# BUG ET DEBUG

# DEBUG des scripts PHP

# Les niveaux d'erreur de PHP

- Pour debug, il faut commencer par la configuration dans php.ini :
  - display_errors = On
  - error_reporting = E_ALL
- Ce dernier peut aussi être défini dans le code :
  - error_reporting(E_ALL ^ E_NOTICE);
- Ou pour SPIP, dans mes_options.php :
  - define('SPIP_ERREUR_REPORT',E_ALL ^ E_NOTICE);
  - define('SPIP_ERREUR_REPORT_INCLUDE_PLUGINS', E_ALL ^ E_NOTICE);

# La méthode débrouille

- On ajoute du code de test pour
  - voir quel chemin suit le script
    - <?php echo « il est passe par ici » ?>
    - die('ici');
  - Voir la valeur d'une variable ou d'une expression
    - var_dump($ma_variable);
  - Voir la pile d'appel
    - debug_print_backtrace();
- C'est intrusif et long
  - On modifie le code « pour voir »
  - On relance le script,
  - et on recommence jusqu'à réussir à voir la bonne variable, ou le bon chemin et comprendre le problème
- Et on laisse des var_dump dans le code …

# Améliorer les affichages

Des affichages plus riches avec XDEBUG

# XDEBUG

- S'installe comme une extension de php
  - Compilée en .dll sous windows ou .so sous *nix (la compilation sous Mac OS nécessite l'installation de Xcode)
  - http://devzone.zend.com/article/2803
  - Guide d'install pour MAMP+Mac OS http://www.netbeans.org/kb/docs/php/configure-php-environment-mac-os.html#installEnableXdebug

# XDEBUG

- http://www.xdebug.org/
- Librairie libre sous licence dérivée de PHP
- Librairie qui facilite le debug
  - Amélioration des affichages de debug
  - Debug interactif
  - Profilage
  - Couverture de code

# Un joli var_dump

```
array
  1 => string 'Apple' (length=5)
  2 => string 'Pear' (length=4)
  3 => string 'Banana' (length=6)

object(Test)[1]
  public 'name' => string 'Test' (length=4)
  protected 'connected' => boolean false
  protected 'foo' =>
    object(Foo)[2]
      protected 'foo' => int 2357231
      protected 'bar' => float 1234723.234
```

- Configurable via php.ini :
  - Longueur maxi des chaines affichées (512 par défaut) : xdebug.var_display_max_data
  - Nombre maxi d'éléments dans les tableaux (128 par défaut) : xdebug.var_display_max_children

# Des erreurs plus explicites

**( ! ) Notice: Undefined variable: m in C:\www\local_vars.php on line *23***

**Call Stack**

| # | Time | Memory | Function | Location |
|---|---|---|---|---|
| 1 | 0.0279 | 55896 | {main}( ) | ..\local_vars.php:0 |
| 2 | 0.0280 | 56144 | foo( ) | ..\local_vars.php:6 |
| 3 | 0.0280 | 56392 | bar( ) | ..\local_vars.php:10 |
| 4 | 0.0280 | 56696 | baz( ) | ..\local_vars.php:15 |

# Des erreurs plus explicites

- xdebug.show_local_vars=1

**( ! ) Notice: Undefined variable: m in C:\www\local_vars.php on line *23***

**Call Stack**

| # | Time | Memory | Function | Location |
|---|---|---|---|---|
| 1 | 0.0100 | 56520 | {main}( ) | ..\local_vars.php:0 |
| 2 | 0.0101 | 56728 | foo( ) | ..\local_vars.php:6 |
| 3 | 0.0101 | 56952 | bar( ) | ..\local_vars.php:10 |
| 4 | 0.0101 | 57256 | baz( ) | ..\local_vars.php:15 |

**Variables in local scope (#4)**

| | |
|---|---|
| $m | *Undefined* |
| $str = | `string 'Hello World' (length=11)` |
| $t = | `int 3` |

# Des erreurs plus explicites

- xdebug.collect_params pour afficher les parametres des fonctions
  - 0 ne les affiche pas, par défaut
  - 1 type et nombre d'éléments des paramètres
  - 2 idem, mais en tooltip (pas idéal pour l'impression)
  - 3 : noms et valeurs des paramètres, mais longues valeurs tronquées
  - 4 : noms et valeurs complètes des paramètres

# Des erreurs plus explicites

- xdebug.show_local_vars=On
- xdebug.dump.SERVER=
  HTTP_HOST,
  SERVER_NAME
- xdebug.dump_globals=On
- xdebug.collect_params=4

**Notice: Undefined variable: m in C:\www\local_vars.php on line *28***

**Call Stack**

| # | Time | Memory | Function | Location |
|---|---|---|---|---|
| 1 | 0.0091 | 68064 | {main}( ) | ..\local_vars.php:0 |
| 2 | 0.0092 | 68736 | foo( $a = 1, $b = 2 ) | ..\local_vars.php:11 |
| 3 | 0.0092 | 68960 | bar( $x = 4, $y = 1 ) | ..\local_vars.php:15 |
| 4 | 0.0092 | 69264 | baz( $i = 7, $k = 5 ) | ..\local_vars.php:20 |

**Dump *$_SERVER***

| | |
|---|---|
| $_SERVER['HTTP_HOST'] = | string 'localhost' *(length=9)* |
| $_SERVER['SERVER_NAME'] = | string 'localhost' *(length=9)* |

**Variables in local scope (#4)**

| | |
|---|---|
| $m | *Undefined* |
| $i = | int 7 |
| $k = | int 5 |
| $str = | string 'Hello World' *(length=11)* |
| $t = | int 3 |

# Afficher la pile d'appel

- xdebug_print_function_stack
- 
```
<?php
function foo( $far, $out ){
    xdebug_print_function_stack( 'Your own
message' );
}
foo( 42, 3141592654 );
?>
```

( ! ) Xdebug: Your own message in /home/httpd/html/test/xdebug/print_function_stack.php on line 5

Call Stack

| # | Time | Memory | Function | Location |
|---|---|---|---|---|
| 1 | 0.0006 | 653896 | {main}( ) | ../print_function_stack.php:0 |
| 2 | 0.0007 | 654616 | foo( 42, 3141592654 ) | ../print_function_stack.php:7 |
| 3 | 0.0007 | 654736 | xdebug_print_function_stack<br>( 'Your own message' ) | ../print_function_stack.php:5 |

# Faire du debug interactif

Avec XDEBUG et un éditeur de code

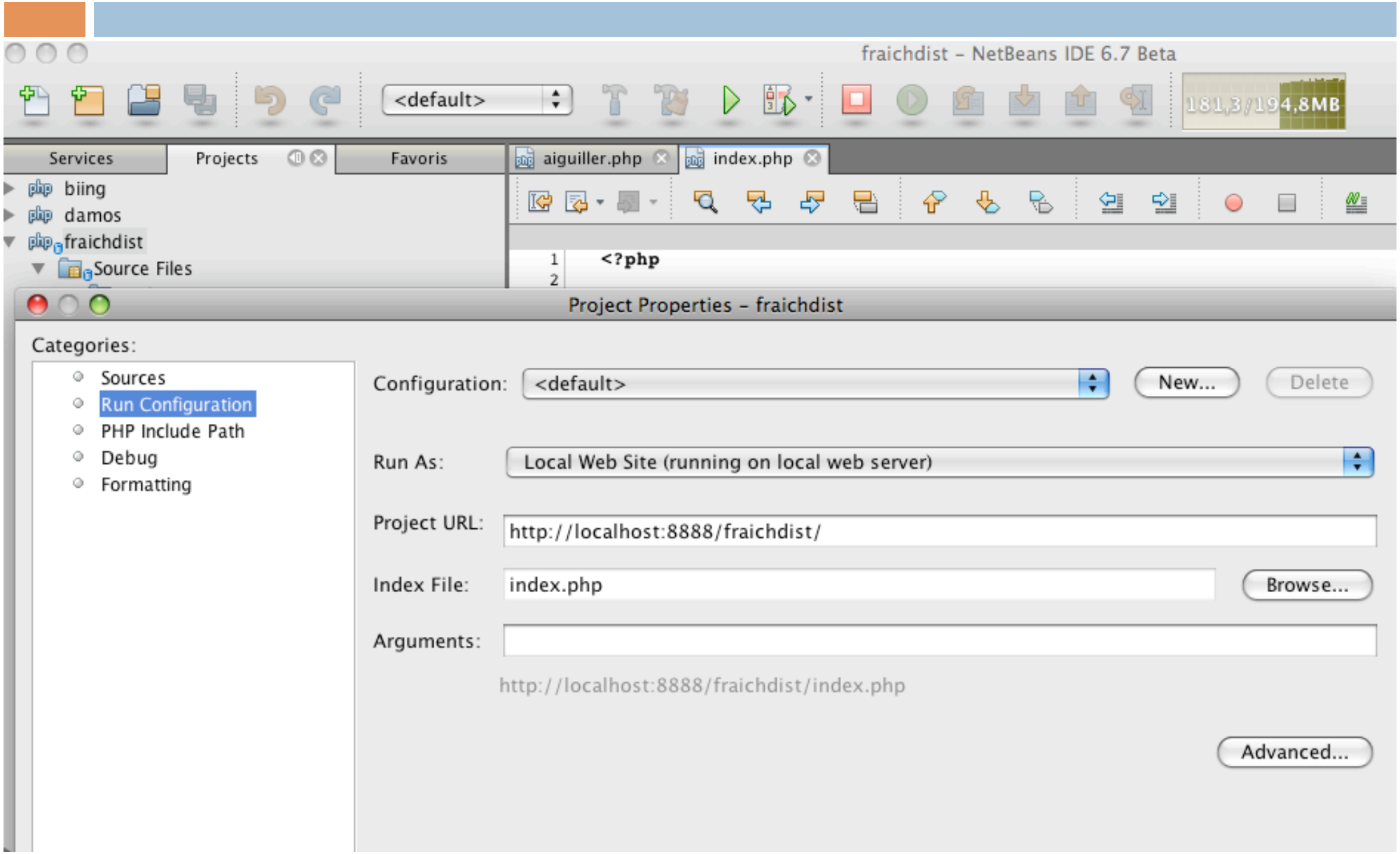
# Nécessite un outil « client »

- Des Editeurs de code évolués ou des IDE (Environnement de Développement Intégré), au choix :
  - Dev-PHP (IDE: Windows)
  - Eclipse plugin, which has been submitted as an enhancement for the PDT (IDE).
  - Emacs plugin (Editor Plugin).
  - ActiveState's Komodo (IDE: Windows, Linux, Mac; Commercial).
  - MacGDBP - Standalone Mac client.
  - NetBeans (IDE: Windows, Linux, Mac OS X and Solaris.
  - Notepad++ plugin (IDE: Windows).
  - WaterProof's PHPEdit (IDE, from version 2.10: Windows; Commercial).
  - Anchor System's Peggy (IDE: Windows, Japanese; Commercial).
  - MP Software's phpDesigner (IDE: Windows, Commercial).
  - PHPEclipse (Editor Plugin).
  - Protoeditor (Editor: Linux).
  - tsWebeditor (Editor: Windows).
  - Xored's TrueStudio IDE (IDE; Commercial).
  - VIM plugin (Tutorial) (Editor Plugin).
  - jcx software's VS.Php (MS Visual Studio Plugin; Commercial).
  - XDebugClient - Standalone Windows client.

# Exemple de debug interactif

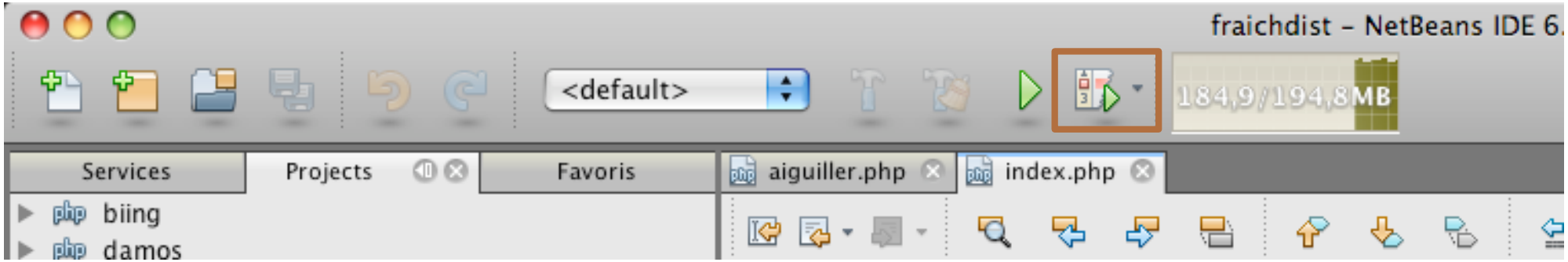
- L'exemple va utiliser
  - NetBeans pour le client qui permet de suivre le code
  - Une extension FF pour démarrer/arrêter le debug: XDEBUG Helper
    - https://addons.mozilla.org/fr/firefox/addon/3960

# Configurer le projet : url d'execution



# Lancer le debuger

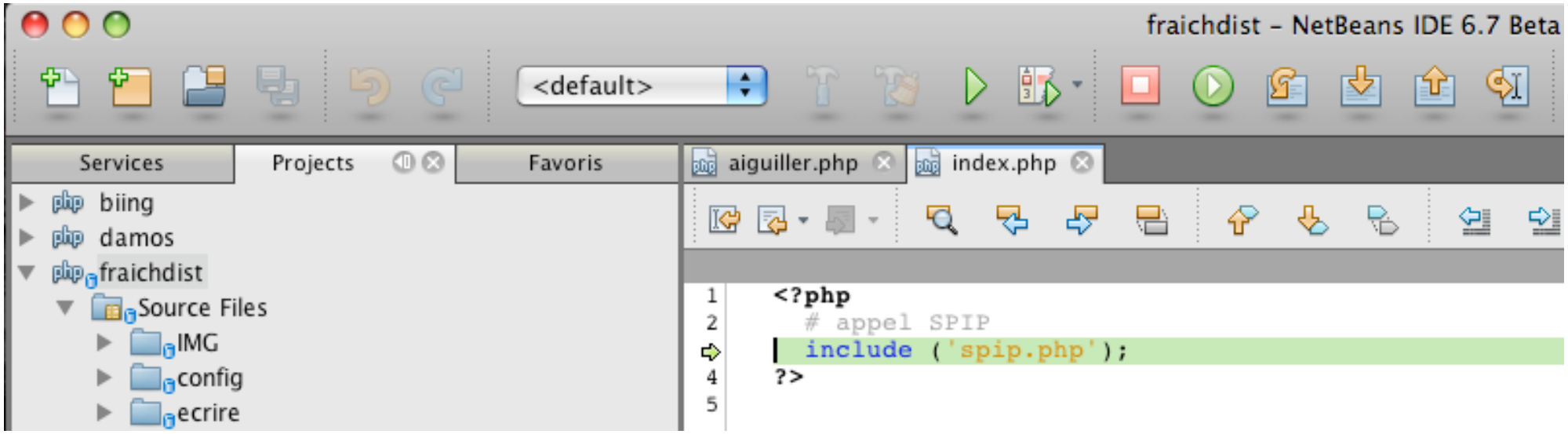
- Depuis NetBeans
  - Debug>Debug Project ou icône dans la barre
  - Va lancer le debuger sur l'url par défaut du projet



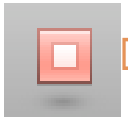

  - L'url est lancée dans le navigateur qui se met en attente

# Lancer le debuger (suite)

- Dans Netbeans, on visualise l'exécution qui s'est arrêtée sur la première instruction

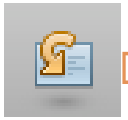


# Contrôler l'exécution



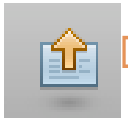
- Finish : arrête la session de debug (éviter de l'utiliser cf après)

- Continue : l'exécution reprend jusqu'à la fin, ou jusqu'au prochain point d'arrêt



- Step Over : exécuter l'instruction sans faire de pas à pas dans les fonctions appelées

- Step In : avance d'un pas, en s'arrêtant dans la première fonction appelée si il y a lieu

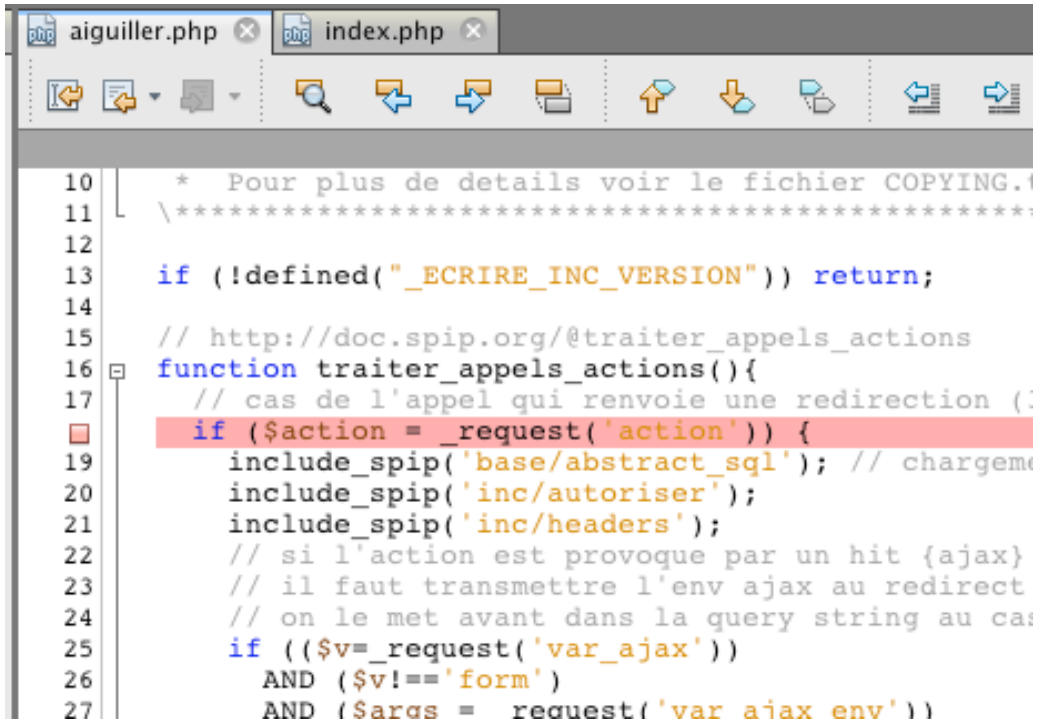


- Step Out : Avancer l'exécution jusqu'à ressortir de la fonction en cours

- Run to cursor : avancer l'exécution jusqu'au curseur

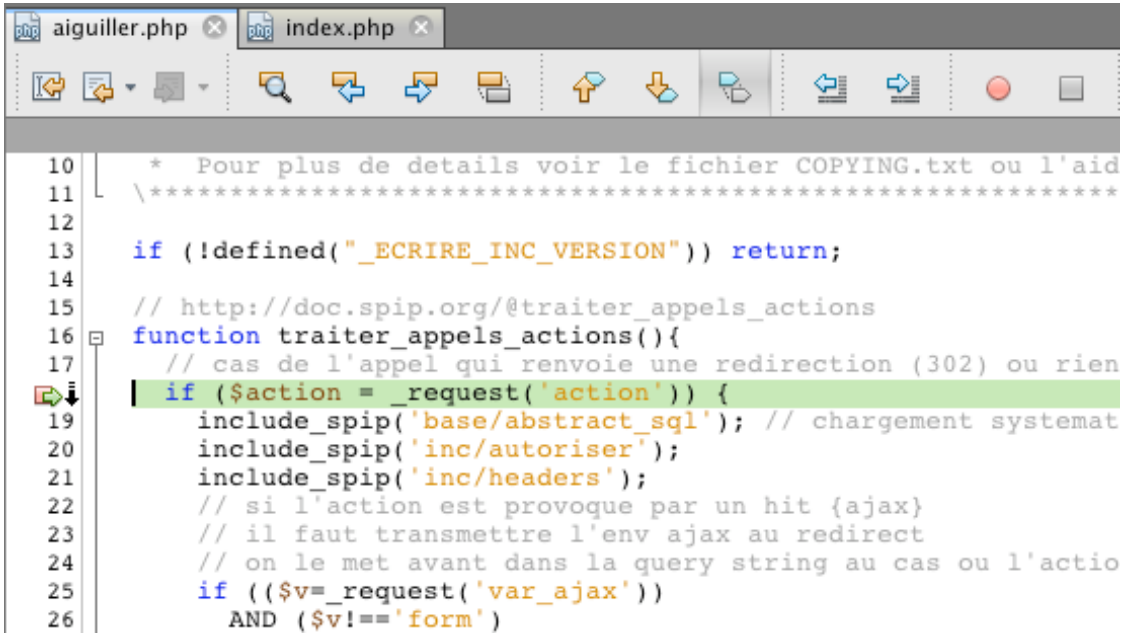
# Placer un point d'arrêt (BreakPoint)

- On se place sur la ligne de code où l'on veut visualiser l'exécution
  - Menu Debug > Toggle Line Breakpoint
  - Ou clic dans la marge pour activer le point d'arrêt



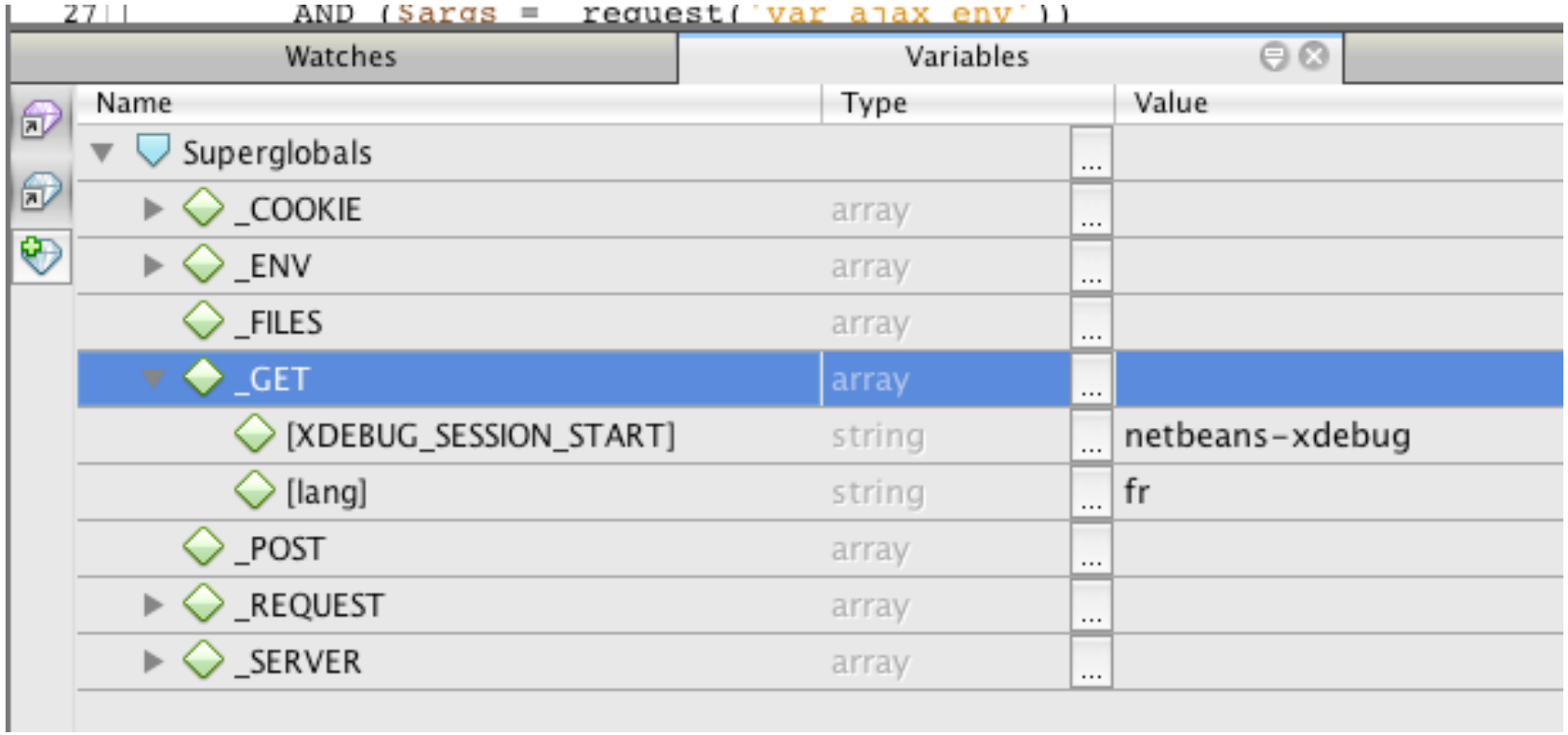
# Que faire avec un point d'arrêt ?

- Permet de jouer l'exécution jusqu'à l'endroit du code où l'on cherche le bug, sans faire du pas a pas intégral
- Faire « Continue » jusqu'à ce que le debugger s'arrête sur cette ligne



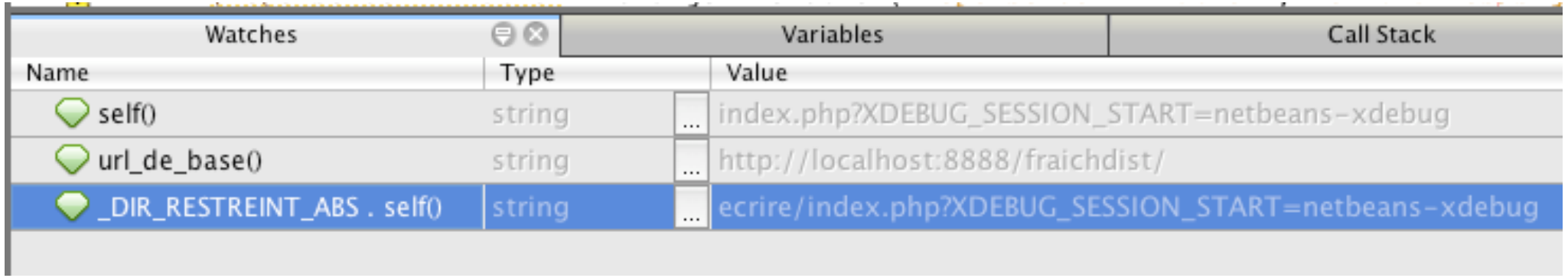
# Variables

- On peut voir toutes les variables existantes à chaque instant
  - Les superglobales, les globales, et les locales



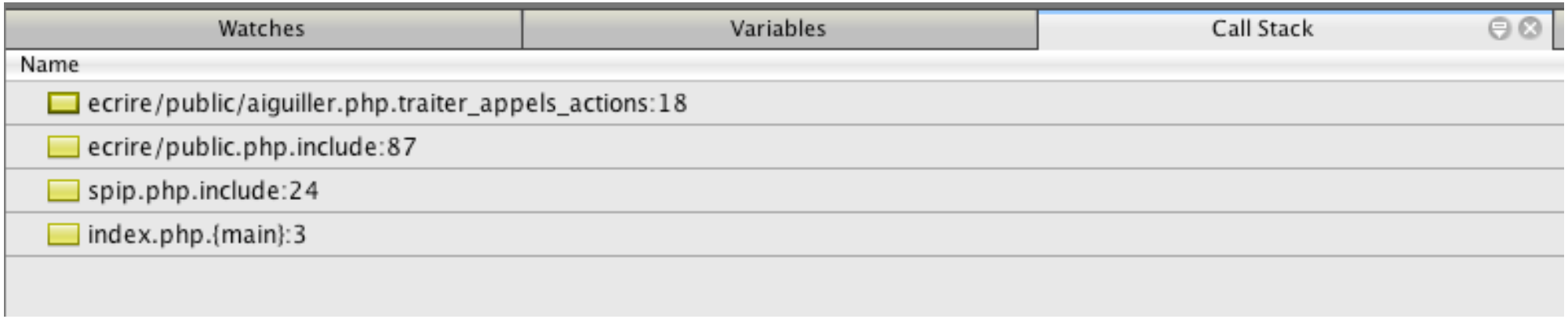
# Evaluations (Watches)

- On peut visualiser la valeur d'expression calculée
  - Appel à une fonction
  - Constante
  - Expressions complexes
- Permet
  - d'analyser la situation 'dans le contexte', en disposant de toutes les valeurs des variables
  - De tester d'un seul coup plusieurs corrections possibles



# Pile d'appel (Call Stacks)

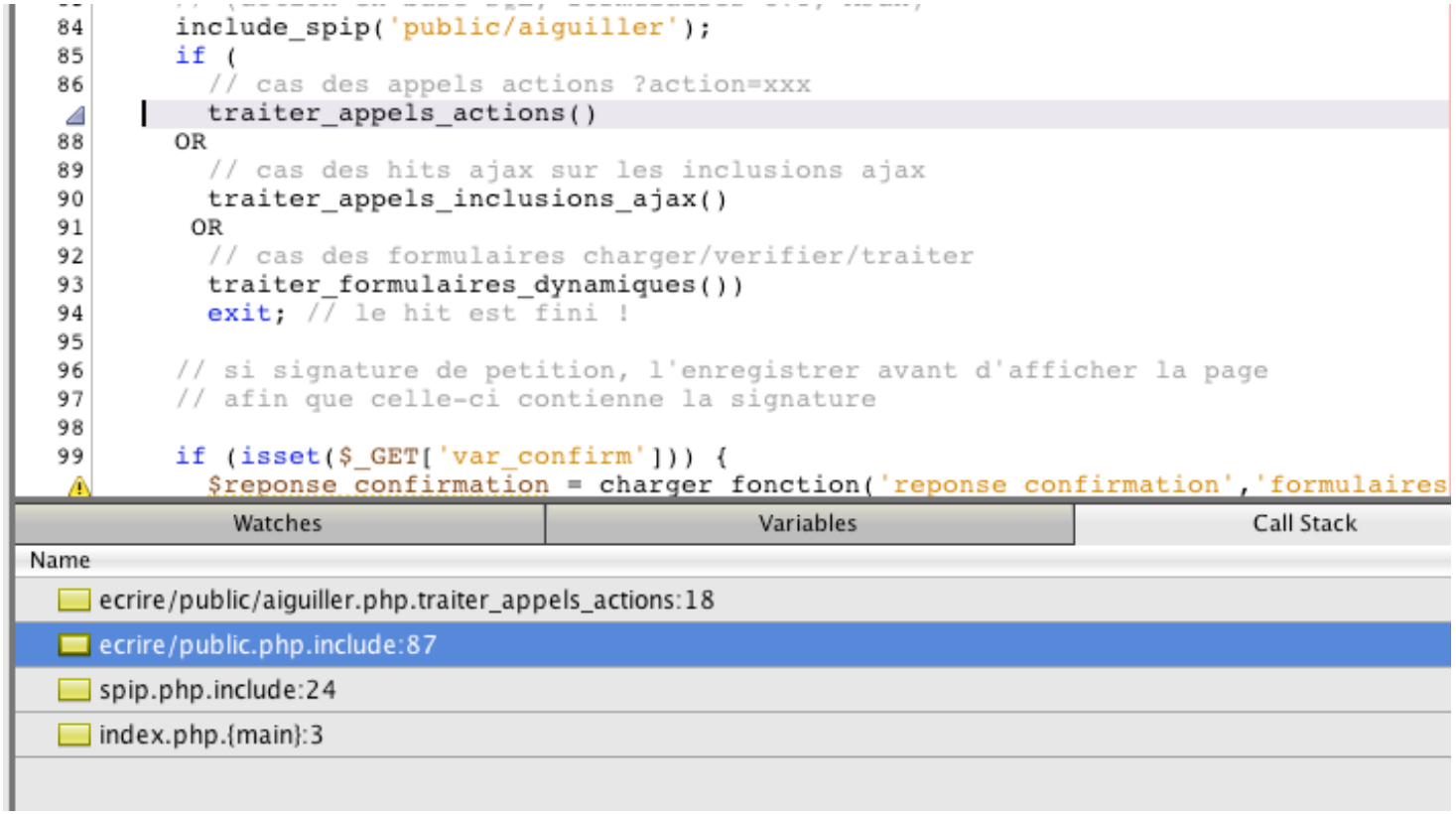
- La pile d'appel permet a tout moment de voir par où on est arrivé sur la ligne en cours




- Il est possible  de remonter la pile pour voir la valeur des variables en amont

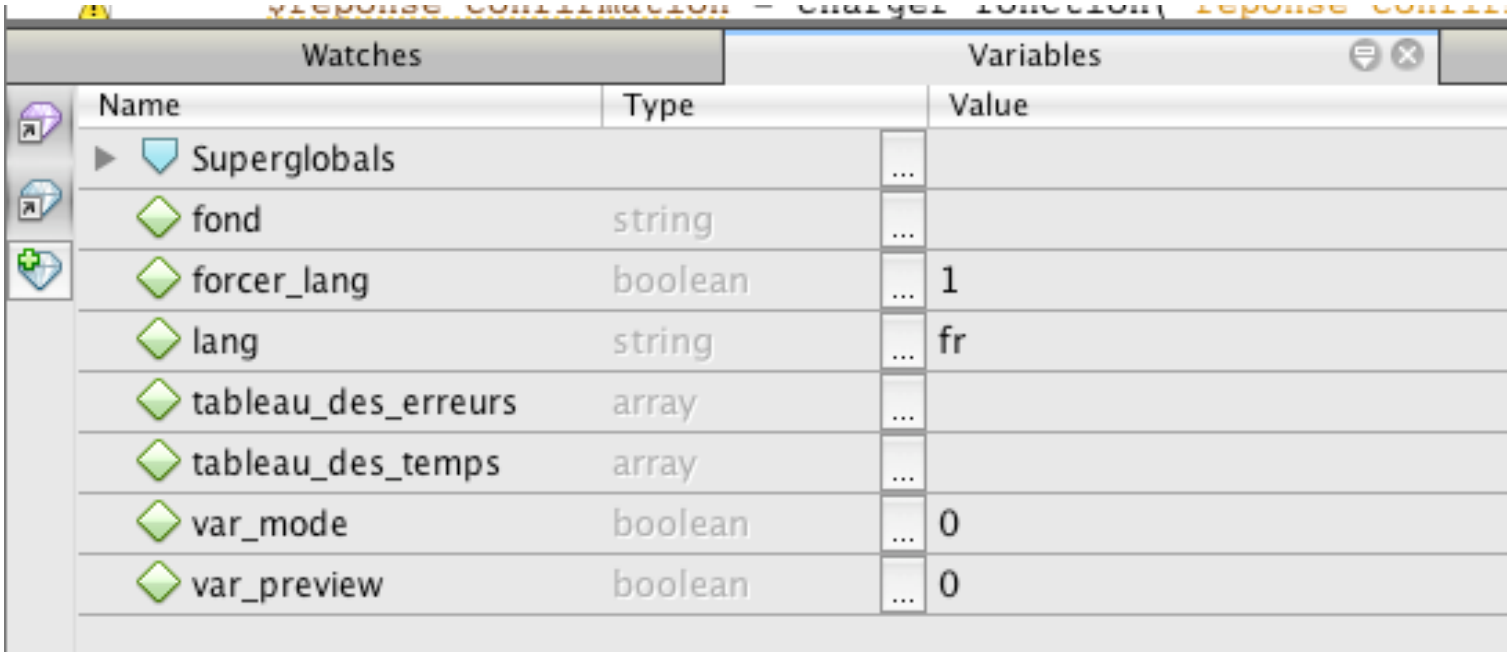
# Pile d'appel

- Dans notre exemple, on peut remonter a l'appel de la fonction traiter_appel_actions() …



# Pile d'appel&variables

… et visualiser les variables contextuelles (ou même les modifier)



# Suite de la session de debug

- Après avoir observé le comportement dans la fonction que l'on voulait analyser, on fait « continue » pour laisser le script se finir
- NetBeans est toujours à l'écoute
  - Si on clic sur un autre lien dans le navigateur, le hit sera a nouveau interrompu sur la première instruction, visible dans NetBeans
  - On peut desactiver l'écoute dans FF avec l'icône XDEBUG
    - Icône allumée : on est en session de debug 
    - Icône éteinte, on peut naviguer normalement 

# Debuger d'autres pages

- Il faut mettre NetBeans à l'écoute par une première session de debug sur la page par défaut
- Desactiver le debug côté navigateur avec l'icône XDEBUG
- Aller sur la page à debuger
- Ré-activer le debug côté navigateur
- Soumettre la page
  - Permet le debug de formulaires en POST par exemple
- Dans la pratique, on laisse NetBeans en mode debug toute la session de travail, et on active/desactive côté navigateur uniquement.

# L'intérêt du debug interactif ?

- Permet de trouver **plus vite** OÙ est le bug :
  - En mettant des points d'arrêt dans les fonctions candidates
  - En observant le pas à pas pour voir si le comportement est conforme à ce qui était attendu
- Permet de comprendre **plus vite** QUEL est le bug
  - En visualisant d'un coup toutes les variables utiles, même celles que l'on aurait pas penser à mettre en var_dump
  - En évaluant des expressions calculées

# Un outil à prendre en main

- Nécessite de faire l'effort d'apprentissage des outils
- Investissement en temps très rentable
  - Le temps gagné pour trouver les bugs compense largement l'effort initial pour apprendre l'outil

# Des outils libres performants

- Tous les outils nécessaires sont disponibles en libre :
  - XDEBUG + un editeur/IDE libre
  - La variété des éditeur/IDE permettant le debug interactif permet a chacun de trouver son bonheur
  - Il y a dans le lots des outils multiplateformes qui permettent d'avoir le même environnement de travail que l'on soit sous Mac OS, Win ou Linux
- Les outils payants type Zend
  - Ne sont pas plus performants (pas de fonctionnalité en plus)
  - Mais sont plus faciles d'utilisation (meilleure intégration editeur/navigateur pour le debug)
  - Sont payants …

# Et en production ?

Retour de la bidouille …

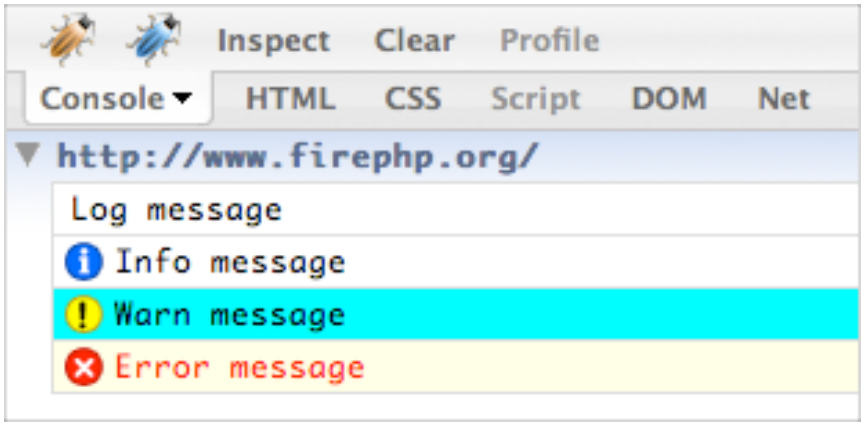
# Développement, production, et serveur distant

- Il est possible de faire de debug sur un serveur distant en y installant xdebug
  - Peut être utile si on dispose d'un serveur de développement identique au serveur de production
  - Pas le lot commun …
- On est parfois obligé de debuger sur le site en ligne
  - Parce qu'on a pas de copie
  - Parce que c'est pas grave si on le casse
  - Parce qu'il y a urgence
  - Et plein d'autres bonnes (?) raisons…

# Voir sans rien montrer aux visiteurs

- Utiliser les logs dans un fichier :
  - spip_log('toto','debug');
    - Ecrit dans le fichier tmp/debug.log ou tmp/prive_debug.log
    - Permet de tracer sans rien afficher sur le site
- Utiliser les headers
  - header()
  - http://www.firephp.org/
  - FirePHP

```
<?php
FB::log('Log message');
FB::info('Info message');
FB::warn('Warn message');
FB::error('Error message');
?>
```



# Et pour anticiper les bugs ?

En amont…

# Profiler le code

pour aller encore plus loin

avec XDEBUG et un outil d'analyse de log

# Profiler le code

- Analyser tout le déroulement de l'exécution sur une page, de manière statistique
  - Visualiser le graphe d'appel
  - Visualiser les fonctions appelées
  - Visualiser le temps d'exécution de chaque fonction
- C'est le pendant du debug interactif :
  - Dans le debug interactif on zoom sur un morceau de code pour voir ce qui s'y passe
  - Dans le profilage, on exécute tout le code en enregistrant ce qui se passe, et on analyse 'globalement'

# XDEBUG : Profilage

- De la doc
  - http://www.xdebug.org/docs/profiler
- Un outil client pour lire les logs de profilage
  - http://www.maccallgrind.com/ pour Mac OS
  - http://kcachegrind.sf.net/ pour KDE
  - http://sourceforge.net/projects/wincachegrind pour Win
- Une extension FF pour démarrer/arrêter le profilage : XDEBUG Helper
  - https://addons.mozilla.org/fr/firefox/addon/3960

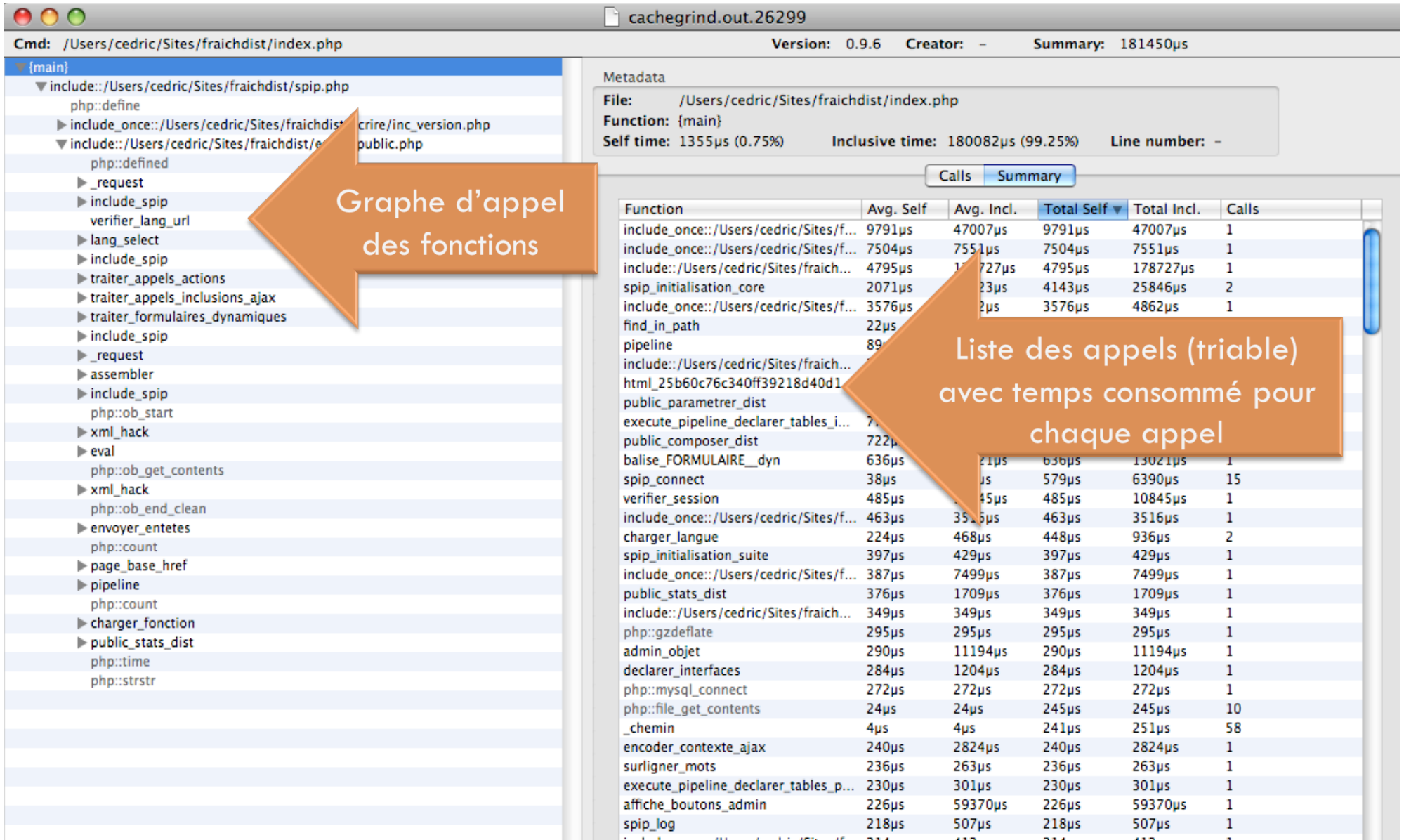
# Configurer le profilage

- Régler la configuration dans php.ini
  - le répertoire où sont stockés les logs de profilage
    - xdebug.profiler_output_dir = …
  - L'activaion du profilage
    - xdebug.profiler_enable = 1
  - La possibilité de trigger le profilage avec l'extension FF, ou un cookie ou un $_GET
    - xdebug.profiler_enable_trigger = 1

# Jouer un scenario

- Dans le navigateur, jouer un scenario
  - Chaque hit apache va donner un fichier de log cachegrind.out.xxxx correspondant aux traces du processus
- Dépouiller avec l'outil d'analyse

# Analyser le profilage (ex de MacCallGrind)



# Outil peu convivial

- Car il est nécessaire de
  - Modifier php.ini mettre en route/arrêter le profilage
  - Jouer un scenario qui alimente les logs
  - Depouiller off line avec un outil
  - Une page SPIP peut lancer plusieurs hits apache (cron, css ou js en skel…), qui génèrent autant de fichier de log. Il est pénible de retrouver le hit principal et de ne pas se mélanger
- Le profiler Zend intégré au Zend Studio est beaucoup plus facile d'utilisation, mais sans donner d'infos supplémentaires in fine
- Outil a utiliser ponctuellement, pour faire des campagnes d'optimisation de performance, mais pas au quotidien
- KCacheGrind semble beaucoup plus convivial que MacCallGrind (A confirmer par un test)

# Analyser la couverture du code

carrément de la science fiction, là

(toujours avec XDEBUG)

# Tester le code

- Avoir des jeux de tests pour une fonction permet de vérifier que la fonction continue à se comporter comme attendue
- Avoir des jeux de tests exhaustifs est souhaitable, mais laborieux
- Quelques tests c'est toujours mieux que rien…

# Encore plus root … direct en php

- xdebug_start_code_coverage([option])
  - option :
    - XDEBUG_CC_UNUSED pour lister le code executable
    - XDEBUG_CC_DEAD_CODE pour analyser aussi le code qui ne peut pas être exécuté (branche morte)
- xdebug_stop_code_coverage()
- xdebug_get_code_coverage( )
  - Renvoie un tableau associatif avec pour chaque fichier inclus un sous tableau
    - N° de ligne => Nombre d'exécution

# A exploiter

- Pour compléter les outils de tests
- par exemple (au hasard)
  - Utiliser des squelettes de la zone pour construire des jeux de tests, et vérifier que l'on a bien couvert tous les cas dans le compilateur
  - Calculer des pages publiques et s'assurer que le service de page couvre toutes les fonctions de ecrire/public/
- Surtout un outil qui peut être utile pour construire des tests, donc

# Exemple

**Example:**

```php
<?php
    xdebug_start_code_coverage();

    function a($a) {
        echo $a * 2.5;
    }

    function b($count) {
        for ($i = 0; $i < $count; $i++) {
            a($i + 0.17);
        }
    }

    b(6);
    b(10);

    var_dump(xdebug_get_code_coverage());
?>
```

# Exemple (suite)

**Returns:**

```
array
  '/home/httpd/html/test/xdebug/docs/xdebug_get_code_coverage.php' =>
    array
      5 => int 1
      6 => int 1
      7 => int 1
      9 => int 1
      10 => int 1
      11 => int 1
      12 => int 1
      13 => int 1
      15 => int 1
      16 => int 1
      18 => int 1
```