**DMA**

**Sommaire :**

1. [**Zend Framework**](#ZF)
2. [**Arborescence du framework**](#Arbo)
   1. **Application**
   2. **Public**
   3. **Library**
   4. **Resources**
3. [**MVC avec Zend**](#MVCZ)
4. [**Modèles**](#modele)
5. [**Contrôleurs**](#controller)
6. [**Vues**](#vue)
7. [**Formulaires**](#formulaires)
8. [**Layout**](#layout)
9. [**Urls**](#urls)
10. [**L’appli**](#appli)
11. [**Le template**](#template)
12. [**La page d’accueil, palmarès**](#palmares)
13. [**Palmarès pour une période**](#palmaresPeriode)
14. [**Informations pour un site**](#infosSite)
15. [**Importer .csv**](#CSV)
16. [**Graphiques**](#graphiques)
17. [**Ajouter un site**](#ajouterSite)
18. [**Admin**](#admin)
19. [**GetInfos()**](#getInfos)
20. [**Datatable**](#datatable)

**ZEND FRAMEWORK**

* **Arborescence**

A la racine on a 4 dossiers : application, public, library, resources

* Application

C’est dans ce dossier que sera mis tout le code métier. L’utilisateur n’a pas accès à ce code On y trouve 6 dossiers et le fichier Bootstrap.php.

Dossiers :

**Configs** : répertorie les fichiers de configuration de l’application. On y trouve seulement application.ini. On y règle l’affichage des erreurs ou non, les dossiers et fichiers à inclure suivant l’environnement.

**Controllers :** C’est ici que sont placés les contrôleurs.

Ils DOIVENT commencer par une Majuscule et terminer par « Controller » !

La classe du contrôleur porte exactement le même nom que le nom du fichier dans lequel elle se trouve.

Un contrôleur étend Zend\_Controller\_Action.

Il regroupe les actions possibles.

Une action Doit se terminer par « Action » ! ex : exempleAction().

Ex : le contrôleur ExempleController.php contient la classe :

Class ExempleController extends Zend\_Controller\_Action { exempleAction(){} autreAction(){} … }

**Forms :** On y place les classes de formulaires.

La classe porte le même nom que le fichier et étend Zend\_Form.

Ex : exFormulaire.php contient la classe : class exFormulaire extends Zend\_Form {}

**Layouts :** Le layout est le template appelé à chaque page.

On y insère le doctype, les balises que l’on doit avoir partout (html, head, body), les fichiers css et javascript et le code html que l’on veut toujours avoir (j’ai mis le menu de gauche).

<?php echo $this->layout()->header; ?> : affiche le [header](#layout), on reverra ca plus bas

On a 2 layout. Layout.phtml est celui toujours appelé, vide.phtml me sers pour les appels en ajax où je n’ai pas besoin de récupérer tous le code html d’une page mais seulement ce qui m’est retourné par l’appel.

**Models :** On y trouve les fichiers qui vont interroger la base de données.

La classe doit elle aussi porter le même nom que son fichier, et doit étendre de Zend\_Db\_Table\_Abstract.

On doit, dans la classe declarer deux variables en protected :

$\_name => doit contenir le nom de la table a laquelle cette classe fait reference (cette variable est obligatoire si le nom de la table est différent de celui de la classe)

$\_primary => la clé primaire dans cette table

**Views :** Le dossier dans lequel sont stockées les vues (plus précisément dans views/scripts)

Pour chaque controller, il doit y avoir un dossier portant le nom du controller en minuscule et sans la fin (« Controller »).

Ex : pour le controller ExempleController.php, il doit y avoir un dossier : exemple dans views/scripts.

Dans chaque dossier de vue, on a une vue par action du controller au format .phtml

Ex : ExempleController contient indexAction et exempleAction , alors dans le dossier scripts/exemple il doit y avoir index.phtml et exemple.phtml.

**Le fichier BootStrap.php :** Il est appelé par index.php et se charge de la connexion à la base de données, de définir les constantes, les layouts…

J’y ai réglé la configuration de l’application, de la base de données et de sa connexion, le chargement des classes externes (modèles, forms…), et les fichiers de traduction, mais on peut ajouter d’autres méthodes.

* Public

Le contenu de ce dossier est accessible par tout le monde.

On trouve 4 dossiers :

**CSS :** On y place tous nos fichiers css

**JS :** On y place les fichiers javascripts

**IMG :** On y place les images

**HTML2PDF :** C’est une libraire externe que j’ai ajouté. Elle n’est pas obligatoire au fonctionnement du framework. Elle sert à la génération de pdf.

Et 3 fichiers :

**.htaccess :** Par defaut, une url en zend framework est du style

Domaine.fr/index.php/controller/action

Ce fichier permet de retirer la chaine ‘index.php’ pour une meilleur lisibilité de l’url et le bon fonctionnement avec le bootstrap.

**index.php :** on déclare des constantes qui seront accessibles partout dans l’application.

On créé l’application, le bootstrap et on l’exécute.

**loader.txt :** ce fichier contient une requete SQL permettant de sortir les données de la table T\_DATA\_COLLECTE et de les remplir la table T\_COLLECTE à partir de ces données.

* Library

Ce dossier contient le framework Zend, et si besoin des librairies que l’on surcharge.

* Resources

Dans ce dossier, on a le dossier archives, qui regroupe tous les fichiers csv qui ont étés importés depuis l’application.

Le dossier languages contient un pack de traduction, la langue utilisée est paramétrée dans le Bootstrap.

Le dossier temp, sert juste a copier un fichier csv prendre les données a partir de ce fichier, puis supprimer le fichier. Il est donc normalement vide.

* **MVC avec Zend**

Le MVC permet de séparer le code métiers (controllers), les accès en base de données (models) et l’affichage (views).

Chaque action de controller a une vue (du même nom) qui lui est associée.

Pour récupérer des données de la base :

Dans le Controller, on instancie un objet de la classe de la table qui nous intéresse.

On fait appel à une méthode de cet objet.

Dans le modèle, on a donc la méthode qui fait récupère ce qu’on souhaite et qui retourne son résultat dans un tableau (il existe plusieurs façons de récupérer le résultat d’une requête, [on les verra plus bas](#recuperation))

Dans le Controller on a donc une variable qui a le résultat de la requête. Il reste à passer cette variable à la vue. Cela se fait de cette manière :

Coté controller :

$this->view->nomVariable = $lavariable ; // on envoi la variable

Coté vue :

Echo $this->nomVariable ; // on affiche la variable

* **Modèle**

Requeter la base de données se fait de cette manière :

* **Select**

Protected $\_name = ‘table1’ ;

Public function getContenuTable($id) {

$sql = $this->select()

->distinct()

->setIntegrityCheck(false)

->from(array(‘tb1’=>$this->\_name, ’nom’)

->join(array(‘tb2’=>’autreTable’, ‘tb1.id=tb2.id’, ‘\*’)

->where(‘tb1.id = ?’, $id)

->group(‘colonnesAGrouper’)

->order(‘tb1.id ASC’);

$res = $this->fetchAll($sql)->toArray() ;

return $res ;

}

**Explication du code :**

$this->\_name appelle la variable protected $\_name de la classe, qui contient le nom de la table.

Ce code créera la requête sql suivante :

SELECT DISTINCT tb1.nom, tb2.\*

FROM table1 tb1

INNER JOIN autreTable tb2 ON tb1.id=tb2.id

WHERE (tb1.id = \*$id\*)

GROUP BY (\*colonnesAGrouper\*)

ORDER BY (tb1.id ASC) ;

**->distinct()** : ajoute le DISTINCT au select

**->setIntegrityCheck(false)** : cette ligne est obligatoire pour pouvoir faire une requête sur plusieurs tables (si on veut faire un join).

**->from()** : en premier paramètre, on met le nom de la table (si on veut lui ajouter un alias, on met un tableau avec l’alias en clé, et le nom de la table en valeur), en deuxième paramètre, les champs à sortir de cette table.

**->join()** : premier parametre = nom de la table, deuxieme parametre = condition de jointure, troisieme parametre = champs a sortir de cette deuxieme table.

Il existe aussi les methodes joinLeft(), joinRight(), joinFull()…

**->where()** : le nom du champ sur lequel faire la restriction. Le ‘ ? ‘ prend la valeur du deuxième paramètre (souvent une variable). Si ce paramètre est un tableau, il faut alors mettre le ‘ ? ‘ entre parenthèse : ‘(?)’ (et la condition ne sera plus = mais IN).

Pour faire un AND, on ajoute des lignes ->where(condition2)->where(condition3)

Pour faire un OR, on ajoute une ligne ->orWhere(condition2)->orWhere(condition3)

**->group()** : le nom des champs sur lesquels on veux grouper

**->order()** : le nom des champs sur lesquels on veux trié avec le sens du tri ‘ASC’ ou ‘DESC’

La ***récupération*** des lignes ressorties par la requête se fait de différentes manières suivant le nombre de lignes et de champs retournées par la requête :

* Si la requête renvoi plusieurs lignes ayant plusieurs champs (ex 3 lignes avec des champs nom, prenom, age…). La récupération des lignes se fait grâce la méthode fetchAll :

$res = $this->fetchAll($sql)->toArray() ;

(toArray(), permet de n’avoir que le résultat de la requête, autrement, $res contient des données qui nous sont inutiles)

* Si la requête retourne une ligne de plusieurs champs : on utilise fetchRow()

$res = $this->fetchRow($sql) ;

* Si la requête retourne plusieurs lignes, avec un champs par ligne (ex SELECT id FROM table). On utilise la méthode fetchCol :

$res = $this->\_db->fetchCol($sql)

* Si la requête retourne une colonne d’une ligne (ex SELECT nom FROM table WHERE id = 1)

On utilise la méthode fetchOne :

$res = $this->\_db->fetchOne($sql) ;

Attention : pour fetchCol() et fetchOne(), on doit ajouter ->\_db pour pouvoir récupérer ces méthodes

Pour voir la requête SQL générée, on peut faire echo $sql->assemble() ;

* **Insert**

Public function insertion($donneesAInseree, $id) {

$where = $this->getAdapter()->quoteInto(‘id = ?’, $id) ;

$this->insert($donnees,$where) ;

}

$donnees doit etre un tableau ayant en clé : le nom du champs dans la table, et en valeur : la valeur de ce champs à insérer.

Ex : $donnees = array(‘nom’=>’toto’,’prenom’ => ‘titi’, ’age’ => 45) ;

Avec ce tableau la requête générées sera :

INSERT INTO \*tableDansLaquelleOnEst\* (‘nom’,’prenom’,’age’) VALUES (‘toto’,’titi’,45) ;

La fonction quoteInto de $where permet d’échapper la valeur.

On peut récupérer le nombre de lignes insérées en ajoutant return devant la requête :

Return $this->insert(…) ;

* **Update**

Il a exactement le même fonctionnement que l’insert sauf que l’on fera :

$this->update($donnees,$where) ;

On peut récupérer le nombre de lignes modifiées en ajoutant return devant la requête :

Return $this->update (…) ;

* **Delete**

Public function supprime($id) {

$where = $this->getAdapter()->quoteInto(‘id = ?’, $id) ;

$this->delete($where) ;

}

Presque le même fonctionnement que l’insert ou l’update sauf que l’on a seulement le where.

La requête sera : DELETE FROM \*tableDansLaquelleOnEst\* WHERE id = \*$id\* ;

On peut récupérer le nombre de lignes supprimées en ajoutant return devant la requête :

Return $this->delete(…) ;

* **CONTROLLER**

Dans chaque action (ou presque) d’un controller, j’ai mis la ligne :

$this->\_helper->actionStack('header', 'index', 'default', array());

Cette ligne permet d’empiler les actions et donc leurs codes et leurs vues.

Ici j’appelle l’action header du controller Index.

Le troisième paramètre est le module (je n’y ai pas touché).

Enfin on peut passer des variables a l’action appelée dans le dernier paramètre.

Si on ne veut pas passer de variable on peut écrire :

$this->\_helper->actionStack('header', 'index');

Pour passer des variables a la vue :

$this->view->nomDeLavariableDansLaVue = $nomDeLaVariableDansLeController ;

Pour récupérer une variable passée dans une Url, un formulaire ou un actionStack :

$variable = $this->getRequest()->getParam(‘nomVariableDansUrl’, ‘valeurParDefaut’) ;

Si aucune variable n’est passées dans l’url, $variable prendra la valeur ‘valeurParDefaut’.

* **VUE**

C’est le code affiché a l’écran.

Afficher une variable reçue d’un controller :

Echo $this->nomDeLaVariableDansLaVue ;

* **Formulaires**

$this->setMethod(‘post’) // indique que les données seront envoyées en post

->setAction(‘url’) // indique l’action

->setName(‘nom’) // indique l’attribut name

->setAttrib(‘nomAttrib’,’valeurAttrib’) // peut ajouter une classe par exemple

Créer un nouvel élément :

$element = new Zend\_Form\_Element\_X(‘nomElement’) ;

X représente le type de l’élément soit :

Text, Password, File, Checkbox, MultiCheckbox, Radio, Select, MultiSelect, etc…

Chaque élément à ses méthodes. Mais il existe des méthodes communes à tous :

$element->setLabel(‘label’)

->setAttrib(‘nomAttrib’,’valeurAttrib’)

->setRequired()

Etc…

Une fois l’élément créé, il faut l’ajouter au formulaire :

$this->addElement($element) ;

Ou ajouter tous les éléments en même temps à la fin :

$this->addElements(array($element1, $element2, $element3)) ;

On peut, imposer des règles à chaque élément, permettant de ne pas valider le formulaire si ces règles ne sont pas respectées.

Ex sur un champ text :

$elementText->addValidators(array(‘alnum’,’Digits’, ‘Float’…)) ;

Pour valider un formulaire, on va dans l’action qui est spécifiée dans son attibut action=’’ ‘’,

On créé un nouvel objet qui instancie ce formulaire : $form = new monForm() ;

Et on test si il est valide avec la methode isValid de $form :

If ($form->isValid($\_POST))

{ // formulaire valide }

S’il n’est pas valide, les erreurs seront affichées en dessous des champs non valides.

* **LAYOUT**

Dans le dossier layouts, on trouve un fichier nommé layout.phtml.

Ce fichier est appelé par toutes les vues (sauf quand on indique de changer de layout).

Il contient donc le code html qui sera à réutiliser partout, et des lignes en php telles que :

<?php echo $this->layout()->header; ?>

// ici est placé le code Html du menu, il sera comme le reste, répété partout

<?php echo $this->layout()->content; ?>

<?php echo $this->layout()->footer; ?

Ces lignes appellent la vue header.phtml, footer.phtml.

Content est le contenu de la vue dans laquelle on est.

Si on se trouve dans la vue index.phtml, par exemple.

Alors tout le code affiché à l’écran sera :

le début du layout, la vue header.phtml, le menu, la vue index.phtml (content prend cette vue comme valeur), la vue footer.phtml, et enfin la fin du layout.

* **Les URLs**

Avec l’url, on peut récupérer plusieurs informations pour savoir dans quel fichier se trouve le code que l’on exécute.

Une url est du type : http://domaine.fr/controller/actionducontroller/param1/valeurparam1/param2/valeurparam2

Avec de vraies valeurs :

http://domaine.fr/exemple/index/id/10/nom/toto

Avec cette url, on sait donc que l’on se trouve dans le controller exempleController, et plus précisément dans son action indexAction.

On voit aussi que l’on passe 2 variables dans l’url (id et nom).

Pour les récupérer, dans l’action on fait :

$id = $this->getRequest()->getParam(‘id’) // prendra 10

$nom = $this->getRequest()->getParam(‘nom’) // prendra toto

**L’appli**

* Le style de la page :

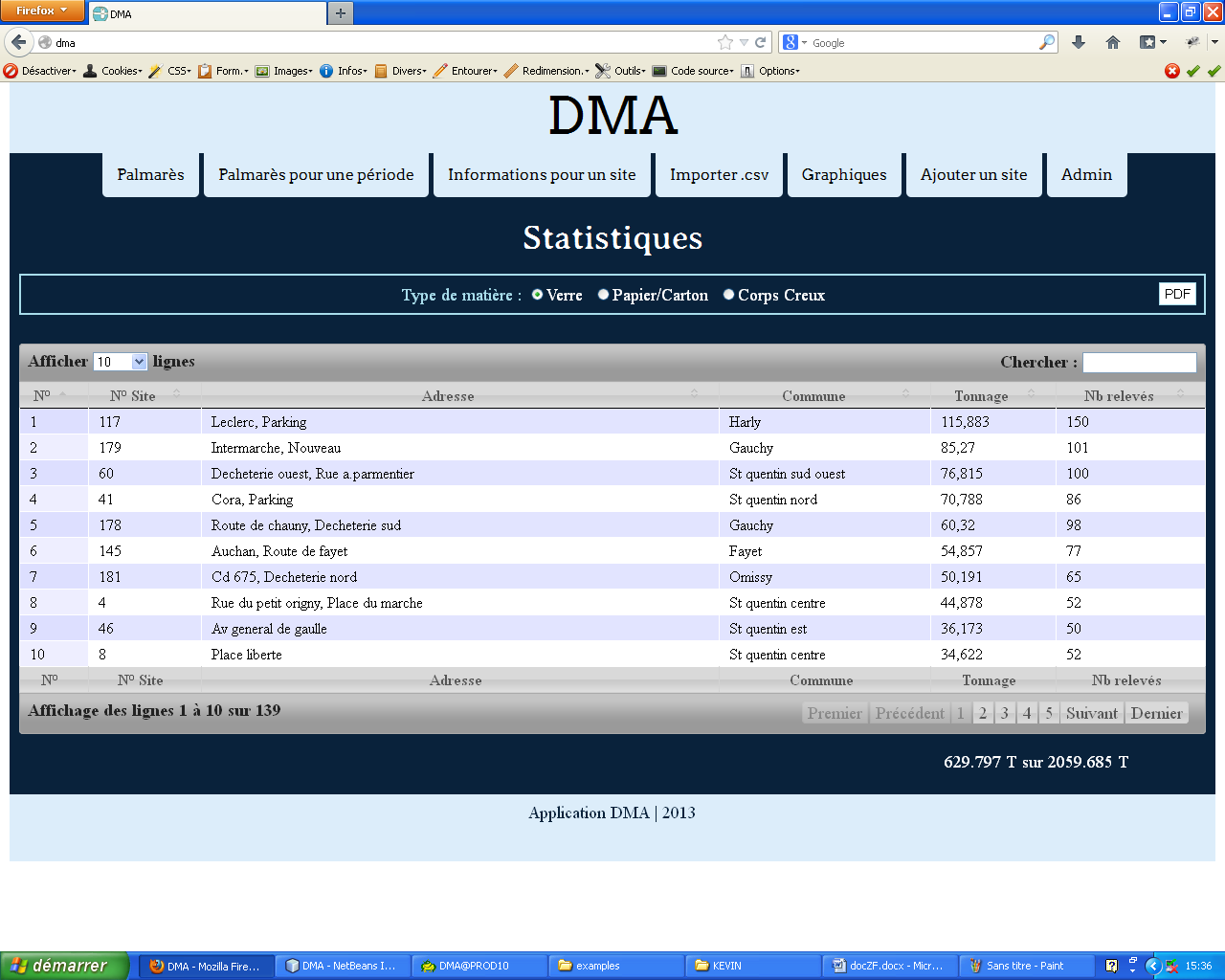
Le code qui doit être affiché partout tel que le header, le footer, le menu et des balises html pour la mise en forme se trouve dans le fichier de [layout](#layout).

Dans toutes les action ou presque, on retrouve en premiere ligne :

$this->\_helper->actionStack('header', 'index', 'default', array());

Cette ligne permet de récupérer le header, qui s’affichera et qui appellera le footer. Pour récupérer le Template de l’application on met donc cette ligne.

* La page d’accueil, palmarès :



Cette page récupère tous les sites en bdd, leurs infos, calcule leur tonnage et compte le nombre de levés.

Le controller appelé est PalmaresController et l’action est indexAction (pour cette page cela à été réglé dans le fichier de configuration application.ini)

L’action index a pour seule fonction de créer un formulaire, de l’afficher et d’appeler à son tour l’action infosAction (sans passer de paramètres).

InfosAction lui, recupere la matiere cochée dans la checkbox, s’il y en a pas il prend ‘Verre Couleur’ par défaut. Il vérifie que l’on vient d’une requête ajax ou non et si l’on veut sortir les données en pdf.

Puis, il récupère les données en base en faisant appel au modele TSITE qui fait référence à la table T\_SITE en bdd. Il appelle sa méthode [getInfos()](#getInfos) que nous verrons plus tard.

Il passe toutes les variables à la vue pour l’affichage.

Puisque l’action index appelle une autre action avec un ‘actionStack’, ce sont les vues de ces 2 actions qui seront affichées à la suite.

La vue index affiche juste un titre et le formulaire contenant les checkbox.

La vue infos affiche le tableau rempli.

C’est dans cette vue que le PDF (s’il est demandé) sera créé.

Le contenu du pdf est le code HTML situé entre les fonction php : ob\_start() et ob\_get\_clean().

(les balises <page>, <page\_header> et <page\_footer> sont spécifiques à l’API utilisé pour générer le pdf, HTML2PDF).

Si on ne veut pas de pdf mais un affichage normal de la vue, on commence d’abord par convertir le nom de la matiere en entier (pour la passer dans l’url d’un formulaire (les espaces auraient gênés)). On créé ce formulaire qui contient un bouton, celui qu’il faudra cliquer pour générer le pdf et un champs hidden pour savoir si ce bouton à été cliqué ou non.

Ensuite, le tableau est créé en parcourant toutes les lignes récupérées en bdd et affiché.

On en profite pour compter le tonnage total, qui sera affiché en dessous.

On affiche le tonnage affiché à l’écran (on ne voit pas toutes les lignes de la table, il faut parcourir plusieurs pages) et le tonnage total.

Le tonnage affiché se fait grâce au javascript.

Le code javascript en fin de page permet d’appeler une api permettant de mettre en forme une table, [l’api dataTables.](#datatable)

J’appelle une fonction que j’ai créé pour ajouter un paramètre. Ce paramètre sert à identifier le type de la 5eme colonne de la table ("sType": "numeric-comma" => type flottant).

J’ai du ajouter ce paramètre car dans la partie [admin](#admin) que l’on verra plus tard, la 5eme colonne contient des checkbox et pas des float.

En javascript :

Au changement de matière dans le formulaire, on récupère le nom de cette matière et on rappelle en ajax l’action infosAction (qui retournera chercher les infos sur les sites de la matiere nouvellement cochée).

(la fonction appelée au changement de matière est l’évenement :

$(‘.rad’).change(function() {

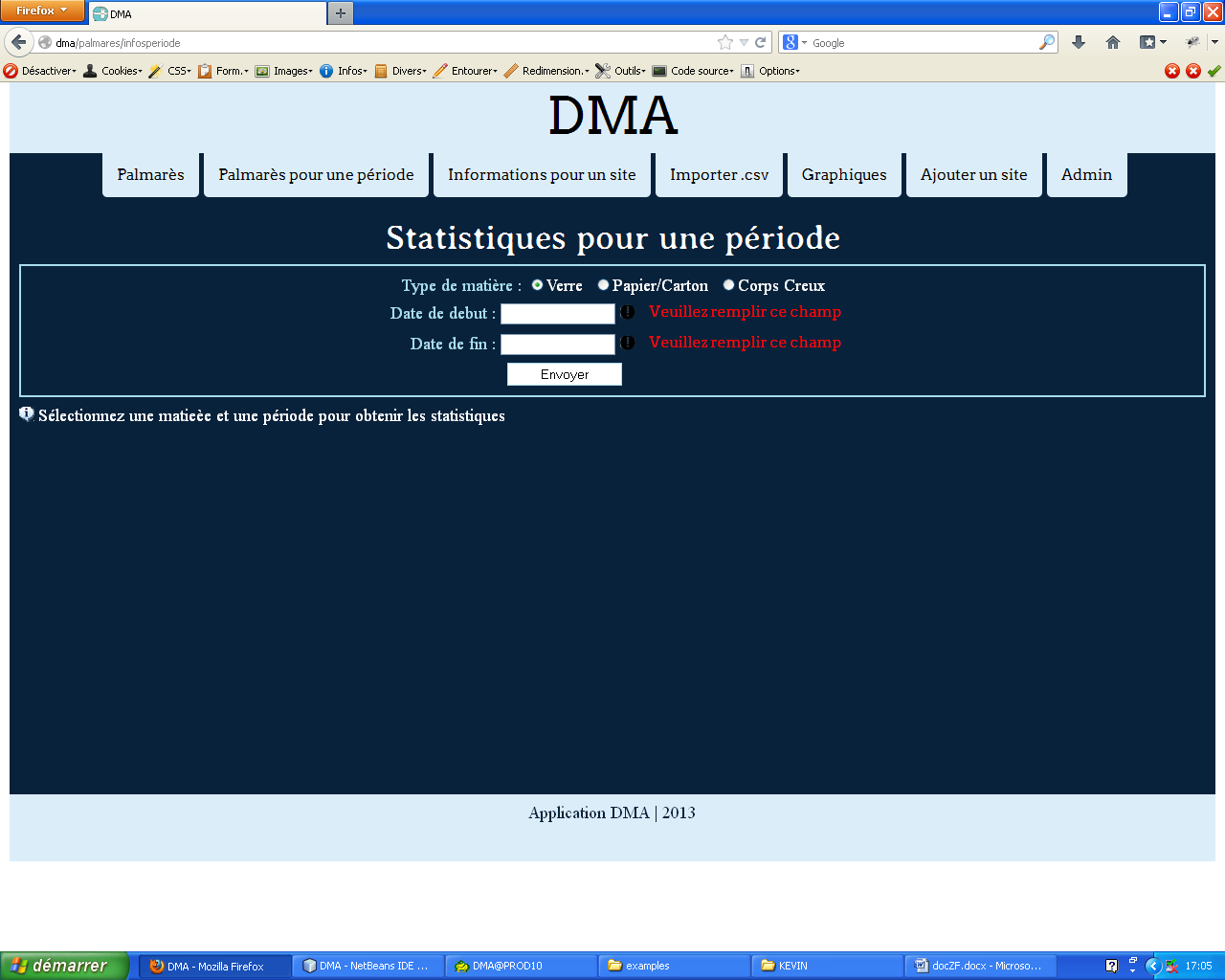
/\* appel ajax, vidage du div et remplissage par le nouveau contenu \*/

});

On vide toute la vue de son code Html, on récupère le code retourné par l’action appelé (ce code est en fait la nouvelle vue à afficher).

Enfin on insert le code dans la page (par l’intermédiaire du div que l’on a précédemment vidé).

* Palmares pour une période :

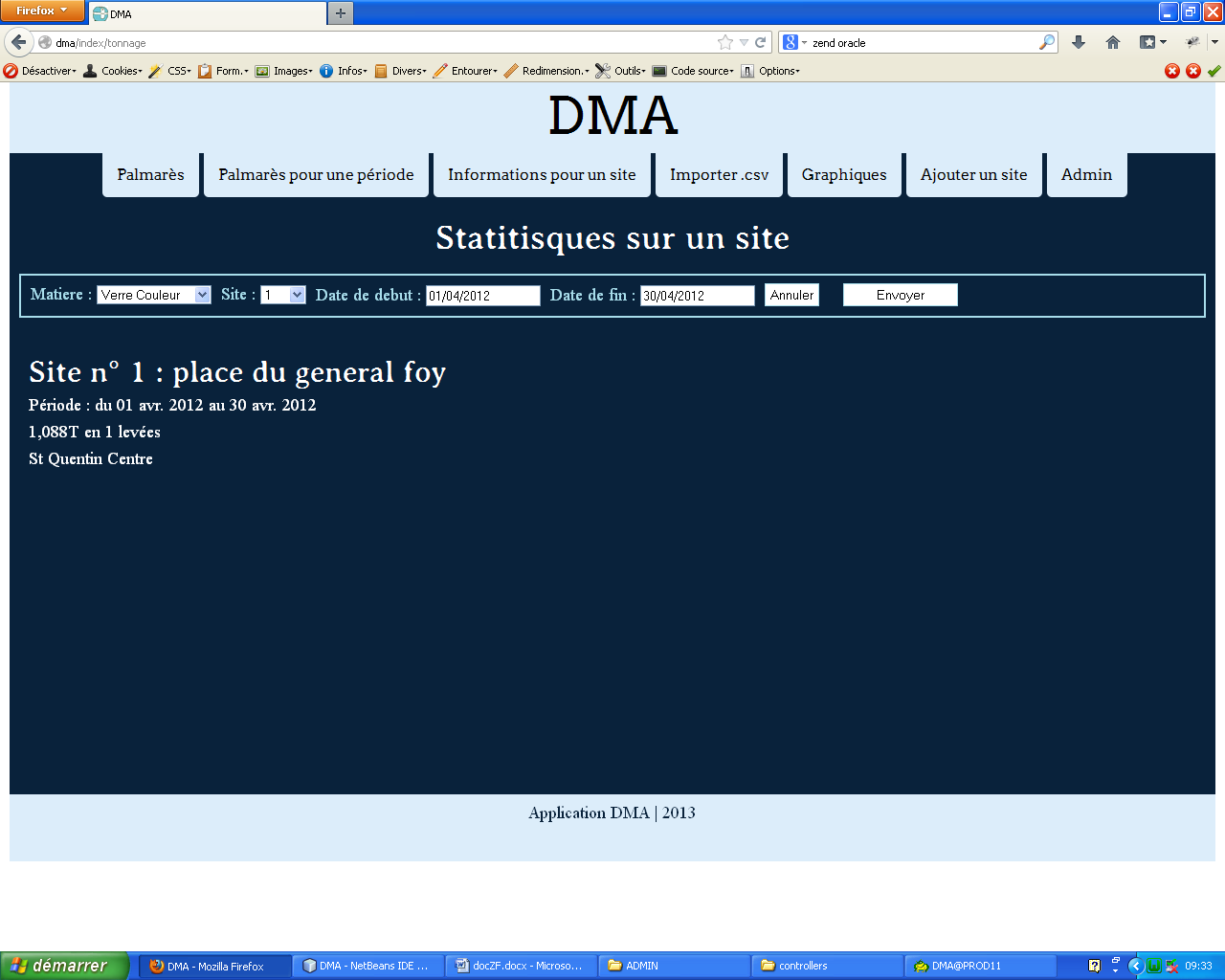


Cette page est appelée par PalmaresController / infosperiodeAction.

Elle a exactement le même but que la page précédente, mais en ajoutant une restriction sur la date.

Elle gère également la validation du formulaire, les dates doivent être spécifie pour exécuter une requête correcte.

* Informations pour un site :



Cette page est appelée par IndexController / tonnageAction.

Cette page à pour but de récupérer des informations sur un site seul. Il est possible de restreindre la durée de recherche en spécifiant une date de début et une date de fin.

Ici on retrouve le même principe que plus haut.

L’action tonnage ne fait que d’afficher une partie de l’html (le titre de la page et la balise ouvrante où sera mis le reste de la page).

Puis elle appelle l’action tonnageajax, dans laquelle tous le reste de la page sera créé.

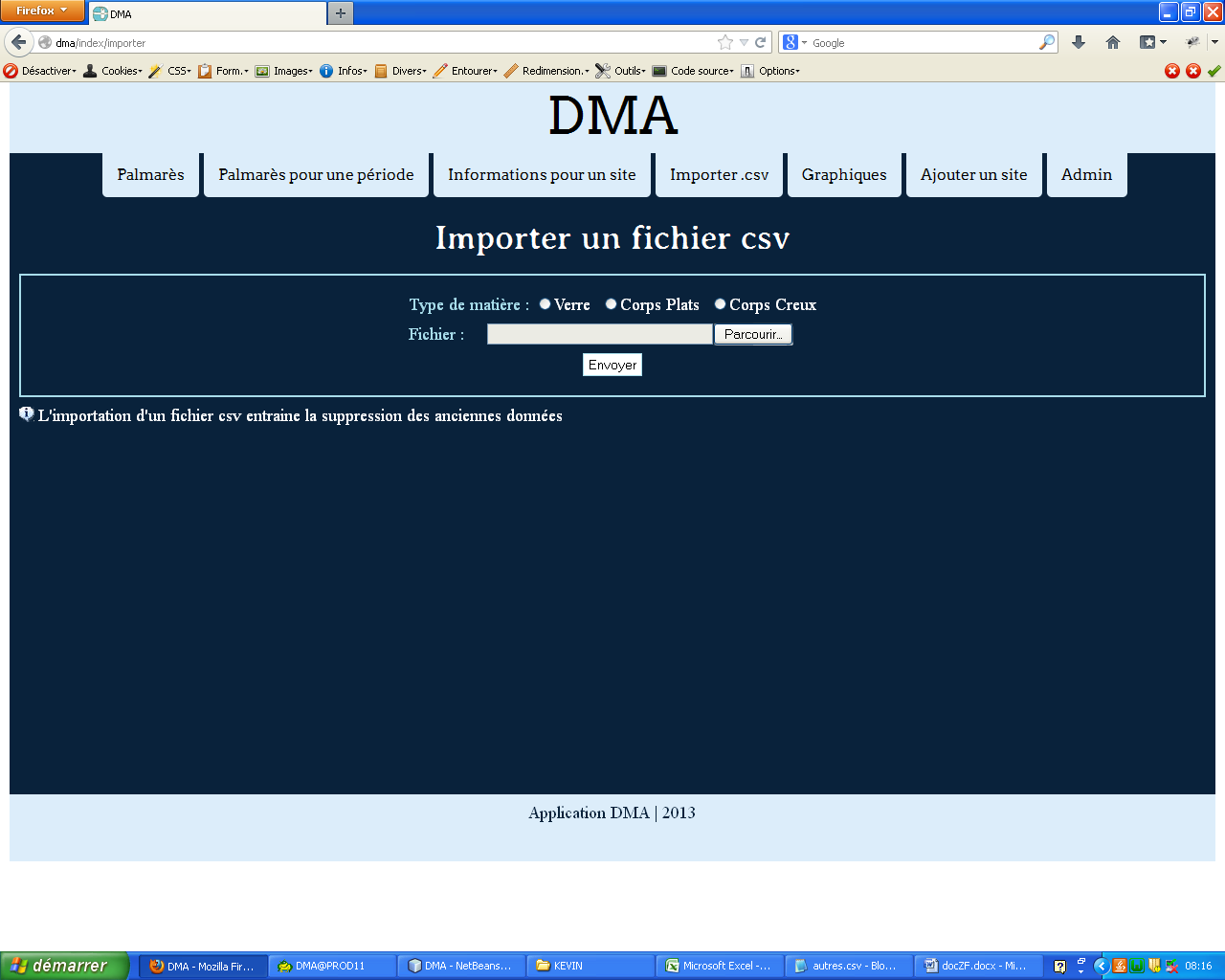
Cette division de page est faite pour pouvoir changer le contenu en ajax. On sert de la balise créée dans la première action pour la vider et remplacer son contenu par l e code de la deuxième action.

Tonnageajax récupère la matière et un indicateur pour savoir si l’on provient d’une requête ajax.

Le formulaire est ensuite créé en passant trois paramètres (le tableau de toutes les matières différentes, récupérées avant, les sites qui ont un conteneur la matière sélectionnée, liste récupérée également plus tôt (par défaut : verre couleur), puis la matière sélectionnée).

Ensuite on vérifie que le formulaire à été validé, si oui on vérifie qu’il est valide, et si oui on récupère les données, on va chercher les infos [( avec la requete getInfos() )](#getInfos) et on les passes à la vue.

* Importer .csv :



La page appelle IndexController / importerAction.

Elle permet d’insérer des données dans la base par l’intermédiaire d’un fichier csv. Ce fichier doit être formaté correctement pour que la requête fonctionne.

Ici, on créé le formulaire, s’il n’à pas été validé, on l’affiche seulement (aucun autre traitement n’est fait).

Il faut obligatoirement saisir une matière avant d’importer le fichier correspondant (c’est pour cette raison qu’aucun bouton radio n’est coché par défaut).

Si le formulaire vient d’être validé, on vérifie qu’il est valide, si c’est le cas, on récupère les informations du fichier, et la matière qui ont été envoyés.

Si l’extension du fichier en bien csv, on copie le fichier dans 2 répertoires :

* Archives : ce dossier stockera tous les fichiers importés (on ajoute ‘\_anneeEnCours’ au nom du fichier (pour ne pas voir de doublons au fil des années))
* Temp : le fichier est mis dans ce dossier le temps de l’insertion en base puis supprimé

Si la copie du fichier dans les 2 dossiers a réussie, on l’ouvre en lecture.

Si l’ouverture a fonctionnée, on vide la table dans la base de données, on le parcourt ligne par ligne.

Pour chaque ligne, on créé un tableau avec en clé, le nom des champs dans la table, et en valeur la valeur du champ à ajouter.

On parcourt ligne par ligne le fichier et on teste la matière que l’on est en train d’insérer.

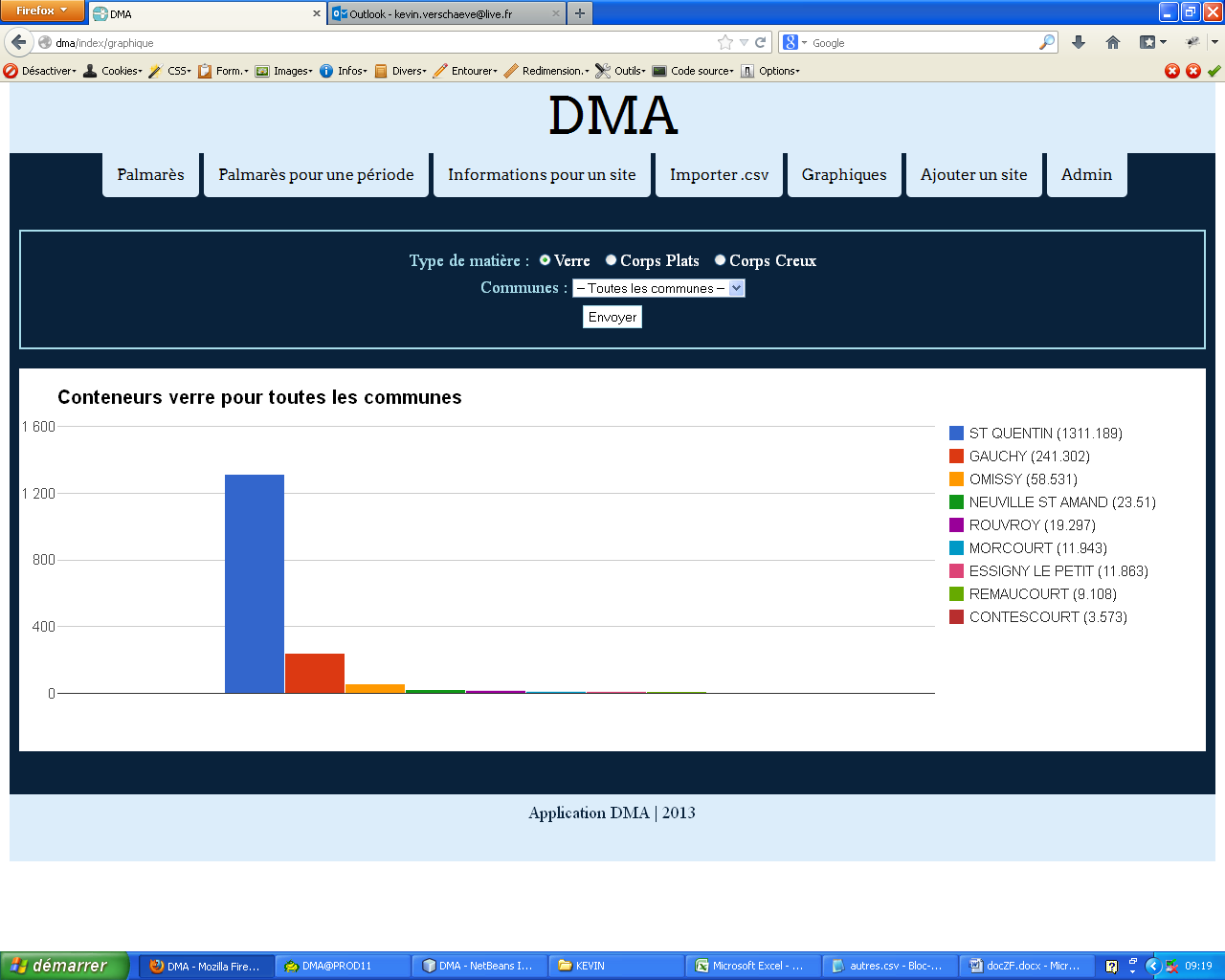
On récupérera et insérera les infos de la ligne en cours différemment suivant la matière Verre

et les autres matières ‘Corps plats’ et ‘Corps creux’.

Une fois les lignes insérées dans la table temporaire T\_DATA\_COLLECTE, il faut redistribuer les données dans les autres tables (la requête pour cela est créée suivant la matière).

Ensuite on exécute un script qui récupère la connexion à la base de données, prépare la requête et l’exécute.

* Graphiques :



Cette page appelle IndexController / graphiqueAction.

Elle permet de voir les tonnages, soit de chaque commune soit de chaque site d’une commune, et par type de conteneur sous forme de graphique.

Ici, on récupère l’id de la commune sélectionnée (on met à false si il n’y en a pas) et la matière du conteneur (‘VERRE’ par defaut).

On va chercher les communes dans la base (et on ajoute à l’indice 0, la valeur ‘Toutes les communes’).

Si l’id de la commune est différent de false (on veut les infos sur une commune particulière), on appelle getTonnagesite(), en passant l’id et la matière. Cette fonction retourne un tableau avec les sites de la commune et le tonnage de chaque site pour la matière choisie.

Si l’id de la commune est a false (on veut une vue d’ensemble sur toutes les communes), on appelle getTonnageCommunes() en passant la matière. Cette fonction retourne le tonnage total (comprenant tous les sites de la commune) par commune pour la matière sélectionnée.

Dans la vue, on appelle l’API google permettant de générer des graphiques.

La fonction drawVisualization(), permet de spécifier le type de graphique (charType), les données à afficher (dataTable), les options du graphique (options) et l’id du div qui contiendra ce graphique une fois créé.

L’attribut dataTable :

C’est un tableau à 2 dimensions de cette forme :

[ [‘’,‘nom1’,’nom2’,’nom3’] , [‘’,tonnage1,tonnage2,tonnage3] ]

nomX = le nom de la commune OU le nom du site

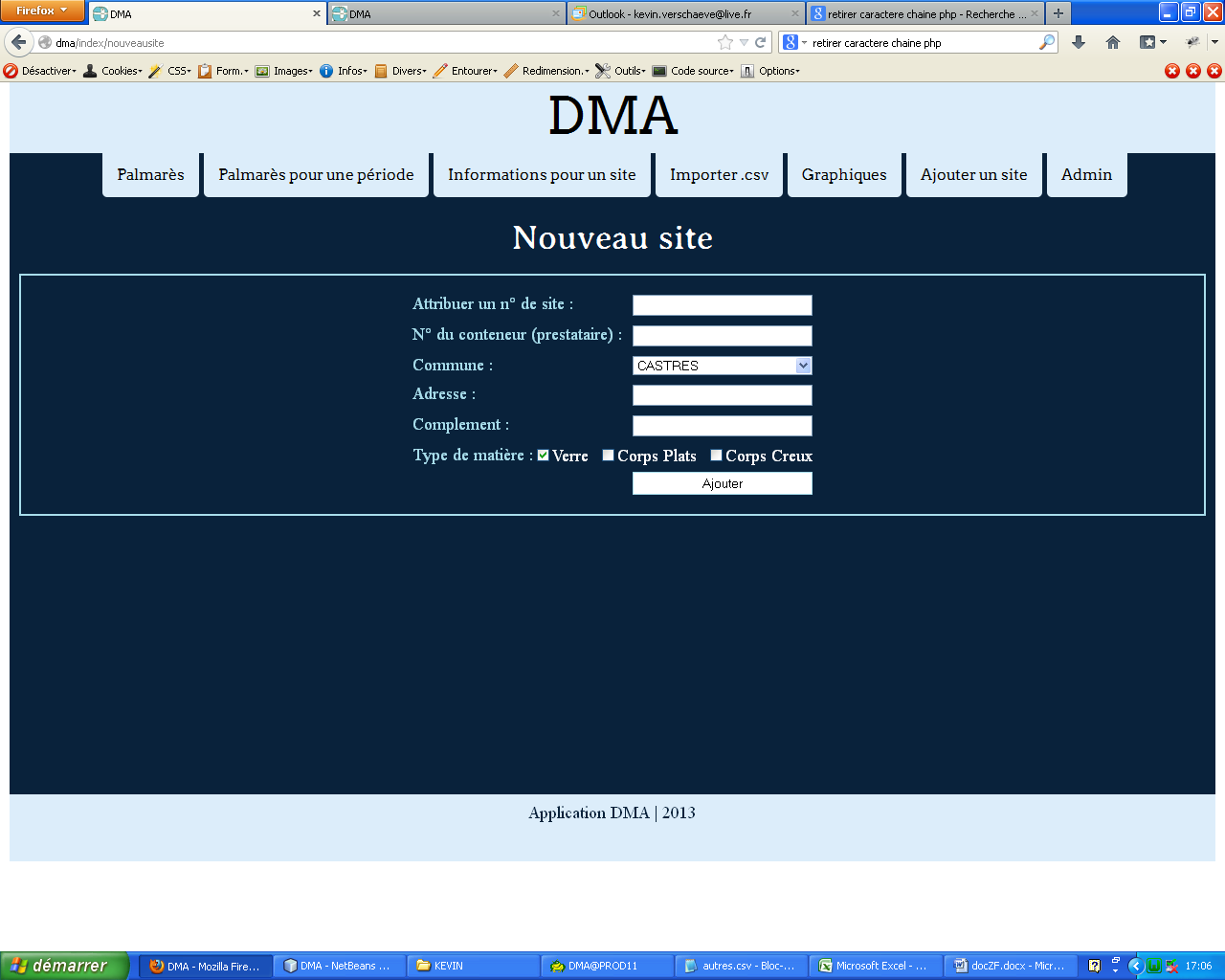
tonnageX = le tonnage pour la commune ou le site correspondant => C’est un float, en français les flottant s’écrivent 12,45 : la virgule modifie la structure du tableau et en anglais les flottants s’écrivent 12.45. Pour pouvoir correctement générer le graphique, on remplace la virgule par un point.

Pour pouvoir faire ce tableau dynamiquement, on dois parcourir 2 fois notre tableau d’entrée.

Une fois pour avoir le nom de la commune ou du site, une deuxième fois pour avoir le tonnage.

Les options permettent de changer l’affichage du graphique (couleur du texte, du titre, de la légende, emplacement de la légende, etc…)

* Ajouter un site :



Cette page est appelée par IndexController / nouveausiteAction.

Elle permet l’ajout d’un site en base de données.

Ici, on créé le formulaire d’ajout de site.

Si il à été envoyé et que les données sont valides, on les récupère, puis on dois ajouter 2 lignes dans la base de données.

Ajouter une ligne dans la table T\_PRESTATAIRE :

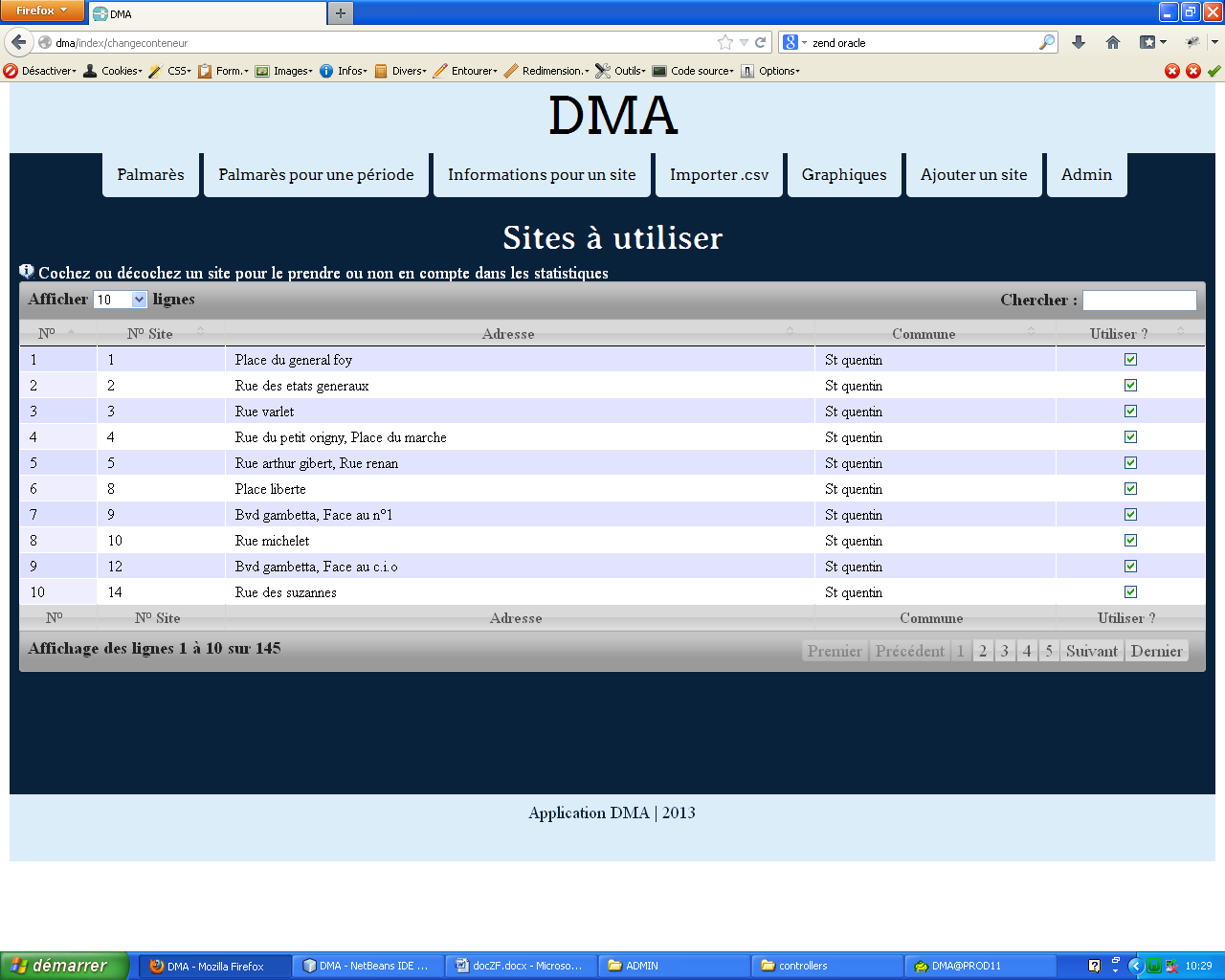
On créé un premier tableau ayant en clés, les noms des champs de cette table et en valeur les données a ajouter dans la table.

On fait ensuite ‘insert.

Ajouter une ligne dans T\_SITE :

On créé un deuxième tableau de la même structure que l’autre tableau mais avec les champs et les données correspondants à la table T\_SITE.

* Admin :



Cette page est appelée par IndexController / changeconteneurAction.

Elle permet d’utiliser un site ou non dans les statistiques et les graphiques. Elle modifie le champ STAT\_SITE d’un site.

On récupère, ici encore, un indicateur permettant de savoir si on est en ajax ou pas.

Si on n’y est pas :

On récupère tous les sites avec leurs infos grâce à la requête getSitesCommunes(), et on les envois à la vue.

Si on est en ajax :

C’est que l’on vient de cliquer sur un checkbox.

On récupère donc le checkbox (son attribut value étant égal à l’id du site du checkbox, on récupère donc l’id du site que l’on veut modifier), puis le nouvel état du checkbox (après le clic).

On change dans la base de données l’état en faisant appel à changeEtatStat().

Dans la vue :

On créé le tableau en parcourant chaque lignes retournées par la requête getSitesCommunes(), et dans le dernier td du tableau on y place un checkbox.

Pour la ligne en cours dans le foreach, on regarde la valeur du champ STAT\_SITE.

S’il est à 1, on coche le checkbox, sinon on laisse le checkbox normal.

**La requête GetInfos() du modele TSite :**

Elle peut recevoir jusqu'à 5 paramètres :

* nSite => on le spécifie si on veut les informations sur un et un seul site (il ajoutera la clause where ID\_SITE = $nSite; ), il prend null par défaut.
* matiere => Pour indiquer la matière sur laquelle on veut les informations. Si on ne l’indique pas, elle prend ‘Verre Couleur’ par défaut.
* dateDebut => pour spécifier si on veut des résultats à partir d’une certaine date
* dateFin => pour spécifier jusqu'à quelle date on veut les informations

/!\ si une des deux date est renseignée, il faut que l’autre le soit aussi !

* stat => Pour savoir si l’on doit prendre en compte le champ STAT\_SITE

Les informations renvoyées sont :

L’id de site, le nom du site, l’emplacement du site, la quantité collecté sur ce site, le nombre de relevés pour cette quantité, et le nom de la localité.

**L’api dataTable :**

Appliquer cette api à une table se fait de cette manière (avec jquery)

$(‘#idDeLaTable’).dataTable( { options } ) ;

« bJQueryUI »

Ajoute un style par défaut compris dans l’api

« bProcessing »

Affiche une information de chargement quand la table est trop longue à afficher

« sPaginationType »

Le type de navigation des pages de la table. Avec des numéros de pages, des flèches…

« aLengthMenu »

Cette option créé la liste déroulante permettant de modifier le nombre de lignes affichées à l’écran.

C’est un tableau de 2 tableaux.

Le premier tableau contient le nombre de résultats possibles à afficher.

Le deuxième, fait la correspondance avec le premier, et contient le nombre à afficher dans la liste déroulante.

« fnFooterCallback »

Cette option est une fonction appelée a chaque création de la table (la première fois, après un tri, un filtre, une nouvelle page).

Dans cette fonction, on parcourt toutes les lignes affichées a l’écran, et on additionne le tonnage, pour pouvoir ensuite l’affiché sur la page dans l’élément a l’id=’ttotal’

« aoColumns »

Déclare le type de chaque colonne (qui sert pour le tri).

Mettre à null, laisse l’api détecter le type.

Les deux fonctions jQuery.fn.dataTableExt.oSort permettent de trier de nombres à virgules.

Sans ces fonctions le tri ne retourne pas ce qu’il faut.