Part 2: Grails and services [TO TRANSLATE]

This a is an include of this guide: Guide: Add a new resource type on Cytomine

Ce document reprend(ra) les différentes étapes de l'ajout d'une ressource dans Cytomine.
Il ne s'agit pas:

- d'un manuel complet sur le fonctionnement interne de Cytomine,
- d'un tutoriel sur Grails (Domain, Controller, ...).

Il s'agit simplement d'une série d'étape a suivre afin d'ajouter de nouveaux types de données.

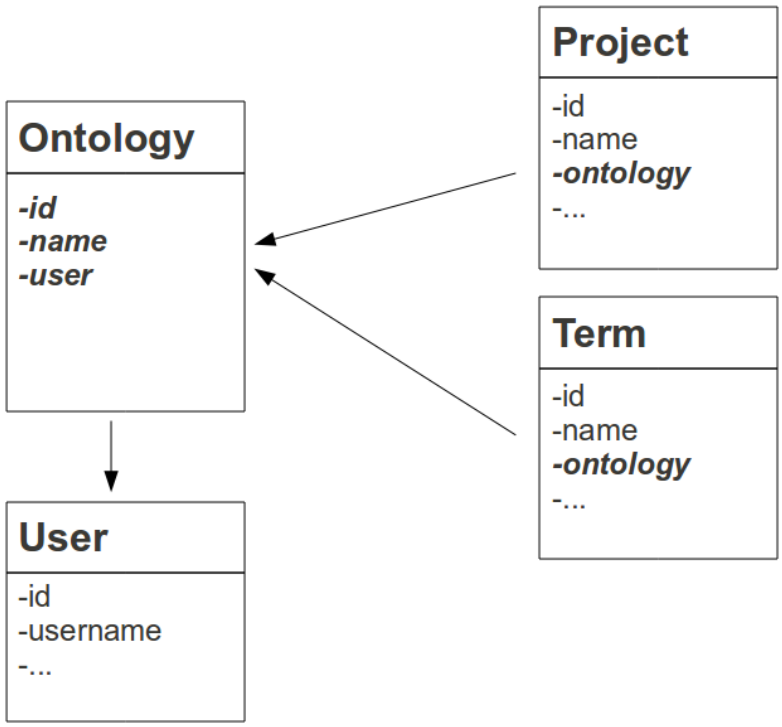
Pour un manuel de Cytomine: Documentation technique (pas à jour),
Pour un manuel de Grails: Livre "The definitive Guide to Grails" (G. Rocher et J. Brown) disponible au Giga ou via l'intranet de l'ULg gratuitement sur http://dx.doi.org/10.1007/978-1-4302-0871-6 (chapitre 1, 2, 3, 4, 6 et 11 conseillés)

L'architecture de Cytomine évoluant de temps en temps, ce document pourrait ne pas être totalement à jour.

Définition de la resource

Nous utiliserons comme exemple l'ajout de la resource Ontology.
Une ontologie est un ensemble structuré de termes (Term) que nous pourrons lier à un projet.
Nous souhaitons stocker, en plus de son identifiant, le nom de l'ontologie ainsi que que son créateur.
Exemple: Projet "project" utilise l'ontologie "ontology" qui dispose des termes "term1", "term2",...

Nous aurons donc le schéma ci-dessous:



Il sera nécessaire:
-d'implémenter la ressource Ontology avec tous ses champs,
-d'ajouter un lien depuis Ontology vers User,
-d'ajouter un lien depuis Project vers Ontology,
-d'ajouter un lien depuis Term vers Ontology.

Action disponible pour la resource

- Lister toutes les ontologies accessibles.
- Lister toutes les ontologies définies par un utilisateur.
- Récupérer une ontologie précise (sur base de son identifiant).
- Ajouter/Modifier/Supprimer une ontologie.

Sur base de ces actions, nous pouvons définir les URL selon le principe REST.
La plupart du temps, nous suivrons le shéma suivant:

URL	Action
`api/resource.json`	`Action GET: lister toutes les éléments de type 'resource'` `Action POST: ajouter un élément de type 'resource'`

`api/resource/$id.json`	`Action GET: récupérer l'élément de type 'resource' dont l'id est $id`
	`Action PUT: modifier l'élément de type 'resource' dont l'id est $id`
	`Action DELETE: supprimer l'élément de type 'resource' dont l'id est $id`

`api/domain/$id /resource.json` ou `api/resource.json? domain=$id`	`Action GET: lister les éléments de type 'resource' filtrer par le domain de type 'domain' dont l'id est $id`

Pour notre exemple, nous aurons:

URL	Action

api/ontology.json	Action GET: lister toutes les ontologies (accessibles par l'utilisateur) Action POST: ajouter une ontologie
api/ontology/$id.json	Action GET: récupérer l'ontologie dont l'id est $id Action PUT: modifier l'ontologie dont l'id est $id Action DELETE: supprimer l'ontologie dont l'id est $id
api/user/$id/ontology.json ou api/ontology.json?user=$id	Action GET: lister les ontologies définie par l'utilisateur dont l'id est $id

Création du domaine

Les domaines Grails sont situés dans le répertoire de développement /grails-app/domain/package1/.../packageN/MyDomain.groovy.
Nous créons donc la classe Ontology (be.cytomine.ontology.Ontology) dans le fichier /grails-app/domain/be/cytomine/ontology/Ontology.groovy.

Création des champs

Les différents champs du domaine seront automatiquement converti en colonne dans la table du domaine.
Hibernate fera la correspondance entre les types Java (String, Long,...) et les types SQL (Varchar, BigInt,...).

- Ne pas ajouter d'id, de date de création et de date d'édition (ils seront hérité grâce à CytomineDomain, voir section suivante))
- Eviter les types primitifs au profit de leurs équivalents objets (~~long~~ => Long),
- Eviter les HasMany (difficilement compatible avec notre système de commande) sauf si l'ajout/suppression d'élément dans le HasMany ne se fait que via l'édition du domain.

Nous ajoutons le champ name ainsi qu'un lien vers le domaine User. Pour l'instant, nous n'ajouterons pas encore l'identifiant.
Ses contraintes seront d'avoir un nom not null et non vide (""). Le user devra être not null (par défaut, si pas de contrainte définie).

```
class Ontology {

        String name
        User user

        static constraints = {
                name(blank: false)
        }
}
```

Dans Term et Project, nous ajouterons un champs: "Ontology ontology"

Héritage Cytomine Domain

Les domaines utilisés par Cytomine doivent hérités de CytomineDomain (src/groovy/be/cytomine/CytomineDomain.groovy).
Cette classe fournit:
-un id (automatique),
-des champs created/updated de type Date (automatiquement mis à jour).
-des méthodes "utiles"

En cas de redéfinition de *static mapping*, ne pas oublier d'ajouter l'assignation de l'identifiant définie dans CytomineDomain: "id(generator: 'assigned', unique: true)".

Contrainte d'unicité

La méthode checkAlreadyExist() est automatiquement appelée pour chaque ajout/modification.
Si elle n'est pas surchargée, elle ne fait rien, sinon elle vérifie que l'objet que l'on tente d'insérer ou de modifer, ne violera pas une contrainte d'unicité.
En cas de violation, elle devra déclencher une "AlreadyExistException" qui correspondra au code HTTP 409 (conflict).
Il est nécessaire d'effectuer la vérification dans une nouvelle session (withNewSession). En effet, l'objet courant n'est pas encore sauvegardé dans la DB mais est quand même présent dans la session Hibernate courante.
La contrainte d'unicité est appelée lors de l'insertion et de l'édition, il est donc nécessaire de ne pas déclencher d'exception si l'identifiant est identique (l'objet édité vs le même objet dans la DB pas encore édité).

- Effectuer la vérification dans une nouvelle session (myDomain.withNewSession)
- Déclencher une AlreadyExistException (HTTP 409) si nécessaire.

Dans notre exemple, nous n'accepterons pas d'ajouter une Ontologie avec le même nom.

```
void checkAlreadyExist() {
        Ontology.withNewSession {
                if(name) {
                        Ontology ontology = Ontology.findByName(name) //must be null or with the same id
                        if(ontology!=null && (ontology.id!=id)) {
                                throw new AlreadyExistException("Ontology " + ontology.name + " already exist!")
                        }
                }
        }
}
```

Formattage du JSON

Les données reçues et envoyées par le serveur seront au format JSON. Cela implique donc de devoir convertir un domaine en chaîne JSON.
Afin de définir le JSON complet obtenu lors d'une consultation, ajout, modification ou suppression, nous définissons la méthode static def getDataFromDomain(def domain).
Au démarrage du serveur, un script parcours l'ensemble des domaines afin d'enregistrer cette méthode pour être implicitement utilisée pour la génération des JSON.
La méthode renvoyer une Map dont les associations seront les noms des attributs et leur valeur respective.

- Pour les champs de type class (autres domaines, ...), ne renvoyer que l'id (exemple: user.id => {'user': 14}), sinon Grails imbriquera le JSON complet (exemple: user => {'user': {'id': 14, 'username': ...}}
- Pour les champs data, renvoyer le timestamp (Date.getTime())
- D'une manière générale, attention au performance (éviter les requêtes dans ce code => problème SELECT n+1)
- Pour les longs listing, il est préférable d'effectuer une requête SQL directe et d'ajouter les résultats dans une liste de Map (chaque ligne sera une Map), les performances seront **très** fortement supérieures (voir UserAnnotationService).

```
static def getDataFromDomain(def domain) {
    def returnArray = CytomineDomain.getDataFromDomain(domain)
    returnArray['name'] = domain?.name
    returnArray['user'] = domain?.user?.id
    return returnArray
}
```

Remarque: On pourrait également ajouter l'arbre complet des termes (méthode tree()) avec: returnArray['tree'] = domain.tree()

Création du domain depuis le JSON

Il est également nécessaire de faire la traduction JSON vers Domain pour les ajouts, les modifications et les undo/redo.
Pour cela, chaque domaine implémentera une méthode statique *insertDataIntoDomain(def json,def domain)* dont le but sera d'insérer les informations du JSON dans le domaine en paramètre.
Si il s'agit d'un ajout, le domaine ne sera pas passé en paramètre (car inexistant), les paramètres seront donc *(def json,def domain = new Domain())* (si domain n'est pas passé en argumant, on utilisera new Domain()).

- Utiliser les méthodes de la classe *JSONUtils* qui permette d'extraire proprement des valeurs d'un JSON. Elles sont disponibles pour les types String, Long, ...
- Pour l'extraction d'une référence à un domaine, utiliser JSONUtils.getJSONAttrDomain(json,"domain",new Domain(),mandatory), cette méthode récupérera la valeur de json.domain et renverra l'objet de type Domain dont l'id correspond à json.domain (si mandatory = true, déclenche une exception si l'objet n'est pas trouvé). Une autre méthode plus complète permet de spécifier à quel champs exact correspond json.domain (par défaut: id).

```
static Ontology insertDataIntoDomain(def json,def domain = new Ontology()) {
        domain.id = JSONUtils.getJSONAttrLong(json,'id',null)
        domain.name = JSONUtils.getJSONAttrStr(json, 'name')
        domain.user = JSONUtils.getJSONAttrDomain(json, "user", new SecUser(), true)
        return domain;
}
```

Création du Service

Les services doivent contenir le code logique de l'application (requêtes, calculs,...).
Ils contiendront donc les méthodes de CRUD (create, read, update and delete).

> Ce document ne tient pas compte de la gestion des droits et de la sécurité (un document spécifique sera mis en ligne prochainement) ainsi que des problèmes de performances (idem).

Les services sont définis dans *grails-app/services*. Ils hériteront du service *ModelService* qui fournira des propriétés et des méthodes utiles.

```
class ResourceService extends ModelService {

}
```

Propriété à définir

Deux propriétés sont héritées de *ModelService*:
-static transactional (*true* par défaut)
-boolean saveOnUndoRedoStack (true par défaut)

Le premier est une propriété de Grails pour définir l'atomicité des méthodes du service.
Si une méthode effectue les étapes suivantes:

1. add annotation a
2. add term t to annotation a

Si la deuxième opération échoue, il sera généralement préférable d'annuler la première. Hibernate effectuera automatiquement le rollback si *transactional = true*.

La seconde propriété permet d'indiquer que le domain est "undo/redo"-able (l'utilisateur peut annuler/restaurer une opération add/update/delete sur ce domaine).

Méthodes à définir

Un mécanisme COC (Convention over configuration) est mis en place pour la gestion des create, update et delete (ces méthodes seront définies plus tard). Pour que ce mécanisme soit fonctionnel, il est nécessaire de définir une méthode *currentDomain()* qui renvoi la classe du domaine associé au service.
Cette méthode indiquera le domaine à utiliser lors des opérations de CRUD.

```
def currentDomain() {
        return Resource // its just the corresponding domain class
}
```

La méthode *def retrieve(JSONObject json)* doit permettre de récupérer l'objet dont les informations sont fournies en JSON. Elle sera utilisé lors des opérations de modifications et de suppressions pour récupérer l'objet concerné.
Si nous lui fournissons un JSON '{id: 1234}', elle devra renvoyer l'instance du domaine dont l'id est 1234.
Cette méthode est définie dans ModelService. Par défaut, elle utilise l'id du JSON pour récupérer l'objet avec cet id.

```
def retrieve(JSONObject json) {
    CytomineDomain domain = currentDomain().get(json.id) //if we are in OntologyService, we should have
Ontology.get(...)
    if (!domain) {
        throw new ObjectNotFoundException("${currentDomain().class} " + json.id + " not found")
    }
    return domain
}
```

Si les URL de suppression (ou de modification) du domain utilise uniquement l'id du domaine, il ne sera pas nécessaire de la redéfinir dans le service.
Ce sera par contre nécessaire pour les ressources qui n'utilise pas un identifiant "simple" dans les URL (mais une clé "composite").

Exemple:
DELETE /api/project/**$id**.json => pas besoin de réécrire la méthode retrieve
DELETE /api/user/**$idUser**/group/**$idGroup** => nécessaire de réécrire la méthode retrieve, car pas d'id "simple" {~~id: 1234~~, idUser: 12, idGroup: 34}. Cette méthode sera:

```
def retrieve(JSONObject json) {
     def user = User.read(json.idUser)
     def group = Group.read(json.idGroup)
     CytomineDomain domain = UserGroup.findByUserAndGroup(user,group) //or currentDomain().findByUserAndGroup
(user,group)
     if (!domain) {
        throw new ObjectNotFoundException("UserGroup ${json.idUser}/${json.idGroup} not found")
     }
     return domain
}
```

Le service de base de notre exemple sera donc:

```
class OntologyService extends ModelService {

    static transactional = true
    boolean saveOnUndoRedoStack = true

    def currentDomain() {
        return Ontology
    }

    //no need to override retrieve, Ontology use its main id in DELETE/UPDATE URL
}
```

Méthodes d'accès

Get

Cette méthodes permettent de récupérer un domain selon son id.

```
Resource get(def id) {
    Resource.get(id)
}
```

Read

Un read est identique à un get mais empêche la modification de l'objet (préférable si l'objet est juste utilisé en lecture seule).
La méthode read est identique mais utilise Resource.read(id) au lieu de Resource.get(id).

List

Nous définirons généralement une méthode list sans paramètre, ainsi que des méthodes list dont les paramètres seront les filtres.

```
List<Resource> list() {
    Resource.list()
}

List<Resource> list(Something something) {
    Resource.findAllBySomething(something) //if not possible or more complex request, use HQL or createCriteria
}
```

Remarque: comme signalé précédemment, nous ne gérons pas les droits d'accès dans ce document, tous les éléments seront accessibles.

Pour notre exemple, nous aurons donc:

```
class OntologyService extends ModelService {

        ...

    Ontology read(def id) {
        Ontology.read(id)
    }
    Ontology get(def id) {
        Ontology.get(id)
    }
    def list() {
        Ontology.list()
    }
    def list(User creator) {
        Ontology.findAllByUser(creator)
    }

        ...
}
```

Méthodes d'ajout, de suppression et de modification

Cytomine utilise un système de commande. Chaque action "add/update/delete" sera emballée dans une commande.
L'interêt des commandes est de pouvoir suivre l'activité et surtout la possibilité de les annuler/restaurer (undo/redo).
Pour cela, les commandes seront stockées sur une pile d'annulation (*UndoStack*), elles pourront être annulées et déplacées sur la (*RedoStack*) afin d'être restaurées.

Exemple:
-L'utilisateur ajoute une annotation => Ajout de la commande sur la pile *UndoStack*, exécution du code d'ajout,
-L'utilisateur annule sa commande d'ajout => Suppression de la pile *UndoStack*, exécution du code de suppression ("inverse" d'ajout), ajout sur la pile *RedoStack*,
-L'utilisateur restaure sa commande d'ajout => Suppression de la pile *RedoStack*, exécution du code d'ajout, ajout su la pile *UndoStack*.
Pour plus de détail, voir le manuel du développeur de Cytomine.

Implémentation de Add

Une méthode classique add d'un service est la suivante:

```
def add(def json) {
    //don't forget to add some security check here...
        SecUser currentUser = cytomineService.getCurrentUser()
        Command command = new AddCommand(user: currentUser)
        return executeCommand(command,null,json)
}
```

Elle contient généralement des appels à des méthodes de sécurité. Nous n'en tiendrons pas compte dans ce document ([Guideline] Gestion des droits d'accès (ACL) dans Cytomine Server).

Le paramètre de la méthode add est le JSON reçu en paramètre de la requête. Dans cette méthode, nous créons une nouvelle *AddCommand* avec l'utilisateur qui a effectué la requête (currentUser).
La méthode executeCommand héritée de *ModelService* prendra en paramètre notre nouvelle commande ainsi que le JSON. Son deuxième paramètre est le domaine, nous passerons null car il n'existe pas encore.
Cette méthode détectera automatiquement la table a modifier en base de données (via currentDomain() définie au début de cette section) et le type d'opération à effectuer en fonction de la command (*Add*, *Update* ou *Delete*).
Elle sauvegardera également la commande et l'ajoutera sur la pile des Undo (si *saveOnUndoRedoStack = true* dans le service).

Implémentation de update

Similaire à *Add* mais utilise une *EditCommand*.

```
def update(Domain domain,def json) {
        SecUser currentUser = cytomineService.getCurrentUser()
    Command command = new EditCommand(user: currentUser)
        return executeCommand(command,domain,json)
}
```

Implémentation de delete

La méthode *delete* prendra, en plus du domaine à supprimer, deux autres arguments: *transaction* et *task*.
Le contrôleur initialisera une nouvelle transaction Cytomine à chaque requête *delete* (afin que les autres données supprimées en cascade soient dans la même transaction).
Parfois un paramètre *Task* peut être défini. L'objet *Task* permettra au client de consulter l'évolution d'une (longue) tâche.
Si il est passé en argument, le code de suppression mettra l'objet Task à jour afin que l'utilisateur puisse visualiser l'avancement de son opération (via une barre de progression par exemple).

```
def delete(Domain domain, Transaction transaction = null, Task task = null) throws CytomineException {
    SecUser currentUser = cytomineService.getCurrentUser()
    Command command = new DeleteCommand(user: currentUser,transaction:transaction)
    return executeCommand(command,domain,null,task)
}
```

La méthode *delete* sera également appelée lors des suppressions en cascade (voir section Suppression des dépendances).

Déclencheur

Le code qui effectue réellement les opérations d'ajout/suppression/modification étant définis dans *ModelService*, il est a priori impossible d'effectuer des opérations spécifique à un domaine juste avant ou après un add/update/delete.
Pour permettre cela, nous avons un mécanisme de déclencheur avec 6 méthodes:

```
def beforeAdd(def domain) {

}

def beforeDelete(def domain) {

}

def beforeUpdate(def domain) {

}

def afterAdd(def domain, def response) {

}

def afterDelete(def domain, def response) {

}

def afterUpdate(def domain, def response) {

}
```

Elles pourront être redéfinies dans le service de notre ressource afin d'exécuter un bout de code particulier.
Les méthodes *before* prennent en argument uniquement le domaine qui va être ajouté/modifié ou supprimé. Les méthodes *after* prennent en plus du domaine, la réponse complète (message, le domaine créé,...) afin de pouvoir la modifier.

Pour *Ontology*, il sera nécessaire par exemple, d'ajouter les droits d'administrations à l'utilisateur qui vient de la créer.

```
def afterAdd(def domain, def response) {
        permissionService.addPermission(domain, cytomineService.currentUser.username, BasePermission.
ADMINISTRATION)
}
```

Génération du message

Grails utilise un mécanisme *i18n* pour générer les messages. Concretement, on trouvera par défaut dans le répertoires *i18n* de *grails-app*, une série de fichier *message_LA.properties* où *LA* représente les codes de langues ainsi qu'un *message.properties* (pour toutes les langues).
Ce mécanisme n'est pas (encore) complètement utilisé par notre application. Nous définirons juste les messages en anglais dans *messages.properties*.

Pour chaque ressource, nous aurons 3 définitions de message:

```
be.cytomine.AddResourceCommand = Resource {1} (id={0}) added
be.cytomine.EditResourceCommand = Resource {1} (id={0}) edited
be.cytomine.DeleteResourceCommand = Resource {1} (id={0}) deleted
```

Chaque {x} sera remplacé par le paramètre fournit à la position x. Si nous créons "*be.cytomine.AddResourceCommand*" avec en paramètre [123,"test"], nous aurons le message "Resource test (id=123) added".

Nous avons un mécanisme qui injecte automatiquement ces informations selon la méthode *getStringParamsI18n(def domain)* définie dans le service de la resource.

```
def getStringParamsI18n(def domain) {
     return [domain.id, domain.name] // => [123,"test"]
}
```

Pour notre exemple, nous pourrions juste imprimer le message: *Ontology myOntology (creator=johndoe) added* (ou edited ou deleted)
Dans message.properties du répertoire *i18n*, nous ajoutons:

```
be.cytomine.AddOntologyCommand = Ontology {0} (creator={1}) added
be.cytomine.EditOntologyCommand = Ontology {0}  (creator={1}) edited
be.cytomine.DeleteOntologyCommand = Ontology {0} (creator={1}) deleted
```

Dans *OntologyService*:

```
def getStringParamsI18n(def domain) {
    return [domain.name, domain.user.username]
}
```

Suppression des dépendances

Cytomine dispose d'un mécanisme de gestion de dépendances entre les domaines.
Si B dépend de A et qu'on supprime A, il sera en théorie nécessaire de supprimer B.
La problématique et le mécanisme complet de gestion de dépendance sont expliqués dans Dependency [TOTRANSLATE].

Concrètement, Cytomine détectera automatiquement les dépendances entre les types de domaine. Il le signalera via un échec du test *testMissingDeleteMethodDependency* (classe GenericDependencyTests) avec un ou plusieurs messages.
Notre domaine Ontology dispose d'une dépendance vers User et deux autres domaines ont une dépendance vers Ontology (Project et Term).
Nous devrons donc:

- Traiter le cas des projets associés à l'ontologie à supprimer,
- Traiter le cas des termes associés à l'ontologie à supprimer,
- Traiter le cas des ontologies associées à l'user à supprimer.

L'exécution du test *testMissingDeleteMethodDependency échouera et nous aurons* 3 messages dans les logs:

- "Service **Ontology**Service must implement deleteDependent**Project**(Ontology,transaction) !!!"
- "Service **Ontology**Service must implement deleteDependent**Term**(Ontology,transaction) !!!"
- "Service **SecUser**Service must implement deleteDependent**Ontology**(SecUser,transaction) !!!"

La méthode *delete(Ontology ontology,...)* de *OntologyService* définie précédemment appellera automatiquement les méthodes commençant par *deleteDependent* du service *OntologyService*.

Les méthodes *deleteDepent* peuvent utiliser 3 paramètres:

- L'instance du domaine à supprimer afin de récupérer ses domaines dépendants,
- La transaction qui sera utile pour associer les domaines supprimés en cascade à la même transaction,
- Une tâche qui pourra être mise à jour par la méthode de suppression de dépendance.

```
 def deleteDependentTerm(Ontology ontology, Transaction transaction, Task task = null) {

        taskService.updateTask(task,"Delete ${Term.countByOntology(ontology)} terms") //no prob if task is null
but better to check that before doing request countBy (useless if task is null)
    Term.findAllByOntology(ontology).each {
        termService.delete(it,transaction, null, false)
    }
}
```

Ces méthodes peuvent:

- Supprimer les domaines via des commandes (pour les retrouver dans un undo/redo).
  Exemple dans *OntologyService*: si je fais un undo d'un delete ontology, en plus de restaurer l'ontologie, je veux restaurer ses termes aussi.

```
    def deleteDependentTerm(Ontology ontology, Transaction transaction, Task task = null) {
        Term.findAllByOntology(ontology).each {
            termService.delete(it,transaction, null,false)
        }
    }
```

- Supprimer les domaines sans passer par des commandes (perdu en cas de undo/redo)
  Exemple dans *ProjectService*: les infos de dernière connexion sur une projet pourraient être oubliées en cas de delete (undo/redo) de projet.

```
    def deleteDependentLastConnection(Project project, Transaction transaction,Task task=null) {
        LastConnection.findAllByProject(project).each {
              it.delete()
        }
    }
```

- Changer des infos sur le domain dépendant
  Exemple dans *SecUserService*: si on supprime un user, on pourrait associer ses dépendances à quelqu'un d'autres.

```
    def deleteDependentOntology(SecUser user, Transaction transaction, Task task = null) {
        if(user instanceof User) {
            Ontology.findAllByUser(user).each {
                                it.user = cytomineUser.currentUser //all ontologies created by x will now be
created by current user (user that delete x)
                                it.save()
            }
        }
    }
```

- Refuser la suppression

  Exemple dans *OntologyService*: il n'est pas souhaitable que la suppression d'une ontologie entraîne la suppression complète des projets associés à cette ontologie.

```
def deleteDependentProject(Ontology ontology, Transaction transaction, Task task = null) {
    if(Project.findByOntology(ontology)) {
        throw new ConstraintException("Ontology is linked with project. Cannot delete ontology!")
    }
}
```

- Si une méthode deleteDependent...() fait appel à à une méthode delete() d'un autre service, les méthodes de dépendances seront appelées en cascade.
  Exemple: La suppression d'une ontologie nécessite la suppression des termes qui elle-même nécessite la suppression des liens entre ces termes et les annotations.
  => requête web
  => ontologyService.delete(json, sec)
  => transaction.start()
  => **ontologyService.delete(ontology,transaction)**
  => **ontologyService.deleteDependentTerm(ontology,transaction)**
  ===> **termService.delete(term, transaction)**
  =====> **termService.deleteDependentAlgoAnnotationTerm(term, transaction)**
  =======> //delete en cascade d'autres domaines...
  =====> **termService.deleteDependentAnnotationTerm(term, transaction)**
  =======> //delete en cascade d'autres domaines...
  => **delete ontology**

Pour notre exemple, nous ajouterons les méthodes *deleteDependentTerm* et *deleteDependentProject* définies ci-dessus.

```
class OntologyService extends ModelService {

    static transactional = true
    boolean saveOnUndoRedoStack = true

    def termService

    Ontology read(def id) {
        Ontology.read(id)
    }
    Ontology get(def id) {
        Ontology.get(id)
    }
    def list() {
        Ontology.list()
    }
    def list(User creator) {
        Ontology.findAllByUser(creator)
    }

    def add(def json) throws CytomineException {
        SecUser currentUser = cytomineService.getCurrentUser()
        json.user = currentUser.id
        return executeCommand(new AddCommand(user: currentUser), null, json)
    }

    def update(Ontology ontology, def json) throws CytomineException {
        SecUser currentUser = cytomineService.getCurrentUser()
        return executeCommand(new EditCommand(user: currentUser), ontology, json)
    }

    def delete(Ontology ontology, Transaction transaction = null,Task task = null,boolean printMessage = true)
throws CytomineException {
        SecUser currentUser = cytomineService.getCurrentUser()
        return executeCommand(new DeleteCommand(user: currentUser,transaction:transaction), ontology, null,task)
    }

    def getStringParamsI18n(def domain) {
        return [domain.id, domain.name]
    }

    def afterAdd(def domain, def response) {
                //add the ADMIN permission for current user after create ontology
        aclUtilService.addPermission(domain, cytomineService.currentUser.username, BasePermission.
ADMINISTRATION)
    }

    def deleteDependentTerm(Ontology ontology, Transaction transaction, Task task = null) {
        Term.findAllByOntology(ontology).each {
            termService.delete(it,transaction, null, false)
        }
    }

    def deleteDependentProject(Ontology ontology, Transaction transaction, Task task = null) {
        if(Project.findByOntology(ontology)) {
            throw new ConstraintException("Ontology is linked with project. Cannot delete ontology!")
        }
    }
}
```

Création du Controller

L'utilité principale des controlleurs est de lire/décoder les paramètres d'une requête et d'effectuer l'appel à la bonne méthode du service.
Nous les stockerons dans le répertoire grails-app/controllers/be/cytomine/api/package1/.../packageN avec par convention comme nom RestResourceController (où Resource est le nom du nouveau type de ressource).

- Le code logique (requêtes, calculs,...) doit le plus possible se situer dans les services.

```
package be.cytomine.api.ontology

class RestOntologyController {
}
```

Héritage de RestController

Le controlleur RestController fournit de nombreuses méthodes "utiles" pour les controlleur de domaine.

- responseSuccess(data): transforme data en JSON et renvoie le contenu en réponse avec le code HTTP 200,
- responseNotFound(DomaineName,def id): crée un message d'erreur signalant que la ressource DomaineName n'a pas été trouvée avec cet id (d'autres méthodes de ce type avec plus de paramètres existes).
- add(service, json): fait appel a la méthode add du service en gérant le traitement des exceptions, la sécurité, les ajouts multiples (si on ajoute un json array, au lieu d'un json object...).
- update(service, json): idem mais pour update
- delete(service, json): idem mais pour delete

- Les méthodes responseSuccess/responseNotFound/response n'interrompe pas la méthode en cours (!= return). Le code responseSuccess sera toujours exécuté dans le code ci-dessous (Pour éviter ce problème, il faut le mettre dans un else).

```
if(!data) {
        responseError
}
responseSuccess
```

Injection du service

Comme expliqué précédemment, le contrôleur ne doit pas contenir de code logique, il doit pour cela faire appel au service correspondant.
L'injection des services avec Grails est très simple: il faut définir 'def myService' (première lettre en minuscule) et le service MyService sera automatiquement injecté.

```
package be.cytomine.api.ontology

class RestOntologyController extends RestController {

        def ontologyService

}
```

Définition des méthodes

Au début de ce document, nous définissons les actions disponibles.
Elles correspondront aux méthodes list (lister toutes les ontologies accessibles), listByUser (lister les ontologies accessibles définie par utilisateur), show (récupérer une ontologie), add, update et delete.

```
    def list() {

    }

    def listByUser() {

    }

    def show() {

    }

    def add() {

    }

    def update() {

    }

    def delete() {
    }
```

List

La méthode renvoie simplement la réponse de la méthode list du service.

```
    def list() {
        responseSuccess(ontologyService.list())
    }
```

ListByX

La méthode doit extraire le(s) paramètre(s) sur lequel(s) on filtrera le listing. Elle doit ensuite faire appel à la méthode list(Domain filter) du service.
Il est préférable de faire une vérification sur le paramètre afin de s'assurer que le domaine avec l'identifiant en paramètre existe bien.

```
    def listByUser() {
                User user = userService.read(params.id)
                if(user) {
                         responseSuccess(ontologyService.list(user))
                } else {
                        responseError("User",params.id) //=> cannot find "User" with id = ...
                }

    }
```

Show

La méthode doit extraire le paramètre "id", tenter de lire le domain avec cet id. Si il n'existe pas (domain==null), il faut renvoyer une erreur.

```
    def show() {
        Ontology ontology = ontologyService.read(params.long('id'))
        if (ontology) {
            responseSuccess(ontology)
        } else {
            responseNotFound("Ontology", params.id) // => cannot find "Ontology" with id = ...
        }
    }
```

Add

Le code logique de la méthode add est dans RestController. Il suffira de faire appel à add(service,json).

```
    def add() {
        add(ontologyService, request.JSON)
    }
```

Update

```
    def update() {
        update(ontologyService, request.JSON)
    }
```

Delete

Dans le cas d'un delete, le seul paramètre est l'id fournit dans l'url (pas de JSON).
Nous créerons donc un JSON qui ne contient que l'id fournit en paramètre.

```
    def delete() {
        delete(ontologyService, JSON.parse("{id : $params.id}"),task)
    }
```

Création de l'URLMapping

L'URLMapping permet d'aiguiller les requêtes sur les méthodes des controlleurs en fonction du formattage de l'URL.
Pour chaque nouveau type de ressource, nous créerons un fichier ResourceUrlMappings.groovy dans grails-app/conf/.
Ce fichier contient essentiellement des définitions de type "Avec cet URL + cette action HTTP => Aller sur la méthode xxx du controlleur RestYYYController.

- Pour chaque nouvelle ressource, créer un nouveau fichier UrlMappings
- Pour la définition du controlleur à utiliser, la première lettre doit être en minuscule et il ne faut pas mettre Controller à la fin (RestResourceController deviendra restResource)

Pour rappel, voici les actions qui devront être disponible dans notre exemple:

URL	Action
`api/ontology.json`	`Action GET: lister toutes les ontologies (accessibles par l'utilisateur)` `Action POST: ajouter une ontologie`

`api/ontology/$id.json`	`Action GET: récupérer l'ontologie dont l'id est $id`
	`Action PUT: modifier l'ontologie dont l'id est $id`
	`Action DELETE: supprimer l'ontologie dont l'id est $id`

`api/user/$id/ontology.json`
ou
`api/ontology.json?user=$id`
	`Action GET: lister les ontologies définie par l'utilisateur dont l'id est $id`

Nous créons le fichier OntologyUrlMappings.groovy

```
class OntologyUrlMappings {
    static mappings = {
        "/api/ontology.$format"(controller:"restOntology"){
            action = [GET: "list",POST:"add"]
        }
        "/api/ontology/$id.$format"(controller:"restOntology"){
            action = [GET:"show",PUT:"update", DELETE:"delete"]
        }
        "/api/user/$id/ontology.$format"(controller:"restOntology"){
            action = [GET:"listByUser"]
        }
    }
}
```

Création des Tests

L'implémentation des tests (définis dans test/functional/) est très utile pour vérifier si le code est bien fonctionnel, pour le debugger et, surtout, détecter très rapidement de nouveau bug introduit plus tard par la modification de code (refactoring, modification d'une dépendance,...).
Les tests de Cytomine sont des tests fonctionnels via un client HTTP.
Pour chaque nouveau type de ressource, nous définirons également un fichier ResourceAPI.groovy dans src/groovy/be/cytomine/test/http. Cette classe effectuera les appels HTTP et fournira le code ainsi que la réponse du serveur.

- Par convention, on ajoutera un fichier ResourceTests.groovy pour chaque nouvelle resource.
- Les méthodes doivent commencer par 'test' et indiquer clairement ce qui est testé: testAddOntology, testDeleteOntologyNotExist, ...

```
class ResourceTests  {

        void testASpecificAction() {
                def code = ...
                assert 200 == code
        }

}
```

Création de la classe de test

Nous nous basons sur les actions disponibles dans notre exemple:

- Lister toutes les ontologies accessibles.
- Lister toutes les ontologies définies par un utilisateur.
- Récupérer une ontologie précise (sur base de son identifiant).
- Ajouter/Modifier/Supprimer une ontologie.

Voici la liste des tests que nous pourrons écrire (avec leur code http attendu):

- testListOntology (200)
- testListOntologyByUser (200)
- testListOntologyByUserNotExist (404)
- testGetOntology (200)
- testGetOntologyNotExist (404)
- testAddOntology (200)
- testAddOnrtologyWithNameAlreadyExist (409)
- testUpdateOntology (200)
- testUpdateOntologyNotExist (404)
- testDeleteOntology (200)
- testDeleteOntologyNotExist (404)

De nombreux autres tests pourraient être écrits.

```
class OntologyTests  {

        void testListOntology() {

        }

        void testListOntologyByUser() {

        }

        ...
}
```

Création de la classe API

Pour simplifier l'utilisation des tests, chaque type ressource dispose d'une classe effectuant les requêtes et fournissant le code et la réponse.
Elle doit hériter de la classe DomainAPI qui lui fournira de nombreuses méthodes utiles.
Les principales méthodes sont doGET/DELETE(url, username, password) et doPOST/PUT(url, data,username, password) qui effectue l'action définie dans le nom de la méthode sur l'URL, avec éventuellement des données en paramètre, en étant authentifier avec l'utilisateur username.
Chacune de ces méthodes renvoie [data: response, code: code] où response est la String de réponse (normalement un JSON) et code est le code HTTP.

```
class ResourceAPI extends DomainAPI {

    static def show(Long id, String username, String password) {
        String URL = Infos.CYTOMINEURL + "api/resource/" + id + ".json"
        return doGET(URL, username, password)
    }

    static def create(String json, String username, String password) {
        String URL = Infos.CYTOMINEURL + "api/resource.json"
        def result = doPOST(URL,json,username,password)
        result.data = Resource.get(JSON.parse(result.data)?.resource?.id) //read the created object from DB
        return result
    }

        ...
}
```

Pour notre exemple, nous définirons show, list, listByUser, create, update et delete.

```
class OntologyAPI extends DomainAPI {

    static def show(Long id, String username, String password) {
        String URL = Infos.CYTOMINEURL + "api/ontology/$id.json"
        return doGET(URL, username, password)
    }

    static def list(String username, String password) {
        String URL = Infos.CYTOMINEURL + "api/ontology.json"
        return doGET(URL, username, password)
    }

    static def listByUser(Long id, String username, String password) {
        String URL = Infos.CYTOMINEURL + "api/user/$id/ontology.json"
        return doGET(URL, username, password)
    }

    static def create(String json, String username, String password) {
        String URL = Infos.CYTOMINEURL + "api/ontology.json"
        def result = doPOST(URL,json,username,password)
        result.data = Ontology.get(JSON.parse(result.data)?.ontology?.id)
        return result
    }
    static def update(def id, def jsonOntology, String username, String password) {
        String URL = Infos.CYTOMINEURL + "api/ontology/$id.json"
        return doPUT(URL,jsonOntology,username,password)
    }
    static def delete(def id, String username, String password) {
        String URL = Infos.CYTOMINEURL + "api/ontology/$id.json"
        return doDELETE(URL,username,password)
    }

}
```

Ajout des méthodes d'initialisation dans BasicInstanceBuilder

Afin d'alléger le code dans les tests, la classe BasicInstanceBuilder dispose de nombreuses méthodes pour initialiser de nouveau domaine.
Pour chaque type de ressource, nous avons:

- static *Resource getResource()*: récupère une ressource (si elle n'existe pas, la créer).
- static *Resource getResourceNotExist(boolean save = false)*: récupère une nouvelle ressource qui n'existe pas encore, si on passe "true" en argument, sauvegarder la nouvelle instance.

La première méthode permet de récupérer rapidement une instance de *Resource* sans la créer.
La deuxième méthode est très utile pour tester l'ajout puisque la resource ne doit pas encore exister en base de données (si save = false).

Nous définirons donc:

```
    static Ontology getOntology() {
        def ontology = Ontology.findByName("BasicOntology")
        if (!ontology) {
                        //create if not exist
            ontology = new Ontology(name: "BasicOntology", user: User.findByUsername(Infos.SUPERADMINLOGIN))
            saveDomain(ontology)
        }
        ontology
    }

    static Ontology getOntologyNotExist(boolean save = false) {
        ontology = new Ontology(name: getRandomString(), user: User.findByUsername(Infos.SUPERADMINLOGIN))
        save ? saveDomain(ontology) : checkDomain(ontology)
    }
```

- *checkDomain* fera échouer le test si des contraintes *Grails* n'auront pas été respectées (dépendance null, String vide,...)
- *saveDomain* effectuera un *checkDomain* et sauvegardera en plus le domaine (en cas d'échec, le test échouera).

Ecriture des méthodes de tests

- Nous parcourons ici uniquement les tests "simples", d'autres types de tests existent: tests de sécurité, tests de dépendance,...
- La attributs statiques ANOTHERLOGIN et ANOTHERPASSWORD de la classe Infos fournissent les login/password HTTP pour les tests.
- En cas de réponse 500, il s'agit souvent d'un bug côté serveur à corriger.

List

Le test de l'action List va effectuer un list grâce à OntologyAPI.
Elle va vérifier que le code de retour est bien 200 et que le résultat est bien un JSON de type JSONArray

```
void testListOntology() {
        def result = OntologyAPI.list(Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
    assert 200 == result.code
    def json = JSON.parse(result.data)
    assert json instanceof JSONArray
}
```

Remarque: Nous pourrions créer une ontology via la méthode BasicInstanceBuilder.getOntology() et vérifier si elle est bien présente dans le JSON de réponse.

Show

Le principe est très proche de List. Dans ce cas, nous créons une nouvelle ontologie et tentons de la récupérer via l'appel HTTP au serveur.

```
void testShowOntologyWithCredential() {
          Ontology ontology = BasicInstanceBuilder.getOntology()
          def result = OntologyAPI.show(ontology.id, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
      assert 200 == result.code
      def json = JSON.parse(result.data)
      assert json instanceof JSONObject
}
```

Add

Nous tenterons d'ajouter l'ontologie définie par getBasicOntologyNotExist(). Nous fournissons toutes les informations en JSON.
Après avoir vérifier que l'ajout s'est bien passé (200), nous vérifions si nous pouvons récupérer l'ontologie via un show.

```
    void testAddOntologyCorrect() {
        def ontologyToAdd = BasicInstanceBuilder.getOntologyNotExist()
        def result = OntologyAPI.create(ontologyToAdd.encodeAsJSON(), Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 200 == result.code
        int idOntology = result.data.id

        result = OntologyAPI.show(idOntology, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 200 == result.code
    }
```

Cytomine disposant d'un système d'undo/redo, il pourrait être intéressant de tester l'annulation et la récupération d'un ajout.

- l'ajout doit être fait avec succès (add = 200), l'ontologie doit exister (show = 200)
- l'undo doit être fait avec succès (undo = 200), l'ontologie doit être supprimée (show = 404)
- le redo doit être fait avec succès (redo = 200), l'ontologie doit réapparaître (show = 200)

```
    void testAddOntologyCorrect() {
        def ontologyToAdd = BasicInstanceBuilder.getOntologyNotExist()
        def result = OntologyAPI.create(ontologyToAdd.encodeAsJSON(), Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 200 == result.code
        int idOntology = result.data.id

        result = OntologyAPI.show(idOntology, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 200 == result.code

        result = OntologyAPI.undo()
        assert 200 == result.code

        result = OntologyAPI.show(idOntology, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 404 == result.code

        result = OntologyAPI.redo()
        assert 200 == result.code

        result = OntologyAPI.show(idOntology, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 200 == result.code
    }
```

Update

Le principe est fort similaire que Add.
Néanmoins, nous devons vérifier si les champs ont bien été modifiés et si l'undo/redo annule ou restaure bien la valeur des champs.

La classe *UpdateData* dispose de méthode générique *createUpdateSet(Resource* resource,changeList).
Le paramètre *changeList* doit être une liste d'ancienne et nouvelle valeur: *[prop1: ["oldValue","newValue"], prop2: ...]*

La méthode modifiera les propriétés en ajoutant les anciennes valeurs à la ressource, créera un JSON de la ressource avec les nouvelles valeurs et renverra une map avec 3 infos:

- postData: ressource telle qu'elle doit apparaître après modification,
- mapOld: ancienne valeur des champs à modifier,
- mapNew: nouvelle valeur des champs modifiés.

Pour Ontologie, nous pouvons utiliser:

```
 def data = UpdateData.createUpdateSet(ontology,[name: ["OLDNAME","NEWNAME"], user: [BasicInstanceBuilder.user1,
BasicInstanceBuilder.user2]])
```

La méthode appliquera à ontology le nom "OLDNAME" et l'utilisateur 1 et elle générera un JSON où le nom sera "NEWNAME" et l'utilisateur 2. Nous utiliserons ce JSON comme paramètre de requête.

Nous aurons donc les étapes suivantes:

1. Création de l'ontologie,
2. Appel à *createUpdateSet*,
3. update de l'ontologie *ontology.id* avec les infos *data.postData*,
4. Vérification que les champs ont changés (*BasicInstanceBuilder.compare(data.mapNew, json)*)
5. Annulation
6. Vérification que les champs ont les anciennes valeurs (*BasicInstanceBuilder.compare(data.mapOld, json)*)
7. Restauration
8. Vérification que les champs ont changés (*BasicInstanceBuilder.compare(data.mapNew, json)*)

```
    void testUpdateOntologyCorrect() {

        Ontology ontologyToAdd = BasicInstanceBuilder.getOntology()
         def data = UpdateData.createUpdateSet(ontology,[name: ["OLDNAME","NEWNAME"], user:
[BasicInstanceBuilder.user1,BasicInstanceBuilder.user2]])
        def result = OntologyAPI.update(ontologyToAdd.id, data.postData,Infos.ANOTHERLOGIN, Infos.
ANOTHERPASSWORD)
        assert 200 == result.code
        def json = JSON.parse(result.data)
        assert json instanceof JSONObject
        int idOntology = json.ontology.id

        def showResult = OntologyAPI.show(idOntology, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        json = JSON.parse(showResult.data)
        BasicInstanceBuilder.compareOntology(data.mapNew, json)

        showResult = OntologyAPI.undo()
        assert 200 == result.code
        showResult = OntologyAPI.show(idOntology, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        BasicInstanceBuilder.compareOntology(data.mapOld, JSON.parse(showResult.data))

        showResult = OntologyAPI.redo()
        assert 200 == result.code
        showResult = OntologyAPI.show(idOntology, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        BasicInstanceBuilder.compareOntology(data.mapNew, JSON.parse(showResult.data))

    }
```

Delete

Le delete est fort similaire au Add (mais inversé pour les codes 200 - 404).
Il est préférable d'essayer de supprimer une ressource qui vient d'être créer (utiliser getBasicXXXNotExist au lieu de createOrGetBasicXXX) car d'autres tests auraient pu lier des données à l'instance de createOrGetBasicXXX (rendant sa suppression impossible).
Avec getBasicXXXNotExist() suivi d'un save, la ressource existera en base de données.

```
    void testDeleteOntology() {
        def ontologyToDelete = BasicInstanceBuilder.getBasicOntologyNotExist()
        assert ontologyToDelete.save(flush: true)!= null
        def id = ontologyToDelete.id
        def result = OntologyAPI.delete(id, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 200 == result.code

        def showResult = OntologyAPI.show(id, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 404 == showResult.code

        result = OntologyAPI.undo()
        assert 200 == result.code

        result = OntologyAPI.show(id, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 200 == result.code

        result = OntologyAPI.redo()
        assert 200 == result.code

        result = OntologyAPI.show(id, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 404 == result.code
    }

    void testDeleteOntologyNotExist() {
        def result = OntologyAPI.delete(-99, Infos.ANOTHERLOGIN, Infos.ANOTHERPASSWORD)
        assert 404 == result.code
    }
```