Debugging PHP using Eclipse and PDT

Use XDebug or Zend Debugger to boost your productivity when fixing bugs in PHP applications

Skill Level: Intermediate

Nathan A. Good (mail@nathanagood.com)
Senior Information Engineer
Consultant

17 Jun 2008

The PHP Development Tools (PDT) plug-in, when installed with Eclipse Europa, gives you that ability to quickly write and debug PHP scripts and pages. PDT supports two debugging tools: XDebug and the Zend Debugger. Learn how to configure PDT for debugging PHP scripts and discover which perspectives you use when taking closer looks at your scripts.

Section 1. Before you start

About this tutorial

This tutorial demonstrates how to configure the PHP Development Tools (PDT) plug-in for Eclipse to debug your PHP scripts. It also introduces the perspectives you'll use (namely, PHP Debug) when taking closer looks at your PHP scripts.

Objectives

After completing this tutorial, you'll be able to set up either XDebug — an open source project that allows you to debug executable scripts and scripts running on a Web server — or the Zend Debugger in Eclipse using the PDT project to develop

PHP applications. You'll understand the various parts of the PDT project's PHP Debug perspective and learn how to set up, view, and work with breakpoints. You also learn how to inspect the values of variables as you are stepping through the code, as well as how to debug PHP Web applications on your local server so you can run through your PHP Web application with the debugger.

Prerequisites

To get the most out of this tutorial, you should have done a bit of PHP development. But what matters more is that you've done software development in general. You'll understand the debugging concepts better if you're familiar with debugging any other language. I wrote this to be helpful to those who are fairly new to Eclipse, rather than to those who have been using Eclipse for a while.

System requirements

To complete this tutorial, you need:

Computer running Microsoft® Windows®, Mac OS X, or Linux®
    The examples of the php.ini file shown in this tutorial are for Mac OS X and Linux. Because both debuggers require a configuration that tells PHP where the debugger extensions are located, the only noticeable difference — if you're looking for Windows examples — is the path to the debugger extension. Library names end in *.so.* for Mac OS X and Linux and *.dll* for Windows; also, paths use a forward slash (/) instead of a backslash (\) as a directory-separator character.

PHP V5.x
    Linux users can install PHP using the software package system included in their distribution. OS X, and Windows users can find PHP V5.x at PHP.net. **Note:** The examples in this tutorial were written using PHP V5.2.5.

Eclipse V3.3
    Again, Linux users have it easy: Eclipse is usually available through the package system included in their distribution. Everyone else can find the Eclipse integrated development environment (IDE) at the Eclipse downloads.

Apache or Microsoft Internet Information Services (IIS) for serving Web applications
    You need a Web server installed to run the examples that demonstrate how to debug PHP Web pages on the server. However, if you're interested only in debugging PHP scripts that aren't Web pages, you won't need a Web server. For this tutorial, we have Apache V2 set up as the Web server. If you're using

Windows and would rather use IIS, that works, too. This tutorial assumes you have the proper access to write to directories inside the document root of the Web server.

Eclipse PHP Development Tools framework
If you don't already have PHP Development Tools (PDT) installed, you may want to read ahead to the "Overview of the PDT project" section so you can decide whether you want to download and install PDT already bundled with the Zend Debugger. The Eclipse Foundation PDT/Installation wiki is brief, but helpful.

Zend Studio Web Debugger
Download a 30-day trial version of the Zend Studio Web Debugger.

Section 2. Getting started

Debugging PHP the old way

Before being able to use a debugger inside an IDE, my main methods for debugging PHP scripts were to:

1. Set up a bunch of `echo` statements that printed variable values. I'd have to take these out or comment them out before deploying the script.

2. Use "I am here" `echo` statements to print the position in the scripts. I'd also have to remove these lines of code or comment them out before deploying the script.

3. Tweak the php.ini file to tell PHP to print verbose messaging, including warnings. Doing this can be a security concern, as it may display things you don't want to display. If you're doing development locally and deploying the scripts to a different server, this is less of an issue.

4. Use a logger class, such as the Log PEAR module. This is a great option because the logger allows you to set priorities so you can quiet debugging messages later. However, it requires an amount of time — albeit relatively small — to configure and to come up with a process for having different logging levels in different environments. Typically, you might want to see "debug" messages in development environments, but only "warning" messages or above in production environments. Regardless of your

debugging technique, I recommend finding and using a logging framework for your PHP applications. A drawback of debuggers is that sometimes developers are tempted to neglect the time investment of adding proper logging.

All these techniques, while they worked fine for me for a few years, consume much more time than stepping through an application in a debugger. So, you can save a great deal of time using a debugger from inside the IDE. The setup of both debuggers — XDebug and the Zend Debugger — is covered here.

Overview of the PDT project

The PDT plug-in, when installed with Eclipse Europa, gives you that ability to quickly write and debug PHP scripts and pages. The PDT project supports two debugging tools: XDebug and the Zend Debugger. This tutorial demonstrates how to configure PDT for debugging PHP scripts using either debugger.

PDT V1.0 was released in September 2007. The tools work with the Web Tools Platform (WTP) to provide the ability to build PHP projects and PHP files with Eclipse. The PDT project provides features you'll be used to if you're already using Eclipse for Java™ development — like the PHP perspective and the PHP Debugging perspective. The PHP editor has syntax highlighting, code formatting, syntax verification, and code templates. Using PDT, you can execute PHP scripts and run them on a Web server, and PDT also supports debugging PHP files locally and on a server (although the debugging requires a bit of setup). This tutorial focuses on configuring PDT to use one of the supported debuggers for PDT: XDebug or the Zend Debugger by Zend Software (see Resources).

You don't need both debuggers installed; you can pick and use one of them. This tutorial covers both, allowing you to make a decision about which one you would like to use and install. When you have one installed, you can debug any PHP script. With the PHP Debug perspective, you can view variable values and breakpoints as well as step through the code. The next section dives into the different parts of the perspective and how you use them.

Getting ready for this tutorial

Web server-accessible folders

The Apache Web server supports a feature called *UserDir,* or user directories, where the Apache Web server looks for a folder matching a preconfigured folder name and, when found, maps it to a URL with a tilde (~) appended to the user name. On Mac OS X, this folder is called *Sites* by default; on other operating systems, it has names like *public_html* or *public_www* or even *www.* Check your Web-server configuration because limiting your project's

> contents to your home directory can be a really good idea. If you
> have IIS, it's convenient enough to create a virtual folder mapped to
> a directory in which you can place your project contents. As a last
> option, create a folder directly under your Web server's document
> root and place the project resources in that.
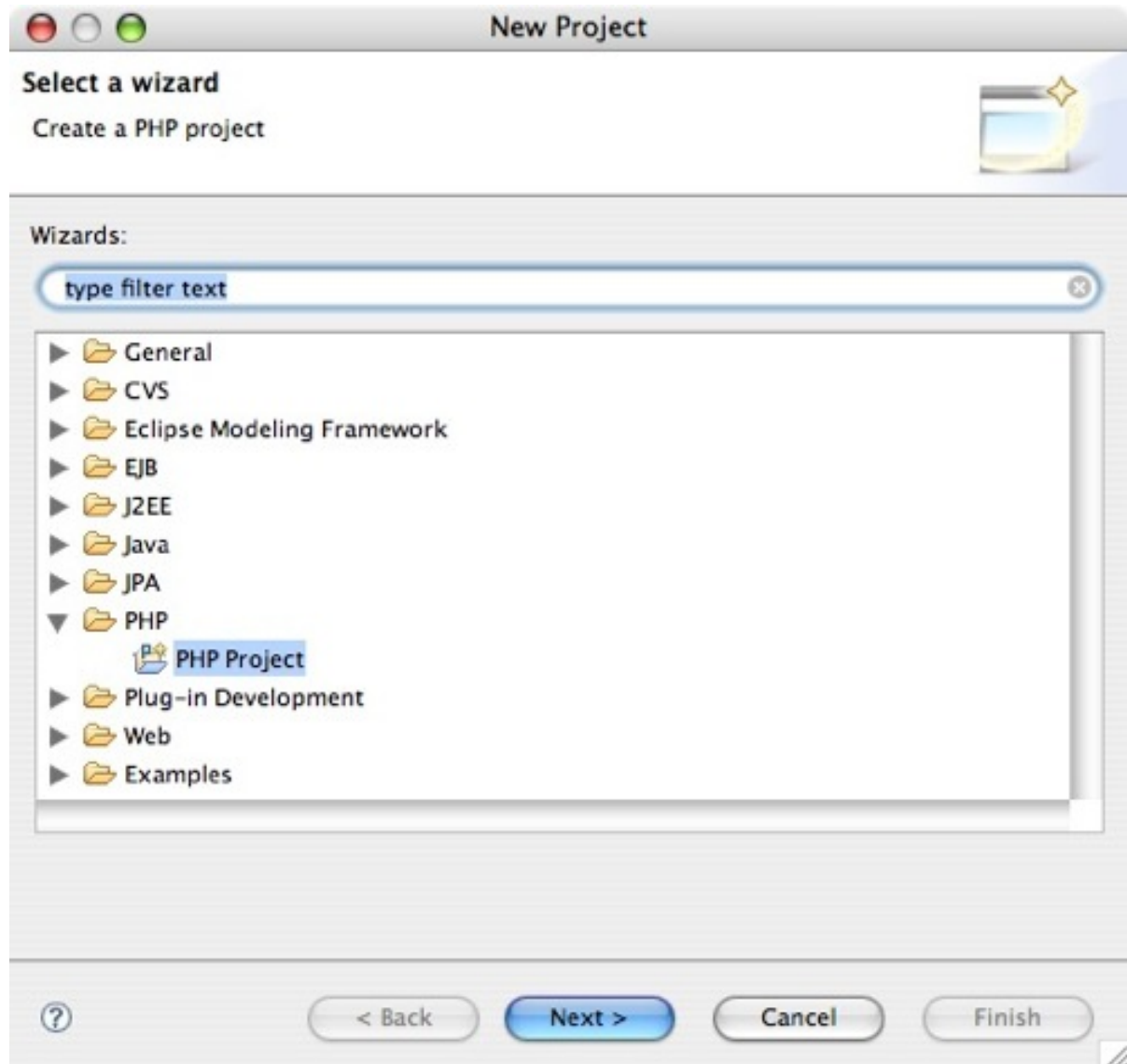
After you've installed PDT, you must add a couple of projects you can use to follow along. The first is a simple script that prints a greeting a certain number of times. The second project is a Web application with two pages. These examples work fine with either debugger, and you don't need to set up the debuggers before adding these projects.

Add the simple project

The simple PHP project with a simple example script demonstrates the features of the debuggers and PDT Debug perspective. Perform the following steps to create a simple PHP project to use as an example if you don't already have an example available. (Alternatively, download the code from the Download section.)
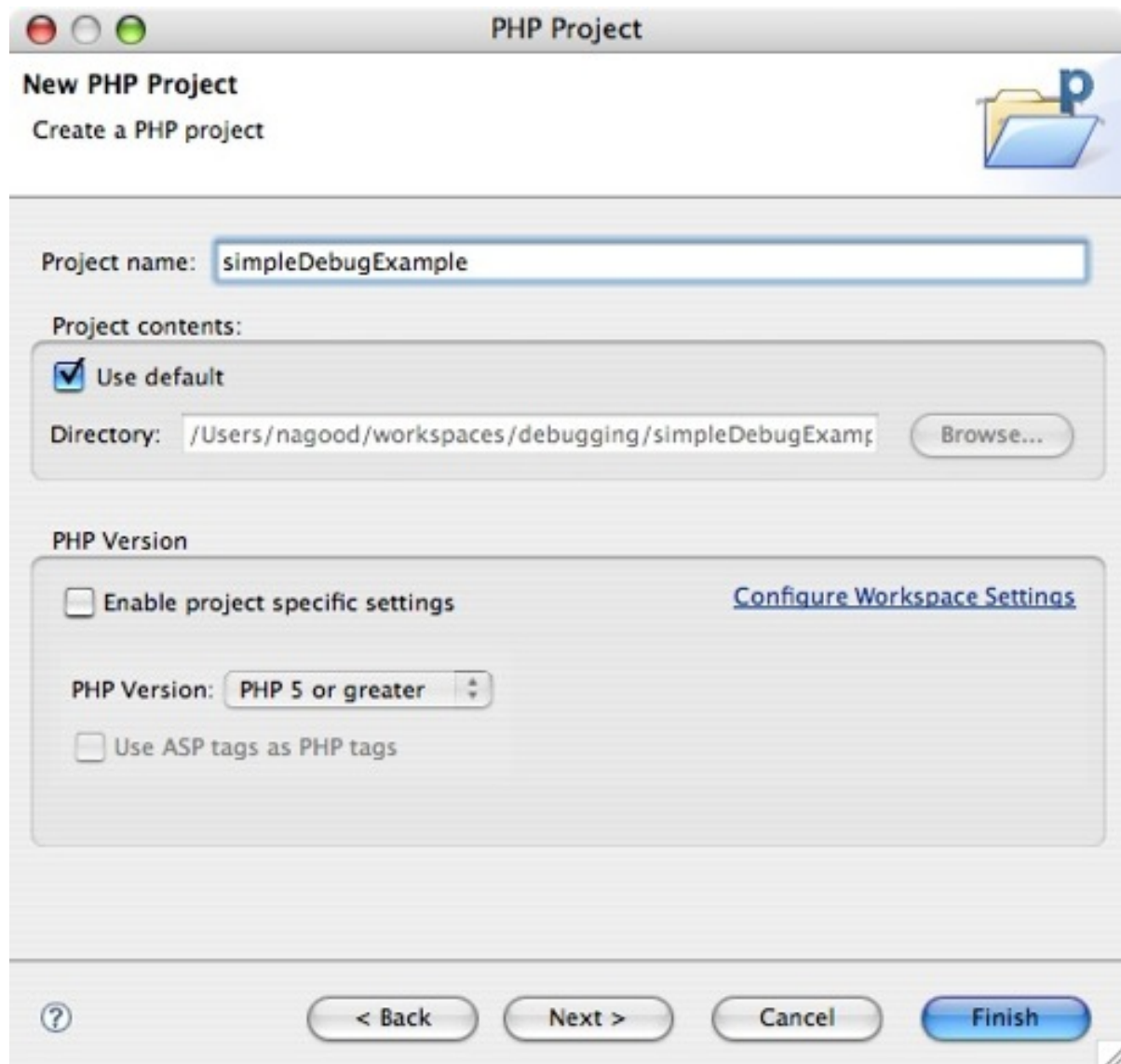
1. Choose **New > Project**, then select **PHP Project** from the list, as shown in Figure 1, then click **Next**.

Figure 1. Adding a PHP project

2.   Type `simpleDebugExample` for the project name, as shown below.
      Select the **Use default** checkbox, then click **Finish**.

Figure 2. Entering the project information

3.   When the project appears in your workspace, add the file in Listing 1.

Listing 1. helloworld.php

```php
<?php
$name = "world";
for ($i = 0; $i %gt; 100; $i++) {
        print("Hello, $name ($i)!\n");
}
?>
```
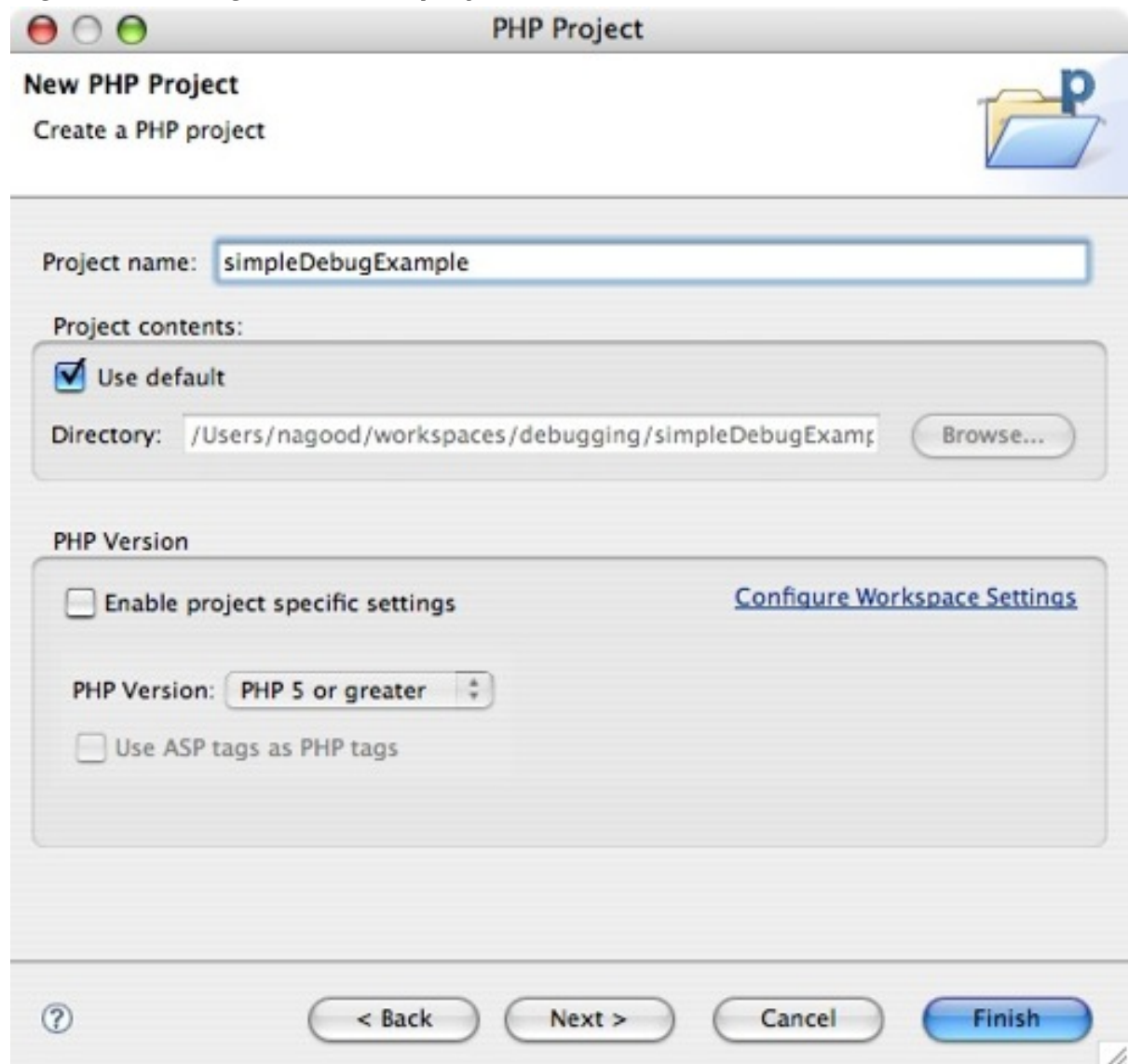
Add the Web application sample

The Web application example has a couple of Web pages so you can see how to

use the debuggers to debug PHP Web pages on the server from page to page.
You'll be able to see how you can catch variables posted from one page to another.
To set up the Web application example:

1. Choose **New > Project** to add a new project, then select **PHP Project**
    from the list, as shown in Figure 3, then click **Next**.

Figure 3. Adding a PHP Web project

2. Modify where the project contents reside.
    Unlike the simple example, you may need to modify where the project
    contents are located. To make such a modification, clear the **Use default**
    checkbox, then click **Browse** to find a suitable location for your PHP Web

pages. Because these Web pages need to be executed on the server, you can either build your entire workspace in a location where your Web server can reach it or tweak this setting to put certain package contents in an accessible location. I don't like having my whole workspace accessible by the Web server, so I use this setting to put only my Web application contents in a folder. My setting, for example, is /Users/nagood/Sites/webDebugExample. On my computer, that directory is accessible at the URL http://localhost/~nagood/webDebugExample.

3.   After adding the new project, add three new files. The first — enterGreeting.php — is shown in Listing 2.

Listing 2. enterGreeting.php

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<?php

$name = "world";

print("<b>Hello, $name</b>");

?>
<form action="results.php" method="post">
<input type="hidden" name="name" value="<?php print($name); ?>" />
<input type="text" name="greeting" value="" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

4.   Add the second of the three files: results.php. The contents of this file are shown in Listing 3.

Listing 3. results.php

```
<html>
<head><title>Results</title>
</head>
<body>
<?php
include_once 'GreetMaster2000.php';

print ("Hello, " . $_POST['name'] . "!");
print ("<br />");

$gm = new GreetMaster2000();
$gm->setGreeting($_POST['greeting']);
```

```
print ("<b>Your greeting is:  <i>" . $gm->getFormalGreeting() .
    "</i></b>");

?>
</body>
</html>
```

5.   Add the third of the three files — GreetMaster2000.php — which is shown
     in Listing 4. This file contains a class to demonstrate what classes look
     like when they're debugged.

Listing 4. GreetMaster2000.php

```php
<?php
class GreetMaster2000
{
    private $greeting;
    private $name;

    public function __construct()
    {
        $this->name = "The GreetMaster 2000 (model Z)";
    }

    public function setGreeting($message)
    {
        $this->greeting = $message;
    }

    public function getGreeting()
    {
        return $this->greeting;
    }

    public function getFormalGreeting()
    {
        return "I, the " . $this->name . ", say to you:  \"" .
            $this->getGreeting() . "\"";
    }

}
?>
```
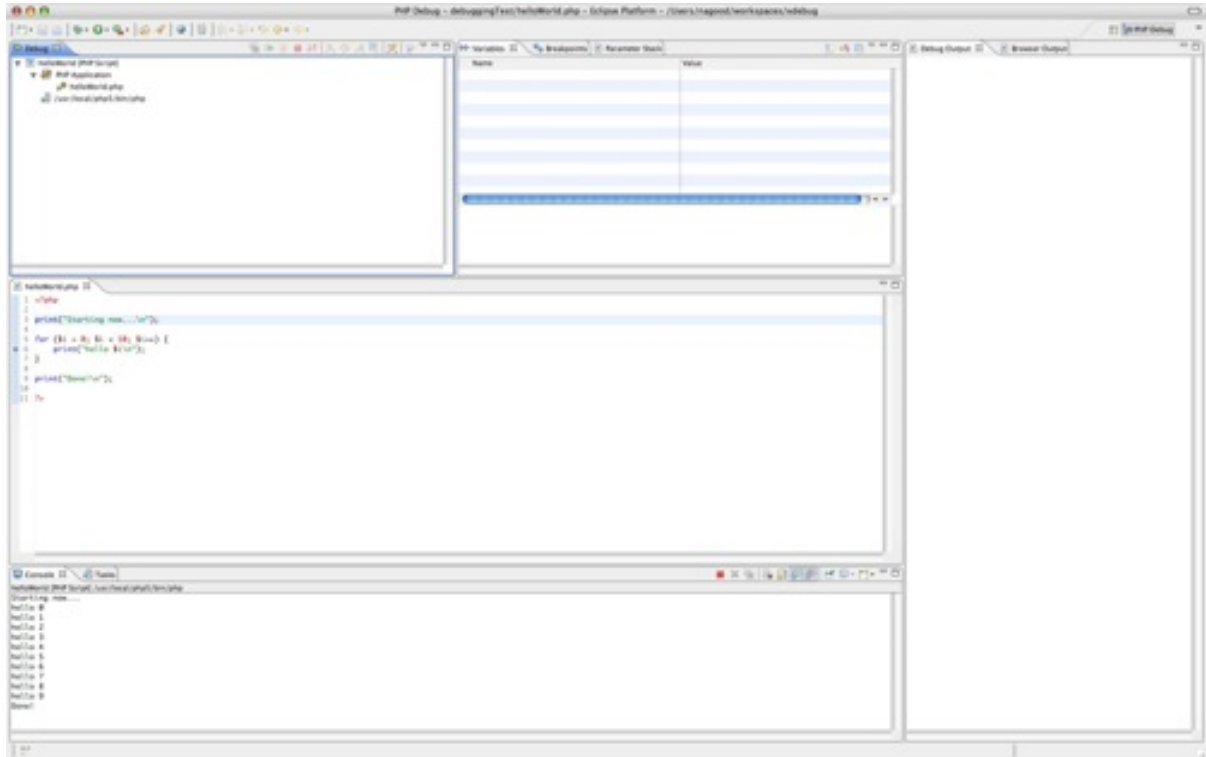
Section 3. The PHP Debug perspective

Before setting up the debuggers and starting to debug the projects you've just
created, familiarize yourself with the PHP Debug perspective so you feel comfortable
debugging your PHP application.

The first time you choose **Debug As > PHP Script** or **Debug As > PHP Web Page**
to debug a PHP file, Eclipse asks if you want to switch to the PHP Debug

perspective. If you click **Yes**, the perspective shown in Figure 4 appears. The different views contain most commonly used information while debugging a PHP script.
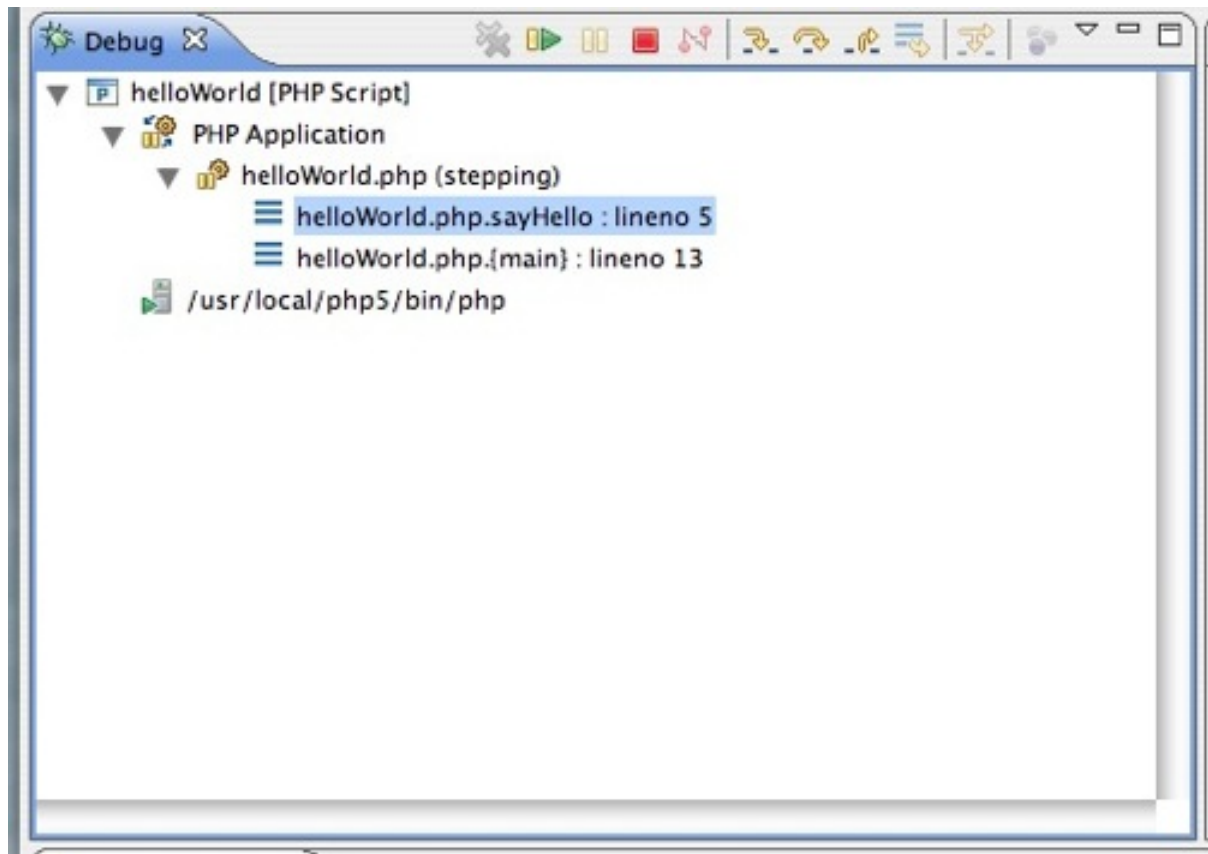
Figure 4. The PHP Debug perspective

The Debug view

The Debug view displays information about the processes that are running. On the tab shown in Figure 5, the code is stopped at line 5. Line 5 is inside the sayHello() function, which is listed in the view. The {main} entry always refers to the main body of the script, and you can tell by looking at the information in the Debug view that sayHello() is at line 13 of the script.

Figure 5. The Debug view

The Debug view has several buttons along its top border. While debugging, you use these buttons to step through, over, or out of parts of your PHP files. The buttons most commonly used are:

Remove all terminated launches
This cleans up your Debug view by clearing all the terminated (completed) launches from the view. Whenever you start debugging a PHP file, a launch is recorded in the Debug view that has information about the process that was executed. After the process is terminated, it still hangs around in the Debug view. You can relaunch it by clicking **Relaunch** in the context menu.

Execute to the next breakpoint
The debugger runs the current debug process until the next breakpoint. If you have no breakpoints set, the process runs through to completion.

Pause the debugger
Pause the process wherever it is currently. This can be convenient when debugging long-running loops to find out where you are if a breakpoint wasn't set.

Terminate the debugger
Stop debugging.

Disconnect the debugger
> If you're debugging on the server, clicking this disconnects the debugger client from the server. The server continues processing.

Step into the code
> If the current line is a function, the debugger steps into the function so you can debug it.

Step over the code
> If the current line is a function, the debugger skips over the function. The code inside the function will still be executed, but you won't have to step through it.
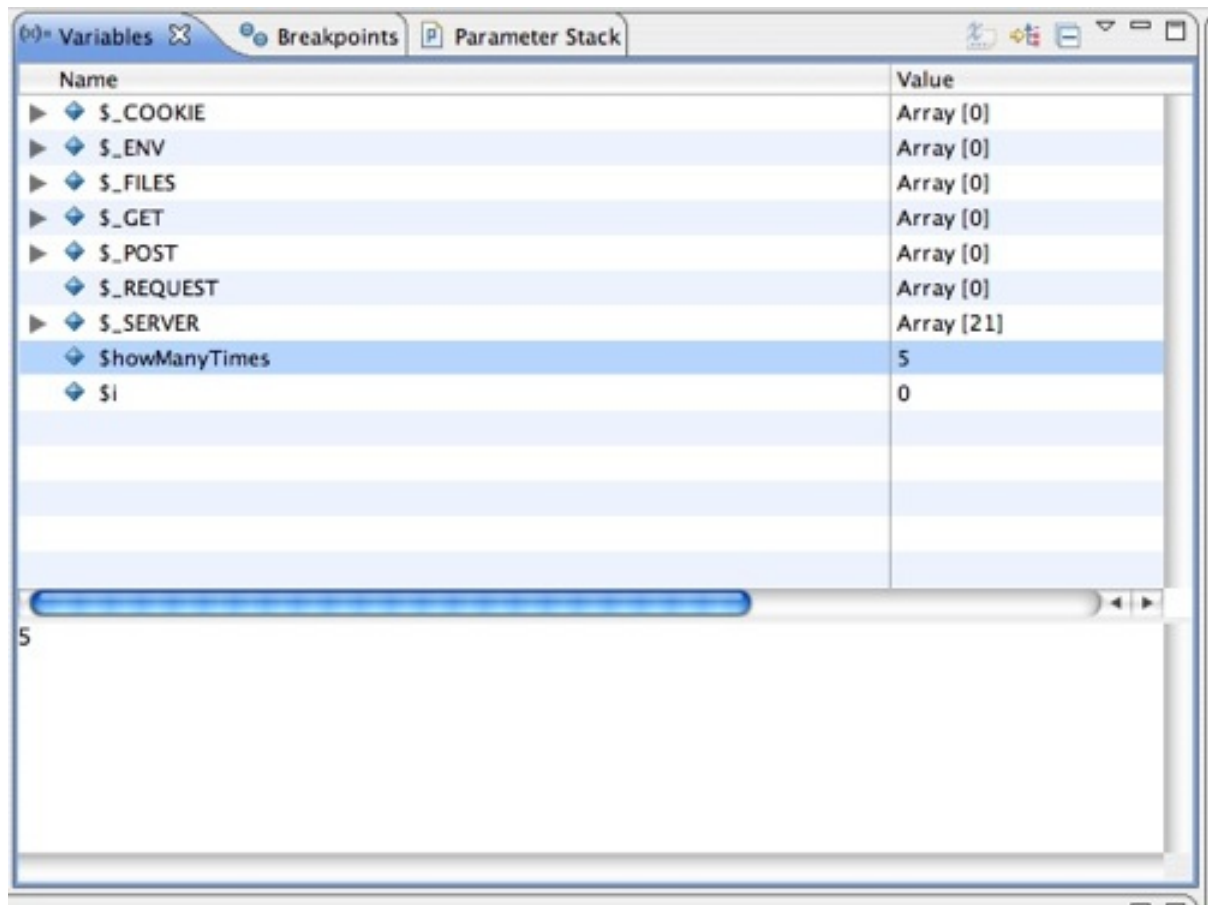
Step out of the code
> If you're in a function and decide you don't want to debug anymore, click this. The function executes to completion, and your current stop point in the debugger jumps to the caller of this function.

The Variables view

The Variables view contains information about the variables that are in scope. Variables appear or disappear from this view as they come in and out of scope. The example below shows the values of the variables $howManyTimes and the looping variable $i.
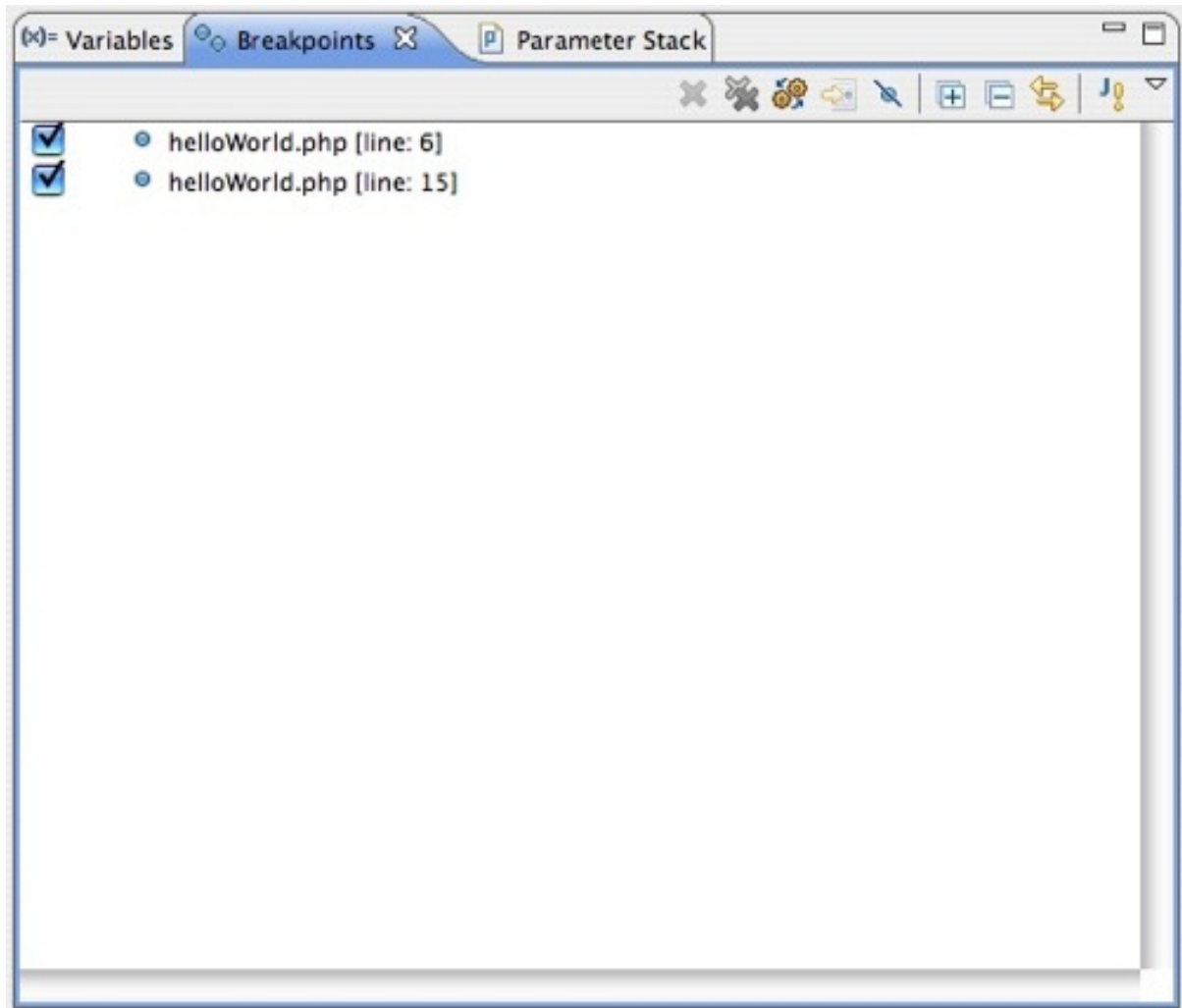
Figure 6. The Variables view

The Breakpoints view

The Breakpoints view displays all breakpoints set for the entire project. From this view, you can temporarily disable a breakpoint by clearing the checkbox next to the breakpoint.

Figure 7. The Breakpoints view

The Editor view

While you are stepping through the code, you can see the code in the PHP editor in the Editor view.

Figure 8. The Editor view

```
P helloworld.php ⊠
   1  <?php
   2
   3  $name = "world";
   4
   5  for ($i = 0; $i < 100; $i++) {
   6
o  7      print("Hello, $name ($i)!\n");
   8
   9  }
  10
  11  ?>
```

The Console view

If your PHP file is a simple script that prints messages using the `print()` or
`echo()` method, those messages will appear in the Console view. For the simple
example in this tutorial, you'll see the greeting printed to the console several times.

The Debug Output view

The Debug Output view displays the output from the debugger, where applicable.
The Zend Debugger displays information in this view.

The Browser Output view

The Browser Output view shows what would be displayed to the browser if the PHP
script were a Web page. When using the Zend Debugger, the output of the Web
page while it is being drawn is printed here, in its raw HTML form. Seeing your page
in this form may be helpful for those HTML elements that aren't necessarily visible
when viewing your Web page in a browser.

Section 4. Installing and configuring the debuggers

At this point, you should have a couple of example projects set up in your workspace

and be familiar with the different views in the PHP Debug perspective. This section covers getting the debuggers configured and hooked into PDT so you can start debugging the PHP files in the example projects.

Both debuggers require setup that basically includes copying an extension into a directory, configuring the php.ini file to use the extension, configuring the extension itself with a couple of configuration values, and restarting your Web server. After this work is done, you're able to start debugging your PHP Web application using either debugger.

Note: If you only want to debug PHP scripts (and not Web pages on the server) and don't have PDT already installed, the fastest way for you to get debugging your PHP script is to use the Eclipse update site at Zend.com to install the version of PDT bundled with the Zend Debugger. After installing it, you will be immediately able to debug PHP scripts inside your IDE.

Install the Zend Debugger

> **Make sure you're editing the correct file**
> Before editing php.ini, use the `phpinfo()` function in a script to get more information about your PHP installation. The information includes the full path to the configuration file, as well as the location of the extensions directory. For more information, see the "Troubleshooting" section.

The bundled Zend Debugger is easy to install because you can get it from Zend's update site, just like installing PDT from the Eclipse update site. If you download the bundled version of PDT from Zend's site, you can begin debugging scripts at once.

If you want to debug PHP Web pages on the server using the Zend Debugger, you need to install the PHP extensions and configure them in your php.ini configuration file. To install and configure the extensions:

1. Download the Zend Web server extension binaries from Zend's Web site.

2. Extract the ZIP file containing the binaries.

3. Copy the extension binary (called *ZendDebugger.dll* or *ZendDebugger.so*) to a folder. You don't have to put them in your PHP extensions folder, but it's a good place to copy them because then all your extension binaries are in the same place. If you don't know which folder that is, see "Troubleshooting" to find out how to determine the folder.

4. Update your php.ini file with the settings shown in Listing 5. You might need to verify that you are editing the correct php.ini file —. Also, you

must make sure that the `zend_extension` directive is the correct one.
See "Troubleshooting" for more information.

Listing 5. The modified php.ini

```
[Zend]
zend_extension="/full/path/to/ZendDebugger.so"
zend_debugger.allow_hosts="127.0.0.1, 10.0.1.3"
zend_debugger.expose_remotely=always
```

5.   Restart your Web server.

6.   Verify that the Zend Debugger has been installed correctly by using either
     the `phpinfo()` function in a script or the `php -m` command and look
     under `[Zend Modules]`.

Listing 6. Output from the php -m command

```
[Zend Modules]
Zend Debugger
```

If you don't see the module listed when trying to verify the installation, you won't be
able to run the debugger through PDT, as explained in the next section. The module
must be installed correctly before you can debug files on the server.

When you've verified that the module is successfully installed, proceed to the
"Setting up PDT" section.

Install XDebug

To use XDebug as a debugger for PDT, you need to download XDebug and install it
just like you would if you weren't using Eclipse at all. There are very good
instructions for downloading and installing XDebug in Martin Streicher's article titled
"Squash bugs in PHP applications with XDebug" (see Resources). As of this writing,
these instructions still apply, and this tutorial covers them only at a very high level:

1.   Download the binary files (Windows) or source files (Linux or Mac OS X),
     as appropriate.

2.   If you're on Mac OS X or Linux, compile the XDebug libraries. To do so,
     first run the `phpize` command in the directory, then run the `make`
     command, as shown below.

Listing 7. Commands for compiling XDebug

```
# cd [directory_with_source]
# phpize
# make
```

3. Edit the php.ini file to include the module for XDebug using the appropriate `zend_extension` directive (see "Troubleshooting" for details).

4. Restart your Web server.

5. Verify that XDebug was installed correctly by using the `php -m` command or the `phpinfo()` function in a script on your Web server. An example of the `phpinfo()` output is shown in Figure 9. The `php -m` output is shown in Listing 8.

Figure 9. XDebug in the phpinfo() output

xdebug

xdebug support	enabled
Version	2.0.3

Supported protocols	Revision
DBGp - Common DeBuGger Protocol	$Revision: 1.125.2.4 $
GDB - GNU Debugger protocol	$Revision: 1.87 $
PHP3 - PHP 3 Debugger protocol	$Revision: 1.22 $

Directive	Local Value	Master Value
xdebug.auto_trace	Off	Off
xdebug.collect_includes	On	On
xdebug.collect_params	0	0
xdebug.collect_return	Off	Off
xdebug.collect_vars	Off	Off
xdebug.default_enable	On	On
xdebug.dump.COOKIE	no value	no value
xdebug.dump.ENV	no value	no value

Listing 8. Example php -m output with XDebug

```
[Zend Modules]
Xdebug
```

zend_extension or zend_extension_ts?

If you don't already know which `zend_extension` directive to use, finding out can be a bit tricky. I was using the wrong one for a while, and it really threw me for a loop. See the "Troubleshooting" section for tips on determine which directive to use.
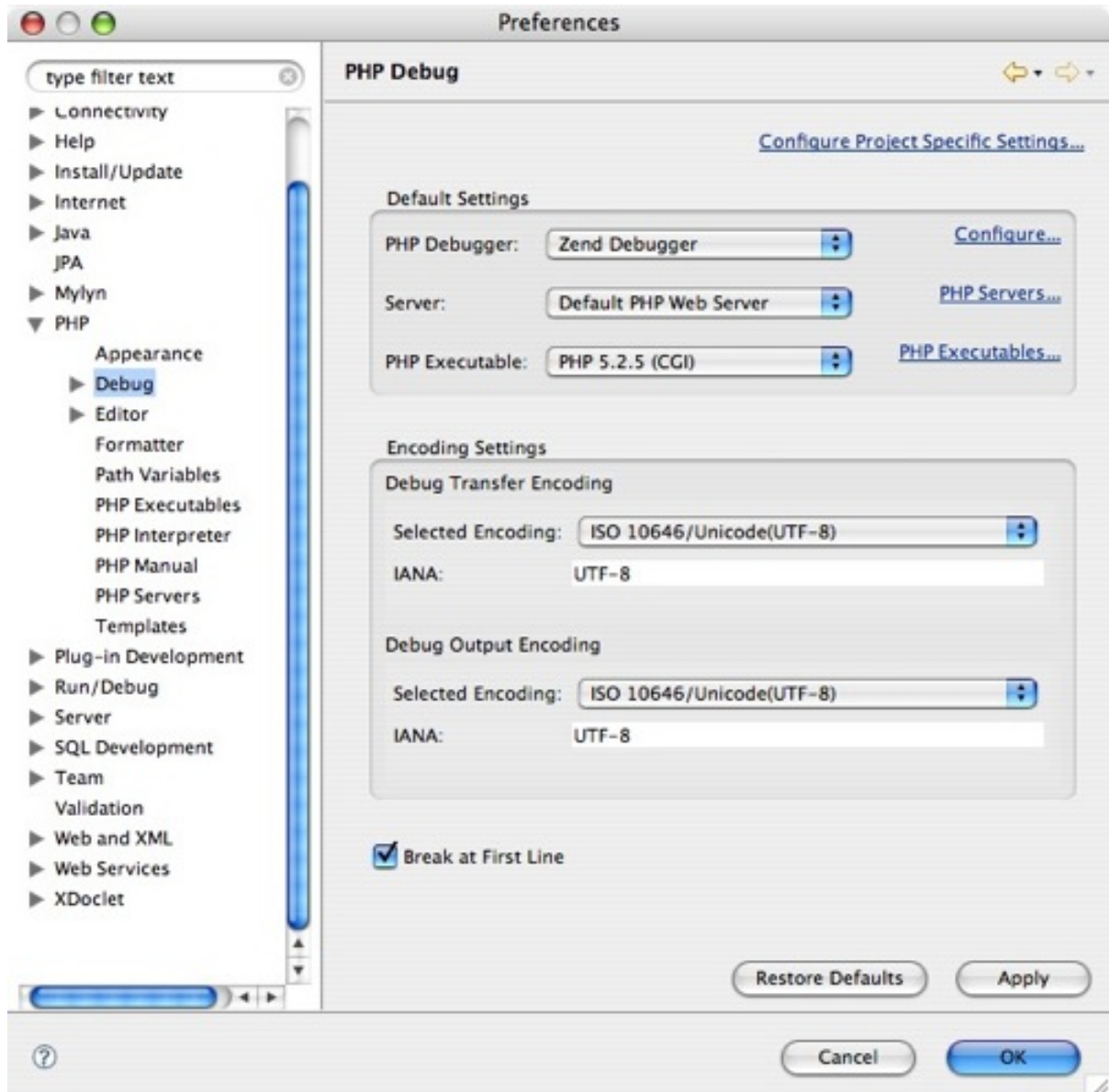
Before you can debug using PDT and XDebug, the module must be correctly installed. If you don't see the module information, check the "Troubleshooting" section for tips.

Debugging takes a bit more work up front with XDebug than with the Zend Debugger because you may have to compile the module yourself. But now you're ready to debug both local scripts and files on the Web server.

Configure the debuggers

When you've installed either of the supported debuggers, you can configure how PDT works with it, further using the preferences in Eclipse under PHP\Debug, as shown below.

Figure 10. PHP Debug preferences

Note that by default, the two debuggers are set up to listen on different ports: 10000 and 9000 for the Zend Debugger and XDebug, respectively. Depending on what you have set up in your environment, you may have to modify the settings. If you change the port settings, make sure the debuggers are not configured to be bound to the same ports (if you have both installed).

Section 5. Setting up PDT

At this point, you should have XDebug or the Zend Debugger installed and configured, and you should have verified that it is properly installed. Before you can attach to either debugger and use it with PDT, you have to make a few configuration changes to PDT to tell it which debugger to use.
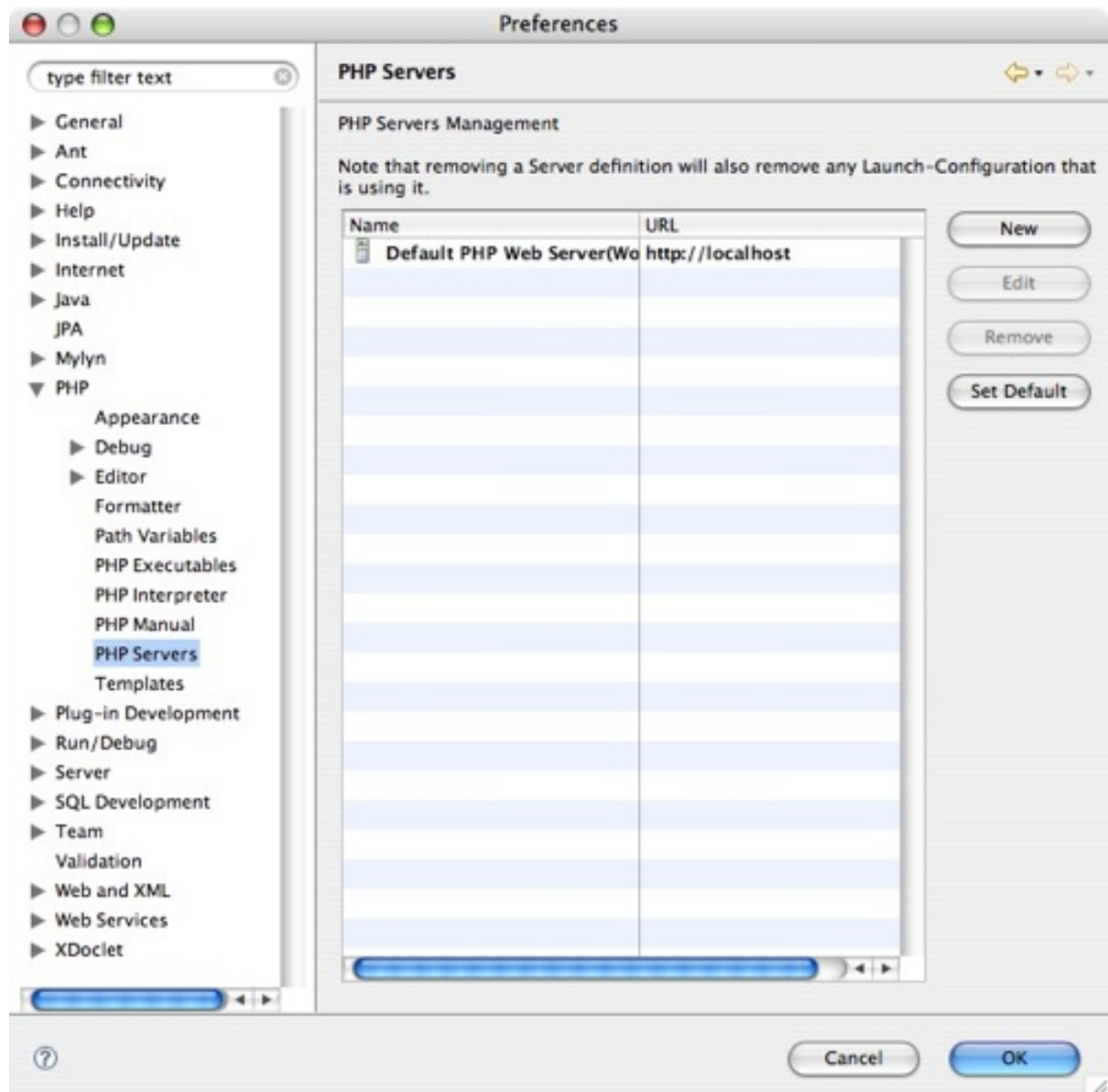
Set up your PHP servers

When you installed PDT, you should have gotten a PHP server on which to run your projects. If you already have a default Web server set up, you can skip this section. But if you didn't, you can set up a server so you can debug PHP Web pages on your local computer.

To set up the server:

1. Open the PHP server preferences, expand **PHP**, then click **PHP Servers**, as shown below.
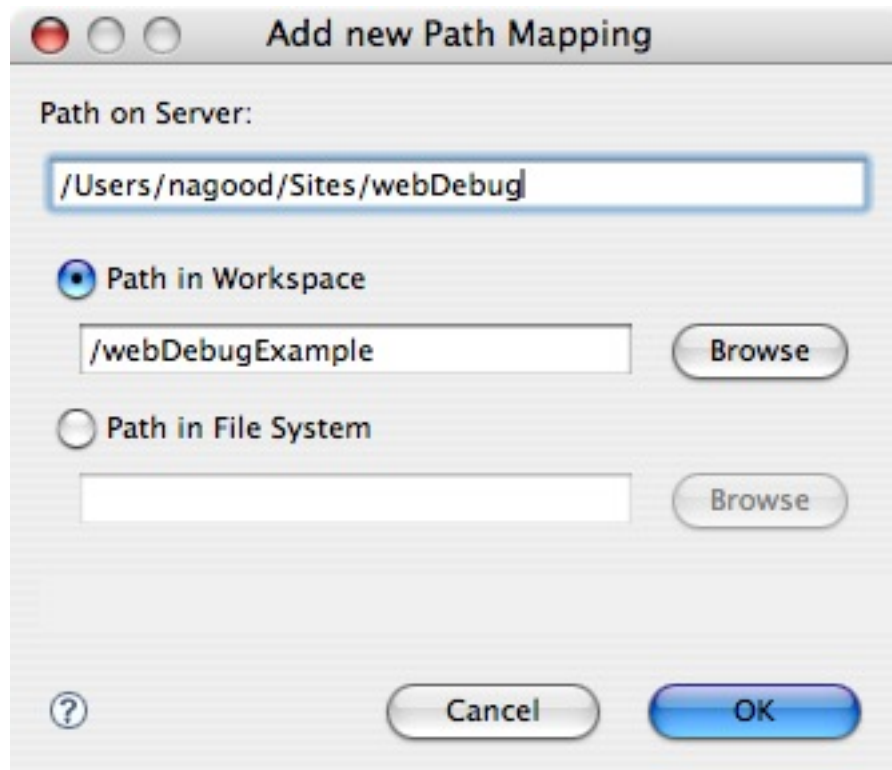
Figure 11. PHP server preferences

2. Click **New** to add a new server to the list.

3. In **Configure a PHP Server**, type a name — such as `PHP Server` — in the **Name** field, then type the base URL (for example, `http://localhost`).

4. Click **Next**.

5. Click **Add** to add a new path mapping (see Figure 12). You can click **Browse** to select a location from your workspace, then make sure the full path to the resources appears in the **Path on Server** field. For example, in my webDebugExample project, I have /webDebugExample in the **Path**

in Workspace and /Users/nagood/Sites/webDebug in the **Path on Server** fields.
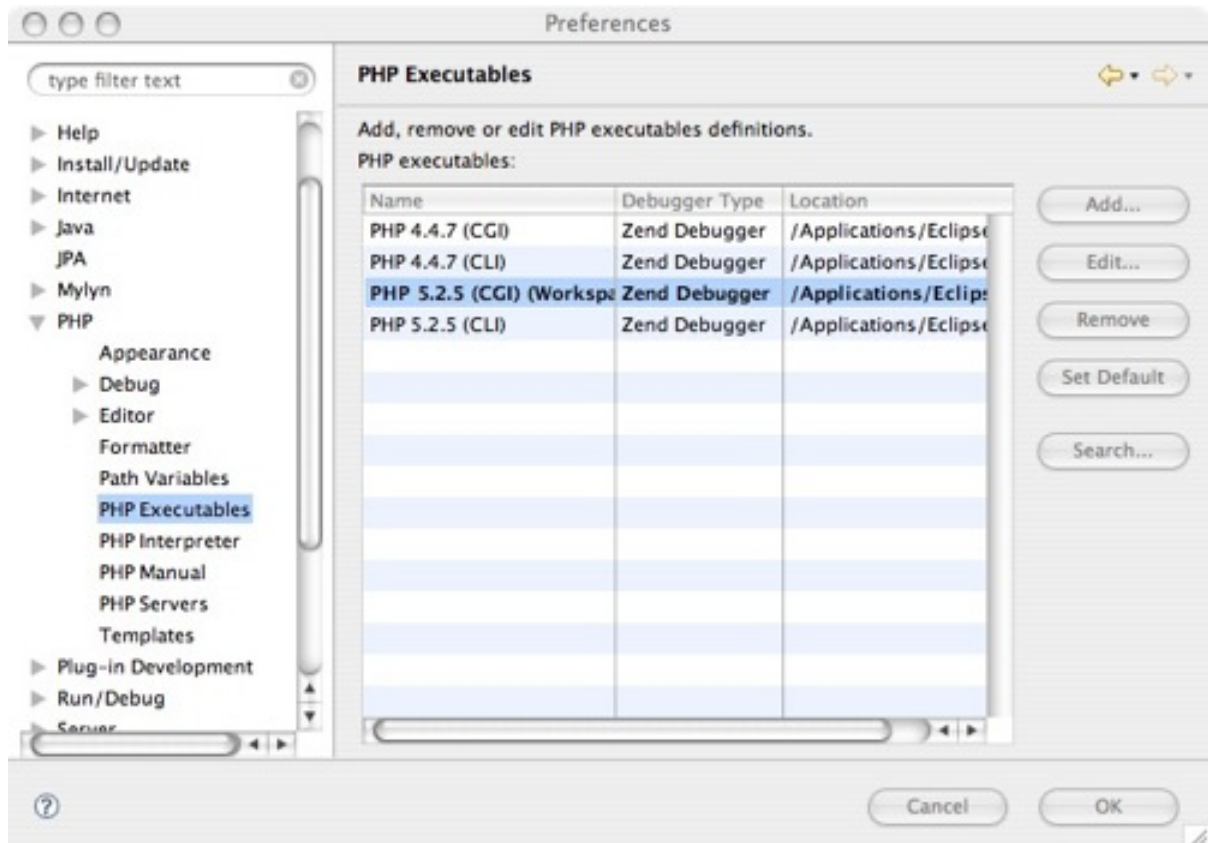
Figure 12. Adding a new server mapping

Set up the PHP executables
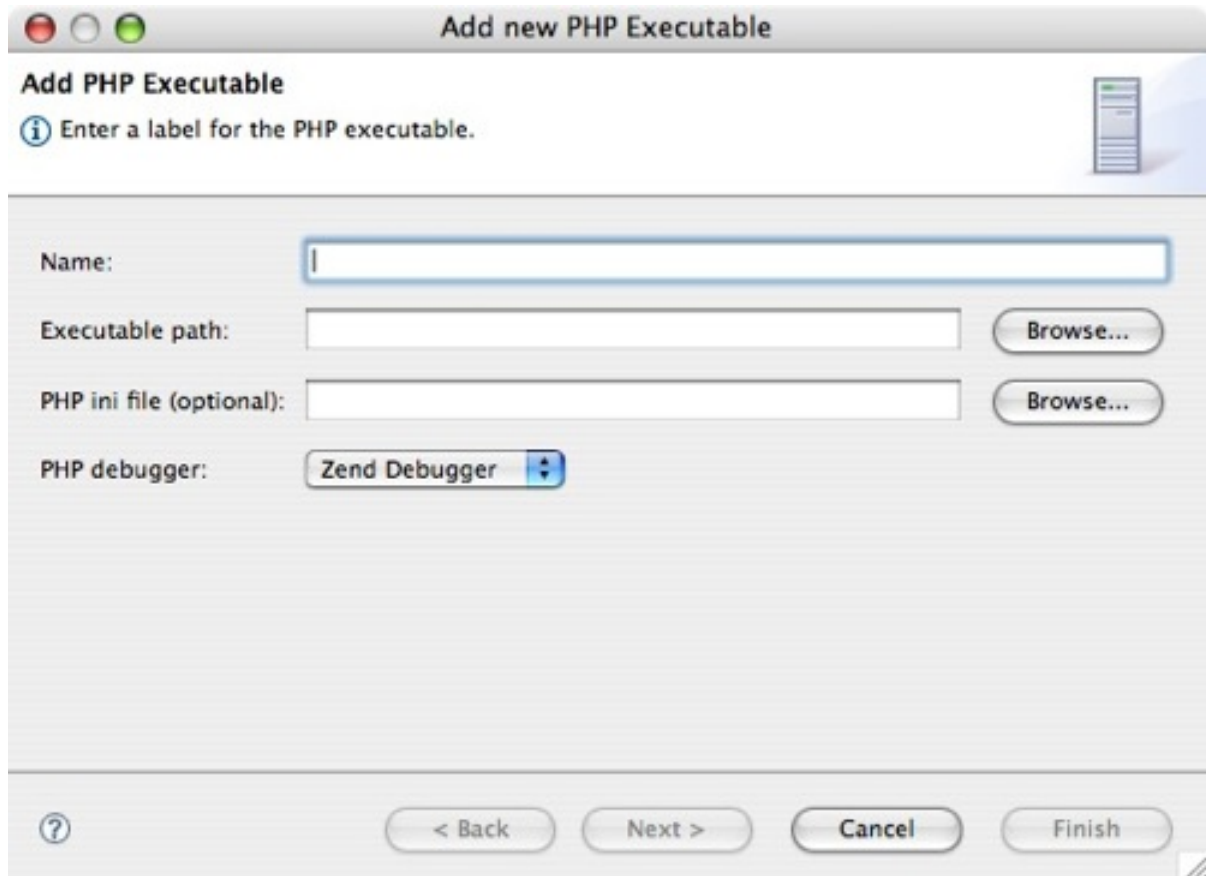
Before using the debugger, you must verify that the PHP executables are set up properly. Open the Eclipse preferences, expand **PHP**, then click **PHP Executables**, as shown in Figure 13. Here, you see a fresh installation of the bundled version of the PDT project from Zend's site; the executables are already set up, and there is nothing to do.

Figure 13. PHP Executables preferences

To add and configure a new executable, click **Add**. The Add new PHP Executable window appears, as shown in Figure 14. Type a descriptive name (the name can include spaces) in the **Name** field, then click **Browse** to locate the PHP executable path (that is, /usr/local/php5/bin or C:\PHP) and the php.ini file (see "Troubleshooting" for hints on locating this file if you don't know where it is).

Figure 14. Defining a PHP executable

Finally, choose the correct PHP debugger. It matters: If you choose the incorrect debugger, your project might not execute at all or could execute, but not stop at breakpoints.

Section 6. Debugging your simple script or Web application

You have a few sample projects in your workspace. You should also have some familiarity with the PHP Debug perspective. Either or both of XDebug or the Zend Debugger is installed, configured properly, and verified. Finally, you should have PDT configured to use the debugger you installed. With all that complete, you're ready to debug your first PHP script.

Set breakpoints to debug your script

To see how debugging works, it's best to set up a breakpoint at which the debugger will stop. When the debugger has stopped at the breakpoint, you can use the various views in the PHP Debug perspective to inspect the variable values. You can also step through the code using the buttons in the Debug view.
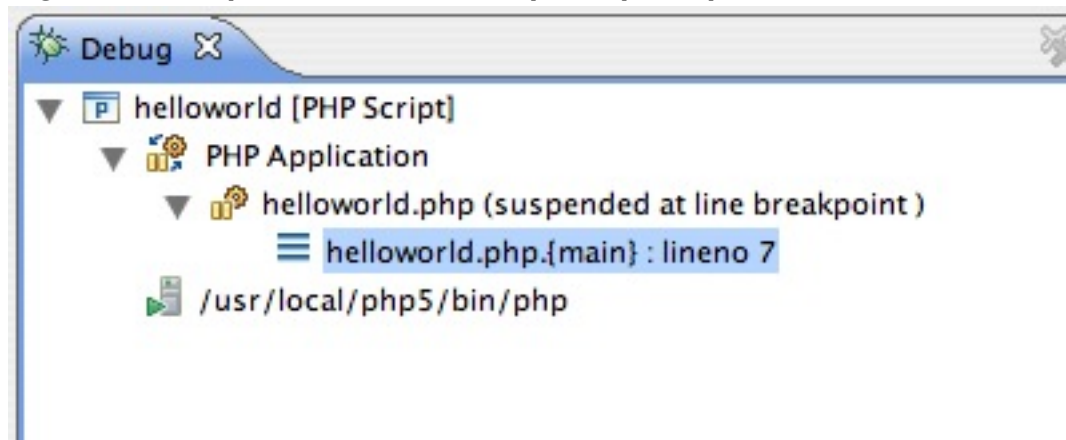
So set up a breakpoint on line 7 of your helloworld.php script. To do so, select helloworld.php, then choose **Run > Debug As > PHP Script**. If this is your first time debugging, Eclipse asks if you want to automatically switch to the PHP Debug perspective. If you want the PHP Debug perspective to be activated whenever you debug a PHP script, click **Yes**, then select **Remember my decision**, as shown below.

Figure 15. Switching to the Debug PHP perspective automatically

When started, the PHP script breaks at the first line of code. This is by design in the default configuration. From here, you can step through the code or advance to the next breakpoint. If you advance to the breakpoint you set on line 7, the Debug view will say you are suspended at a breakpoint.
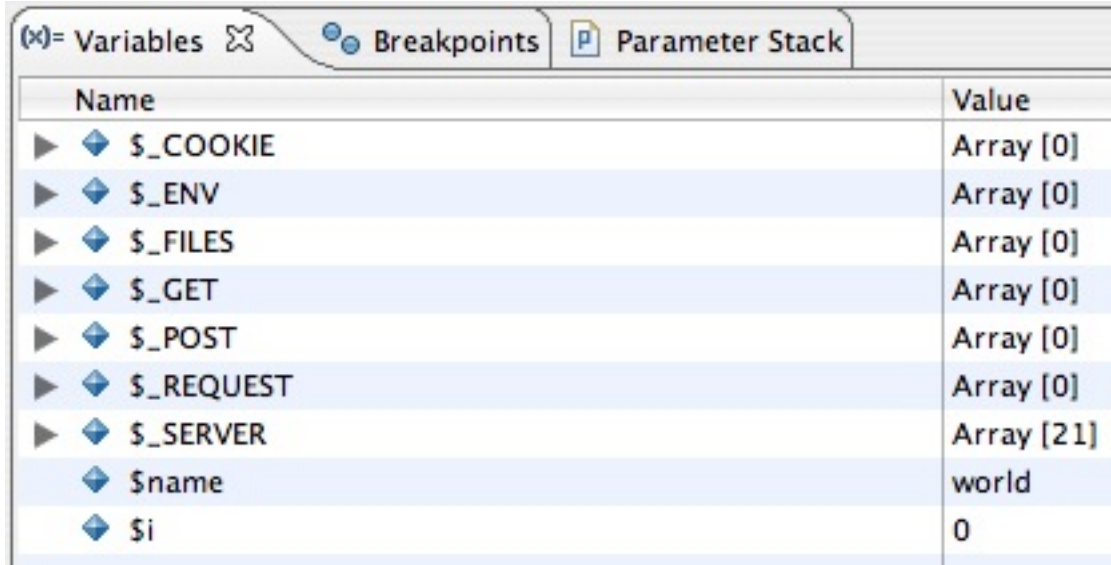
Figure 16. Suspended at the breakpoint (line 7)

While you're still paused at the breakpoint, look at the Variables view (see Figure

17). Both of the variables, $name and $i, are now in scope and you can see their values. The $name variable contains the string world, and the $i variable is set to 0.

Figure 17. Variables at line 7 of helloworld.php
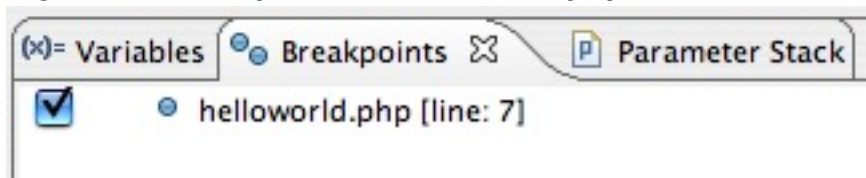
Because the breakpoint is inside a loop, executing to the next breakpoint goes to the next iteration in the loop. You can watch the value of $i increment in the Variables view.

To continue execution without stopping at every iteration, open the Breakpoints view, as show in Figure 18. Temporarily disable the breakpoint by clearing the checkbox beside helloworld.php (line: 7). Now when you execute again to the next breakpoint, the debugger will skip all the next iterations and run to the end.

Figure 18. Breakpoints in helloworld.php

You can add variables and modify the code to experiment and see how the debugger behaves.

Debug your Web application

Odds are you're using PHP to build a Web application and you want to debug your PHP scripts on the server. Debugging your PHP Web page on the server allows you to see how server variables such as $_GET and $_POST are set, so you know their value and can debug from page to page.

To try debugging a PHP application, use the webDebugExample project. This is a simple example that allows you to type a value in an input field, then submit the form for processing by the results.php script. In completing this exercise, you can follow how the `$_POST` variables are populated with the names and values of the variables from your form.

Select enterGreeting.php, then choose **Run > Debug As > PHP Web Page**. If this is your first time running the debugger, you will be prompted for the launch URL for the file. Here, you type the full URL to your file, remembering that it will be at the location in which your Web resources reside. For example, mine is at http://localhost/~nagood/webDebug/enterGreeting.php. Next, set a breakpoint at line 14 in the results.php file.

Listing 9. Breakpoint at line 14

```
print ("<b>Your greeting is:  <i>" . $gm->getFormalGreeting()
    . "</i></b>");
```

When you launch the PHP Web application, the debugger stops at the first line in the enterGreeting.php file. You can step through it to see the debugger in action or just execute to the end.

Type a value in the text input box in the form, then click **Submit**. Now the debugger stops at the first line of the results.php file. Execute to the breakpoint you set at line 14 of the script, and you will see that the `$gm` variable is populated with a value and that the `$_POST` array has the values from the previous page.

Figure 19. Variables in results.php

From line 14, you can execute to the end or you can click **Step Into** in the Debug

view to step into the `getFormalGreeting()` function on the `GreetMaster2000` class.

Section 7. Troubleshooting

This section provides processes and techniques for troubleshooting the debuggers and associated files.

Finding the correct php.ini file

When configuring PHP to use the debugger extensions — either XDebug or the Zend Debugger — it's important to make sure that you're editing the correct php.ini file and that you have the correct `zend_extension` variable for the debugger. To find the full path of the php.ini file your installation of PHP uses, use the `phpinfo()` function in a simple script placed in a Web directory.

Listing 10. Simple phpinfo.php script

```
<?php
    phpinfo();
?>
```

Alternatively, type the command `php -i | grep "Loaded Conf"` and you'll get a result like `Loaded Configuration File => /usr/local/php5/lib/php.ini`.

Use the correct zend_extension directive

Now that you're sure which php.ini file you need to edit, it's important to get the correct `zend_extension` directive to use for your installation. Use the `phpinfo()` or `php -i` method again to find two values: `Thread Safety` and `Debug Build`. Table 1 can help you determine which one to use. A common mistake is using `zend_extension` when you need to use `zend_exention_ts`, and it's a difficult mistake to catch (at first).

Table 1. Choosing the correct configuration key

Key name	Thread safety	Debug
zend_extension	Off	Off
zend_extension_ts	On	Off

zend_extension_debug	Off	On
zend_extension_debug_ts	On	On

The debugger doesn't stop

If the debugger doesn't stop at all, most likely one of three issues is occurring:

- The first is that you don't have the extension module set up correctly (see "Use the correct `zend_extension` directive" for the likely cause).

- If you've verified that the extension is set up correctly, you may not have PDT set up to use the debugger you have installed. If you've selected the Zend Debugger as the debugger for your PHP executable, but have XDebug set up in your php.ini file, you'll get some very unexpected behavior.

- Or it could be a firewall issue. Make sure you have the ports configured in your PHP Debugger preferences your firewall allows (if you have a firewall installed and running).

Determining your extension directory location

To install your XDebug or Zend Debugger extension binary in the same place as all your other extensions, use `phpinfo()` or `php -i` to find out where you directory is. See Listing 11 for an example from the `php -i` command.

Listing 11. Finding the extension directory

```
...
extension_dir => /usr/local/php5/lib/php/extensions/
...
```

Section 8. Summary

The PDT project supports two debuggers: XDebug and the Zend Debugger. Using a combination of PDT and one of the supported debuggers, you can graphically and easily debug PHP scripts and Web applications. Debugging PHP with an IDE is a huge time-saver, especially when compared to debugging scripts using older techniques or debugging with the command line.

Downloads

Description	Name	Size	Download method
Source code	os-php-eclipse-pdtdebug_source.zip	8KB	HTTP

Information about download methods

Resources

Learn

- "Squash bugs in PHP applications with Zend Debugger" tells you more about Zend Debugger and PHP.

- "Squash bugs in PHP applications with XDebug" explains XDebug and PHP.

- *XDebug Support in PDT 2.0* is an overview of the features and quirks of XDebug support in PDT.

- "Get started with Project Zero, WebSphere sMash, and PHP" provides a tour of Project Zero, an Eclipse-based project that also enables PHP development.

- PHP.net is the central resource for PHP developers.

- Check out the "Recommended PHP reading list."

- Browse all the PHP content on developerWorks.

- Expand your PHP skills by checking out IBM developerWorks' PHP project resources.

- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.

- Using a database with PHP? Check out the Zend Core for IBM, a seamless, out-of-the-box, easy-to-install PHP development and production environment that supports IBM DB2 V9.

- Stay current with developerWorks' Technical events and webcasts.

- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.

- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

- Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

- Check out the "Recommended Eclipse reading list."

- Browse all the Eclipse content on developerWorks.

- New to Eclipse? Read the developerWorks article "Get started with Eclipse Platform" to learn its origin and architecture, and how to extend Eclipse with plug-ins.

- Expand your Eclipse skills by checking out IBM developerWorks' Eclipse project

resources.

Get products and technologies

- Download the Eclipse IDE and install it from the official site.

- Download XDebug and learn more about it from the official site.

- Download the PDT bundled with the Zend Debugger from Zend's site.

- Innovate your next open source development project with IBM trial software, available for download or on DVD.

- Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

- Check out the latest Eclipse technology downloads at IBM alphaWorks.

- Download Eclipse Platform and other projects from the Eclipse Foundation.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.

- Participate in the developerWorks PHP Forum: Developing PHP applications with IBM Information Management products (DB2, IDS).

About the author

Nathan A. Good
Nathan Good lives in the Twin Cities area of Minnesota. When he isn't writing software, he enjoys building PCs and servers, reading about and working with new technologies, and trying to get all his friends to make the move to open source software. When he's not at a computer (which he admits isn't often), he spends time with his family, at his church, and at the movies. Visit his Web site.