

Document d'exploitation

Portail d'administration pour SAS Viya

Pierre Portal

2022-08-08

Public © 2022 CGI Inc.

CGI

Table of contents

Table of Contents –

REMOVE THIS NOTE PRIOR TO DISTRIBUTION

It is not necessary to manually update the table of contents (TOC) for this template. The TOC will update automatically when you use the heading styles within the template (Heading 1, 2, 3 and 4). To update your TOC (e.g., if you change a heading), right click on the TOC and select **Update Field**. Then, choose **Update page numbers only** or **Update entire table**. If you do not need a TOC, simply highlight the TOC and delete it.

1	Contexte	6
2	Architecture	
3	Design	
4	Présentation des fonctionnalités	6
4.1	Authentification	12
4.1.1	Enregistrement de l'application	12
4.1.2	Implémentation de l'authentification	13
4.2	Requêtes API	15
4.3	Configuration	16
4.4	Gestion des erreurs et notifications	17
4.5	Mise en cache	18
4.6	Etat de la plateforme	19
4.7	Suivi en temps réel	21
4.8	Processus en arrière-plan	22

1 Contexte

Dans le cadre de sa mission auprès du client Covéa, CGI est responsable de l'administration applicative d'une plateforme hébergeant la solution SAS Viya. Cette solution est utilisée par plus d'une centaine d'utilisateurs pour y développer des rapports de données. L'outil SAS Viya est un produit récent que la stratégie de l'éditeur destine à être le futur de SAS.

SAS Viya comprend de nombreuses fonctionnalités supplémentaires et des innovations au niveau de l'architecture en comparaison des versions précédentes :

- elle permet l'implémentation "clef en main" de modèles de Deep Learning, à l'aide d'une interface cliquable ou avec du code
- les différents logiciels qui permettent d'utiliser SAS deviennent des applications web centralisant davantage les fonctionnalités. Ainsi, pour les tâches d'administration, la nouvelle web application SAS Environment Manager comprend les fonctionnalités des anciens logiciels SAS Environment Manager, SAS Management Console, SAS Visual Analytics Administrator et SAS Deployment Manager.
- en se basant sur le concept du microservice, les différents services de SAS Viya deviennent des modules indépendants qui communiquent les uns avec les autres au moyen d'une API REST. Cela permet une grande flexibilité pour le déploiement de la plateforme et facilite aussi le développement d'applications tierces qui viennent s'interfacer aux différents services de SAS Viya.

Les actions d'administration sur la plateforme sont nombreuses et croissantes avec le nombre d'utilisateurs sur la plateforme : création d'environnement de travail pour les équipes, création de groupes consommateurs et créateurs, habilitation des utilisateurs, création de bibliothèques et de dossiers SAS, application des permissions sur les bibliothèques et les dossiers SAS. Il en va de même pour le besoin de monitorer la plateforme le plus précisément possible (suivi en temps réel de différents indicateurs tels que : le nombre d'utilisateurs connectés, utilisation des ressources CPU et RAM, état des services, occupation des disques dur).

Les actions d'administration ainsi que le monitoring de la plateforme sont opérés en utilisant une solution existante : l'application web « Environment Manager ». Cette solution présente cependant de nombreuses contraintes qui nous poussent à vouloir développer de nouveaux outils qui viseront à améliorer les différentes tâches d'administration en simplicité et en rapidité.

L'idée de ce projet est de créer un tableau de bord sous la forme d'une web application qui permettrait aux administrateurs de la plateforme SAS Viya d'effectuer des actions d'administration et de se tenir rapidement au courant de l'état de la plateforme. Cette application interfacerait avec les services SAS relatifs à l'administration au moyen de l'API REST.

2 Architecture

On souhaite rendre l'application facilement modulable pour ajouter de nouvelles fonctionnalités dans le futur. Pour se faire, on a séparé les fichiers relatifs aux différentes fonctionnalités suivant leurs types et suivant leurs domaines d'application (Dossiers, Utilisateurs, Groupes, Services, etc ...).

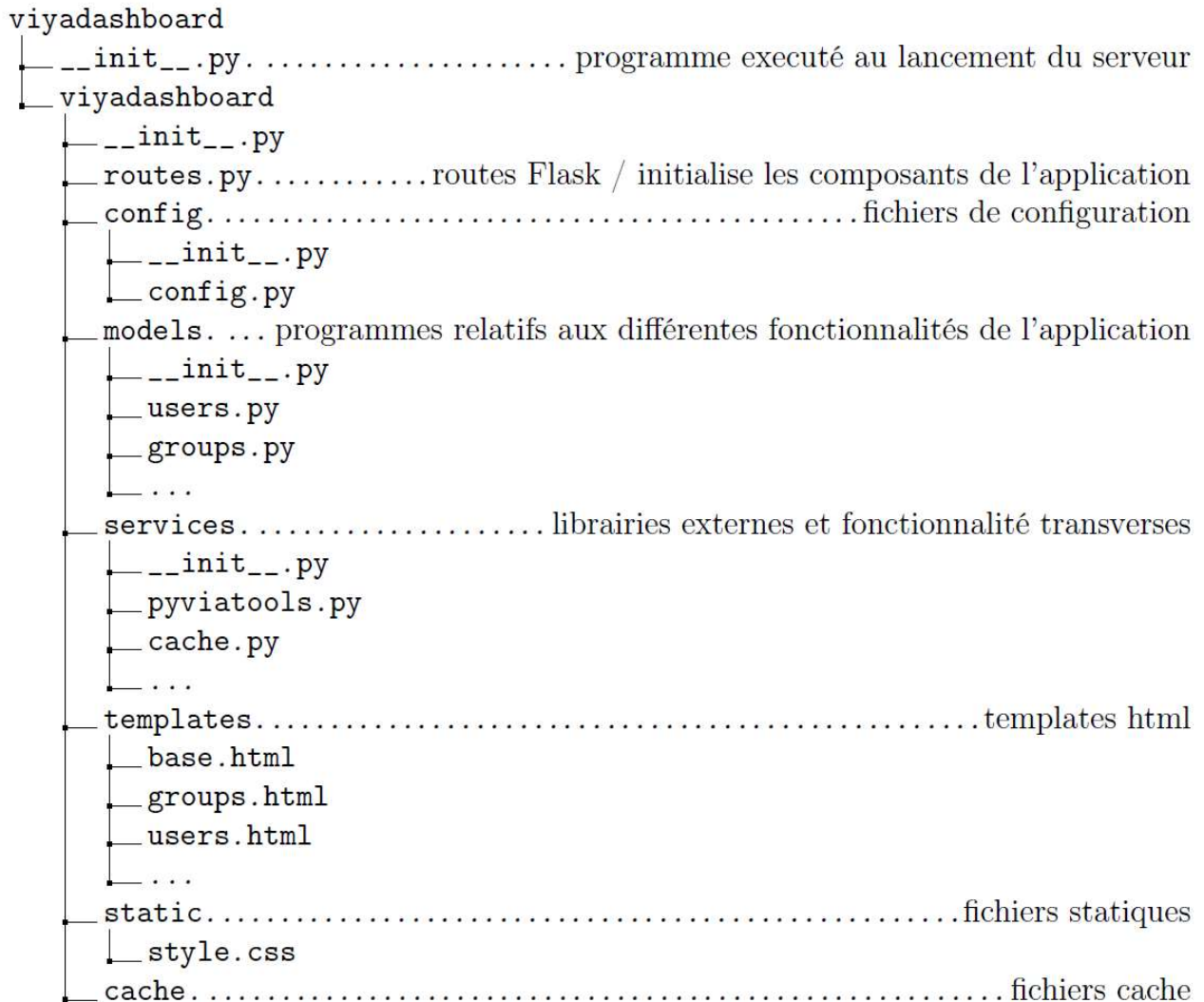


Figure 1 - Architecture de l'application

Pour faciliter la structuration de l'application, il est nécessaire de diviser son code Python en plusieurs fichiers qui seront importés suivant nos besoins, de la même manière qu'une bibliothèque Python. Pour permettre cette structuration, on doit convertir son projet en tant que package qui permettra alors d'utiliser la fonction import sur tous les fichiers Python de l'application. Pour convertir son projet en package, il est nécessaire d'inclure un fichier `__init__.py` dans chacun des dossiers contenant des fichiers Python. On pourra alors les importer comme des bibliothèques en spécifiant le chemin du dossier et le nom du fichier :

```
import viyadashboard.models.groups
import viyadashboard.models.users
import viyadashboard.services.pyviatools
```

3 Design

Flask comprend un moteur de rendu de template HTML appelé Jinja. Il permet d'augmenter les fichiers HTML en y implémentant, par exemple, de la généricité, des conditions if, l'affichage de variables Python ou encore des boucles for.

Toutes les pages web de l'application implémentent le fichier base.html. Celui-ci contient la logique qui permet de vérifier que l'utilisateur est bien authentifié ainsi que la barre latérale. Si l'utilisateur est considéré comme authentifié alors on affiche la barre latérale avec le template HTML qui hérite notre fichier base.html. Si l'utilisateur n'est pas connecté, on le renvoie vers l'URL d'authentification. Ce mécanisme d'authentification est présenté dans la section 4.1 de ce document. Le fichier base.html contient aussi le bloc qui permet l'affichage des messages de retour qui font suite aux actions d'administration, le fonctionnement de ce système est expliqué dans la section 4.4.

Les templates HTML utilisent le framework CSS Bootstrap pour tout ce qui concerne les éléments de design : boutons, textes, cadres, modal, sélecteurs, formulaires. Concernant les tableaux, on utilise le framework DataTables pour augmenter les fonctionnalités des tableaux Bootstrap. Cela apporte de nombreuses fonctionnalités telles que : la recherche dans le tableau, le tri par colonne, le choix du nombre de lignes par page, un chargement dynamique des données avec AJAX.

Pour naviguer de page en page, on utilise une barre latérale qui renvoie vers les différents modules de l'application.

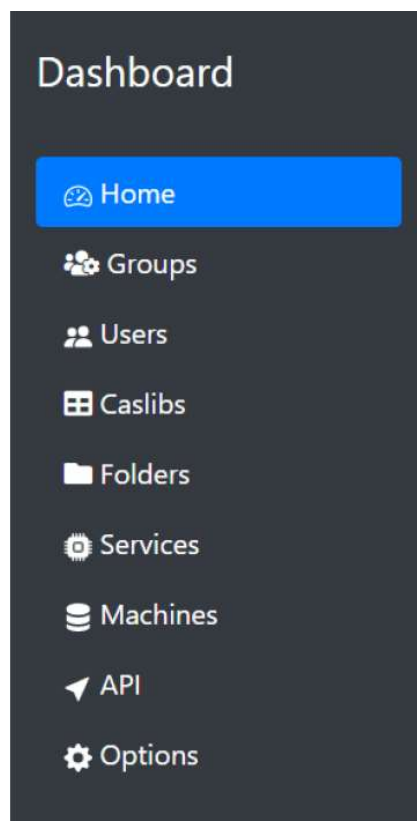


Figure 2 - La barre latérale

4 Présentation des fonctionnalités

Chaque page de l'application correspond à un module qui regroupe les fonctionnalités liées à un même domaine.

La page d'accueil permet d'informer sur l'état général de la plateforme et cela en temps réel. Parmi les informations qui sont représentées, on a : le nombre d'utilisateurs, le nombre de groupes, le nombre de sessions, le nombre d'utilisateurs connectés, l'état des services et l'état des machines. Ces informations sont actualisées toutes les minutes, ce mécanisme de mise à jour automatique est expliqué dans la section 4.7 de ce document.

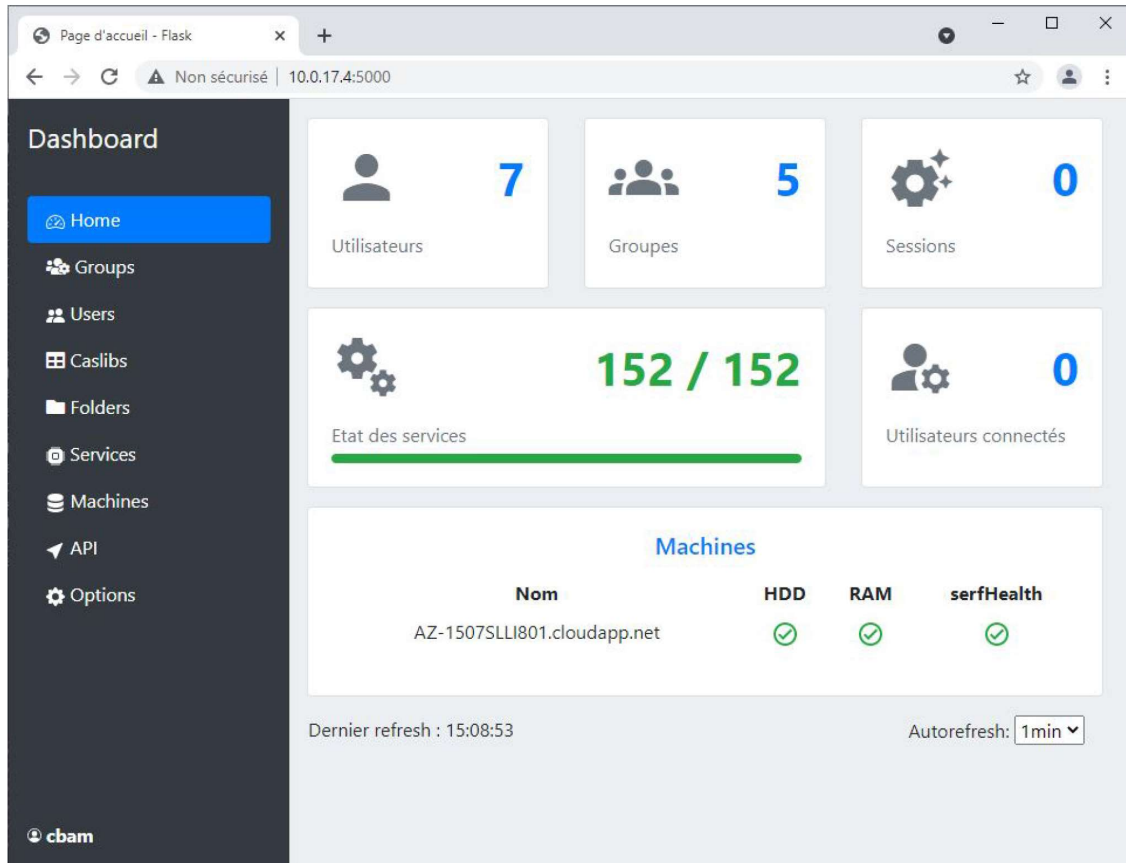


Figure 3 - Page d'accueil

Dans la partie "Groupes", on peut consulter la liste des groupes de la plateforme. Les informations sur ces groupes sont représentées dans un tableau DataTable. Chaque groupe est une ligne dans le tableau et il possède un bouton d'action qui permet d'opérer des actions d'administration. On peut aussi créer des groupes avec le bouton correspondant. Cette création peut se faire manuellement ou via un fichier csv.

Dashboard

- Home
- Groups**
- Users
- Casilbs
- Folders
- Services
- Machines
- API
- Options

PCGI102@societe.mma.fr

Liste des groupes personnalisés

Show 50 entries

Search:

Ajouter groupe

Group ID	Group description	Creation Date	Last modified	Action
ACT_SANTE_PREV_CREAT				...
ACT_SANTE_PREV_GMF_CONSO	ACT_SANTE_PREV_GMF_CONSO			...
ACT_SANTE_PREV_MAN_CONSO	Pilotage Managers	14:38	14:38	...
ACT_SANTE_PREV_OPE_CONSO	Pilotage Opérationnel	17/03/2020 14:39	17/03/2020 14:39	...
APP_ASSUREX_GMFVIE_CONSO	APP_ASSUREX_GMFVIE_CONSO	09/04/2021 11:27	09/04/2021 11:27	...
APP_ASSUREX_GMFVIE_CREATEUR	APP_ASSUREX_GMFVIE_CREATEUR	09/04/2021 11:27	09/04/2021 11:27	...
APP_CARTODATAHUB_CONSO	APP_CARTODATAHUB_CONSO	08/04/2021 08:27	08/04/2021 08:27	...
APP_CARTODATAHUB_CREATEUR	APP_CARTODATAHUB_CREATEUR	08/04/2021 08:29	08/04/2021 08:29	...
APP_DEMAT_LOG_CONSO	APP_DEMAT_LOG_CONSO	31/08/2020 14:07	31/08/2020 14:07	...

Ajouter des utilisateurs
Ajouter des groupes
Ajouter des groupes personnalisés
Supprimer

Figure 4 - Page des groupes

Ajouter groupes

Ajouter des groupes personnalisés :

Group ID

Group name

Group description

+

-

? Importer des groupes depuis un fichier :

Choisir un fichier

Aucun fichier choisi

Submit

Figure 5 - Formulaire de création de groupe

La page des utilisateurs dresse la liste complète des utilisateurs de la plateforme avec les informations qui les concernent. Dans cette page, une fonctionnalité intéressante est la colonne groupes qui permet de consulter la liste des groupes pour un utilisateur donnée. Cette fonctionnalité n'est pas présente dans l'Environment Manager (on peut consulter les membres d'un groupe mais on ne peut pas connaître la liste des groupes pour un utilisateur donné). Cela s'explique par le manque de fonctionnalités offertes par l'API. Pour obtenir la liste des groupes par utilisateur, il est d'abord nécessaire de récupérer la liste de tous les groupes puis de faire une requête à l'API pour chaque groupe pour récupérer leur liste d'utilisateurs. Le temps d'exécution de cette opération peut durer plusieurs minutes, il a donc été nécessaire d'implémenter un système de mise en cache pour accélérer grandement l'appel à la fonction. Ce mécanisme de mise en cache est expliqué en détail dans la section 4.5 de ce document

The screenshot shows a web application titled 'Users - Flask'. The main content area is titled 'Liste des utilisateurs'. It features a search bar and a table with the following data:

ID	Name	Groups
cbam	Charif Bamba	DataBuilders SASAdministrators
demo	demo	SASAdministrators
fpol	Fabrice Pollet	DataBuilders SASAdministrators
jgut	jgut	DataBuilders SASAdministrators
ktel	ktel	DataBuilders SASAdministrators
mlev	mlev	DataBuilders SASAdministrators
ppor	Pierre Portal	DataBuilders SASAdministrators

At the bottom of the table, it says 'Showing 1 to 7 of 7 entries'. There are navigation buttons for 'Previous', '1' (selected), and 'Next'. The sidebar on the left contains links to Home, Groups, Users (highlighted), Caslibs, Folders, Services, Machines, API, and Options. The user 'cbam' is logged in.

Figure 6 - Page des utilisateurs

La page des dossiers est sensiblement similaire à celle des groupes : un tableau DataTable permet de représenter les informations sur les dossiers et chaque dossier est une ligne dans le tableau. En cliquant sur un dossier, on peut accéder à son contenu et la barre de navigation est actualisée. Sur cette page on peut aussi créer des dossiers de façon manuelle ou alors via un fichier csv. Les tâches permises par le bouton d'action pour un dossier sont: la suppression du dossier ou la consultation de ses droits d'accès.

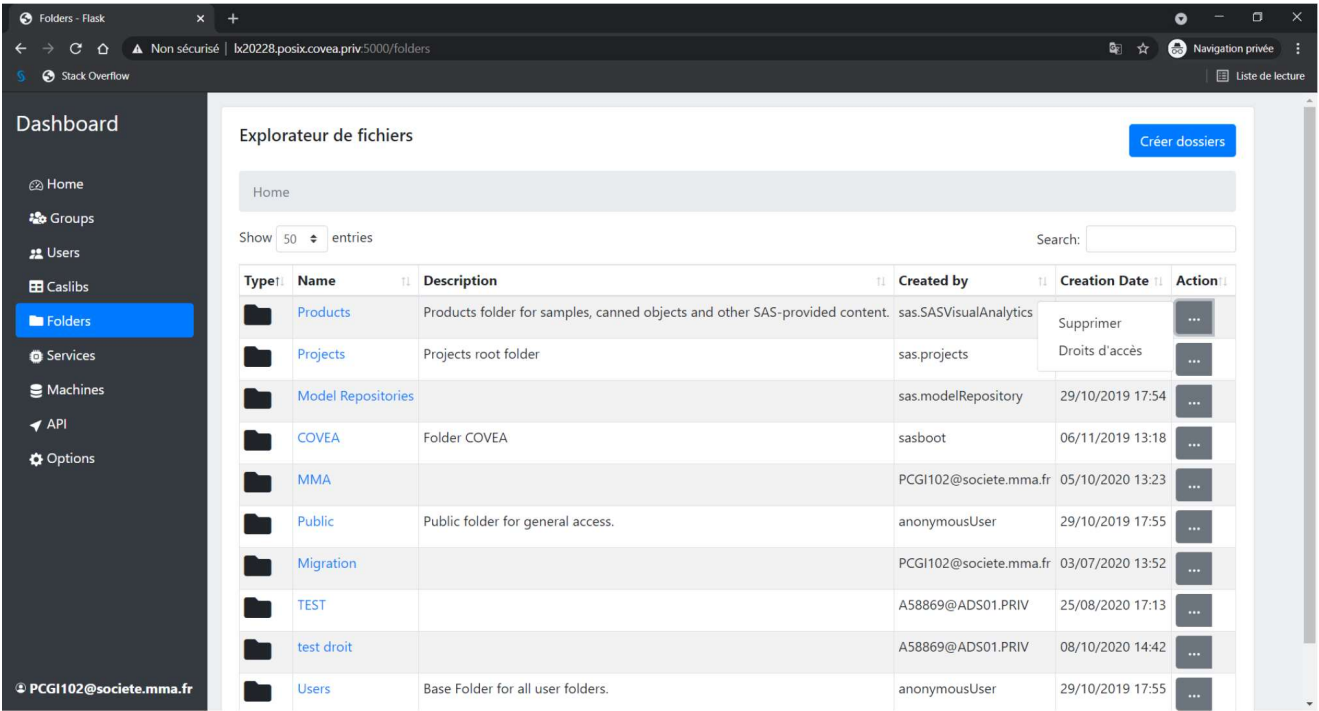


Figure 7 - Page des dossiers



Figure 8 - Consultation des droits sur un dossier

Comme pour les dossiers, la page des caslibs permet de consulter les informations sur les caslibs et d'explorer leur contenu. Quand on clique sur une caslib, on a accès à sa liste de castables avec leurs informations.

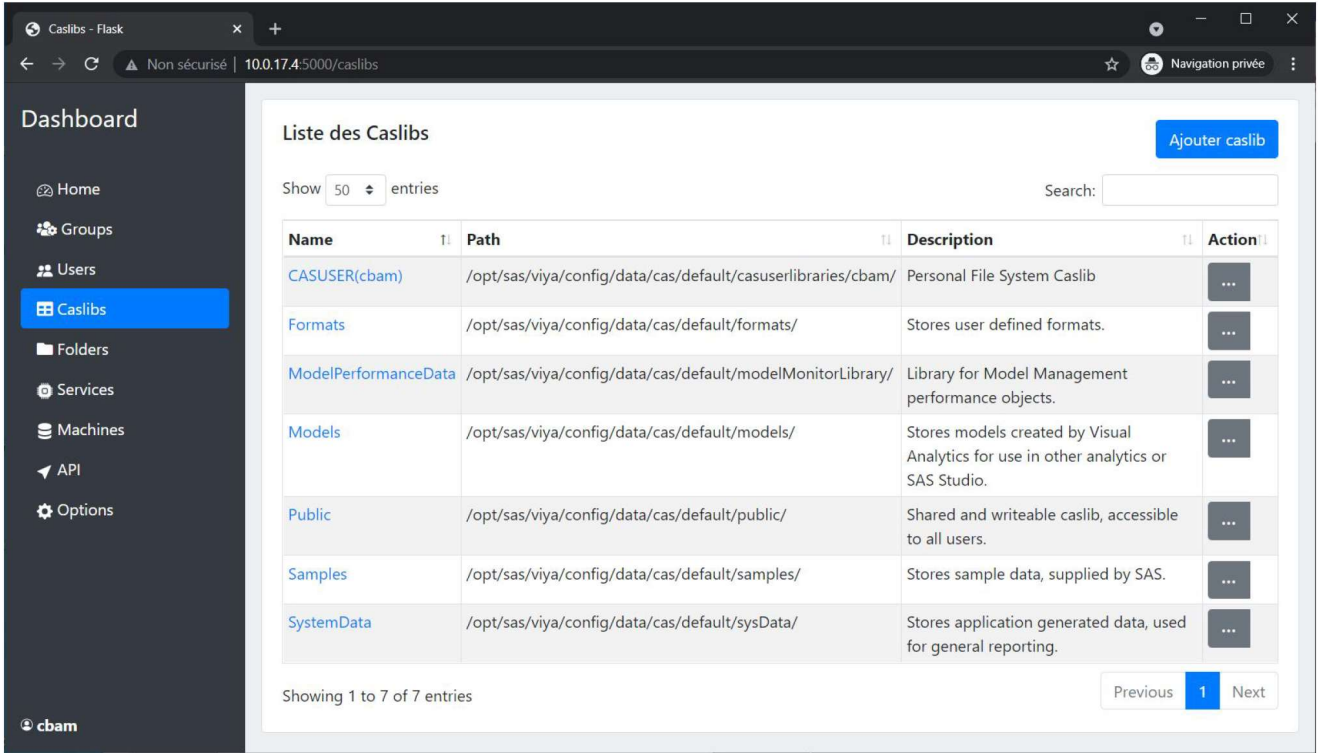


Figure 9 - Page des caslibs

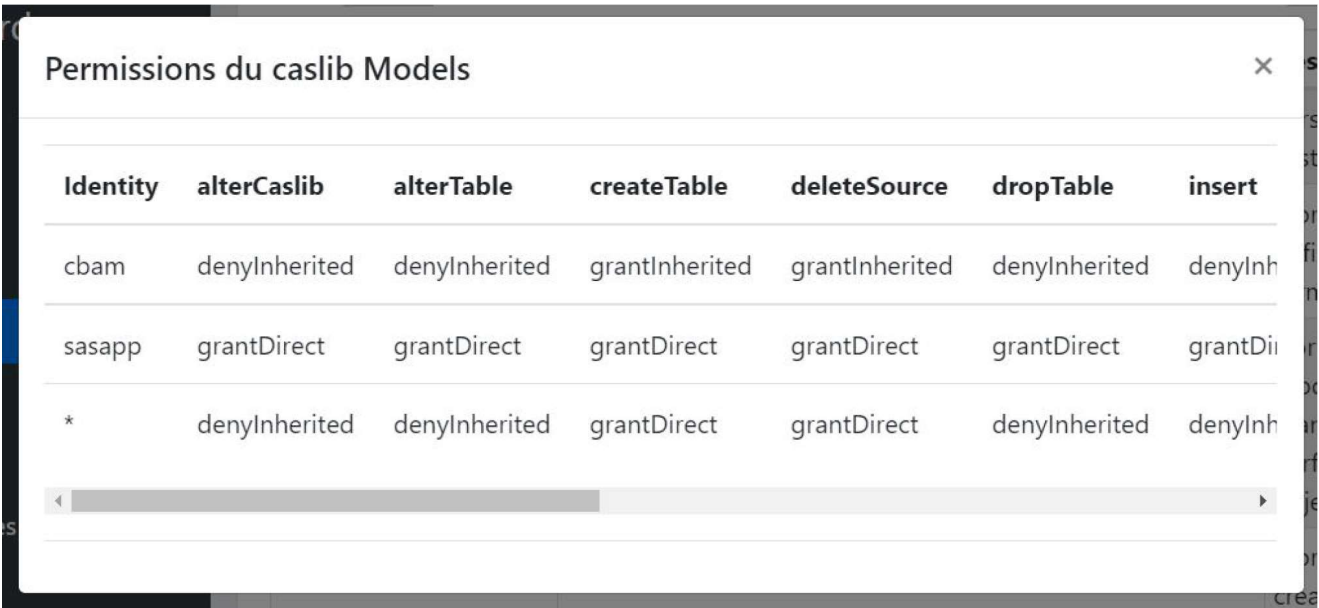
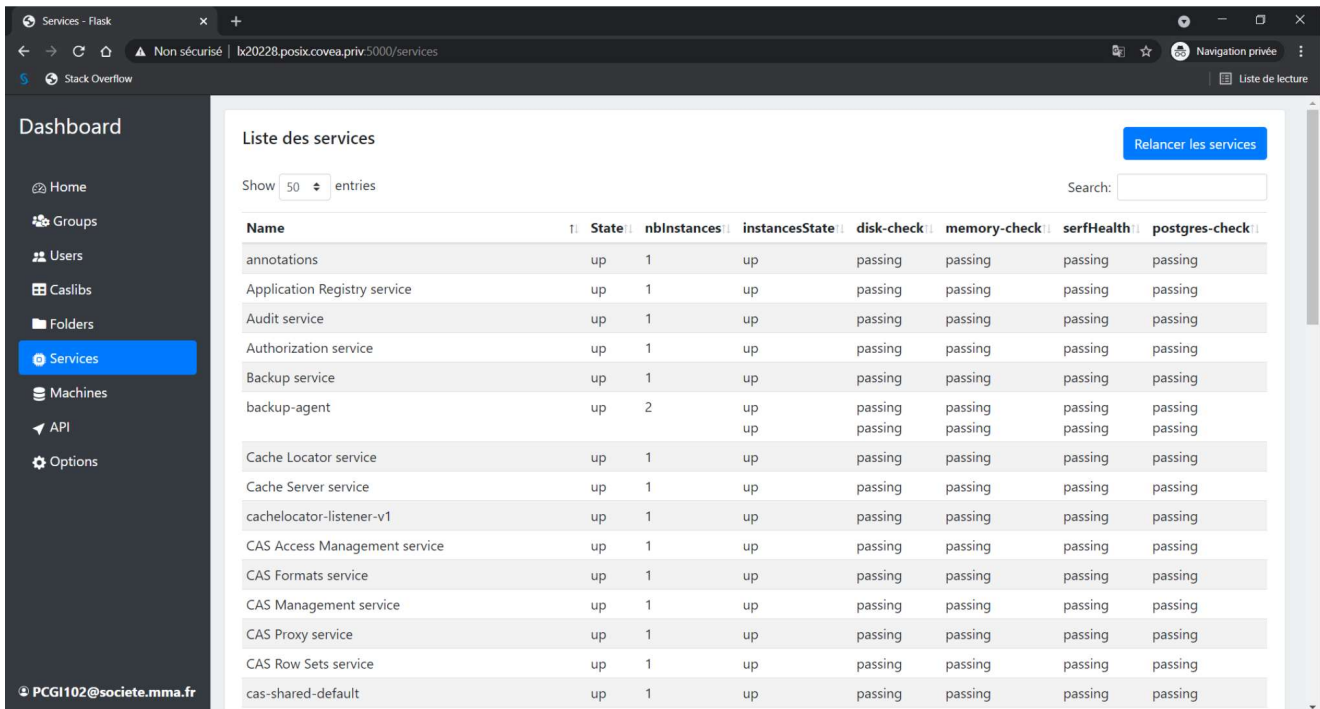


Figure 10 - Consultation des droits sur une caslib

Les pages des services et des machines permettent de consulter en détail les informations sur l'état de la plateforme. La page des services permet aussi d'exécuter le script de relance des services. Ce script est un fichier bash qui doit être spécifié au moment de la configuration et qui donne la commande pour relancer les services sur le serveur Linux, cette commande peut grandement varier selon les environnements Viya. La relance des services peut prendre plus de 15 minutes, c'est pourquoi il est nécessaire de vérifier l'état du processus de relance des services et d'informer l'utilisateur de son avancement. Ce procédé est expliqué dans la section 2.3.8 de ce document.



Services - Flask

Non sécurisé | lx20228.posix.covea.priv:5000/services

Stack Overflow

Navigation privée

Liste de lecture

Dashboard

Home

Groups

Users

Caslibs

Folders

Services

Machines

API

Options

PCGI102@societe.mma.fr

Liste des services

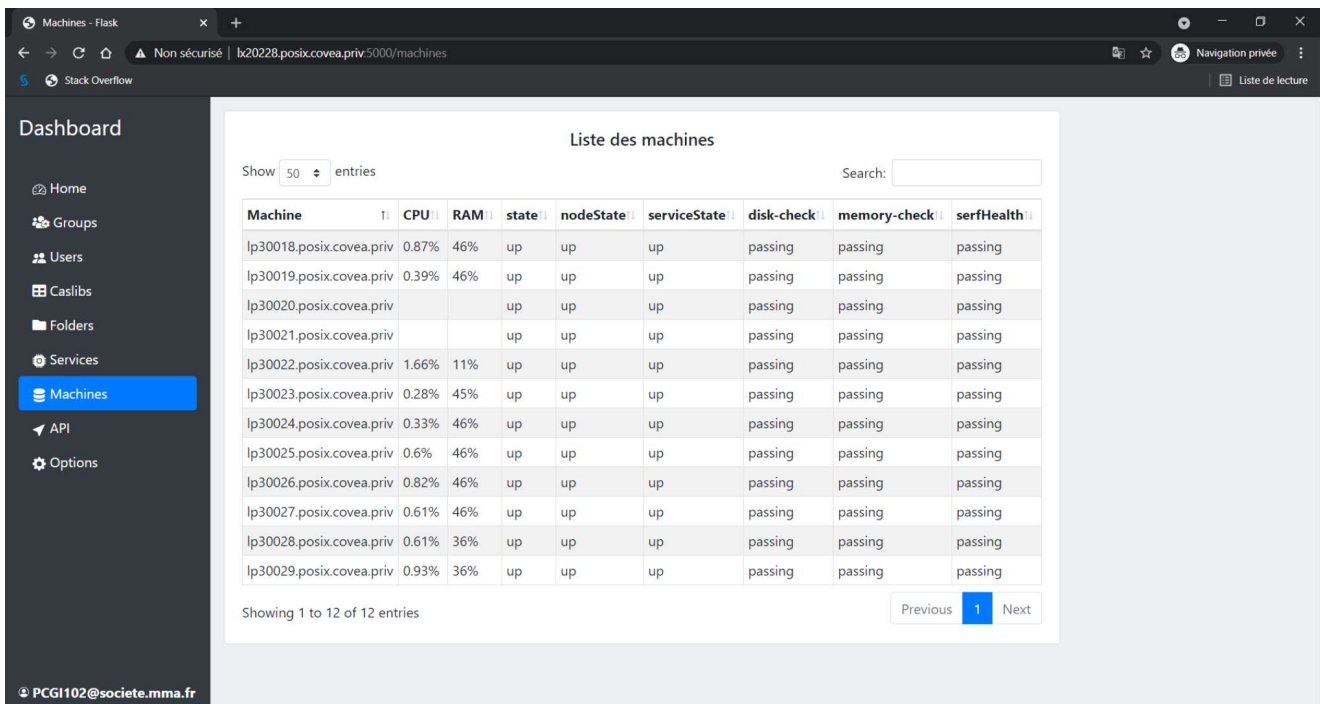
Relancer les services

Show 50 entries

Search:

Name	State	nbInstances	instancesState	disk-check	memory-check	serfHealth	postgres-check
annotations	up	1	up	passing	passing	passing	passing
Application Registry service	up	1	up	passing	passing	passing	passing
Audit service	up	1	up	passing	passing	passing	passing
Authorization service	up	1	up	passing	passing	passing	passing
Backup service	up	1	up	passing	passing	passing	passing
backup-agent	up	2	up	passing	passing	passing	passing
Cache Locator service	up	1	up	passing	passing	passing	passing
Cache Server service	up	1	up	passing	passing	passing	passing
cachelocator-listener-v1	up	1	up	passing	passing	passing	passing
CAS Access Management service	up	1	up	passing	passing	passing	passing
CAS Formats service	up	1	up	passing	passing	passing	passing
CAS Management service	up	1	up	passing	passing	passing	passing
CAS Proxy service	up	1	up	passing	passing	passing	passing
CAS Row Sets service	up	1	up	passing	passing	passing	passing
cas-shared-default	up	1	up	passing	passing	passing	passing

Figure 11 - Page des services



Machines - Flask

Non sécurisé | lx20228.posix.covea.priv:5000/machines

Stack Overflow

Navigation privée

Liste de lecture

Dashboard

Home

Groups

Users

Caslibs

Folders

Services

Machines

API

Options

PCGI102@societe.mma.fr

Liste des machines

Show 50 entries

Search:

Machine	CPU	RAM	state	nodeState	serviceState	disk-check	memory-check	serfHealth
lp30018.posix.covea.priv	0.87%	46%	up	up	up	passing	passing	passing
lp30019.posix.covea.priv	0.39%	46%	up	up	up	passing	passing	passing
lp30020.posix.covea.priv			up	up	up	passing	passing	passing
lp30021.posix.covea.priv			up	up	up	passing	passing	passing
lp30022.posix.covea.priv	1.66%	11%	up	up	up	passing	passing	passing
lp30023.posix.covea.priv	0.28%	45%	up	up	up	passing	passing	passing
lp30024.posix.covea.priv	0.33%	46%	up	up	up	passing	passing	passing
lp30025.posix.covea.priv	0.6%	46%	up	up	up	passing	passing	passing
lp30026.posix.covea.priv	0.82%	46%	up	up	up	passing	passing	passing
lp30027.posix.covea.priv	0.61%	46%	up	up	up	passing	passing	passing
lp30028.posix.covea.priv	0.61%	36%	up	up	up	passing	passing	passing
lp30029.posix.covea.priv	0.93%	36%	up	up	up	passing	passing	passing

Showing 1 to 12 of 12 entries

Previous 1 Next

Figure 12 - Page des machines

4.1 Authentification

On voudrait qu'un utilisateur qui souhaite accéder à l'application puisse s'y connecter à l'aide de ses identifiants SAS. Pour se faire, il est d'abord nécessaire d'enregistrer l'application en tant que "client" au service Authentification de SAS Viya. Cette étape permet d'obtenir une URL d'authentification qu'il faudra ensuite implémenter dans la solution.

4.1.1 Enregistrement de l'application

L'authentification sur SAS Viya repose sur le protocole OAuth 2.0. Ce protocole propose de nombreuses façons de s'identifier auprès d'un serveur. Ici on utilise la méthode `authorization_code` pour permettre d'enregistrer l'application sur SAS Viya. Cette méthode se base sur l'utilisation d'un code d'autorisation que l'on envoie au serveur d'authentification. En retour, on reçoit un jeton d'accès que l'on pourra utiliser pour toutes nos requêtes vers l'API. Dans le cas de notre plateforme Viya, un code d'autorisation appelé `CONSUL_TOKEN` est fourni par la solution.

```
$ export CONSUL_TOKEN=`cat /opt/.../tokens/consul/default/client.token`  
54b09c41-5fub-41ce-a413-257a187dd7a1
```

Ce code d'authentification est envoyé via une requête POST à l'endpoint correspondant au serveur d'authentification.

```
$ curl -k -X POST "/SASLogon/oauth/clients/consul?serviceId=app"  
-H "X-Consul-Token: $CONSUL_TOKEN"
```

On reçoit alors un jeton d'accès (`access_token`) que l'on pourra utiliser pour s'identifier à l'API SAS et ainsi pouvoir enregistrer notre application au service Authentification de SAS Viya.

```
{"access_token":"eyJhbGciOiJSUzI1NiIsIm...","token_type":"bearer",  
"expires_in":35999,"scope":"uaa.admin", "jti":"de81c7f3cca645ac807f18dc0d186331"}
```

Pour enregistrer notre application de SAS Viya, on construit un objet json dans lequel on spécifie plusieurs éléments de configuration et qu'on envoie à l'API par une requête POST auprès de l'endpoint `/SASLogon/oauth/clients`.

```
$ curl -k -X POST /SASLogon/oauth/clients -H "Content-Type: application/json"  
-H "Authorization: Bearer $ACCESS_TOKEN."  
-d '{  
  "client_id": "dashboard",  
  "client_secret": "dashboardsecret",  
  "scope": ["openid"],  
  "authorized_grant_types": ["password","refresh_token"],  
  "redirect_uri": "http://dashboardserver.com:5000/"  
}'
```

Parmi les éléments de configuration importants, il y a le `"redirect_uri"` qui doit être l'URL de notre web application et qui sera l'URL vers laquelle l'utilisateur sera redirigé après s'être authentifié.

Une fois que l'application est enregistrée sur SAS Viya, on peut utiliser le service SASLogon pour s'y authentifier à l'aide de l'URL suivante :

```
https://sas.server.demo/SASLogon/oauth/authorize?response_type=code&client_id=dashboa  
rd&redirect_uri=http://dashboardserver.com:5000/
```

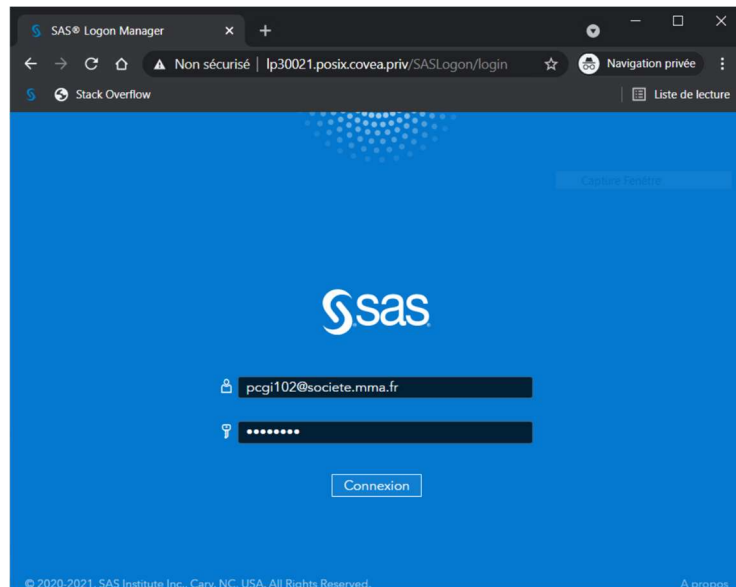


Figure 13 - Page d'authentification

4.1.2 Implémentation de l'authentification

Maintenant qu'on possède une URL d'authentification, il est nécessaire de l'implémenter dans la solution. Comme on l'a vu précédemment, le fichier `base.html` est hérité par tous les templates HTML de l'application. Ce fichier contient un mécanisme qui se sert de l'objet session fourni par Flask pour vérifier l'authentification de l'utilisateur. Ce mécanisme fonctionne de la manière suivante : si l'utilisateur n'a pas d'objet `access_token` dans son objet session, alors l'utilisateur est renvoyé vers l'URL d'authentification générée précédemment.

```
{% if 'access_token' not in session %}
<!-- l'utilisateur n'est pas authentifié -->
<script>
function logon(){
// construction de l'URL d'authentification à partir de la configuration
let auth_url ='{{config['VIYA_SERVER']}}SASLogon/oauth/authorize' +
'?response_type=code' +
'&client_id={{config['CLIENT_ID']}}' +
'&redirect_uri={{config['APP_URI']}}'
// redirection vers l'URL d'authentification
window.location.replace(auth_url);
}
</script>
</head>

<body onload="logon()">
<div>
</div>
</body>

{% else %}
<!-- l'utilisateur est authentifié -->
<!-- chargement du contenu de l'application -->
{% endif %}
```

Une fois que l'utilisateur s'est authentifié en renseignant ses identifiants par le biais de l'interface de l'URL d'authentification, celui-ci est redirigé vers l'URL de l'application avec une variable code dans l'URL.

```
http://dashboardserver.com:5000?code=Vx01bdp411
```

Ce code permet de prouver que l'utilisateur a réussi son authentification. Pour vérifier que ce code est valide côté serveur, il est nécessaire d'effectuer une requête au service SASLogon de l'API en lui mentionnant ce code. Si la réponse est positive, on reçoit un `access_token` de la part de l'API. À l'aide de cet `access_token` on récupère l'ID de l'utilisateur et on vérifie que cet utilisateur est bien un administrateur SAS.

Si le code d'authentification est valide et l'utilisateur est bien un administrateur SAS, on crée un objet `access_token` dans l'objet session de l'utilisateur et on actualise la page de l'utilisateur. Le mécanisme d'authentification présenté précédemment détecte qu'il y a bien un objet `access_token` dans l'objet session de l'utilisateur et le contenu de l'application est donc chargé.

```
# si il y a une variable code dans l'URL
if 'code' in request.args:
# on demande à l'API si l'authentification de l'utilisateur est valide
code = request.args['code']
headers = {'Accept': 'application/json',
'Content-Type': 'application/x-www-form-urlencoded'}
auth = (app.config['CLIENT_ID'], app.config['CLIENT_SECRET'])
data = "grant_type=authorization_code&code=" + code +
"&redirect_uri=" + app.config['APP_URI']
response = requests.post(app.config['VIYA_SERVER'] + "SASLogon/oauth/token/",
headers=headers, auth=auth, data=data)
auth_res = response.json()
# valide si il y a un objet access_token dans la réponse
if 'access_token' in auth_res:
# on obtient l'id de l'utilisateur à l'aide de son access_token
headers = {'Accept': 'application/json', 'Content-Type': 'application/json',
"Authorization": "Bearer " + auth_res['access_token']}
response = requests.get(app.config['VIYA_SERVER'] +
"identities/users/@currentUser", headers=headers)
user_res = response.json()
user_id = user_res['id'] # récupère l'id utilisateur
session['user_id'] = user_id

# on verifie que l'utilisateur est bien dans le groupe admin
# défini dans config.py
admin_users = getUsersFromGroupe(app.config['ADMIN_GROUP'])
isAuth = False
for user in admin_users:
if user['id'] == user_id:
isAuth = True
break

# on crée un objet access_token dans l'objet session
if isAuth:
session['access_token'] = auth_res['access_token']
```

4.2 Requête API

De nombreux appels à l'API sont effectués dans le cadre de l'application. Pour implémenter cette fonctionnalité, on utilise la fonction `callrestapi` fournie par la librairie `Pyviatools`. Cette fonction implémente les différentes fonctions de requête HTTP de la librairie `Requests` de Python. En paramètre on spécifie l'endpoint de l'API (`reqval`) auquel on veut accéder ainsi que le type de requête (`reqtype`) que l'on veut effectuer (GET, POST, PUT, DELETE).

```
def callrestapi(reqval, reqtype, acceptType='application/json',
contentType='application/json',data={},stoponerror=1):
# obtient l'url de l'api
baseurl=getbaseurl()
# obtient le jeton d'accès
oaval=getauthtoken(baseurl)
# inclut le jeton d'accès à la requête pour l'authentification
head= {'Content-type':contentType,'Accept':acceptType}
head.update({"Authorization" : oaval})
# appel à l'API suivant le type de requête
if reqtype=="get":
ret = requests.get(baseurl+reqval,headers=head,data=json_data)
elif reqtype=="post":
ret = requests.post(baseurl+reqval,headers=head,data=json_data)
elif reqtype=="delete":
ret = requests.delete(baseurl+reqval,headers=head,data=json_data)
elif reqtype=="put":
ret = requests.put(baseurl+reqval,headers=head,data=json_data)
else:
result=None
print("NOTE: Invalid method")
sys.exit()
# gestion des codes d'erreur
if (400 <= ret.status_code <=599):
# l'utilisateur est notifié de l'erreur
flash(ret.text)
result=None
if stoponerror: sys.exit()
# retourne le résultat de la requête
else:
try:
result=ret.json()
except:
try:
result=ret.text
except:
result=None
print("NOTE: No result to print")
return result
```

4.3 Configuration

Un des objectifs dans la conception de la solution est de la rendre "clef en main", c'est-à-dire qu'on peut l'installer et la configurer facilement. Pour se faire, on a centralisé tous les éléments de configuration dans un seul fichier (config.py) en essayant de réduire au maximum les éléments de configuration à spécifier :

```
# URL du serveur Viya
VIYA_SERVER='http://sas.server.demo/'
# URL de l'application
APP_URI='http://dashboardserver.com:5000/'
# clé secrète
SECRET_KEY=b'\x16\xcd\x00_\xa8\x0bC"\xfeX\xa4\xa1\x80\xacT5'
# id client de l'application enregistré sur SAS Viya
CLIENT_ID='dashboard-app'
# client secret
CLIENT_SECRET='???'
# id du groupe d'utilisateurs SAS autorisés à utiliser l'application web
ADMIN_GROUP='SASAdministrators'
# chemin de l'exécutable sas-admin
CLI_PATH='/opt/sas/viya/home/bin/sas-admin'
```

Concrètement, un utilisateur qui voudrait installer la solution devra d'abord enregistrer son application au service Authorization de SAS Viya (cette procédure est décrite dans la section 4.1) puis modifier le fichier config.py avec les éléments de configuration. Une fois ces étapes de faites, le serveur Flask peut être lancé et l'application est opérationnelle.

4.4 Gestion des erreurs et notifications

Dans le cadre de l'application, on souhaite faire un retour à l'utilisateur sur les actions d'administration qu'il effectue (messages de réussite et messages d'erreur).

Flask fournit un moyen très simple de faire un feedback à un utilisateur avec le système appelé flash. Le système de flash permet d'enregistrer un message à la fin d'une requête et d'y accéder à la requête suivante et uniquement à la requête suivante. Pour cela, il suffit d'utiliser la fonction flash en mettant comme paramètre le texte que l'on souhaite flasher. Pour accéder à ces messages, on utilise la fonction `get_flashed_messages()` qui renvoie une liste de ces messages. Cette fonction est implémentée dans le template `base.html` avec une boucle `for` sur les éléments de la liste. Ainsi, à chaque fois qu'on effectue une action, le ou les messages de retour seront obligatoirement affichés sur la page qui va succéder.

Dans notre fichier `base.html` :

```
{% with messages = get_flashed_messages() %}
{% if messages %}
<ul class="flashes alert alert-primary m-3" style="list-style: none;">
{% for message in messages %}
<li>{{ message }}</li>
{% endfor %}
</ul>
{% endif %}
{% endwith %}
```

Dans le cas de la création de deux groupes à l'aide du formulaire de création de groupes (voir figure 5), on obtient l'affichage suivant :



Groupe Test Group created.
Groupe Test Group 2 created.

4.5 Mise en cache

Lors de la conception des différentes fonctionnalités de l'application, il s'est avéré que certaines fonctions qui utilisent des appels API soient trop longues à s'exécuter.

Par exemple, il n'existe pas d'endpoint spécifique pour récupérer la liste complète des utilisateurs de la plateforme. Par contre il existe bien un endpoint pour récupérer la liste des utilisateurs d'un groupe donnée. Ainsi, pour implémenter la fonction qui renvoie la liste complète des utilisateurs de la plateforme il est d'abord nécessaire de récupérer les listes de tous les groupes puis de faire une requête pour chaque groupe pour récupérer leur liste d'utilisateurs pour enfin faire une concaténation sur ces résultats.

Ce procédé s'avère très long et peut prendre plusieurs minutes sur de gros environnements. Pour remédier à ce problème, il est nécessaire d'implémenter la mise en cache de ces fonctions. Concrètement, Flask propose une librairie externe appelée Flask-Caching qui permet de configurer un système de mise en cache sur son application.

```
cache = Cache(app, config={'CACHE_TYPE': 'FileSystemCache',
'CACHE_DIR': '../cache',
'CACHE_DEFAULT_TIMEOUT': 0,
'CACHE_THRESHOLD': 0})
```

On décide d'utiliser un cache de type FileSystem, les fichiers de cache sont alors stockés en dur dans un dossier spécifié. Après avoir initialisé le cache, on peut mettre en cache le résultat de n'importe quelle fonction en utilisant le décorateur `@cache.cache()` ou `@cache.memoize()`. Des options de mises en cache sont disponibles comme le timeout qui permet de définir le temps qu'on souhaite garder le résultat en cache. Le décorateur `@cache.memoize()` est un peu particulier, car il permet de garder le résultat d'une fonction suivant le paramètre qu'on lui a spécifié.

```
@cache.memoize()
# retourne la liste des utilisateurs pour un groupe donné
def getUsersFromGroupe(groupeId):
    endpoint="/identities/groups/"+groupeId+"/members
    method='get'
    result=callrestapi(endpoint,method)
    return result['items']

# retourne la liste des utilisateurs de la plateforme
def listUsers():
    userslist = []
    groupes = listGroupes()
    groupes_id = [g['id'] for g in groupes]
    for groupe in groupes:
        # cette fonction est mise en cache
        users = getUsersFromGroupe(groupe['id'])
        # concaténation des listes d'utilisateurs
        for user in users:
            if user not in userslist and user['id'] not in groupes_id:
                userslist.append(user)
    return userslist
```

Ici on ajoute le décorateur `@cache.memoize()` à la fonction `getUsersFromGroupe(groupeId)`. De cette façon, la liste des utilisateurs pour chaque groupe sera mise en cache et le temps d'exécution de la fonction `listUsers()` sera grandement amélioré.

4.6 Etat de la plateforme

La page d'accueil de l'application (voir figure 1) nous renseigne sur l'état général de la plateforme. Il existe aussi des pages spécifiques à l'état des services et des machines (voir figure 11) qui nous renseignent encore plus en détail sur l'état de ces ressources.

Pour vérifier l'état des services, on utilise l'API REST de Viya. Celle-ci comprend un endpoint qui permet d'obtenir rapidement la liste des services en cours. Concrètement, au lancement de l'application, on effectue une requête à l'API pour obtenir la liste de tous les services et ceux-ci sont enregistrés dans un fichier json. Ensuite, à chaque chargement de la page d'accueil, on effectue de nouveau une requête à l'API et on compare le résultat de cette requête avec le fichier json précédemment enregistré. Ainsi, on pourra détecter si des services sont down.

```
# retourne la liste des services en cours d'exécution
def getServicesStatus():
    endpoint='/monitoring/services/'
    method='get'
    result=callrestapi(endpoint,method)
    return result['items']

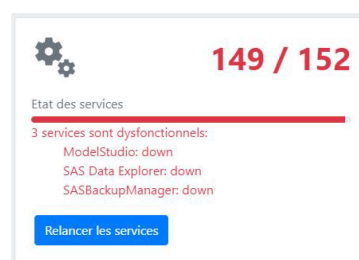
def init_services():
    services_file = Path("./viyadashboard/config/services_list.json")
    # si le fichier des services existe déjà
    if services_file.is_file():
        with open("./viyadashboard/config/services_list.json", "r") as jsonFile:
            services_recap = json.load(jsonFile)
            jsonFile.close()
        # retourne le contenu du fichier
        return services_recap
    else:
        services_recap = getServicesStatus()
        if isinstance(services_recap, list):
            with open("./viyadashboard/config/services_list.json", "w") as jsonFile:
                json.dump(services_recap, jsonFile)
                jsonFile.close()
            return services_recap

# au lancement du serveur Flask
services_init = init_services()
```

On implémente graphiquement le résultat de la comparaison des services up et des services down avec une jauge Bootstrap placée sur la page d'accueil :



Quand tous les services sont UP



Quand certains services sont DOWN

Concernant l'état des machines, on utilise aussi un endpoint API qui renvoie des informations sur l'état des machines. Parmi les informations renvoyées on a:

- HDD: entre dans l'état d'avertissement si l'utilisation de l'espace disque dépasse 95%
- RAM: entre dans l'état d'avertissement si l'utilisation de la mémoire RAM dépasse 95%
- serfHealth: agent vivant et joignable

Ces informations sont représentées dans un tableau:

Machines			
Nom	HDD	RAM	serfHealth
lp30018.posix.covea.priv	✓	✓	✓
lp30019.posix.covea.priv	✓	✓	✓
lp30020.posix.covea.priv	✓	✓	✓
lp30021.posix.covea.priv	✓	✓	✓
lp30022.posix.covea.priv	✓	✓	✓

Figure 14 - Tableau de l'état des machines

4.7 Suivi en temps reel

On souhaite suivre en temps réel l'état de la plateforme. Pour se faire, on actualise les informations de la page d'accueil dans un intervalle de temps donné. Cela est permis par la technologie AJAX qui permet d'effectuer des requêtes HTTP asynchrones côté client.

Concrètement, on utilise la fonction JavaScript `setTimeout()` qui permet d'exécuter une fonction à un intervalle de temps donné. Cette fonction est exécutée au chargement de la page `index.html` et à chaque intervalle elle effectue une requête AJAX vers le serveur Flask pour récupérer les informations actualisées sur l'état de la plateforme. Une fois récupérées, ces informations sont actualisées sur la page au moyen de la librairie JQuery qui facilite grandement les manipulations sur une page web.

Côté serveur (Python):

```
# requête de l'actualisation des informations de la page d'accueil
# (nombre de sessions, état des services, état des machines)
# utilisée par une requête AJAX dans index.html
# renvoie un dump json
@app.route('/refreshIndex')
def refreshIndex():
    sessions, connected_users = getActiveSessions(getConnectedUsers=True)
    ret = {'nb_sessions': len(sessions), 'service_status': getServicesRecap(),
          'machines': listemachines(), 'connected_users': len(connected_users)}
    return json.dumps(ret)
```

Côté client (JavaScript):

```
// mise à jour des informations sur l'état de la plateforme
function auto_refresh(){
    setTimeout(function(){
        // requête AJAX sur l'endpoint /refreshIndex vers le serveur Flask
        $.getJSON( "/refreshIndex", function( data ) {
            //on récupère l'objet data correspondant :
            // { 'nb_sessions': 52,
            //   'service_status': {...},
            //   'machines': {...},
            //   'connected_users': 12}
            // et on met à jour les informations sur le site avec JQuery
            $('#nbSessions').html(data['nb_sessions'])
            ...
        })
    }, 3000); // la fonction s'exécute toutes les 30 secondes
```

4.8 Processus en arrière-plan

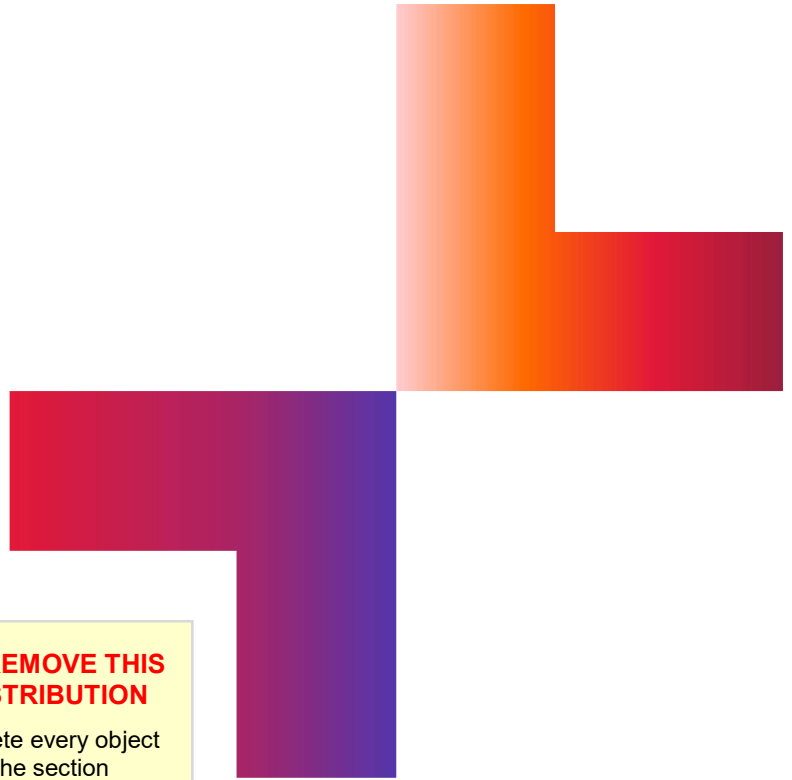
Certaines fonctionnalités nécessitent l'exécution de commandes shell comme le script de relance des services. Comme celui-ci peut durer plus de 15 minutes, il est nécessaire de vérifier son avancement et que l'utilisateur en soit informé.

Pour exécuter des commandes shell, on a implémenté la fonction `shellCommandHandler`. Cette fonction exécute la commande puis passe le processus qui correspond à la commande dans une liste `backgroundProcess`. Ensuite, à chaque fois qu'une page de l'application est chargée, on utilise la fonction `checkBackgroundProcess` pour vérifier le statut des processus en cours d'exécution et on utilise la fonction `flash` de Flask pour en informer l'utilisateur.

```
# à chaque requête HTTP, on vérifie le statut des processus en arrière-plan
@app.before_request
def before_request_callback():
    checkBackgroundProcesses()

# verifie l'etat des processus
def checkBackgroundProcesses():
    if backgroundProcesses['relanceServices']:
        poll = backgroundProcesses['relanceServices'].poll()
        if poll is None: # le processus n'est pas terminé
            flash("La relance des services est en cours.", "relanceServices")
        else: # le processus est terminé
            flash("Relance des services terminée.", "relanceServices")
            backgroundProcesses['relanceServices'] = None

# execute une commande système et flash le résultat
def shellCommandHandler(command):
    print("Executing command: "+command)
    result = subprocess.run(command, shell=True, stdout=subprocess.PIPE,
                             stderr=subprocess.PIPE, encoding='utf-8')
    flash(str(result.returncode) + " " +
          str(result.stdout) + " " +
          str(result.stderr))
```



**Back cover page – REMOVE THIS
NOTE PRIOR TO DISTRIBUTION**

To delete this page, delete every object
on the page and delete the section
break on the previous page.