Développer

plus efficacement

En tant que développeur PHP, vous avez sûrement du avoir affaire à des scripts regorgeant de fonctions, de lignes de codes entremêlées, et surtout bon nombre de bugs tous plus mystérieux les uns que les autres.

Ce que cet article explique :

- Présentation de la bibliothèque Xdebug.
- Comment déboguer avec XDebug sous Eclipse.
- Sensibilisation au débugging et profiling.

Ce qu'il faut savoir :

- Connaître la notion de bibliothèques pour PHP.
- Connaître les bases de PHP.
- Avoir été confronté à des bugs au sein d'un projet.

Niveau de difficulté

Du coup, vous passez votre temps à déboguer grâce aux fidèles *echo* et autres *print_r*. Si vous êtes dans cette situation et que vous n'utilisez pas encore d'outils de *debugging*, cet article vous ouvre la voix vers un mode de développement un peu plus efficace.

Présentation d'XDebug

XDebug est une bibliothèque pour PHP, distribuée sous la forme d'extension, qui est capable de fournir bon nombre d'informations précieuses parmi lesquelles :

- Les différentes traces (des appels sur la pile, au sein d'une fonction...),
- Les allocations mémoire,
- Protection contre les récursivités infinies,
- *Profiling* de code (voir plus loin).

Notez également qu'XDebug s'intègre parfaitement avec symfony (via la *toolbar de debug*).

L'utilité d'une telle bibliothèque

Deux jours. Deux jours que vous venez de finir le développement d'un nouveau module de gestion de panier en ligne, et vous n'arrivez toujours pas à comprendre l'origine de ce message d'erreur si insignifiant. Il vous indique une ligne, certes, mais celle-ci vous semble totalement anodine. C'est là qu'XDebug intervient.

L'utilisation d'une bibliothèque tierce comme celle-ci permet – entre autre - de rendre les messages d'erreur beaucoup plus explicites, et détaillés. L'affichage complet des appels de fonctions effectués vous permettra de découvrir que le problème vient peut être du cœur même de votre application, un problème non soupçonné jusque là ! Alors que bon nombre de développeurs tentent de résoudre leur problème en insérant eux-même des instructions telles que *debug_backtrace*, *echo* ou *print_r* dans leur code, une utilisation optimale d'une bibliothèque comme Xdebug, couplée à un débogueur intégré à votre IDE, vous fera gagner de très, très précieuses minutes, heures ou jours.

Cet article s'adresse aux développeurs de tous niveaux, non seulement parce que cela fait partie des habitudes à prendre lorsqu'on débute, mais de surcroît car opter pour l'utilisation d'XDebug peut rendre bien des services dans un projet complexe. Je m'en sers très souvent dans mes projets symfony, et je m'imaginerais très mal développer sans dès lors que j'ai eu l'occasion d'essayer.

Xdebug... et les autres ?

Faisons si vous le voulez bien un tour d'horizon des solutions existantes... Il n'y en a pas pléthore ! Nous pouvons citer, parmi les plus connues, *DBG, Pear::Benchmark, Pecl::APD*... Or il se trouve que la plupart de ces solutions ne proposent pas autant de fonctionnalités qu'XDebug. Bien sûr chacune a son petit avantage à proposer, mais notre choix nous permet de profiter de toutes les fonctionnalités citées au dessus via une seule et unique extension, plutôt efficace et fort bien documentée.

Installation d'XDebug

Commencez par vous rendre sur le site officiel du projet, *http://www.xdebug.org*. Vous trouverez directement sur la page d'accueil différents liens. Choisissez celui qui convient à votre système d'exploitation et à votre version de PHP. Notez qu'une installation par PEAR est également possible. Il vous suffit de taper, dans un terminal :

```
pecl install xdebug
```

L'installation par PEAR sous Unix nécessite la présence du paquet *php5-dev* (ou *php-dev*, ou encore *php5-devel*...). Une fois XDebug téléchargé, il vous reste à configurer correctement PHP pour prendre en compte cette nouvelle extension. Ouvrez votre fichier de configuration *php.ini*, et ajoutez-y les lignes suivantes :

```
; Remplacez par le chemin vers le
      fichier .dll
zend_extension_ts="C:\wamp\bin\php\
      php5.2.6\ext\php_xdebug.dll"
xdebug.remote_enable=1
xdebug.remote_handler=dbgp
xdebug.remote_mode=req
xdebug.idekey=default
xdebug.idekey=ECLIPSE_DBGP
xdebug.profiler_enable=1
xdebug.remote_port=9000
xdebug.show_local_vars=1
Sous Linux, remplacez la première ligne par
      zend_extension="/usr/lib/php5/
      apache2/xdebug.so"
```

Tester l'installation de la bibliothèque

Afin de contrôler la bonne installation de la bibliothèque, je vous invite à écrire un script très basique, exploitant une fonctionnalité d'XDebug :

```
<?php
   function fix_string($a) {
```

```
        echo "Called @ " . xdebug_call_file()
              . " : " . xdebug_call_line() .
              " from " . xdebug_call_
              function();
    }
    $ret = fix_string(array('Tekover'));
?>
```

Si l'installation s'est correctement déroulée, vous devriez obtenir un message du genre *Called @ /home/httpd/html/test/xdebug_caller.php:12 from {main}.*

Dans la suite de cet article, nous ne ferons plus appel aux fonctions fournies par la bibliothèque par nous-même, mais rien ne vous empêche de le faire au sein de vos scripts. La documentation en ligne est très bien détaillée.

Intégration d'XDebug avec Eclipse PDT

Désolé pour tous les utilisateurs de *Netbeans* ou d'un autre IDE, je ne vais traiter ici que de l'exemple d'Eclipse PDT. La manipulation est néanmoins réalisable - et plus ou moins similaire - au sein des autres IDE (autres distributions d'Eclipse incluses). Vous pouvez retrouver la liste des éditeurs supportés sur le site officiel d'XDebug, à cette page : *http://www.xdebug.org/docs/remote*.

Ouvrez les préférences d'Eclipse, et rendez-vous dans la section *PHP... Debug*. Sélectionnez le débogueur XDebug, comme sur la Figure 1. Le port 9000 est normalement configuré par défaut, mais vous pouvez toujours vérifier par vous-même.

Utilisation du débogueur

Maintenant que tout est - normalement - en place, nous pouvons passer à un cas pratique d'utilisation d'XDebug. Pour prendre en main ce nouvel outil, créez-vous un script basique, du type :

```
<?php
function add_two($nb) {
    return $nb + 2;
} function foo($nb) {
    return add_two($nb);
}
$a = 42;
$b = foo($a);
?>
```

Dans Eclipse, faites un clic droit sur le script, et choisissez *Debug as... PHP Web Page* (Figure 2). Eclipse peut vous demander, la première fois, d'entrer l'URL de la page ou du serveur (Figure 3). Montrez-vous sympathique et obéissez aux ordres de votre IDE. Votre script doit alors s'ouvrir au sein de votre navigateur, et doit sembler attendre quelque chose. Si Eclipse ne l'a pas fait de lui même, passez dans la vue *PHP Debug* (menu *Window... Open Perspective... Debug*). Vous devriez avoir une fenê-

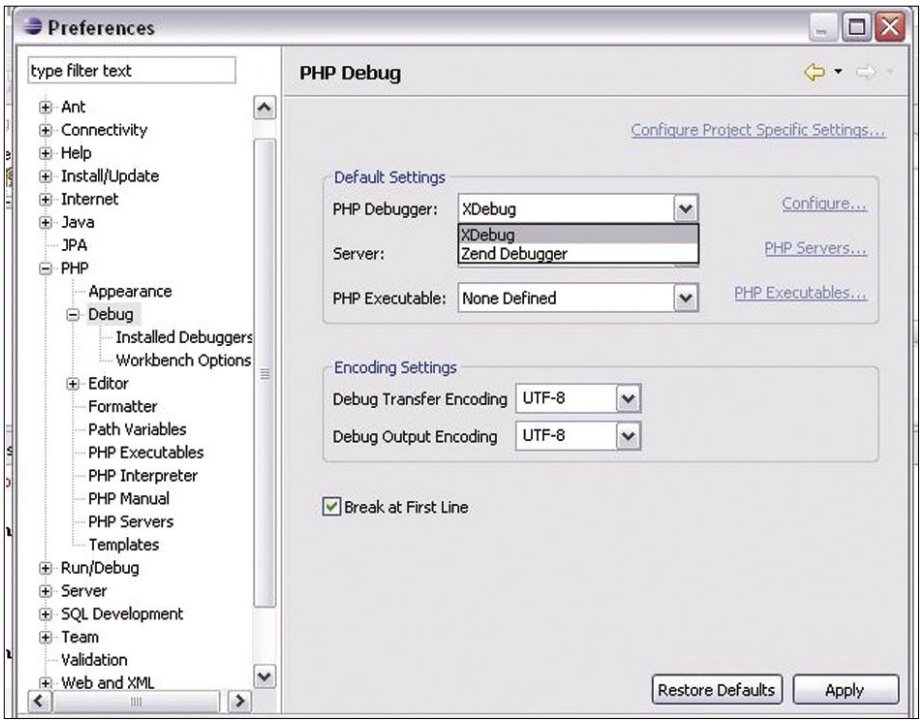

Figure 1. *Panneau de préférences*

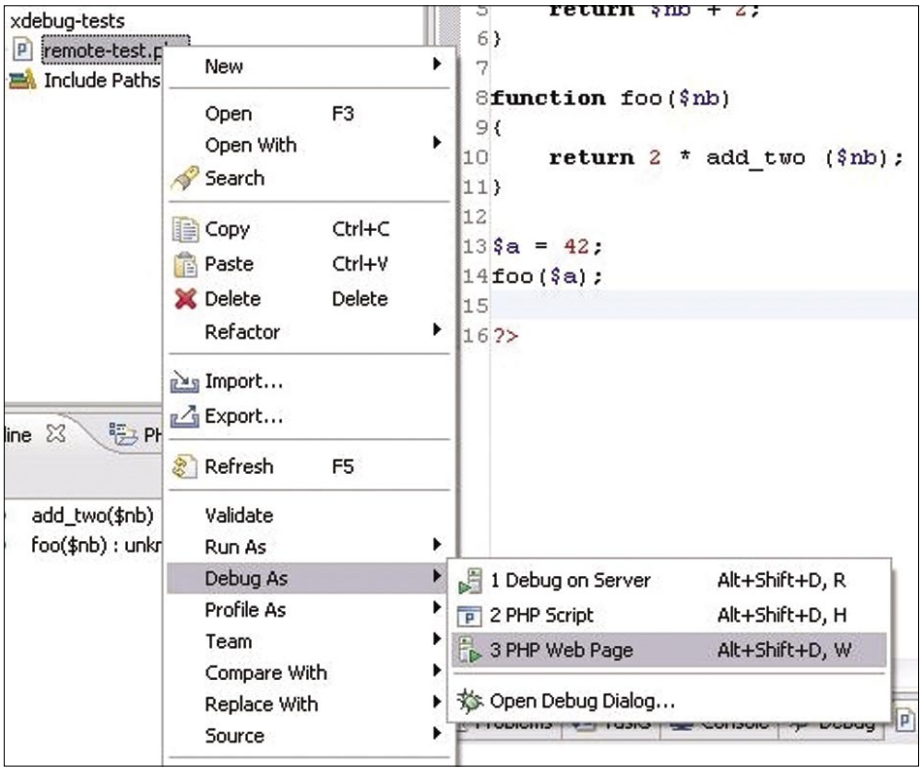

Figure 2. *Lancement d'une session de débogage*

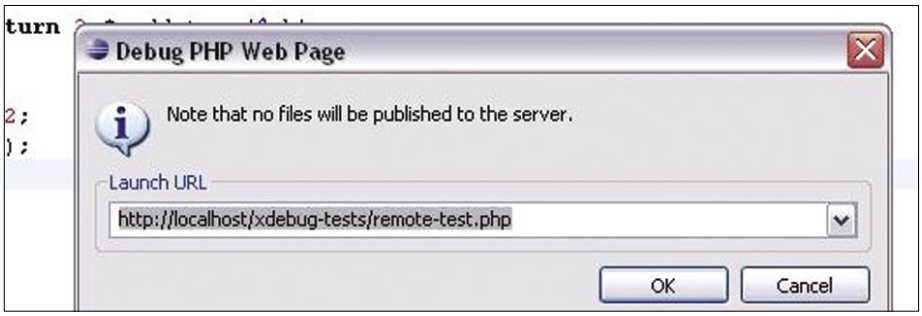

Figure 3. *Spécification de l'URL à utiliser*

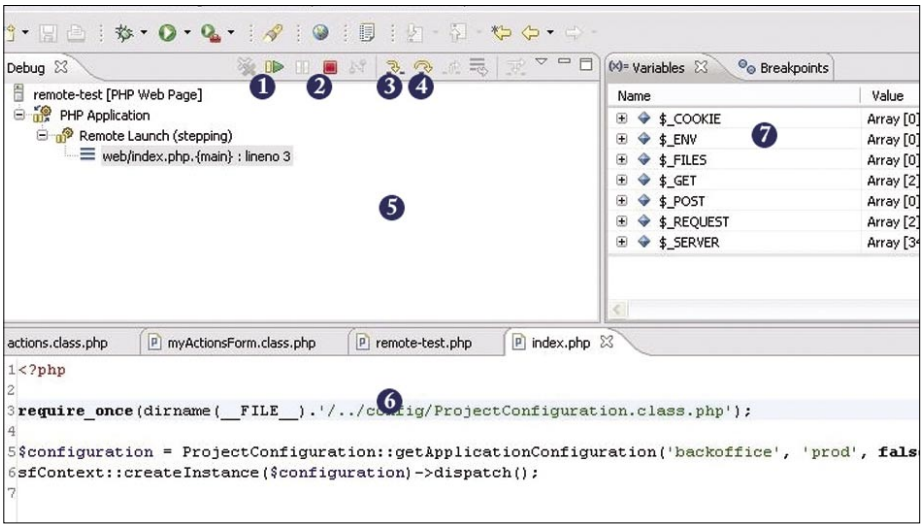

Figure 4. *Interface du débogueur Eclipse*

tre comme sur la Figure 4. Voici un descriptif du comportement de chacun des boutons :

- 1: Lancer le script jusqu'au prochain point d'arrêt,
- 2: Arrêter le script,
- 3: Avancer à la ligne suivante,
- 4: Passer la boucle/fonction suivante,
- 5: Trace actuelle (pile d'appel),
- 6: Votre code,
- 7: Variables.

Il y a deux autres fonctionnalités et concepts de base que vous devriez connaître pour une utilisation efficace du débogueur : premièrement, la notion de *breakpoint* (point d'arrêt). Vous pouvez en effet indiquer à votre débogueur d'exécuter normalement votre script, mais de s'arrêter à une ligne précise. Habituellement, il suffit de cliquer (ou double cliquer) à gauche du numéro de ligne pour poser un point d'arrêt sur cette ligne. Dans notre exemple, essayez de lui dire de s'arrêter au début de la fonction `foo`.

La deuxième chose à connaître est la notion de *watcher*. Ces *watchers* vous permettent de surveiller le contenu d'une variable. Dans notre exemple précédent, sélectionnez la variable `$a`, faîtes clic-droit et sélectionnez *watcher*. À chaque évaluation de la variable `$a`, vous pouvez suivre sa valeur dans le cadre dédié.

Maintenant que vous connaissez le comportement et le fonctionnement général du débogueur, entraînez-vous sur notre exemple. En cliquant sur le *bouton 3*, essayez d'observer le comportement. Mettez des *watchers* où bon vous semble. L'exemple ici est très simple, mais cette méthode est parfaitement exploitable dans le cadre d'un développement au sein d'une application complexe (avec un framework par exemple). Lors d'une telle situation, notez que vous avez juste à ouvrir une page via Eclipse (clic droit sur *index.php* par exemple, puis *Debug As... PHP Web Page*). Vous pouvez ensuite naviguer au sein de votre application (tenter de vous identifier exemple) et suivre le parcours de interpréteur PHP via le débogueur. Par défaut, ce dernier s'arrête au début de l'exécution de chaque page. Si cela vous ennuie, et c'est fort probable, vous pouvez désactiver ce comportement en décochant la case *Break at first line* dans le panel de *configuration* (voir Figure 1).

Profiling de code

Une des fonctionnalités intéressante d'XDebug est sa capacité de fournir des fichiers de *logs* avec l'intégralité des actions effectuées à l'affichage d'une page, avec le temps requis pour chacune de ces actions. Pour activer cette fonctionnalité, rajoutez les lignes suivantes dans votre fichier de configuration *php.ini* (ici, les fichiers seront automatiquement placés dans le répertoire *c:\wamp\log*) .

```
xdebug.profiler_output_dir="C:\wamp\log\"
xdebug.profiler_output_name=
      "%t.cachegrind.out"
xdebug.profiler_enable_trigger=1
```

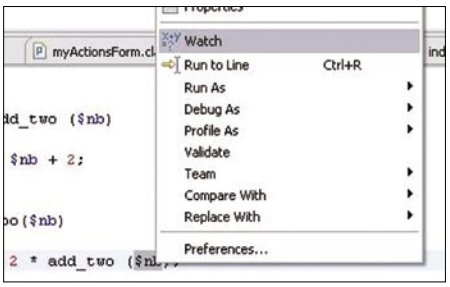

Figure 5. *Positionnement d'un Watcher*

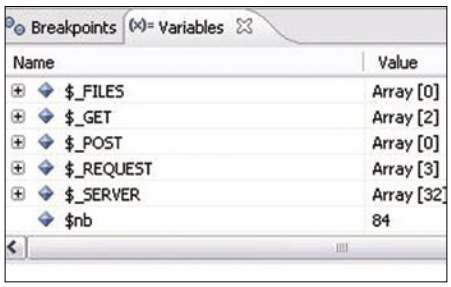

Figure 6. *État en direct des variables*

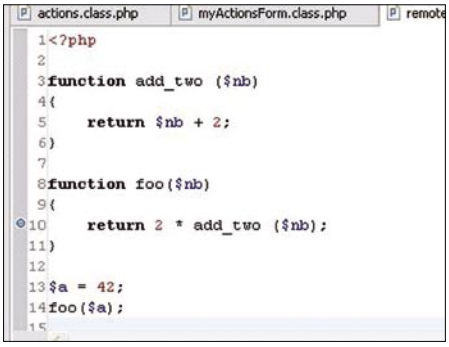

Figure 7. *xxxxxxxxxx*

N'oubliez pas de redémarrer le serveur pour que les modifications soient prises en compte. Lancez l'affichage d'une page, et regardez le dossier *c:\wamp\log*. Un fichier y a été créé. Vous pouvez ouvrir ces fichiers avec *WinCachegrind* (Windows) ou *KCachegrind* (sous l'environnement KDE), par exemple.

Conseil : pensez à bien désactiver cette option si votre serveur doit servir dans un environnement de production ! L'écriture de ces fichiers prend un temps (et un espace disque) non négligeable.

Pourquoi du profiling de code ?

La lecture de ces fichiers vous donnera de très précieuses informations sur les différents appels effectués, et vous laissera sûrement penser qu'une partie de votre code mériterait d'être revue et optimisée, sous peine de coûter énormément de temps en cas de forte charge. Vous avez maintenant tout le loisir de décortiquer les temps d'exécution de chaque appel de fonction (les fonctions natives de PHP sont préfixées par `php::`), les différentes inclusions de fichiers pas forcément toujours judicieuses, ou encore la consommation mémoire de votre script.

Une telle approche peut se révéler utile tant pour contrôler le bon fonctionnement de son application, que pour découvrir quelle fonction est particulièrement lente, au point de faire perdre 2 secondes à chaque génération.

Conclusion

J'ai pris le temps d'exposer ces différentes méthodes de travail car trop souvent on assiste à des scènes de debuggages laborieuses pour lesquelles l'utilisation d'outils comme ceux-ci s'avèrerait déterminante et primordiale. Il ne me reste plus qu'à vous souhaiter bon courage dans l'utilisation de votre débogueur. J'espère que vous adopterez ces pratiques et qu'elles vous feront gagner des heures bien précieuses. N'hésitez pas à me contacter personnellement pour toute remarque, question ou commentaires !

ADRIEN MOGENET

Actuellement étudiant à l'EPITA, Adrien Mogenet est passionné des technologies Open Source et par toutes les problématiques d'optimisation du développement d'un projet.

Contact : adrien@frenchcomp.net