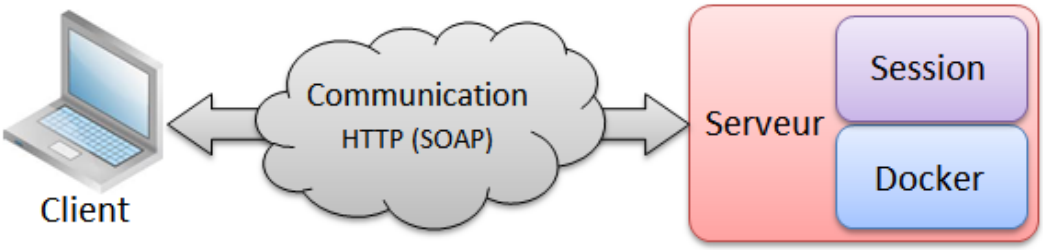
Mémoire du projet de programmation :

Plateforme de compilation et d'exécution à distance.



Salmane Bah, Tristan Braquelaire, Wassim Romdan, Timothée Sollaud

Université de Bordeaux

10 août 2014

Table des matières

Chapitre 1

Introduction

1.1 Contexte

Dans le cadre de l'UE *Projet de Programmation* (PdP), suivie durant notre formation de Master 1 Génie Logiciel à l'Université de Bordeaux, des sujets de projets nous ont été proposés par des clients du *Laboratoire Bordelais de Recherche en Informatique* (LaBRI). Après les avoir rapidement étudiés, nous avons contacté M.Guillaume Blin afin d'obtenir plus d'informations sur la *plateforme de compilation et exécution à distance sécurisée et basée sur Docker [Inc]* qu'il souhaitait réaliser. À la suite d'une réunion autour des grandes lignes du projet, il nous a chargé de la conception et de la réalisation de cette plateforme, sur laquelle nous avions particulièrement envie de travailler.

1.2 Présentation du sujet

M.Guillaume Blin, notre client, cherchait à mettre en place une plateforme d'apprentissage en ligne des langages de programmation. Pour son bon fonctionnement, un tel outil a besoin de compiler et d'exécuter du code écrit dans divers langages. C'est cette partie qu'il nous a proposé de concevoir et de réaliser, indépendamment de la plateforme d'apprentissage.

1.3 Étude de l'existant

Au début de notre période de travail sur le projet, la plateforme d'enseignement en ligne était en cours de développement par un groupe d'étudiants issus d'une école d'ingénieur. Ce centre d'apprentissage repose sur Moodle et propose une certaine forme d'interactivité avec l'étudiant (utilisateur). En effet, ce dernier doit écrire des portions de code dans le but de résoudre des défis. C'est pourquoi, afin de valider ou non le défi, il est nécessaire de compiler et d'exécuter le code proposé par l'étudiant. Plusieurs plateformes de compilation et d'exécution qui étaient susceptibles de répondre à ce besoin sont disponibles en ligne. Voici une liste non exhaustive des plus populaires :

- Ideone : `http://ideone.com/`
  Ideone prend en charge un grand nombre de langage et propose ses services via une "Application Programming Interface" (API) "Simple Object Access Protocol" (SOAP [Sud03]). Il offre à l'utilisateur la possibilité de télécharger le code source soumis ainsi que, pour les utilisateurs abonnés, de choisir la visibilité du code (public, privé ou partagé). Cependant, le nombre de requêtes gratuites est limité, de même que le temps de compilation (10 secondes), la durée d'exécution (5 secondes) et l'espace mémoire

(256Mo).

– Compileonline : `http://compileonline.com/`
Compileonline reconnaît un très grand nombre de langages et de métalangages et permet parfois, de traiter plusieurs fichiers. Ce service, en plus d'être entièrement gratuit, offre la possibilité de télécharger le code source et de configurer le mode d'édition de la console (type Emacs ou Vim). En revanche le temps de compilation et d'exécution est limité à un nombre indéterminé de cycles, au même titre que l'espace mémoire maximum pouvant être alloué à un projet, qui reste inconnu. De plus, l'architecture des programmes à compiler est contrainte par un nombre fixe de fichiers sources, typés selon le langage. Enfin, ce service ne met aucune API à disposition du public.

– Compilr : `https://compilr.com/`
Compilr intègre un IDE complet qui nécessite une inscription pour être utilisé.

La compilation et l'exécution du code étaient jusqu'alors déléguées à Ideone, dont un certain nombre de services répondant aux besoins sont disponibles gratuitement via une API. Cependant, la version gratuite étant limitée en terme de nombre de requêtes, il était bel et bien nécessaire de réaliser notre propre plateforme. Libre et gratuite, elle pourra être réutilisée pour divers projets.

Chapitre 2

Cahier des charges

Suite à nos entretiens avec le client, nous avons élaboré une liste des exigences, fonctionnelles et non-fonctionnelles, auxquelles la plateforme devait répondre. Certaines extensions non prioritaires sont signalées par la mention "optionnel".

2.1 Besoins fonctionnels

Voici les grandes fonctionnalités que notre plateforme de compilation et d'exécution à distance doit intégrer :

1. Établir une connexion
   - Le serveur doit permettre aux clients de se connecter.
   - Le serveur doit ouvrir et maintenir une session pour chaque client.
2. Transmettre les langages pris en charge
   - Le serveur doit permettre au client de connaître les langages qu'il peut traiter.
3. Compiler le code
   - Le serveur doit récupérer les paramètres de compilation ainsi que le code source, sous forme de chaîne ou de fichier (optionnel) et le langage transmis par un client.
   - Le serveur doit vérifier que la requête de compilation est bien valide.
   - Le serveur doit créer un fichier temporaire contenant le code source reçu.
   - Le serveur doit lancer une sandbox [Gal] (docker) dans laquelle il va compiler le fichier temporaire.
   - Le serveur doit retourner le résultat de la compilation au client ou un message d'erreur.
4. Exécuter le code (optionnel)
   - Le serveur doit lancer dans une sandbox (docker) l'exécution d'un projet déjà compilé.
   - Le serveur doit retourner le résultat de l'exécution au client ou un message d'erreur.
5. Compiler-exécuter le code
   - Le serveur doit récupérer les paramètres de compilation, les paramètres d'exécution ainsi que le code source, sous forme de chaîne ou de fichier (optionnel) et le langage transmis par un client.
   - Le serveur doit vérifier que le langage est bien pris en charge.
   - Le serveur doit créer le fichier temporaire qui contient le code source reçu.
   - Le serveur doit lancer une sandbox (docker) dans laquelle il va compiler le fichier temporaire.
   - Le serveur doit retourner le message d'erreur du compilateur si la compilation échoue.
   - Le serveur doit exécuter le fichier compilé.

   - Le serveur doit retourner le résultat de l'exécution au client ou un message d'erreur.
6. Fermer une connexion
   - Le serveur doit permettre aux clients de se déconnecter.
   - Le serveur doit supprimer le(s) fichier(s) temporaire(s) créé(s).
7. Gestion des services (optionnel)
   - Le serveur doit permettre à son administrateur de récupérer les langages qu'il (le serveur) peut compiler à tout moment.
   - Le serveur doit permettre à son administrateur d'ajouter, supprimer, modifier les langages et les compilateurs pris en compte.
   - Le serveur doit permettre à son administrateur de modifier les paramètres (limite mémoire, temps, ...) d'exécution/compilation des clients à chaud.

2.2 Besoins non fonctionnels

Nous avons également abordé, au cours d'un entretien avec le client, différentes contraintes appliquées au serveur, telles que la sécurité, la disponibilité et la flexibilité qui sont répertoriées dans les besoins non-fonctionnels suivants :

1. Sécurité
   - Le serveur doit gérer des sessions et assurer l'imperméabilité entres elles. (Un client ne doit pas être en mesure de se faire passer pour un autre client afin d'exécuter du code qui ne lui appartient pas.)
2. Disponibilité
   - Le serveur doit assurer un maximum de ressources disponibles pour de nouveaux clients potentiels. Il faut donc libérer toutes les ressources utilisées par un client dès lors qu'il est inactif depuis un temps donné ou qu'il se déconnecte. La durée d'inactivité avant déconnexion sera configurée par l'administrateur de la plate-forme.
   - Le serveur doit limiter le temps d'exécution afin de se prémunir des boucles infinies. Cette limite sera fixée par l'administrateur de la plateforme et pourra être fonction du langage (optionnel).
   - Le serveur doit limiter l'espace mémoire disponible pour l'exécution du code (cas de bombe fork, etc..). Cette limite sera fixée par l'administrateur de la plateforme et pourra être fonction du langage (optionnel).
3. Flexibilité & Fiabilité
   - (optionnel) Le système développé doit permettre de répartir le traitement des requêtes sur une multitude de serveurs traitant éventuellement un panel de langages différent selon les besoins.

Chapitre 3

Spécification

Dans ce chapitre, nous allons aborder les différentes solutions techniques que nous avons envisagées et justifier les choix que nous avons faits.

3.1 Un Web-Service

La plateforme attendue devant se comporter comme un serveur utilisable par divers client qui nous sont inconnus, nous avons conçu un web-service [Kre01]. Pour ce faire, nous avons choisi d'utiliser SOAP, qui est disponible via des bibliothèques dans un grand nombre de langages et repose intégralement sur des protocoles et standards ouverts, tels que Web-Service Description Language (WSDL). Nous nous sommes également intéressés à REST [Sun], que nous avons mis de côté parce qu'il offre moins de flexibilité. En effet, REST oblige au client la conservation locale de toutes les données nécessaires aux bon déroulement des requêtes, ce qui pose problème, par exemple lors d'une connexion au travers d'un proxy.

3.2 Le langage : Java

Après avoir hésité dans le choix du langage entre le C et le Java, nous avons finalement réalisé la plateforme en Java. En effet, si le C offrait un avantage en terme de performance, notamment dans la communication inter-processus avec Docker, il rendait toute la partie web-service (avec SOAP) bien plus délicate. Nous avons donc préféré le Java qui, malgré la perte de performance liée à la machine virtuelle associée, offre la bibliothèque Java API for XML Web Service (JAX-WS) [Fro12] facilitant l'implémentation d'un web-service SOAP grâce à l'utilisation d'annotations et à la génération automatique du WSDL. En outre, si par la suite on souhaitait optimiser la partie de traitement des tâches, Java offre la possibilité de s'interfacer avec du code C via Java Native Interface (JNI).

3.3 La standardisation des retours

Pour plus de clarté et dans le but de faciliter l'utilisation de notre API, nous avons choisi de "standardiser" les différents codes de retour pouvant êtres rencontrés. Pour ce faire, nous avons mis en place des énumérations et notre web-service retourne régulièrement des dictionnaires contenant une clé entière et une chaîne descriptive. Ainsi un développeur, travaillant autour de l'API, pourra relativement facilement gérer les résultats et erreurs reçues, tant en s'appuyant sur le code retourné pour définir une action spécifique dans le code, qu'en utilisant la chaîne associée pour informer l'utilisateur.

3.3.1 retour d'erreur (clé : retCode)

-1 error (erreur)
0 success (succès)
1 server overloaded (serveur surchargé)
2 unknown language (langage inconnu)
3 unknown reference (référence inconnue)
4 unknown key session (clé de session inconnue)
5 null pointer for source code (pointeur nul pour le code source)
6 null pointer for compilation option(s)(pointeur nul pour les options de compilation)
7 null pointer for command line argument(s) (pointeur nul pour les arguments de la ligne de commande)
8 nul pointer for standard input (pointeur nul pour l'entrée standard)

3.3.2 retour des résultats d'une tache (clé : result)

-1 error (erreur)
0 success (succès)
1 compilation error (erreur de compilation)
2 execution error (erreur d'exécution)
3 unknown (inconnu)

3.3.3 retour du statut d'une tache (clé : status)

-1 error (erreur)
0 done (traité)
1 on compilation (en compilation)
2 on execution (en exécution)
3 on interpretation (en interprétation)
4 waiting (en attente)

3.4 Description de l'API

3.4.1 Présentation générale

Afin de répondre aux besoins du client, nous avons défini une API au plus près de celle proposée par ideone [Ide], tout en ajoutant la possibilité d'exécuter plusieurs fois un code déjà envoyé et compilé. Cette nouvelle fonctionnalité nécessite d'inclure la gestion d'une session pour chaque client, obligeant ce dernier à transmettre sa clé de session pour la plupart des fonctions. Afin d'assurer la sécurité du serveur et d'empêcher un utilisateur B de manipuler la session d'un utilisateur A, cette clé se doit d'être unique et difficile à deviner. Au départ, nous avions imaginé un système reposant sur le couple adresse IP et port de communication du client, supposé unique pour une session. Cependant, nous étions confrontés à deux problèmes. D'une part, usurper l'identité d'un client était relativement aisé et d'autre part, certaines bibliothèques telle que celle de python changent de port à chaque requête, au cours de ce qui pour nous restait une même session. Nous avons donc opté pour le hash

d'une valeur certifiée unique. Ainsi à sa connexion, un client reçoit une clé unique qui lui est propre, qu'il doit fournir chaque fois qu'il souhaite agir sur sa session.

Concrètement, nous offrons la possibilité à un utilisateur de demander, à tout moment, la liste des langages pris en charge par notre serveur. Il peut ensuite se connecter afin d'ouvrir une session. Une fois connecté, il peut envoyer des requêtes de compilation suivies ou non d'une exécution. Dès lors qu'une tâche est compilée avec succès, il peut demander son exécution avec de nouvelles valeurs en entrée ou de nouveaux arguments. Il peut également interroger une tâche pour connaître son état ou récupérer les détails comme les logs ou le temps de compilation. Dans l'idéal, l'utilisateur rigoureux et consciencieux, doit se déconnecter afin que le serveur détruise les données liées à sa session. S'il ne le fait pas, le serveur fini par se délester automatiquement des données du client inactif depuis un temps fixé par l'administrateur.

Voici l'automate des possibles (cf.figure 3.1), lors de l'utilisation de notre API :

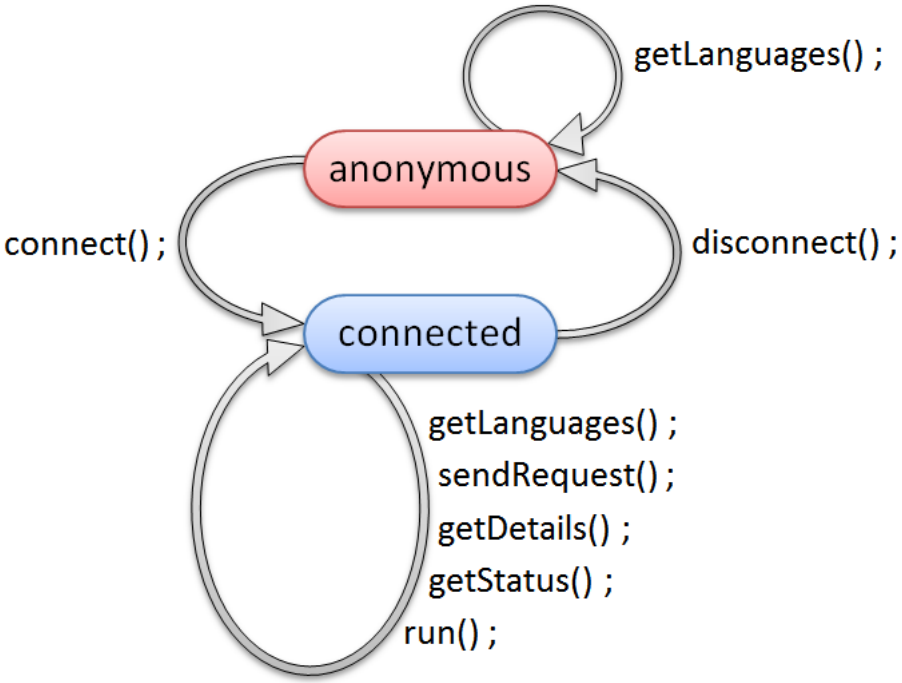


FIGURE 3.1 – API-Automate des possibles

3.4.2 Caractéristiques des méthodes proposées

Méthode	*getLanguages*		
Description	Retourne les langages supportés		
Paramètres	Aucun		
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	0	succès
	languages : dico	{1 : "C gcc", 2 : "Java javac"}*	dictionnaire des langages idLang : int (id du langage) details : String (nom & compilateur)

*Valeur arbitraire (dépend de la configuration du serveur).

Méthode	*connect*		
Description	Initialise une connexion avec le serveur		
Paramètres	Aucun		
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	0	succès
		1	serveur surchargé
	sessionKey : String	A03458...D4*	Clé de session unique

*Valeur arbitraire. Si retCode est à 1, la chaîne est vide.

Méthode	*disconnect*	
Description	Ferme une connexion avec le serveur	
Paramètres	sessionKey : String, clé de session (cf.connect())	
Type de retour	int	
Valeur de retour	**Valeur**	**Description**
	0	succès
	5	clé de session inconnue

Méthode	*sendRequest*		
Description	Soumet une requête de compilation (et exécution) au serveur		
Paramètres	**Nom**	**Type**	**Description**
	sessionKey	String	clé de session (cf.connect())
	codeLang	int	code du langage (cf.getLanguage())
	srcCode	String	code source à soumettre
	compilOpt	String	option de compilation
	cmdArgs	String	arguments d'exécution
	stdin	String	chaînes d'entrée standard
	toRun	boolean	active ou non l'exécution
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	-1	erreur
		0	succès
		1	serveur surchargé
		2	langage inconnu
		4	sessionKey inconnue
		5	source nulle
		6	options de compilation nulles
		7	arguments d'exécution nuls
		8	entrée standard nulle
	reference : int	1*	identifiant unique de la requête

*Valeur arbitraire relative au client. Vaut -1 si retCode n'est pas à 0.

Méthode	*run*		
Description	Exécute le code compilé de la tache référencée		
Paramètres	**Nom**	**Type**	**Description**
	sessionKey	String	clé de session (cf.connect())
	reference	int	reference de la tache (cf.sendRequest)
	cmdArgs	String	arguments d'exécution
	stdin	String	chaînes d'entrée standard
Type de retour	int		
Valeur de retour	**Valeur**	**Description**	
	-1	erreur	
	0	succès	
	3	référence inconnue	
	4	sessionKey inconnue	
	7	arguments d'exécution nuls	
	8	entrée standard nulle	

Méthode	*getDetails*		
Description	retourne les informations associées à une requête déjà soumise		
Paramètres	**Nom**	**Type**	**Description**
	sessionKey	String	clé de session (cf.connect())
	reference	int	reference de la tache (cf.sendRequest)
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	-1	erreur
		0	succès
		3	référence inconnue
		4	sessionKey inconnue
	status : int	-1	erreur
		0	succès
		1	en compilation
		2	en exécution
		3	en interprétation
		4	en attente
	result : int	-1	erreur
		0	succès
		1	erreur de compilation
		2	erreur d'exécution
		3	indéterminé
	stdOut : String	""	Sortie produite par la compilation ou l'exécution
	stdErr : String	""	Sortie d'erreur produite par l'exécution
	timeInfo : float	""	info sur la durée d'exécution

Méthode	*getStatus*		
Description	retourne l'état de la requête référencée		
Paramètres	**Nom**	**Type**	**Description**
	sessionKey	String	clé de session (cf.connect())
	reference	int	reference de la tache (cf.sendRequest)
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	-1	erreur
		0	succès
		3	référence inconnue
		4	sessionKey inconnue
	status : int	-1	erreur
		0	succès
		1	en compilation
		2	en exécution
		3	en interprétation
		4	en attente
	result : int	-1	erreur
		0	succès
		1	erreur de compilation
		2	erreur d'exécution
		3	indéterminé

3.5 Paramétrage du serveur

L'administrateur de la plateforme peut configurer plusieurs paramètres via des arguments sur la ligne de commande. Pour simplifier la manipulation des paramètres, nous avons utilisé le package Jcommander qui permet d'analyser efficacement et facilement les arguments passés au lancement du programme. L'administrateur peut ainsi utiliser les options suivantes :

--dockerImg pour indiquer l'image docker à utiliser.
--dockerExecTime pour indiquer le temps maximal d'exécution de docker (en seconde).
--dockerMemLimit pour indiquer la taille mémoire limite allouée à docker. [1]
--sessionCleanerTime pour indiquer la fréquence de déclenchement du délestage des sessions inactives (en minute).
--taskCleanerTime pour indiquer la fréquence de déclenchement du délestage des clients inactifs (en minute).
--sessionMaxInactivity pour indiquer la période d'inactivité après laquelle une session peut être délestée (en minute).
--taskMaxInactivity pour indiquer la période d'inactivité après laquelle une tache peut être délestée (en minute).
--threadsNumber pour indiquer le nombre de connexions simultanées supportées.
--help pour afficher l'aide

Le serveur affiche par défaut des informations sur son activité sur la sortie standard. L'administrateur peut alors aisément créer un fichier log en redirigeant la sortie standard du programme vers le fichier de son choix.

3.6 Sécurité du serveur

Le serveur étant destiné à compiler et exécuter du code distant potentiellement dangereux, il est nécessaire de garantir son intégrité. Pour cela, nous devions cloisonner les environnements de compilation et d'exécution. Nous avons donc mis en place un principe de sandbox. Nous aurions pu utiliser des machines virtuelles, mais pour réellement séparer chaque tâches, il aurait fallu charger autant de machines que de tâches, ce qui aurait été très lourd pour le système hôte. C'est pourquoi, nous avons utilisé le service docker. Basé sur les conteneurs [Hel09] Linux, il permet de créer des espaces clos légers, reposant sur le noyau du système hôte. La configuration d'une image docker se fait via un fichier de configuration, appelé "dockerfile". On y définit l'image du système de base à utiliser, à laquelle on ajoute les compilateurs et interpréteurs gérés par notre plateforme.

La commande de lancement de docker (docker run) offre de nombreuses options permettant de restreindre les actions du conteneur sur l'hôte. Ainsi, nous utilisons l'argument "-m" afin de limiter la taille mémoire maximale occupée par le conteneur et limiter l'impacte d'une bombe fork. Pour éviter que l'utilisateur transmette un code malveillant, cherchant à utiliser les failles du système d'exploitation de l'hôte, nous spécifions l'option "-n" qui ferme l'accès au réseau, tout en conservant la boucle locale au conteneur (localhost) qui nous permet d'exécuter un code client serveur. Pour filtrer les actions malveillantes telles que la suppression de répertoires dans le conteneur, docker est lancé en tant que "nobody" via l'option "–user=nobody". Enfin, le serveur se prémunit des boucles infinies via la commande unix "timeout", assurant la fermeture du processus docker après un temps défini par l'administrateur du service.

1. **Attention :** afin que cette option soit prise en charge par le serveur, il faut ajouter la ligne *GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"* dans le fichier */etc/default/grub* puis exécuter la commande *update-grub* et redémarrer le système pour appliquer les modifications.

Chapitre 4

Architecture

Notre solution se découpe en trois ensembles répartis en trois packages Java (cf.figure 4.1) que nous allons détailler dans ce chapitre. Les diagrammes des classes de chacun des packages sont disponibles en annexe.

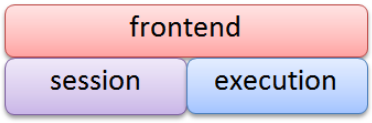


FIGURE 4.1 – Liens entre les packages de la solution

4.1 Le package "frontend"

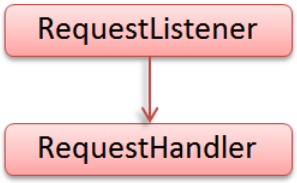


FIGURE 4.2 – Les classes métiers du package frontend

Ce package est la partie la plus visible du serveur. C'est ici qu'est présente l'implémentation du web-service.

La classe "RequestListener" implémente l'interface "RequestListenerInterface" qui définit toutes les méthodes offertes par l'API de la plateforme.

La classe "RequestHandler" implémente l'interface "RequestHandlerInterface" et se charge tout d'abord, de vérifier que les requêtes provenant des clients sont bien formées (pas de paramètres nuls, de clé de session inconnue , de référence à une tâche inexistante, de code langage non valide, etc.). C'est au sein du "RequestHandler" que l'interaction avec le package session va se faire via la classe "ClientsManager" afin au besoin, de générer une nouvelle session ou simplement, d'ajouter une nouvelle tache à un client existant.

La classe "EntryPoint" est le point d'entrée du serveur, c'est notamment ici qu'est publié le web-service. Nous avons utilisé des javaBeans pour représenter les retours de chaque

méthode du web-service. Ceci permet à l'API JAX-WS de construire un objet adéquat pour chaque web-méthode. Par exemple, la méthode *getLangagues()* dispose d'un objet représentant le retour de son appel (précédemment défini dans la spécification de l'API), tel qu'il sera vu par le client. Tous ces objets représentant un retour de web-méthode possèdent un attribut *InfoKey* indiquant au client si la requête s'est effectuée correctement ou non (on y trouve le code de retour et sa signification).

4.2 Le package "session"

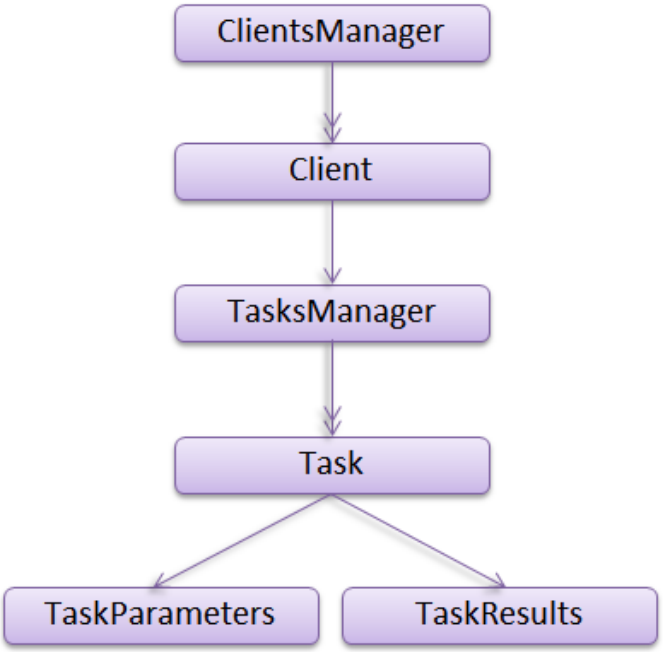


FIGURE 4.3 – Les classes métiers du package session

Le package session réunit toutes les classes permettant la création des objets temporaires liés à l'activité des clients sur le serveur.

Ainsi, on retrouve la classe "ClientsManager" qui implémente l'interface "ClientsManagerInterface", définissant les services permettant l'ajout, la suppression et la récupération des données d'un client. Pour pouvoir rechercher rapidement un "Client", le "ClientsManager" implémente une table de hachage (HashMap) indexant les clients par leur clé de session.

La classe "Client" implémente l'interface "ClientInterface" qui offre un contrat assurant de pouvoir récupérer les données caractérisant une session ainsi que son activité. Chaque client est identifié par sa clé de session (attribut *sessionKey*) et génère sa propre arborescence physique sur le serveur. La destruction correcte et complète de cette dernière est effectuée par une méthode *kill()*. Enfin, chaque client délègue la gestion de ses tâches à un objet dédié qu'il instancie : le "TasksManager".

La classe "TasksManager" implémente l'interface "TasksManagerInterface" assurant un service minimum relativement similaire à celui de "ClientsManagerInterface". On retrouve, ainsi l'ajout et la suppression de tâches, ou encore la recherche rapide d'une tâche. Ici, la liste des tâches d'un client est simplement une simple liste indexée par un entier ("arrayList")

ainsi, chaque tâche ajoutée est identifiée par son index dans la liste. Lors de la suppression d'une tâche, la référence de l'index correspondant pointe vers nulle, afin d'éviter que la liste ne soit ré-indexée. Nous ne nous sommes pas contenté d'utiliser un tableau d'objet implémentant l'interface définissant une tâche afin d'offrir plus de liberté lors du développement éventuel d'un nouveau "TaskManager".

La classe "Task" implémente l'interface "TaskInterface" qui offre la possibilité de connaître le propriétaire d'une tâche et le chemin physique vers le code source, ainsi que de déterminer si elle a été correctement compilée et enfin, de savoir à quand remonte sa dernière activité. Une méthode *kill()* permet de détruire correctement une tâche, en supprimant l'arborescence physique associée. Enfin, les données plus caractéristiques de la tâche sont réparties dans deux classes : "TaskParameters" et "TaskResults".

La classe "TaskParameters" implémente l'interface "TaskParametersInterface" qui propose un accesseur en lecture seule à tous les paramètres de la tâche. On y trouve donc : le code source, le langage, une indication sur la demande d'une exécution possible suivant la compilation, les éventuelles options de compilation ou encore, les arguments d'exécution et l'entrée standard à utiliser. Ces derniers paramètres, propres à l'exécution, sont dotés d'un accesseur en écriture afin de permettre le bon traitement de la commande *run()* définie dans notre API.

La classe "TaskResults" implémente l'interface "TaskResultsInterface" qui met à disposition des accesseurs en lecture et écriture sur tous les résultats suivant le traitement d'une tâche (sortie standard, sortie d'erreur, statistiques, etc.). Des classes d'énumérations (EnumStatus et EnumResult) ont été définies afin de mettre en œuvre la standardisation des retours abordée dans le chapitre sur la Spécification.

4.3 Le package "execution"

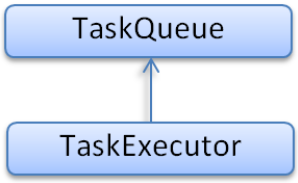


FIGURE 4.4 – Les classes métiers du package execution

Le package execution réunit les classes responsables de l'accomplissement des tâches clientes et donc, de la communication avec docker.

La classe "TaskQueue" implémente l'interface "TaskQueueInterface" qui définit toutes les méthodes nécessaires à la gestion d'une file d'attente. Cette file d'attente a été ajoutée en vue du développement de la version 2, décrite dans le chapitre "perspectives" de ce mémoire.

La classe "TaskExecutor" implémente l'interface "TaskExecutorInterface" dont la méthode principale est nommée *executeTasks()*. L'implémentation de cette méthode va piocher la prochaine tâche à traiter dans le "TaskQueue" instancié au préalable et lancer un environnement docker adapté et sécurisé, selon les paramètres proposés par le constructeur de cette classe (limite du temps d'exécution et de l'allocation mémoire de docker). Afin de réaliser le traitement approprié sur la tâche, la méthode *executeTasks()* va chercher les informations contenues dans l'objet de type défini "TaskParametersInterface" et fournit ensuite les résultats du traitements dans l'objet de type défini "TaskResultsInterface".

4.4 Bilan

L'architecture que nous avons mise en place offre une flexibilité importante. En effet, reposant essentiellement sur des interfaces, un développeur pourra aisément remplacer un module existant par une création qui lui sera propre, simplement en prenant soin d'implémenter l'interface liée.

L'architecture actuelle présente cependant une fuite en terme de performance. Dans cette version, tout est sur un seul et unique serveur physique et nous n'utilisons qu'une seule instance de docker. Malgré cela, nous avons d'ores et déjà mis en place une file d'attente pour les tâches. En effet, si aujourd'hui nos packages sont fortement liés, du fait que c'est le "RequestHandler" qui ajoute une tache à la file avant d'appeler l'unique serveur docker à s'exécuter, nous savons que le client souhaite à terme, offrir une multitude de serveur docker sur des machines physiquement différentes. C'est pour cette raison que nous avons choisi de mettre en place dès maintenant cette file d'attente intermédiaire.

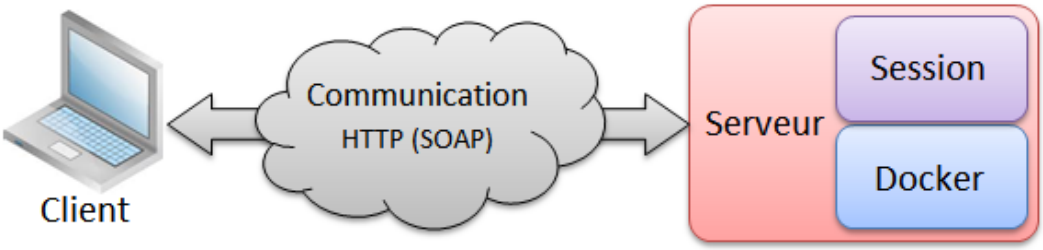


FIGURE 4.5 – Schema bilan de l'architecture

Chapitre 5

Tests

Dans ce chapitre, nous allons aborder la stratégie de test que nous avons mise en place. Elle repose sur trois grandes phases : des tests unitaires, la validation des fonctionnalités et une analyse non-fonctionnelle.

5.1 Les tests unitaires

Pour nous assister dans nos tests unitaires, nous avons utilisé l'API JUnit. Nous avons orienté nos tests sur chaque méthode des classes "Client", "Task", "ClientsManager" et "TasksManager", stockant les données métiers. Nous avons également vérifié le bon fonctionnement de la classe "TaskQueue" du package execution.

Dans un soucis de clarté, nous avons regroupé les tests dans un package test dédié. La classe "TestSuite1" permet d'exécuter successivement une suite de tests validant l'intégralité des classes testées.

Ces tests, dont le détail est disponible en annexe, nous ont permis de révéler quelques erreurs de programmation et incohérences :

1. – **Problème rencontré :** des dates supposées représenter des durées étaient erronées.
   – **Cause :** incohérence dans le type choisi. Nous cherchions à représenter une durée exprimée en année, mois, jour heure, minute et secondes à l'aide du format java "Date". Cela n'avait pas de sens puisque la Date représente un point dans le temps par rapport au 1er Janvier 1970, et en aucun cas une durée.
   – **Résolution :** nous avons réglé le problème en utilisant les valeurs (de type Long) des "timestamp" en milliseconde dont la différence offre bel et bien la durée souhaitée.

2. – **Problème rencontré :** les chemins des répertoires créés puis recherchés étaient incorrects.
   – **Cause :** erreur dans le choix de l'attribut utilisé lors de l'appel à la classe Java File.
   – **Résolution :** nous avons simplement remplacé les occurrences de "File.pathSeparator" en "File.separatorChar".

3. – **Problème rencontré :** "nullpointerexception" dans la méthode *removeTask()* de la classe "TasksManager"
   – **Cause :** dans la liste des tâches, on écrasait la référence vers la tâche à détruire avant d'appeler la méthode *kill()* de la classe "Task". Or, cette dernière méthode chargée de détruire l'arborescence liée à la tâche, cherche à reformer le chemin vers la tâche

à l'aide de l'identifiant normalement retourné par le "TaskManager". Cet identifiant correspond normalement à l'index de la liste où la tâche est trouvée. Or, la tâche ayant été retirée, l'index correspondant qui reste introuvable, lève une exception.
- **Résolution :** nous avons résolu le problème simplement en permutant les instructions. Ainsi, l'arborescence de la tâche est détruite puis la référence vers la tâche est mise à "null".

5.2 Les tests fonctionnels

Afin de tester les différentes fonctionnalités offertes par notre serveur de compilation, nous avons écrit un client python (test-plateform.py) utilisant notre API SOAP via la bibliothèque python "suds". Celui-ci nous a permis de tester divers scénarii. Nous avons ainsi pu vérifier que tous les retours attendus selon les requêtes transmises sont en cohérence avec la spécification précédemment établie. Un problème de conception a pu être résolu dès notre premier test :

- **Problème rencontré :** La session d'un client semblait ne pas perdurer après l'envoi d'une requête *connect() ;*.
- **Cause :** Notre premier système d'identification du client reposait sur une clé entière, incrémentée pour chaque nouvelle session. Afin de nous prémunir de l'usurpation trop évidente d'une telle identité, nous vérifions en interne que la requête transmise provenait bien du couple, théoriquement unique, adresse IP & port du client, utilisé lors de l'ouverture de la session. Or, il s'avère que l'API SOAP proposée par python, ouvre une connexion via un port différent pour chaque requête transmise. Il était donc normal que la session ne puisse pas être reconnue.
- **Résolution :** Nous avons repensé le système d'identification pour appliquer uniquement une clé de session complexe à deviner. Nous utilisons donc un hash unique, qui n'est plus corrélé, ni à l'IP, ni au port du client.

Un problème a été mis en évidence lors de l'exécution d'un code utilisant le paramètre d'entrée standard :
- **Problème rencontré :** Nous ne parvenions pas à rediriger correctement l'entrée standard du processus docker créé pour exécuter le code de la tâche : le serveur attendait que des caractères soient entrés au clavier.
- **Cause :** Le pipe d'entrée dans lequel on recopie la chaîne d'entrée standard passée en paramètre (stdIn) n'avait pas été fermé.
- **Résolution :** La fermeture du pipe a suffit a corriger le problème.

Après avoir testé toutes les fonctionnalités de base, nous avons compilé un code source client-serveur en Java, dont les classes ont été regroupées dans un unique fichier passé sous forme de chaîne. Le *main()* qui s'est naturellement exécuté a instancié un thread client et un thread serveur afin de transmettre une chaîne de caractère. Ce test s'est avéré parfaitement fonctionnel et prouve qu'il est possible d'utiliser la boucle locale (localhost) de l'environnement docker.

5.3 Les tests non-fonctionnels

Les performances du serveur ont été étudiées en réseau local grâce au client python (test-platform.py). Pour cela, il demande la compilation, puis l'exécution, du même code source écrit dans plusieurs langages (C, C++, Java et Python.). On récupère pour chaque cas le temps de transfert de la requête, le temps de compilation ainsi que le temps d'exécution. Voici le tableau statistique obtenu :

	Temps de transfert	Temps de compilation	Temps d'exécution
Java	13	1236	544
C	15	577	456
C++	15	576	479
Python	8	525	522
Moyennes (ms)	12.75	728.5	500.25
Répartition du temps (%)	1.03	58.68	40.29

FIGURE 5.1 – Statistiques

Chapitre 6

Perspectives

L'architecture multithreadée du projet actuel permet de lancer simultanément la même image docker et donc d'exécuter, en parallèle et sans encombre, les tâches de différents utilisateurs. Cependant, nous avions pensé à la mise en place de plusieurs améliorations, afin d'optimiser l'utilisation des services. Ces versions n'ont malheureusement pas pu êtres implémentées par manque de temps.

6.1 Version 2 : Multi-Serveur

La première amélioration que nous pourrions apporter est la gestion de plusieurs serveurs physiques d'exécution. Chacun serait alors capable de charger sa propre image docker prenant en charge un ensemble de langages choisi parmi tous les langages gérés par la plateforme, afin de mieux répartir la charge. De plus, cela limiterait la taille de l'image docker actuelle qui est relativement lourde, puisqu'elle contient les compilateurs et interpréteurs de chaque langage géré. Grâce à ce système, on pourrait offrir plus ou moins de serveurs pour le traitement lié à un langage donné, selon sa fréquence d'usage dans les requêtes des clients. Ce choix d'implémentation implique alors de ne plus utiliser les objets "Task" directement par référence mais de mettre en place une communication avec les différents serveurs afin de leur demander d'effectuer les traitements à réaliser sur le code et de nous retourner les diverses informations résultantes.

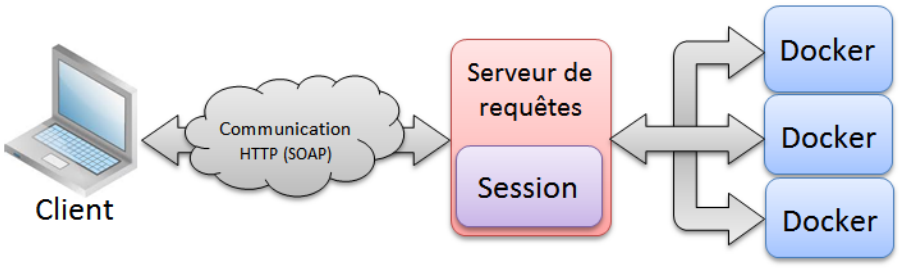


FIGURE 6.1 – Diagramme de l'architecture de la deuxième version du système

La simple file d'attente alors utilisée jusqu'à présent serait ici augmentée vers un objet plus complet : "TaskQueueManager". Son rôle principal serait de répartir efficacement les tâches sur les différents environnements docker à sa disposition. Le "TaskQueueManager" représenterait l'ordonnanceur des tâches et serait le cœur algorithmique de notre projet. Nous avions envisagé plusieurs stratégies de distribution possibles :

1. L'utilisation d'une file globale (les serveurs de compilation viendraient chercher des tâches à exécuter).
   Avantages :
   – Simple à implémenter
   – Peu de travail pour le "TaskQueueManager"
   Inconvénients :
   – Beaucoup de synchronisation
   – Complexité en O(n)

2. L'utilisation d'une file principale dans le "TaskQueueManager" recevant les requêtes du "RequestHandler", afin de répartir les tâches sur chaque serveur selon le(s) langage(s) pris en compte. Chaque serveur possèderait sa propre file de tâches.
   Avantages :
   – il n'y aurait plus de synchronisation entre les différents serveurs de compilation/exécution
   Inconvénients :
   – utilisation mémoire : une file par serveur, une file principale

3. L'utilisation d'une file principale dans le "TaskQueueManager" recevant les requêtes du "RequestHandler", afin de les répartir dans de nouvelles files selon le langage nécessaire à leur exécution. Les serveurs docker viendraient demander une tâche au "TaskQueueManager", qui examinerait les langages pris en charge par le docker demandeur, pour lui fournir la tâche adéquate.
   Avantages :
   – synchronisation réduite pour les serveurs de compilation par rapport à la stratégie 1 (seuls les serveurs concernés par un langage se synchroniseraient)
   Inconvénients :
   – utilisation mémoire : une file principale, une file pour chaque langage (qui augmenterait à l'ajout de langage)

4. L'utilisation d'une file principale dans le "TaskQueueManager" recevant les requêtes du "RequestHandler". Si les tâches associées concernaient des langages traités par plusieurs serveurs docker, elles seraient envoyées dans une liste principale. Sinon, elles seraient envoyées dans la file de l'unique serveur docker traitant le langage de la tâche. Le "TaskQueueManager" observerait alors en continue l'activité des serveurs docker, afin de leur donner les tâches adéquates dès lors qu'ils seraient de nouveau disponibles.
   Avantages :
   – Bonne répartition des tâches concernant les langages les moins utilisés sur les différents serveurs de compilation
   – Équilibre de la charge de travail entre "TaskQueueManager" et les serveurs de compilation/exécution
   Inconvénients :
   – Synchronisation (quoique faible) sur la file principale pour les langages communs aux serveurs de compilation/exécution

6.2 Version 3 : Administration à chaud

Pour faciliter la maintenance et l'évolution des services proposés, nous avions prévu une troisième version, donnant la possibilité à l'administrateur d'intervenir sur le serveur à chaud, via une API réservée. Cette interface aurait pour but de permettre à l'administrateur d'ajouter, modifier ou supprimer, les différents langages et leurs compilateurs ainsi que de gérer la création (et la suppression) des images dockers sur différents serveurs en définissant les langages disponibles dans chacune d'elles.

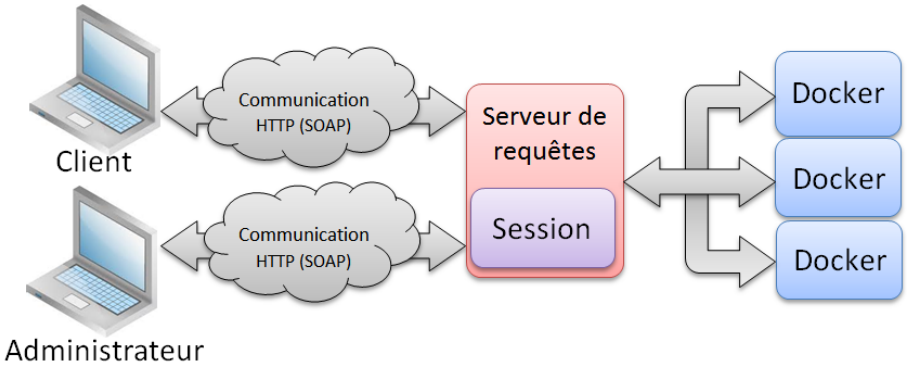


FIGURE 6.2 – Diagramme de l'architecture de la troisième version du système

Pour cela, nous ajouterions l'objet "AdminRequestHandler" qui offrirait un port d'écoute dédié aux requêtes d'administration. Ces dernières agiraient directement sur l'objet "SystemManager" qui serait chargé de mettre en place les modifications au niveau des dockers et d'en informer le "TaskQueueManager", afin qu'il puisse répartir convenablement les tâches. Ce système serait bien entendu protégé par une solide authentification.

Enfin, l'implémentation actuelle des langages devrait également être modifiée pour permettre à l'administrateur d'agir à chaud, une classe "LangagesManager" devrait être créée pour remplacer l'actuelle implémentation statique du tableau dans le "TaskExecutor". Cette liste de langages pourrait être sauvegardée à chaque modification sur le serveur en format XML via une méthode de la classe "LangagesManager". Cette classe devrait également proposer une méthode de lecture du fichier XML afin de reconstruire la liste des langages gérés au redémarrage du serveur.

Chapitre 7

Conclusion

Ce projet proposé par M. Guillaume Blin, concernant le développement d'un service de compilation et d'exécution de code à distance fut très intéressant, tant dans la conception que dans la réalisation de la solution. En effet, nous avons à la fois approfondi et mis en pratique de nombreuses notions abordées durant nos parcours divers. Ce fût également l'occasion de découvrir de nouveaux outils et de goûter à la gestion de projet dans un travail collaboratif. Nous n'avions encore jamais travaillé dans un groupe aussi important. Il nous a donc fallu apprendre à nous organiser, pour la répartition des tâches et l'adaptation de nos méthodes de travail. Nous veillions particulièrement à ce que la relecture de nos travaux soit aisée pour les autres membres du groupe et écrivions donc le code de manière claire et standardisée. Évidement, étant plus nombreux, cela nous a permis de confronter plusieurs points de vue sur notre projet et ainsi d'avancer dans une direction, façonnée par les critiques de chacun et donc finalement bien plus précise que si nous étions seul.

Pour certain, le terme même d'API était nouveau et nous avons dû nous renseigner sur ce sujet tout comme l'usage d'un web-service. Nous avons découvert le protocole SOAP, l'existence des contrats WSDL et avons appris à utiliser la bibliothèque JAX-WS. Ce projet nous a permis d'approfondir notre maîtrise de Java avec la découverte de concepts plus avancés comme les annotations, la Javadoc ou les classes JavaBean. Nous avons également pu mettre en pratique quelques notions de programmation objet vues en cours, comme la flexibilité du code avec l'implémentation d'interfaces. Des notions de programmation système et de sécurité ont également été présentes dans ce projet. Tout d'abord au travers du service Docker qu'il nous a fallu étudier afin d'établir la communication avec celui-ci en veillant à se prémunir des agressions potentielles. Ce projet a également amélioré notre maîtrise de l'environnement de développement Éclipse, couplé à un dépôt SVN. De plus, nous avons eu l'occasion de réaliser des tests unitaires via l'API JUnit, que nous avions découvert lors de notre enseignement de programmation objet en Java, au semestre précédent.

Enfin, cette UE nous a offert un aspect totalement nouveau dans notre formation universitaire : la communication avec un client. Spécifier les besoins, déterminer la faisabilité et veiller à rester le plus fidèle possible aux attentes du client, ont fait du déroulement de ce projet, une expérience inédite qui nous sera très utile par la suite. Nous avons notamment retenu l'importance de prendre le temps de bien mener les phases d'analyse et de conception afin d'optimiser le temps d'implémentation qui s'en suit. Le contact régulier avec le client est également un élément clé pour établir rapidement une spécification correcte.

Nous regrettons cependant de n'avoir pas eu le temps de tester plus en détail notre implémentation, et surtout de ne pas avoir pu atteindre la deuxième version de ce projet. Néanmoins nous avons pris plaisir à réaliser cette plateforme qui, malgré certaines améliorations pouvant y êtres apportées, nous semble être fonctionnelle et réutilisable.

Bibliographie

[Fro12] Annick Fron. *Architectures réparties en Java*. DUNOD, 2e edition, chapitre 9, pages 171–216 edition, 2012.

[Gal] Sarah Fister Gale. Creating a secure 'sandbox' on employee devices. `http://www.workforce.com/articles/creating-a-secure-sandbox-on-employee-devices`. Accédée : 07/04/2014.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[Hel09] Matt Helsley. Lxc : Linux container tools. Technical report, IBM, 2009.

[Ide] Ideone. ideone api specification. `https://ideone.com/files/ideone-api.pdf`. Accédée : 07/04/2014.

[Inc] Docker Inc. Learn what docker is all about. `http://www.docker.io/the_whole_story/`. Accédée : 07/04/2014.

[Kre01] Heather Kreger. Web services conceptual architecture (wsca 1.0). Technical report, IBM, 2001.

[Mic] Microsoft. How rpc works. http ://technet.microsoft.com/en-us/library/cc738291 Accédée : 07/04/2014.

[SH] dotcloud Solomon Hykes. The future of linux containers. `http://youtu.be/wW9CAH9nSLs`. Accédée : 07/04/2014.

[Sud03] Brian Suda. Soap web services. Master's thesis, University of Edinburgh, 2003. `http://suda.co.uk/publications/MSc/brian.suda.thesis.pdf`, accédée : 07/04/2014.

[Sun] Bruce Sun. A multi-tier architecture for building restful web services. `http://www.ibm.com/developerworks/library/wa-aj-multitier/`. Accédée : 07/04/2014.

Annexes

- Description des tests unitaires.
- Diagramme de classe du package frontend.
- Diagramme de classe du package session.
- Diagramme de classe du package execution.

Spécification des tests unitaires

Pour l'ensemble des tests suivants, nous considérons -1 et la chaîne vide (""), comme étant respectivement un numéro de requête invalide et une clé de session invalide.

fonction *connect()*

1. **appel :** connect(1)
   retour attendu : message erreur de SOAP : connect() ne prend pas de paramètre.
2. **appel :** connect()
   retour attendu : retCode=0 ; <sessionKey>
   appel : connect()
   retour attendu : retCode=0 ; <sessionKey>

fonction *disconnect(sessionKey)*

1. **appel :** disconnect()
   retour attendu : message erreur de SOAP : connect() requiert un paramètre.

1. **appel :** disconnect(-1)
   retour attendu : retCode=5

2. **appel :** disconnect(<sessionKey>)
   retour attendu : retCode=0

fonction *getLangages()*

1. **appel :** getLangages(1)
   retour attendu : message erreur de SOAP : getLangages ne prend pas de paramètre.

1. **appel :** getLangages()
   retour attendu : retCode=0 ; <dictionnaire des langages>.

fonction *sendRequest(sessionKey, codeLang, srcCode, compilOpt, cmdArgs, stdin, toRun)*

On attribuera respectivement aux entiers 1 et -999, au langage C et à un langage non attribué.

- code_bon_C : code en langage C ne posant pas de problème de compilation
- code_faux : code en langage C ne compilant pas
- code_1arg_bon : code qui requiert un argument ne posant pas de problème de compilation
- code_stdin_bon : code qui requiert du contenu provenant de l'entrée standard ne posant pas de problème de compilation
- code_bon_JAVA : code en langage JAVA ne posant pas de problème de compilation

1. **appel :** sendRequest()
   retour attendu : message erreur de SOAP : Test sendRequest requiert 7 paramètres.
2. **appel :** sendRequest(<sessionKey>, 1,code_bon_C, "", "", "", 0)
   retour attendu : retCode=0 ; reference=<numéro requete>

3. **appel :** sendRequest(<sessionKey>, -999, code_bon_C, "", "", "", 0)
   retour attendu : retCode=2 ; reference=-1

4. **appel :** sendRequest(-1, 1, code_bon_JAVA, "", "", "", 0)
   retour attendu : retCode=4 ; reference=-1

5. **appel :** sendRequest(<sessionKey>, 1, code_faux, "", "", "", 0)
   retour attendu : retCode=0 ; reference=<numéro requete>

6. **appel :** sendRequest(<sessionKey>, 1, code_bon_C, "mauvaise option", "", "", 0)
   retour attendu : retCode=0 ; reference=<numéro requete>

7. **appel :** sendRequest(<sessionKey>, 1, code_1arg_bon, "", "", "", 0)
   retour attendu : retCode=0 ; reference=<numéro requete>

8. **appel :** sendRequest(<sessionKey>, 1, code_1arg_bon, "", "1 2", "", 0)
   retour attendu : retCode=0 ; reference=<numéro requete>

9. **appel :** sendRequest(<sessionKey>, 1, code_stdin_bon, "", "", "c", 0)
   retour attendu : retCode=0 ; reference=<numéro requete>

10. **appel :** sendRequest(<sessionKey>, 1, code_stdin_bon, "", "", "", 0)
    retour attendu : retCode=0 ; reference=<numéro requete>

11. **appel :** sendRequest(<sessionKey>, 1, code_bon_C, "", "1", "1", 0)
    retour attendu : retCode=0 ; reference=<numéro requete>

12. **appel :** sendRequest(<sessionKey>, 1, code_bon_JAVA, "", "", "", 0)
    retour attendu : retCode0 ; reference=<numéro requete>

13. **appel :** sendRequest(<sessionKey>, 1, null, "", "", "", 0)
    retour attendu : retCode=5 ; reference=-1

Les tests précédents doivent renvoyer les même résultat avec le booléen d'éxécution à 1.

fonction *run(int sessionKey, int reference, String cmdArgs, String stdin))*

- requête 1 : code sans paramètre, sans stdin
- requête 2 : code avec 1 paramètre, sans stdin
- requête 3 : code sans paramètre, avec stdin
- requête 4 : code avec 1 paramètre, avec stdin

1. **appel :** run()
   retour attendu : message erreur de SOAP : Test sendRequest requiert 4 paramètres.
2. **appel :** run(-1, 1, “”, “”)
   retour attendu : retCode=4
3. **appel :** run(<sessionKey>, -1, “”, “”)
   retour attendu : retCode=3
4. **appel :** run(<sessionKey>, 1, “1”, “stdin”)
   retour attendu : retCode=0
5. **appel :** run(<sessionKey>, 1, “”, “stdin”)
   retour attendu : retCode=0

6. **appel :** run(<sessionKey>, 1, “1”, “”)
   retour attendu : retCode=0
7. **appel :** run(<sessionKey>, 1, “”, “”)
   retour attendu : retCode=0

Les quatres derniers tests appliqués respectivement aux requetes 2, 3 et 4 doivent également retourner 0.

fonction *getDetails(int sessionKey, int reference))*

1. **appel :** getDetails()
   retour attendu : message erreur de SOAP : Test getDetails requiert 2 paramètres.
2. **appel :** getDetails("", 1)
   retour attendu : retCode=4 ; status=-1 ; result=-1
3. **appel :** getDetails(<sessionKey>, -1)
   retour attendu : retCode=3 ; status=-1 ; result=-1

fonction *getStatus(int sessionKey, int reference))*

1. **appel :** getStatus()
   retour attendu : message erreur de SOAP : Test getStatus requiert 2 paramètres.
2. **appel :** getStatus("", 1)
   retour attendu : retCode=4 ; status=-1 ; result=-1
3. **appel :** getStatus(-1, <sessionKey>)
   retour attendu : retCode=3 ; status=-1 ; result=-1

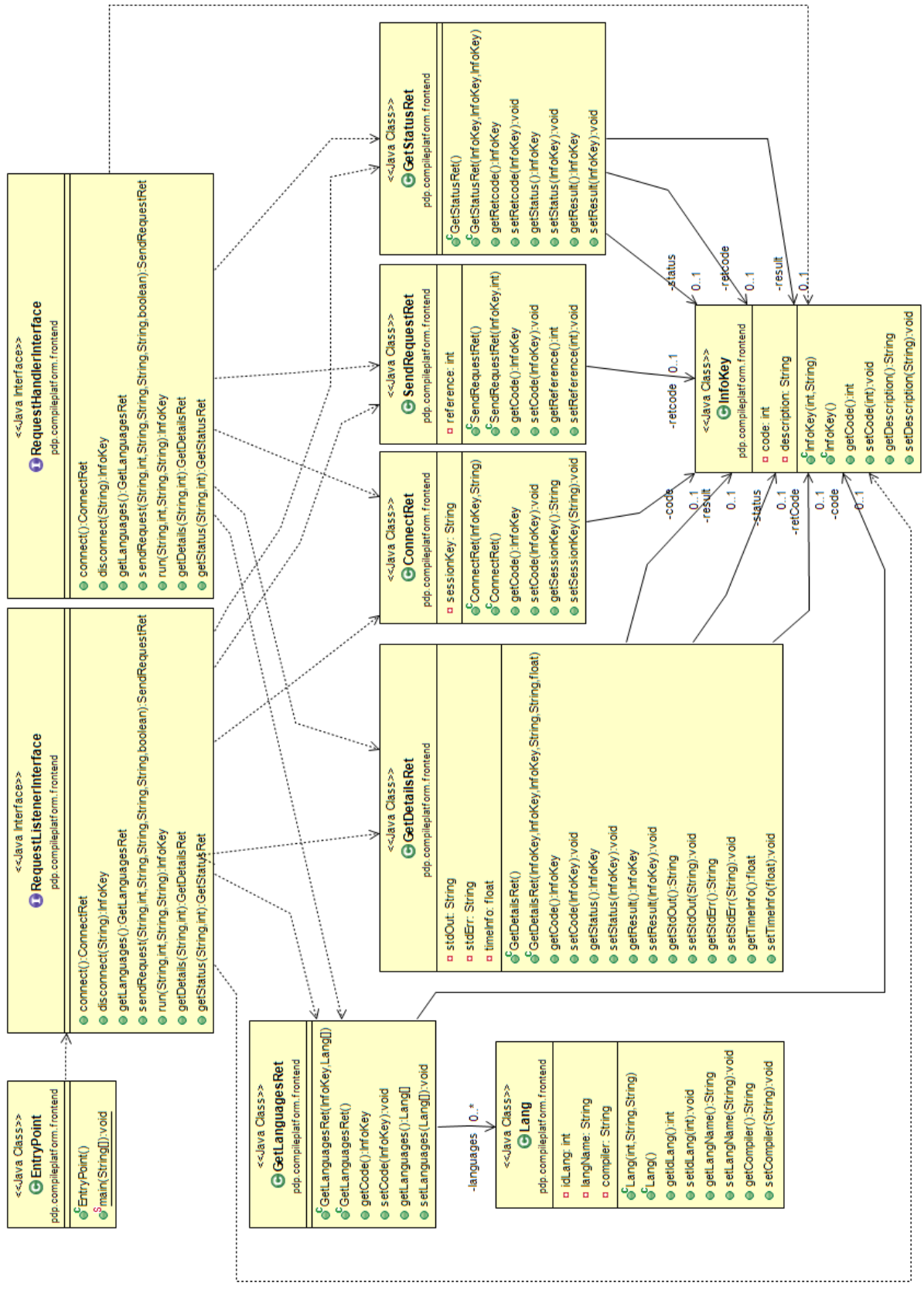


FIGURE 7.1 – Diagramme de classes du package frontend

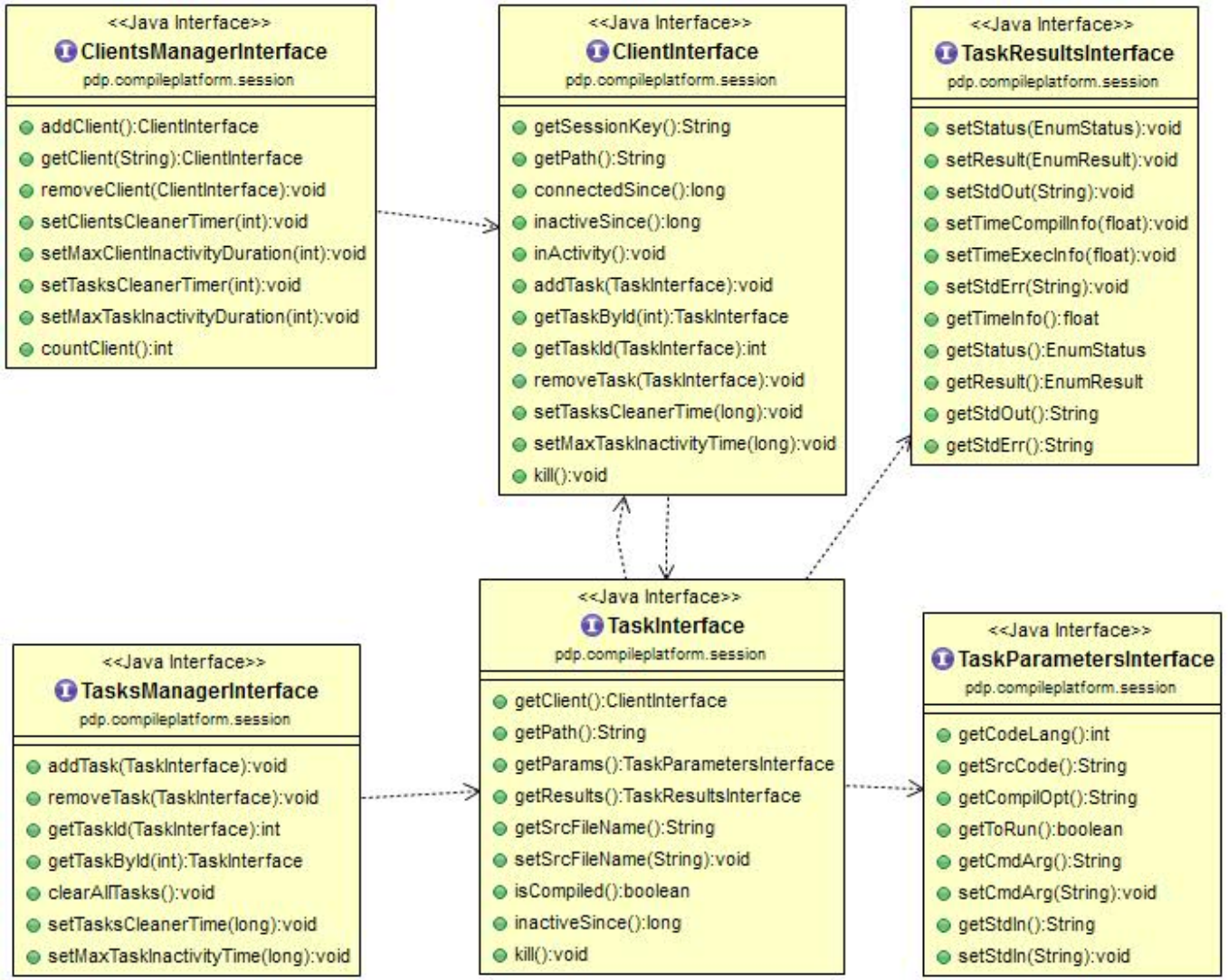


FIGURE 7.2 – Diagramme de classes du package session

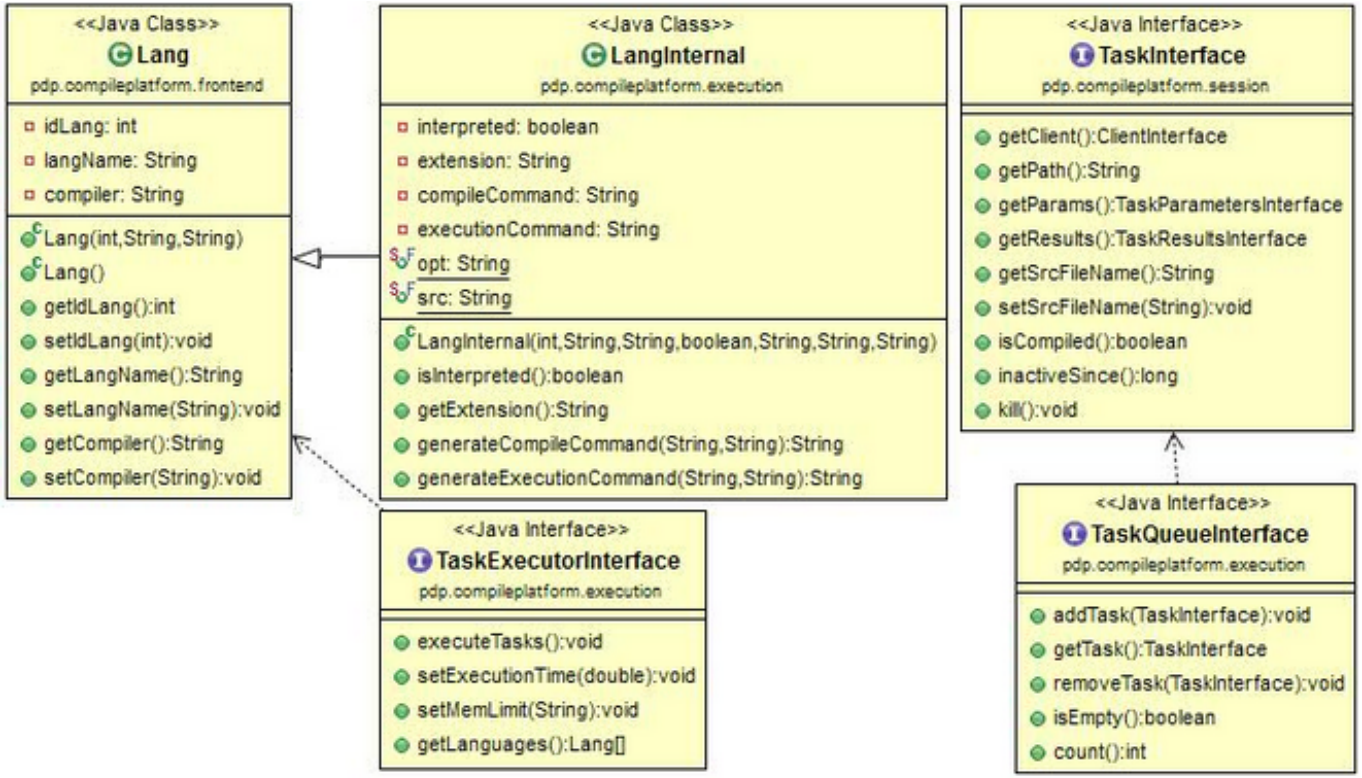


FIGURE 7.3 – Diagramme de classes du package execution