UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Claroffline :

Conception et développement du bundle offline pour Claroline

Promoteur :
Marc Lobelle

Lecteurs :
Chantal Poncin
Stéphane Klein

Mémoire présenté en vue de l'obtention
du grade de master 60 crédits
en sciences informatiques

Par :
Pierre-Yves Légéna
Vincent Van Ouytsel

Louvain-la-Neuve
Année Académique 2013-2014


consortium
claroline
EASY & FLEXIBLE LEARNING SOLUTIONS

Abstract

Résumé anglais
Résumé français

Remerciements

Marc Lobelle Philippe Mercenier Stephane Klein Nicolas Goffin
Marie Dechamps Nicolas Burny

Table des matières

Chapitre 1

Introduction

Au sein de ce travail vous trouverez ... et ...

Ce document constitue une documentation pour développeur qui vous contribuer à notre travail. Nous y détaillons tout les choix d'implémentation que nous avons fait et pourquoi nous avons opéré en ce sens.

En annexe une documentation pour les administrateurs qui voudraient ajouter notre bundle au sein de leur plateforme claroline connect est également fournie. Enfin, une documentation pour l'utilisateur voulant synchroniser sa plateforme est disponible au travers de vue sur le menu dédiés du site internet.

Dans le présent document, nous expliquons le fonctionnement du plugin pour les ressources textes, fichiers répertoires et forum. Des idées quant à la suite de l'implémentation sont données au sein du dernier chapitre.

Ce travail a ete réalisé en collaboration avec les développeurs de Claroline

Ajouter toutes les références [**?**]

Parler du multi-langue supporté ?

Parler de la gestion des exceptions pour le transfert

Chapitre 2

Lexique Claroline

Le lecteur pourra trouver ci-dessous une définition concise des termes régulièrement utilisés tout au long du présent document afin de l'aider dans sa compréhension du texte.

2.1 Lexique Claroline

Espace d'activité : Aussi appelé workspace en anglais, il est à Claroline Connect ce qu'était un cours dans la première version de Claroline. Il s'agit donc d'un espace sur le site regroupant des ressources (fichiers, forums,...), disposant d'un manager (professeur) organisant l'espace d'activité et de collaborateurs (étudiants) consultant son contenu.

Ressource : Au sein de la plateforme Claroline, les ressources sont les composantes d'un espace d'activités, il peut s'agir de fichier `PDF` ou *PowerPoint*, d'un Forum ou d'une Annonce par exemple.

Type de ressource - `Fichier` : Au sein d'un espace d'activités, ce type de ressource définit un fichier chargé par son gestionnaire. Il peut s'agir de fichier *PowerPoint*, de *Portable Document File* (`PDF`) ou de tout autre type de document.

Type de ressource - `Texte` : Il s'agit d'une zone de texte éditable directement depuis la plateforme. Cette zone peut contenir tout type de texte, allant du simple texte informatif au texte dit riche tel que du `HTML`.

Type de ressource - `Répertoire` : Il est l'équivalent d'un dossier au sein d'un système d'exploitation classique ; il s'agit d'une ressource en rassemblant d'autres. En d'autres termes, une ressource répertoire est donc un regroupement d'autres ressources de tout type au sein d'un espace d'activités.

Type de ressource - `Forum` : Les forums sont des espaces d'échanges virtuels sur lesquels des utilisateurs ont la possiblité de consulter des messages postés par les autres utilisateurs et d'y répondre en postant un message de leur choix. Ces messages sont classés par catégories et sujets.

- **Categorie :** Les catégories sont à la base d'un Forum et regroupent plusieurs sujets. Elles sont en quelque sorte un répertoire de sujets.
- **Sujet :** Au sein de chaque catégorie se trouvent des sujets ; ces sujets sont un ensemble de messages. Ils déterminent le thème des conversations.
- **Message :** Au sein de chaque sujet se trouvent plusieurs messages. Ils sont l'expression participative des utilisateurs du forum.

2.2 Vocabulaire informatique

Bundle : Un Bundle est un répertoire qui contient un ensemble de fichiers (classes PHP, feuilles de style, JavaScripts, images, ...) qui implémentent une fonctionnalité unique. Nous définissons *Symfony* et expliquons le fonctionnement des bundles dans le chapitre 4.2.1 page 18.

Conteneur de services : Un conteneur de service est un objet spécial qui gère l'instanciation des `Services` au sein d'une application.

Listener : Un Listener (ou écouteur) d'évènement est un élément logiciel à l'écoute d'évènement qui effectue certains opérations selon ces derniers.

Métadonnées : Les métadonnées sont des données sur les données. Par exemple, dans le cas d'un document `PDF`, l'utilisateur peut disposer de la taille du fichier, de sa date de création ou encore de son auteur.

Repository : Les repository (ou dépôt) sont des classes PHP aidant à récupérer les entités d'une classe particulière.

Requête SQL : Le Structured Query Language (SQL)est un langage de requête, c'est-à-dire un langage informatique destinée à interroger et manipuler des bases de données. Ce dernier permettant de chercher, ajouter voir supprimer des données. SQL manipule des bases de données relationnelles, des stocks d'informations organisés selon le modèle de données relationnel.

Requête DQL : Le Doctrine Query Language (ou DQL) est similaire au SQL à l'exception que les requêtes s'effectuent sur un objet plutôt que sur les lignes d'une table.

Service : Terme générique désignant tout objet `PHP` effectuant une tâche spécifique. Dans *Symfony* un `Service` est souvent configuré et récupéré par un conteneur de services.

YAML : Il s'agit d'un format de représentation de données par sérialisation Unicode.

Chapitre 3

Mise en contexte

Remplacer machine en ligne - hors ligne par machine etudiant, serveur

Notre mémoire trouve son origine dans un besoin des pays en voie de développement : pouvoir continuer un processus d'apprentissage sur les plateformes d'e-learning, terme désignant l'apprentissage sur Internet, alors que l'accès à celui-ci est limité.

Aujourd'hui les plateformes d'apprentissage en ligne sont devenues des outils incontournables dans le partage des connaissances et l'apprentissage des étudiants. Ces plateformes permettent aux formateurs et professeurs de créer un parcours pédagogique incluant notamment des lectures et évaluations. Par ailleurs, les étudiants ont bien souvent l'occasion de participer et de poser leurs questions au travers de forums, ce qui favorise le partage et l'interactivité entre les membres d'un cours. Toutefois, pour profiter de l'ensemble des services fournis par ces plateformes, il faut disposer d'une connexion Internet. Il n'y a pas de grande limitation pour les pays occidentaux étant donné les multiples possibilités de connexions. Mais, la situation n'est pas identique partout et la connectivité dans certaines zones du monde, notamment en Afrique, reste assez faible.

C'est dans ce cadre que nous nous sommes intéressés au mémoire proposé par Marc Lobelle, professeur à l'Université Catholique de Louvain-la-Neuve et avons décidé de contribuer au développement de la plateforme Claroline. Ce travail rencontrait nos attentes, car il s'agissait d'un projet de développement dont l'utilité pourrait être directe. De plus, en tant qu'utilisateur régulier de la plateforme, développer une extension permettant son utilisation en l'absence d'accès à Internet, nous semblait particulièrement intéressant.

Claroline est un logiciel Open Source permettant de déployer une plateforme d'apprentissage sur Internet. Le développement de ce *Learning Management System* (LMS) libre et gratuit a été initié en 2001 par l'Université Catholique de Louvain [1] et est aujourd'hui piloté par le *Consortium Claroline*. Au sein de ce logiciel, un utilisateur a la possibilité de créer ou de rejoindre des espaces d'activités [2]. Si un utilisateur est gestionnaire d'un espace d'activités, il a la capacité d'y créer des ressources [3]. Celles-ci seront alors accessibles à l'ensemble des utilisateurs membres de cet espace. Pour plus d'information sur le projet Claroline, nous invitons le lecteur à se rendre sur le site web officiel de la plateforme [4].

1. Université Catholique de Louvain-la-Neuve - http://uclouvain.be
2. cfr. Lexique, chapitre 2.1 page 13
3. cfr. Lexique, chapitre 2.1 page 13
4. Claroline - http://claroline.net/type/claroline

Ce travail s'adresse à tous les utilisateurs d'une plateforme Claroline souhaitant continuer à travailler sans y être connecté en permanence. Un groupe d'utilisateurs potentiels du travail que nous avons développé serait composé d'étudiants ne disposant pas de connexion Internet fiable et/ou stable. Ces derniers pourraient donc se synchroniser ponctuellement sur la plateforme de référence lorsqu'ils se rendent à leur université ou dans tout autre lieu leur fournissant un accès au WEB.

Environnement supporté

Mais d'autres utilisateurs pourraient être interessés par ce projet. En effet, nous avons également eu l'occasion de rencontrer une personne travaillant pour *Médecin Sans Frontière*[5]. Leurs équipes partant en mission possèdent une clé USB sur laquelle des cours sont chargés au préalable. Sur place, ce personnel ne dispose pas forcément d'un accès Internet constant mais doit cependant être à même de continuer sa formation et également d'envoyer ses questions par l'intermédiaire du système de forum.

L'objectif final de ce mémoire est de créer un plugin pour Claroline, venant compléter les services proposés par la plateforme à ses utilisateurs. Celui-ci permet de synchroniser une plateforme en ligne, la plateforme de référence, et une plateforme hors-ligne, installée sur la machine de l'utilisateur. Dans la pratique, un utilisateur pourrait, s'il dispose d'une connexion à Internet, lancer le processus de synchronisation. Au cours de ce dernier, le plugin va récupérer le contenu créé depuis la dernière synchronisation sur la plateforme en ligne afin de les enregistrer sur la plateforme déconnectée. Et par la suite, il va charger sur la plateforme de référence, le travail qui a été effectué sur la plateforme hors-ligne. On parle alors de synchronisation bidirectionnelle.

De plus, comme ce critère est important pour une partie des utilisateurs potentiels, nous voulons que le produit fini soit portable. C'est-à-dire, que l'utilisateur doit avoir la possibilité de le déplacer d'une machine à une autre, par exemple à l'aide d'une clé USB.

5. Médecin Sans Frontières - http://msf.org

Chapitre 4

Choix de conception et outils

Dans ce chapitre nous allons commencer par expliquer le choix de notre solution parmi les possibilités que nous avons envisagées. Ensuite nous détaillerons les technologies utilisées pour réaliser cette solution. Enfin, nous vous expliquerons notre découpe en sous-problèmes avant de terminer par les détails de modélisation d'*OfflineBundle*.

4.1 Choix de la solution

UPDATE, plus de sync constant !

Afin de réaliser ce projet, nous avons envisagé trois solutions possibles. La première d'entre elles, notre choix, est une solution directement intégrée à Claroline comme n'importe quel autre module composant l'outil. L'avantage principal de cette solution est qu'elle permet à l'utilisateur de continuer à travailler sur Claroline, qu'il dispose ou non d'une connexion Internet. Il retrouve ainsi la totalité de l'environnement et des fonctionnalités lui étant familières. De plus notre choix permet une synchronisation bidirectionnelle, ce qui est une des contraintes pour ce projet, et, de manière analogue aux autres solutions, permet de minimiser l'ensemble des informations échangées sur le réseau.

En revanche, le problème majeur de notre solution est le temps de latence du chargement des pages du site de Claroline via un serveur apache exécuté sur un ordinateur aux capacités hardware plus réduites. Etant donné que nous travaillons à mettre hors-ligne la plateforme Claroline, l'ensemble des contraintes s'appliquant à Claroline s'appliquent également à notre projet. Cela requiert également plus d'espace de stockage. Ces deux défauts ne se trouvent pas dans la troisième solution décrite ci-dessous.

Une autre solution envisagée est l'utilisation du *local storage* d'`HTML5` pour enregistrer via la cache les informations nécessaires au redémarrage de Claroline sans une connexion Internet. Le problème de cette solution est lié au décalage entre les technologies utilisées par Claroline et les possibilités offertes par `HTML5`. En effet, l'implémentation de Claroline utilise majoritairement des technologies web déployées côté serveur alors que dans ce cas-ci, nous aurions du développer l'application côté utilisateur en `HTML5`. Le problème était donc l'ampleur du travail à fournir dans le temps imparti dès lors que nous avions la contrainte de rendre utilisable l'ensemble de la plateforme sans disposer d'une connexion Internet.

La dernière solution à laquelle nous avions pensé était la création d'un logiciel détaché de Claroline, qui s'occuperait de synchroniser l'ensemble des ressources dans un répertoire

défini par l'utilisateur. Cette solution aurait été similaire à d'autre logiciels connus tel que Dropbox[1] ou Google Drive[2].

Les désavantages de cette dernière solution étaient nombreux : l'utilisateur n'aurait plus accès à l'interface qu'il connaît. Nous aurions du concevoir des solutions pour pouvoir lire sur l'ordinateur toutes les ressources spécifiques telles que les forums ou les wikis, alors que tout ce travail est déjà implémenté dans Claroline. Enfin, les possibilités d'extension du logiciel sont plus limitées qu'avec notre solution. Par exemple, il semble compliqué d'envisager une mise hors-ligne des fonctionnalités de modifications des paramètres d'un compte.

4.2 Technologies utilisées

Nous avons utilisé le language PHP[**?**] qui est l'un des plus utilisés sur le WEB et le framework *Symfony* afin de faciliter l'intégration avec la plateforme Claroline car cette dernière est développée avec Symfony. *PHP est un langage de programmation compilé à la volée principalement utilisé pour produire des pages Web dynamiques via un serveur HTTP, mais pouvant également fonctionner comme n'importe quel langage interprété de façon locale*.

4.2.1 Symfony

Symfony est un framework de développement WEB écrit en PHP et se basant sur le patron *Modèle Vue Controller* (MVC). Lancé en 2005, ce framework en est actuellement à sa deuxième version. Un framework est un cadre de travail créé par des développeurs pour d'autres développeurs ; il n'est pas conçu pour des utilisateurs finaux. En d'autres termes, il s'agit donc d'une série d'outils servant à concevoir les fondations de l'architecture et les grandes lignes d'un logiciel. Concretement, *Symfony* permet grâce à ses composants un gain de productivité pour la mise en place d'un site Internet.

Le principe du patron *Modèle Vue Controller* est de structurer les responsabilités au sein d'une application conçue en orienté objet. Celui-ci propose une découpe des classes en trois catégories : le modèle représentant le coeur algorithmique de l'application, la vue contenant les interactions avec l'utilisateur et le controller servant de lien entre la vue et le modèle.

Les bundles

L'architecture d'une plateforme *Symfony* est découpée en plusieurs parties nommées bundles. Ces derniers sont à la source de la modularité du framework. En effet, ils permettent d'intégrer simplement une fonctionnalité qui aurait été développée par quelqu'un d'autre. Nous pouvons citer par exemple *Buzz* le client HTTP que nous utilisons pour implémenter notre transfert, *twig-js-bundle* le moteur de rendu javascript des templates *Twig* ou encore *hwi-oauth-bundle* permettant l'authentification via les réseaux sociaux.

1. https://www.dropbox.com/
2. https://drive.google.com

Le projet Claroline est lui même divisé en plusieurs bundles ; citons notamment le `CoreBundle`, reprenant les éléments de base nécessaires au fonctionnement de Claroline ou encore le `ForumBundle` s'occupant comme son nom l'indique de la gestion des ressources forums. La liste complète des bundles de Claroline peut être consultée avec le code source du projet distribué sur Github[3]. C'est dans ce contexte que vient s'ajouter notre *OfflineBundle* pour Claroline. Il est destiné à être intégré à Claroline, comme n'importe quel autre bundle.

Composer

Composer est un gestionnaire de dépendances écrit en *PHP* utilisant certains composants de *Symfony*. C'est ce gestionnaire de dépendances qui est utilisés par *Claroline*. Par exemple, `ClarolineCoreBundle` requiert `ClarolineForumBundle`. Lorsque l'administrateur système met à jour sa plateforme Claroline, *Composer* est exécuté et ira chercher la dernière version de `ForumBundle` sur le dépot référence de Claroline. La liste exhaustive des bundles dont dépend un projet se trouve dans le fichier `composer.json` situé à la racine de son répertoire.

Il existe des conventions de nommages pour les bundles, ceux-ci résultent de la concaténation du nom du développeur (ou de l'entreprise développant le bundle) et du nom du bundle. Cette nomenclature détermine également l'emplacement des fichiers dans le projet, ceux-ci sont ordonnés dans le dossier vendor et suivant une arborescence similaire au nom.

4.2.2 HTML

HyperText Markup Language (`HTML`)[**?**]est le langage standard utilisé pour créer des pages `Web`. Il s'agit d'un langage fonctionnant par balise c'est-à-dire qu'il est composé d'éléments `HTML` consistant en une série de balises fonctionnant le plus souvent par paires (exemple : `<p> ... </p>` pour un paragraphe) bien qu'il existe quelques exceptions comme par exemple la balise `<br />` pour faire un retour à la ligne au sein d'un document.

4.3 Découpe en sous problèmes : la synchronisation

Maintenant que le cadre a été défini et que les principaux outils ont été donnés, nous pouvons passer à la découpe en sous problèmes de ce projet. Deux parties majeures nous sont apparues : la synchronisation et l'installation. La première partie, la synchronisation, se charge, à la demande de l'utilisateur, de mettre à jour la plateforme en ligne avec le contenu nouvellement créé sur la plateforme hors-ligne et réciproquement. En d'autres termes, l'objectif de cette partie du travail est de maintenir l'utilisateur à jour, tant au niveau des nouvelles ressources créées en ligne au sein de ses espaces d'activités que du contenu qu'il a créé hors-ligne sur son ordinateur personnel.

Lors de notre analyse sur cette phase de synchronisation, nous avons encore découpé la tâche en trois sous-parties : un premier module créant une archive contenant les modifications

3. https://github.com/claroline

effectuées depuis une date repère. Un second, responsable de transférer les données d'une plateforme à une autre. Et le dernier chargeant les modifications contenues au sein d'une archive vers une plateforme. Ces trois sous-sections disposeront chacune d'un chapitre pour y détailler avec précision le travail qui y a été apporté. Nous avons pu établir une découpe clair des tâches, ce qui soutient l'idée que cela est bien modélisé.

4.3.1 Création d'une archive

La création de l'archive consiste à récupérer toutes les modifications qui ont eu lieu dans les espaces d'activités [4] auquel un utilisateur est inscrit depuis la dernière synchronisation et d'intégérer celle-ci dans un fichier transférable d'une plateforme à une autre ; en l'occurence une archive zip.

4.3.2 Transfert d'une archive

La seconde étape du processus consiste à transférer cette archive d'une plateforme à une autre. Pour ce faire, nous nous sommes reposés sur le protocole `HTML` et transférons les données au travers de requêtes `POST`. Par ailleurs, nous avons du prendre en compte le fait que notre projet s'adresse à des zones où la connectivité est limitée, il a donc fallu s'arranger pour que le transfert soit le plus robuste possible et puisse reprendre là où il s'était arrêté tout en minimisant les échanges multiples d'une même donnée.

4.3.3 Chargement d'une archive

Le chargement de l'archive est l'étape lors de laquelle nous intégrons à une plateforme une archive comprenant une série de changements, de travaux qui ont été effectués par l'utilisateur. Cette étape inclut la résolution des éventuels conflits qui pourraient avoir lieu entre deux ressources similaires voire identiques.

4.4 Découpe en sous problèmes : l'installation

La seconde partie du travail, l'installation, se charge de mettre à disposition une plateforme utilisable par l'utilisateur déconnecté. Il s'agit d'une plateforme que l'utilisateur sera à même de mettre en place sur son ordinateur personnel ou sur tout autre dispositif de stockage de masse. Cette plateforme diposera des espaces d'activités de l'utilisateur et de la possibilité de se synchroniser.

Nous avons fait le choix de concevoir ce module d'installation comme une plateforme Claroline vierge de tout contenu. Elle pourra alors être téléchargée et directement utilisée. L'utilisateur devra, une fois le logiciel téléchargé et démarré, entrer ses identifiants pour ensuite effectuer la première synchronisation qui téléchargera l'ensemble des ressources des différents espaces d'activités auxquels il est inscrit. Vous trouverez l'ensemble des détails à propos de l'installation dans le chapitre 9 page 71.

4. cfr. Lexique, chapitre 2.1 page 13

4.5 Modélisation du OfflineBundle

Dans cette section nous allons expliquer les différents composants de *Symfony* et la manière dont nous les avons articulés afin de modéliser notre bundle.

4.5.1 Les routes

Les routes sont les URL du site que l'on définit et auxquelles on associe des actions. Dans le cas de ce OfflineBundle, nous avons entre autre défini la route `"/sync/"` ; route que nous avons associée à la page d'accueil de notre module de synchronisation sur laquelle l'utilisateur reçoit les instructions pour synchroniser son ordinateur personnel. Le framework Symfony appliquant le modèle de design `MVC`, les routes sont définies au sein du controlleur qui rappellons-le sert de lien entre le modèle et les vues.

4.5.2 Les classes controlleurs

Les controlleurs sont les classes qui articulent l'application entre le modèle et les vues. Au sein d'`OfflineBundle`, nous avons défini deux controlleurs : `OfflineController` et `SynchronisationController`. Les actions de notre bundle sont réparties en deux controlleurs distincts car ceux-ci ont des usages et des règles de sécurité différentes.

`OfflineController` est utilisé par la plateforme déconnectée pour la gestion des actions de l'utilisateur au sein des vues de notre module. L'accès à ensemble des actions qu'il permet est restreint à un utilisateur connecté sur la plateforme. Tandis que `SynchronisationController` s'occupe de prendre en charge sur la plateforme connectée les requêtes faites par les utilisateurs déconnectés. De ce fait, l'accès à ses routes doit être libre ; la sécurité sera gérée à l'aide d'une identification établie sur base d'informations contenues dans la requête.

Decouper en 2 paragraphes, 1 par controller, dixit Lobelle

4.5.3 Les classes managers

Un manager est une classe du modèle regroupant un ensemble de fonctions contribuant à un même but. Les managers sont là pour regrouper l'ensemble des actions d'une entité ; cette organisation a également pour objectif d'alléger les controlleurs, ces derniers devant être les plus concis possible. Dans le cas d'`OfflineBundle` nous avons créé quatre managers : `CreationManager` qui contient les fonctions se chargeant de la création de l'archive, `LoadingManager` qui contient les fonctions permettant le chargement d'une archive, `TransferManager` intégrant l'ensemble des fonctions utiles dans le transfert d'une archive et `SynchronisationManager` intégrant l'implémentation de l'articulation des différentes parties.

4.5.4 Les classes repository

Les repository sont des classes contenant les différentes requêtes qui nous sont nécessaires pour accéder à la base de données. Le seul repository que nous avons implémenté

est `UserSynchronizedRepostiroy`, le repository permettant l'accès aux champs de l'entité `UserSynchronized` que nous avons créée et qui est expliquée ci-dessous.

Lobelle a pas compris, plus d'infos !

4.5.5 Utilisation des constantes

Attention, connecté = seveur central

Pour la réalisation d'`OfflineBundle` nous avons choisi de regrouper au sein du fichier `SyncConstant` l'ensemble des constantes globales que nous utilisons. C'est dans ce fichier que l'on fixe l'emplacement des répertoires utilisés pour stocker temporairement les archives chargées et téléchargées lors des synchronisations ou encore la localisation du fichier de configuration.

4.5.6 Twigs

Twig est le moteur de template par défaut de *Symfony 2*. Il s'agit de fichiers `HTML` dans lequel des tags supplémentaires sont ajoutés de manière à être générée dynamiquement lors de l'interprétation du code. Twig fonctionne en compilant le template la première fois qu'il est demandé et est ensuite sauvegardé en cache. Ce système fait qu'en production, un template qui a déjà été compilé et caché ne sera plus modifié, même si les sources du template sont changées à moins de vider le cache pour forcer une nouvelle compilation.

La syntaxe de Twig est fortement inspiré du système de template de *Django*, un web framework *Python* populaire. On retrouve l'ensemble des fichiers twigs dans le répertoire `Resources/Views`. Pour `OfflineBundle` nous l'utilisons entre autre pour afficher : le formulaire de première synchronisation, la vue informative pour l'utilisateur, les résultats de la synchonisation.

4.5.7 UserSynchronized

Dans *Symfony*, les `Entity` représentent des objets du modèle. Ceux-ci sont rendus persistants dans une base de données à l'aide de *Doctrine*, l'`ORM` utilisé dans *Symfony*, donc avec Claroline.En programmation orienté objet, un `ORM` aussi appelé *Object Relationnal Mapping*, est une technique de programmation autorisant le développeur à travailler directement avec une base de données orientée objet ; l'`ORM` définissant les correspondances avec la base de données relationnelles.

`UserSynchronized` est une `Entity` que nous avons créée dans l'`OfflineBundle` afin de répondre à notre nécessité de retenir toutes les informations relatives à la synchronisation. Cet object contient donc une référence à l'utilisateur, la date de dernière synchronisation, la date d'envoi du dernier paquet, le status dans lequel le dernier transfert s'est arrêté [5] ainsi qu'un champ `filename` utile pour pouvoir redémarrer depuis certains états. Le schéma de la base donnée est illustré sur la figure 4.1.

5. Voir explication de l'implémentation du processus global 5.1 page 25

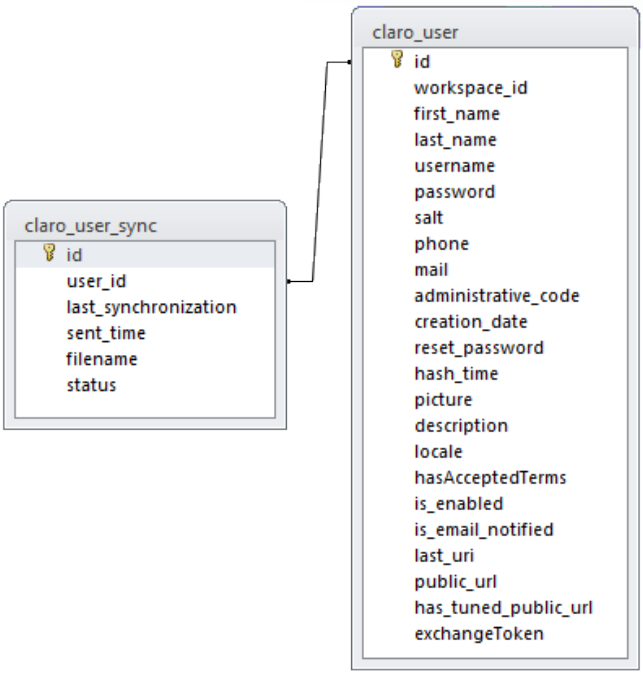

FIGURE 4.1 – Schéma de la table `claro_user_sync` représentant l'objet `UserSynchronized`

Pour implémenter la création de cet objet au sein de Claroline nous avons du utiliser le système de migrations de base de données inclues avec l'application. L'ensemble des explications relatives au système de migrations de Claroline peut être trouvé sur le lien suivant : https://github.com/claroline/MigrationBundle.

Chapitre 5

Le processus global

Dans ce chapitre nous allons vous présenter comment nous combinons les trois parties principales que nous venons de décrire afin de réaliser la synchronisation complète d'un utilisateur déconnecté avec la plateforme Claroline de référence. Nous commencons avec le processus de manière simplifiée afin d'en présenter les bases. Ce schéma sera compléxifié tout au long de ce chapitre dans le but de présenter les différentes réfléxions et solutions que nous y avons apportées.

5.1 Le processus de base

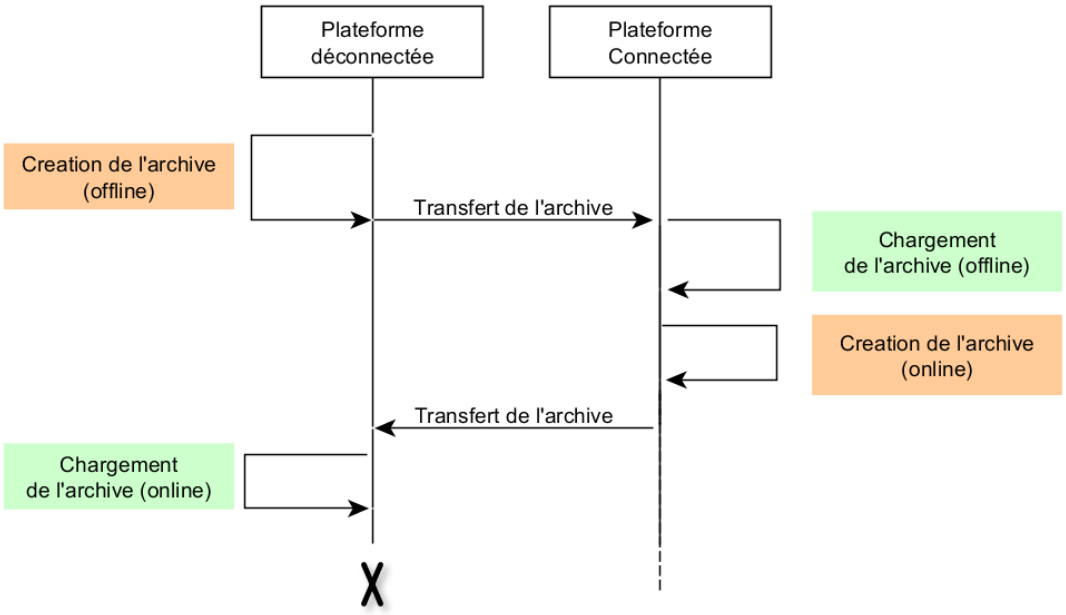

Figure 5.1 – Schéma du processus complet

La figure 5.1 illustre de manière synthétique le processus complet de synchronisation que nous avons mis en place. De haut en bas, l'axe vertical représente l'avancement dans le

temps. Comme nous pouvons le voir sur cette figure, le processus de synchronisation commence par la création de l'archive de synchronisation sur la plateforme déconnectée. Cette archive est ensuite transférée sur la plateforme en ligne où elle sera chargée. Durant le chargement, les éventuels conflits seront résolus. La plateforme en ligne créera alors à son tour son archive de synchronisation. Un transfert de cette archive vers la plateforme déconnectée est ensuite effectué. Le processus se termine par le chargement sur la plateforme déconnectée du nouveau contenu reçu depuis la plateforme connectée.

Les conflits sont résolus en ligne, ce qui signifie que toutes les résolutions de conflits seront similaires pour tous les utilisateurs. Nous pouvons voir sur le schéma, au sein de la plateforme en ligne, que le processus de chargement s'effectue avant celui de création. En procédant comme tel, les conflits sont gérés par la plateforme en ligne. Leurs résolutions seront alors ajoutées à l'archive transférée à la plateforme déconnectée.

5.2 De multiples paquets

Un élément important à noter sur la figure 5.1 est que le transfert y est représenté de manière simplifiée. Dans notre implémentation, comme nous le verrons dans le chapitre 7 page 49, les archives sont découpées en de multiples paquets. Cette découpe est faite afin de permettre un transfert sur les connexions plus modestes et ainsi éviter les transferts lourds. Ces derniers pouvant être corrompus et nécessistant de réenvoyer plusieurs fois les même données en cas d'échec. En envoyant de multiples petits paquets, chaque segment peut être validé indépendamment et le cas échéant seuls les éléments nécessaires sont retransférés. Pour garantir la sécurité de l'accès aux données, l'implémentation de notre système de transfert prend également en charge une identification de l'utilisateur à chaque requête à l'aide d'un identifiant unique.

5.3 Ajout des dates de synchronisation et d'envoi

Afin d'augmenter le nombre de cas gérés par notre processus, ce schéma doit encore être complexifié. Il est nécessaire de pouvoir retenir le dernier moment lors duquel nous nous sommes synchronisés, de manière à ne pas transférer l'ensemble des ressources de chaque espace d'activités à chaque synchronisation. En conservant cette information nous serons à même de détecter les ressources modifiées depuis la dernière synchronisation de l'utilisateur. Nous avons décidé d'enregistrer cette information dans l'entité `UserSynchronized` que nous avons créé à cet effet.

La date de dernière synchronisation est transférée par la plateforme déconnectée dans la requête de synchronisation. De cette manière, la plateforme connectée utilise la date que nous lui fournissons et répond en nous transmettant l'ensemble des changements ayant eu lieu depuis ce moment. Nous récupérons de ce fait le travail effectué qu'il nous manquait. A la fin du processus, la date de dernière synchronisation est alors mise à jour sur la plateforme déconnectée. Dans l'éventualité où l'utilisateur possède plusieurs plateformes déconnectées,

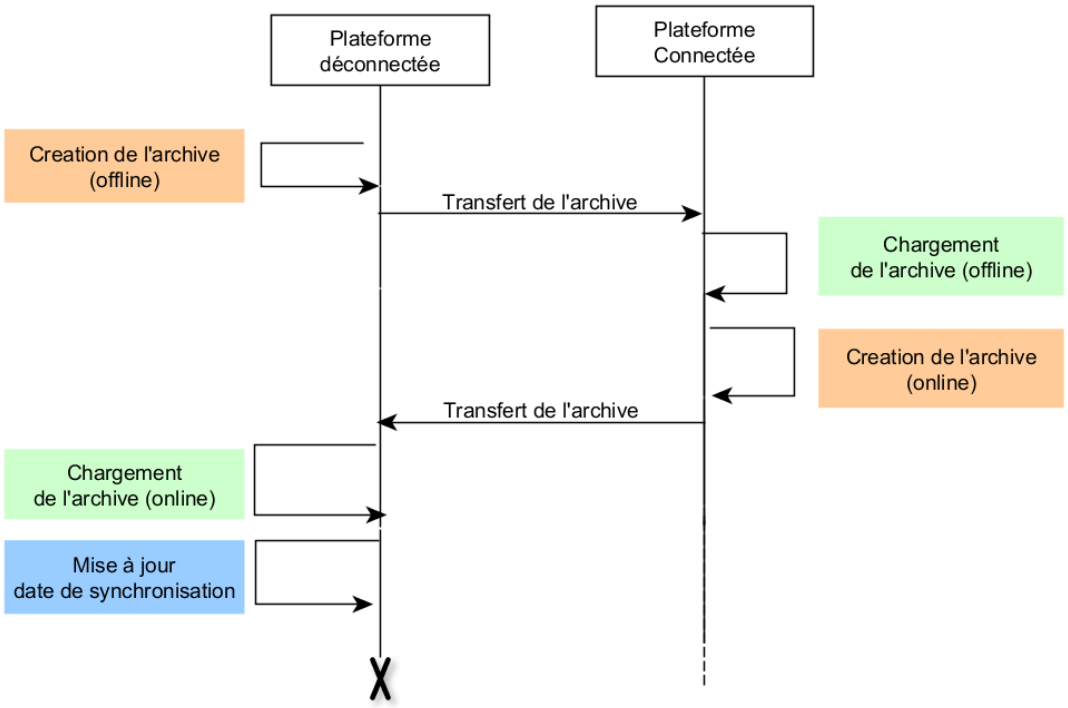

FIGURE 5.2 – Schéma du processus complet avec date de synchronisation

chacune d'entre elles dispose de sa propre date de synchronisation. Cela permet à un utilisateur de synchroniser plusieurs postes de travail différents. En effet, son compte n'est pas lié à une date de synchronisation enregistrée en ligne.

Toutefois nous avons remarqué que ce schéma présentait encore un défaut. Si une ressource est créée sur la plateforme en ligne pendant la période où un utilisateur déconnecté se synchronise, cette ressource ne sera jamais téléchargée sur la plateforme déconnectée. En effet, la date de synchronisation de l'utilisateur n'étant mise à jour sur la plateforme déconnectée qu'après traitement de l'archive reçue depuis la plateforme en ligne ; la date de synchronisation sera forcément postérieure à la date de création de la ressource créée pendant la synchronisation. En conséquence, la ressource créée en ligne ne sera jamais intégrée à l'archive de synchronisation.

La solution que nous avons apportée pour résoudre ce problème est l'enregistrement de la date d'envoi de la requête. Cette date, sauvegardée dans le champ `sentTime` de notre entité `UserSynchronized`, sera utilisée comme valeur pour la date de synchronisation. Lorsque cette date est mise à jour, sa valeur sera donc la date à laquelle le processus a commencé

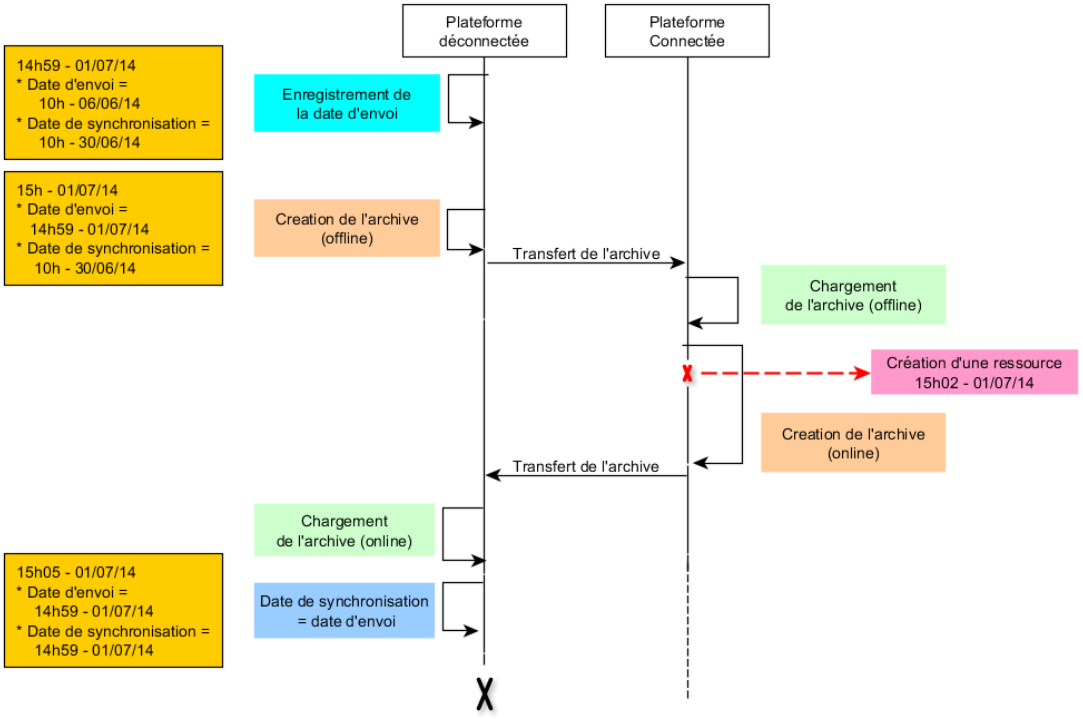

FIGURE 5.3 – Création d'une ressource pendant la synchronisation

plutôt que le moment où il s'est terminé. Il en résulte que la prochaine demande de synchronisation s'effectuera exactement depuis le début de la synchronisation précédente. Il n'y aura plus de ressource ignorée par le processus.

Par exemple sur la figure 5.3, nous pouvons constater que l'utilisateur lance une synchronisation à 14h59. La date d'envoi de l'objet `UserSynchronized` de la plateforme déconnectée est alors mise à jour avec cette valeur. Lors de la création de l'archive du côté de la plateforme en ligne, une ressource est créée (à 15h02 sur le schéma). A la réception de l'archive par la plateforme déconnectée, la date de dernière synchronisation de l'objet `UserSynchronized` sera mise à jour avec la valeur de la date d'envoi, soit 14h59.

Grâce à cela, la prochaine fois que l'utilisateur lancera une synchronisation, il récupèrera toutes les ressources créées depuis le 01/07 à 14h59 ce qui incluera celle créé à 15h02. En l'absence de la date d'envoi, la date de synchronisation aurait eu la valeur de 15h05 et, à la synchronisation suivante, la ressource créée à 15h02 aurait été ignorée.

5.4 Gestion de la résistance aux coupures

Nous avons conçu ce processus de sorte qu'il puisse être redémarré à chaque étape si l'une d'entre elle venait à échouer. Une des premières caractéristiques importantes est notre choix de ne mettre à jour la date de synchronisation qu'à la fin du processus complet. De cette manière, l'étape de création de l'archive peut être reprise comme s'il n'y avait pas eu de coupure. En effet, pour une même date donnée, cette opération produit le même résultat. Par conséquent, si nous relançons le processus après un échec, la date n'aura pas été modifiée et donc la sortie de la fonction de création sera identique à celle obtenue lors de la précédente exécution.

L'implémentation du processus complet, a été organisée au sein du `synchronisationManager`. Cette implémentation a été découpée en différentes étapes, de telle sorte que si le processus est interrompu à un moment ou à un autre, il puisse être relancé automatiquement à l'étape voulue. Cela permet ainsi d'éviter de recommencer inutilement les opérations ayant déjà été réalisées dans certains cas, d'éviter d'utiliser trop de ressources sur le réseau. Afin d'assurer ce redémarrage automatique il nous faut enregistrer la dernière étape effectuée, pour cela nous sauvegardons à la fin de chaque étape le numéro de celle-ci dans le champ `status` de l'objet `UserSynchronized`. Il est important de préciser que cette implémentation permet d'ajouter facilement une nouvelle étape au processus.

Outre le champ `status` dans `UserSynchronized`, le champ `filename` est également utilisé pour la reprise du processus en cas de coupures. En effet, certaines étapes requièrent la mémorisation d'un nom de fichier, en l'occurence le nom de l'archive devant être chargé ou inversémenent, téléchargé. La présence de cette information peut influencer sur la reprise du processus. Par exemple si le processus coupe au moment de l'envoi de l'archive et puis reprend, il va tout d'abord tester la présence de cette archive référencée par le `filename`. Si elle n'est plus disponible (pour une raison ou une autre) il va alors recommencer la construction de cette archive. Ici encore, nous retrouvons un élément qui nous conforte quant à la robustesse de notre processus.

5.4.1 L'implémentation de `SynchronisationManager`

L'ensemble des états que nous avons identifiés sur le processus global sont stockés sous forme de constantes au sein de l'entité `UserSynchronized`. Ces états sont les suivants : `SUCCESS_SYNC`, `STARTED_UPLOAD`, `FAIL_UPLOAD`, `SUCCESS_UPLOAD`, `FAIL_DOWNLOAD` et `SUCCESS_DOWNLOAD`. L'implémentation de l'articulation des étapes se fait dans le `SynchronisationManager`, pour cela nous utilisons la fonction `SynchronizeUser`. Cette fonction consiste en un `switch` qui définit l'étape devant être exécutée en fonction du status de l'entité `UserSynchronized`. Comme cette action est effectuée côté déconnecté, il y a quatre étapes : créer l'archive de la plateforme déconnectée, charger celle-ci en ligne, télécharger l'archive de la plateforme en ligne et charger le contenu de celle-ci au sein de la plateforme déconnectée.

L'enchevêtrement des états et étapes de notre `SynchronisationManager` est représenté

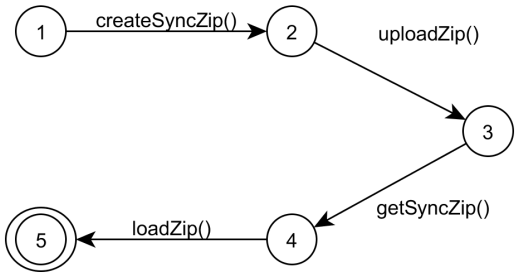

FIGURE 5.4 – Schéma des transitions du processus de synchronisation entre les étapes

Etape	Status
1	`UserSynchronized::SUCCESS_SYNC`
2	`UserSynchronized::STARTED_UPLOAD`
3	`UserSynchronized::SUCCESS_UPLOAD`
4	`UserSynchronized::SUCCESS_DOWNLOAD`
5	`UserSynchronized::SUCCESS_SYNC`

TABLE 5.1 – Correspondance entre les status `UserSynchronized` et les étapes de synchronisation

sur la figure 5.4. Lorsque nous sommes à l'étape 1 et que l'archive est créée nous passons à l'étape 2. De là, lorsque nous envoyons cette archive à la plateforme en ligne nous passons à l'étape 3. Durant cette dernière, la plateforme déconnectée télécharge l'archive de la plateforme connectée ; ce qui nous amène à l'étape 4, qui consiste à charger cette archive au sein de la plateforme déconnectée. Arpès cette dernière étape, le processus est achevé. La table 5.1 fait la correspondance entre les étapes et les états enregistrés.

5.4.2 Coupures électriques

Une des contraintes évoquées par notre promoteur et que nous avons pris en compte est le cas des coupures électriques, qui peuvent arriver de manière inopinée. Les coupures électriques induisent une perte du contenu enregistré dans la mémoire temporaire (`Random Access Memory`). Nous avons donc décidé d'utiliser la mémoire persistante de l'ordinateur, soit le disque dur, soit le solid state drive (SSD) afin d'enregistrer les archives créées et chargées. Tous les fragments de paquets envoyés sont stockés sur la machine. Une fois que l'ensemble des fragments sont réunis, le paquet est recomposé. Les différents fragments sont bien entendu supprimés du répertoire après usage.

5.5 La première synchronisation

Lors de la première synchronisation le processus est légèrement différent. L'utilisateur commence par entrer son identifiant et son mot de passe, une requête est alors envoyée vers la plateforme en ligne afin de vérifier l'existence de cet utilisateur. Si ce dernier est authentifié, il se voit alors inscrit sur la plateforme déconnectée. En plus de son inscription, une instance de l'entité `UserSynchronised` est créée sur la plateforme déconnectée.

L'utilisateur est ensuite amené à effectuer sa première synchronisation. Au cours de cette dernière, l'archive créée offline et envoyée ne contient aucun élément. En effet, à la base la plateforme est vierge de tout contenu, il n'y a par conséquent aucune modification à envoyer à la plateforme en ligne. La plateforme en ligne cependant va préparer et envoyer une archive contenant l'ensemble des cours auxquels l'utilisateur est inscrit, cours qui seront par la suite chargés sur sa plateforme déconnectée. Cela est possible car au début nous forçons la date de synchronisation à la date du 1er Janvier 1970. [1]

1. Temps 0 Unix

Chapitre 6

Création de l'archive

L'objectif de notre premier sous-problème est de créer une archive contenant l'ensemble des modifications apportées à une plateforme par rapport à une date de référence. Nous avons structuré cette archive avec deux éléments. Le premier étant un fichier XML que nous avons appelé "Manifest.xml", comprenant des indications sur toutes les modifications apportées par l'utilisateur depuis la dernière synchronisation. Le second est un dossier contenant l'ensemble des ressources de type Fichier créées et/ou modifiées par l'utilisateur .

6.1 Le manifeste

Le manifeste est un fichier que nous créons, il nous permet de stocker l'ensemble des informations nécessaires à la création ou à la mise à jour des espaces d'activités et des ressources qu'ils contiennent. Pour créer un tel fichier, deux possibilités efficaces, éprouvées et reconnues s'offraient à nous : le XML et le JSON. L'explication de ces deux alternatives et la justification de notre choix se trouvent ci-après.

6.1.1 Définition : EXtensible Markup Language (XML)

Le XML est un format de rédaction de document texte dérivé du standard (ISO 8879) [1]. Ce langage, fut conçu à l'origine pour répondre aux besoins de publications électroniques à grande échelle. Aujourd'hui, il occupe une place importante dans les échanges de données sur le WEB ou dans d'autres systèmes, citons par exemple son usage important sur la plateforme mobile Android.

Le rôle du XML est de permettre le transfert de données comparativement au *HyperText Markup Language* (HTML) qui sert quant à lui à afficher du contenu. Le langage XML autorise le développeur à créer ses propres balises, ce qui permet d'ajouter une couche de sémantique à la syntaxe accompagnant les données. De plus, ce langage est devenu une recommandation du *World Wide Web Consortium* en 1998.

6.1.2 Définition : JavaScript Object Notation (JSON)

De manière similaire au XML, JSON est un format de données textuelles permettant l'enregistrement structuré d'informations en vue de les transférer. La différence entre ces deux

1. http://www.w3.org/XML/

formats réside dans la syntaxe utilisée pour l'enregistrement. Effectivement `JSON` utilise une syntaxe proche de celle Javascript [2] mais est indépendant de ce langage de programmation.

6.1.3 Explication du choix

Etant donné que ces deux solutions sont équivalentes ; notre choix s'est porté sur le `XML` pour l'écriture du manifeste car il s'agit tout d'abord de la première solution complète correspondant à nos besoins. En effet, ces documents sont légers et autorisent un parsing efficace ; ils rencontrent donc les deux caractéristiques recherchées. Cette solution nous est naturellement venue à l'esprit car nous avions déjà travaillé avec ce format de données. De plus, il constitue un standard en la matière.

D'un point de vue implémentation, nous avons utilisé le *Document Object Model (DOM)*. Ce dernier est un standard du W3C [3] pour décrire une interface indépendante de tout langage de programmation qui permet à des programmes ou des scripts d'accéder ou de mettre à jour le contenu, la structure ou le style de documents XML et HTML[**?**]. Ce type d'interface disponible pour PHP, nous a permis de créer et analyser efficacement notre XML.

6.2 Implémentation

6.2.1 Services Taggés : l'injection des dépendances

Nous avons créé un `Service` pour chaque type de ressource de Claroline prise en compte par *OfflineBundle*. Un `Service` est un objet PHP effectuant une tâche particulière qui est utilisée tout au long de l'application. Le principal avantage à travailler avec les `Services` est qu'ils permettent de séparer les différentes fonctionnalités, ce qui est une bonne pratique en programation orientée objet.

Nos `Services` prennent la forme d'une classe PHP étendant une classe abstraite. Chacun de nos `Services` définit les fonctions nécessaires à la gestion de leur ressource par notre bundle. Pour identifier ces `Services`, nous leur apposons un `Tag`, c'est-à-dire une chaîne de caractères génériques. L'utilisation d'un `Tag` d'identification nous permet de demander au constructeur de conteneur de services [4] de nous rendre la liste de tous les `Services` possédant un `Tag` donné. Le `Tag` que nous apposons est `claroline_offline.offline`.

Lors de la passe de compilation nous sommes à même de rassembler les `Services` possédant notre `Tag` et d'en faire prendre connaissance à nos deux managers responsables de la gestion de l'archive que sont `CreationManager` et `LoadingManager`. Durant cette passe, une structure de données contenant l'ensemble des `Services` est créée. Tout ces `Services` seront indexés par le type de ressource qu'ils supportent, ce qui facilite grandement leur

2. Javascript est un langage de programmation de scripts utilisés sur les pages web interactives. Pour plus d'informations : https://developer.mozilla.org/fr/docs/JavaScript
3. W3C - http://www.w3.org
4. Voir lexique : 2.2 page 14

utilisation par la suite.

De plus, l'utilisation de services taggés nous permet de facilement ajouter de nouvelles ressources à celles déjà gérées par notre bundle. La section 6.5 de ce chapitre est consacré à l'explication de l'ajout d'un nouveau service destiné à gérer un nouveau type de ressource. Sur la Figure 6.1 se trouve le détail de nos `Services Offline`. Sur la partie supérieure du graphe se trouve notre classe abstraite `OfflineElement`, cette dernière comprend tous les attributs et méthodes utilisées par les autres classes `Offline`. Deux classes étendent cette dernière, `OfflineWorkspace`, qui comprend les méthodes relatives aux espaces d'activités et `OfflineResource`, qui est une classe abstraite qui reprend les cinq signatures de méthodes nécessaires à la synchronisation des ressources. Tout service destiné à un type particulier de ressource à synchroniser doit étendre `OfflineResource`.

6.2.2 Ecriture du manifeste

DOMDocument

Le manifeste prend la forme d'un objet `DOMDocument`. L'utilisation d'un `DOMDocument` nous permet d'ajouter facilement des éléments au document grâce aux fonctions `createAttribute()` et `appendChild()`. De plus, sur le `DOMDocument`, il existe une fonction `createCDataSection()` utilisée pour transférer le contenu ne devant pas être interprété en `XML` ce qui est le cas des ressources de type texte et des messages des ressources de type forum. Le rôle des section `CData` sera expliqué dans les parties dédiées à ces ressources.

Section Description

Dans un premier temps, nous écrivons au sein du manifeste une description globale nécessaire lors de l'étape de chargement. Avec ses informations nous serons capable de vérifier l'identité de l'utilisateur. Cette description comprend :

- la date de création de l'archive
- la date de dernière synchronisation de l'utilisateur
- le pseudonyme de l'utilisateur
- l'adresse e-mail de l'utilisateur

Section Ressources

Une fois l'écriture de la description terminée, nous ajoutons au manifeste l'ensemble des modifications apportées aux différents espaces d'activités dont est membre l'utilisateur. Nous avons décidé de ne conserver que les ressources modifiées ou créées car ce projet se destinant à être utilisé dans des zones ou la connectivité est faible, voire instable, chaque information à transférer compte d'autant plus. Il nous a donc paru important d'éviter au maximum de surcharger notre archive.

Avant de détailler davantage les spécificités de chaque ressource au point 6.3, il nous est nécessaire d'expliquer la manière dont les ressources sont schématisées dans la base de don-

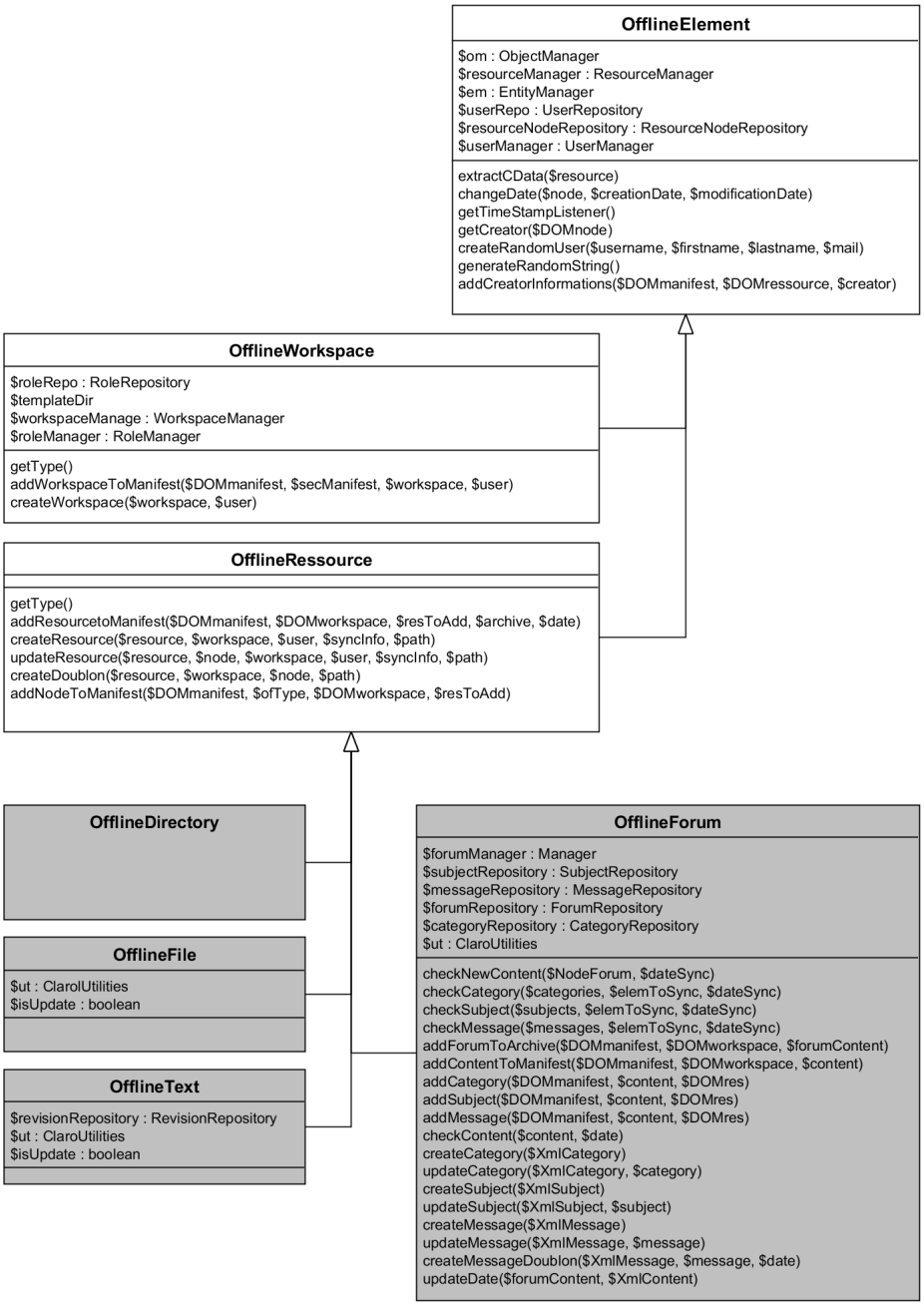

FIGURE 6.1 – Schéma UML des Services Offline

nées. Comme montré sur la FIGURE 6.2, la représentation d'une ressource en base de données se base sur l'entité `resource_node`. Celle-ci contient la référence vers le type de la ressource,

le `resource_type` de même que l'identifiant de l'espace d'activités, le `Workspace`, auquel cette ressource appartient. Par ailleurs, chaque ressource contient l'identifiant de l'entité `resource_node` qui lui correspond.

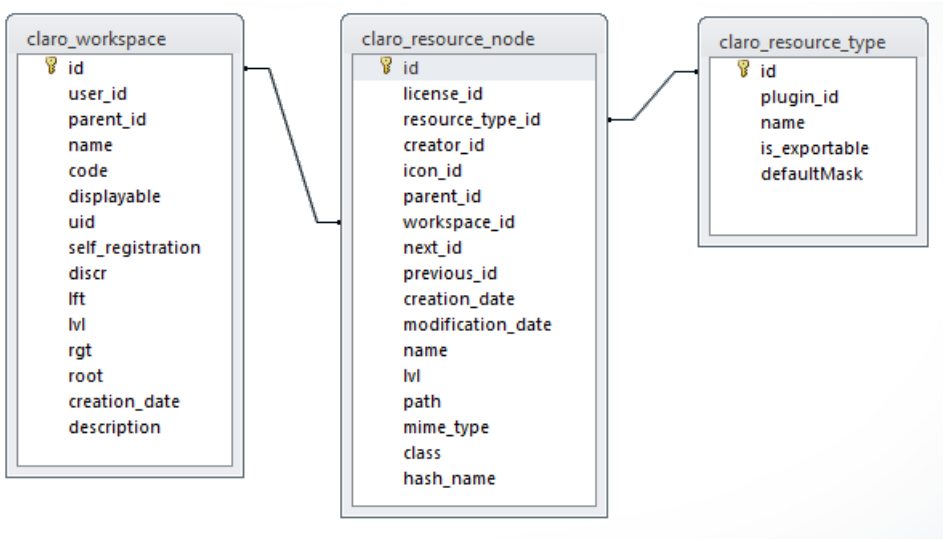

FIGURE 6.2 – Schéma général de la représentation d'une ressource en base de données

Lors de notre étape de création de l'archive, nous retrouvons les ressources à synchroniser comme suit. Nous effectuons une première requête sur la base de données dans le but de récupérer l'entièreté des espaces d'activités auxquels l'utilisateur est inscrit. Une fois cette liste d'espaces d'activités obtenue, nous utilisons une requête `DQL` conçue par nos soins qui a pour but de nous retourner en une seule fois toutes les ressources de cet espace d'activités qui sont à synchroniser et qui correspondent aux types de ressources gérés par nos services décrits ci-dessus. Une requête `DQL` est similaire à une requête `SQL`, la principale différence étant que l'on raisonne en terme d'objets.

```
private function findResourceToSync(Workspace $workspace,
                                    $types, $date)
{
    $query =
        $this->resourceNodeRepo->createQueryBuilder('res')
        ->join('res.resourceType', 'type')
        ->where('res.workspace = :workspace')
        ->andWhere('res.modificationDate > :date')
        ->andWhere('type.name IN (:types)')
        ->setParameter('workspace', $workspace)
```

```
            ->setParameter('types', $types)
            ->setParameter('date', $date)
            ->getQuery();

        return $query->getResult();

}
```

Afin d'obtenir les ressources à synchroniser nous paramétrons la requête `DQL` avec la date de dernière synchronisation (`$date`). Pour rappel, les ressources à synchroniser sont celles dont la date de modification est postérieure à la date de synchronisation de l'utilisateur.

Pour s'assurer que le type de la ressource correspond aux types gérés par nos `Services`, nous passons à la requête un tableau comprenant tous les types supportés. Ce tableau, `$types`, est obtenu en utilisant la fonction `array_keys()` qui retourne l'ensemble des clés d'un tableau. La fonction `array_keys()` est appliquée dans notre cas à la variable `$offline` de notre manager, qui est la structure de données comprenant tout nos `Services taggés` indexés par le type de ressource.

Ainsi nous ne conservons que les ressources ayant été créées et/ou modifiées depuis la date de dernière synchronisation de l'utilisateur. Ressources qu'il est par conséquent nécessaire de transmettre à l'autre plateforme.

Il nous reste maintenant à ajouter au manifeste les informations utiles à l'étape de chargement (cfr. 8 page 61). Cela comprend notamment le type de la ressource, sa clé unique appelée `hashname` en base de données ainsi que son créateur. L'opération s'effectue simplement grâce à la structure de données `$offline` citée précédemment. En effet, étant donné qu'elle contient nos services indexés par leur type, on peut facilement accéder à la méthode d'écriture du manifeste du bon service de la manière suivante :

```
foreach ($ressourcesToSync as $res) {

    $domManifest =
        $this->offline[$res->getResourceType()->getName()]
        ->addResourceToManifest(...);

}
```

Où `$res->getResourceType()->getName()` nous donne le nom, sous forme d'une variable string, du type de la ressource.

6.3 Spécificités des différents types de ressources

Ci-dessous se trouvent les spécificités propres à chaque type de ressource et la manière dont nous les avons traitées. De plus, pour chaque ressource nous citons le nom du `Service` associé. Ces `Services` peuvent être retrouvés dans le dossier `Model\Resource` de notre *OfflineBundle*.

6.3.1 Ressource `Texte`

Lorsque nous avons une ressource de type `Texte`, il est également nécessaire d'aller chercher en base de données le texte auquel elle fait référence. Comme illustré sur la FIGURE 6.3, ce texte est stocké sous l'entité `text_revision`. Comme vu précédemment, nous sommes capables de retrouver l'entité `text` sur base du `resource_node` qui lui est lié. A partir de cette entité `text`, nous pouvons atteindre l'entité `text_revision` désirée.

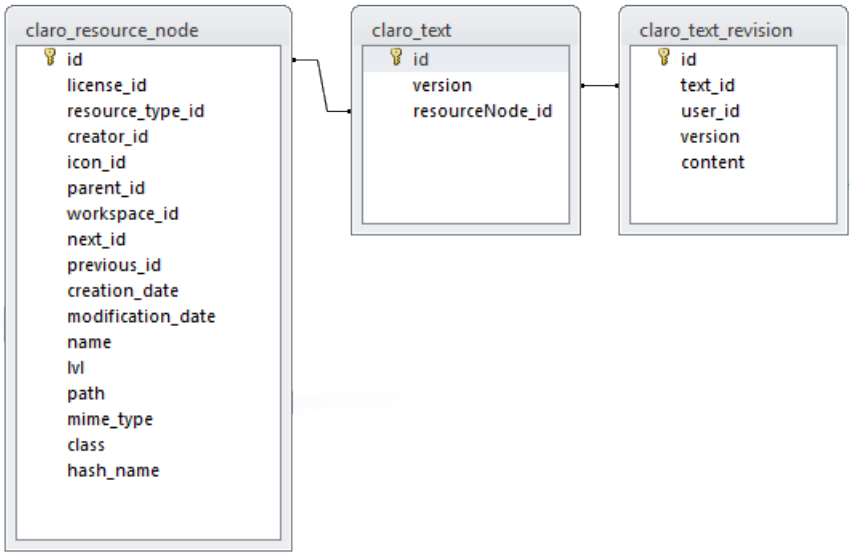

FIGURE 6.3 – Schéma d'une ressource `Texte` dans la base de données

Comme expliqué dans la définition[5], les ressources textes ont la particularité de pouvoir être riches. Toutefois cette richesse fut pour nous une source de complexité dans notre méthode d'enregistrement. Par exemple, s'il y a présence de balises `HTML` à l'intérieur du contenu du texte, cela cause une erreur. En effet, il ne peut y avoir présence de balises au sein d'un attribut `XML`.

5. cfr. Lexique, CHAPITRE 2.1 page 13

Pour ce problème, le `XML` propose une solution : la section `CDATA`. La particularité de cette section est de ne pas interpréter leur contenu. Celle-ci offre la particularité de ne pas intérpréter les balises contenues à l'intérieur jusqu'à la chaine de caractère fermante qui bien entendu ne peut être utilisée comme séquence au sein du contenu. C'est-à-dire que ce dernier est considéré comme du texte et non pas comme du `XML`. Par ailleurs, il n'est pas autorisé à une section `CDATA` d'en contenir une autre.

La gestion des ressources textes dans notre programme est entièrement implémentée par le `Service OfflineText`.

6.3.2 Ressource `Fichier`

Pour rappel, une ressource du type `Fichier` est un document chargé au sein d'un espace d'activités par un de ses gestionnaires[6]. Ces ressources ont besoin d'une information supplémentaire dans le manifeste : la taille du fichier. De plus, ces ressources font appel à un document chargé par l'utilisateur. Ce document est placé au sein d'un répertoire particulier par exemple *files*[7]. La FIGURE 6.4 nous montre que l'entité `file` contient une clé `hashname` qui nous permet de retrouver le répertoire au sein duquel se trouve le document.

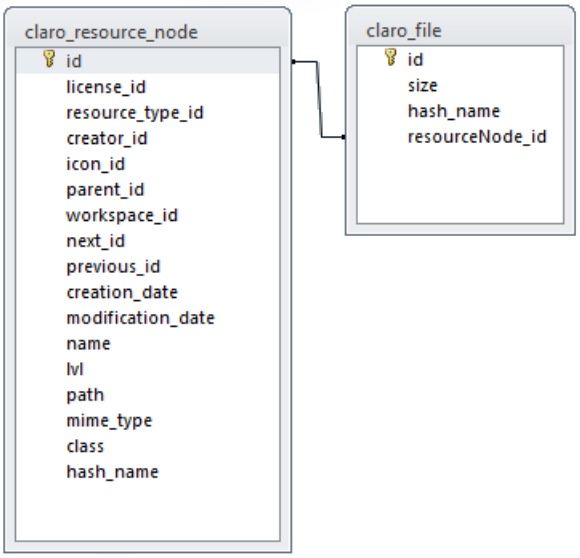

FIGURE 6.4 – Schéma d'une ressource `Fichier` dans la base de données

Il nous est nécessaire en cas de modification ou création d'incorporer ce document à notre archive. Comme l'entité `file` connait le nom du document au sein du répertoire de la plateforme, il nous est alors possible de le retrouver et d'en faire une copie à l'intérieur d'un dossier de notre archive que nous avons nommée `data`.

6. cfr. Lexique, CHAPITRE 2.1 page 13
7. Le chemin complet de ce dossier est défini dans le fichier parameters.yml du `CoreBundle` de *Claroline*

La Service responsable de la gestion des ressources de type fichier est OfflineFile.

6.3.3 Ressource Répertoire

Une ressource Répertoire[8] correspond à un dossier au sein d'un système d'exploitation classique, il s'agit d'une ressource en rassemblant d'autres. Chaque ressource, tout type confondu, se trouve à l'intérieur d'une ressource Répertoire ou d'un espace d'activités, ce qui forme une hiérarchie cohérente sous forme d'arborescence. Le point d'entrée de cette arborescence étant l'espace d'activités. Cela signifie qu'une ressource Répertoire peut-être contenue dans une ressource Répertoire mais qu'en revanche, un espace d'activités ne peut quant à lui être contenu dans un Répertoire.

En base de données, cette hiérarchie est symbolisée par le fait que chaque resource_node connaisse l'identifiant de son parent. C'est-à-dire de la resource_node correspondant à la ressource Répertoire ou à l'espace d'activités la contenant directement. La gestion de ces ressources ne nous a pas posé de problème particulier. Néanmoins, lors de l'analyse de ces ressources, nous avons découvert le problème des parents disparus dont nous discutons dans le chapitre du chargement (cfr. CHAPITRE 8.7.2 page 67). Le Service OfflineDirectory prend en charge la gestion des ressources répertoires.

6.3.4 Ressource Forum

Pour les ressources Forums, implémentées dans OfflineForum, une procédure différente est requise. Comme vu dans le Lexique, un forum est composé de Catégories, elles-mêmes composées de Sujets contenant un ou plusieurs Messages (illustration FIGURE 6.5)[9]. Du fait de cette structure particulière, il nous a fallu trouver un autre moyen de détecter les modifications apportées aux ressources Forums. En effet, dans la base de données, lorsqu'une nouvelle Catégorie, un nouveau Sujet ou Message est créé sur un forum, la date de modification de la resource_node correspondant à ce dernier n'est pas mise-à-jour. Par conséquent, lorsque nous traitons les ressources de type Forum, nous n'avons aucun moyen de savoir si le contenu de cette ressource a été modifié.

Pour résoudre ce problème, nous faisons appel à une sous-méthode qui va parcourir l'ensemble des Forums. Pour chaque forum nous trouvons, à l'aide de requêtes, l'ensemble des catégories, sujets et messages le composant. Nous parcourons ensuite ces élements et à chaque fois qu'une catégorie, qu'un sujet ou qu'un message ont été créés ou modifiés ; c'est-à-dire qu'elle possède en base de données une date de modification ultérieure à la date de dernière synchronisation de l'utilisateur, nous l'ajoutons à une liste dont les éléments seront par la suite ajoutés au manifeste.

Il est à noter que, tout comme pour les ressources Texte, le contenu des messages des forums contiennent des balises HTML. Par conséquent, ils nous a aussi été nécessaire de placer

8. cfr. Lexique, CHAPITRE 2.1 page 13
9. cfr. Lexique, CHAPITRE 2.1 page 13

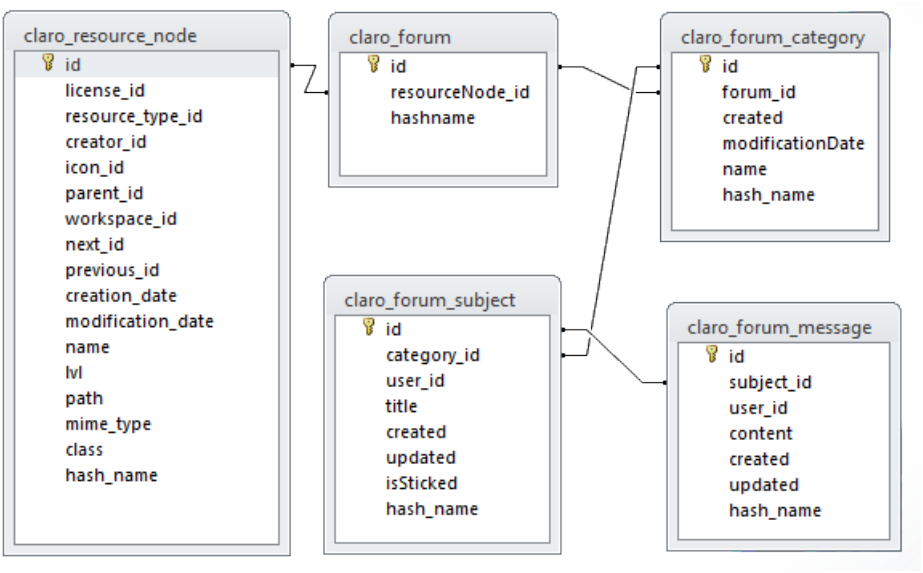

FIGURE 6.5 – Schéma d'une ressource `Forum` dans la base de données

ce contenu dans une section `CDATA` au sein du manifeste.

6.3.5 Espaces d'activités

Un espace d'activités n'est pas une ressource à proprement parler. Toutefois, il est similaire à une ressource `Répertoire`, en ce sens que l'entité `resource_node` lui correspondant est du type 'repertoire' et qu'il est destiné à contenir d'autres ressources. De plus, les `resource_node` correspondant aux espaces d'activités ont la particularité d'avoir un champ `parent` NULL et de cette manière, constituent la racine de l'arborescence des ressources. En outre, les espaces activités définissent les rôles des utilisateurs, c'est-à-dire les possibilités d'actions dont ils disposent sur l'ensemble des ressources qu'ils contiennent. Ces rôles prennent la force d'entités `claro_role` au sein de la base de données. Le lien entre un utilisateur et le rôle dont il dispose sur un espace d'activités est fait au sein de la table `claro_user_role`. Pour compléter le panel d'informations nécessaires à l'entité `Workspace`, une table `claro_workspace` existe. La Figure 6.6 illustre le schéma d'un rôle en base de données.

6.4 Récapitulatif

Review ? pas de changement avec listener, liste de ressources et rôles ?

A la fin de l'étape de création, nous avons une archive contenant :

1. Un fichier `XML` divisé en plusieurs sections :
   - Une section `Description` contenant les métadonnées[10] du manifeste.

10. cfr. Lexique, CHAPITRE 2.2 page 14

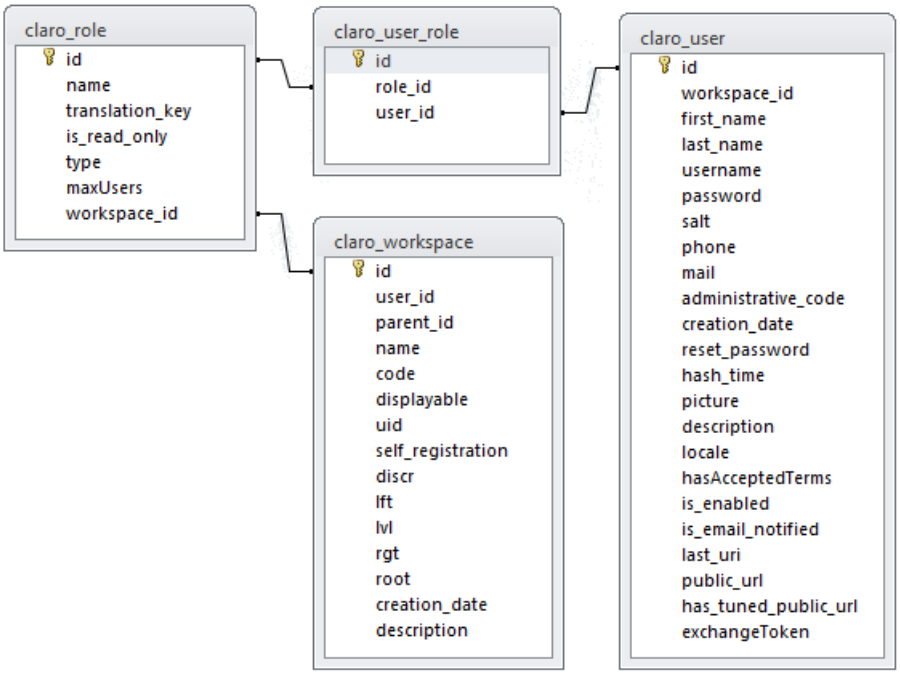

FIGURE 6.6 – Schéma d'un rôle dans la base de données

- Des sections `Workspace` dont les attributs contiennent les informations nécessaires pour retrouver ou créer les espaces d'activités. Au sein de ces sections se trouvent également les éléments à synchroniser du `Workspace` :
  - Des éléments `Ressources`
  - Des éléments de descriptions des `Forums`

2. Un dossier `data` contenant les éventuelles ressources `Fichier` à transférer.

Tous les éléments de cette archive ne concernent que les ressources et espaces d'activités modifiés ou créés depuis la date de dernière synchronisation de l'utilisateur, date obtenue par l'objet `user_sync` créé par nos soins.

6.5 Ajout d'une nouvelle ressource

Notre bundle ne gérant pas l'entièreté des ressources et le nombre de celles-ci étant amené à augmenter, cette section est ici à titre d'exemple supplémentaire pour quiconque souhaiterait ajouter une ressource parmi celles gérées par `OfflineBundle`.

Ajouter une nouvelle ressource est relativement simple. Dans un premier temps, il faut créer une classe PHP dans le dossier `Claroline/OfflineBundle/Model/Resource`. Idéalement cette classe devrait se nommer `'Offline'` suivi du type de la ressource (par exemple :

`OfflineWiki.php`). Une fois ce fichier créé, il faut s'assurer de plusieurs choses.

Premièrement, afin que cette classe soit un `Service`, il faut faire usage de la librairie : `JMS\DiExtraBundle\Annotation`. Cette dernière permet d'utiliser les annotations au sein de la classe ce qui facilitera grandement le travail. Les anontations permettent d'ajouter des repères pour la compilation. Dans notre cas, il faut ajouter `Service ("claroline_offline.offline.NOM_DU_SERVICE")`, (par exemple : `Service ("claroline_offline.offline.wiki")`) avant l'en-tête de déclaration de classe.

Deuxièmement, le `Service` doit étendre la classe abstraite `OfflineRessource`. Etendre cette classe va permettre d'utiliser les fonctions et arguments communs aux autres `Services`. Pour que l'implémentation soit valide, il faut également redéfinir cinq méthodes nécessaires pour la synchronisation :

`getType()` : Cette méthode doit renvoyer le type de la ressource sous forme de string (par exemple `'text'` pour les ressources de type `Text` ou `'claroline_forum'` pour les ressources de type `Forum`). La liste des noms des différents types de ressources se trouve en base de données. Cette fonction sera utilisée afin d'indexer le `Service` et d'y faire référence lors de la gestion de l'archive, comme expliqué précédemment 6.2.1 page 35. Par exemple, on peut voir que le type des ressources `Wiki` est `'icap_wiki'`.

```
// Return the type of resource supported by this service
public function getType(){
     return 'example';
 }
```

`addResourceToManifest()` : Cette méthode va être appelée lors de la phase de création de l'archive. Il faut y indiquer les éléments qui sont intéressants et nécessaires pour de la synchronisation. Idéalement cette méthode devrait faire appel à la méthode `addNodeToManifest()` de la classe OfflineResource, cette dernière se charge déjà d'ajouter les informations nécessaires à la synchronisation du `resource_node` associé à la ressource. Cette étape demande un certain travail d'investigation puisqu'il faut savoir quelles sont les informations dites importantes. Dans le cas d'une ressource `Wiki`, il nous faut aller voir le code du bundle traitant cette ressource et analyser les entités persistées.

Nous pouvons voir qu'une entité `Wiki` est composée de plusieurs entitées `Sections` ; ces `Sections` étant elles-même complétées par des `Contributions`. Pour que ces ressources `Wiki` soient gérées par `OfflineBundle`, il faudrait enregistrer dans le manifeste les informations nécessaire pour reconstruire les entités précédement citées. En plus des trois entités spécifiques aux `Wiki`, une ressource node est créé de manière

similaire à toutes les autres ressources de Claroline.

```
public function addResourceToManifest($domManifest,
    $domWorkspace, ResourceNode $resToAdd,
    ZipArchive $archive, $date)
{
     parent::addNodeToManifest($domManifest,
         $this->getType(), $domWorkspace, $resToAdd);
     /*
        *   Ajouter ici les informations supplementaires
        *   necessaires a la bonne gestion
        *   de la ressource par la suite.
        */
     return $domManifest;
}
```

createResource() : Cette méthode va être appelée durant la phase de chargement de l'archive[11] s'il s'avère que la ressource a besoin d'être créée. Il faut initialiser ici les champs nécessaires à la création de la ressource désirée et appeler la méthode `create()` du `ResourceManager` qui se chargera de créer la resource. La méthode `createResource()` renvoie un objet de type SyncInfo, qui est une structure de données que nous avons créée et qui sera expliquée dans le chapitre dédié au chargement de l'archive[12].

```
    public function createResource($resource,
            Workspace $workspace, User $user,
            SyncInfo $wsInfo, $path)
    {
        /*
        *   Ajouter ici les operations necessaires
        *   a la re-creation de la ressource.
        */
        return $wsInfo;
    }
```

updateResource() : Cette méthode va être appelée durant la phase de chargement de l'archive s'il s'avère que la ressource existe déjà et doit donc être modifiée. Il faut ici gérer une éventuelle mise-à-jour de la ressource. Cette méthode est également susceptible d'appeler la méthode `createDoublon()` expliquée ci-après.

11. cfr. CHAPITRE 8.7.2 page 67
12. cfr. CHAPITRE 8.7.2 page 67

```
public function updateResource($resource,
    ResourceNode $node, Workspace $workspace,
    User $user, SyncInfo $wsInfo, $path)
{
     /*
      * Ajouter ici les operations necessaires
      * a la mise-a-jour de la ressource.
      */
      return $wsInfo;
}
```

`createDoublon()` : Cette méthode est appelée lorsqu'une mise-à-jour d'une ressource est nécessaire mais que cette ressource a également été modifiée sur l'autre plateforme. Cette méthode peut servir à réaliser un compromis. Par exemple, pour nos ressources de type `Text` nous créons une nouvelle ressource et lui ajoutons un tag `'@offline'` pour les différencier. Ainsi sur la plateforme en ligne, il est aisé de faire la différence entre la ressource modifiée en ligne et celle modifiée hors-ligne.

```
public function createDoublon($resource,
     Workspace $workspace, ResourceNode $node, $path)
{
    /*
    *   Rajouter ici les operations necessaires
    *   a creation d'un doublon pour la ressource
    *   si necessaire.
    */
    return;
}
```

Finalement, il faut écrire le constructeur de ce `Service` afin d'initialiser tous les attributs déclarés par sa classe mère et tous les attributs spécifiques dont il aurait besoin. Les attributs indispensables au bon fonctionnement du `Service` et déclarés dans la classe mère sont les suivants :

`ObjectManager om` : Cet objet permet au `Service` d'accéder aux différents dépôts (`Repository`). Un dépôt permet d'effectuer des requêtes sur un type particulier d'objet. Il existe un dépôt par entité (User, ResourceNode, Workspace, ...)

`ResourceManager resourceManager` : Cet objet va permettre au `Service` d'accéder aux différentes fonctions appliquables aux ressources (comme la fonction de création par exemple)

`UserManager userManager` : Cet objet va permettre au `Service` d'accéder aux différentes fonctions appliquables aux utilisateurs.

`EntityManager em` : Cet objet n'est utilisé qu'afin de mettre à jour les différentes dates de création et modification. Son utilité sera expliquée dans le chapitre dédié au chargement ((cfr. CHAPITRE 8.7.2 page 67)

Une fois ces étapes effectuées, nous avons donc un service capable de gérer une nouvelle ressource, dans notre cas les `Wiki`. Une dernière étape à effectuer est d'ajouter notre tag à ce service. Comme expliqué précédemment, ce tag sera repéré lors de passe de compilation et le service sera alors accessible dans notre `CreationManager` et notre `LoadingManager`. Ceux-ci vont prendre automatiquement connaissance du nouveau type de ressource à gérer. Il est en effet peu intéressant de s'occuper des ressources d'un type non-pris en compte par notre Bundle.

Vous trouverez un exemple complet de ce à quoi doit ressembler un service désireux d'être pris en compte par notre Bundle en Annexe A page III.

Chapitre 7

Transfert de l'archive

Dans ce chapitre nous aborderons en détail le fonctionnement du transfert de données entre les deux plateformes. Les objectifs de cette tâche étaient donc de pouvoir assurer qu'une archive créée au sein d'une plateforme (déconnectée ou connectée) soit transférée de manière intégrale, complète et sans erreur à l'autre plateforme. Au sein du processus global détaillé précédemment, nous avons vu que le transfert est utilisé principalement deux fois. La première fois pour charger sur la plateforme connectée l'archive créée sur l'ordinateur personnel. Et la seconde fois, pour télécharger l'archive de synchronisation de la plateforme en ligne afin de l'enregistrer sur la plateforme déconnectée.

Ce problème est complexe car le transfert doit être autant que possible résistant aux coupures et utilisable sur des connexions à faible bande passante. De plus, celui-ci doit être effectué dans la mesure du possible de manière sécurisée. L'implémentation du transfert est principalement réalisée dans deux classes PHP : `SynchronisationController` contenant l'implémentation des routes que nous contactons et `TransferManager` contenant les fonctions utilisées pour les routines d'exécution.

7.1 `TransferManager`

7.1.1 Des requêtes `POST`

Une requête `POST` est un des types de requêtes possible du protocole `HTTP`. Une requête `POST` est utilisée pour faire en sorte qu'un serveur web accepte des données incluses dans le corps de la requête et destinées à être stockées. Elle est principalement utilisée pour charger un fichier ou soumettre un formulaire. Dans notre cas, nous nous en sommes servis pour charger l'archive que nous avons construite dans l'étape de création.

7.1.2 *Buzz* et `JSON`

Les outils utilisés pour réaliser la communication entre les deux serveurs sont *Buzz* et `JSON`. *Buzz* est un client `HTTP` distribué sous la forme d'un bundle de *Symfony*. Désireux de faire un travail s'intégrant facilement à la plateforme, nous avons utilisé *Buzz* car les développeurs de Claroline s'en servent déjà. De plus, celui-ci remplissait parfaitement nos attentes. `JSON` quant à lui est utilisé pour structurer le contenu de la requête.

Nous utilisons *Buzz* afin de créer les requêtes. Pour cela, nous créeons un client `Curl` afin d'obtenir un objet `Browser` sur lequel nous pouvons exécuter nos requêtes `POST`. Ces

dernières nécessitent une route à contacter et le contenu à transférer. *Buzz* impose que le contenu soit formaté en String, c'est pourquoi nous utilisons la fonction `json_encode()`. Cette fonction est un outil très pratique car elle permet de transformer un tableau `PHP` en un string structuré qui pourra aisément être décodé à l'aide de la fonction `json_decode()` par la suite.

Lors de l'envoi nous construisons un tableau contenant l'ensemble des métadonnées[1] ainsi que l'archive qui est envoyée à l'autre plateforme. C'est donc ce tableau qui est encodé en `JSON` et décodé à la réception. Un champ du tableau que nous envoyons contient l'archive sous forme de données binaires. Afin de ne pas l'interpréter, il faut convertir cette séquence binaire en texte.

Pour palier à ce problème nous utilisons la fonction `base64_encode()` à l'envoi et `base64_decode()` à la réception. L'objet de la fonction `base64` est précisément de transformer du contenu binaire en un encodage 8bits afin qu'il soit supporté dans divers type d'applications telles que les emails ou l'échange de données sur Internet.[**?**]

7.2 `SynchronisationController`

7.2.1 Les routes

Comme nous venons de l'expliquer, l'ensemble des échanges d'informations se fait au travers de routes. Ces dernières sont implémentées au sein de `SynchronisationController` car nous avons choisi de laisser l'accès libre aux routes de ce controlleur. Dans le cas contraire, nous n'aurions pas été à même de contacter les routes de ce controlleur sans être connecté sur la plateforme distante. Pour garantir la sécurité, nous avons mis en place un système d'authentification avec les requêtes. Ce système d'authentification est expliqué dans la section 7.4 de ce chapitre.

L'ensemble des échanges effectués entre les deux plateformes l'est toujours à la demande du client. Les données devant être transférées de la plateforme hors-ligne vers la plateforme en ligne sont transmises au sein du contenu de la requête. Tandis que les données transférées depuis la plateforme en ligne vers la plateforme hors-ligne arrivent sur la plateforme hors-ligne par l'intermédiare de la réponse à la requête `HTTP`. L'échange se fait à la demande du client car la plateforme déconnectée ne dispose pas d'une adresse sur laquelle le serveur en ligne peut la contacter.

Voici la liste exhaustive des routes que nous avons implémentées dans `SynchronisationController` suivie pour chacune d'entre elles d'une brève description.

`claro_sync_upload_zip` : Cette route est contactée pour charger une archive de synchronisation sur la plateforme en ligne.

1. Voir lexique chapitre 2.2 page 14

`claro_sync_get_zip` : Cette route est contactée pour télécharger une archive de synchronisation depuis la plateforme connectée vers la plateforme déconnectée.

`claro_sync_unlink` : Cette route permet de supprimer une archive présente sur la plateforme connectée. Elle est utile en fin de synchronisation pour effacer les fichiers devenus inutiles.

`claro_sync_user` : Cette route permet de récupérer les informations utiles à la création d'un utilisateur déconnecté.

`claro_sync_last_uploaded` : Cette route sert à la reprise d'un processus de synchronisation, elle permet de connaître le dernier fragment qui a été chargé en ligne. La découpe en multiples fragments est expliquée dans la section 7.3 de ce chapitre.

`claro_sync_number_of_packets_to_download` : Cette route permet à la plateforme déconnectée de connaître le nombre de paquets à télécharger. Elle est également utilisée pour l'envoi d'une archive en de multiples fragments.

`claro_sync_config` : Cette route est celle utilisée pour la configuration d'un profil utilisateur.

7.2.2 Mécaniques de tests

La communication entre deux plateformes n'est pas une action habituelle, il a donc fallu réfléchir à la manière dont nous allions pouvoir tester son fonctionnement.

Client `REST`

Afin de tester les routes que nous avons créées dans `SynchronisationController`, nous avons utilisé un add-on pour le navigateur Google Chrome (*Simple REST Client*[2]) nous permettant de formuler des requêtes `POST`. En effet, à la différence des requêtes `GET` qui sont elles, destinées à aller récupérer des informations, les requêtes `POST` ne peuvent être générées directement par le navigateur Internet.

Environnement de développement

Afin de tester le transfert entre deux serveurs, nous avons dupliqué la plateforme. Le serveur web de développement sur notre ordinateur personnel comportait donc deux Claroline. Appelons *Claroline 1* la plateforme représentant la plateforme déconnectée et *Claroline 2* la plateforme connectée. *Claroline 1* possédait l'adresse du serveur local à contacter pour atteindre *Claroline 2*. Procéder comme tel nous a permis de constater facilement le bon fonctionnement du processus puisqu'il suffisait de vérifier si les résultats étaient ceux escomptés (création et mise à jour des ressources).

Nous avons alors pu faire des tests plus complexes, en exécutant les actions de *Claroline 1* qui contactent *Claroline 2* et observer le résultat. Pour s'assurer du bon fonctionnement

2. Télécharger Simple REST Client - https://chrome.google.com/webstore/detail/simple-rest-client/fhjcajmcbmldlhcimfajhfbgofnpcjmb/reviews

et de l'indépendance des deux plateformes, nous avons également fait des tests en plaçant chacune des plateformes sur des serveurs web différents. Dans notre cas nous avons utilisé *IIS* et *XAMPP*. Cette séparation a été possible car nous avons démarré les serveurs sur des ports différents.

7.3 Division en multiples fragments

Pour que le transfert soit fonctionnel sur une connexion Internet de faible débit, il nous a fallu diviser l'archive en plus petits éléments que nous appelons fragments. De cette manière, un minimum de données sont perdues en cas de coupure de la connexion, ce qui signifie également qu'il ne faudra réenvoyer qu'une petite quantité de données. Sans cette découpe de l'archive en plusieurs fragments, l'ensemble de l'archive de synchronisation devrait être retransférée en cas d'échec. De plus l'envoi s'effectuerait en un seul bloc, ce qui peut poser problème lorsqu'elle est volumineuse ou que la connexion est très instable.

Nous avons fixé la taille de ces fragments à 512Ko. Cette valeur a été déterminée au travers de quelques tests de performance que nous décrivons ci-dessous. Si l'administrateur souhaite adapter celle-ci car elle ne correspond pas à sa situation ou qu'il rencontre des problèmes de performance, il suffit de changer la valeur de la constante `MAX_FRAG_SIZE` du fichier `SyncContant`.

7.3.1 Détermination de la taille des fragments

Afin de déterminer la taille idéale des fragments nous avons effectué un échange répété d'une archive de synchronisation entre deux plateformes Claroline. Les variables de cet échange furent la taille de l'archive transportée et la taille des fragments créés par la plateforme. Pour ce test les deux plateformes Claroline étaient installées localement sur nos ordinateurs de développement, les temps calculés sont donc essentiellement fonction de la performance de nos ordinateurs de travail et non de la qualité de la transmission entre les deux plateformes Claroline, celle-ci étant pratiquement parfaite.

Les conditions du tests étaient les suivantes : nous avons transféré dix fois chaque archive et pris la moyenne du temps de transfert. Le temps de transfert fut calculé comme étant la différence entre le temps précédent le début du transfert et le moment où la confirmation de réception du dernier fragment est reçue. Nos cinq archives transférées avaient approximativement les tailles suivantes : ~1Mo (856Ko), ~5Mo (4581Ko), ~10Mo (10169Ko), ~25Mo (25072Ko) et ~50Mo (49288Ko). La valeur entre parenthèses correspond à la taille exacte des archives que nous avons utilisées pour nos tests. La taille des fragments a, quant à elle, eu les quatre valeurs suivantes : 128Ko, 256Ko, 512Ko et 1024Ko.

La Figure 7.1 et la Figure 7.2 sont deux illustrations des résultats, que nous avons obtenus. Avant d'analyser ces résultats il est important de rappeler que le temps de transfert

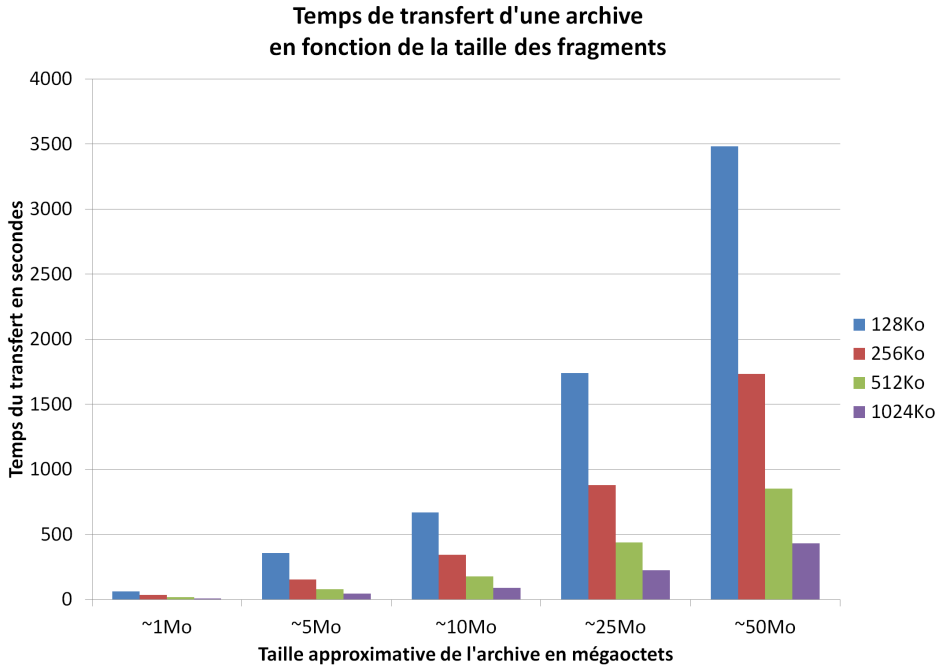

FIGURE 7.1 – Graphique du temps de transfert d'archives de synchronisation en fonction de la taille des fragments

est directement proportionnel aux performances de l'ordinateur ayant servi pour les tests. Ce qu'il est intéressant d'observer sur ces graphes est le gain relatif que l'on obtient en faisant varier la taille des fragments pour une archive de taille donnée et non le temps de transfert comme valeur absolue.

Les résultats obtenus par notre méthode de test sont cohérents. En effet, en diminuant significativement la taille des fragments créés par notre système de transfert, le nombre de requêtes envoyées à la seconde plateforme va croitre proportionnellement. La Figure 7.1 illustre clairement l'augmentation du temps nécessaire pour le transfert d'une archive de plus grande taille et le gain que peut apporter une découpe en fragments plus grands. En effet, nous pouvons voir que passer de fragments de 256Ko à 512Ko divise le temps de transfert d'une archive par deux. Ce gain est d'autant plus intéressant que le temps de transfert requis est élevé, autrement dit que la taille de l'archive augmente.

Comme nous pouvons l'observer sur la Figure 7.2, le temps de transfert diminue de manière exponentielle lorsque l'on augmente la taille des fragments. Toutefois il est important de nuancer cette affirmation car même si elle n'est pas présente sur notre graphe, il doit exister une limite supérieure. En effet, si la taille maximale est augmentée de manière trop importante, des erreurs de transfert surviendront. Il faudra dès lors réenvoyer les fragments

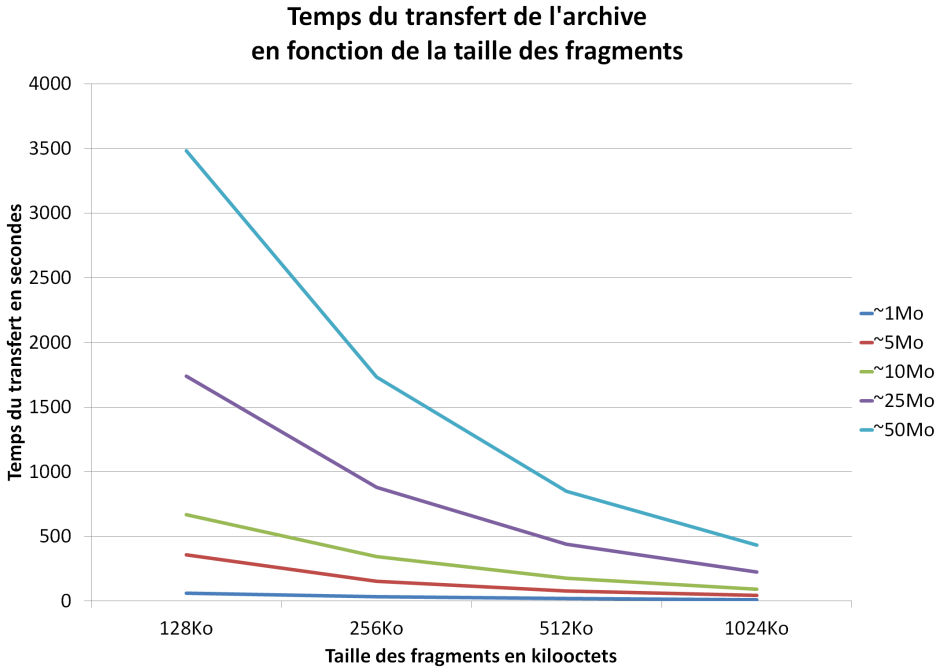

FIGURE 7.2 – Graphique du temps de transfert d'archives de synchronisation en fonction de la taille des fragments

et le temps total augmentera. Afin d'être complet, le Tableau 7.1 contient l'ensemble des résultats que nous avons obtenus. Il montre pour chaque taille de fragments et pour chaque taille d'archive de synchronisation le temps de transfert ayant été mesuré sur l'ordinateur de test.

Finalement, pour établir le choix de la taille par défaut des fragments au sein du *OfflineBundle*, nous sommes partis de l'hypothèse que l'utilisateur du bundle se trouve dans un pays à faible connectivité et dont la bande passante est limitée. Mettons le débit de la connexion internet comme vallant 56Ko/s soit une connexion que l'on pourrait qualifier de mauvaise. De plus, la taille maximale par défaut des fichiers dans Claroline étant fixée à 2Mo, nous avons posé comme hypothèse que l'archive de synchronisation devrait contenir moins de 25Mo de données. Cette hypothèse nous semble englober la plupart des cas. Sur base de nos mesures, nous avons réglé la taille du fragment par défaut à 512Ko.

Partant des hypothèses que nous venons de décrire, le choix de fragments à 512Ko signifie qu'un fragment serait transféré en moins de 10 secondes sur une ligne à faible débit (56Ko/s). Cette situation semble acceptable dans la mesure où nous fixons le timeout de la requête `Curl` effectuée avec Buzz à 60 secondes. De plus, les fragments de 512Ko montrent des performances acceptables pour les archives de moins de 25Mo dans nos tests. Enfin, ce

Taille des fragments	Taille de l'archive de synchronisation				
Ko	~1Mo	~5Mo	~10Mo	~25Mo	~50Mo
128	61s	356s	669s	1740s	3481s
256	33s	153s	342s	880s	1734s
512	18s	79s	176s	438s	850s
1024	9s	45s	90s	224s	432s

TABLE 7.1 – Résultat des tests de transfert d'archives de synchronisation en faisant varier la taille de l'archive et la taille des fragments

choix nous semble être l'équilibre idéal entre performances et stabilité.

7.3.2 Modifications dans l'implémentation

Pour permettre l'envoi d'une archive en plusieurs fragments des fonctions spécifiques ont été créées dans le `TransferManager`. La fonction, `getTotalFragments($filename)`, permet de déterminer le nombre de fragments à transférer pour une archive donnée. La fonction `getFragment($fragmentNumber, $filename, $user)` retourne quant à elle le nième fragment demandé. Ces informations permettent de compléter les métadonnées envoyées avec une requête. La plateforme réceptionnant la requête connaîtra maintenant le numéro du fragment par rapport au total de fragments à transférer.

Une fonction est également responsable d'enregistrer le fragment reçu sur le disque dur. Lorsque l'ensemble des fragments a été transféré, une fonction se charge de les rassembler afin de reconstruire l'archive initiale. Pour en vérifier l'intégrité, nous utilisons une fonction de hashage établie sur le fichier initial. Cette information est enregistrée dans les métadonnées transmises à la plateforme. Cette dernière pourra donc comparer le résultat de sa propre fonction de hashage sur le fichier reconstitué avec celui figurant dans les métadonnées. Si le hash est identique entre le fichier d'origine et le fichier reconstitué, nous concluons que ces deux-ci ont été correctement transférés. Dans le cas contraire une erreur est retournée à la plateforme déconnectée, ce qui entrainera le redémarrage du transfert.

En cas de redémarrage du processus de transfert après une interruption volontaire ou non, le dernier paquet transféré peut être retrouvé. Cette information n'est pas enregistrée dans la base de données contrairement au status du processus global, mais nous pouvons la déterminer en regardant au sein des fichiers dans le répertoire où sont stockées temporairement les données. Pour ce faire, la fonction `getDownloadStop($filename, $user)` du `SynchronisationManager` établit le dernier fragment transféré sur base du nom du fi-

chier et du contenu du dossier temporaire. Sachant que chaque fragment enregistré possède un nom de fichier construit sur base du nom de l'archive concaténée avec son numéro de fragment.

7.3.3 Stockage temporaire

Afin de stocker les paquets transmis lors d'un échange, nous avons fixé deux sous-répertoires du dossier `claroline.param.files_directory`. Le premier sert à stocker les archives qui vont être envoyées à la plateforme en ligne et le deuxième à stocker l'archive téléchargée. Le chemin d'accès de ces deux répertoires sont paramétrables, est fixé au sein de `SyncConstant` [3]. Le répertoire `DOWN` contient l'archivé créée par la plateforme et le répertoire `UP` contient celle qui est téléchargée depuis l'autre plateforme. Ces deux dossiers sont également divisés en sous dossier dont le nom est l'`id` de l'utilisateur.

7.4 Authentification

Relecture, pas de confusion entre exchangeToken attribut utilisateur et Token Symfony

Afin de garantir la sécurité de notre transfert nous avons mis en place un système d'authentification sur les requêtes pour pallier à l'absence de limitation d'accès sur les routes utilisées. Cette authentification est réalisée à l'aide de notre classe `OfflineAuthenticator`.Nous avons écrit une condition ternaire, si l'utilisateur est authentifié alors l'action continue, sinon la requête échoue avec le status `401`.

Pour authentifier l'utilisateur, il faut utiliser des informations qui lui sont personnelles ; il est courant de transmettre le nom d'utilisateur et le mot de passe pour se connecter à une plateforme. Dans notre cas, nous avons fait le choix d'utiliser un token d'identification qui est unique à chaque utilisateur. Ce token est un champ que nous avons ajouté à l'entité `User` et qui est créé par la fonction suivante : `hash("sha256", $this->username.time().rand())`. Le hash est établi comme unique et est créé sur base du nom d'utilisateur, de la date à laquelle il est créé ainsi que d'un facteur aléatoire.

Nous avons choisi d'utiliser un token pour s'identifier lors de l'échange plutôt qu'une combinaison habituelle nom d'utilisateur et mot de passe car celle-ci présentait selon nous des défauts dans le contexte de *OfflineBundle*. Premièrement, la méthode d'authentification utilisée requiert que nous lui transmettions un mot de passe en clair ; c'est à dire un mot de passe qui ne soit pas encrypté. Or, au sein de la base de données, le mot de passe n'est pas enregistré comme tel et est bien entendu crypté. Deux options s'offraient alors à nous : soit nous demandions le mot de passe à l'utilisateur à chaque demande de synchronisation, ce que nous jugions inutile. Soit nous aurions du effectuer une comparaison entre les deux mots de passe cryptés.

Cette deuxième option pose également un problème, le mot de passe crypté résulte du choix de l'utilisateur et d'un champ unique appelé `salt` généré aléatoirement à la création

3. 4.5.5

de l'utilisateur. Ce qui signifie que pour un même mot de passe choisi par un utilisateur, le mot de passe enregistré de manière cryptée dans la base de donnée est différent. En utilisant le mot de passe crypté, nous nous confrontions à des problèmes de synchronisation de mots de passe entre la plateforme connectée et déconnectée. De plus, notre *OfflineBundle* étant focalisé sur l'apprentissage au travers de Claroline, se concentrer sur la synchronisation des ressources nous a semblé plus important que les paramètres utilisateurs tels que la mise à jour du mot de passe.

Enfin, utiliser un token c'est utiliser une donnée moins sensible que le mot de passe sachant que l'usage du token est limité aux synchronisations. Envoyer le mot de passe en clair ou en crypté n'aurait pas été souhaitable si la plateforme à contacter ne dispose pas d'un certificat lui permettant d'établir une connexion sécurisée `HTTPS`.

7.4.1 Implémentation

L'authentification dans Claroline utilise le système de Security Context de Symfony. Lors de la connexion d'un utilisateur un Token est ajouté au SecurityContext afin d'identifier l'utilisateur jusqu'à la fin de sa session. Ce Token permet alors au système de droits de fonctionner. Des classes Voter se chargent de définir si l'utilisateur identifié dans le sécurity context peut ou non exécuter une action.

Review Here

Comme nous utilisons un identifiant différent du mot de passe, l'exchangeToken, pour connecter l'utilisateur nous avons du créer un nouvel objet Token de Symfony pour "remplacer" le UsernamePasswordToken. UserExchangeToken est notre object étendant le Token de Symfony que nous créons sur base de l'attribut exchangeToken de l'utilisateur. Ce UserExchangeToken est créé et placé dans le SecurityToken par la classe OfflineAuthenticator qui est appelée lors de l'Authentification d'une requête de connexion.

Rappeler explicitement que les droits seront les mêmes via UserExchangeToken que si on s'etait authentifier avec l'autre ?

7.5 Gestion des erreurs

Toujours dans l'idée d'implémenter un système de transfert qui soit le plus stable possible, nous utilisons les status des requêtes `HTTP`[4] afin d'obtenir des informations sur la manière dont la requête s'est exécutée. Si l'exécution s'est déroulée comme prévu, le status 200 est retourné. Par contre, si l'authentification a échoué nous retournons le code 401. De cette façon, nous pouvons interpréter le code lors de la réponse retournée au client offline et décider de relancer le transfert ou d'avertir l'utilisateur. C'est la méthode `analyseStatusCode($status)` du `TransfertManager` qui est chargée de l'analyse des codes d'erreur et de déclencher si nécessaire une exception.

Dans l'éventualité où nous avons une erreur de procédure sur la plateforme en ligne, le code 424 est retourné. Par erreur de procédure, nous entendons une erreur de chargement

4. La liste complète des codes `HTTP` peut être consultée sur http://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

ou de création. Une des causes pouvant provoquer une erreur de chargement est la présence d'un fragment erroné reçu lors d'un transfert. Dans le cas d'un code 424 une erreur est déclenchée et un message d'avertissement est affiché à l'utilisateur. Un autre type d'erreur que peut rencontrer notre module de transfert est une erreur de timeout déclenchée par *Buzz*. Le timeout signifie que l'hôte n'a pas pu être joint dans le temps imparti. Le transfert du fragment est alors redémarré une fois. Si un second timeout est reçu, l'exception est lancée.

Relire ici cohérence avec le pragraphe précédent

catch

Afin de donner à l'utilisateur la meilleure information possible, nous interceptons au sein des Controllers générant la vue du transfert les exceptions qui pourraient être déclenchée par la méthode `analyseStatusCode()`. De cette manière il nous est possible d'afficher un message à l'utilisateur et dans la mesure du possible de lui conseiller une solution.

7.6 Alternatives aux requêtes `POST`

Pour l'ensemble des modules de ce travail, nous avons exploré plusieurs pistes avant de fixer notre choix sur une solution. C'est également le cas pour le module de transfert.

7.6.1 WebSocket

En consultant le mémoire de Cédric Vanderperren de 2010, une autre idée d'implémentation pour le transfert nous est venue. Cette idée consiste en la création d'un socket entre les deux plateformes. Le socket permet d'ouvrir une connexion persistante entre deux entités jusqu'à ce que l'une d'entre elles décide de mettre fin à l'échange. Cette solution semblait séduisante car elle évite d'ouvrir et de fermer une connexion à chaque envoi de fragments. Toutefois, en discutant avec Stéphane Klein, il est apparu que ce type de procédure en `PHP` est bloquant pour le serveur qui l'exécute. En d'autre termes, cela signifie que le serveur ne peut effectuer d'autres actions tant qu'un transfert est en cours. Et ceci est génant car cela pourrait bloquer des utilisateurs utilisant le serveur sur lequel la requête est faite.

Pour solutionner ce problème, il est possible de faire tourner un daemon. C'est-à-dire, d'exécuter le processus du transfert sur un thread indépendant. De cette manière le serveur peut traiter le transfert en parallèle des actions des autres utilisateurs. Pour implémenter un daemon, il faut utiliser l'API `PCNTL` de `PHP`. Cependant cette librairie n'existe pas sur Windows, ce qui est problématique pour notre projet car certains de nos utilisateurs travailleront sur cette plateforme. En définitive et compte tenu de ces difficultés, nous avons fait le choix d'implémenter nous-même les fonctionnalités dont nous avions besoin pour notre protocole telles que nous les avons détaillées dans ce chapitre.

7.6.2 Torrent

Une troisième idée que nous avons examinée est l'usage d'un protocole de torrent pour échanger les fichiers. Ce type de protocole est utilisé pour diffuser un même fichier à plusieurs utilisateurs, chacun d'entre eux étant à la fois un émetteur et un récepteur du contenu. L'objectif de cette technique est de répartir la charge reseau sur l'ensemble des participants

à l'échange. Dans le cas d'`OfflineBundle`,ce type de protocole semble peu intéressant car notre échange ne concerne que deux entités, un client et un serveur. Aucun avantage ne serait donc tiré de l'aspect réparti et collaboratif de ce protocole. De plus, ce type de protocole est bloqué par certains pare-feu sur certains réseaux, il y aurait donc des restrictions d'usage.

7.7 Amélioration du transfert

To Review

Le transfert écrit actuellement chaque petit paquet sur le disque dur. Les performances de ce processus pourraient être améliorées si l'on passait par une queue virtuelle et un enregistrement en base de données du dernier paquet transféré.

reflechir au restart, ça me semble moins évident ici

En effet, cette implémentation ne devrait pas poser de problème au niveau de l'ordinateur personnel de l'etudiant mais pourrait être une contrainte sur une infrastructure plus importante car l'écriture sur disque dur est une opération coûteuse en temps. Le problème réside ici dans la répétition fréquente de l'opération d'écriture. De plus le nombre d'opération d'écriture sera proportionnelle à la taille de l'archive de synchronisation échangées. rappelons toutefois qu'elle permet de redémarrer facilement au dernier paquet échangé en cas de coupure de courant.

Gestion différente en ligne - hors ligne. RAM disk ONLINE performance

L'alternative pour éviter l'écriture répétée sur le disque du consisterait à stocker les fragments reçus dans une liste virtuelle et de n'enregistrer sur disque dur que le fichier global une fois le dernier fragment échangé.

ok et enregistrement db ? ?

Chapitre 8

Chargement de l'archive

La troisième étape de notre processus de synchronisation est le chargement de l'archive. L'objectif étant ici de charger l'ensemble des modifications rassemblée lors de l'étape de création au sein d'une archive. Ceci comprend la création des nouvelles ressources, la mise à jour de certaines ressources et la gestion des conflits éventuels entraînés par la modification d'une même ressource au sein des deux plateformes. Il s'agit donc de charger les modifications apportées à la plateforme hors-ligne (réciproquement en ligne), sur la plateforme en ligne (réciproquement hors-ligne).

8.1 Mise en contexte

Titre : conventions ?

Afin de simplifer la lecture de ce chapitre, nous décrirons le chargement comme se faisant d'une plateforme A vers une plateforme B. C'est-à-dire que la plateforme B chargera le contenu du manifeste créé par A. Etant donné que ce processus est appliqué de manière similaire sur la plateforme en ligne et sur la plateforme hors. La plateforme A peut être aussi bien la plateforme en ligne que hors-ligne et B l'autre plateforme.

8.2 Gestion des espaces d'activités

Dans un premier temps, la plateforme B charge la section `description` du manifeste, ce qui nous permet entre autre de vérifier sur base de son `exchangeToken` que l'utilisateur effectuant le chargement est bel et bien celui décrit dans le manifeste. Après quoi, nous rassemblons tout les espaces d'activités du manifeste et chargeons au sein de la plateforme B les ressources contenues dans ceux-ci.

Nous tentons de trouver l'homologue de ces espaces d'activités au sein de la base de données de la plateforme B. Chaque espace d'activités étant rendu unique par son attribut `GUID`, c'est sur base de ce dernier que la recherche est effectuée. Les espaces d'activités n'ayant pas d'homologue dans la base de données vont être créés sur base des informations contenues dans le manifeste. Pour ce faire, nous appelons la méthode `createWorkspace()` de notre `Service OfflineWorkspace`.

Nous nous inspirons de la méthode de création de Claroline pour implémenter la fonction `createWorkspace()` de notre service ; a ceci prêt que la méthode du noyaux de Claroline se base sur un formulaire complété par un utilisateur. Afin de réutiliser au mieux la fonction du

`WorkspaceManager` de Claroline nous précomplétons un objet formulaire avec les informations que nous avons extraites du manifeste. De base, le GUID est généré automatiquement par la fonction de création de Claroline, néanmoins dans notre cas cela n'était pas idéal. En effet, il fallait garder la consistence entre la base de données en ligne et hors-ligne et ce GUID servait de clé unique afin de distinguer les différents espaces d'activités.

Pour résoudre ce problème nous avons ajouter parmi les informations du Manifest, le GUID de l'espace d'activités. Ce GUID est ensuite placé dans le formulaire que nous précomplètons. Enfin, nous avons modifié la fonction de création de Claroline afin que celle-ci vérifie si le formulaire qu'elle recoit contient ou non un champs GUID avec une valeur différentes de NULL, si c'est le cas alors le GUID n'est pas généré automatiquement et est à la place initialisé à la valeur présente dans le formulaire.

Il était essentiel de fixer le GUID à la bonne valeur à la création car c'est à ce moment que les rôles associés à l'espace d'activités sont créés. Une fois l'espace d'activités créé, des rôles sont automatiquement générés et le nom de ces rôles est établi sur base du GUID et passe en lecture seule près leur création.

8.2.1 Gestion du créateur

Lors de la création d'un espace d'activités et des ressources qu'il contenait, nous avons dû faire face au problème de la gestion du créateur. En effet, chaque entité `resource_node` possède un champ indiquant l'ID du créateur de la ressource et il fallait s'assurer de la cohérence de ce champ entre les bases de données de la plateforme utilisant Claroffline et les bases de données du serveur distant. Rappelons que pour chaque ressource et espaces d'activités une entrée `resource_node` est ajoutée en base de données.

foonote ?

Nous ne pouvions pas réellement nous baser sur l'ID en base de données puisqu'il n'est en aucun cas garanti que ce dernier sera le même dans les deux bases de données. Lorsque nous devons recréer, sur le serveur distant, une ressource crée par l'utilisateur sur sa plateforme personnelle nous pouvons nous baser sur son `username`. En effet, ce champ est supposé être unique donc nous pouvons facilement retrouver l'utilisateur dans la base de données du serveur distant et ainsi pouvoir mettre à jour le champ `creator` de la ressource.

faut se fixer sur le nom qu'on donne aux plateformes connectées/déconnectées)

Le problème s'avèrait plus complexe pour amener une ressource existante sur le serveur local vers la plateforme personnelle. En effet, le créateur de cette ressource n'était pas forcément présent dans la base de données de la plateforme personnelle et il n'était pas raisonnable de mettre l'utilisateur de la plateforme comme créateur de cette ressource. Pour résoudre ce problème, nous avons implémenté une fonction qui tente dans un premier temps de vérifier sur le créateur de la ressource qu'on désire créé est présent dans la base de données de la plateforme déconnectée. Si ce n'est pas le cas, une fonction se chargera de créer un utilisateur pour symboliser ce créateur en base de données. Nous entendons par là qu'un utilisateur avec le même nom et prénom que le créateur sera introduit en base de données et, pour des raisons de sécurité, son e-mail, jeton d'échange et mot de passe seront générés aléatoirement.

Dire en quoi ca resoud le probleme ?

Couper ce paragraphe

8.2.2 Gestion des rôles

Une autre chose importante dont nous devions nous assurer est le respect des rôles. En effet, un utilisateur ayant le statut de `collaborateur` sur un espace d'activités sur le serveur distant ne devrait pas avoir un statut différent sur la plateforme déconnectée. Les différents rôles possibles sont créés à la création d'un espace d'activités et leur nom va dépendre du champ `GUID` de cet espace d'activités.

pas fini

Nous avions également la possibilité de changer ce GUID après la création. Mais cela posait problème a cause des `Role`. Dans Claroline, un utilisateur peut disposer d'un rôle par rapport à un espace d'activité. Un rôle est un ensemble de permission ; par exemple un enseignant disposera du rôle de manager pour un espace d'activité défini, il aura donc l'autorisation de modifier l'ensemble des ressources de cet espace ainsi que leurs propriétés. A contrario, un étudiant aura lui le rôle de collaborateur, il aura donc la possibilité de consulter l'ensemble des ressources d'un espace d'activité mais pas de les modifier. C'est d'ailleurs de cette manière que l'on peut savoir à quel espace d'activité un étudiant est lié, il s'agit de l'ensemble des rôles qu'il dispose. Un table reprend les connexions entre `User` et `Role`.

Ca j'ai pas changé à voir ce qu'on en fait

8.3 Création d'une nouvelle ressource

8.4 Mise à jour d'une ressource

8.5 Affichage des opérations

8.6 Cas problématique : la date de modification

Au cours de cette phase de réalisation, nous nous servions abondament de la date de modification des ressources afin de savoir si celles-ci avaient été modifiées depuis notre dernière synchronisation par exemple. Malheureusement, nous avons rencontré un bug récurent sur ces dates. Ce bug provenait d'un 'Listener' chargé de mettre automatiquement la date de modification des ressources dès qu'un champs de ces dernières étaient modifiés.

Définition ? ou retour vers lexique ?

Le problème étant qu'à la création de nouvelles ressources, le champs next_id de la dernière ressource présente dans la base de données était automatiquement mis à jour afin de préserver un certain ordre entre les ressources d'un même espace.. Hors cette mise à jour automatique déclenchait le Listener cité précédement et ce dernier modifiait alors la date de modification de la ressource, ce qui entrainait souvent des incohérences ou des erreurs.

c'est un peu flou... en gros les ressources d'un même repertoire/workspace connaissent l'id de leur voisine

Exemple :
Prenons le cas de deux plateformes, une en ligne (le serveur distant) l'autre hors-ligne (l'ordinateur personnelle d'un utilisateur), avec un Espace d'activité et dans cet espace 2 ressources : R1 et R2.
Imaginons que, côté 'en ligne' un utilisateur modifie R1 et que, côté 'hors-ligne', l'utilisateur modifie les deux ressources et tente de se synchroniser.

Suivant notre démarche, un Manifest va être créer contenant la description des deux ressources R1 et R2 modifiées par l'utilisateur, en parsant ce Manifest du côté en-ligne, nous voyons que la ressource R1 est déjà présente, car R1 côté 'en ligne' et 'hors-ligne' aura le même hashname. On va donc regarder les dates de modifications de ces deux ressources R1, étant donné qu'elles sont différentes (puisque R1 a aussi été modifiée par un utilisateur sur la plateforme en ligne), un doublon va être créé.
Mais, la création de ce doublon va faire une nouvelle entrée dans la base de données et donc mettre à jour le champs 'next_id' de R2 et, via le Listener, mettre à jour sa date de modification.
Dés lors, lorsque nous arrivons sur la ressource R2 dans le manifest, nous allons voir si elle n'est pas déjà présente en base de données, comme c'est le cas nous comparons leurs deux dates de modifications, comme elles sont différentes nous créons un doublon de R2 alors qu'à la base R2 n'a pas été modifié en ligne et donc aurait du être remplacé par le R2 du Manifest.

Ici on parle encore de l'ancienne manière dont on gérait les doublons, l'exemple est peut-être obsolète du coup

Une autre situation, moins complexe, est la suivante : soit deux ressources R1 et R2, une plateforme en ligne (le serveur) et une plateforme déconnectée (l'ordinateur personnelle de l'utilisateur). Si R1 est une ressource présente à la fois en ligne et hors-ligne et R2 une ressource nouvellement créée par l'utilisateur sur sa machine personnelle. Prenons l'hypothèse que R1 n'a été modifié sur aucune des plateformes.
Lorsque l'utilisateur va lancer sa synchronisation, la ressource R1 ne va pas être prise en compte mais la ressource R2 oui, celle-ci étant nouvelle. R2 va donc être recréée sur la plateforme en ligne et va entrainé une nouvelle entrée dans la base de données et la mise à jour le champs 'next_id' de R1. La date de modification de R1 ayant été modifiée, cette ressource sera prise en compte et ramenée sur la machine personne de l'utilisateur alors qu'aucune 'véritable' modification n'y a été apportée.

Cet exemple là en revanche est toujours bon

Fort heureusement et grâce à *Stéphane Klein* de chez Claroline, nous avons pu résoudre ce problème. L'idée de base était de désactiver le 'Listener' responsable de ces mises à jour de dates automatiques. Néanmoins ce dernier étant également utilisé et important pour d'autre partie de la plateforme Claroline, il a été décidé à place de surcharger ce 'Listener' afin de lui faire adopter le comportement désiré pour notre cas, c'est-à-dire ignorer les modifications apportées à la position d'un noeud.

noeud = resource_node mais comment expliquer ça clairement ?

Lexique : ajouter LISTENER

T'en penses quoi ?

8.7 Cas problématique : parents disparus

Lors de notre analyse, nous avons vu un autre problème qui pourrait surgir.
Pour expliquer cela, une petite mise en contexte s'impose. Dans la base de données, chaque ressource dispose d'un champs lui indiquant sa ressource 'parent', c'est-à-dire la ressource contenant cette ressource. Hors un problème susceptible de se produire est le suivant : que se passera-t-il, lors de la synchronisation, si un utilisateur déconnecté crée une ressource au sein d'un répertoire et que ce même répertoire a été supprimé sur le site en ligne ?
Une possibilité était, lorsque nous constatons l'absence d'une ressource 'parent' dans la base de données de la plateforme en ligne, de simplement niée la ressource à créé en partant du

postulat que, si ce répertoire a été supprimé, les ressources contenues dans celui-ci devraient l'être aussi.
Néanmoins, nous avons opté pour une autre solution moins 'punitive', si lors de la création nous constatons que la ressorce 'parent' n'existe plus, nous indiquons alors l'Espace d'activité de cette ressource comme étant son parent, ainsi nous nous assurons que la ressource soit toujours créée.

+ schéma ?

Une fois que l'espace a été trouvé ou créé, nous parcourons dans le manifeste les éventuelles ressources de celui-ci. De manière similaire aux espaces d'activités, pour chacunes des ressources, nous effectuons une requêteafin de vérifier si cette ressource est déjà présente dans la base de données de `B`. Pour cela nous appuyons une fois encore notre recherche sur une clé unique, le `hashname`.

Vraiment une requete a chaque fois ? j'ai peur que Lobelle pique sa crise

Si aucune ressource n'a été trouvée avec cette clé, cela signifie qu'elle est nouvelle et doit être créée dans notre base de données. Dans le cas contraire nous devons mettre à jour cette ressource. Ces deux points, de même que la gestion particulière des `Forums` sont discutés plus en détail ci-dessous.

Explication du Hashname, quoi exactement, comment est-il choisit de quel type de donnée s'agit-il.... pourquoi un generate random et pas un hash des donnée

8.7.1 Creation d'une ressource

Dans le cas où une ressource doit être nouvellement créée, nous récuperons tout les champs nécessaires à sa création dans le manifeste.Une information importante est le type de cette ressource, en effet en nous basant sur ce dernier nous pouvons alors créer une instance de la ressource correspondante.Nous disposons alors de l'entité souhaitée qu'il nous reste à persister avec le `ResourceManager`. Voyons ci-dessous quels sont les détails d'implémentation spécifique aux différents types de ressources gérés par notre bundle ainsi que, pour terminer, les éléments à modifier pour l'ajout d'un nouveau type de ressource au sein du chargement.

Schéma récapitulatif avec les choix (arbre de décision) sera beaucoup plus parlant pour le lecteur

Ressource `Texte`

Pour les ressources `Texte`, il nous a également été nécessaire d'effectuer deux autres tâches. La première consiste en la création d'une entité `text_revision` symbolisant le texte riche à proprement parler dans la base de données. L'autre devant extraire le contenu de la section `CDATA` du manifeste afin de récupérer le texte désiré. Ceci est fait au moyen d'une méthode implémentée par nos soins.

j'ai le sentiment que cette phrase est bateau, pour le lecteur technique c'est une evidence mais il ne sait pas comment, pour le casu ca ne lui parle pas

Ressource `Fichier`

Comme expliqué dans la section de la création de l'archive, la particularité des ressources du type `Fichier`, est d'être lié à un fichier spécifique. C'est pourquoi, il faut extraire ce document du dossier `data` de l'archive et copier celui-ci dans le répertoire `./files/` de la plateforme `B`.

correction : "nous pouvons alors déterminer l'instance de ressource correspondante." — reste instance que j'ai envie de modifier

toujours juste, meme en utilisant le DOMDocument[**?**] ? => de quelle méthode s'agit-il ? un comportement particulier à détailler ?

8.7.2 Mise à jour d'une ressource

Combien, préparer le lecteur

Dans la cas d'une mise à jour, nous avons identifié plusieurs possibilitées :

Donc check

1. La date de modification de la ressource de B est antérieure à la date de syncronisation de l'utilisateur (doncsur la plateforme B) :
   Cela signifie par conséquent que la ressource a bel et bien été modifiée par l'utilisateur au sein de la plateforme A. Dans le cas contraire elle ne figurerait pas dans le manifeste. En revanche celle-ci n'a pas été modifiée sur la plateforme eB depuis sa dernière synchronisation. Dans ce cas,notre solution est d'écraser la plus ancienne ressource par la nouvelle, supposée être plus à jour. En d'autres mot, la ressource de la plateforme B sera remplacée par celle transmise par A

de B

2. La date de modification de la ressource de B est postérieure à la date de synchronisation et est différente de la date de modification de la ressource figurant dans le manifeste :
   En d'autres mots, la ressource a donc été modifiée à la fois sur la plateforme A et sur la plateforme BAfin de préserver ces modifications, nous créons une copie de la ressource que nous avons appelé un `doublon`.

   Pour créer ce `doublon`, nous suivons un cheminement similaire à la création d'une ressource à quelques différences près. En effet, pour préserver les modifications, nous allons modifier en base de données le hashname, la clé, de la ressource déjà présente sur la base de données en ligne et modifions le nom de la ressource hors-ligne en y ajoutant *@offline*.

Ressource `Texte`

Create doubon ou create ressource sur les ressource text c'est quasi la même chose, y a juste le fameux changement niveau name et hashname.

Ressource `Fichier`

Lorsque nous créons un *doublon* d'une ressource File, il est également important de gérer les fichiers auxquels ces ressources font référence. Pour la ressource mise à jour depuis la plateforme déconnectée, il nous suffit d'aller placer ce fichier dans le bon dossier au sein de la plateforme en ligne. Cependant, avant de faire cela il est important d'aller modifier le nom du fichier stocké dans base de données de la plateforme en ligne.
En effet, le nom du fichier est basé sur la clé de la ressource le symbolisant en base de données, hors lors de création d'un doublon, nous modifions cette dernière. Par conséquent, si nous n'allons pas également modifié le nom du fichier la base de données deviendra donc inconsistante.

+ schéma du style :

offline online

R1 (clé : A1E2) R1 (clé : A1E2) A1E2.txt A1E2.txt -update- -update- -sync- -doublon-
R1 (clé : B2E3) A1E2.txt ————————-> ERROR ! R1@offline (clé : A1E2) A1E2.txt
need to change the first A1E2.txt par B2E3.txt ——> OK !

Ressource Directory

on change juste le nom

Ressource Forum

on change juste le nom du forum

8.7.3 Contenu des Forums

Peut-être expliqué la différence entre RessourceNode Forum et Forum ?

Lorsque nous rencontrons une ressource Forum et après avoir gérée celle-ci, nous traitons de suite son contenu via une sous-méthode. Cette dernière parcours la partie du Manifest correspondant au dit contenu et pour chaque type de contenu (pour rappel un Forum contient des Catégories, des Sujets et des Messages) utilise une requête pour vérifier si ce contenu est déjà présent dans la base de données de la plateforme en ligne.

Si tel est le cas, nous vérifions alors la date de modification de ce contenu présente dans le Manifest à la date de modification du contenu présent dans la base de données de la plateforme en ligne. Si la date présente dans le Manifest lui est postérieure, nous mettons à jour le contenu avec les informations du Manifest. Sinon, nous n'appliquons aucun changement.

Si il n'y a aucun contenu correspondant dans la base de données, nous créons alors ce dernier depuis le début. En créant un objet du type correspondant (Category, Subject ou Message) et en remplissant les champs requis sur base du Manifest.
Dans le cas d'un Message, nous utilisons une autre sous-méthode nous permettant d'extraire le contenu de ce message compris dans la section *CDATA* du Manifest.

8.8 Ajout d'une nouvelle ressource

8.9 Classe `SyncInfo` - changer ce titre

8.10 Problèmes et cas particuliers

8.10.1 Création d'un espace d'activités

8.10.2 Date de Modification

Au cours de cette phase de réalisation, nous nous servions abondament de la date de modification des ressources afin de savoir si celles-ci avaient été modifiées depuis notre dernière synchronisation par exemple. Malheureusement, nous avons rencontré un bug récurent sur ces dates. Ce bug provenait d'un 'Listener' chargé de mettre automatiquement la date de modification des ressources dès qu'un champs de ces dernières étaient modifiés.

Le problème étant qu'à la création de nouvelles ressources, le champs next_id de la dernière ressources présente dans la base de données était mis à jour afin de préserver un certain ordre. Hors en faisant cela, le Listener cité précédement était déclenché et par conséquent la date de modification modifiée, ce qui entrainait souvent des incohérences ou des erreurs.

Exemple :
Prenons le cas de deux plateformes, une en ligne l'autre hors-ligne, avec un Espace d'activité et dans cet espace 2 ressources : R1 et R2.
Imaginons que, côté 'en ligne' un utilisateur modifie R1 et que, côté 'hors-ligne', l'utilisateur modifie les deux ressources et tente de se synchroniser.
Suivant notre démarche, un Manifest va être créer contenant la description de ces deux ressources, en parsant ce Manifest du côté online, nous voyons que la ressource R1 est déjà présente, car R1 côté 'en ligne' et 'hors-ligne' a le même hashname. On va donc regarder les dates de modifications de ces deux ressources R1, étant donné qu'elles sont différentes (puisque R1 a aussi été modifiée par un utilisateur sur la plateforme en ligne), un doublon va être créé.
Mais, la création de ce doublon va faire une nouvelle entrée dans la base de données et donc mettre à jour le champs 'next_id' de R2 et, via le Listener, mettre à jour sa date de modification.
Dés lors, lorsque nous arrivons sur la ressource R2 dans le manifest, nous allons voir si elle n'est pas déjà présente en base de données, comme c'est le cas nous comparons leurs deux dates de modifications, comme elles sont différentes nous créons un doublon de R2 alors qu'à la base R2 n'a pas été modifié en ligne et donc aurait du être remplacé par le R2 du Manifest.

Fort heureusement et grâce à /textitStéphane Klein de chez Claroline, ce problème a pu être résolu. L'idée était de désactiver le 'Listener' responsable de ces mises à jour de dates automatiques. Néanmoins ce dernier étant également utilisé et important pour d'autre partie de la plateforme Claroline, il a été décidé à place de surcharger ce 'Listener' afin de lui faire adopter le comportement désiré pour notre cas, c'est-à-dire ignorer les modifications apportées à la position d'un noeud.

noeud = resource_node mais comment expliquer ça clairement ?

Lexique : ajouter LISTENER

T'en penses quoi ?

8.10.3 Parents disparus

Lors de notre analyse, nous avons vu un autre problème qui pourrait surgir.
Pour expliquer cela, une petite mise en contexte s'impose. Dans la base de données, chaque ressource dispose d'un champs lui indiquant sa ressource 'parent'. Hors un problème susceptible de se produire est le suivant : que se passe-t-il lors de la synchronisation si un utilisateur déconnecté crée une ressource au sein d'un répertoire et que ce même répertoire a été supprimé sur la plateforme en ligne ?
Une possibilité était, lorsque nous constatons l'absence d'une ressource 'parent' dans la base de données de la plateforme en ligne, de simplement niée la ressource à créé en partant du postulat que, si ce répertoire a été supprimé, les ressources contenues dans celui-ci devraient l'être aussi.
Néanmoins, nous avons opté pour une autre solution moins 'punitive', si lors de la création

nous constatons que la ressorce 'parent' n'existe plus, nous indiquons alors l'Espace d'activité de cette ressource comme étant son parent, ainsi nous nous assurons que la ressource soit toujours créée.

+ schéma

8.11 Ajout d'une nouvelle ressource

8.11.1 Gestion du créateur de ressources

8.11.2 Gestion des rôles

Chapitre 9

Installation

L'objectif de cette partie du travail est de permettre à un utilisateur de récupérer sur son ordinateur personnel le logiciel qui lui permettra d'utiliser Claroline de manière déconnectée. Pour réaliser celui-ci nous avons du faire un choix parmis les différentes options possibles et en tenant comptes des contraintes qui nous étaient imposées. Ce chapitre à propos de l'installation servira à expliquer la manière dont nous avons choisi de diffuser la solution que nous avons développée. Nous traiterons dans ce chapitre de la manière de créer un installateur dans le but de le mettre à disposition des utilisateurs et non de l'intégration de notre plug-in à une plateforme Claroline qui sera elle discutée dans le manuel B page VII.

explication choix serveur apache, distribué via zip

9.1 Contraintes

Plusieurs contraintes étaient à prendre en compte lors de la réalisation de cette phase du projet. La première d'entre elles est la simplicité pour l'utilisateur. Une des orientations demandée était le fait qu'il soit le plus simple possible pour un utilisateur d'installer Claroline muni du `OfflineBundle` sur sa machine. Une autre contrainte est que notre installateur puisse fonctionner sur les systèmes d'exploitations Windows et Linux.

Comme notre projet est implémenté comme un bundle venant se joindre à la plateforme Claroline, nous avions besoin de pouvoir exécuter le code PHP de Claroline sur l'ordinateur personnel de l'utilisateur. Par ailleurs Claroline a besoin d'une base de donnée pour fonctionner. De plus, l'utilisateur doit avoir la possiblité de démarrer sans disposer de connexion Internet, nous avons donc choisi de doter chaque utilisateur d'un serveur Apache. C'est sur celui nous intégrerons les fichiers et la base de données de Claroline.

9.2 Choix de la solution

La question suivante qui nous a occupé est de savoir de quelle manière serait réalisée cette intégration des fichiers de Claroline sur le serveur apache qui équiperait notre installateur. Deux options se sont présentées à nous, générer dynamiquement une plateforme pour chaque utilisateur souhaitant se synchroniser et personnaliser celle-ci immédiatement en y intégrant ses cours personnels ou créer une version générique de l'installateur qui ne disposerait d'aucun contenu.

Nous choisi la deuxième option, créer une version générique de la plateforme Claroline sans aucun contenu. Nous dirons que cette version de Claroline est vierge, car aucun espace

d'activités, ressources ou utilisateurs n'y sont enregistré. Ce type de diffusion comporte un avantage majeur, il est économe en ressource à plusieurs niveaux. D'une part, il évite de générer une plateforme spécifique pour chaque utilisateur. Et d'autre part il permet de copier cette plateforme installateur d'ordinateur en ordinateur sans avoir à le télécharger chaque fois.

Le fait de pouvoir copier l'installateur localement d'un ordinateur à un autre permet d'éviter de saturer la connexion. Par exemple, dans une salle informatique avec une faible connectivité, si un grand nombre d'utilisateurs souhaitent obtenir la plateforme en même temps, il sera plus efficace de copier le fichier d'un poste de travail à un autre que de le télécharger depuis chaque poste de travail et le serveur distant. Si l'on pousse cette réflexion plus loin, nous pourrions même envisager qu'un chargé de cours ou une administration prépare un ensemble de CD ou de clé USB qui contiendraient une copie du logiciel ; il n'y aurait ensuite plus qu'à distribuer ces copies matérielles.

Nous avons choisi de distribuer notre application sous la forme d'un fichier zip. Ainsi, l'utilisateur n'a qu'à le récupérer, par téléchargement ou copie locale, le décompresser et l'utiliser directement. De plus, il est aisé pour un administrateur système de créer sa propre archive d'installation personnalisé car cette archive serait un assemblage des différentes parties requises.

9.3 Choix des outils

9.3.1 Xampp

Voici la définition de Xampp proposée par ses dévelloppeurs, Apache Friends : "*XAMPP est une distribution Apache entièrement gratuite, facile à installer qui contient MySQL, PHP et Perl. Le package open source XAMPP a été mis en place pour être incroyablement facile à installer et à utiliser.*" Bien que ce ne soit peut-être pas la plus performante, nous avons choisi de proposer cette distribution du serveur Apache pour notre installateur car cette distribution est proposée pour les plateformes Linux, Windows et Mac. Elle est donc disponible pour l'ensemble des utilisateurs potentiels de Claroline.

9.3.2 Google Chrome Portable

En plus des fichiers de Claroline au sein du serveur Apache, nous proposons d'intégrer le navigateur Google Chrome. Notre choix s'explique car de cette manière nous pouvons être sur que l'utilisateur disposera d'un navigateur à jour et performant.

De plus, Google Chrome dispose d'un mode permettant de simuler un site web comme étant une application native. Lorsque ce mode est activé, le navigateur s'affiche en plein écran, faisant disparaitre la bare des taches et la bare de navigation. Ce mode nous semble particulièrement intéressant car combiné avec un racourcis sur le bureau, l'utilisateur sera imergé dans l'expérience Claroline et ne sera pas perturbé par l'URL de la plateforme décon-

nectée[1]. En utilisant le mode offline de Claroline de cette manière, l'utilisateur sera à même de distinguer de manière plus claire son accès à la plateforme connectée en naviguant sur son chemin habituel et la plateforme déconnectée qu'il accèdera par le racourcis sur son bureau.

9.4 Construire l'archive de synchronisation

Avec le code source de ce travail nous fournissons un installateur générique distribuable auprès d'utilisateurs de la version 3.1 de Claroline. Nous sommes conscient des limites déterminées par notre installateur, il n'a été créé que pour une version de Claroline et dans un contexte établi. C'est pourquoi vous trouverez dans cette section l'ensemble des informations nécessaires pour créer vous même l'installateur pour les utilisateurs déconnectés. Il s'agit ci-dessous des instructions correspondant aux actions que nous avons effectuées pour créer notre propre installateur.

réfléchir à comment faire pour intégrer les spécificités d'une plateforme dédiée comme iCampus, a t on déjà expliqué le contenu général de notre plateforme ?

9.4.1 Télécharger les outils

Pour créer le zip d'installation, vous devez commencer par réunir les différents composants qui y seront intégrés : un serveur web, une plateforme claroline et optionellement un navigateur.

Voici les liens vers les outils que nous avons choisi :

- Chrome Portable : http://portableapps.com/apps/internet/google_chrome_portable
- XAMPP : https://www.apachefriends.org/fr/index.html
- Claroline : http://www.claroline.net/

9.4.2 Assembler les outils

subsubsection pour les différentes étapes ? itemlist ?

Maintenant que nous avons rassemblé les outils dont nous avions besoin, l'objectif est de les mettre ensemble pour obtenir un dossier qui sera prêt à être compressé pouvant ensuite être distribué. Pour commencer créons un dossier qui contiendra l'ensemble de fichiers qui seront compressé dans un second temps, appelons le "claro_install".

Installer Google Chrome Portable

Une fois votre dossier créé, installons Google Chrome Portable à moins que vous n'ayez choisi de vous en passer, auquel cas allez directement à l'étape suivante. Pour installer Google Chrome, il nous suffit d'exécuter le fichier que nous venons de télécharger. Le wizard d'installation nous demande alors dans quel répertoire nous souhaitons l'installer. Nous choisirons "\claro_install\chrome". Attendons la fin de l'exécution de l'installation, nous pouvons alors vérifier que celle-ci c'est correctement déroulée en lançant Google Chrome. Ce dernier doit être stockés à l'emplacement référencé et doit être capable d'afficher le rendu d'une page

confirm dir

1. localhost/Claroline/web/app.php

web de votre choix. Si l'installation a rencontré des problèmes nous vous invitons à vous rendre dans la documentation de Google Chrome [2].

Installer Xampp

Après avoir installé votre navigateur si vous avez choisi d'en intégrer un, il faut installer le serveur web, dans notre cas Xampp. Attention, dans le choix du server Web, celui-doit supporter PHP 5.4.1 ou supérieur car il s'agit là d'un des prérequis de Claroline. La liste exhaustive des prérequis est sur le Github du projet Claroline. [3]

Pour l'installation, nous suivons les instructions recommandées. Ensuite, plusieurs aménagement dans le fichier PHP.ini doivent être effectués :

1. Le champ *memory_limit* doivent être supérieure ou égale à 256Mb

```
; Maximum amount of memory a script may consume (128MB)
; http://php.net/memory-limit
memory_limit=256M
```

2. Le champ *max_execution_time* doit être mis à 60 sec

```
;;;;;;;;;;;;;;;;;;;
; Resource Limits ;
;;;;;;;;;;;;;;;;;;;

; Maximum execution time of each script, in seconds
; http://php.net/max-execution-time
; Note: This directive is hardcoded to 0 for the CLI SAPI
max_execution_time=60
```

3. Enfin pour les utilisateurs Windows pour des questions de performances nous vous conseillons d'activer l'extension *zend_opcache* et désactivier *XDEBUG*

```
[eAccelerator]
zend_extension = "C:\xampp\php\ext\php_eaccelerator_ts.dll"

;[XDebug]
;zend_extension = "C:\xampp\php\ext\php_xdebug.dll"
;xdebug.profiler_append = 0
;xdebug.profiler_enable = 1
;xdebug.profiler_enable_trigger = 0
;xdebug.profiler_output_dir = "C:\xampp\tmp"
```

2. https://support.google.com/chrome/
3. https://github.com/claroline/Claroline

```
;xdebug.profiler_output_name = "cachegrind.out.%t-%s"
;xdebug.remote_enable = 0
;xdebug.remote_handler = "dbgp"
;xdebug.remote_host = "127.0.0.1"
;xdebug.trace_output_dir = "C:\xampp\tmp"
```

Installer Claroline

Maintenant que nous avons installé un serveur web nous devons y ajouter claroline. Nous détaillerons ici comment intégrer la plateforme en partant de l'installation de Claroline de base. Si votre plateforme comporte des spécificités qui lui sont propres telle qu'un template ou des extensions particulières veuillez sauter au point suivant.

Pour installer Claroline à partir du WebInstaller fournis par le site http://www.claroline.net/ il faut extraire le contenu de l'archive contenant les fichiers de Claroline dans le dossier web du serveur ; dans le cas de Xampp ce dossier est `xampp/htdocs`. Ce répertoire est l'équivalent de `www` pour le serveur Wamp. Il s'agit du répertoire qui sera concidéré par le serveur web comme le répertoire contenant les fichiers à interpréter.

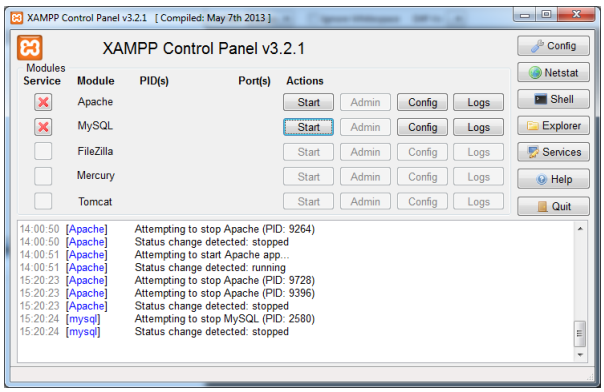

FIGURE 9.1 – Fenêtre Xampp control

À ce niveau nous disposons des fichiers de Claroline disponible sur un serveur web et accessible via un navigateur. Toutefois la plateforme claroline n'est pas encore configurée. Nous allons donc demarrer le serveur Apache et Mysql et nous connecter au site. Pour démarrer le serveur Apache, rendez-vous dans le dossier d'installation de Xampp, exécutez `xampp-control.exe` (voir Figure 9.1), et cliquez ensuite sur le bouton start faisant face à Apache et à MySql (si vous rencontrez des erreurs à cette étape, veuillez vous référer à la documentation disponible en ligne). Si les 2 services ce sont correctement lancé le bouton start se grise et le bouton stop devient actif ; ceci est illustré sur la Figure 9.2.

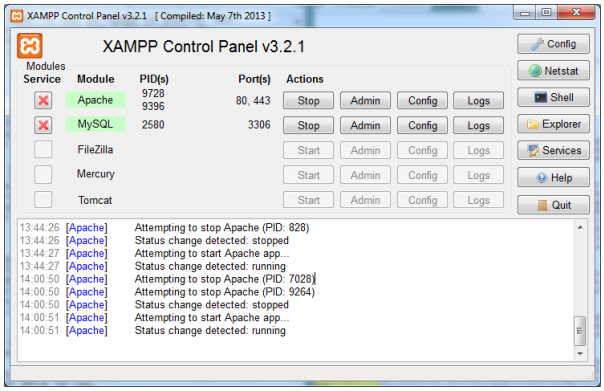

FIGURE 9.2 – Serveur Apache et MySql démarrés sous Xampp

Prénom	root
Nom	root
Nom d'utilisateur	root
Mot de passe	password
Email	root@claro.net

TABLE 9.1 – Informations pour la création de l'utilisateur administrateur

Vous pouvez alors vous connecter sur la page localhost/Claroline/web/install.php pour executer l'installation de claroline afin que la plateforme soit configurée.

ATTENTION, il doit s'agir d'une plateforme avec offline intégré, sinon ajout des étapes d'intégration

Vous arrivez alors sur une vue à 5 panneaux vous interrogeant sur la configuration. La première étape vérifié que les pré-requis soient respectés pour l'installation, ce devrait être le cas. La seconde étape vise à créer le profil administrateur de la plateforme, nous vous proposons d'entrer les informations données dans la Table 9.1 sachant que ce profil ne devrait pas être utilisé ultérieurement et ne devrait pas nuire au bon fonctionnement de l'extension offline.

La dernière étape est celle de la configuration du serveur mail et ne nous concerne pas je vous invite donc à cliquer sur skip. Si cette dernière étape s'est déroulée sans erreur nous disposons maintenant de tout que nous avons besoin pour distribuer claroline équipé du bundle offline. Enfin l'installateur de claroline s'exécute. Après quoi, nous vous invitons à passer à l'étape de finalisation 9.4.3.

configurateur ?

Ajout offline bundle

Installer Claroline personnalisé

Dans le cas où vous disposez d'une plateforme personnalisée la solution consiste à cloner manuellement votre plateforme dans le répertoire web de Xampp. Par cloner nous entendons copier les fichiers et créer une base de données contenant les informations minimales nécessaire. Avant de suivre les étapes ci-dessous, assurez-vous d'avoir préalablement installé le plug-in OfflineBundle sur votre plateforme Claroline.

Vous trouverez les instruction d'installation du plug-in OfflineBundle au point...

La première étape consiste à copier tout les fichiers de votre plateforme, à l'exception du répertoire contenant les ressources dans le répertoire web de Xampp. Ensuite vous devez modifier le fichier `Claroline/app/config/parameters.yml` pour y introduire les informations de connexion à la base de données. Si vous n'avez pas changés ces paramètres depuis l'installation de Xamp, ils sont `root` et pas de password ( ). Lors de la troisième et dernière étape, il faut créer la base de données minimale nécessaire au fonctionnement de Claroline. Pour ce faire, démarrez le serveur Apache et MySql, en exécutant le fichier `xampp-control.exe`. La fenêtre illustrée à la Figure 9.1 s'ouvre, cliquez ensuite sur le bouton `Start` pour le serveur Apache et pour MySql, le paneau de control prend à alors l'aparence représentée en Figure 9.2.

RELIRE !

Enfin, ouvrez un terminal et déplacez-vous dans le dossier `claro_install\Claroline`. Entrez la commande suivante :php app/console claroline :install Vous disposez maintenant d'un clone de votre Claroline personnalisé, veuillez terminer la création de l'installateur en suivant les instructions du dernier point.

Ajouter code style

9.4.3 Préparer la plateforme pour l'utilisateur déconnecté

paragraphe que je vient d'écrire ci-dessus

Pour faciliter la vie de l'utilisateur déconnecté nous avons écrit un petit script qui automatise le lancement des différents éléments nécessaires au bon fonctionnement. . Il démarre Google Chrome sur la page correspondante à Claroline après avoir démarré le serveur web. Nous vous invitons donc à ajouter ce script dans le dossier claro_install.Vous pouvez maintenant compacter le dossier claro_install et distribuer cette archive.

code du script

attention autostart apache et mysql sur xampp-controller

```
CODE de notre script

Pour Windows

Pour Linux
```

9.4.4 Les erreurs connues/possibles

Attention probleme de classpath possible

9.4.5 Inno Setup

open source

Lors de notre travail de recherche et choix de la solution que nous avons fait pour l'installateur, nous avons trouvés le logiciel Inno Setup. Ce logiciel permet de créer des l'installateur pour Windows. Nous avons trouvé que ce logiciel pourrait s'avérer intéressant pour créer un fichier exécutable pour les utilisateurs de ce système d'exploitation. En effet, pour créer un fichier exécutable avec Inno Setup un administrateur doit uniquement complèter quelques paramètres et référencer les sources. De l'autre côté, un utilisateur Windows utilisant ce fichier exécutable profite d'une fenêtre d'installation telle qu'il a l'habitude d'en voir. En

terme foireux

terme d'immersion utilisateur l'effet est donc plus fort pour un résultat similaire à celui de notre archive d'installation. Toutefois comme cette solution est spécifique aux utilisateurs

annoncer la couleur au début ?

Windows nous ne la présentons que dans un second temps.

———— Ajout de offlinebundle sur une plateforme

modifier composer.json ajout du bundle ajout du repo exe "composer update" exe "php app/console claroline :update" // php app/console claroline :plugin :install ClarolineOffline-Bundle

Chapitre 10

Récupération d'un compte

10.1 Première connexion

En plus des routes que nous utilisons pour le processus de synchronisation, il est tout à fait possible d'en mettre d'autres en place. C'est d'ailleurs ce que nous avons fait pour la première connexion. En effet, avant de pouvoir échanger des archives entre l'ordinateur personnel et le serveur distant nous devons récupérer l'`exchangeToken` d'un utilisateur. Dans ce chapitre nous allons vous détailler la manière dont nous pouvons ajouter un utilisateur à la plateforme fonctionnant en mode hors-ligne.

Afin de s'assurer que l'utilisateur récupère son compte, nous avons créé un `Listener`[1]. Ce dernier est à l'écoute de tout évenement de connexion et vérifie si l'utilisateur a déjà récupéré son compte. Si tel est le cas, le `Listener` n'effectue aucune tâche particulière et laisse l'utilisateur faire ce qu'il désirait. Dans le cas contraire, le `Listener` va rediriger l'utilisateur vers la vue lui demandant de récupérer son compte (illustré sur la Figure 10.1).

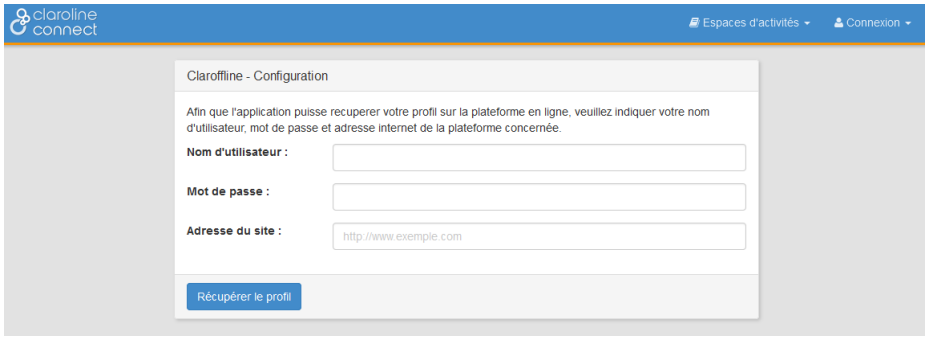

Figure 10.1 – Vue de configuration affichée à l'aide du `Listener` lors du premier démarrage

Dans le `Listener`, afin d'établir s'il s'agit ou non de la première connexion de l'utilisateur, nous utilisons le fichier "`app/config/sync_config.yml`" enregistré dans les fichiers de Claroline. Si ce fichier n'est pas présent alors nous considérons que l'utilisateur n'a pas encore récupéré les informations de son compte sur le serveur distant. En effet, lorsque nous récupérons un profil, nous créerons le dit fichier. Toutefois, il ne faut pas que ce `Listener` agisse sur le serveur distant, c'est pourquoi nous avons ajouté une seconde condition. Cette

Flag si implémentation environnement développement

1. Voir Lexique 2.2 page 14

seconde condition consiste à tester l'URL sur laquelle nous sommes connecté. S'il s'agit de `localhost`, c'est-à-dire l'adresse du serveur `Apache` exécuté localement sur l'ordinateur personnel de l'utilisateur déconné, alors nous savons que nous sommes sur la plateforme destinée à travaillé sans connexion.

Au final, si l'utilisateur n'a pas encore initialisé son compte (créé le fichier `app/config/sync_config.yml` et tente d'utiliser l'application Claroline déconnectée (via `localhost`) alors nous déclenchons une redirection vers la fenêtre de configuration illustrée à la Figure 10.1. Sur cette fenêtre l'utilisateur est invité à entrer son nom d'utilisateur, son mot de passe et l'adresse de la plateforme à contacter. Lorsque l'utilisateur a entré ces informations il doit cliquer sur le bouton RÉCUPÉRER LE PROFIL. Nous exécutons alors l'action se chargeant contacter et récupérer l'identité de l'utilisateur sur le serveur distant que l'utilisateur a précédemment indiqué.

L'action se déroule comme suit :

1. L'ordinateur personnel effectue une requête sur une route précise du serveur donné par l'utilisateur. Dans cette requête, nous insérons les identifiants donnés dans le formulaire.
2. Si le serveur arrive à authentifier l'utilisateur sur base de ses identifiants il répond à la requête en retournant l'ensemble des informations nécessaire pour la création d'un profil utilisateur. Dans le cas où il n'a pas pu confirmer l'identité il retourne une un code d'erreur `HTTP 424`.
3. Enfin nous créons un profil utilisateur sur la plateforme Claroline de l'ordinateur personnel et créons notre fichier `app/config/sync_config.yml`.

De manière plus détaillée, nous devons récupérer les informations de l'utilisateur pour pouvoir créer celui-ci dans la base de données locale. Lors de l'étape une, pour télécharger ces informations incluant l'`exchangeToken`, car rappelons-le la plateforme installable ne dispose d'aucunes informations. Nous contactons la route `claro_sync_user` qui requiert le nom d'utilisateur et le mot de passe pour s'identifier. Cette route est la seule exception qui n'utilise pas le token d'échange car c'est précisément celle-ci qui nous permet de le récupérer sur le serveur distant. Une fois le compte récupéré, nous enregistrons quelques informations sur l'utilisateur dans un fichier `YAML`, que nous appelons `sync_config`. Ce fichier à la forme d'un tableau de tableau où chaque sous-tableau représente un utilisateur et contient son nom d'utilisateur, son adresse mail et l'url qu'il a entré pour charger son compte.

Comme nous l'avons expliqué dans la section sur l'authentification [2], la méthode d'authenfication requiert le password non crypté ; c'est donc de cette manière que nous le transférons dans le contenu de la requête. Pour cette raison, il est important que la route de contact de la plateforme en ligne soit sécurisée au moyen du protocol `HTTPS`. Nous n'avons malheureusement pas pu tester l'utilisation de *Buzz* avec une adresse `HTTPS` car nous ne disposons pas de certificat. Après vérification dans la documentation et discussion avec Stéphane Klein, aucuns changements ne devraient être fait dans notre code pour utiliser le protocol `HTTPS`.

2. Section Authentification, voir Chapitre 7 page 49

Seule l'adresse à contacter, donnée par l'utilisateur lors de la configuration doit comporter le préfixe.

Lors de la seconde étape, la méthode `getUserAsTab()` que nous avons créé dans l'entité `User` du `CoreBundle` est appelée si l'authentification a réussi. Cette méthode se charge de collecter les informations dont nous avons besoin pour recréer l'utilisateur sur l'ordinateur personnel. Ces informations sont alors envoyées comme réponse à la requête formulée lors de la première étape.

Enfin la dernière étape se charge de créer l'utilisateur dans la base de donnée de la plateforme de l'ordinateur personnel. Cet utilisateur est recréé avec un `exchangeToken` identique à celui qu'il a sur le serveur distant. Pour que la base de données locale de l'ordinateur personnelle soit consistante avec la base de données du serveur distant nous devons également mettre à jour le `GUID` de l'espace d'activités personnel de l'utilisateur pour qu'il coïncide avec celui en ligne.

10.1.1 Message d'erreurs

Le processus de récupération de compte peut déclencher des erreurs. Une partie d'entre elle étant similaire à celles rencontrées lors de la synchronisation, nous ne les reprendrons pas ici. En plus de ces dernières, deux erreurs sont spécifiques à la tentative de récupération de compte : celle qui intervient lorsque l'on tente de récupérer un compte qui l'a déjà été sur ce même ordinateur personnel et la seconde qui se présente lorsque l'utilisateur tente de récupérer un compte qui n'existe pas sur la plateforme qu'il a référencée.

10.2 Ajout d'un autre utilisateur

La gestion de plusieurs utilisateurs sur un même ordinateur personnel utilisant Claroffline est intégrée à notre bundle. En plus du premier utilisateur qui s'est synchronisé sur la plateforme installée sur l'ordinateur personnel, un autre utilisateur peut récupérer son compte s'il le désire. Pour cela, il lui suffit de reprendre le formulaire que nous imposions au premier utilisateur. Si le nouvel utilisateur entre des informations correctes, son compte pourra être récuperé de la même façon que celui du premier utilisateur le fut. Lorsque nous récupérons un compte, après création du nouvel utilisateur dans la base de données, nous ajoutons un nouveau tableau contenant les informations de ce nouvel utilisateur dans le fichier `sync_config.yml`.

Toutefois, il est important d'avoir en tête que les informations de ces deux utilisateurs seront présentes dans la même base de données et donc que le propriétaire de l'ordinateur pourrait avoir accès à l'ensemble de ces informations s'il consulte la base de données de l'application.

10.3 Modification de l'adresse du site

En plus de la vue permettant à l'utilisateur d'ajouter un autre compte à sa plateforme, nous avons également créé une vue pour lui permettre d'éditer l'adresse du site internet enregistrée dans le fichier `sync_config.yml`. L'idée était d'autoriser l'utilisateur à modifier l'url si nécessaire, par exemple si le site avec lequel il se synchronisait a changé de nom de domaine. Pour cela, il suffit à l'utilisateur de se rendre sur la vue du plugin Claroffline, comme s'il désirait se synchroniser et de cliquer sur le bouton PARAMÈTRES.

Chapitre 11

Tests

Tests automatiques dans Claroline,

11.1 Tests Unitaires

tests unitaires PHP Unit
Expliquer PHP Unit ?
Pq PHP unit ?
ne pas oublier de remplacer localhost/Claroline/web/app_dev.php >< http :/path/to/-web/app_test.php

11.2 Tests d'intégration

Utilisation de Behat + Mink (encore à faire) similaire à Cucumber pour Ruby.
Expliquer/Présenter Behat + Expliquer/Présenter Mink.
Pq Behat ? Même justification que pour Genlog (compréhensible, sous forme de scénario, ...) + minimum d'expérience avec Cucumber et tests concluants lors de l'utilisation de cet outil pour Genlog.
+ c'est également ce qu'utilise Claroline actuellement.

11.3 Tests de transfert

Pour faire les tests de requete POST HTML grace a Simple REST Client chrome extension

11.4 Tests d'installation

11.4.1 Windows 7

11.4.2 Windows XP

PHP 5.5 n'est pas supporté par windows XP. C'est pourquoi une autre distribution du Xampp est nécessaire ; actuellement la version la plus à jour est XAMPP 1.8.2

11.4.3 Linux Ubuntu

Chapitre 12

Discussions

Ce chapitre traitera de réflexions que nous avons menées sur certaines problématiques que nous avons soulevées en réalisant ce travail ainsi que des limites de ce dernier. Enfin nous donnerons des pistes pour d'éventuelles contributions futures.

12.1 La fonction de suppression

Une fonction qui pourrait poser problème dans l'usage de notre bundle, du fait qu'elle n'est pas implémentée, est la fonction de suppresion. En effet, dans le menu de gestion d'une ressource, il est possible pour l'utilisateur disposant des droits appropriés de supprimer une ressource d'un espace d'activités. Or, si une ressource est supprimée dans un espace d'activités sur le serveur distant ou sur l'ordinateur utilisant Claroffline cette modification ne sera pas communiquée par notre bundle. Une ressource supprimée sur le serveur distant ne le sera donc pas sur l'ordinateur utilisant Claroffline. Par exemple, si un professeur en vient à supprimer une ressource, l'étudiant continuera à en disposer même après synchronisation.

A l'heure actuelle la suppression d'une ressource est directe. Quand une ressource est supprimée les entrées dans la base de données et les éventuels fichiers (pour les resources Fichier) sont effacées immédiatement. Lors d'une discussion avec Stéphane Klein, ce dernier nous a annoncé que la gestion d'une suppression plus douce des ressources était en discussion mais ne faisait pas partie des priorités.

Cette suppression plus douce ("soft delete") consisterait à rendre invisible pour l'utilisateur une ressource dîtes supprimées et n'effacer effectivement les entrées en base de données et les fichiers que dans un second temps. Intégrer de telles modifications nécessite une manipulation majeure de la fonction de suppression de Claroline. Effectuer ces modifications nous même nous aurait cependant dissocié du développement principal de Claroline. Or notre but premier était de faire un plug-in directement intégrable à la version principale et non pas sur une version qui nous était propre.

Néanmoins, nous avons réfléchi à une manière pour notre OfflineBundle d'implémenter cette suppression douce. Un champ textttisDelete pourrait être ajouté à l'entité `resource_node` pour déterminer si celle-ci est dans un état visible ou non pour l'utilisateur. Lorsque celui-ci effectuerait une action de suppression sur une ressource, le champ `isDelete` sera enregistré à vrai. Du fait de la modification d'un attribut de la variable, sa date de modification serait mise à jour, ce qui aurait pour conséquence d'intégrer cette ressource dans la synchronisation

suivante. Et par l'ajout du champ dans les valeurs transmises dans le manifeste, la ressource serait également rendue invisible sur l'autre plateforme.

Une variante à cette solution consiste à ne pas synchroniser le champ textttisDelete mais à le comparer. Dans le cas où les deux champs sont vrai alors la ressource est supprimée. Etant donné que cette question n'est pas fixée chez Claroline, elle est resté en suspend chez nous également. Toutefois les mises à jour à effectuer dans notre code sont mineures. Pour que cela soit pris en charge dans notre code, la seule adaptation à faire serait d'ajouter le champ `isDelete` dans `OfflineResource`.

12.2 Inscription à un nouvel espace d'activités

Une question de réflexion qui nous a occupé pour le travail du OfflineBundle est celle de l'inscription d'un étudiant à un nouvel espace d'activités. Cette possibilité est en effet essentielle car il ne faudrait pas qu'un étudiant ait à télécharger une nouvelle fois l'ensemble de la plateforme uniquement pour y ajouter un espace d'activités. Notre travail à rendre la synchronisation la plus légère possible serait plus d'actualité.

Dans notre implémentation actuelle, pour s'inscrire à un nouvel espace d'activités, un étudiant doit se connecter sur son site de référence via un naviagteur web, s'inscrire au cours de son choix et ensuite lancer une synchronisation depuis son ordinateur personnel utilisant Claroffline. De cette manière l'ensemble des ressources nouvellement mise à sa disposition en ligne seront téléchargées.

Nous sommes arrivés à ce choix car après réflexion, l'inscription hors ligne (sans connexion Internet) à un espace d'activités ne montre que peu d'intérêt. En effet, nous avons analysé deux possibilités de gérer l'inscription à un espace d'activités dit publique sans avoir d'accès Internet.

Soit quand l'utilisateur s'inscrit à un nouvel espace d'activité, il bénéficie de directement de l'ensemble des ressources de cet espace. Cette solution implique le téléchargement préalable de toutes les ressources de tous les espaces d'activités publiques. Ceci représente une quantité relativement élevée d'informations transférées et stockées sur l'ordinateur. De plus si l'utilisateur ne désire s'inscrire qu'à peu, voire aucun d'espace d'activités ces informations seront inutiles.

Soit l'utilisateur pourrait s'inscrire à un nouvel espace d'activités mais ne disposerait pas de son contenu. Le contenu de celui-ci ne serait téléchargé que lors de la synchronisation suivante requierant, elle, une connexion Internet. Nous avions sérieusement commencé à envisager cette solution et l'avions implémenté comme suit. Lors de l'étape de synchronisation, nous ajoutions la liste des espaces d'activités publiques en fin de manifeste. Cette liste d'espace d'activités publiques était alors chargées et les espaces d'activités inexistant dans la base de données de l'ordinateur utilisant Claroffline étaient créés. En procédant ainsi,

l'utilisateur voyait apparaitre les espaces d'activités sur la page d'inscription.

Toutefois, après de plus amples analyses, nous avons abandonné cette solution car elle présente un défaut de taille. Elle ne permet pas de tenir compte efficacement des contraintes d'inscription à un espace d'activités. Ces contraintes peuvent être diverses : un nombre maximum d'inscrits, une période limitée d'inscription,... Au final nous avons décidé d'écarter la possibilité de s'inscrire à un cours hors ligne car l'intérêt en était limité comparativement à la complexité engendrée sur le bundle.

12.3 Gestion des fuseaux horaires

Lors de la phase d'analyse de ce projet, M Mercenier nous a rappelé qu'il était possible de rencontrer un cas dans lequel un utilisateur serait amené à se synchroniser depuis un fuseau horaire différent de celui sur lequel est localisé la plateforme de référence. En effet, cette remarque soulève des questions avec notre module de synchronisation puisque les données transmises dépendent de la date de la synchronisation et de la date de modification des ressources. La date actuellement utilisée est l'heure de l'ordinateur qui exécute Claroline. Le problème ne survient donc que lorsque le fuseau horaire diffère entre l'ordinateur utilisant Claroffline et le serveur distant.

Actuellement notre code ne prend pas cette spécificité en compte, nous comparons la date de l'ordinateur avec la date sauvegardée en base de données. Une solution qui pourrait être appliquée pour prendre en charge les fuseaux horaires serait d'enregistrer en base de données toutes les dates sur un fuseau horaire fixé arbitrairement, par exemple GMT. Il faudrait alors adapter ces dates au fuseau horaire de la machine à chaque utilisation de celles-ci. Bien entendu pour appliquer cette solution il faudrait réviser l'ensemble du code de Claroline et ne pas se limiter au OfflineBundle. Etant donné le temps impartis et ayant mis l'accent sur les fonctionnalités premières de notre bundle, nous n'avons pu révisé l'ensemble du code de Claroline.

Chapitre 13

Conclusion

Quelle belle conclusion [**?**, **?**, **?**, **?**]
Le code a ete clean a l'aide de PHP CS Fixer [**?**]
CODE REVISE et APPROUVE par les développeurs de Claroline

Citer les références dans le texte !

Bibliographie

[1] Boney, L., Tewfik, A.H., and Hamdy, K.N., “Digital Watermarks for Audio Signals," *Proceedings of the Third IEEE International Conference on Multimedia*, pp. 473-480, June 1996.

[2] Goossens, M., Mittelbach, F., Samarin, *A LaTeX Companion*, Addison-Wesley, Reading, MA, 1994.

[3] Kopka, H., Daly P.W., *A Guide to LaTeX*, Addison-Wesley, Reading, MA, 1999.

[4] Pan, D., “A Tutorial on MPEG/Audio Compression," *IEEE Multimedia*, Vol.2, pp.60-74, Summer 1998.

Annexes

Annexe A

`OfflineRessource` Exemple

Vous trouverez ci-dessous le code source d'exemple pour l'ajout d'une ressource au OfflineBundle.

```
<?php

/*
 * This file is part of the Claroline Connect package.
 *
 * (c) Claroline Consortium <consortium@claroline.net>
 *
 * For the full copyright and license information,
 *                          please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Claroline\OfflineBundle\Model\Resource;

use Claroline\CoreBundle\Entity\User;
use Claroline\CoreBundle\Entity\Resource\ResourceNode;
use Claroline\CoreBundle\Entity\Workspace\Workspace;
use Claroline\CoreBundle\Manager\ResourceManager;
use Claroline\CoreBundle\Persistence\ObjectManager;
use Claroline\OfflineBundle\Model\SyncInfo;
use JMS\DiExtraBundle\Annotation as DI;
use \DOMDocument;
use \DateTime;
use \ZipArchive;

/**
 * @DI\Service("claroline_offline.offline.example")
 * @DI\Tag("claroline_offline.offline")
 */
class OfflineExample extends OfflineResource
{
```

```
/**
 * Constructor.
 *
 * @DI\InjectParams({
 *    "om"
      = @DI\Inject("claroline.persistence.object_manager"),
 *    "resourceManager"
      = @DI\Inject("claroline.manager.resource_manager"),
 *    "userManager"
      = @DI\Inject("claroline.manager.user_manager"),
 *    "em"
      = @DI\Inject("doctrine.orm.entity_manager")
 * })
 */
public function __construct(
    ObjectManager $om,
    ResourceManager $resourceManager,
    UserManager $userManager,
    EntityManager $em
)
{
  $this->om = $om;
  $this->resourceNodeRepo =
   $om->getRepository('ClarolineCoreBundle:Resource\ResourceNode');
  $this->userRepo =
   $om->getRepository('ClarolineCoreBundle:User');
  $this->resourceManager = $resourceManager;
  $this->userManager = $userManager;
  $this->em = $em;
}

// Return the type of resource supported by this service
public function getType(){
    return 'example';
}

/**
 * Add informations required to check and recreated
        a resource if necessary.
 *
 * @param \Claroline\CoreBundle\Entity\Resource\ResourceNode
        $resToAdd
```

```
    * @param \ZipArchive $archive
    */
   public function addResourceToManifest($domManifest,
                $domWorkspace, ResourceNode $resToAdd,
                ZipArchive $archive, $date)
    {
        parent::addNodeToManifest($domManifest,
            $this->getType(), $domWorkspace, $resToAdd);
        /*
        *   Rajouter ici les informations supplementaires
            necessaires a la bonne gestion
        *   de la ressource par la suite.
        */
        return $domManifest;
    }

/**
 * Create a resource of the type supported
      by the service based on the XML file.
 *
 * @param \Claroline\CoreBundle\Entity\Workspace\Workspace
      $workspace
 * @param \Claroline\CoreBundle\Entity\User $user
 * @param \Claroline\OfflineBundle\Model\SyncInfo $wsInfo
 * @param string $path
 *
 * @return \Claroline\OfflineBundle\Model\SyncInfo
 */
public function createResource($resource,
                  Workspace $workspace, User $user,
                  SyncInfo $wsInfo, $path)
{
    /*
  *   Rajouter ici les operations necessaires
            a la re-creation de la ressource.
  */
    return $wsInfo;
}

/**
 * Update a resource of the type supported
      by the service based on the XML file.
 *
```

```
     * @param \Claroline\CoreBundle\Entity\Resource\ResourceNode
                                                $node
     * @param \Claroline\CoreBundle\Entity\Workspace\Workspace
                                                $workspace
     * @param \Claroline\CoreBundle\Entity\User $user
     * @param \Claroline\OfflineBundle\Model\SyncInfo
                                                $wsInfo
     * @param string $path
     *
     * @return \Claroline\OfflineBundle\Model\SyncInfo
     *
     */
    public function updateResource($resource, ResourceNode $node,
      Workspace $workspace, User $user, SyncInfo $wsInfo, $path)
    {
        /*
      *   Rajouter ici les operations necessaires
                a la mise-a-jour de la ressource.
      */
        return $wsInfo;
    }

    /**
     * Create a copy of the resource in case of conflict
      (e.g. if a ressource has been modified both offline
      and online)
     *
     * @param \Claroline\CoreBundle\Entity\Workspace\Workspace
                                                  $workspace
     * @param \Claroline\CoreBundle\Entity\Resource\ResourceNode
                                                   $node
     * @param string $path
     */
    public function createDoublon($resource, Workspace $workspace,
                    ResourceNode $node, $path)
    {
    /*
     *    Rajouter ici les operations necessaires a creation
              d'un doublon pour la ressource si necessaire.
     */
          return;
      }
}
```

Annexe B

Manuel d'installation

Mode d'emploi

Ici on va expliquer des solutions détaillées plus performante en fonction de l'OS

PHP en variable d'environnement ! ! ! !

Configuration de XAMPP à la racine.

Erreur de lancement tomcat, filezilla et je ne sais plus quoi, simplement parce qu'on les a enlever pour gagner de la place.

Configuration du php.ini => les extensions requises, le timeout.

Time out du CURL dans le code ?

Configuration des constantes.

Voir SyncConstant => Plateforme de destination

Explication des constantes SYNC_DOWN_DIR et SYNC_UP_DIR

B.1 Prérequis

Installation de PHP

Extensions de PHP intl zend_opcode curl gd2

B.2 Configuration de Claroline sur une plateforme connectée

ajout de OfflineBundle sur une plateforme Claroline existante : OfflineBundle a été conçu comme un plugin pour Claroline.

Son ajout sur une plateforme Claroline Connect existante se fait actuellement manuellement. Claroline ne dispose aujourd'hui (version 3.1.0) pas encore de panneau de configuration pour l'installation des plugin.

Il faut donc passer par composer, ClarolineOfflineBundle est distribué sur Packagist le dépot (principal ? ? ?) de composer. L'adminsitrateur d'une plateforme claroline doit introduire les commandes suivantes :

```
:~$ composer require ClarolineOfflineBundle
:~$ claroline install
```

Le plugin devrait alors être intégré à votre plateforme Claroline

B.3 Configuration de Claroline sur un ordinateur personnel

Full ScreenShoot

Référence

Variante si .exe créé avec InnoSetup

Attention utilisateurs Linux

Ou à la racine d'un autre repertoire pour les cle USB ? ?

A vérifier

Epxliquer plus ?

Comment utiliser Claroline Offline Bundle sur un ordinateur personnel ?Nous avons détaillé dans le chapitre Installationcomment il était possible pour un administrateur système de créer une archive d'installation afin de la rendre disponible pour les utilisateurs offline.

Cette section s'adresse aux utilisateurs offlines munis d'un tel installateur. Décompressez l'archive dans votre répertoire `C:\xampp\`Vous pouvez maintenant lancer l'application Claroline Offline en double cliquant sur le fichier `claroffline`. Si vous souhaitez avoir cette application disponible en raccourcis sur votre bureau en procédant de la manière suivante : faites un clic droit sur le fichier `claroffline` et selectionnez "Send To>Desktop (create shortcut)" en français "Envoyer vers>Bureau (créer un raccourcis).

Questions de perfomances

Linux Installation sur un linux : Installer un serveur apache et php 5.5 (requiert linux >= 13.04) pour se faire documentation sur http://doc.ubuntu-fr.org/apache2 ainsi que l'installation de MySql http://doc.ubuntu-fr.org/mysql

B.3.0.1 Windows XP

PHP5.5 n'est pas supporté, il faut donc php 5.4 pour cette version de PHP, c'est la distribution 1.8.2 de XAMPP qui est aujourd'hui la version la plus à jour.

Annexe C

Manuel d'utilisation

Il est important de noter que ce bundle a été développé lorsque la plateforme Claroline Connect était encore en développement.

Ajout d'un nouvel utilisateur

Le CoreBundle est actuellement à la révision 2.14

ENOORME FLAG Mot de passe utilisateur

EXPLIQUER a l'utilisateur le coup d'une synchronisation arrete et donc la date est fixée à date d'envoi pour reprendre la ou l'on c'etait arreté. En cas de coupure, les infos chargées seront donc celles ou ça a aboutie. Mais il faut relancer une deuxième synchro. Pour avoir un taf complet il faut relancer la synchro.

Contenu :

C.1 Introduction

Bienvenue sur le manuel d'utilisation du OfflineBundle de Claroline. Ce bundle permet à l'utilisateur de synchroniser son ordinateur personnel avec un serveur claroline distant. En quoi consiste la synchronisation de notre bundle ? La synchronisation effectuée par notre bundle vous permet d'ouvrir Claroline et de travailler sur la plateforme dès lors que vous n'avez pas de connexion internet. Vous disposerez alors d'un accès à l''ensemble de vos espaces d'activités et aux différentes ressources qu'ils contiennent. Si vous êtes formateur, vous pourrez également continuer à produire du contenu pour les espaces d'activités dont vous êtes manager.

Lorsque vous étudiants ou formateur, vous pourrez, dès que vous disposerez d'un accès à Internet, lancer une synchronisation avec le serveur distant. Cette synchronisation se chargera d'envoyer tout le travail que vous avez effectué hors ligne et de récupérer l'ensemble des nouveaux contenu disponible en ligne. La contenu de la plateforme Claroline installée sur votre ordinateur personnel sera maintenant à jour et vous pourrez travailler avec les nouvelles ressources même si vous n'avez plus accès à Internet.

C.2 Installer Claroffline sur son ordinateur

Si vous lisez ce manuel, nous supposons que vous êtes utilisateur d'une plateforme Claroline et que vous disposez d'un installateur de Claroffline créé par l'administrateur de votre serveur Claroline. Dans le but de proposer des instructions les plus spécifiques possibles, nous invitons les utilisateurs Windows à ce rendre à la section C.2.1, les utilisateurs Linux à passer au point C.2.2. Si vous êtes un utilisateur avancé disposant des connaissances techniques

Pas trop technique comme entrée en matière ?

Autres OS ?

nécessaire et que vous souhaitez comprendre le fonctionnement de l'installateur et des pré-requis pour disposer de Claroline sur votre ordinateur personnel nous vous invitons à lire le Chapitre 9 page 71 détaillant l'installation d'*OfflineBundle*

C.2.1 Windows

C.2.2 Linux

C.2.3 Autres système d'exploitation

Si vous n'utilisez ni Windows, ni Linux, nous n'avons pas prévu d'installateur intégrants les différents composants requis pour faire fonctionner Claroffline.Rassurez-vous, Claroline et son plugin OfflineBundle ne nécessitent pas de pré-requis très spécifique. Nous vous invitons à le Chapitre afin vous informer sur les différents composants nécessaire à la plateforme. Il est possible que vous soyez confronté à des concepts plus techniques. Nous nous excusons pour les désagréments occasionnés.

Nous nous excusons de ne pas prendre en charge ? Nous ne supportons pas...

reference au manuel ou au chapitre d'install

C.3 Configurer son compte

Une fois la plateforme installée et démarrée, la première chose que vous serez amené à faire est récupérer votre compte. Au démarrage, vous serez donc redirigé sur un formulaire vous demandant votre nom d'utilisateur, votre mot de passe et le serveur Claroline à contacter.

Hum hum j'ai l'impression d'etre une compagnie d'assurance

C.4 Se synchroniser

C.5 S'inscrire à un nouveau cours

C.6 Cas particulier de synchronisation

Discussino en cas d'échec, reprise....

Annexe D

Code Source

review

Le code source du OfflineBundle est accessible en ligne sur le site *Github*. *Github* est le service d'hébergement de code source web utilisé comme dépot pour le projet Claroline.

L'URL des sources de notre bundle est : https://github.com/vVYou/OfflineBundle.

Une version électronique en PDF de ce mémoire y est également disponible.