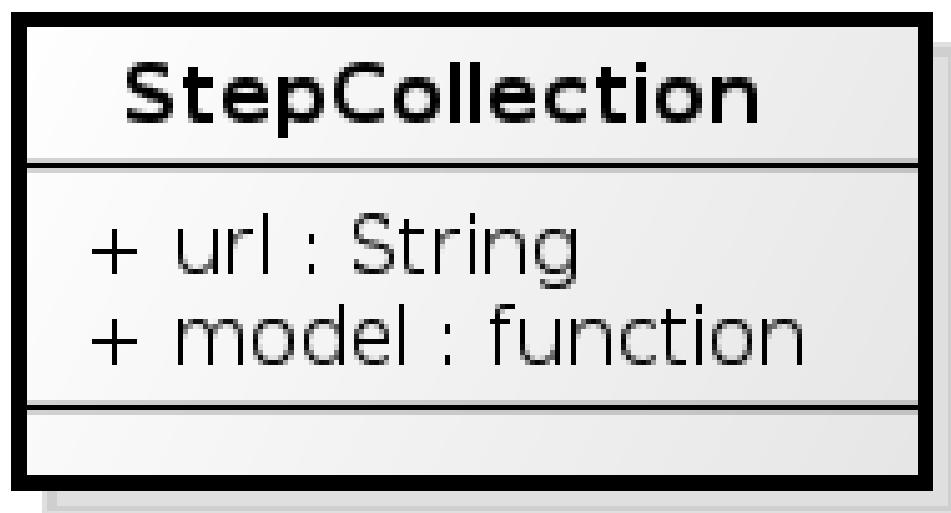


Figura 48: Diagramma classe *StepCollection*

7.1.2.3 StepCollection

- **Descrizione:** Classe che permette di gestire un insieme di passi di un processo;

- **Relazioni con altri componenti:**

La classe definisce una collezione di
`com.sirius.sequenziatore.client.model.StepModel`.

- **Attributi:**

- + `String url`:
campo dati di ridefinito da `Backbone.Collection` che contiene l'indirizzo *url_G* per comunicare con il *server_G*;
- + `function model`:
campo dati di ridefinito da `Backbone.Collection` che contiene la definizione della classe
`com.sirius.sequenziatore.client.model.StepModel`;

7.2 Server

Il *model* lato *server* gestisce la persistenza dei dati all'interno del *database* consentendo interrogazione, inserimento, cancellazione e aggiornamento.

La componente è formata dalle seguenti *classi*:

- `com.sirius.sequenziatore.server.model.IDataAccessObject;`
- `com.sirius.sequenziatore.server.model.ITransferObject;`
- `com.sirius.sequenziatore.server.model.UserDao;`
- `com.sirius.sequenziatore.server.model.ProcessDao;`
- `com.sirius.sequenziatore.server.model.ProcessOwnerDao;`
- `com.sirius.sequenziatore.server.model.StepDao;`
- `com.sirius.sequenziatore.server.model.User;`
- `com.sirius.sequenziatore.server.model.Process;`
- `com.sirius.sequenziatore.server.model.Step;`
- `com.sirius.sequenziatore.server.model.DataSent;`
- `com.sirius.sequenziatore.server.model.IDataValue;`
- `com.sirius.sequenziatore.server.model.TextualValue;`
- `com.sirius.sequenziatore.server.model.NumericValue;`
- `com.sirius.sequenziatore.server.model.ImageValue;`
- `com.sirius.sequenziatore.server.model.GeographicValue;`
- `com.sirius.sequenziatore.server.model.UserStep;`
- `com.sirius.sequenziatore.server.model.ProcessOwner;`

7.2.1 Package `com.sirius.sequenziatore.client.model`

7.2.1.1 `IDataAccessObject`

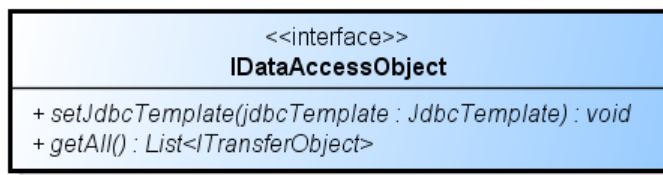


Figura 49: Diagramma interfaccia `IDataAccessObject`

- **Descrizione:** Interfaccia che permette di gestire la comunicazione e l'interrogazione con il *database*.
- **Metodi:**
 - + void setJdbcTemplate(JdbcTemplate jdbcTemplate):
Imposta i parametri per l'accesso alla sorgente dei dati;
 - + ITransferObject getAll():
Ritorna tutti i dati di competenza della classe che estende questa interfaccia.

7.2.1.2 ITransferObject

- **Descrizione:** Interfaccia realizzata dai tipi che modellano i dati del *database*.

7.2.1.3 UserDao

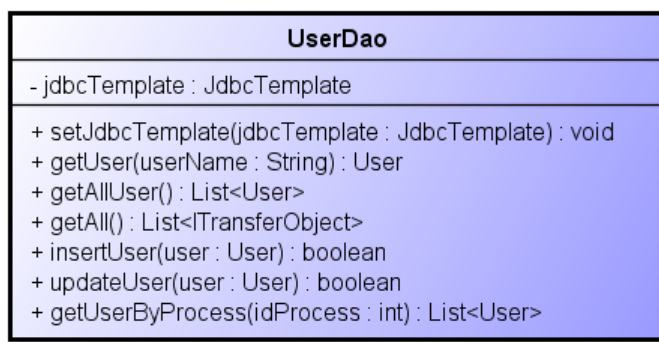


Figura 50: Diagramma classe UserDao

- **Descrizione:** Classe che si occupa delle interrogazioni del *database* relative agli utenti del sistema.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:
 - com.sirius.sequenziatore.server.model.IDataAccessObject.

La classe invoca i metodi della classe:

- com.sirius.sequenziatore.server.model.User.

- **Attributi:**
 - JdbcTemplate jdbcTemplate:
Oggetto che fornisce l'accesso alla sorgente dei dati;

• Metodi:

- + User getUser(String userName):
Ritorna l'utente con il nome utente specificato;
- + List<User> getAllUser():
Ritorna tutti gli utenti;
- + boolean insertUser(User user) :
Aggiunge l'utente passato come parametro;
- + public boolean updateUser(User user) :
Aggiorna i dati dell'utente con il nome utente corrispondente a quello dell'utente passato, con i dati dell'utente passato.

7.2.1.4 ProcessDao

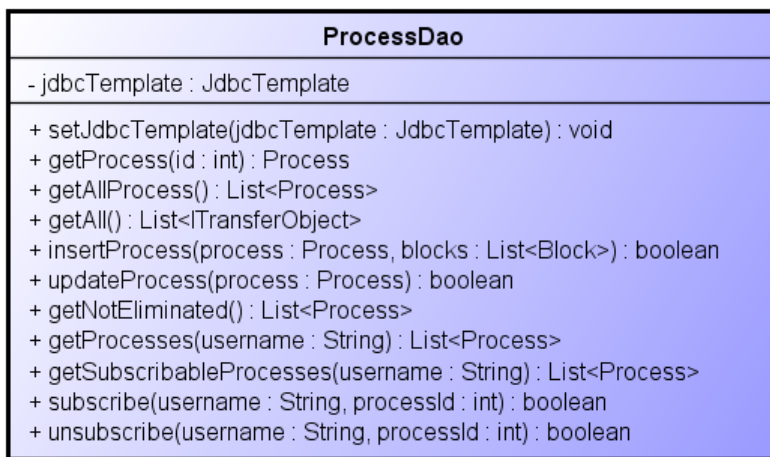


Figura 51: Diagramma classe ProcessDao

- **Descrizione:** Classe che si occupa delle interrogazioni del *database* relative ai processi.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:

- com.sirius.sequenziatore.server.model.IDataAccessObject.

La classe invoca i metodi della classe:

- com.sirius.sequenziatore.server.model.Process.

• Attributi:

- - JdbcTemplate jdbcTemplate:
Oggetto che fornisce l'accesso alla sorgente dei dati;

• Metodi:

- + Process getProcess(int id):
Ritorna il processo con l'id specificato;
- + List<Process> getAllProcess():
Ritorna tutti i processi;
- + boolean insertProcess(Process process) :
Aggiunge il processo passato come parametro;
- + public boolean updateProcess(Process process) :
Aggiorna i dati del processo con lo stesso id di quello del processo passato,
con i dati del processo passato.

7.2.1.5 ProcessOwnerDao

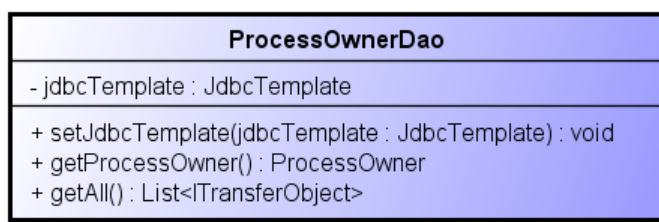


Figura 52: Diagramma classe ProcessOwnerDao

- **Descrizione:** Classe che si occupa delle interrogazioni del *database* relative all'autenticazione del *ProcessOwner*.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:

- com.sirius.sequenziatore.server.model.IDataAccessObject.

La classe invoca i metodi della classe:

- com.sirius.sequenziatore.server.model.ProcessOwner.

• Attributi:

- - JdbcTemplate jdbcTemplate:
Oggetto che fornisce l'accesso alla sorgente dei dati;

• Metodi:

- + Process getProcessOwner():
Ritorna l'oggetto rappresentante il *ProcessOwner*.

7.2.1.6 StepDao

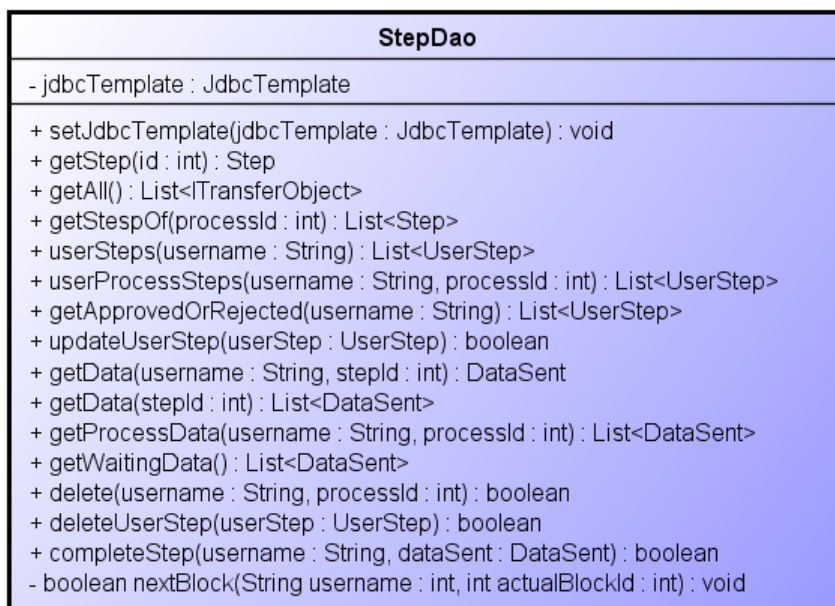


Figura 53: Diagramma classe StepDao

- **Descrizione:** Classe che si occupa delle interrogazioni del *database* relative a tutte le operazioni sui passi dei processi.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:

- com.sirius.sequenziatore.server.model.IDataAccessObject.

La classe invoca i metodi della classe:

- com.sirius.sequenziatore.server.model.Step;
- com.sirius.sequenziatore.server.model.UserStep;
- com.sirius.sequenziatore.server.model.DataSent.

- **Attributi:**

- - JdbcTemplate jdbcTemplate:
Oggetto che fornisce l'accesso alla sorgente dei dati;

- **Metodi:**

- + Step getStep(int id):
Ritorna il passo con l'id specificato;
- + List<Step> getAllStep():
Ritorna tutti i passi;
- + List<Step> getStepOf(int ProcessId):
Ritorna tutti i passi appartenenti al processo di cui si è passato l'id;
- + boolean insertStep(Step step) :
Aggiunge il passo passato come parametro;
- + public boolean updateStep(Step step) :
Aggiorna i dati del passo con l'id corrispondente a quello del passo passato, con i dati del passo passato;
- + List<UserStep> userStep(String userName)
Ritorna una lista di oggetti informativi sullo stato dei passi in corso da parte dell'utente di cui si è passato il nome utente;
- + List<UserStep> userProcessStep(String userName, processId)
Ritorna una lista di oggetti informativi sullo stato dei passi in corso appartenenti al processo di cui si è passato l'id da parte dell'utente di cui si è passato il nome utente;
- + boolean updateUserStep(UserStep userStep):
Aggiornato lo stato del passo per l'utente in questione.
- + List<DataSent> getData(Step step)
Ritorna tutti i dati da tutti gli utenti relativi al passo passato;
- + DataSent getData(String userName, Step step)
Ritorna tutti i dati inviati dall'utente di cui si è passato il nome utente relativi al passo passato;
- + List<DataSent> getWaitingData()
Ritorna tutti i dati di tutti i passi in attesa di approvazione;
- + boolean completeStep(String userName, Step step, DataSent data, Step next)
Notifica e aggiorna nel *database* lo stato dell'utente quando completa o tenta di completare un passo.

7.2.1.7 User

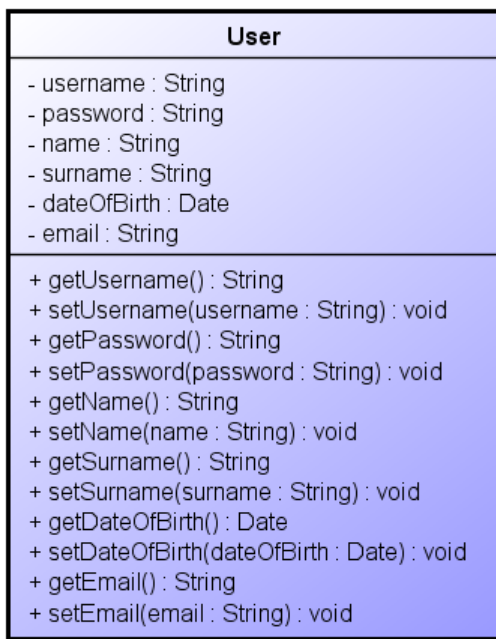


Figura 54: Diagramma classe User

- **Descrizione:** Classe che modella gli utenti del sistema e che funge da interscambio dei dati di quest'ultimi con il *database*.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:

– com.sirius.sequenziatore.server.model.ITransferObject.

- **Attributi:**

- `String userName:`
Nome utente;
- `String password:`
Password dell'utente;
- `String name:`
Nome anagrafico dell'utente;
- `String surName:`
Cognome dell'utente;
- `Date dateOfBirth:`
Data di nascita dell'utente;
- `String email:`
Indirizzo di posta elettronica dell'utente;

- - `int id`:
Codice identificativo `id` associato all'utente.

- **Metodi:**

- + `String getUsername()`:
Ritorna il nome utente;
- + `void setUsername(String userName)`:
Imposta il nome utente;
- + `String getPassword()`:
Ritorna la password dell'utente;
- + `void setPassword(String password)`:
Imposta la password dell'utente;
- + `String getName()`:
Ritorna il nome anagrafico dell'utente;
- + `void setName(String name)`:
Imposta il nome anagrafico dell'utente;
- + `String getSurName()`:
Ritorna il cognome dell'utente;
- + `void setSurName(String surName)`:
Imposta il cognome dell'utente;
- + `Date getDateOfBirth()`:
Ritorna la data di nascita dell'utente;
- + `void setDateOfBirth(Date dateOfBirth)`:
Imposta la data di nascita dell'utente;
- + `String getEmail()`:
Ritorna l'indirizzo di posta elettronica dell'utente;
- + `void setEmail(String email)`:
Imposta l'indirizzo di posta elettronica dell'utente;
- + `int getId()`:
Ritorna il codice `id` associato all'utente;
- + `void setId(int id)`:
Imposta il codice `id` associato all'utente.

7.2.1.8 Process

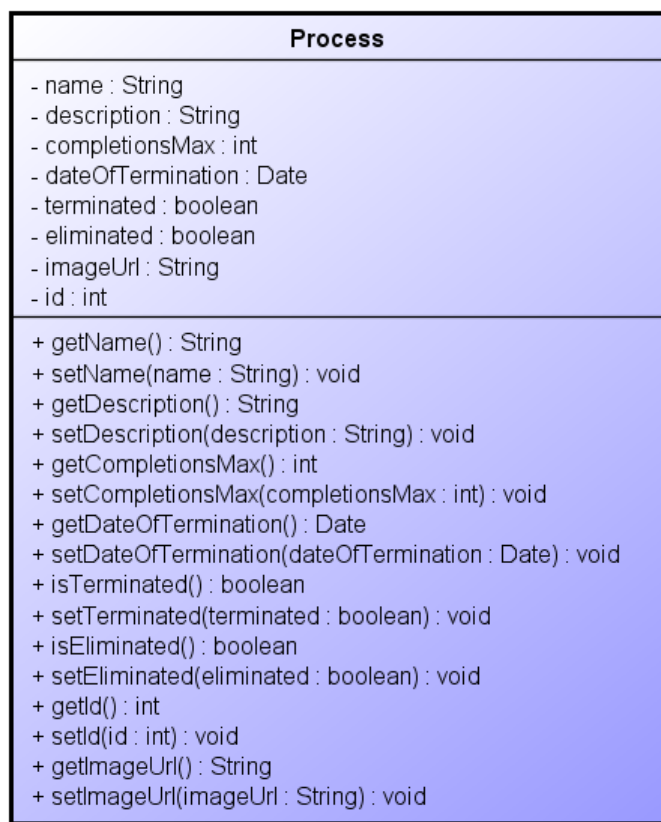


Figura 55: Diagramma classe Process

- **Descrizione:** Classe che modella i processi del sistema e che funge da interscambio dei dati di quest'ultimi con il *database*.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:
 - `com.sirius.sequenziatore.server.model.ITransferObject`.
- **Attributi:**
 - `String name`:
Nome del processo;
 - `String description`:
Descrizione del processo;
 - `int completionsMax`:
Numero massimo di completamenti del processo;
 - `Date dateOfTermination`:
Data di terminazione del processo;

- - `boolean terminated`:
 Booleano vero quando il processo è terminato;
- - `int maxTree`:
 Massimo alberi del processo;
- - `List<Integer> stepsId`:
 Lista di codici id relativi ai passi del processo;
- - `int id`:
 Codice identificativo id associato al processo.

- **Metodi:**

- + `String getName()`:
 Ritorna il nome del processo;
- + `void setName(String name)`:
 Imposta il nome del processo;
- + `String getDescription()`:
 Ritorna la descrizione del processo;
- + `void setDescription(String description)`:
 Imposta la descrizione del processo;
- + `int getCompletionsMax()`:
 Restituisce il numero massimo di completamenti del processo;
- + `void setCompletionsMax(int completionsMax)`:
 Imposta il numero massimo di completamenti del processo;
- + `Date getDateOfTermination()`:
 Ritorna data di terminazione del processo;
- + `void setDateOfTermination(Date dateOfTermination)`:
 Imposta la data di terminazione del processo;
- + `boolean isTerminated()`:
 Ritorna vero se il processo è terminato;
- + `void setTerminated(boolean terminated)`:
 Imposta vero se il processo è terminato;
- + `int getMaxTree()`:
 Ritorna il massimo alberi del processo;
- + `void setMaxtree(int maxTree)`:
 Imposta il massimo alberi del processo;
- + `List<Integer> getStepsId()`:
 Ritorna lista di codici id relativi ai passi del processo;

- + void setStepsId(List<Integer> stepsId):
Imposta lista di codi id relativi ai passi del processo;
- +int getId():
Ritorna codice identificativo id associato al processo;
- +void setId(int id):
Imposta codice identificativo id associato al processo.

7.2.1.9 Step



Figura 56: Diagramma classe Step

- **Descrizione:** Classe che modella i passi del sistema e che funge da interscambio dei dati di quest'ultimi con il *database*.

- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:

- `com.sirius.sequenziatore.server.model.ITransferObject`.

La classe contiene istanze di:

- `com.sirius.sequenziatore.server.model.Condition`;
 - `com.sirius.sequenziatore.server.model.Data`.

- **Attributi:**

- `int id`:
Codice identificativo `id` associato al passo;
 - `String description`:
Descrizione del passo;
 - `List<Data> data`:
Lista con i campi dato del passo;
 - `List<Condition> conditions`:
Lista delle condizioni di avanzamento del passo;
 - `int processId`:
Codice identificativo `id` associato al processo padre;
 - `boolean first`:
Booleano vero se il passo è primo per il processo padre.

- **Metodi:**

- `+ int getId()`:
Ritorna codice identificativo `id` associato al passo;
 - `+ void setId(int id)`:
Imposta codice identificativo `id` associato al passo;
 - `+ String getDescription()`:
Ritorna descrizione del passo;
 - `+ void setDescription(String description)`:
Imposta descrizione del passo;
 - `+ List<Data> getData()`:
Ritorna lista con i campi dato del passo;
 - `+ void setData(List<Data> data)`:
Imposta lista con i campi dato del passo;

- + List<Condition> getConditions():
Ritorna lista delle condizioni di avanzamento del passo;
- + void setConditions(List<Condition conditions):
Imposta lista delle condizioni di avanzamento del passo;
- + int getProcessId():
Ritorna codice id associato al processo padre;
- + void setProcessId(int processId):
Imposta codice id associato al processo padre;
- + boolean isFirst():
Ritorna vero se il passo è primo per il processo padre;
- + void setFirst():
Imposta vero se il passo è primo per il processo padre.

7.2.1.10 DataSent

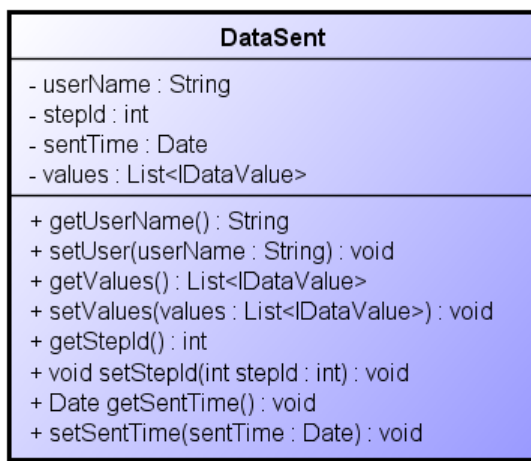


Figura 57: Diagramma classe DataSent

- **Descrizione:** Classe che modella i dati ricevuti dagli utenti che funge da interscambio con il *database*.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:

- com.sirius.sequenziatore.server.model.ITransferObject.

La classe contiene istanze della classe:

- com.sirius.sequenziatore.server.model.IDataValue.

- **Attributi:**

- - **String user:**
Nome utente dell'utente che ha inviato il dato;
- - **List<IDataValue> values:**
Oggetti con il valori dei dati;
- - **int stepId:**
Codice id del passo richiedente il dato.

• Metodi:

- + **String getUser():**
Ritorna nome utente dell'utente che ha inviato il dato;
- + **void setUser(String user):**
Imposta nome utente dell'utente che ha inviato il dato;
- + **List<IDataValue> getValues():**
Ritorna lista di oggetti con il valori dei dati;
- + **void setValues(List<IDataValue> values):**
Imposta lista di oggetti con il valori dei dati;
- + **int getStepId():**
Ritorna codice id del passo richiedente il dato;
- + **void setStepId(int stepId):**
Imposta codice id del passo richiedente il dato.

7.2.1.11 IDataValue

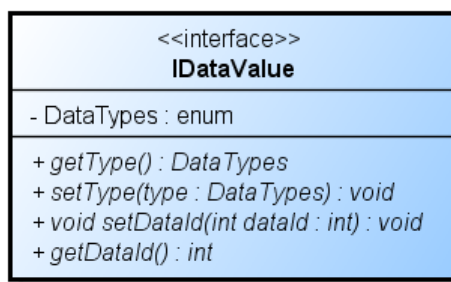


Figura 58: Diagramma interfaccia IDataValue

- **Descrizione:** Interfaccia che modella i valori dei dati ricevuti.
- **Metodi:**

- + **int getId():**
Ritorna codice id associato al valore;

- + void setId(int id):
Imposta codice id associato al valore.
- + DataType getType():
Ritorna il tipo del valore.

7.2.1.12 TextualValue

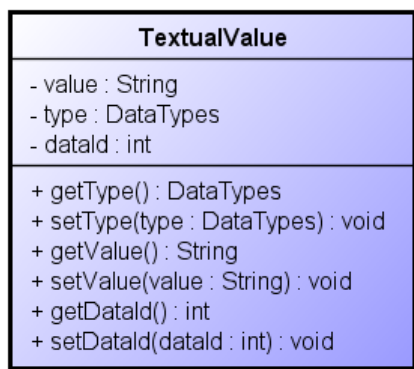


Figura 59: Diagramma classe TextualValue

- **Descrizione:** Classe che modella i valori dei dati testuali.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:
 - com.sirius.sequenziatore.server.model.IDataValue.
- **Attributi:**
 - - int id:
Codice id associato al valore;
 - - String value:
Valore testuale.
- **Metodi:**
 - + String getValue():
Ritorna valore testuale;
 - + void setValue(String value):
Imposta valore testuale.

7.2.1.13 NumericValue

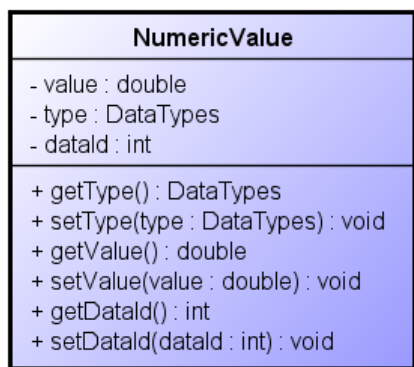


Figura 60: Diagramma classe NumericValue

- **Descrizione:** Classe che modella i valori dei dati numerici.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:
 - `com.sirius.sequenziatore.server.model.IDataValue`.
- **Attributi:**
 - `int id`:
Codice id associato al valore;
 - `double value`:
Valore numerico.
- **Metodi:**
 - `+ double getValue()`:
Ritorna valore numerico;
 - `+ void setValue(double value)`:
Imposta valore numerico.

7.2.1.14 ImageValue

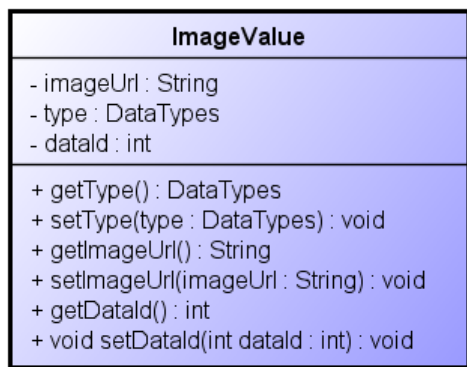


Figura 61: Diagramma classe ImageValue

- **Descrizione:** Classe che modella i valori dei dati immagine.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:

– `com.sirius.sequenziatore.server.model.IDataValue`.

- **Attributi:**

- `int id`:
Codice `id` associato al valore;
- `String imageUrl`:
Percorso *URL* dell'immagine.

- **Metodi:**

- `+ String getImageUrl()`:
Ritorna percorso *URL* dell'immagine;
- `+ void setImageUrl(String imageUrl)`:
Imposta percorso *URL* dell'immagine.

7.2.1.15 GeographicValue

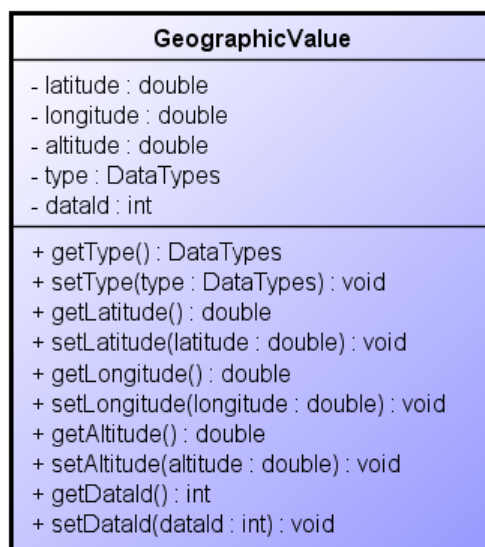


Figura 62: Diagramma classe GeographicValue

- **Descrizione:** Classe che modella i valori dei dati geografici.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:
 - com.sirius.sequenziatore.server.model.IDataValue.
- **Attributi:**
 - `int id`:
Codice id associato al valore;
 - `double latitude`:
Latitudine;
 - `double longitude`:
Longitudine;
 - `double altitude`:
Altitudine.
- **Metodi:**
 - `double getLatitude()`:
Ritorna latitudine;
 - `void setLatitude(double latitude)`:
Imposta latitudine;

```

- + double getLongitude():
    Ritorna longitudine;
- + void setLongitude(double longitude):
    Imposta longitudine;
- + double getAltitude():
    Ritorna altitudine;
- + void setAltitude(double altitude):
    Imposta altitudine.

```

7.2.1.16 UserStep

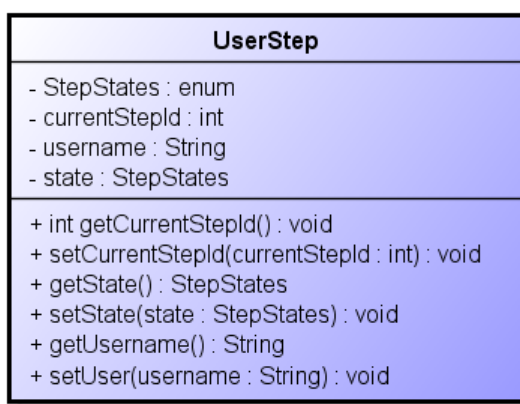


Figura 63: Diagramma classe UserStep

- **Descrizione:** Classe che modella i passi in corso e che funge da interscambio dei dati di quest'ultimi con il *database*.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:
 - `com.sirius.sequenziatore.server.model.ITransferObject`.
- **Attributi:**
 - + `enum stepStates{ONGOING, EXPECTANT, REJECTED, APPROVED}`: Enumerazione stato avanzamento;
 - - `int currentStepId`: Codice id del passo attuale;
 - - `stepStates state`: Stato avanzamento;
 - - `String user`: Nome utente dell'utente del caso.

- **Metodi:**

- + `int getCurrentStepId()`:
Ritorna il codice id del passo attuale;
- + `void setCurrentStepId(int currentStepId)`:
Imposta il codice id del passo attuale;
- + `stepStates getState()`:
Ritorna stato avanzamento;
- + `void setStates(stepStates state)`:
Imposta stato avanzamento;
- + `String getUser()`:
Restituisce nome utente dell'utente in caso;
- + `void setUser(String user)`:
Imposta nome utente dell'utente in caso.

7.2.1.17 ProcessOwner

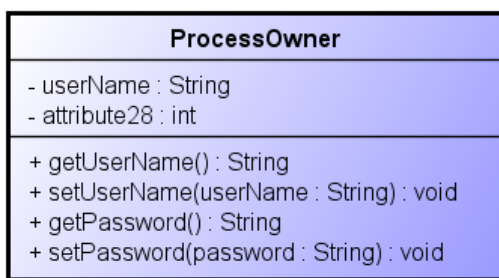


Figura 64: Diagramma classe ProcessOwner

- **Descrizione:** Classe che modella il ProcessOwner e che funge da interscambio dei dati di quest'ultimo con il *database*.
- **Relazione con altre componenti:** la classe implementa la seguente interfaccia:

- `com.sirius.sequenziatore.server.model.ITransferObject`.

- **Attributi:**

- - `String userName`:
Nome utente *Process Owner*;
- - `String password`:
Password *Process Owner*.

• Metodi:

- + String getUsername():
Ritorna il nome utente del *Process Owner*;
- + void setUsername(String userName):
Imposta il nome utente del *Process Owner*;
- + String getPassword():
Ritorna la password del *Process Owner*;
- + void setPassword(String password):
Imposta la password del *Process Owner*.

8 Diagrammi di sequenza

8.1 Creazione di un processo

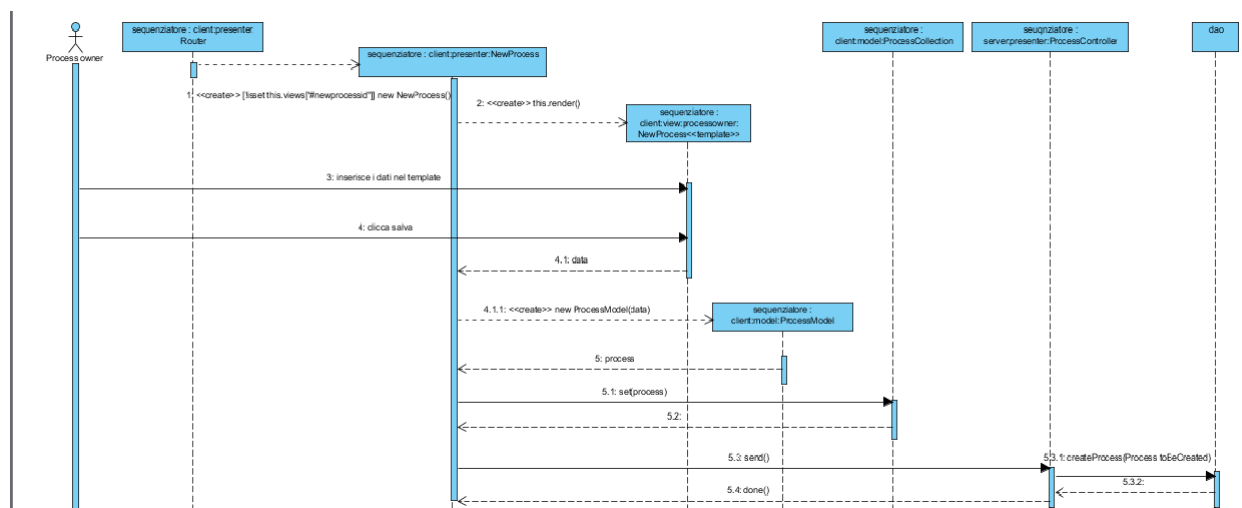


Diagramma di sequenza - Creazione di un processo

8.1.0.18 Descrizione della creazione di un processo La sequenza inizia con l'evento creazione di un nuovo processo da parte di un utente *process owner*, quindi il client:presenter:router crea un oggetto di tipo client:presenter:NewProcess che a sua volta crea una view (tramite il relativo template) in modo che l'utente possa inserire i dati relativi al processo. Una volta che l'utente salva il processo, i dati vengono ritornati all'oggetto NewProcess, il quale crea un'istanza (process) del client:model:ProcessModel; tramite il metodo *set(process)* viene aggiornata la processCollection e viene inviata al server (messaggio sincrono *send()*) restando quindi in attesa del messaggio di conferma, da qui il server si occupa di creare effettivamente il suddetto processo (metodo *createProcess(ProcessToBeCreated)*), ed una volta fatto, comunicherà al client l'avvenuta creazione, terminando la sequenza.

8.2 Approvazione di un passo

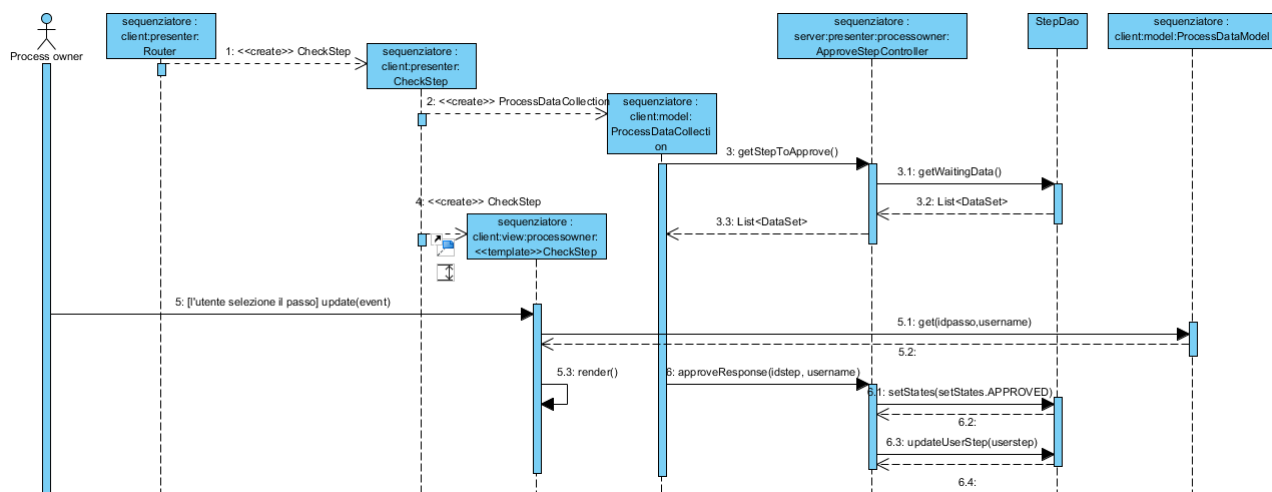


Diagramma di sequenza - Approvazione di un passo

8.2.0.19 Descrizione dell'approvazione di un passo Il goal è quello di riuscire a confermare un passo in attesa di approvazione. La sequenza inizia con il *client:presenter:Router* che crea un nuovo oggetto *client:presenter:CheckStep*, tale oggetto a sua volta crea un oggetto *client:model:ProcessDataCollection* che richiede al server tramite il metodo *getStepToApprove()* la lista di tutti i passi in attesa di conferma. Lato server il *presenter:processowner:ApproveStepController* viene istanziato, in seguito tale oggetto invia una richiesta (*getWaitingData()*) allo *StepDao* e resta in attesa di ricevere i suddetti dati, ossia la lista dei passi in attesa di approvazione. Una volta ottenuta tale lista, essa viene ritornata al client (più precisamente al *model:ProcessDataCollection*) tramite *List<DataSet>*, altempo viene creata una view tramite la quale l'utente può scegliere il passo da approvare, una volta fatto, tramite il metodo *approveResponse(idstep, username)* il client comunica al server il passo approvato, ed il server tramite *setStates(setStates.APPROVED)* attua la reale modifica nello *stepDao* confermando il passo selezionato, terminando la sequenza.

8.3 Registrazione

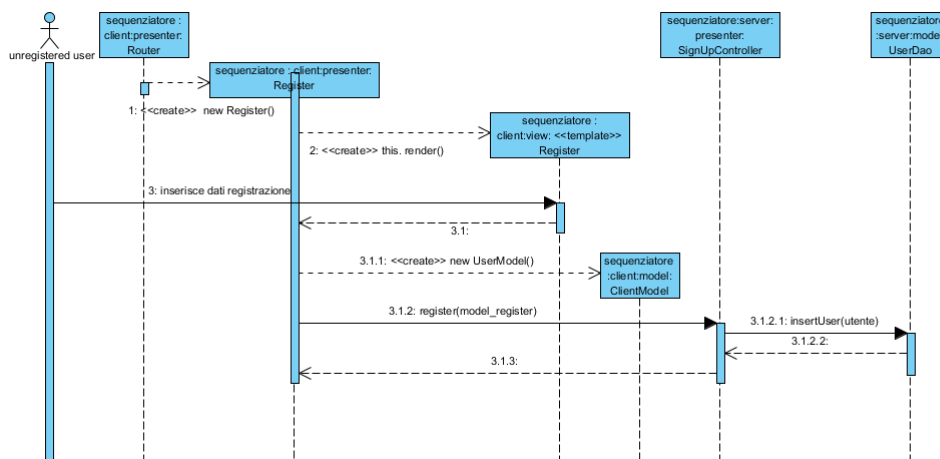


Diagramma di sequenza - Registrazione di un utente

8.3.0.20 Deacrizione della Registrazione utente In questo diagramma di attività viene mostrato lo scenario di registrazione di un nuovo utente. La sequenza inizia sempre dal *client:presenter:router* che crea un oggetto della classe *client:presenter:Register*, tramite il metodo *render* si crea la view con cui l'utente può interagire inserendo i dati relativi alla sua registrazione. Raccolti i dati essi vengono ritornati al *client:presenter:Register* il quale istanzia un nuovo oggetto (attraverso il costruttore: *new UserModel()*) di classe *client:model:ClientModel* tale oggetto è poi utilizzato come parametro nel metodo *register(model_register)*. Quest'ultimo messaggio sincrono attende la risposta del server circa l'avvenuta inserzione del nuovo user nel database. Lato server quindi tramite il metodo *insertUser(utente)* l'utente viene effettivamente registrato, e di conseguenza il server lo comunica al client, terminando la sequenza.