php|architect's Guide to **PHP 5 Migration**

Practical porting techniques for
the professional developer

Stefan Priebsch

nb™ php|architect nanobooks

php|architect's
PHP 5 Migration Guide

by Stefan Priebsch

php|architect's PHP 5 Migration Guide

First Edition: June 2008
ISBN: **978-0-9738621-9-5**
Produced in Canada
Printed in the United States

Disclaimer

Although every effort has been made in the preparation of this book to ensure the accuracy of the information contained therein, this book is provided ”as-is” and the publisher, the author(s), their distributors and retailers, as well as all affiliated, related or subsidiary parties take no responsibility for any inaccuracy and any and all damages caused, either directly or indirectly, by the use of such information. We have endeavoured to properly provide trademark information on all companies and products mentioned in the book by the appropriate use of capitals. However, we cannot guarantee the accuracy of such information.

Written by	Stefan Priebsch
Published by	Marco Tabini & Associates, Inc. 28 Bombay Ave. Toronto, ON M3H 1B7 Canada (416) 630-6202 / (877) 630-6202 info@phparch.com / www.phparch.com
Publisher	Marco Tabini
Layout and Design	Arbi Arzoumani
Managing Editor	Elizabeth Naramore
Finance and Resource Management	Emanuela Corso

Contents

Chapter 1

Introduction

A long journey begins with the first step —*(Chinese proverb)*

A Short History of (Internet) Time

The roots of the Internet as we know it today, date back to 1960s. Back then, various universities and research facilities were linked together by the so-called ARPANET to simplify data exchange and make better use of the calculating capacities of the connected machines. In the following years, the programming language C and the operating system Unix were developed in separate projects. The TCP/IP protocol, that is still the prevalent protocol on the Internet today, was not introduced until 1983. In that year, ARPANET consisted of about 400 computers.

In the year 1987 the term *Internet* was coined. At that time, the network comprised about 27,000 computers. Some services that were offered were e-mail, FTP for file transfer, Usenet discussion forums or Telnet to access remote systems. While these services are still used today, other services like the WAIS (Wide Area Information Service) search or Gopher to navigate through a linked information network are no longer significant. Still, those forgotten services were the forerunners of today's World Wide Web.

The true triumphal course of the Internet began 1989 with an idea of CERN employee, Tim Berners-Lee. His idea was to introduce a *hypertext system.* Hypertext breaks up the linear structure of text documents. This allows for structuring the information in small blocks and linking them together. In contrast to reading printed text the reader dictates his own, personal navigation path. The envisioned hypertext system would allow navigating through a network of cross-linked documents using a graphical user interface. The motivation for creating this system was to enable a more efficient exchange of data, with a special emphasis on the fact that information gets quickly outdated in research and development.

While specifying the technology, Tim Berners-Lee called his idea "Mesh," but later changed the name to World Wide Web. The original proposal of the Mesh of 1989 can still be found on the Internet and is an interesting read[1]. Today we know that the introduction of the World Wide Web as a simple and intuitively usable user interface is one of the main reasons for the Internet's rapid growth and success. In acknowledgement for his invention of the WWW, Tim Berners-Lee was knighted by Queen Elizabeth II in 2004. Today he is working on the Semantic Web[2], to make the integration of different applications on the Internet easier.

In its early days the World Wide Web was primarily used to access static information, that was stored in pre-written HTML pages. Until about 1990, the commercial use of the Internet was even forbidden! Starting about 1993, the commercial use of the Internet started to take off. At that time, there were about 500 Web servers on the Internet.

In 1995 the HTML 2.0 standard was introduced, which contained the specification of HTML forms. This was the foundation for an interactive and dynamic World Wide Web. It was now possible to process data entered by the user and create personalized web pages. Processing input data, of course, requires program logic, so a clever way of coupling the form-processing programs with the Web server was needed because until then, the Web server's primary use was delivering static content.

To avoid re-compiling the Web server when the form processing logic changes, the form processing software should be loosely coupled to the Web server. This is achieved by making these programs external to the Web server. The Web server calls the program to process the form input, passing it the form input that has been transmitted to the server. The program creates an HTML page as the result, which

[1] Berners-Lee, Tim: The original proposal of the WWW, 1989, available at `http://www.w3.org/History/1989/proposal.html`

[2] World Wide Web Consortium: W3C Semantic Web Activity, 2001, available at `http://www.w3.org/2001/sw`

is passed back to the Web server, which in turn sends the page back to the client, usually a Web browser. Except maybe for the answering time, it does not make any difference to the browser whether a static or dynamic web page was requested.

This way of cooperation between a Web server and external programs was standardized as the Common Gateway Interface (CGI) in 1993[3]. Still today, the CGI principle is the technical foundation for almost any dynamic web page and web application.

The big advantage of CGI is that the programs can be written in virtually any programming language. In the early days, Perl was widely used. However, CGI programs written in C or another compiled language are usually faster than CGI programs in an interpreted language, since the source code need not be parsed at runtime. On the other hand, interpreted CGI programs are easier to modify, because they are available in source code on the target system. The time-consuming compile and link cycle is not required. Sadly, the fact that interpreted programs are so easy to change can lead to a somewhat undisciplined way of working.

A major disadvantage of CGI programs is the complex execution environment that is required to run them. To run and test a CGI program, you need a Web server and a browser, otherwise you are forced to test the resulting HTML code by analyzing it. (Today, it is not too unreasonable to test a CGI program in a simulated Web server environment. In the last century, however, this was quite a problem, since only few usable test tools were available.) Testing CGI programs independently from the Web server is tedious, because you have to supply simulated GET and POST parameters to the program.

The Birth of PHP

The first PHP version was created by Rasmus Lerdorf from Denmark, who had put his resume online in 1994 and wanted to know who was looking at it. He wrote a couple of Perl scripts to log access to his resume. A little later he began to rewrite these scripts as CGI programs in C. He called the package Personal Home Page Tools (PHP) and published it under GNU Public License (GPL) as open source software.

[3] University of Illinois at Urbana-Champaign: Common Gateway Interface, 2008, available at `http://hoohoo.ncsa.uiuc.edu/cgi`

The official announcement of PHP from 8th of June, 1995 can still be found in the Google Group Usenet archive[4]. Of course, according to today's expectations, the scope of services PHP offered is not particularly spectacular. One has to remember, though, that HTML forms had just been standardized in the year 1995.

In 1997, PHP version 2.0 was published as PHP/FI, with FI meaning Forms Interpreter. PHP 2.0 was inspired by Perl and was probably used on some 50,000 domains on the Internet.

In the same year two students form Israel, Zeev Suraski and Andi Gutmans, became aware of PHP/FI. They had the idea to rewrite the PHP parser for a university project and in due course created a full-blown programming language that became the official successor of PHP/FI under the name PHP 3. The original meaning of the acronym PHP was changed to PHP: Hypertext Preprocessor.

PHP 3 was targeted at powering small to medium web sites. The PHP scripts were interpreted line by line, which made PHP very slow for complex programs, because every line of the source code had to be interpreted again on every program run. Opcode caching as we know it today in PHP was not possible by design. Also, PHP 3 had no built-in session support and was thus missing important features to create complex web applications.

Zeev and Andi rewrote the PHP core for PHP 4.0 and named borrowed the "ze" and "nd" letters from their first names to form the name "Zend Engine". At the same time, they founded the company Zend that offered commercial add-on products to the open sourced PHP. Today, Zend is based in the Silicon Valley and has an annual revenue of over twenty million dollars.

PHP 4 was published in May 2000. The dotcom boom was at its peak, which helped PHP to spread quickly. At the turn of the millennium, it was en vogue to have a (dynamic) Web site, as companies began to use the Internet as a sales and communication channel.

PHP is easy to learn and heavily oriented towards Web programming. Compared to using a complex Java-based application server with hefty license fees or the clumsy traditional CGI programming, PHP programming is very rapid, and quickly shows results. With PHP, anybody can create a dynamic web site based on a shared hosting server at virtually no initial cost. The large number of available documenta-

[4] Lerdorf, Rasmus: Announcing the Personal Home Page Tools version 1.0, 1995, available at `http://groups.google.de/group/comp.infosystems.www.authoring.cgi/msg/cc7d43454d64d133`

tion and code samples on the Internet lower the bar for less experienced programmers even further.

PHP 4 has a modular architecture, which allows for extending its functionality by extensions written in C. Today, a large number of open source extensions for PHP are available for virtually any intended use, for example creating PDF documents, or dynamically creating Flash animations. Through extensions, PHP supports most Internet protocols like FTP or SSH.

Thanks to the Internet boom, many open source applications like Content Management Systems, user interfaces for servers and services, Wikis, or weblogs were created that are still widely used today. Today, the large number of publicly available applications is a good reason to use PHP, especially because you can extend and customize those applications.

Originally, PHP was a procedural language. In version 4, rudimentary support for object-oriented programming (OOP) was added. The syntax was only intended to simplify the access of complex data structures. Though OOP support was so limited in PHP 4, many developers started using OOP with PHP. The success of PHP 4 had accounted for the fact that increasingly larger applications were created with PHP, which made the developers look for ways to make their code more modular and maintainable.

The biggest shortcoming of OOP with PHP 4 was the fact that PHP treated objects by copying them, instead of working with references, as one would expect when familiar with classic OOP languages like Java. In PHP, you had to make liberal use of the & operator to force the use of references instead of creating copies. Forgetting one single & sign often resulted in long and challenging debugging sessions.

An important precursor of object-oriented PHP programming is Stig Bakken from Norway, who started the PEAR project back in 1999, even before PHP 4 was officially released. PEAR (PHP Extension and Application Repository) is a collection of open source software components for PHP, inspired by CPAN (Comprehensive Perl Archive Network). PEAR contains only object-oriented code.

Even in its early days, PEAR made use of advanced OOP concepts like destructors and error objects as a replacement for exceptions, which were not available in PHP back then. The PEAR project as precursor for object-oriented programming with PHP was probably a main reason for the inclusion of proper OOP support in PHP 5.

PHP 5 and the Big Migration

July 13, 2004 was the day PHP 5.0 was released. PHP 5.0 was the first PHP version with full-blown support for object-oriented programming. PHP 5.0 has exceptions, interfaces, abstract classes, access protection for methods and members, a reflection API and by default handles objects by reference, instead of creating copies.

In light of the new possibilities, it seemed only a matter of time until PHP 5 had completely replaced PHP 4 in the field. But the migration did not really happen, and PHP 5 spread only very slowly. By the end of the year 2005, more than a year after PHP 5 had been released, about 90% of all PHP installations on the Internet still ran on PHP 4[5].

One reason is the fact that a migration never goes by without problems, which means work. Even if you run your program in an updated environment and get no error message immediately, this does not mean that your program behaves the same as on the original system in every situation. You will have to thoroughly test your application in the new environment.

Most of the time, an application that was written in a certain PHP version does not work on a new PHP version without modifications to the code. A major version bump almost guarantees you compatibility problems, but even a bump of the minor version number can lead to problems (There is a problem with keeping backwards compatibility. When a new software version is created, a decision has to be made between removing legacies and introducing incompatibilities. You hardly ever find a compromise that satisfies every user. Regardless how you decide, there are always users that would have decided differently.) Even if only minimal adjustments to the code are required in the end, you still have to thoroughly analyze the code.

Hosting companies were in an especially difficult situation. Had they just switched existing PHP 4 installations to PHP 5, a lot of customers would have experienced trouble from one day to the next. So PHP 5 was offered as an alternative to PHP 4, requiring every customer to choose the ideal time for migration. Unfortunately, most customers just never migrated.

Not all PHP users are shared hosting users, but the shared hosts make up for the major part of PHP installations on the net. All users administering their own PHP

[5] Seguy, Damien: Statistiques de déploiement de PHP en décembre 2005, 2006, available at http://www.nexen.net/chiffres_cles/phpversion/14847-statistiques_de_deploiement_de_php_en_decembre_2005.php#versions

server can also choose the time for migration by themselves, but have even more work, because they need to upgrade the installed PHP version first.

Since PHP 4 was so wide-spread in the last years, many developers were afraid of making PHP 5 the minimum requirement for their applications. They kept their software PHP 4-compatible, even when they were already developing under PHP 5. This hinders making use of all advanced OOP concepts in PHP, of course.

In a way, this was a comfortable situation for everybody involved. Developers were not forced to introduce OOP to their existing procedural code. Users were not forced to use PHP 5, because most applications they used would also run on PHP 4. So why should users bother with migrating to PHP 5 at all?

The situation, comfortable as it may have been for PHP developers and PHP users, was not at all comfortable for the PHP core developers. For years, the were forced to maintain PHP 4 while PHP 5 was already released, and PHP 6 was already in development. It is a waste of sparse resources to force the PHP core developers to work on three major versions of PHP in parallel.

Exactly three years after releasing PHP 5, on July 13th 2007, the PHP core developers announced that support of PHP 4 would be dropped by then end of the year 2007. Critical security fixes would be supplied until 8th of August 2008, the beginning of the Summer Olympics in Beijing. After this day, PHP 4 users are left on their own, if the have not migrated to PHP 5 yet.

Also in the early summer of 2007, the gophp5 initiative came to life. Under the URL `http://www.gophp5.org` open source projects are encouraged to make PHP 5.2 the minimum requirement for their software by February 5th, 2008. Hosting providers are asked to offer PHP 5.2-based hosting by default from the same date on. gophp5 tries to break the vicious circle of users and developers forever sticking to PHP 4.

A Look Ahead: PHP 6

In fall 2005, a number of PHP core developers met in Paris to discuss the features planned for PHP 6[6]. A first preview version of PHP 6 was originally announced for the end of the year 2006, and then for the end of 2007. Unfortunately, the road to

[6] Rethans, Derick: Minutes from PHP Developers Meeting, 2005, available at `http://www.php.net/˜derick/meeting-notes.html`

PHP 6 was long and rocky. The main new feature, namely full Unicode support, was more difficult to implement than it was originally thought.

Migrating existing PHP applications to Unicode is not an easy task, because the assumption that one character matches one byte is not valid for Unicode any more. The built-in functions in PHP 6 take this into account, so that functions like `strlen()` or `substr()` still work flawlessly. In most cases, though, existing code will have to be modified to work on a Unicode-based PHP.

Some PHP core developers think that Unicode support in PHP is not required, and thus do not support implementing Unicode features wholeheartedly. In late 2007, it was decided to delay PHP 6 and backport all other planned features except Unicode support to PHP 5.3. Aside from the missing Unicode support, PHP 5.3 equals PHP 6 as is was originally planned. Some of these features like the integration of the XML-Writer and XMLReader extension were already incorporated into PHP 5.1 or PHP 5.2. In the near future new features will probably appear in PHP 5 and PHP 6 in parallel, as is was the case with new features in PHP 4 and PHP 5 for quite a while.

From today's viewpoint, PHP 6 could share the same fate as PHP 5: after its release, the new version spreads slowly, and another vicious circle arises: since PHP 6 is only rarely used, no applications will be written for PHP 6. And since most existing applications work under PHP 5, there is no reason to migrate to PHP 6. Thus PHP 6 will spread as slow as PHP 5 did.

Since any migration from one major version to the next requires careful planning and unfortunately, as was said before, quite often even the migration between minor versions can be problematic (well, otherwise you would not read this book), it is quite understandable that PHP users are rather skeptical against new PHP versions.

Still, there is a fundamental difference between the migration from PHP 4 to PHP 5 and the migration from PHP 5 to PHP 6. The main new feature in PHP 5 was improved OOP support, thus making software easier to extend and maintain. While PHP application developers profit from OOP, the end users do not get usable features out of it short-term.

The most important new features in PHP 6 is the Unicode support, which immediately provides increased value for the end user, because the application will better support different character sets and encodings. So there is still hope that history of slow PHP 5-adoption will not repeat itself for PHP 6.

Chapter 2

Strategies

Whether you walk or run, the distance will always be the same. —*(Chinese proverb)*

Migration Strategies

There are no definitive solutions in the IT world. Like the real world, the IT world changes constantly and rapidly. Software is extended, because new features are required. Software is optimized, because results are needed more quickly or more concurrent users must be served. Bugs or security holes are found, so the software must be modified.

Hardware changes constantly as well. According to Moore's Law, the number of transistors in a microchip doubles roughly every 18 months. Every few years the structural shape of mainboards, memory, and hard disks changes. Users expect software to run not only on the most current hardware, but also on older systems.

It is not always easy for application developers to keep pace with this development. Thanks to the work of operating system and driver developers, most of the hardware changes do not directly affect application developers, but they have to adapt to a constantly changing software environment. The more tightly an application is integrated with the operating system, the more difficult it is to keep backwards compatibility while supporting the latest operating system version.

System administrators have the toughest job of all. They usually are on a small budget, but have to adapt the systems to rapidly changing business requirements.

Since users do not like to change their habits, system administrators must take great care to keep continuity.

The migration of software and computer systems happens at the cut surface between developers and administrators. To be successful, it is important to gather a holistic view of the system to migrate and its environment. Also, the users should not be left out of the equation, since they ultimately decide wether a system is going to be used or not.

In this chapter we will deal with basic migration strategies. For a migration, the system administrator's concerns and the developer's concerns are equally important. Also, the users and management need to be aware of the fact that no solution in information technology is final. Migration is inherently a part of IT life. The question is not whether another migration will ever be required, but when the next migration will be required.

Never Touch a Running System

Anyone who was ever responsible for system administration, even if it was only their own computer, knows the problem: right before you quit work, start the weekend or even vacation, a small software update is required, the system configuration must be changed, or a new application must be installed.

Even when things seem to be in order at first glance, you never know whether your system is still going to work after the first reboot. Sometimes changing a small and seemingly innocent setting is enough to give you a few hours of troubleshooting after the first reboot.

Of course it is purely superstitious, but sometimes you could start believing that a computer actually realizes that you are short on time or an important deadline is approaching, only to cause trouble. An often-quoted golden rule (if not the one golden rule) of information technology therefore is: never touch a running system.

System Environment

The term *running system* primarily refers to production systems. If a system needs only be available during working hours, maintenance work can be done in the evening or at the weekend. Before starting, you should plan for enough spare time to get the system up and working again before it is required in production again, just in case.

Today, many systems must be available 24x7. For some businesses, downtime must be announced weeks in advance, requiring careful planning of system maintenance work. To keep downtime short, all changes to the production system should be tested on an identical test system, before you apply any changes to the production system.

Unfortunately, in reality, identical test systems are not always available because they can be quite expensive to keep if a high-capacity production system is used. Virtualizing systems can cut the costs, but the bigger the differences between the test system and the production system are, the more sources of trouble there are. Maybe this is one of the reasons for the recent trend to also virtualize production systems.

The less documentation there is available for a system, the better off you are not to make any unnecessary, seemingly harmless changes. If, after such a change, the system does not work as expected anymore, and you fail to put the system back into a working state, the question is how a replacement system needs to be installed and configured.

If the original system can still be used as a guideline to install and configure the replacement system, it is rather easy to set up an identical or at least similar system. If no replacement hardware is available and you have to reinstall the old system, you never know the outcome, since you deprive yourself of any fallback position when you erase the old system.

In any case, you need a backup of all relevant data and settings. Looking at today's amount of data that is being pushed around, it can be a couple of hours until the data has been transferred to another system. The best solution is to keep the original hard disk and reinstall the system on a new hard disk. Of course, this only works when you have direct access to the server. It is not really an option if you use a rented server in a remote data center.

"Never touch a running system" is often used as an excuse not to install required updates and patches. It is always a good idea to only make changes to the system that have a clear benefit. Before installing anything, test on another system and always keep at least one fallback position that allows you to roll back to a working system, in case things go really wrong.

Because a system is not clearly documented, and you have no idea what is going to happen when you make changes to it, as soon as “never touch a running system” becomes a necessity, it is very dangerous to keep waiting. The next serious problem can cause a chain reaction, as the following example shows.

One day you decide to clean up behind your server so you figure that it might be a clever idea to unscramble all the cords. You shut down the server and clean up. Once you are done, you try to reboot the server, but realize that it does not work any more, because the hard disk has died. (As a matter of fact, hard disks rarely fail while running, but mostly when rebooting the server. This is one of the reasons why servers are seldom rebooted and the hard disk’s power saving features are turned off.)

You need to get a new hard disk. Unfortunately, no a hard disk with an adequate interface is in stock. The hard disks you and your computer dealer have in stock are not supported by your server. Since you cannot wait for a compatible hard disk to be delivered, you have to switch to a different server hardware.

Unfortunately, the full backup of the old server cannot be installed to the replacement system, because the hardware required different drivers. Since you do not want to risk an unstable system, you decide not to install the existing backup, but to install the new server from scratch.

After two or three failed installations, you realize that the operating system version you were using on the old hardware does not install on the new hardware. Even if you could complete the setup, you would never know wether the system would actually be stable.

So, you are forced to install a more current operating system version. Installation on the new hardware works seamlessly, so your hopes are raised, at least until you realize that not all software components required by your application work on the new operating system.

Since you cannot combine new versions of certain software components with older versions of other components, because the API has changed between versions. You need to install a current version of all software components to make them work together.

Of course, it is bound to happen that your application does not work in the new environment, at least not flawlessly. You have already spent hours with installing a stable system and now are forced to get your application to work in the new environment. Things get even worse when the errors that the new environment induces are not immediately visible. In that case, you are likely to face a time of trouble.

You would probably want to avoid a situation like this, because you will be forced to catch up with all shortfalls now. This is the Sword of Damocles you live under when you've been living to the "never touch a running system" philosophy for too long.

Program Code

Sometimes, the "never touch a running system" rule is also used as justification not to change existing program code. At first glance, this philosophy is warranted, because in the first place, any modification to source code can lead to a program that does not work any more, or does not work as expected.

Unfortunately, it is way too easy to introduce a syntax error when making even a small modification to the source code. In a compiled language the compiler would discover syntax errors quickly, since the code would not compile any more. PHP has no compiler, though, so even simple syntax errors can go unnoticed until the program is actually loaded and executed.

A good IDE will show you syntax errors as you edit the code. Even without this feature, you should try to discover these obvious errors as early as possible. Thus, after modifying the source code, make sure to run a syntax check. In Chapter 7 we will learn how to run a syntax check on a PHP file at the command line.

The fact that source code has no syntax errors does not mean that the program will work flawlessly. A PHP file that is syntactically correct can still contain errors that will show up at runtime, for example when you try to call a non-existing method of an object. Since the object is created at runtime, PHP cannot know at compile time wether the called method actually exists.

Runtime errors that cause a program to stop may be annoying at first glance, but in fact help you find an error rather quickly. Warnings and notices can also help you discover potential errors that might show up in a completely different spot. A notice about using an uninitialized variable can indicate a security problem in certain PHP configurations. Ideally, PHP code should be written so that no warnings and notices are output, at least not in the default case. In some situations, this means unnecessary effort to prevent PHP warnings, for example when connecting to a database fails.

The biggest problem are errors in the program logic. If, for example, you subtract the items of an invoice instead of adding them up, this is an error you can only find by testing. PHP will not output a warning or error message, except when you have built a plausibility check into your software that will warn you when the grand total becomes negative.

The more complex your application becomes, the more tests you need to run to make sure that your application does what you expect it to do. In theory, this sounds very simple: after every change to the code, just fully test the application, and if no test fails, you know that you have not introduced a bug or changed the application's behavior.

In practice, this is virtually impossible, because program code can consist of an arbitrary level of nested statements. Full test coverage would require testing every combination of execution branches. A program with 10 `if` statements executed one after each other has not only 20 possible execution paths, but 2^10=1024 theoretically possible execution paths.

A large part of these execution paths may only be theoretically possible, and it does not make much sense to test execution paths containing code to handle rare error conditions like a failing database or a full hard disk. The effort to create tests for these situations is usually not worth the benefit. But even if only 10% of all theoretically possible execution paths were to be tested, we would have to create about 100 tests for a small application.

Due to the large number of tests that are necessary even for a small application, tests should be automated. An automated test compares a computational result to a pre-calculated known good value. Since human testers can easily overlook errors in a program's output, an automated test is not only more reliable, but cheaper, since it can be repeated at any time, at no additional cost. Automated tests are an important precondition for changing and extending existing program code. After the modification, rerun the test, and you will know whether your program's behavior has changed.

In practice, automated tests are not as widely used as they should be. Especially for older procedural PHP code, automated tests rarely exist. Since procedural code is less modularized than object-oriented code and thus more difficult to test. In older applications that have grown over time, different concerns like presentation, program logic and data access are not clearly separated, making unit testing (testing of individual classes or parts of the code) rather difficult.

Instead of using unit tests, one could consider testing the application as a whole. Of course, the environment required to test a whole application is way more complex than the environment of a unit test. In addition to the database that must be pre-populated with sensible default values, a browser is required to execute the Javascript code if you do not want to rely on parsing the generated HTML code.

Application tests are coupled to the user interface rather tightly. Sometimes, even cosmetic changes to the user interface make is necessary to adjust a test. Thus testing an application as a whole is no replacement for unit tests, because creating and maintaining them is just too much effort. In addition, it is difficult to test certain edge cases.

There are different doctrines about when and how program code should be changed. In classical software development, program code is owned by the developer who wrote it, and nobody except the owner is allowed to change the code. This restrictive approach is probably also due to the fact that not enough automated tests are present to assure the software quality. This leads to a “never touch a running system” attitude towards source code. Since nobody ever changes the code, after some time no developer is familiar with the code anymore.

The agile programming methods propose collective code ownership. Program code is not owned by one single developer, but by the whole team of developers. This means that any team member can change program code. That way, programmers do not only deal with the code they have written by themselves, but also with other people’s code, which helps them better understand the big picture of the application.

This agile approach only works well when automated tests are in place. The advantage of constantly changing and improving existing code is that unreadable code and code that does not adhere to the coding guidelines will sooner or later be improved. The result is an overall better code quality.

Nobody should make modifications to existing code without a good reason. However, as soon as you refrain from making changes to the code just because there are not proper quality assurance measures in place so you cannot test the application or parts of it, program code will start to age.

Program code ages, because the technology evolves. New software versions offer new possibilities, programmers gain more experience and find better solutions than in the past. When developers look at code they have written a couple of years ago, they usually ask themselves whether or not they have actually written that piece of code.

Like in system administration, you can get away with the "never touch a running system" philosophy for a while and keep existing program code as it is. Sooner or later, however, you will end up rewriting your application from scratch, not only because no developer knows and understands the code, but also because the code has grown too old.

Always Use the Latest Version

No software is free of bugs, be it an operating system, an application, or embedded software. Sooner or later, an error will occur. Even when a program has been thoroughly tested, bugs may show up under certain conditions after months or even years of running in production.

Security holes in the software are particularly annoying, because they might allow attackers to break into a system, and steal or modify data. Security is a big issue, especially in the web environment, because every system on the Internet is subject to potential attacks every day and around the clock. Compared to this, computers that are not accessible from the Internet live in relative peace.

The biggest problem for computers on the Internet are not even targeted attacks, but the white noise of automated attacks that is omnipresent on the Internet. Arbitrary systems are automatically scanned for known vulnerabilities. Sometimes, to prepare an attack, it is even sufficient to know which software a server runs.

Another golden rule of information technology is to always use the latest version. By using the latest version you can make sure that your system has no known vulnerabilities. Administrators are often blamed for not using the latest software versions. Unfortunately, things are not easy as they may seem.

We know that it's not possible to prove that a program is correct by testing. Tests can only prove the presence of bugs, not their absence. We have already learned that even small programs can have a large number of execution paths, so that full test coverage of a real application is hardly possible.

Even when an application in isolation works flawlessly, problems can arise when taking the execution environment into account. To guarantee that an application always works correctly, the operating system, all installed drivers, and all shared libraries would have to be error-free. This is just not realistic. Typically, you ensure by integration tests that a production system works correctly, at least on normal use of the application.

If you install a system from scratch, you should use the latest available version of all software components. Sometimes, however, the latest versions of different software components do not work together. Before Apache version 2.0.40, PHP would just not work together with Apache2, for example.

When developing software, you often have to take the risk of working with software that has not been officially released. This allows you to release your application as soon as the software it depends on has been officially released. Most of the time, technical aspects like new features that you require turn the balance for using a development version.

Using an older version of a software product also has its advantages. Older versions are better tested, and even if they have known problems, chances are good that there is also a known workaround for them. It is usually easier to circumvent a bug than to first prove that a certain behavior is a bug, and then find a way of circumventing it.

When a system has been deployed, time does not stand still. Sooner or later you will face the problem of having to update a certain software component, for example because a security issue has been found. In theory, you would have to repeat the full integration test to make sure that your system still works as expected with the updated software component. Quite often, though, software components are updated without any proper quality assurance.

Luckily, many bug fixes are released as patches today. This shortens the period of time until a bug fix is available, because not a full software release must be created. Still, many bugfix patches are not tested as extensively as a full software release. Quite often, patches require another patch shortly after they have been released. Either the original problem was not properly solved, or unwanted side effects occurred. It may therefore be a good idea to wait for some time before installing patches. You may be able to learn from other people's experience and even skip some patches that turn out to be unnecessary.

Any responsible administrator will never just blindly install a patch or an update, but make an educated decision whether there is a real benefit that outweighs the risks. If no critical vulnerability is present in your system, if the users' daily work is not affected, and if data integrity is ensured, there is nothing wrong with not installing a patch.

Many of the vulnerabilities found in PHP over time can only be exploited from the local system. If the attacker has no shell access and cannot execute arbitrary PHP programs, there is no real risk for the system. In a shared hosting environment, however, such a vulnerability can be absolutely critical.

In general, any modification to a production system bears the risk of damaging the system integrity and putting the system into an unusable state. Many administrators have had bad experiences with "always use the latest version" and have in turn started to live the "never touch a running system" philosophy. Of course, it is very wrong to overreact like this.

Starting from Scratch

If you have been living "never touch a running system" for too long, you will usually pay by having to setup the whole system environment from scratch, or rewriting the whole application from scratch.

Having to reinstall a system from scratch may not be fun, especially when you are under time pressure because every hour of downtime causes massive costs. But even then, after a few days of tweaking and sweating, even stubborn systems should be back online and relatively stable.

Rewriting an application from scratch, in contrast, is not a matter of days, but requires months or even years. In addition, it is obvious that nobody can come to a decision about rewriting an application within a couple of hours. A decision of such importance requires careful planning and analysis.

Rewriting Program Code from Scratch

We already know that program code ages over time. When code is being changed and extended, the original structure and architecture slowly blurs. Code must be regularly cleaned up by refactoring to counterbalance the aging effects.

Never changing existing code will in most cases ultimately lead to the need to rewrite the application from scratch because no developer knows the code any more. If you have no good source code and development documentation, you are in trouble. In that situation, the question is whether somebody should spend time on familiarizing themselves with the code and to whip it into shape, or if it is better to rewrite from scratch. Writing new software is always risky, as many software projects fail.

There are different approaches to rewriting a system. Besides having the option of using new tools and methods, developers will try not to repeat the same mistakes they made in the first place.

If you rewrite software, you can use the existing system as specification, so there is a clear vision of how the software should look like. The danger is that developers focus too much on making everything better. Frederick Brooks wrote about this phenomenon in his legendary book *The Mythical Man Month* and called it the second system effect[1]. Brooks realized that many teams that had successfully completed a software project together were bound to fail in the follow-up project, because they tried to build too much functionality into the software to make up for everything they felt they had forgotten in the first project.

When working on existing software, you can be sure that certain functionality is present, unless you break it while modifying the software. When starting from scratch, there is no guarantee that the result is a working piece of software. Many software projects, especially big ones, go through great troubles until the result is satisfactory.

In theory, it is just a matter of calculating the cost for a new development and compare these with the estimated costs for maintenance and extension of existing code. In reality, this calculation it is virtually impossible. Generally, at short term maintenance of existing software is cheaper, while in the long run a newly developed software will probably have lower maintenance costs. In any case, a good code basis makes maintaining and extending software easier and thus more cost-effective.

Rewriting software means new possibilities. You can get rid of old code that is hard to maintain, and can start using existing third-party components and classes. In addition, you can use new PHP features. Before starting to rewrite, though, you

[1] Brooks, Frederick: The Mythical Man Month, 1995, originally published by Addison-Wesley, Amsterdam, 1975

try to find out whether there is an existing software that you could use instead of writing everything from scratch. If there is no software that completely fulfills all your needs, you can still possibly use an existing product as the starting point for your own development. It is one of the great advantages of open source software that you can reuse existing products. Even if you do not reuse actual code, you can still learn from others people's work.

To reuse existing components, you do not have rewrite your software, though. When migrating the software or by code refactoring you can replace existing code by another implementation, thus introducing third-party code into your application.

Rebuilding the System Environment from Scratch

Compared to the extremely sumptuous task of rewriting software, rebuilding a system seems like a walk in the park. But since most production systems are used to earn money, directly or indirectly, downtime is expensive.

While the decision to rewrite software usually is reached because existing software is too tedious or expensive to maintain, it is very helpful to rebuild a system from scratch instead of making modifications to an existing system. While you are installing, the production system is still running, so there is no downtime. Once you are done with setting up, configuring, and testing the new system, you can switch the two systems, making the newly installed system the production system. If anything goes wrong, you can switch back to the original system. By switching between the two systems, there is virtually no downtime.

Of course, it does not always make sense to perform a full installation of a second system, just to install one patch. For major updates when migrating a system, the effort is definitely worthwhile. To make this approach of installing a new system actually work, you need a good and up-to-date documentation about how to set up the system, which software to install, and how to configure it.

Striking a Balance

In the previous sections, we have learned about basic strategies and procedures for a migration. Like so often, in information technology it is not a good idea to fall from one extreme into the other. You should try and strike a good balance.

Avoid the ball to be placed in your court. It is much better to be anticipatory and proactive. This helps you plan ahead and lets you avoid panic reactions when things go really wrong at some point. Knowing the market trends will help you plan better and recommend a sensible course of action.

In real IT environments, there are a lot of influencing factors, and not all of them are technical. Besides the business environment, IT decisions are often influenced by corporate politics.

A fixed budget is an important precondition for a successful ongoing development of an IT environment. You would probably not want to enter a plane that has not been serviced for months, just to cut the costs. Similarly, nobody should entrust their data and business processes to a badly-maintained IT system.

Information technology means accepting changes as an integral part of daily life. Not only the technology changes, but also the business requirements and, last but not least, the persons involved.

Only if you have a solid understanding of the IT systems involved, you can accept permanent changes and adjust to them. You need to know which software runs on the system, how it is configured, and which dependencies there are between the various system modules. The better the documentation, the easier it becomes to make changes.

You can learn an important lesson for system administration - and therefore the migration of a system - from the agile development methods: work in little iterations (or steps). This simplifies management and is more productive. Multiple small steps will take you as far as one big step, but should problems turn up, it is easier to isolate them. Should you really screw something up on the way, it is easier to roll back one little step than to spend hours searching for the mistake.

A good balance between “never touch a running system” and “always use the latest version” could be to update the production system every quarter of the year. This gives you enough time for quality assurance and lets you make the necessary changes to your software, if need be. You do not always run the latest software in production, but can refine your testing and deployment process over time.

Should a critical update become necessary, you have a proven process that you can rely on. Instead of pushing the necessary migration of your environment farther and farther ahead, until you finally are forced to act, you constantly proceed in small and manageable steps. This usually gives you enough time to sort out any problems without putting the production system into danger.

Regardless of when you migrate, you will run your application in a new environment. This usually requires some modifications of the existing PHP code, to adapt it to the new environment. In the following chapters we will explain the course of action. Chapter 7 introduces you to tools that will help you migrate your code, while Chapter 8 describes in detail the potential problems that can arise when migrating PHP code.

This book only deals with forward migration, which means adapting an existing PHP application to work in a software environment of newer versions. In a backwards migration, the PHP application is adapted to working with environment of older software versions. Though the basic procedure in both cases is similar, we will not deal with backwards migration. It does not usually make much sense to adapt code that was written for a recent PHP version to work under an older PHP version.

Chapter 3

Migration Aspects

“Discover a well before you are thirsty.” —*(Chinese Proverb)*

Important Aspects of Migration

A migration does not only affect the program code itself, but also the whole system environment. There are a lot of aspects to keep in mind when planning and acting out a migration. This chapter deals with the most important aspects that you have to keep in mind when migrating a PHP environment.

These aspects are the platform, operating system, web server, database, PHP itself, the PHP source code and some other aspects that might seem unimportant at first glance, like system configuration, external programs, or interfaces to third-party systems. Figure 3.1 shows an overview of these components and how they interact. The dependencies are shown by arrows.

The starting point of our consideration is the platform, meaning the computer architecture of the application we are planning to migrate runs on. The platform-specific operating system hides the platform details from the applications running on it. There are some important differences between the main operating system families. When migrating, you should know them, even if you are not planning to switch the operating system family while migrating.

The operating system executes the various applications. For us, the two most important applications are the database and the web server. The system programs are

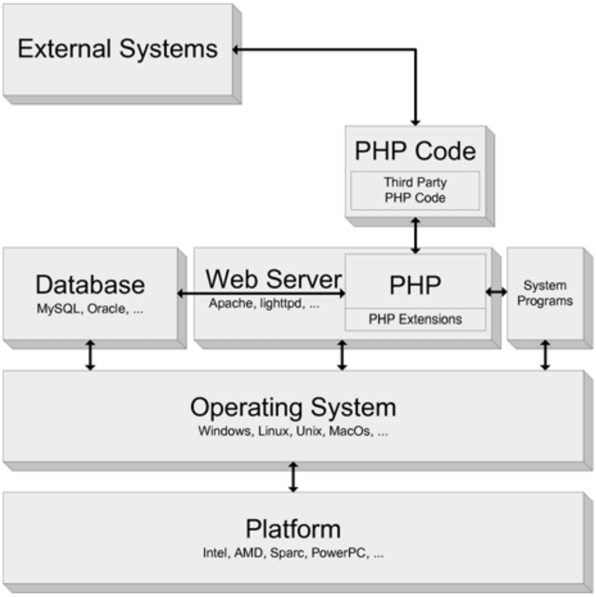


Figure 3.1

usually not overly relevant in a PHP migration, but must still be kept in mind. Luckily, Apache and MySQL, the most widely used web server and database on PHP servers, are available on all common operating systems. Still, there are some differences in behavior between the platforms with these applications, especially the web server.

The web server runs PHP, which can access system programs and the database. (Strictly speaking, this is not always correct, but depends on how PHP is integrated with the web server.) An important part of PHP is the collection of extensions that provide additional functionality to the PHP core. In fact, the most features of the language are provided by extensions.

For example, the various database interfaces, regular expressions or XML functionalities are PHP extensions that are bundled with PHP by default. (“Bundled by default” does not mean these extensions are always available, since they could have been disabled when configuring the source code). The PHP core itself offers a rather small set of user visible features.

PHP, obviously, executes PHP code. In addition to your own PHP code, you might also use third-party PHP code in the form of libraries, components or frameworks. Usually, you will not have to migrate third-party code, since the manufacturer should have taken care of making their code work under recent PHP versions.

Interfaces to external systems are especially important if your application is service-oriented, because every service client is an interface to an external system. The advantage of service-oriented architectures is that the various services or modules of the application are decoupled, which makes maintenance much easier.

It is also easier to balance the load by distributing the services to different servers. Compared to one monolithic system, a service-oriented architecture is much more flexible and scalable.

In this chapter, we will take a closer look at the various aspects of a migration and the system components involved. You will be provided with the necessary background knowledge to get a feeling for which problems you might face during migration.

Platform

Today, computers are ubiquitous. They exist not only in the form of PCs or game consoles, but also as chip cards, super computers and most electronic devices like MP3 players, DVD players, and even in everyday objects like elevators, exercise machines, cars, or digital watches.

All of these computers are unlike each other, but adapted to special use cases. In principle, though, all modern computers are based on a concept for a universal programmable calculator by John von Neumann[1] that dates back to 1945. The then revolutionary idea was to store program and data in the same memory. This was the cornerstone for the transition from hard-wired calculating machines to programmable computers.

[1] von Neumann, John: First Draft of a Report on the EDVAC, 1945, `http://wps.com/projects/EDVAC/index.html`

Architecture

The origin of today's prevalent computer architecture is the so-called IBM compatible PC, that is based on IBM model 5150. Model 5150 was IBM's attempt to gain a greater market share than companies like Commodore, Atari, Apple and Tandy, which were very successful in the home computer market. The IBM 5150 was based on Intel's 8088 processor, a low-cost version of the 16 bit processor 8086, that Intel had released in 1979.

The especially noteworthy fact is that IBM made the specification public, allowing third-party vendors to build compatible computers and add-on parts. In the following years a big market for computers and peripheral devices emerged, which of course was extremely beneficial for spreading the architecture.

The term IBM compatible was a main sales argument for a long time. Today, this architecture is also called the x86 architecture, since the model numbers of Intel processors used to end in 86 (80286, 80386, 80486). Since the 32 bit processor 80386 was introduced in 1985, the extended 32 bit architecture is also referred to as IA32 or i386 architecture.

There are also alternative computer architectures in the PC world, but all of them are less common than i386. An example is the Sun SPARC architecture, which is open source today, and is mostly used on high-end workstations or servers running a Solaris operating system.

Another example is the 68000 processor architecture. Some once famous computers like Commodore Amiga, Atari ST, but also the first Apple Macintosh models were based on this architecture. Later, IBM, Apple and Motorola jointly developed the PowerPC architecture. Starting in the mid-nineties, all of Apple's PowerPC's were based on this architecture.

Yet in 2005, Apple started using Intel processors for their Mac computers, because Motorola had withdrawn from the PC processor market and IBM focused the further processor development on game consoles. The majority of today's game consoles like Xbox 360 or Wii are based on the PowerPC architecture whereas the Personal Computer market is dominated by the x86 architecture, or, respectively, its successor, the IA32 architecture.

Processor

The heart of a computer is the processor, as we have already seen in the last section. The whole computer architecture is built around the processor. The operating speed of the processor is a main influence factor for the system's overall performance. While in 1979, a 8086 processor ran at a moderate 4.77 MHz, the processor's clock speed was subsequently increased up to 3 GHz.

To speed up the processor even further, it does not suffice to increase the clock speed, since physical limits will soon be reached due to the energy consumption and the generated heat. Thus, recently, further increase of processing speed is being achieved by parallel processing.

Modern processors are comprised of multiple cores, effectively consisting of two or more processors in one chip. This allows for executing multiple programs concurrently, at least if the operating system supports this. Using a multi-core processor only speeds up your computer when multiple programs can be executed at once, or one program can be concurrently executed on multiple cores.

A web application that runs on a computer with two processors or cores could benefit from running the database on one core, while running the web server on another core. Due to the big number of search and sort operations required, databases are rather processor-intensive applications, so the system's response time should get lower when the database runs on its own processor core.

The advantage of this solution compared to a dedicated database server is that the fast system bus can be used for communication between the web server and database, as opposed to the potentially slow network connection between two systems. The disadvantage is that database and web server are rivaling for the one server's memory, which may become a bottleneck.

Instruction Set

Unfortunately, microprocessors do not understand high level programming languages, but only a certain machine language. The machine language, also called *instruction set*, consists of very simple instructions that, for example, can read a value from memory and put it into a processor register, or branch conditionally or unconditionally.

Machine language has no high-level program constructs as we know them. Without variables, functions, or objects writing complex applications would be extremely tedious, and the resulting unstructured programs would be very hard to read. Instead of programming in machine language or even Assembler, we program in high-level languages that are converted into machine language by a compiler.

Today, all PC processors, regardless of the manufacturer, support the IA32 instruction set, that was introduced when the first 32 bit processor 30386 was released by Intel in 1985.

After Intel, AMD is the second largest manufacturer of IA32 compatible processors. The modern 64 bit AMD processors natively use the AMD64 instruction set, but also support IA32, so they can execute machine code that was written for Intel processors.

In most cases, end-users do not need to care about which processor type their system is build around. Current operating systems and thus applications work on Intel, AMD, and any other IA32 compatible processors. However, you should try to use an operating system that fits your processor's architecture. Especially Linux distributions come compiled for different architectures. IA32 (sometimes also called i386) is the lowest common denominator, whereas on modern 64 bit processors, only the native AMD64 or IA64 versions can unlock the full potential of the system.

Word Length

An old joke says computers are dumb, because they cannot even count to two. In fact, computers are binary, and thus work with only two states, 0 and 1. This smallest information unit is called a *bit*. For historic reasons, a group of eight bits makes up one byte. The number eight was chosen arbitrarily. Some early computers models worked with five or six bit bytes.

A byte consisting of eight bits can represent 256 distinct states. This amount is not sufficient for real work, of course, so a number of bytes are grouped together to form a so-called *word*.

Over time, the word length of computers has increased from one byte (8 bit) to eight bytes (64 bit). At the same time, operating systems and applications were adapted to work with this word length and thus exhaust the processor's full potential.

Windows 3.1 was a 16 bit operating system. Windows 95 was the first 32 bit operating system by Microsoft that was widely used. There is a 32 bit and 64 bit version of

Windows XP and Windows Vista today. The successor of Windows Vista is supposed to be released primarily in a 64 bit version, with an additional 32 bit “legacy” version. All current Unix-based operating systems should be available in a 32 bit and a 64 bit version.

An operating system does not necessarily have to use the full word length of the platform it runs on. You can install a 32 bit operating system on a 64 bit platform, but will be wasting resources.

A main reason for increasing the word length of computers is that the word length also determines the amount of memory the processor can address. One word is used to address one memory cell.

With 32 bit word, you can address 2ˆ32 bytes (4 GB) of memory. For a desktop system, that may still be sufficient, but a powerful server requires much more memory today. Large memory is a main precondition for a server to be fast, since a disk access is about 1,000 times slower than a memory access. Thus, a server that can keep all relevant data in memory is very fast.

With a word length of 64 bit the processor can address 16 exabytes of memory, which roughly equals 16 million terabytes. Even when technology continues to advance at the same pace it does today, it should be 50 years until we hit this memory limit.

A bigger word length also increases the variable range in PHP. Numeric values like integer and float variables are stored in one word. Thus a 32 bit signed integer has a range from -2,147,483,648 to 2,147,483,647.

In contrast, a signed 64 bit integer has a somewhat greater range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Increasing the word length also allows us to store float variables with more precision, since more binary digits are available.

If an integer variable is repeatedly increased, after the largest presentable number has been surpassed, the count starts over with the smallest possible value. This is called overflow. This happens regardless of whether the variable is signed or unsigned.

When migrating, you should keep in mind that the overflow will occur depending on the word length of the system you are working on. If you migrate to a system with a different word length, the overflow will happen at a different time. If an application uses overflows as a feature, the program behavior on the target system will differ.

It gets even more complicated when you are migrating to a platform or an operating system with a smaller word length. Copying of binary data, for example, will not work without actually converting the data. If the data contains values out of the range on the target system, you will work with wrong values if you do not realize that an overflow took place.

Since the precision of floating-point calculations also depends on the word length, the results of calculations on systems with different word length are usually not the same. If you use automated tests to compare the results of floating-point calculations, you might have to increase the delta that is allowed between the results, at least if you migrate to a system with smaller word length. If you migrate to a system with bigger word length, you can probably decrease the delta, since you work with a higher precision.

Byte Order

Most computers today have a 32 bit or 64 bit word length. One word therefore consists of four or eight bytes. In which order are these bytes stored in memory?

Obviously, there are two possibilities. You can start with the low order byte, or start with the high order byte. The first version is called *little-endian*, while the second is called *big-endian*. The meaning of these terms is easy to confuse: big-endian does not mean that the high order byte is last, as the name might suggest. Big-endian means that the highest order byte is stored first, just like we would first write down the digit of highest place value in a decimal number.

The terms big-endian and little-endian stem from Jonathan Swift's Gulliver's Travels, where there were two sovereigns bickering over the question whether to open an egg at the round or the pointed end. Those who opened their eggs at the round end were called big-endians, the others little-endians.

The byte order depends on the computer architecture. That means that when copying binary data from one computer architecture to another, you might have to convert it. While Sparc, PowerPC and Java use the big-endian format, Intel uses a little-endian format. Storing a two-byte word in big-endian and little-endian byte order is shown in Figure 3.2.

It is important to know the endianness of binary data, to correctly interpret the stored multi-byte words. If you stick to one computer architecture, you will not have

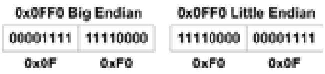


Figure 3.2

to worry about endianness at hardware level. In some cases, though, data formats also need to distinguish between big-endian and little-endian byte order. Multibyte character sets like UTF-16 are an example. You need to make sure that all software components involved know in which endianness data should be interpreted.

When transmitting data over a network, the network protocol defines endianness. The IP protocol, for example, uses big-endian, so on Intel platforms, the byte order is switched when storing or reading the data.

Mounting forms and interfaces

Though this book deals with software migration, we will touch on the basic problem of hardware migration, namely the overwhelming diversity of different components and mounting forms.

Almost every processor model has its own socket specification, requiring an appropriate motherboard. Mounting forms of other system components change less frequently, but still we are looking at a new case shape, motherboard format, or memory module design every couple of years.

There are countless different SCSI standards with different bus width and even more different types of sockets and plugs. Current hard disks have a SATA connector, whereas older hard disks have an ATA connector. Modern laptops come with a PCI Express interface that renders all your PCMCIA cards useless (I learned this lesson the hard way).

It is not always easy to find suitable spare parts for older systems. Either they have long delivery times, or the price is outrageous, so that you could get a much newer component at a lower price.

Operating system

Since Apple released Mac OS X, which is Unix-based, there are basically two operating system families left in the PC world, namely Windows and Unix.

Windows dominates the desktop computer market, while many servers, especially on the Internet, run on Unix. Of course, there are lots of other operating systems for mainframes or real-time operating systems as well, but throughout this book, we will focus on the Windows and Unix families. Mac users are indirectly addressed when Unix is mentioned, and I ask you to forgive me for not directly mentioning you.

There are way more Windows versions and editions than you might have thought. Currently, we have Vista which comes in a Starter, Home Basic, Home Premium, Business, Enterprise and Ultimate edition, Windows Server 2003, Windows XP in a Home, Professional, and a Media Center edition, Windows 2000 with a Professional, Server, Advanced Server and Datacenter Server edition, Windows NT and for reasons of nostalgia even Windows ME, Windows 98, and Windows 95. All these versions and editions are available in different languages, yet increasing the variety.

The predominant representative of the Unix family today is probably Linux, even though there are alternatives like Solaris or BSD that are also widely used. In the Linux world, there exists a plethora of different distributions, for example by SuSE (Novell), RedHat, Debian, Ubuntu, Gentoo, Fedora, or Slackware, just to name some of them. Each of these distributions has different versions, and sometimes even offer a different server and a desktop edition, let aside the different support options you can purchase.

Though changing a system environment usually does not cause too much trouble, as long as you stick to one operating system edition, there are differences between the various versions and editions, as you can tell just by looking at the sheer diversity alone. Windows sometimes even tends to behave differently when installed multiple times on identical hardware.

The plethora of Linux distributions differ greatly from each other, especially in the way the install and manage software. While Debian and thus also Ubuntu use APT, the advanced packaging tool, SuSE has a custom installer YAST, which is the acronym for yet another setup tool to manage rpm packages.

The Linux Standard Base (LSB) (`http://www.linux-foundation.org/en/LSB`) tries to increase compatibility between the different distributions. LSB, for example, defines what the directory structure of a system should look like, and which programs and libraries have to be available on any system.

A PHP application running on Linux should make as few assumptions about the system as possible, to be portable to other Linux distributions. Unix is more standardized, so it is easier to make valid assumptions which, in turn, might lock you in to a certain vendor or version, though.

Fortunately, PHP programs are not tightly coupled to the operating system by nature. With a little effort, you can create portable PHP applications that run on Windows and on Unix. This makes migrating to another operating system or operating system family much easier. If the application was not written with portability in mind, you will likely have to do more work when you are migrating.

We already know that most modern processors have more than one core. To allow efficient use of these cores, the operating system must explicitly support multi-core processors. All current operating systems like Unix, the SMP Linux kernel, Windows XP and Windows Vista do support multi-core processors. On Windows, it might be necessary to install a special driver, though.

On a multi-core architecture, the operating system shares the available computing time between all running applications. Every application is allowed to use the processor for a little while, then it is preempted, and another application starts using the processor. While an application is not using the processor, it is put in a waiting state, until it may use the processor again. If this switch is done rapidly and very quickly, it seems to the user as if multiple applications were executed at the same time. This way of working, as we know, is called multitasking, and each application is executed in a so-called process or task.

The time it takes to switch between tasks, however, is unproductive time. To reduce the number of necessary task switches, a task can be subdivided into a number of threads. A thread is a lightweight process inside another process. All threads within one task share the same memory.

Not every program can be executed in multiple threads ("multi-threaded"), though. Under certain circumstances, even using a global variable can make a program fail in a multi-threaded environment. To make a program thread-safe, careful programming is required. The problem with programs that are not thread-safe is that they might run flawless, but cause trouble under certain conditions. It is virtually impossible to reproduce these conditions, so it is particularly difficult to create a test that demonstrates the problem.

The core of PHP is usually thread-safe, but can also be compiled in a non-threadsafe way. If you are not sure how the PHP you are using has been compiled, call the function `phpinfo()` and look for the line *Thread Safety*.

Since Windows is multithreaded, PHP for Windows must usually be compiled thread-safe. IIS, Apache 1.3 and the Apache 2.x series run and have always been running multi-threaded under Windows.

Unix has been supporting multithreading for some years now as well. Using Apache2 on Unix, you can choose between two execution models, namely the multi-threaded Worker MPM (multi-processing module), or the single-threaded Prefork-MPM.

Executing a thread-safe application in a non-threaded environment does work, but will probably impose some performance penalty. If you are sure that PHP will always be run single-threaded, you might try and compile it non-threadsafe to increase performance.

The majority of commonly used PHP extensions are thread-safe as well. Some extensions like GD or IMAP are known not to be thread-safe. For other extensions, it is just not known for sure whether the extension is thread-safe or not. This is because PHP extensions use external libraries, and you sometimes just do not know whether a library is thread-safe. The author of a PHP extension has little control over exactly which libraries are used to compile the extension, and how they are compiled.

At PHP level, the concepts of multitasking or multithreading do not exist. Every PHP program runs in its own memory space, so PHP developers do not have to care about thread safety problems. Still, when multiple PHP programs access external resources like databases or files are executed concurrently, synchronization problems and resulting inconsistencies similar to the problems of multithreading can arise. The PHP programmer has to deal with these issues to make an application stable and robust.

Word Length

An operating system is laid out to a certain word length. This means that the operating system has been compiled into a machine language of that very word length. The underlying platform determines the maximum possible word length for the operating system to use.

Most Windows operating systems contain an environment to execute code of lesser word length. In Windows 95 you could execute old 16 bit code. Current 64 bit Windows versions contain a Win32 subsystem to execute 32 bit programs. This allows Windows to transparently execute code of lesser word length. This is not possible in Linux, but since most code is compiled on the target system, and thus for the available word length, this is not a problem.

The maximum word length PHP can use is determined by the operating system's word length, not the platform's word length. This is evident, since PHP does not access the hardware directly, but uses the operating system to do so.

The built-in constant `PHP_INT_SIZE` tells you the word length PHP is using. The constant `PHP_INT_MAX` contains the maximum integer value for that word length.

Line Endings

For historic reasons, there are two characters to end a line, carriage return and newline. This dates back to electric typewriters. As computers spread, the manufacturers did not agree on a ways of representing the end of a line. Today, every operating system family uses their own line endings, as Table 3.1 shows.

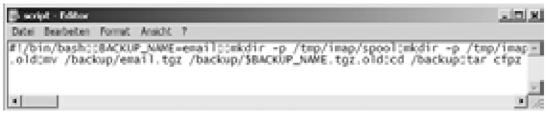
Operating System	Line Ending	Hex Value
Windows	\r\n	0x0a 0x0d
Unix	\n	0x0d
Macintosh	\r	0x0a

In a HTML file, it does not matter which kind end-of-line characters are used. A browser interprets any number of whitespace characters (blanks, tabs, carriage return, line feed) as one blank. This makes it possible to indent HTML code without screwing up the rendered page. A new line is created with a `<br>` tag.

For PHP source code, it does also not matter which line endings you use, since the PHP parser ignores line endings when compiling the code. If your application contains created files, it is best to use the format of the operating system your application is running on. You can use the constant `PHP_EOL` to determine the operating system's line ending. Thus, this constant can also be used as a simple operating system detector.

Text files (including PHP source code) might not be displayed correctly when opened on a different operating system, like Figure 3.3 shows. Most modern editors (except Windows Notepad, maybe) detect and handle all line endings correctly and can convert the files on request. To redeem Windows, it should be noted that

Wordpad can handle Windows and Unix line endings and displays files from both systems correctly. Figure 3.3 shows the wrong display of a file with Unix line endings in Windows Notepad.



```
#!/bin/bash::BACKUP_NAME=email::mkdir -p /tmp/imap/spool:mkdir -p /tmp/imap
.old:mv /backup/email.tgz /backup/$BACKUP_NAME.tgz.old:cd /backup:tar cfpz
```

Figure 3.3

Unfortunately, there are not only display problems with “wrong” line endings. On Unix, you can add a so-called *shebang* to a text file. That means that the first line of the file starts with the special character combination `#!` followed by the full path to a command to execute. You can use a shebang that points to the PHP executable to start a PHP source file by calling it like an executable program.

For the following example, let us assume the following program has Windows line endings, and the PHP interpreter is installed in `/usr/local/bin/php`:

```
#!/usr/local/bin/php

<?php

  echo 'Hello World';

?>
```

Directly calling the PHP interpreter with the file name as parameter works as expected:

```
> php test.php

Hello World
```

When we call the file directly, to start PHP by the shebang line, we are out of luck, since Unix gets confused by the Windows line ending and thus cannot locate the PHP interpreter:

```
> ./test.php
```

```
: bad interpreter: No such file or directory
```

The problem with line endings is that you cannot see them and you cannot tell without analyzing which line ending a file has. Some, but not all editors on Unix display the "surplus" `\r` as `ˆM`.

If you want to be really sure about the line endings of a file, display a hex dump or, on Windows, open the file in a hex editor:

```
> od -h test.php

0000000 2123 752f 7273 6c2f 636f 6c61 622f 6e69
0000020 702f 7068 0a0d 0a0d 3f3c 6870 0d70 0d0a
0000040 200a 6520 6863 206f 4827 6c65 6f6c 5720
0000060 726f 646c 3b27 0a0d 0a0d 3e3f 0a0d
```

The highlighted values show that this file has Windows line endings. To convert files between Windows and Unix format, you can use the Unix command line tools `unix2dos` and `dos2unix` as well:

```
> dos2unix test.php

dos2unix: converting file test.php to UNIX format ...
```

Unfortunately, these tools create the converted files without the `execute` right, so you have to explicitly set the execute bit manually after conversion:

```
chmof 755 test.php
```

You can also use a small PHP program to convert files, which might be a good solution on Windows systems. The program reads the file, replaces the line endings by `str_replace()`, then saves the file again:

```
<?php

  $content = file_get_contents('dos_format.txt');
  $content = str_replace("\r\n", "\n", $content);
  file_put_contents('unix_format.txt', $content);
```

```
?>
```

Never attempt to replace line endings in a binary file. This will most likely make the file unusable.

Access Rights

Modern operating systems can be used by multiple users at once, which means the access to system resources and especially files has to be limited in a sensible way. Unfortunately, neither all users on a system nor all programs have only good intent.

The operating system must protect the programs and data of each user against accidental or deliberate unauthorized access. In the file system, this protection is provided by defining access rights to each file and directory. These access rights are implemented differently in Unix and Windows.

While Unix users are very used to working with file access rights, Windows users rarely modify file access rights. This is mostly due to the fact that Windows users usually work as Administrator and thus can access every file anyway. In fact, file access rights on Windows are more sophisticated than on Unix, but this also makes them more difficult to understand.

Unix differentiates between three different access rights, namely read, write and execute. These three rights can be set for each file and directory, individually for three categories of users, namely the owner of the file, a user group the file is assigned to, and all users.

When listing the files these rights are indicated by the letters `r` for read, `w` for write, and `x` for execute. First, the user's rights are shown, then the group's rights, then everybody's access rights:

```
> ls -al

total 24
drwxr-xr-x   2 root   root  4096 Dec 14 12:53 .
drwx------  13 root   root  4096 Dec 14 12:51 ..
-rwx------   1 wwwrun www     62 Dec 12 19:37 index.php
-rwxr-xr-x   1 root   users 5201 Nov 17  2006 library.php
-rw-r--r--   1 root   root  2127 Nov 17  2006 test.txt
```

In this example, the file index.php may only be read, written to, and be executed by the owner wwwrun.

Since PHP files are loaded by the PHP interpreter, they require no execute right be set. To run a PHP file directly by adding a shebang as we have learned in the previous section, you must set the execute right. Since internally, each right is represented by one bit, sometimes we will also refer to this by "setting the execute bit."

Windows uses access control lists (ACL) to regulate file access. For each object, an access control list defines which actions are allowed or forbidden for certain users or groups. ACLs can be inherited, for example from a directory to the contained files. The effective restrictions for an object are calculated from all entries of the access control list attached to it.

ACLs are very flexible and allow a fine-grained level of access control, but are rather difficult to understand. You will always have to look at the effective rights on an object to know what access restrictions really are in place for it. To display access rights, Windows uses the letters N for no access, R for read, W for write, C for create and F for full access.

```
> cacls phpinfo.php

C:\www\phpinfo.php PREDEFINED\Administrators:F
                   NT-AUTHORITY\SYSTEM:F
                   MY_COMPUTER\Steve:F
                   PREDEFINED\User:R
```

In this example, the user "Steve" has full access to the file. All administrators (which is a user group, rather than the user Administrator) and the system user, which is the account services usually run under, also have full access to the file. In addition, all users have read rights by default.

On Unix there are several commands to modify access rights for a file or directory (chmod), or change the owner (chown) or the group (chgrp) a file belongs to. These three Unix commands have a PHP equivalent which have no effect under Windows, though no warning or error message is displayed when they are uses on Windows.

Paths and File Names

One of the main responsibilities of an operating system is the file system administration. In regard to the file system, the two operating system families Windows and Unix differ in various aspects.

All common file systems use a tree structure. Starting with a root directory, each directory can contain files or subdirectories. Every file can be identified by its absolute path in the file system. Unix and Windows use a different character as directory separator, though. While Unix uses the forward slash `/`, Windows uses a backslash `\`
.

Drives on Windows

On Unix, the whole file system is one organized in one tree. You can mount additional file systems at any point in the tree. On Windows, every volume has its own file system tree. (Newer Windows versions also allow you to mount another volume in the file system tree, but this feature is rarely used.) Each volume is identified by a drive letter, followed by a colon.

Traditionally, the first hard disk is drive `c:`. Usually, but not always, the computer boots from this drive. The drive letters `a:` and `b:` are reserved for the floppy disk drives, when present. The drive letters following `c:` are used by additional hard disks, partitions, or other drives like CD-ROM or DVD. Windows assigns them in an order that is not always really comprehensible.

All systems know the concept of the current working directory, which usually is set to an appropriate value by the running application. For PHP scripts, for example, this is the directory where the script that is running is located. The concept of the current working directory allows for using relative paths to address other files. These relative paths stay the same regardless of the absolute location of the application in the directory tree. Use of relative paths makes applications more portable to other systems, thus you should avoid using absolute paths whenever possible.

Windows, at least at the command line, also has a current drive. Every drive has its own current working directory. When you switch between Windows and Unix, drive letters in path names are problematic, because Unix does not have this concept. One could try to restrict themselves to only work on the system drive on Windows (which usually is `c:`) and remove all drive letters from the path names.

As soon as the program is executed on a Windows system where the web server's `htdocs` directory (or, for IIS uses, the `wwwroot` directory) is not located on drive `c:`, it is not possible to access the temporary directories on drive `c:` without a drive letter.

The root directory on Unix is `/`, while on Windows it is `\` . Unix has only one system-wide root directory, whereas in Windows every volume has a root directory. Keep in mind that URLs have a forward slash as directory separator on Windows as well, even in file URLs. A valid file URL on Windows would be `file://localhost/c:/windows/php.ini`. For `localhost`, you could replace the hostname by another forward slash after the colon.

Apparently, a file URL like this is not portable between Windows and Unix. Since Windows has a completely different directory structure than Unix anyway, this should not be a big problem.

Upper and Lower Case

Windows does not distinguish between upper and lower case in filenames and paths, while Unix does. In other words, Unix is case-sensitive, while Windows is case-insensitive. While `Filename.php`, `FILENAME.php` and `FileName.php` are three different files on Unix, all possible combinations of upper and lower case in a filename refer to one and the same file on Windows. This difference in file handling can cause problems when migrating between different operating system families. We will cover these problems in greater detail in Chapter 8.

Allowed Special Characters

Not all characters are allowed in file names. For obvious reasons, the directory separator may not appear in a file or directory name, since this would make path names ambiguous.

On Unix, all characters except for the forward slash and the *NULL* character `\0` are allowed in filenames. Certain special characters like quotes or double quotes have to be escaped by a backslash.

Windows is more restrictive. The following special characters are not allowed in file names:

- <

- >
- ?
- "
- :
- |
- \
- /
- *

Also, on Windows, certain reserved words may not be used in file names, like `CON` or `NUL`. To make sure that file names are portable between Unix and Windows or any other platform, you should use conservative names, for example limiting yourself to alphanumeric characters, underscores and blanks. It is rather easy to add a regular expression to your program that checks whether file names consist of only these characters. If a user uploads a file name with exotic characters, you could rename the file and replace any non-standard character by an underscore, for example.

Current Windows versions are Unicode-based (see later in this chapter), which means that except for the special characters listed above all Unicode characters may be used in filenames. However, if an application is not Unicode-based, you will likely run into trouble with exotic characters, which is another reason to stick with conservative file and path names.

Especially on Windows, path and file names containing space characters can cause problems, for example when used as parameters of command line calls. The operating system can not tell where one file name ends and the next parameter starts. To fix this problem, you must quote file and path names that contain spaces, which in turn causes problems with file names containing quotes, as these quotes inside the file name would have to be escaped.

It can make your life a little easier if you use underscores instead of space characters to separate words in file names. Since in most cases you will have to work with path names containing spaces, especially on Windows, you should consistently quote all path names.

Windows also has the bad habit of automatically removing trailing spaces from file names. If you copy such a file from Unix to Windows, the name implicitly changes, which usually means that the file will not be found any more. Therefore, never use trailing spaces in filenames.

Temporary Files

Applications do not only need the system memory they have been assigned. Some algorithms have been especially designed to work with external data. This allows, for example, for sorting of large data sets that would not fit into memory.

Every operating system has a temporary directory, in which users can create and modify files. This temporary directory is not meant for long-term file storage and is usually cleaned out every now and then. On Unix, the temporary directory for all users is `/tmp`. In this directory, every user can only access and delete their own files, so using one temporary directory for all users is not an immediate security risk.

On Windows, every user has a private temporary directory, and there is an additional, system-wide temporary directory. The environment variable `%temp%` contains the path of the current user's temporary directory:

```
> echo %temp%

C:\DOCUME~1\steve\LOCALS~1\Temp
```

Those strange names ending in `~1` ensure backwards compatibility to the old 8.3 file names that are a relic of DOS times. Still today, Windows sometime outputs them at the command line. Do not use these names, but the real path instead:

```
C:\Documents and Settings\steve\Local Settings\Temp
```

The system-wide temporary directory is `%windir%\temp`, with the environment variable `%windir%` containing the drive letter and path of the directory Windows has been installed to. Usually, this is `c:\windows`, but a different name could have been used as well.

The Search Path

We already know that the file system is organized as a tree. Executable programs are stored to various different directories. To relieve the user from memorizing the full path to each executable program, most operating systems automatically search different directories for a given file name if it is called without an absolute path.

The list of directories to search is called the search path. This search path is a system-wide setting stored in an environment variable. This environment variables is called `PATH` on Windows and on Unix. In both operating system families, the path to the executable PHP program should be part of this search path, to allow for easy execution of PHP programs at the command line.

```
> echo %PATH%

C:\WINDOWS\system32;C:\WINDOWS;c:\php
```

On Unix, the environment variable is accessed by `$PATH` instead of `%PATH%`. Please note that the displayed path has been edited for brevity in both examples:

```
> echo $PATH

/sbin:/usr/local/bin:/usr/bin:/bin
```

Thanks to the search path, the exact location of a program need not be known to execute it. Programs can be started from any working directory, which makes everyday work much easier and the applications more portable to different systems. The executable files need not be kept in the same directory on various systems, as long as they can be found using the search path.

The individual entries of the search path must be separated by a special character. To avoid ambiguities, this character should not occur in valid path names. Unix uses a colon, whereas Windows uses a semicolon as path separator.

PHP has a built-in constant `PATH_SEPARATOR` that contains the path separator of the operating system PHP is running on. To make a PHP program portable, you should always use this constant rather than a hard-coded character.

The PHP search path that is configured by the `php.ini` setting `include_path`, uses the same separator as the system search path.

Character Sets

Computers process data in the form of strings mostly, so defining a character set is a very important precondition. Just like number representations have to be standardized to make different systems compatible and allow data exchange between them, it has to be regulated which character is to be represented by which byte value. Internally, every string is stored as a byte sequence.

The ASCII (American Standard Code for Information Interchange) character set can be viewed as the mother of all modern character sets. Its origins date back to typewriters and teleprinters. ASCII is originally a seven bit character set, which allows for encoding 128 different characters, some of them non-printable. The printable ASCII characters are:

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^
‘abcdefghijklmnopqrstuvwxyz{|}~
```

Since ASCII only has 128 characters, there is no room for special characters and the diacritic marks used in many European languages. To accommodate this, other character sets based on ASCII were defined, that used 8 bits to encode one character.

Of course, the 256 characters that can be encoded by 8 bits are not enough to represent all special characters, especially not the complex Asian characters. Different character sets for the various countries and cultures emerged.

Until the end of the last century, and maybe even today, the most commonly used family of character sets was ISO-8859. This norm contains 15 character sets that all conform to ASCII in the first 128 characters. The commonly used western character set is ISO-8859-1, which is often also called Latin-1. Other ISO-8859 character sets are Baltic (ISO 8859-3), Cyrillic (ISO 8859-5), Arabic (ISO-8859-6), Greek (ISO-8859-7), Hebrew (ISO-8859-8), Turkish (ISO-8859-9), Thai (ISO-8859-11) and Celtic (ISO-8859-14). ISO-8859-15 corresponds to ISO-8859-1, but contains the Euro sign (€) to accommodate the new common European currency introduced in 1999.

To solve the problems arising from the use of different character sets, 1991 the international standard Unicode was brought into being. Unicode is based on a standardized encoding of all characters known throughout the world. The first Unicode version had 65,536 characters, which proved to be not sufficient on the longer run.

Currently, Unicode has about 100,000 characters, with room for over a million characters. There are 19 different blank characters alone in Unicode, some of which have no width! Blank characters without width are used in Thai language, for example, where no space is written between words.

Strictly speaking, Unicode has no characters, but code points. A code point is an entry in the Unicode character table. The difference between a code point and a character is that one character can be created by combining various code points. Unicode does not only know the German umlaut character ä, for example, but also allows to combine this character from the two code points a and the vertical colon. Unicode contains a set of rules, to make the comparison of "built-in" code points with combined code points possible.

The Unicode standard also contains sort criteria, rules for converting case and for inserting line breaks, as not all languages allow inserting line breaks where blank characters or hyphens are.

There are various possibilities to encode Unicode characters. The version that is easiest to understand is UTF-32, which uses 4 Bytes to encode each Unicode character. UTF-32 requires four times the memory of an ASCII representation, which is not always desirable.

To balance length and ease of use, other variable-length encodings have been developed. The oldest one is UTF-16, which uses two bytes to encode the most commonly used Unicode characters, and four bytes to encode the rarely used characters. Windows and Java use UTF-16, and also the Unicode support in PHP 6 will be based on UTF-16.

An alternative variable-length encoding is UTF-8, which uses between one and four bytes to encode one character. A special feature of UTF-8 is that the first 128 characters that are encoded in one byte match the ASCII character set. Therefore, every valid ASCII string is also a valid UTF-8 string, making UTF-8 fully backwards-compatible to ASCII.

Because of the backwards compatibility to ASCII, UTF-8 is the preferred character encoding on the Internet. All communication protocols on the Internet are required to support UTF-8. UTF-8 is the most space-saving Unicode encoding, since compared to UTF-16 only one byte is used to encode common ASCII characters.

Windows has been Unicode-based since Windows NT, but in the early years most applications would not support Unicode. Today, you can expect common applications to fully support Unicode, but there is still a major technical problem with Unicode support: The commonly used TrueType or OpenType fonts and also the Postscript fonts that are frequently used in printers can contain 65,536 characters at most. Since Unicode today has 100,000 characters, not all Unicode characters can be contained in one font. When a character cannot be displayed, it is usually substituted by a question mark or sometimes a box character.

To successfully display a Unicode string, the used font must contain all characters (code points, to be exactly) that occur in the string. On Windows, you can install the Arial Unicode Font, which is distributed with certain versions of Microsoft Office and contains about 40,000 characters. Other characters sets shipped with Windows like Lucida Sans Unicode contain a mere 2,000 characters and can only display the characters contained in the ISO-8859 family. The Arial Unicode font is over 22 MB in size, so on older systems with not much available RAM, it may not always be a wise decision to install this font.

All current Unix and Linux distributions should use UTF-8 as the default encoding. Still, the problem that the character sets cannot display every Unicode character is the same on these operating systems. A list of different and partially free Unicode character sets can be found at Wikipedia.

One of the biggest problems with different character encodings is that one cannot easily tell the encoding of a file by looking at it. For the operating system, a file is just a sequence of bytes. It is the programmer's responsibility to make sure a file is interpreted correctly when it is read.

Unfortunately, many developers do not really give attention to the problems of character encoding. An application may work fine on one system with a given configuration when only using western characters. When migrating to another system, display problems often surface because data is not encoded in the character set the operating system and the applications expect.

Character sets and encodings are a complex topic that touches different system components like database, operating system PHP, and the browser. Chapter 3 shows typical problems and solutions in regard to character sets.

Databases

Most applications today deal with larger amounts of data than anybody could have imagined a couple of years ago. In the early 1980s a floppy disk was 5 1/4“ in size and held 170 KB. The floppy disks uses for IBM compatible personal computers were 3,5” in size and had a capacity of 1.44 MB. Today, floppy disk drives are rarely used, but have been replaced by USB sticks that hold up to several Gigabytes.

Large amounts of data are usually stored in databases. Databases offer random access to the stored data and are optimized for fast and efficient sort and search operations. Since originally storage space was limited and expensive, databases were designed to eliminate as much data redundancy as possible by storing data in several tables that were linked together. This process of eliminating redundancy is called normalization.

Today, much more storage space is available, so eliminating redundancy has become less important. In many larger systems, a certain degree of data redundancy is accepted to allow for quicker read access.

Web applications are usually read-intensive. A typical web 2.0 application like a portal does seven to ten read accesses on one write access. Databases for web applications are therefore optimized for read access, and slower writes are being tolerated.

The speed of concurrent read accesses is mainly determined by the database's locking strategy. Some databases lock a table or even the whole database on write access. Every write access must be completed before the next one can start.

Alternatively, a database can lock only the one table row that is currently being written to. This allows for executing multiple write operations in parallel. Locking individual table rows has a higher administration overhead than locking tables, but allows for potentially faster write access. The amount of parallel read and write accesses to a database determines which locking strategy will ensure the best application performance.

SQL Does Not Equal SQL

SQL is the standard query language for databases. SQL originates from the query language SEQUEL and was standardized for the first time in 1986. The second SQL version was standardized as SQL92 in 1992. To take object orientation more into account, a third version of SQL was standardized in 1999 as SQL3.

In 2003 and 2006 even newer versions of the SQL standards have been published, but they are still not very relevant in the field today. Most databases today more or less support the SQL99 standard. Unfortunately, the situation with SQL in database is similar to the situation with HTML at the heights of the browsers wars: every vendor adds proprietary extensions to the standard.

A commonly known extension to the SQL standard is the possibility of limiting the result set of a query. In MySQL, this is achieved by a `LIMIT` clause: instead of skipping over 1000 records to access records 1001 to 1010, the clause `LIMIT 1001,1010` can be appended to the query to access these records directly. The `LIMIT` feature was invented by Rasmus Lerdorf, by the way, who had originally implemented this feature in the MSQL database.

Proprietary extensions to the SQL standard can be very useful, but are also a means of winning over a customer. As a developer, you should make a conscious decision whether or not to use proprietary database features. They will help you reach your goals more quickly, and benefit from higher performance, but take away the freedom to quickly switch to another database vendor, should the project require this.

To keep an application portable, all SQL queries should strictly be limited to standard SQL, unless you want to abstract the SQL code for each database, which means more effort when developing the application, but comes with the benefit of higher performance.

When migrating, an important question to be answered is which SQL standard a database supports and what the deviations are. Today, virtually no database is 100% compatible to a certain SQL standard. This makes it rather hard to create database-independent applications. When migrating, you will ultimately have to check each and every SQL statement on the target system and make the necessary adjustments to them.

Program Code in the Database

Databases not only store data, but can also execute program code. While from a theoretical point of view, it does not make a difference whether program code is being executed in the database or at PHP level, but it can increase application performance when code is executed in the database.

Code running in the database can quickly access the whole database, without latency of communication between the database and the application. If various applications access the database, redundant code can be avoided by moving code from the applications to the database.

Not all programmers are in favor of executing code in the database, because this can mean moving application logic to the database, which contradicts the separation of the application into a presentation, logic and data access layer.

Program code in the database is also called *stored procedures*. Most databases support a full procedural programming language with local variables, branches and loops to write stored procedures. Unfortunately, these programming languages are proprietary and not portable between databases; Oracle stored procedures are written in PL/SQL, while stored procedures for IBM's DB2 database are written in SQL PL (SQL Procedural Language). Microsoft databases support T-SQL (Transact SQL), while stored procedures for PostgreSQL must be written in PL/pgSQL. SQLite does not support stored procedures at all. Some databases also support executing stored procedures in alternative programming languages. Oracle, for example, allows for executing Java code, while PostgreSQL supports various scripting languages, among them PL/php.

MySQL has been supporting stored procedures since version 5.0. This feature is rarely used with PHP applications today, since most PHP application were written for older MySQL versions.

When migrating a database, the first step is to gain an overview over all stored procedures. Each database requires a different special SQL statement to list all stored procedures:

- DB2: SELECT * FROM syscat.routines;
- MySQL: SELECT * FROM information_schema.routines;
- Oracle: SELECT * FROM all_procedures;
- PostgreSQL: SELECT * FROM information_schema.routines;
- SQL Server: SELECT * FROM sysobjects WHERE type='P' and category=0

Those statements may not work with older versions of the databases. If in doubt, consult your database documentation.

Just like regular SQL statements, you will have to make sure that all stored procedures work as expected on the target system. While migrating to another version of the same database usually is rather painless, you will probably have to rewrite all stored procedures when you switch to another database vendor.

To ensure maximum portability of your application, you should avoid using stored procedures. On the other hand, when you can be sure that your application will always run on a certain database, why should you not take advantage of the increased performance of stored procedures? Keep in mind, however, that this locks you in to the respective database vendor.

Data Types

Just like programming languages, databases use different data types as well to store and process data efficiently. The basic data types `int`, `float` and `boolean` correspond to the data types also used in programming languages. Most databases know additional datatypes like `varchar`, `date`, `time`, `timestamp`, and `blobs` (binary large objects) or `clobs` (character large objects) to store larger amounts of data.

Not every database supports exactly the same data types or interprets them like other databases. Time and date fields are particularly problematic. Even for basic data types like int, you should make sure that the data type has the same range on the target database (see Chapter 3). Floating-point numbers can have a different precision, which can lead to different calculation results when another database is used. If you use automated tests, some of them might fail.

The size limits of various data types like `varchar`, `blob` and `clob` differ between databases as well. MySQL, for example, allowed a maximum length of 255 characters for a `varchar` field until version 5.0.3, while today the limit is 65,536 characters.

Character Sets

When storing strings in the database, just like with operating systems, a decision has to be made as to how to encode strings. If the database uses a different character set than the application, the strings must be converted when reading from and writing to the database. This conversion needs computing time and will decrease the database and overall system performance.

To avoid unnecessary conversions, it is important to know which encodings are used by the different system components. The best idea might be to use Unicode and UTF-8 throughout. In addition to storing the strings as Unicode, you must also make sure that the database server and database driver use the same character set.

Since UTF-8 is backwards compatible to ASCII, every ASCII string is a valid UTF-8 string as well. This makes the transition from ASCII to ISO-8859 (which is ASCII compatible as well) to UTF-8 deceptively easy. As long as only "regular" characters are used, it does not matter whether you work with ASCII or UTF-8 in your application. As soon as you start using special or exotic characters, problems will arise when you don't consistently use UTF-8. Please refer to the section entitled "Character Sets" earlier in this chapter for more information.

If you use a database as a mere dumb data storage, you can avoid most problems with character sets and encodings. Every string is stored as a binary sequence of bytes. In this case, you cannot expect the database to sort or work on substrings correctly. As soon as you starting working with multibyte character sets, the database must know the encoding that is being used, otherwise it will treat strings as if one byte would correspond to one character, which will lead to spurious results when calculating the string length or working with substrings.

Databases also contain rules describing how to sort data. These sorting rules are also called collation. Most databases allow setting the collation independently from the character set to adjust to local commodities, as alphabetical sorting differs between languages and countries, for example in the way umlaut characters are treated.

If you work with multibyte character sets that use variable-length encodings, you cannot predict how much storage space you will need for a string. When using UTF-8, one character can be up to four bytes in size. You may need to adjust the size of `BLOB` and `CLOB` fields to store multibyte strings.

For every database, you must define the character set to be used. Some databases like MySQL even allow defining the character set by table. This can help you save a lot of memory, when you can be certain that only ASCII characters are going to be stored in a certain table.

You should stick to the character set of each database table. If you declare a table as UTF-8 and store ISO-8859-1 data into it, errors will occur when special characters are used (as will be discussed later in this chapter). It is equally important to define the character set PHP uses to send data to the database, and expects to receive data from the database. In both cases, the encodings used need not be the same encoding the database itself uses.

According to the SQL standard, the clause `SET NAMES <character-set>` must be used to set the character set of a database connection. MySQL offers three additional proprietary SQL clauses to define which character set to use:

```
SET character_set_client = <character-set>
SET character_set_connection = <character-set>
SET character_set_results = <character-set>
```

These statements allow you to separately define the character set used by the database client, by the database connection and for the results returned by the database.

Keep in mind that converting between character sets requires resources. From that viewpoint, I would recommend working with UTF-8 data throughout all system components, but PHP does currently not support multibyte character sets very well, as we will see in chapter 3.9.

Backup and Restore

Reliable backups are vital in IT. Regardless of how dependable a system or may be, when it gets destroyed by lightning, water, or vandalism, you are forced to setup a new system and restore software, data, and configuration.

To successfully restore a system, you will need not only the backup itself, but also a proven procedure. Too many people have already made the excruciating experience that their backup was not fit to restore the damaged system. To avoid this, you should try out the restore procedure as early as possible, but without pressure.

Sometimes, no data at all is written to the backup medium, or the data is incomplete or inconsistent. This is no problem if the old system is still available, and you can still access the missing data, but if some days have passed between the backup and the restore attempt, you can get into trouble rather quickly. If your hard disk is faulty or the database became corrupted, it may take a few days until you even realize that you have to restore a backup. You may in fact have created several generations of unusable backups before you notice.

A data backup is not only a safety belt protecting you in case of a catastrophe, but also an important precondition for a migration, where, in essence, you want to run an application with all its data on a different or updated system at a given point in time.

Since databases are often optimized while in production, it is important to have a backup of the current database structure, not the original database structure used when the application was deployed. You do not want to lose all indexes and optimizations you have made to the database when restoring or migrating a system.

There are two general strategies for backing up a database. Since any database is ultimately kept in the file system, it may be tempting to just backup the binary database files. Such a binary backup is easy to create, but has a high risk of being inconsistent. As databases get large, the backup can take minutes to complete, and if the database content is changed while the backup runs, the backup will be inconsistent and potentially unusable.

MySQL, for example, creates a directory for each database and one file for each database table. If you backup these files one by one, the backup will not contain changes to files that were already copied when the database changes. This may destroy the referential integrity of your backup database.

To make sure a binary database backup is consistent, you should shutdown the database. If your application must be available 24x7, you will have to backup the database while it is running. This is called a *hot backup*. Some databases have special features that allow creating consistent hot backups, for example by logging all changes that were made to the database while the backup ran.

Another strategy is to create an export or dump of the database. A database dump, also called SQL dump, is a text file containing all the SQL statements required to recreate the database. Since the database export contains all data in clear text, this kind of backup is usually more reliable than a binary backup. Even when single rows or tables of a database are destroyed, it is possible to retrieve the rest of the data. When a binary database backup is damaged, it can be hard, and in fact often is impossible, to salvage or retrieve the contained data.

Make sure that you dump the database structure and the data so that your export contains the SQL statements to create the database tables and fill them with data. For MySQL, you can create an export using the frequently used tool phpMyAdmin or the `mysqldump` utility that comes with the MySQL distribution. A backup using the command line-tool is easier to automate with `cron`, `at`, or `schtasks`, depending on the operating system.

If the database gets large, restoring it through phpMyAdmin can be problematic due to limits in upload size or PHP script execution time. It is quicker and more reliable to restore a database at the command line, if you have shell access.

When exporting the database, you have to make sure that the export is consistent, since the database is exported table by table and row by row. When changes are made to the database while it is being exported, again, the backup will be inconsistent.

Some databases or backup tools help you create consistent database exports. One possibility to achieve this is to temporarily shut down every application accessing the database, and thus ensure that the database is not changed. It can be sufficient to prevent write access to the database, since read accesses are harmless. This way, the application can still be used while its data is being exported.

To make sure that your database backup is consistent, you can shut down the database, restart it in single user mode, export it, and then restart the database in multi user mode. Not all databases support a single user mode, however.

It is up to you which of the two backup strategies you want to use. Personally, I prefer exporting databases. Different versions of the same database are not necessarily binary compatible. The same holds true for the same database running on different operating systems. When you plan to migrate to a database of a different vendor, you will probably have to export the database and convert the SQL dump to fit the new database.

Web Server

At first sight, a web server is no complex software. It receives HTTP requests and sends back files from the local file system. Since HTTP is stateless, the necessary connection management is not very complex.

In reality, web servers are very complex pieces of software, optimized to run permanently under high load, efficiently handle peak demand, and guarantee a short answering time even for a high number of parallel requests.

Today's web servers do not only serve static content, but support creating dynamic web pages utilizing languages like PHP, Perl, Python or Ruby. The client browser does not really care whether a web page was delivered statically or created dynamically on the fly.

There are many different web servers. Most of them are free software and run on common operating systems. The most commonly used web servers are Apache and Microsoft's Internet Information Services (IIS). IIS is free, but not open source, and can only be used in production in Windows Server operating systems. On regular Windows operating systems like Windows XP, only 10 client connections are allowed for IIS, which may be suitable for a development or test system, but not for a production system.

Apache has a market share of about 50%, followed by IIS with a market share of 35%. Other, less frequently used web servers are Zeus, Sun Webserver or lighttpd. Especially in conjunction with PHP, Apache is the most commonly used web server, so we will mainly focus on migrating Apache web servers throughout this book.

Apache and Apache2

Since 2000, there are two major versions of Apache, namely Apache 1.3 and Apache2. Apache2 currently has two minor version branches, 2.0 and 2.2, which has some interesting new features like load balancing, improved caching, and support for large files.

Apache2 is more modular than the first version. The web server itself is independent from the operating system it runs on. This is made possible by a software abstraction layer, the Apache Portable Runtime (APR). This design allows for performance optimizations on each operating system, especially on Windows.

Through so-called multi processing modules (MPM) Apache2 can run in different modes, to better utilize the respective operating system's capabilities. For Unix, there are currently two multi processing modules for productive use, namely the Worker MPM and the Prefork MPM.

The Worker MPM creates various processes to handle HTTP requests, and every process runs multithreaded. The idea is to combine the performance of multithreading with the stability of multitasking. When one Apache process crashes, other processes are still in place to serve requests, but all requests served by other threads of this process are affected.

The prefork MPM pre-starts processes that run single-threaded. Apache 1.3 works exactly this way, by serving every request by an isolated process. When one process crashes, no other processes are affected, which is a stability advantage over the worker MPM.

Windows has its own optimized MPM, the so-called `winnt` MPM. One control process starts a single worker processes that serves HTTP requests in multiple threads. This MPM takes into account that thread switches are rather fast on Windows, whereas task switches are rather slow. To save on the slow task switches, all requests are served by one process. Should the worker process crash, however, all requests served by the web server are affected.

Besides the performance aspects, the big difference between both MPMs on Unix is that worker always runs multithreaded, whereas prefork always runs single-threaded. If PHP is installed as an Apache module, you should only use thread-safe PHP extensions with the worker MPM. On Windows, Apache always runs multi-threaded, so you should be cautious with non-threadsafe PHP extensions when PHP is installed as an Apache module. We discussed thread-safety earlier in this chapter.

Web servers offer a network service, so they must be accessible from the net. To a certain extent, this makes a web server a potential security risk, because it can permanently be subject to attacks. To confine the potential damage in case a web server is compromised, a web server should run under a non-privileged user account. Often, `wwwrun` or `daemon` is used.

On Windows, Apache runs as a system service, which in turn runs under the System account by default. The System user is not allowed to access the network, but can read and modify every file of the system. Should an attacker take control over Apache on Windows, it is rather easy to make arbitrary modifications to the system.

On Unix, the user the web server is running under has only limited access rights and cannot modify critical system files. Still, read access to system files could be sufficient to steal a password file.

Security

Just like performance, web server security is a rather complex topic. Wether a solution is good or bad depends on various circumstances, and there is no universal solution, otherwise it would probably have been integrated into the software for long.

In addition, security and performance are usually competing goals. More security means worse performance. For example, by adding a firewall to a system, every IP packet going though the firewall will be analyzed, which requires computing time.

When migrating a system, it can be a good idea to neglect security issues first, because every security measure is a potential source of errors, which makes testing more complicated.

A basic rule to increase security of Internet servers is to run server processes with as few rights as possible, as was already mentioned in the previous section. To further reduce the potential damage when a process has been compromised, file system access can be limited to certain directories. Under Unix, the `chroot` function can be used to define the root directory for a process. In this directory, the system's subdirectory structure with copies of all required files must be present.

The whole accessible file system for the process is the `chroot` directory, also called *chroot jail*. There are different views about whether `chroot` is a security feature, because a process can break out of its chroot jail when it gains root privileges.

Another way of increasing web server security is using CGI wrappers. Using a CGI wrapper, processes can be executed under different user accounts to better seal them off from each other. Configuring a CGI wrapper is not easy. All access rights must be set correctly, or the wrapper will not work properly. Some CGI wrappers commonly used with PHP are CGIwrap or the Apache module suExec.

Compiling Apache

An open source web server like Apache can be either installed as binary distribution or be compiled from source. To start with, it is usually easier to use a binary distribution that can be installed by a single command. These binary packages are not optimized for your system, but created to run as many different systems as possible.

Binary distributions on Unix do often not suit special needs. If you require certain extensions or want to optimize the web server for your system, it can be a good idea to compile from source.

On Windows, it is virtually impossible to compile Apache from source. You should not even bother trying, but use a binary distribution instead.

Compiling the web server from source is less difficult than it may look like, and makes you independent from your operating system vendor's release cycles. If required, you can quickly install a newer web server version.

Apache and Apache2 are modular web servers. The web server itself offers only a basic functionality. Additional features are added by installing extensions, the so-called modules. This makes a web server easily customizable.

There are two ways to compile Apache modules, either as static module or as a dynamically loadable module. The dynamically loaded modules are also called dynamic shared objects (DSO). Static modules are faster, while dynamic modules can be activated and deactivated by configuration settings, without re-compiling Apache.

Multiple Web Servers on One System

When you use more than one web server on a system, they will compete for the TCP ports 80 and 443 that are used for HTTP and HTTPS traffic. You will have to configure the web servers to use different ports, for example 81 or 8080.

If you run a firewall on the server, you will have to open the correct ports in the firewall. On Windows, when using a personal firewall, you will probably have to allow the web server to listen to the respective ports.

Many PHP applications are not prepared to generate URLs that access ports other than 80 or 443. You might have to adapt the application to use non-default ports. Alternatively, you can configure one web server to pass through certain URLs or domains to another web server. Using Apache, you could use `mod_backhand` to accomplish this.

The advantage of this approach is that the second web server is transparent to the application. Sometimes, a very lean second web server is used just to serve static content. This cuts down the required amount of heavy-weight web server processes with PHP and can save lots of system resources.

To install multiple instances of Apache on one system, you have make sure that the server's configuration and administration files have unique names and do not interfere with each other. You might have to adjust the start scripts as accordingly.

If you are not a real Unix expert, it is often difficult to find out which files and directories are actually used when compiling and running your web server. If can be time-saving to use two distinct systems to run the two web servers and make configuration of each server easier. By using virtual systems (see Chapter 7), you do not even need additional hardware to do so.

When migrating a system, I would suggest starting with one web server on one system, to reduce the complexity of your test system. If you want to improve the performance later, you can add a second web server for load balancing or to serve only static content.

PHP

One of the first questions asked when migrating PHP is which PHP version the system to migrate uses. Of course this is one of the most important aspects, but as we will see in this chapter, the way PHP is integrated with the web server, how PHP is configured, and which PHP extensions are used play an equally important role.

The installed PHP version can be found out in different ways, either by calling `php -v` at the command line, at script runtime by calling the function `phpversion()` or by reading the built-in constant `PHP_VERSION` or, last but not least, by looking at the output of `phpinfo()`.

The function `version_compare()` compares two version numbers in PHP's major-minor-patch style. This allows you to check whether a sufficient PHP version is used right at the top of your program. It is better to quit right there with an error message instead of getting a fatal error later because of trying to use an undefined PHP function.

Web Server Integration

PHP can be integrated with web servers in different ways. The PHP core is web server independent, which allows you to use PHP with most, if not all, existing web servers. The interface that is used to integrate PHP and a web server is called server API (SAPI).

The most important server APIs in PHP are:

- apache to run PHP as Apache 1.x module
- apache2handler to run PHP as Apache2 module

- cgi to integrate PHP and the web server through the CGI interface
- cli to run PHP at the command line
- isapi to integrate PHP with the IIS web server

To find out how PHP is integrated with the web server, you can use the built-in constant `PHP_SAPI`, call the function `php_sapi_name()`, of just look at the *Server API* line in the `phpinfo()` output.

The various methods of integrating PHP with a web server differ in performance and in the user account and associated rights PHP scripts run as.

There are two different usage scenarios for PHP, namely a shared hosting environment, with different users having access to a server, and a dedicated server running only one web application or a set of application that must not be isolated from each other. Even when more than one user account is present on a dedicated server, all users are usually considered friendly to each other. In a shared hosting environment, the users have to be isolated from each other to prevent them from accessing each other's files.

PHP as Web Server Module

In dedicated servers, the most common way of using PHP is probably as an web server module. Running PHP as a module usually yields better performance than any other way of integrating PHP and the web server. If you use Apache, you can either compile PHP into the web server as a static module or compile PHP separately as a dynamically loadable module (DSO).

When PHP was compiled as a DSO, you need to explicitly load this module in the Apache configuration file: `LoadModule php5_module modules/libphp5.so`.

For older operating systems or different Apache versions the `LoadModule` entry may differ. The PHP manual at `http://www.php.net/manual/en/install.php` provides you with detailed instructions on how to integrate PHP and the web server in different scenarios.

On shared hosting servers, running PHP as a web server module is problematic. Since PHP runs as part of the web server process, PHP scripts are executed with the same access rights as the web server itself. This means that the PHP scripts of all different users are executed under the same system user account, which allows every user to access every other user's files through PHP.

To isolate the different system users from each other at PHP level, we need to be able to execute PHP under different user accounts. To accomplish this, PHP must be installed as CGI or FastCGI.

Another problem lies with the files that system users upload to the server via FTP or SFTP. Users work under a different account than the web server, so PHP cannot write to uploaded files and directories, because they are owned by another user, unless, of course, the files are made world-writeable, which again is not a good idea from a security perspective, especially with various users having access to the system. On a dedicated server, making files world-writeable is usually not a big problem, since all users on the system can be considered friendly. On a shared hosting server, however, world-writeable files are a recipe for disaster.

Running PHP as an Apache module is less secure "inside" the server, but more secure on the outside, since CGI has a few security issued by design.

Let aside the better performance, installing PHP as an Apache module has other advantages. PHP can access the HTTP headers and more request and environment information than in a CGI setup. When a user has completed HTTP authentication, even when PHP was not involved, it is possible to read the user name from the `$_SERVER` superglobal variable, for example. It is even possible to explicitly end an Apache child process by PHP. This functionality is not available under other web servers or in CGI mode.

What is more, PHP can only process HTTP authentication when running as an Apache module, which allows for authenticating users not only against a flat file, but also a database or an LDAP server through PHP.

Last but not least, configuration of an Apache module is more flexible than a CGI module. In addition to `php.ini`, you can make configuration settings in `httpd.conf` by using the directives `php_admin_value` and `php_admin_flag`. By using `php_value` and `php_flag`, you can modify certain settings in `.htaccess` files as well, if Apache allows `.htaccess` configuration. This requires the `AllowOverride` Apache configuration directive to contain *Options* or be set to *All* in `httpd.conf`.

This way, many PHP configuration settings can be fine-tuned per directory. It is not possible, though, to change security-relevant configuration settings or settings affecting PHP startup per directory, as Apache pre-starts web server processes.

PHP Over the CGI Interface

The tighter PHP is integrated with the web server, the better the performance. Starting PHP takes up some computing time that adds up to the regular script compilation and execution time if PHP runs in CGI mode, because a new PHP process is started for every request. The advantage of this scenario is that each process can run under a different user account, and thus have different access rights. Also, a crashing PHP process does not affect the web server or any other PHP process.

These advantages of the CGI interface come at the cost of less performance, which is especially painful when only short PHP scripts are executed, because the PHP startup time can actually exceed the script execution time. Even if the script execution time itself is low, the start time overhead remains. Also, CGI scripts require more memory, because every process contains the full PHP interpreter.

The big advantage of CGI is that you can run PHP programs under a different user account, which is important on shared hosting servers, as we have already seen. Unless the files of each user are deliberately made accessible for the group or even all users, each user's PHP programs and data can be isolated from each other.

To start a CGI script, a `shebang` is required. The `shebang` specifies the path to the interpreter required to execute a file. This adds a dependency to the directory layout on a certain system, and is thus undesired. Luckily, PHP can be configured to run scripts without a shebang. If you encounter scripts with a `shebang` with a path that is not valid on your system, you can use a symbolic link that points to the actual PHP interpreter. Make sure that PHP files with `shebang` have correct line endings. Files with Windows line endings do not work on Unix, because the surplus character at the end of the `shebang` line prevents Unix to locate the interpreter.

To configure PHP as CGI handler for Apache, you must add the following entry to `httpd.conf`:

```
Action application/x-httpd-php "/php/php-cgi.exe"
```

This requires `mod_action` to be loaded. Also, you must call the CGI binary, not the CLI binary, otherwise PHP will not work. The name and location of the PHP binary will differ between systems and versions. The example was taken from a Windows system.

CGI scripts are usually executed by a CGI wrapper that takes care of changing the user id for the process. If you do not use shared hosting server, you can do without a CGI wrapper, and PHP processes run with the user id of the web server starting the CGI processes.

When running in CGI mode, PHP cannot process HTTP authentication requests, since there is no way of passing the required information between the web server and PHP. This may be a main reason why most web application do not rely on HTTP authentication, but use a HTML-based login. Please note that HTTP authentication is generally still possible, but it is not possible for PHP to process it.

The CGI standard (`http://hoohoo.ncsa.uiuc.edu/cgi`) defines some environment variables that must always be present in CGI programs. Some of the environment variables that are available to programs when PHP runs as an Apache module are not available in CGI mode.

PHP and the FastCGI Interface

The FastCGI standard tries to combine the best of the two worlds. By pre-starting processes, the startup delay that slows down CGI scripts is not applicable any more. This principle is very similar to the way Apache works, but still allows for starting processes under different user accounts. The FastCGI performance is comparable to the performance of an Apache module, still FastCGI will probably be a little slower due to the looser coupling between Apache and PHP scripts.

Apache supports FastCGI through the FastCGI module. In addition, the suExec module is used to change the user id the FastCGI processes are running under. Configuring FastCGI is a rather complex issue, though. The following example shows an example for the required settings in `httpd.conf`:

```
FastCgiWrapper .../suexec2
FastCgiServer .../php-fcgi-scripts /php-fcgi-starter -user <userid>
                                                      -group <groupid>

AddHandler php-fastcgi .php
Action php-fastcgi /cgi-bin/php-fcgi-starter

<Location /cgi-bin/php-fcgi-starter>
        SetHandler fastcgi-script
        Options +ExecCGI
```

```
</Location>
```

Executing PHP Programs

Regardless of how PHP and the web server are integrated, you must define which files are to be parsed as PHP code. Usually, this will be all files with the extension `.php`. For an Apache web server, this is configured by adding the following line to `httpd.conf`:

```
AddType application/x-httpd-php .php
```

Older applications sometimes have PHP files containing functions, libraries or classes that have the extension `.inc`. By this extension, the author tries to indicate that theses files should not be called directly, but are included from other PHP files. To make this work, the web server must be configured to also parse `.inc` files as PHP code:

```
AddType application/x-httpd-php .php .inc
```

If you omit this, `.inc` files will be treated like regular text files and sent to the browser, which will show the source code. Imagine a visitor of your web site seeing the source code of a configuration file containing the database username and password!

Should you encounter PHP files not having the extension `.php`, you should rename all these files. Make sure to also rename all links to these files (there shouldn't be any), and all include and require statements in all PHP files.

If you insist on indicating the type of a file by the name, use composite extensions like `.inc.php`, `.lib.php`, or `.class.php` so that the name still end in `.php` will be executed as PHP by default on any PHP-enabled web server.

Compiling PHP

If you use PHP on Windows, you will probably use a binary installation program. It is very difficult to compile PHP on Windows, because you need certain of versions libraries and tools, and a certain compiler version. There is usually no reason, however, to compile PHP yourself on Windows, since up-to-date binaries are available and all PECL extensions can be downloaded as DLL files from `http://pecl4win.php.net`.

On Unix, the question is whether to use a binary package or compile by yourself. I would suggest using a binary package from your operating system vendor or Linux distribution for starters at least if you do not compile by yourself. If you use a binary package of a third party, you can never be sure that PHP was compiled with sensible default settings and is optimized for your target system. (From a security perspective, installing a binary package it is also a matter of trusting its source. You can never really know what somebody has build into a binary package. This also holds true for operating system vendors and Linux distributions, but if you do not trust a certain operating system, you should not use it anyway.)

Operating system vendors and Linux distributions often backport security fixes to older software versions. From a security aspect, this is a good idea, but due to lack of documentation you can never be really sure which changes they made to the software.

When compiling PHP, you need certain libraries in certain versions, and, unfortunately, not always the latest version of some common build tools. To compile PHP, the parser generator Bison is required, for example. The current version of Bison is 2.3, but to compile PHP, using version 1.28, 1.35, or 1.75 is recommended. For Flex, a tool for lexical analysis, the situation is similar: the current version is 2.5.34, but to compile PHP version 2.5.4 is recommended. If required tools are not present in these versions on your build system, it may be necessary to also compile them from source.

To compile PHP from source, you need the Apache developer package unless you have also compiled Apache from source. Without the tool `apxs`, you cannot compile PHP as an Apache module. It may be a good idea to also compile the web server from source, because this gives you better control over where its files are put on your system. With binary packages, you sometimes do not know where they put some of the files and why. Instead of searching your file system for the requires files, you might consider compiling the web server from source, which also forces you to better understand the environment.

If you compile PHP, you must choose the server API (SAPI) when configuring the source code. This decision determines how PHP and the web server have to be integrated later on. If you use a binary package, the package installer will usually automatically configure the web server for the chosen SAPI automatically.

Thread Model

PHP can run multithreaded or single-threaded, depending on the web server and how PHP is integrated with the web server (as discussed earlier in this chapter). If PHP is running as a dynamically loadable or static web server module, the thread model the web server is running under also applies to PHP. On Windows, applications including web server always run multithreaded, whereas Unix can work single-threaded or multi-threaded. When you use an Apache web server, the MPM you choose determines the thread model used for Apache and thus PHP.

When PHP works in CGI mode, each process runs single-threaded, because it serves only one request. This affects performance, but guarantees maximum stability by isolating the processes from each other. This allows you to execute non-threadsafe software as CGI programs. The same holds true for FastCGI, where it would thoretically even be possible to run multi-threaded FastCGI processes as well.

PHP itself is threadsafe, this means it can safely be executed in a multithreading environment. It is not exactly known, however, which PHP extensions are threadsafe, since it is not always clear which compiler options and external libraries were used to compile an extension. Some of them, like GD, are known not to be threadsafe. While you can run a non-threadsafe extension in a multithreaded environment, and may even face no problems, under higher system load you are very likely to see strange and hard to reproduce errors.

To safely use a non-threadsafe PHP extension, you must run PHP single-threaded, try to find an alternative extension that is threadsafe, or use an external program instead. The free software ImageMagick, for example, can be used as a replacement for the GD extension. There is a PECL extension imagick that exposes the ImageMagick features to PHP.

If you desperately need the functionality of a certain non-threadsafe extension, you can consider creating a web service powered by a single-threaded PHP extension. This way, you do not reduce the stability of your main PHP installation but can still, say, create thumbnail images with GD.

PHP Configuration

There are many ways to configure PHP, and many configuration options. Many configuration settings can be made in various places, and even at script runtime. There are quite a few possibilities:

- At script runtime using `ini_set()`
- On Apache, in `.htaccess` files. When the Apache configuration directive `AllowOverride` contains *Options* or is set to *All*, Apache looks for `.htaccess files` in the directory where the called PHP program is located, and its parent directories up to the directory specified by the `DocumentRoot` directive. The configuration settings in all `.htaccess` files are applied in the order they are found. This allows for a hierarchical, per-directory PHP configuration
- In the Apache configuration file `httpd.conf`. Here it is also possible to make settings that apply to certain directories
- In the configuration file `php-<sapi>.ini`, where `<sapi>` is the SAPI used to integrate PHP and the web server. If you want to know which SAPI PHP is currently using, call the function `php_sapi_name()` or use the built-in constant PHP_SAPI
- In the configuration file `php.ini`

Not all PHP settings can be changed everywhere. Security-relevant settings cannot be changed at runtime or in a `.htaccess` file. Naturally, settings that affect PHP startup can also not be changed at runtime. It is not possible, for example, to load a PHP extension or enable dynamic loading of extensions at runtime.

The `php.ini` and `php-<sapi>.ini` files are being searched for in various locations. PHP will load the first configuration file found according to the following rules :

- For Apache2 web servers the directive `PHPIniDir` can be used in `httpd.conf`, specifying the directory to load the ini file from. This path must always contain forward slashes, even on Windows, otherwise the configuration file will not be found

- The configuration file will be loaded from the directory specified in the environment variable `$PHPRC`

- On Windows, there are several registry keys that can specify the ini file location. The registry keys allow a fine-grained level of control over which configuration file to load for each PHP version. The registry key is `HKEY_LOCAL_MACHINE\Software\PHP<version>\IniFilePath`, where `<version>` can be specified in the format major, major-minor, or major-minor-patch. To force loading a certain configuration file for all PHP versions, you can also omit `<version>`

- The configuration file in the current working directory. Please note that Apache changes the current working directory to its root directory on startup. For security reasons, in CLI mode, no configuration file is read from the current working directory

- The configuration file in the web server directory

- On Windows, the configuration file in the PHP directory is loaded. The PHP directory is the directory where the PHP binary is located

- On Windows, the configuration file is loaded form the Windows directory, which is stored in the environment variable `%Windir%`

- On Unix, the configuration file ultimately be loaded from the directory that was specified in the `-with-config-file` switch when configuring the source code. Usually, this directory is `/etc` or `/etc/php`

Please note that when you have changed the PHP configuration, you will have to restart the web server, unless you work in CLI or CGI mode, where the configuration file is re-read on every request. This is one of the factors that slow down PHP in CGI mode, by the way.

In PHP 5.1, a configuration file in the current working directory, if present, was used. This was an undocumented feature and removed in PHP 5.2, because of potential undesired effects when by coincidence a PHP configuration file was present in the working directory. Should your system rely on this behavior, you must move the configuration file to another directory.

When a PHP server runs multiple applications requiring different PHP configuration, you can configure PHP individually per directory. It is not always easy to find, which configuration is in effect in a certain directory. When migrating, make sure that you carefully analyze the system as to which individual configurations apply to certain applications or scripts. Chapter 7 shows how you can find out the current value of PHP configuration settings at runtime.

Keep in mind that PHP programs executed at the command line can run in a different configuration than PHP programs in the web server. When working with unit tests, you should keep in mind that your CLI configuration does not differ from your SAPI configuration too much, otherwise the results of unit tests may not reflect the results the same programs yield in a web server environment.

Most, though not all, PHP configuration settings are relevant for a migration. It is not always a good idea to reuse the status quo configuration on the target system. Instead, you should work towards running PHP in default configuration and only change the settings that cause compatibility problems.

Tags and Separators

asp_tags

This setting is a relic of old times, when many users would use Microsoft Frontpage. Since Frontpage had problems with displaying `<?` and `<?php` tags in HTML source code, the alternative tags `<% %>` were introduced, inspired by the ASP tags. You can activate these tags with the `asp_tags` configuration setting, which is disabled by default. This setting also defines `<%=` as a shortcut tag for `<% echo` (see `short_open_tag`).

If you encounter programs using the ASP tags `<%` and `%>`, you should replace them by `<?php` and `?>`.

arg_separator.input

Specifies the character used to separate URL parameters. The default value is `&`, and you can also specify multiple separator characters.

If other separator characters are configured here, you should consider keeping them to make sure that your application can process all generated URLs. On the long term, it might be a better idea to modify `arg_separator.output` to generate URLs with the ampersand as separator, but this may require changes in your application code.

arg_separator.output

Specifies the character used to separate URL parameters when URLs are generated by PHP. The default value is `&`, though, strictly speaking, `&amp;` should be used. Most programs silently rely on using `&` as separator, so you should keep this value.

If your system uses a different value here, you should keep it to make sure that the URLs your application still generates the same URLs on the target system.

short_open_tag

By default, PHP scripts start with the `<?php` tag. There is an alternative short form `<?` which had originally been used to denote PHP sections in HTML code. Since `<?` is also being used to start XML files, using the short tag can cause problems in certain situations, because XML sections embedded into HTML would be parsed as PHP. To make embedding PHP into templates easier, a shortcut tag `<?=` is defined, which is equivalent to `<? echo`.

`short_open_tag` is set to *On* by default. If your scripts use the short open tag, PHP will not be executed on your target system when `short_open_tag` is turned *Off*. You should not use `<?` or `<?=`. If you encounter them in any script, replace them by `<?php` and `<?php echo`, respectively.

Processing Input

always_populate_raw_post_data

If enabled, the array `$HTTP_RAW_POST_DATA` will be filled with the original POST data sent to the PHP script, but only under certain circumstances (see Chapter 8 for more information). The default value is *Off*. If `always_populate_raw_post_data` is activated on your system, check whether your application uses the variable `$HTTP_RAW_POST_DATA`.

auto_detect_line_endings

When enabled, PHP will recognize line endings in Macintosh format (as we discussed earlier in this chapter) when reading from files with `file()` or `fgets()`. The default value is *Off*, and instead of activating this setting, it would probably be a better idea to convert all files with Macintosh line endings.

Since the new Mac OS X operating system is a Unix derivative, the old Macintosh line ending should not be an issue any more today.

register_argc_argv

Determines whether the GET parameters are also passed to the PHP program in the `$argv` variable (which also populates the argument count variable, `$argc`). The default value is *On*.

If your application works with `$argc` and `$argv`, you must keep this setting *On*.

register_globals

This much disputed setting determines whether global variables are created for any input PHP receives from the outside. The default value is *Off*. Please refer to Chapter 8 for more information about `register_globals`.

register_long_arrays

When enabled, the so-called long arrays, predecessors of the superglobal variables, are created and populates with input values. Defaults to *On*, but is deprecated and will be removed in future PHP versions. For more information, please refer to Chapter 8.

variables_order

Defines which superglobal variables PHP creates. The default value is *EGPCS*, which means that the superglobal arrays `$_ENV`, `$_GET`, `$_POST`, `$_COOKIE`, and `$_SERVER` will be created. Most programs rely on all superglobal variables to be present, so you should not change this setting.

This setting also determines the order in which the superglobal `$_REQUEST` is populated. Changing `variables_order` can alter the behavior of your application, if `$_REQUEST` is used, because a `GET` parameter could overwrite a `POST` parameter. It is good practice not to rely on `$_REQUEST` at all; see Chapter 8 for more information.

Error Handling

display_errors

When enabled, PHP error messages are displayed as part of the generated HTML output. While this feature is very useful on development and test systems, it should be disabled on production servers. Please refer to the section entitled “Storing the Previous Error Message” in Chapter 8 for more information.

error_reporting

Defines which error messages, warnings, and notices PHP displays. This setting is very important when migrating, so there is a dedicated section in this book (entitled “Configuring Error Display” in Chapter 8) dealing with it.

track_errors

Determines whether the last error message is stored in the variable `$php_errmsg`. Please refer to the section entitled "Storing the Previous Error Message" in Chapter 8 for more information.

Auto-creating HTTP Headers

default_charset

A browser must know the character set HTML pages are encoded with to correctly display the page. To communicate the character set to the browser, the server sends a `Content-Type` header. The configuration setting `default_charset` defines which `Content-Type` header is sent to the browser by default, if the page does not create such a header itself.

The value defaults to ISO8859-1, sometimes UTF-8 is used. Keep in mind that this setting only determines which header will be sent to the browser, no conversions are performed. It is the programmer's responsibility to actually deliver the page in the correct character set.

Make sure that this setting reflects the character set your PHP scripts are sending. We will discuss this later in this chapter.

default_mimetype

Defines the MIME type of PHP-generated output. By default, this is `text/html` as PHP usually outputs HTML pages. If you create XHTML pages (see chapter 7), you should use `text/xml` instead.

You can change the MIME type at any time having your scripts send a `Content-Type` header at runtime. `default_mimetype` determines the MIME type sent when your program does not create a `Content-Type` header itself.

You should keep the status quo of this setting on your target system, otherwise browsers may have problems displaying pages that do not send a `Content-Type` header.

Floating Point Precision

precision

Defines the amount of digits being output for floating point values. This has nothing to do with computational accuracy, but only refers to displaying the numbers.

The default value is *14*. If you migrate to a system with different setting, it may seem as if the application makes calculation errors, though this is not necessarily the case. If you encounter a different value, you might be tempted to keep it so that the results still look the same, but it is probably a better idea to switch to the default value so that the results look the same on any default PHP installation.

serialize_precision

This value determines the precision used to serialize floating point values. The default value is 100, and you should always keep this value. If you encounter a different value, calculation errors can happen when you transfer serialized data from between different systems. The same problem can occur when you mix serialized values of different precision.

Sending Email

sendmail_from

Defines the sender of an email on Windows systems. If the given address does not exist and/or its MX lookup fails, the email might be rejected or marked as spam by mail servers. Make sure you keep the existing sender address, and that a mailbox exists for this address.

mail.force_extra_parameters

Overrides the fifth parameter passed to `sendmail` calls done by the `mail()` function. With this setting, you can force email to have a certain sender address, without the PHP program being able to change it. This can be useful to make sure that you actually send email that is not rejected by other mail servers.

sendmail_path

Even when no sendmail installed on a certain system, a sendmail-compatible binary to send email is usually available. You may have to adjust the default value `/usr/sbin/sendmail -t -i` to your target system to be able to send email.

On test systems, it can be useful to not actually send emails, but just to store them into a directory in the file system. You could do this by setting `sendmail_path` to a program that just stores the emails instead of sending them (or sends and stores them). This allows you to send as many test emails to any address you like, without bothering customers or developers with them.

If you use a PHP class like PEAR_Mail to send email by SMTP, changing sendmail_path has no effect. In this case, you can modify your PHP program so that messages are stored in the file system and sent by email, or just stored in the file system. A constant that puts the application into test mode could determine what to do:

```
<?php

  define('DEBUG', true);

  function send_mail($aRecipient, $aSubject, $aBody)
  {
    $mail = Mail::factory($smtp, ...)

    $filename = '/tmp/' . uniqid();
    file_put_contents($filename, ...);

    if (!defined('DEBUG') $mail->send($aRecipient, ...);
  }

?>
```

The advantage of this solution is that emails will also be stored when the system is productive. You must make sure, however, that the stored emails are not accessible over the web, and should delete the stored messages from time to time to prevent the directory to grow too big.

Limits and Resource Limitations

allow_url_fopen
When enabled, access to remote files is possible with functions like fopen() or file_get_contents(). The default value is *On*. Many programs rely on this setting, so you should not change it, but set allow_url_include to *Off* to make sure that no remote programs can be executed by include or require.

default_socket_timeout
The socket timeout in seconds, defaulting to *60*. Should your application work with reliable sockets, you can choose a lower value. As opposed to the timeout value

passed to `fsockopen()`, this value is in effect while the connection is open, and not only for the initial socket opening.

If the value is too low, you risk wrongly terminating socket connections to slow systems or systems under high load.

disable_classes

A list of built-in PHP classes that cannot be instantiated. As soon as a program tries to instantiate a class contained in this list, a fatal error occurs. I would not recommend using this setting, but reduce the number of installed PHP extensions instead.

disable_functions

A list of built-in PHP functions that cannot be used. Some system administrators do not allow certain PHP functions like external system calls. This does increase the system security, but comes at the cost of less reliability: as soon as a program tries to call a disabled function, a fatal error occurs. Since you never know in advance which PHP functions an application is going to use in good intent, you should not restrict PHP functionality using `disable_functions`.

file_uploads

Determines whether file uploads are allowed. Most web applications rely on file uploads, so you should keep this setting enabled. If absolutely no application on your system requires file upload, you could consider disabling this setting to increase security.

ignore_user_abort

Defines the behavior when a client (usually a browser) terminates a HTTP connection before the PHP script has completed. The default value is *Off*, which means that PHP programs are terminated when the HTTP connection is terminated. When *On*, PHP programs will be allowed to complete, which can be useful to make sure that all data has been stored consistently. This setting can also be modified at script runtime using the *ignore_user_abort()* function.

You should keep the status quo of this setting on your target system to make sure application behavior does not change.

max_input_time

Determines how long a PHP script may take to process input data. The default value is *-1*, which means no time limit. If you set a limit, you always risk terminating PHP programs just because processing the input took too long. The problem is that you can never tell whether a program just needs a little more time, or will never terminate.

max_execution_time

This setting defines how much execution time a PHP script may use. If this time limit is exceeded, the program is terminated. Please note that time spend with system calls and i/o operations does not count, just the CPU time of the PHP script itself. You can disable the time limit by calling `set_time_limit(0)` at runtime. It is important to note that the web server usually also puts a time limit on processes.

The problem with this setting, again, is that terminating a program just because it takes a lot of computing time can cause more damage than do good. On development systems, the setting may be useful to prevent a single process from locking up the whole system. On a live system, I would not recommend using this setting, though, because there is a high probability of causing data inconsistencies.

memory_limit

{{memory_limit}}Determines how much memory a PHP script may use. Accurately measuring a script's memory consumption is technically not easy, but you can at least prevent a PHP script from using the whole system memory. As soon as a script tries to allocate more memory than specified in `memory_limit`, it is terminated. Since newer PHP versions measure the memory consumption more accurately, the default memory limit has been increased from 8 MB to 128 MB. Please note that this does not mean that newer PHP versions require more memory.

When setting memory limit to *-1*, no limit is enforced. Like with all resource limitations, you can either set the limit too high, so that the system is already destabilized before the limit is reached, or too low, so that programs are terminated though nothing went wrong.

I would recommend using a memory limit on a development system, which also gives you a feeling for the memory consumption of your application, but to disable memory limit on production systems.

post_max_size

Determines the maximum size of a `POST` request. Since file uploads are transmitted via `POST` requests, `post_max_size` should be set to at least the value of `upload_max_filesize`.

upload_max_filesize

Limits the size of uploaded files. The default value is 2M, which is not enough for many of today's applications. Since files are uploaded via `POST` requests, you must also set `post_max_size` to an appropriate value, otherwise an upload may already be terminated because the `POST` request is too large.

Another pitfall is the `memory_limit` setting. As soon as a PHP script uses more memory than specified in `memory_limit`, the script is terminated. It is therefore not possible to upload files larger than `memory_limit`.

open_basedir

The `open_basedir` directive limits file access from PHP to a certain base directory. Strictly speaking, `open_basedir` is a prefix: only when beginning of the absolute path of the file to open matches `open_basedir`, the operation is allowed. Otherwise, PHP outputs an error message.

It is a good idea to set `open_basedir` individually for different users to limit access to other people's files in a CGI setup. Be warned, however, that `open_basedir` offers no real security, because through PHP extensions or systems calls, it is easy to circumvent its restrictions.

Session Management

session.auto_start

Determines whether a PHP session is started automatically. The default value is *Off*, and you should keep this default to avoid the overhead of initializing a session when it is not needed.

If your application relies on automatically started sessions, you can turn this setting to *On* or modify the program to explicitly start a session with `session_start()` where appropriate.

session.bug_compat_42

PHP versions before 4.2.3 would allow you to register a session variable in the global namespace, even when `register_globals` was deactivated (see Chapter 8 for more information). This setting ensures compatibility to PHP 4.2.3 by reproducing this behavior on newer PHP versions.

You should disable this setting and enable `session.bug_compat_warn`, which causes PHP to issue a warning whenever the program relies on the old behavior. You should then modify the code to use the superglobal array `$_SESSION` instead of registering a global variable as session variable:

```
<?php

  $a = ’test’;
  $_SESSION[’a’] = $a;

?>
```

session.bug_compat_warn
When enables, PHP issues a warning when programs try to initialize a session variable in the global namespace:

```
<?php

  $a = ’test’;
  session_register("a");

?>
```

When `session.bug_compat_warn` is enabled, this program shows the following warning:

```
Warning: Unknown: Your script possibly relies on a session side-effect which
     existed until PHP 4.2.3. Please be advised that the session extension does
     not consider global variables as a source of data, unless reg-ister_globals
     is enabled. You can disable this functionality and this warning by setting
     session.bug_compat_42 or session.bug_compat_warn to off, respectively. in
     Unknown on line 0
```

The default value is *On*. You should always keep this setting enabled.
session.cookie_domain
Determines the domain used for the cookie with the session identifier. The default value is the hostname of the server sending the cookie. If the name specified here does not match the hostname or DNS name of the server, respectively, there will be problems when setting and reading the cookie, which will prevent you from using sessions at all.

Files and Directories

auto_append_file

Defines a PHP script that is automatically executed after the called PHP script has been executed. You must specify the full, absolute path to the script (see Chapter 7 for more information).

auto_prepend_file

Defines a PHP script that is automatically executed before the called PHP script is executed. You must specify the full, absolute path to the script (see Chapter 7).

include_path

Like the system's search path, the directories in this list are searched when PHP files are included by include or require. When accessing files with `fopen()`, `file()`, `readfile()`, or `file_get_contents()`, the path is also searched, but accessing subdirectories of directories in the include path only seems to work with include and require. The first entry should be the current working directory, which is denoted by a dot. Like in all PHP scripts, you can use the forward slash as directory separator on Windows and on Unix.

The character used to separate the paths (path separator) is system-dependent (as discussed earlier in this chapter). If in doubt, have a PHP program output the built-in constant to find out what path separator your system uses.

When migrating, you must search all directories listed in `include_path` for code that your application uses. Keep in mind that the behavior of an application can change when you reorder the entries in the include path, as different files may be loaded.

upload_tmp_dir

Defines the directory to store uploaded files. This directory must exist and be writeable by the system user PHP scripts are running as. The default value is the system-wide temporary directory.

When migrating, make sure that the directory defined here does exist on the target system. Quite often, a housekeeping script is used to delete old files in temporary directories. You should also migrate these scripts to the target system, even if they may not seem to be part of the PHP application.

New Settings since PHP 5

allow_url_include

Since PHP supports stream wrappers, include and require can not only load files from the local file system, but also from remote systems, for example via a protocol like HTTP. In older PHP versions these URL wrappers could be allowed or disallowed for all kinds of file access by enabling or disabling `allow_url_fopen`. Allowing an application to use `file_get_contents()` on a remote URL to retrieve a web page or call a REST service always came at the risk of executing remote code with include or require.

The problem with executing remote code that you never know what code you are executing. Unless you are not in control of the remote system yourself, you should, for security reasons, never execute remote code. It is better to copy the code to the local system or set up a web service on the remote system.

By default, `allow_url_include` is disabled sine PHP 5.2. In my opinion, there is no reason to ever enable this setting.

date.timezone

Defines the server's time zone. Since PHP 5.1 a valid time zone must be set either in `php.ini` or at script runtime, otherwise PHP will output an `E_STRICT` error when a date function is called. For more information please refer to Chapter 8.

filter.default and filter.default_flags

This setting exists since PHP 5.2. It determines how `GET`, `POST`, and cookie input are filtered. The contents of `$_REQUEST` are filtered as well. The default value is `unsafe_raw`, which means that no filter is applied.

max_input_nesting_level

This configuration setting, available since PHP 5.2.3, closes a security hole in PHP that allowed an attacker to crash PHP by supplying deeply nested input data. The default value is 64, which should be sufficient for most applications.

Should you have to process input data that is nested more deeply, you must increase this value.

pcre.backtrack_limit and pcre.recursion_limit

This setting was introduced in PHP 5.2.0 to prevent PHP from crashing on complex or incorrect regular expressions that require too much stack memory. The default value is 100,000, but there are bug reports of PHP users that complained about correct regular expressions hitting this limit. You might want to increase these limits to 1,000,000 or 10,000,000, which you can do at runtime as well:

```
<?php

  ini_set('pcre.backtrack_limit', 1000000);
  ini_set('pcre.recursion_limit', 1000000);

?>
```

If the limit is too high, you risk a PHP crash because too much stack memory has been used.

A regular expression hitting the limit is stopped. The problem is that no error message is generated, and the regular expression just returns NULL as a result. You have to use `preg_last_error()` explicitly to check whether an error occurred. The function `preg_last_error()` returns an integer value matching one of the following constants:

- `PREG_NO_ERROR` (value 0)
- `PREG_BACKTRACK_LIMIT_ERROR` (value 2)
- `PREG_RECURSION_LIMIT_ERROR` (value 3)

For a full list of all constants and error messages defined by the preg extension please refer to `http://de.php.net/pcre`. If you work with complex regular expressions or your regular expressions process large strings, you should secure them by checking for an error:

```
<?php

  $text = preg_replace('/b.*b/', '', $text);

  if (preg_last_error() == PREG_BACKTRACK_LIMIT_ERROR ||
      preg_last_error() == PREG_RECURSION_LIMIT_ERROR)
  {
    throw new Exception('Regular Expression limit reached');
  }

?>
```

If the exception thrown in the example is not caught, a fatal error occurs, stopping program execution. In most cases, it is better to stop the program than to continue using a bogus NULL value as regular expression result, which will probably sooner or later cause a consecutive fault.

user_ini.filename

The name of a user-specific configuration file, like `.htaccess`, but also processed in CGI and FastCGI mode, not only when PHP is installed as an Apache module. Allows configuring PHP per-directory and is available from PHP 5.3. The default value is `.user.ini`.

session.cookie_httponly

This setting exists since PHP 5.2 and makes the cookie that stores the session id unreadable for Javascript running in the client browser. The so-called `HttpOnly` cookies provide a better level of protection against session hijacking by stealing cookies, but are not supported by all browsers. The default value is *Off*.

If you use Javascript that reads the session cookie, for example to retrieve the session id when creating an AJAX request, your application may not work correctly any more when you enable this setting.

Settings That Will be Removed in PHP 6

allow_call_time_pass_reference

Defines whether passing references at runtime is allowed. For compatibility reasons, this setting defaults to *On*. If you disable `allow_call_time_pass_reference`, you must check your PHP programs for `E_STRICT` errors indicating that references are passed by runtime. For more information please refer to Chapter 8.

enable_dl

When enabled, PHP extensions can be loaded at runtime using the `dl()` function. Though the default value is *On*, you should never rely on extensions this feature, since dynamically loading extensions is only possible in certain PHP setups. Instead of relying on `dl()`, you should configure your server to load all required PHP extensions by default.

magic_quotes_gpc

When enabled, single quotes, double quotes, backslashes and the *NULL* character in `GET`, `POST` and cookie input will automatically be escaped by a backslash. The default value is *On*. Please refer to Chapter 8 for more information about magic quotes.

magic_quotes_runtime

When enabled, quotes in data from external sources like databases and text files will be automatically escaped by a backslash. The default value is *Off*. Please refer to Chapter 8 for more information.

magic_quotes_sybase

This setting modifies the behavior of *magic_quotes_gpc* and *magic_quotes_runtime*. When enabled, single quotes with be escaped by another single quote instead of a backslash. This way of escaping complies with the SQL standard. The default value is *Off*. Please refer to Chapter 8 for more information about `magic_quotes_sybase`.

safe_mode

Determines whether PHP runs in the much debated *Safe* mode. The default value is *Off*. When PHP runs in Safe Mode, file access is only allowed when the user specified in `php.ini` is the owner of the file. The idea is to provide a similar security like CGI or FastCGI do by executing PHP in different user accounts.

In addition, PHP functionality can be disabled in Safe Mode. It is possible to only allow starting of external programs in a given directory, or limit access to certain environment variables.

Like `open_basedir` restrictions, the Safe Mode restrictions can be circumvented by PHP extensions or system programs called from PHP. Since, due to these limitations, Safe Mode does provide real security, it will be removed in PHP 6. It is better to isolate different users from each other at the operating system level instead of giving administrators a false sense of security through Safe Mode.

If you are migrating a system running in Safe Mode, you should disable Safe Mode and make sure that the operating system provides a similar or better level of security. These restrictions cannot be bypassed by PHP extensions or system programs.

Since disabling Safe Mode removes restrictions, every application working in Safe Mode should work flawlessly when Safe Mode is disabled.

zend.ze1_compatibility_mode

This setting forces PHP 5 to work with references like PHP 4 did, in other words, pass everything by value. It is disabled by default, and for good reason. Please refer to the section entitled "The PHP 4 Compatibility Mode" in Chapter 8 for more information.

Databases and other extensions

The various database extensions have different configuration settings that can contain default values like database host name, IP address, database username, password, socket name, and port number. For reasons of brevity, we will not list these configuration settings in detail, since they differ between databases.

The idea being putting connection information here is probably that `php.ini` is located outside the `www` directory, so there is less danger of passwords leaking out due to a misconfigured PHP server.

PHP Extensions

PHP is easy to extend, which is one of PHP's great advantages. PHP extensions written in C can add new PHP functions, classes, or constants.

PECL is a repository containing many open source PHP extensions. Unfortunately, not all extensions are compatible with every PHP version. This is because the internal PHP API sometimes changes between versions. Consult the extension's documentation to find out which PHP versions it is compatible with.

Not all PHP extensions can be combined with each other. The deeper an extension is integrated into PHP, the more likely conflicts with other PHP extensions are. It does probably not work to load two different PHP debuggers, for example. The reason is obvious: both extensions will try to use the same hooks and make the similar modifications inside the Zend Engine. Common PHP extensions, however, can usually be combined without problems.

As a developer, you often have to decide whether to use an extension or a PHP-level implementation. Generally, extensions are faster because they are implemented in C and compiled to machine code. Using an extension, however, adds a dependency to it. Your application will not work without this PHP extension. For a default or commonly used extension, this is not a problem, but you should make a conscious decision before relying on an exotic extension. When your application is supposed to work on shared hosts, you should keep the number of required extensions as small as possible.

PHP extensions must be enabled in php.ini and, in most cases, have their own, additional php.ini settings. To find out which PHP extensions are used in your PHP installation, you can use a command line command:

```
> php -m

[PHP Modules]
bcmath
bz2
[...]
xsl
zip
zlib

[Zend Modules]
```

PHP will output a list of all installed extensions ("modules"). The same list is available in the phpinfo() output, less compact, but with additional information about extension versions and their configuration.

You can also use the function get_loaded_extensions() to output a list of installed extensions. You can also use extension_loaded() to check at runtime whether all extensions required by your application are present:

```
<?php

  if (!extension_loaded('mysqli'))
  {
    throw new Exception('Missing mysqli extension');
  }

?>
```

It is probably better to end the program in a controlled manner when required extensions are missing, instead of risking a fatal error at runtime that might leave the application or database in an inconsistent state.

To find out which PHP extensions an application requires, you can use the PEAR package PEAR_CompatInfo. For more information, please refer to Chapter 7.

Loading PHP extensions at runtime is problematic, especially when PHP is running multithreaded. If, within one thread, a PHP extension would be loaded, it would suddenly become available to all PHP programs executed in this thread, since they share the same memory. How would you explain to your PHP script that half way through execution, additional PHP function have suddenly become available?

You should not rely on loading extensions dynamically, but load all required extensions in `php.ini`. In shared hosting environments, it is usually not possible to modify `php.ini` and load additional extension that your application might require. If you need certain extensions that are not available by default, you can try to convince your shared hosting provider to install these extensions. If this does not work out, you may have to consider switching your hosting provider, or switch to a dedicated server, which gives you full control over the server configuration, but leaves you responsible for the server security.

Recently, the PHP developers have started to move PHP extensions to PECL, which makes it easier to maintain and update extensions independently from PHP. Some extensions that used to be bundled with PHP are not distributed through PECL. Today, you may have to manually install certain PHP extensions from PECL that used to be distributed with PHP. Table 3.2 lists these extensions.

Extension	in PECL since/from	Remarks/Restrictions
Cyrus IMAP	PHP 5.0.0	not on Windows
Crack	PHP 5.0.0	
CyberCash	PHP 4.3.0	was bought by PayPal
DBX	PHP 5.1.0	
DirectIO	PHP 5.1.0	only on Windows
CyberMUT	PHP 4.3.0	not on Windows
filePro	PHP 5.2.0	
Hyperwave API	PHP 5.2.0	
Lotus Notes	PHP 5.0.0	
Mailparse	PHP 4.2.0	requires mbstring
MCAL	PHP 5.0.0	not on Windows
MCVE	PHP 5.1.0	not on Windows
MSession	PHP 5.1.3	not on Windows
Ncurses	PHP 6.0.0	only on Unix
Newt	2004	not on Windows, only CGI and CLI-SAPI
PDFlib	2004	Since PHP 4.3 PDFlib >= 4.0, before that PDFlib >= 3.0
POSIX RegExp	PHP 6.0.0	Use Perl-RegExp
Printer	PHP 4.1.1	only on Windows
qtdom	PHP 5.0.0	not on Windows
Socket	PHP 5.3.0	
TCP Wrappers	2003	not for Unix
vpopmail	PHP 4.3.0	
win32ps	2005	only on Windows
win32service	2005	only on Windows
xattr	2004	only on Unix
xdiff	2004	only on Unix
XSLT	PHP 5.0.0	
YAZ	PHP 5.0.0	
YP/NIS	PHP 5.1.0	not on Windows

Other PECL extensions, though developed and maintained in the PECL project, are bundled with PHP to make sure that they are always available in default PHP installations. These extensions are listed in table 3.3.

Table 3.4 shows an overview of PHP extensions that are no longer maintained with possible alternatives.

Extension	in PHP since/from	Remarks/Restrictions
PDO	PHP 5.1.0	
XMLReader	PHP 5.1.0	
XMLWriter	PHP 5.1.0	
ZIP	PHP 5.2.0	uses a library other than the PHP 4 ZIP extension
zlib	PHP 4.3.0	

Some PECL extensions are no longer maintained today. If you need old and unmaintained PECL extensions, you will find them in CVS which is accessible through a web interface at `http://cvs.php.net/pecl`. Unless you are a PHP and C expert, though, you will probably have a hard time making these extensions work.

Extension	Unmaintained since/from	Possible Alternatives
Activescript	PHP 5.0.1	
Aspell	PHP 4.3.0	Pspell
CCVS	PHP 4.3.0	MCVE
Classkit	end of 2004	Runkit
ClibPDF	PHP 5.1.0	Haru, PDFLib
ctype	PHP 6.0.0	Unicode
DBM	PHP 5.0.0	dba
DOM XML	PHP 5.0.0	DOM in PHP 5
fam	PHP 5.1.0	
ICAP	early 2002	MCAL
Informix	PHP 5.2.1	PDO_Informix
Ingres II	mid-2005	ingres
IRC Gateway	PHP 5.1.0	
mhash	PHP 5.3.0	Hash
Mimetype	early 2004	fileinfo
mnogoSearch	PHP 5.1.0	
muscat	around 2000	Xapian (`http://www.xapian.org`)
Object Overloading	PHP 5.0.0	Native Overloading in PHP 5
Oracle	PHP 5.1.0	OCI or PDO_oci
Ovrimos SQL	PHP 4.4.5 bzw. 5.1.0	
Payflow Pro	PHP 5.1.0	
Satellite CORBA	around 2003	Universe (`http://is.gd/u3B`)
SWF	PHP 5.0.0	MING
Tidy 1.0	PHP 5.0.0	Tidy 2.0 for PHP 5
w32api	PHP 5.1.0	ffi
XSLT	PHP 5.0.0	XSL for PHP 5
ZIP for PHP 4	mid-2003	ZIP in PECL

Installing PECL extensions is usually very easy. On Windows, you can download binary DLL files from `http://pecl4win.php.net`. Make sure you download the DLL for the correct PHP version.

Having downloaded the DLL, you must activate the extension in `php.ini`. Make sure to specify the full, absolute path to the directory containing the DLLs:

```
extension_dir="c:\php\ext\"
extension=php_ssh2.dll
```

To avoid problems with path names that contain blanks, enclose the path in double quotes (as discussed earlier in this chapter).

Keep in mind, that the order in which extensions are loaded does matter. You must load the PDO extension, for example, before you can load any database-specific PDO extension. Some extensions, like `mcrypt` and `mhash`, require additional DLLs. Unfortunately, these dependencies are not always clearly documented, so sometimes an extension just does not load or does not load properly.

On Unix, you can try installing a PECL extension using the following command:

```
pecl install <package name>
```

This does not work on every system, though. If it does not work, you have to compile the PHP extension from source. To make this work, you might have to install some required libraries first.

When you have installed PHP as a binary package, you also need the developer package containing the headers files and additional tools for compiling, like `phpize`. Sometimes it is easier to compile PHP from source instead of searching your system for the various required files.

Some extensions must be activated when the source code is configured. You might have to supply a library path for some switches.

Like you can compile PHP as a static or dynamically loadable Apache module, you can compile PHP extensions either statically into PHP or compile them as dynamically loadable extensions that must be activated in `php.ini`. The advantage of dynamically loadable extensions is that you do not have to re-compile PHP to update an extension.

It is recommendable to statically compile the required extensions into PHP, and leave it up to the system administrator to decide which additional extensions to activate. Every extension takes up memory, so you should only load required extensions to save on system resources.

When PHP is not running as a web server module, but in CGI or FastCGI mode, you can configure PHP individually for each user. In conjunction with a service oriented architecture, you could also create individual minimal PHP configurations for the different services.

Keep in mind, that the PHP extensions that are activated by default can still be disabled by `-disable` or `-without`, respectively. You should never disable those extensions, however, since most PHP applications rely on them to be present.

Some PHP extensions that use Unix-specific features do not run on Windows. Among these are:

- Posix
- Process Control
- Readline
- Recode
- Semaphore
- Session pgSQL

The Process Control extension only works when PHP does not run as a web server module, but only in CGI and CLI SAPI.

The `shmop` extension runs on Unix and Windows, but only Windows 2000 and later. Under both operating systems, PHP must be installed as a web server module to make `shmop` work. The CGI and CLI-SAPI are not supported.

Installing Multiple PHP Versions

When migrating, you might be tempted to install multiple PHP versions on one server, to be able to test your code in different versions.

When you have installed PHP as an Apache module, you cannot run multiple PHP version in parallel, because the symbol tables would conflict. If you are not willing to make extensive changes to the source code of at least one PHP version, there is no solution to this problem.

Running multiple PHP version in parallel is only possible when you run PHP in CGI or FastCGI mode. Still, you could combine both and install one PHP version as an Apache module, and additional PHP versions as CGI. Keep in mind, however, that there are some differences in functionality (as discussed earlier in this chapter).

Many shared hosting providers use this approach to offer their customers to use either PHP 4 or PHP 5. This allows every customer to select the preferred PHP version, and the appropriate to switch between the two versions.

I would not recommend installing multiple PHP versions on one system. It is more work to install two independent versions, and there are many sources of errors on the way. I would recommend setting up two dedicated test systems with one PHP version each.

PHP Code

The PHP code itself is one of the most important aspects of a migration. Chapter 8 deals with the problems you might face when migrating PHP code and, of course, solutions for these problems.

When migrating, you will probably differentiate between your own code and third party code.

Third party PHP code

In the early PHP years, a plethora of code snippets and examples was available on the Internet. Much of this code, though, was not of good enough quality to be used in professional projects. This situation has changed a lot in the last years. The PEAR project offers a lot of reusable open source PHP components, and well-known PHP companies like Zend or ezPublish offer free and open source frameworks and components that offer reusable solutions to common coding problems. Even when you do not want to use the code, there is a lot that can be learned from just studying the sources.

A migration can be a good reason to use more third party code. This makes your own code base smaller, which means there is less code to maintain. You delegate some code maintenance to the third party.

It is not always easy to decide whether to use third party code (or which one to use). You should ask yourself the following questions before making a decision:

- Which dependencies and system requirements does the code have? Is a special PHP configuration required? Do these requirements fit into your project?
- What is the latest version, and what is the maturity (alpha, beta, release candidate, stable)? Is the software still maintained, and how often are new versions released?
- Under what license is the code? Is the code readable, obfuscated or encoded?
- How about the code quality? Does the code adhere to a coding standard? Is the code well documented and modularized?
- How many developers work on the code? Is there a company that backs up the development and guarantees continuity?
- Is there a roadmap, informing about the planned features? Does the roadmap fit to your project?
- Is the API stable over longer periods of time, and are new versions backwards compatible or will client code have to be adapted to new versions?
- Are there any open bugs? Are there many, potentially unanswered support questions, that might indicate general problems with the code?
- How is the documentation? Is example code available?
- Who is already using the code? Are there reference projects?

When you use third party code, you add a dependency to your application. You should therefore carefully compare the advantages and disadvantages of using third party code. If your answers to the above questions indicate a high risk, you should maybe refrain from using the code.

When migrating, you will have to make sure that any third party code is compatible with the target PHP version and configuration you are planning to use. You cannot rely on promises, but still have to test the third party code on the target system. When the API has changed, you might have to modify your application to fit to the new API when you switch to a newer library or component version.

If you are using third-party code that is not maintained any more, the question is whether it will still work on the target system, or how much effort it will take to make it work on the target system. You should do some research for potential alternatives, but can also consider taking over the maintenance of the third party code, if the license allows. Not all PHP code is open source code that allows derivative work, however.

Your Own PHP Code

Programming is life-long learning. When looking at code that was written months or years ago, there is usually much room for improvements, regardless of whether it is your code or somebody else's code. This is quite normal and not necessarily a sign of bad quality.

A migration is a good occasion to improve existing code. To keep the effort and the risks of a migration low, you should not necessarily start a general overhaul of the code. On the other hand, existing code is not unchangeable. A migration requires code to be adapted, otherwise you will not be able to fix the problems.

The right amount of code changes, for example by careful refactoring (see Chapter 5), can make a migration much easier, for example when duplicate code is removed before making changes, so that less changes have to be made. This will result in fewer errors that you have to fix or debug.

The big question how many changes to existing code will be necessary when migrating is not easy to answer. I have seen projects where existing PHP 4 code ran on PHP 5 without any change. Other projects had to make quite a lot of changes to make their PHP 4 code work on PHP 5. I have also seen projects where migrating from PHP 5 to PHP 5.1 or 5.1 to 5.2 had caused problems that required changes to the code.

Many of the required code changes are not called for by the new PHP version itself, but by changes to the PHP configuration. Projects with a strong PHP 4 history that keep using a PHP 4-like PHP configuration tend to run into more problems when migrating.

An interesting fact is that the effort that a PHP migration requires depends less on the code quality (which is hard to measure anyway), but more on which of PHP's language features the code uses to which extent.

Migrating PHP code can be done with not too much effort, when you work in turn and very disciplined. Chapter 7 introduces tools that will help you to successfully migrate your PHP application. Chapter 8 explains the problems that can occur in PHP code and shows possible solutions.

External Programs

It is easy and tempting to call external programs from PHP. There are lots of freely available command line tools, and some of them offer functionality that would require quite some PHP code to implement.

There are various PHP functions like `system()` and `exec()` to call external programs. For a full list and further explanations please refer to Chapter 8.

The output of command line programs depends on the operating system, language, and version used. You cannot expect the output to be the same across versions. What is more, the same program can behave different on another operating system, like the `ping` example shows.

As we know, `ping` sends packets to the given target system. On Unix, `ping` send packets until you quit the program, unless you specify the number of packets to send using the `-c` command line switch. The result of a `ping` on Unix could be:

```
> ping -c 4 google.de

PING google.de (66.249.93.104) 56(84) bytes of data.
64 bytes from google.com (66.249.93.104): icmp_seq=1 ttl=245 time=22.6 ms
64 bytes from google.com (66.249.93.104): icmp_seq=2 ttl=245 time=22.3 ms
64 bytes from google.com (66.249.93.104): icmp_seq=3 ttl=245 time=22.2 ms
64 bytes from google.com (66.249.93.104): icmp_seq=4 ttl=245 time=22.8 ms

--- google.de ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 22.216/22.508/22.813/0.251 ms
```

Windows also has a ping program. On Windows, ping always sends 4 packets, but does not understand the -c command line switch. The ping result on Windows could look like this:

```
> ping google.de

Pinging google.de [216.239.59.104] with 32 bytes of data:

Reply from 216.239.59.104: bytes=32 time=82ms TTL=245
Reply from 216.239.59.104: bytes=32 time=80ms TTL=245
Reply from 216.239.59.104: bytes=32 time=82ms TTL=245
Reply from 216.239.59.104: bytes=32 time=80ms TTL=245

Ping statistics for 216.239.59.104:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times im milli-seconds:
    Minimum = 80ms, Maximum = 82ms, Average = 81ms
```

Though both results show that Google is available, the result is different. If you do not only rely on the return value (or error code) of an external program, you will probably have to parse the output. So, in addition to making a system-dependent call, you will also have to parse the result depending on the system. The language is also an issue, because on a German Windows the output would be:

```
> ping google.de

Ping google.de [216.239.59.104] mit 32 Bytes Daten:

Antwort von 216.239.59.104: Bytes=32 Zeit=82ms TTL=245
Antwort von 216.239.59.104: Bytes=32 Zeit=80ms TTL=245
Antwort von 216.239.59.104: Bytes=32 Zeit=82ms TTL=245
Antwort von 216.239.59.104: Bytes=32 Zeit=80ms TTL=245

Ping-Statistik für 216.239.59.104:
    Pakete: Gesendet = 4, Empfangen = 4, Verloren = 0 (0 Verlust),Ca.
         Zeitangaben in Millisek.:Minimum = 80ms, Maximum = 82ms, Mittelwert =
         81ms
```

In reality, you will not only have to parse the “success” result, but also the various error messages.

While you can use the forward slash as directory separator in PHP on Windows and on Unix, any command line calls and parameters must use the backslash on Windows. Keep in mind that you have to escape the backslash with another backslash when it occurs in a string in your PHP code. When replacing backslashes slashes by forward slashes, keep in mind that the backslash is an allowed character in path names on Unix. You should take care to only replace on Windows, otherwise you risk damaging your paths.

```
<?php

  if ("\r\n" == PHP_EOL) $path = str_replace(’/’, ’\\’, $path);

?>
```

This program uses the built-in PHP constant `PHP_EOL` which contains the system's end of line character to detect whether it is running on Windows. Only then, forward slashes are replaced.

If possible, do not depend on any output of external programs in your application. Instead, try to find an alternative implementation as PHP code or a PHP extension that reliably works cross-platform.

Special care must be taken when calling external programs with path and file names containing blanks. If the external program expects command line parameters, it cannot distinguish between a blank in a path and the separator between two command line parameters. Put path names in double quotes, but make sure to escape any quotes in the name.

Since path names can change when migrating, it is good practice to always quote paths to keep your application portable. Also keep in mind that paths differ between systems. On Unix, the temporary directory is `/tmp`, whereas on Windows it usually is `c:\windows\temp`, at least when PHP runs inside a web server. When run at the command line, the temporary directory of the current user is used. The path to this temporary directory usually contains blanks.

Try to never use fixed paths when calling external programs, since you make your application dependent on a certain directory structure on the target system. Use the system search path instead (as we discussed earlier) to keep your application portable. To have use the search path, you must execute external programs in a shell rather than call them directly. Without a shell, the environment the external program is executed may differ from the PHP user's environment. This is a rather frequent source of errors.

To execute an external program, execute rights on the executable file are required for the user PHP is running as. You need to know which user account PHP runs under and whether PHP runs in a chroot jail. If so, you must make sure that the program is available in the chroot jail. On Windows, when PHP runs under Apache, it runs in the System account, which is usually allowed to access all files on the system. IIS runs in its own user account, so you may have to set the access rights accordingly to allow PHP to start an external program.

Try to avoid calling external programs on production systems as far as possible. Quite often, external programs are just used to save some lines of PHP code. In that case, a PHP implementation allows you to get rid of another external dependency your application has.

External programs should be replaced for testing, to make your tests less dependent on the system environment. Either you replace the external program by a dummy program that provides you with the expected answers, or you modify the PHP source to not call the external program in test mode, but use a pre-calculated result instead. Earlier in this chapter, we have seen code that behaves differently when in test mode.

Replacing external programs also helps you to test certain errors conditions. Many conditions like wrong access rights, a broken network link, or a file system that runs our of free space are hard to test otherwise.

Interfaces to Third-Party Systems

Since Gartner Group has introduced the term "service oriented architecture" in 1996, it has become one of the big buzzwords in IT. The idea of a service oriented architecture is to create a set of independent services instead of a monolithic software. Basically, this is the logical consequence of modular, object-oriented thinking.

Today, most PHP applications are consumers of web services, often without the developers even noticing. Using an RSS feed, for example, is using a web service. Most web services are used via HTTP. Other, more complex protocols include XML-RPC or SOAP. SOAP has become rather popular recently, because the new PHP versions have built-in SOAP support. There are other, partly proprietary protocols to use remote systems like Corba, or DCOM.

Typical interfaces to third-party systems are order and reservation systems, external search engines, or payment processing systems. Quite often, APIs of services like Flickr or YouTube are used today, for example to selectively integrate content into an application.

You should decouple your application from third-party services when migrating, since you probably do not want to start transactions in remote systems when testing your software. You'll probably also not want to deal with real-world credit cards when testing software.

Many web service providers offer a test mode that is triggered by using a different URL, for example. In test mode, often called *sandbox*, the remote system will behave exactly like in a real transaction, but never execute the transaction. You should add a switch to your application, selecting either test mode or production mode. If the test mode is activated, remote calls are either simulated or the service's sandbox mode is being used.

If your test system has limited or no Internet access, you cannot reach the third-party system. In this case, you could write a dummy web service simulating the remote system, or have the software component using the remote service return a pre-calculated result in test mode.

Character Encodings

In the previous sections, we have already mentioned problems with different character encodings. Though this is an important topic, many developers do not really care about encodings too much. As a matter of fact, many PHP applications today work rather well with UTF-8 data read from a database. As soon as you start to modify the data at PHP level, however, the real problems start. Character encoding problems usually come to light as presentation problems, with weird characters showing up in the output. Frequently, this happens when migrating a system.

A string, like a text file or HTML page, in essence is just a sequence of bytes. This byte sequence must be interpreted to process the content it represents. There are many different ways of encoding text, which includes PHP source code. Since one can normally not tell the encoding of a byte sequence just by looking at the data, developers must keep track of which encodings are used in their files.

The basics of character encoding were already covered in the section entitled “Character Sets” in this chapter, where we also gave an overview of common encodings. When UTF-8 is used, the advantage is that you will only have to use five HTML entities, because all special characters are directly supported. Table 5.5 shows special characters that must be encoded as HTML entities in UTF-8.

Character	HTML-Entity	Name
&	&amp;	Ampersand
"	&quot;	Double Quotes
’	&apos;	Single quote
<	&lt;	Less than sign
>	&gt;	Greater than sign

To learn how to avoid typical character encoding problems, we will experiment a little with text in different character encodings. We will use a German sentence that does not make any sense, but contains many umlaut characters, which are non-ASCII characters:

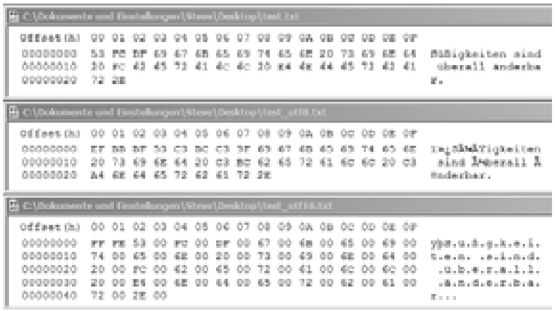
```
Süßigkeiten sind überall änderbar.
```

We will store this text in different encodings, namely ISO 8859-1, which is still the default in many text editors, in UTF-8, and in UTF-16. We will use UTF-8 with and without byte order mark (BOM), and UTF-16 in big-endian and little-endian format (see chapter 3). While UTF-16 always contains a byte order mark, the byte order mark in UTF-8 is superfluous, because the encoding already defines the byte order. Still, many text editors add the byte order mark at the beginning of a UTF-8 file, to denote UTF-8-encoded text.

```
15.11.2007  12:35                34 test.txt
15.11.2007  12:50                68 test_utf16.txt
15.11.2007  12:52                68 test_utf16_little_endian.txt
15.11.2007  12:37                41 test_utf8.txt
15.11.2007  12:46                38 test_utf8_ohne_bom.txt
               5 File(s)            249 Bytes
```

The directory listing shows that the file length is different for the various encodings. UTF-16 uses the most memory, since every character is encoded with at least two bytes. In our example, UTF-8 is only a little larger than ISO 8859, because the store text mainly consists of ASCII characters.

You will probably find the UTF-8 byte order mark annoying in PHP source code, since is appears as ï»¿ in the output. Since this starts the output before PHP code is even parsed, it is not possible to send HTTP headers any more (see Chapter 8 for more information). Since the BOM appears before the first PHP tag, even turning on output buffering will not help. You must remote the BOM from the source files. This is not easy, because the BOM are no visible characters, so you cannot use a text editor to remove it. You must either explicitly store the file as UTF-8 without BOM or use a command line tool to remove the first three bytes. Figure 3.4 shows the different character encodings as displayed in a hex editor.

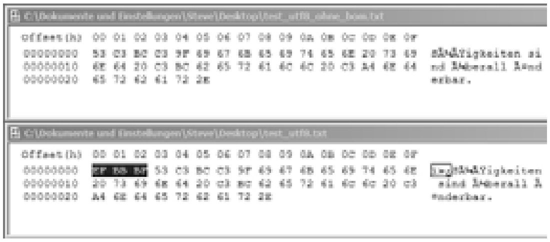


```
C:\Dokumente und Einstellungen\Steve\Desktop\test.txt
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  53 FC DF 69 67 6B 65 69 74 65 6E 20 73 69 6E 64  Süßigkeiten sind
00000010  20 FC 62 65 72 61 6C 6C 20 E4 6E 64 65 72 62 61   überall änderba
00000020  72 2E                                            r.

C:\Dokumente und Einstellungen\Steve\Desktop\test_utf8.txt
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  EF BB BF 53 C3 BC C3 9F 69 67 6B 65 69 74 65 6E  ï»¿SÃ¼ÃŸigkeiten
00000010  20 73 69 6E 64 20 C3 BC 62 65 72 61 6C 6C 20 C3   sind Ã¼berall Ã
00000020  A4 6E 64 65 72 62 61 72 2E                       ¤nderbar.

C:\Dokumente und Einstellungen\Steve\Desktop\test_utf16.txt
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  FF FE 53 00 FC 00 DF 00 67 00 6B 00 65 00 69 00  ÿþS.ü.ß.g.k.e.i.
00000010  74 00 65 00 6E 00 20 00 73 00 69 00 6E 00 64 00  t.e.n. .s.i.n.d.
00000020  20 00 FC 00 62 00 65 00 72 00 61 00 6C 00 6C 00   .ü.b.e.r.a.l.l.
00000030  20 00 E4 00 6E 00 64 00 65 00 72 00 62 00 61 00   .ä.n.d.e.r.b.a.
00000040  72 00 2E 00                                      r...
```

Figure 3.4

To better understand the various text encodings, let us view the files in a text editor. In the top window in Figure 3.4, we can see that every character is encoded by one byte. The second example starts with the byte order mark 0xEFBBBF, and each character is encoded in variable length, as you can tell by looking at the umlaut characters. In the ASCII view to the right, they are scrambled. Figure 3.5 shows the UTF-8 encoded file with and without byte order mark. In the bottom window, the BOM is highlighted. In ASCII view, it appears as ï»¿.

As Figure 3.6 shows, UTF-16 encoding uses two bytes for each character. Our example text contains no characters that require four bytes to encode. The byte order

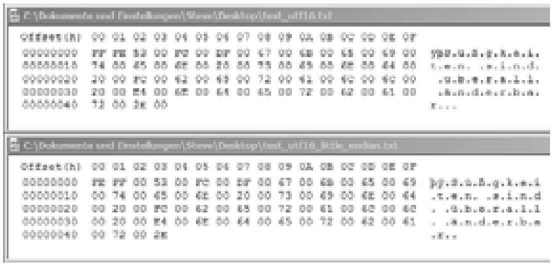


```
C:\Dokumente und Einstellungen\Steve\Desktop\test_utf8_ohne_bom.txt
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  53 C3 BC C3 9F 69 67 6B 65 69 74 65 6E 20 73 69  SÃ¼ÃŸigkeiten si
00000010  6E 64 20 C3 BC 62 65 72 61 6C 6C 20 C3 A4 6E 64  nd Ã¼berall Ã¤nd
00000020  65 72 62 61 72 2E                                erbar.
```

```
C:\Dokumente und Einstellungen\Steve\Desktop\test_utf8.txt
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  EF BB BF 53 C3 BC C3 9F 69 67 6B 65 69 74 65 6E  ï»¿SÃ¼ÃŸigkeiten
00000010  20 73 69 6E 64 20 C3 BC 62 65 72 61 6C 6C 20 C3   sind Ã¼berall Ã
00000020  A4 6E 64 65 72 62 61 72 2E                       ¤nderbar.
```

Figure 3.5

is defined by the a 16 bit byte order mark, which is stored at the beginning of the file. 0xFFFE denotes big-endian UTF-16, whereas 0xFEFF denotes Little Endian UTF-16.



```
C:\Dokumente und Einstellungen\Steve\Desktop\test_utf16.txt
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  FF FE 53 00 FC 00 DF 00 67 00 6B 00 65 00 69 00  ÿþS.ü.ß.g.k.e.i.
00000010  74 00 65 00 6E 00 20 00 73 00 69 00 6E 00 64 00  t.e.n. .s.i.n.d.
00000020  20 00 FC 00 62 00 65 00 72 00 61 00 6C 00 6C 00   .ü.b.e.r.a.l.l.
00000030  20 00 E4 00 6E 00 64 00 65 00 72 00 62 00 61 00   .ä.n.d.e.r.b.a.
00000040  72 00 2E 00                                      r...
```

```
C:\Dokumente und Einstellungen\Steve\Desktop\test_utf16_little_endian.txt
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  FE FF 00 53 00 FC 00 DF 00 67 00 6B 00 65 00 69  þÿ.S.ü.ß.g.k.e.i
00000010  00 74 00 65 00 6E 00 20 00 73 00 69 00 6E 00 64  .t.e.n. .s.i.n.d
00000020  00 20 00 FC 00 62 00 65 00 72 00 61 00 6C 00 6C  . .ü.b.e.r.a.l.l
00000030  00 20 00 E4 00 6E 00 64 00 65 00 72 00 62 00 61  . .ä.n.d.e.r.b.a
00000040  00 72 00 2E                                      .r..
```

Figure 3.6

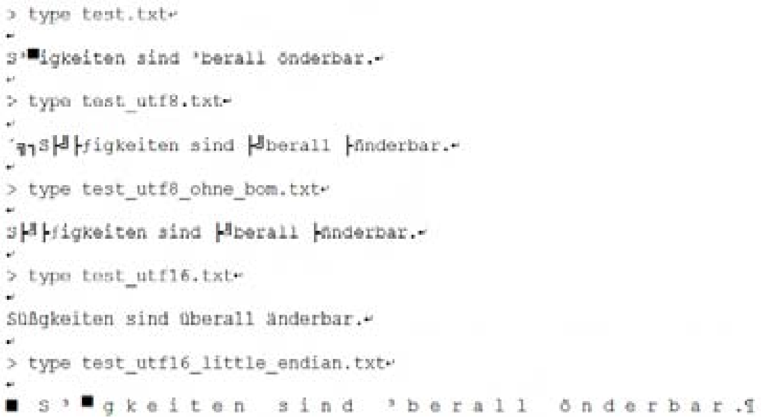
Let us now output the text that we have stored. We will start on Windows:

Only the big-endian UTF-16 file is displayed correctly, which is no surprise, as Windows works with this encoding internally.

Figure 3.8 shows the same output on Linux. Here, only UTF-8 without byte order mark is displayed correctly, since this Linux internally uses this encoding. The output looks a little better than on Windows, because the Linux shell replaces every non-printable character by an inverted question mark, whereas Windows displays all kinds of strange characters.

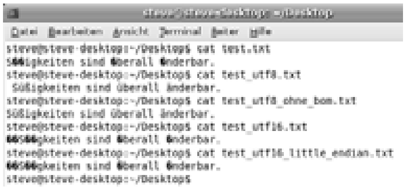
Now let us write a little PHP script that outputs our example sentence. We will then store this program in various encodings and experiment with different Content-Type headers and character set conversions:

```
<?php
```



```
> type test.txt↵
↵
S³■igkeiten sind ³berall önderbar.↵
↵
> type test_utf8.txt↵
↵
´╗┐S├╝├ƒigkeiten sind ├╝berall ├ñnderbar.↵
↵
> type test_utf8_ohne_bom.txt↵
↵
S├╝├ƒigkeiten sind ├╝berall ├ñnderbar.↵
↵
> type test_utf16.txt↵
↵
Süßgkeiten sind überall änderbar.↵
↵
> type test_utf16_little_endian.txt↵
↵
■ S ³ ■g k e i t e n   s i n d   ³ b e r a l l   ö n d e r b a r .¶
```

Figure 3.7



```
steve@steve-desktop:~/Desktop$ cat test.txt
S��igkeiten sind �berall �nderbar.
steve@steve-desktop:~/Desktop$ cat test_utf8.txt
 Süßigkeiten sind überall änderbar.
steve@steve-desktop:~/Desktop$ cat test_utf8_ohne_bom.txt
Süßigkeiten sind überall änderbar.
steve@steve-desktop:~/Desktop$ cat test_utf16.txt
��S��gkeiten sind �berall �nderbar.
steve@steve-desktop:~/Desktop$ cat test_utf16_little_endian.txt
��S��gkeiten sind �berall �nderbar.
steve@steve-desktop:~/Desktop$
```

Figure 3.8

```
  // Stored as ISO8859-1

 echo "Die Süßigkeiten sind überall änderbar.";

?>
```

In default configuration, PHP sends a Content-Type header announcing ISO 8859-1 encoded text. Since the source code is in this very character set, everything will work just fine:

```
Die Süßigkeiten sind überall änderbar.
```

Now we change the Content-Type header and announce UTF-8-encoded text:

```
<?php
  // Stored as ISO8859-1
  header('Content-Type: text/html; charset=UTF-8');
  echo "Die üßSigkeiten sind überall änderbar.";
?>
```

We are announcing UTF-8, but sending ISO 8859-1. PHP does not convert the data, so the umlaut characters are not displayed correctly:

```
Die S??igkeiten sind ?berall ?nderbar.
```

We can work around the problem by actually sending UTF-8 instead of ISO 8859-1. To convert the string we output, we use the function `iconv()`, which converts between different character sets:

```
<?php
  // Stored as ISO8859-1
  header('Content-Type: text/html; charset=UTF-8');
 echo iconv("ISO-8859-1", "UTF-8", "Die Süßigkeiten sind überall änderbar.");
?>
```

In this example, the ISO 8859-1 string in the source code is converted to UTF-8, which leads to correct display, since the used character set matches the headers we send:

```
Die Süßigkeiten sind überall änderbar.
```

When we store the source file as UTF-8, we do not have to convert at PHP level:

```
<?php
```

```
  // Stored as UTF-8
  header(’Content-Type: text/html; charset=UTF-8’);
 echo "Die Süßigkeiten sind überall änderbar.";
?>
```

This program works with UTF-8 throughout, so again the output matches the header, and the string is displayed correctly:

```
Die Süßigkeiten sind überall änderbar.
```

If we store the source code with a byte order mark, it will show up in the output:

```
ï»¿Die Süßigkeiten sind überall änderbar.
```

We can also convert the other way, and store UTF-8 but send ISO-8859 to the browser:

```
<?php
  // Stored as UTF-8
  header(’Content-Type: text/html; charset=iso8859-1’);
 echo iconv("UTF-8", "ISO-8859-1", ""Die Süßigkeiten sind überall änderbar.");
?>
```

This program will display the string correctly:

```
"Die Süßigkeiten sind überall änderbar.
```

Keep in mind that “down” converting from UTF-8 to ISO 8859 will only work when the string contains only characters that also exist in ISO-8859. Let us try an example with a character that does not exist in ISO-8859:

```
<?php

  // Stored as UTF-8

  header('Content-Type: text/html; charset=iso8859-1');

  var_dump("Total: Σ");

?>
```

Since the sum sign does not exist in ISO-8859, the conversion is not successful:

```
string 'Total: âˆ‘' (length=10)
```

One could argue whether the conversion should fail in this case, instead of returning nonsense characters.

Avoid converting text multiple times, which can happen by accidentally converting text that is already in the desired encoding. In the next example, we store our file as UTF-8 and send UTF-8 to the browser, but still convert the string at runtime as if it were ISO-8859-1:

```
<?php

  // Stored as UTF-8

  header('Content-Type: text/html; charset=iso8859-1');

 echo iconv("ISO-8859-1", "UTF-8", "Die Süßigkeiten sind überall änderbar.");

?>
```

Since the string is already in UTF-8, the ASCII characters remain unchanged, whereas all non-ASCII characters that are encoded with two bytes are interpreted as two single ISO 8859-1 characters, and thus converted to nonsense:

```
Die ÃƒÂ¼ÃƒÂŸSigkeiten sind ÃƒÂ¼berall ÃƒÂ¤nderbar.
```

Though PHP can deal with multi-byte string, as we have seen, the built-in string functions like substr() assume that one character is always encoded by one byte. Let us try our luck with calculating a substring of a UTF-8 encoded word:

```
<?php
  // Stored as UTF-8
  header('Content-Type: text/html; charset=UTF-8');
  echo substr("üßSigkeiten", 0, 3);
?>
```

We would expect PHP to output three characters, but since the ü is encoded with two bytes, the result only contains two characters:

```
string 'Sü' (length=3)
```

As you can see, xdebug, which is responsible for counting the length of the string, counts bytes and not characters. To avoid these errors, you must use the PHP extension `mbstring`, or use the string functions of the `iconv` extension. The advantage of `mbstring` is that, when loaded, it overrides the native PHP functions and makes them work with multi-byte strings. `iconv`, on the other hand, is enabled in PHP by default, but requires to not use the native PHP string handling functions.

```
<?php
  // Stored as UTF-8
  header('Content-Type: text/html; charset=UTF-8');
 var_dump(iconv_substr("Süßigkeiten", 0, 3, 'UTF-8'));
?>
```

Now the output is as expected, but again, xdebug counts bytes, not characters, so it reports an incorrect string length.

```
string ’Süß’ (length=5)
```

PHP 6, by the way, will be able to deal with multi-byte character sets out of the box.

Browser

The browser, one of the most important components of a web application was not even displayed in the figure at the beginning of this chapter. There are various browsers, but many of them are not very widespread. The most commonly used browsers are Internet Explorer, Firefox, Opera, Safari, and Konqueror. Browsers send HTTP requests to the server and render the HTML code that the server sends back. Javascript, which in former years was often disabled for security reasons, has become indispensable due to the AJAX boom. Today, many web sites and web applications don’t work without Javascript.

HTML was never intended to be pixel-perfect. Different browsers may render HTML pages differently, and also Javascript is not always guaranteed to behave identically on different browsers. Many factors like browser configuration, screen resolution, color depth, installed fonts, and used font size can influence rendering of a HTML page. All these factors are beyond our control from the PHP viewpoint, so we will not deal with them any further.

Security

In its early days, the Internet was a friendly environment. Today, security is a very important of every system and application, and firewalls play an important role in most security concepts.

One or more firewalls can be placed between the browser and the server, limiting access in various ways. If just one of the firewalls is too restrictive, your application might not seem to work correctly. When migrating, I suggest to start in a clean and safe environment without any firewall security, then adding security when you are sure that your applications and services work as expected. It can be very difficult to debug a problem caused by a firewall.

Chapter 4

Preparing the Migration

“Not the wind, but the sail determines the direction.” —*(Chinese Proverb)*

Steps to Preparing the Migration

This chapter deals with planning and preparing the migration. A migration is a complex issue, and you can save yourself a lot of time and effort when you plan ahead and proceed in little steps.

The first step is analyzing the existing application and environment. The various aspects you have to keep in mind have been discussed in the previous chapter. Analyzing the existing PHP code will allow you to get a feeling for the code quality. The better structured and modularized the code is, the easier it becomes to make local modifications to the code without affecting the whole application.

The next step is to define the target system. In most cases, you will want to use the current version of all software modules used. Sometimes, though, there are dependencies between certain versions that must be taken into account.

Now you can break down the whole migration process into small steps, and prioritize them. Since the priority of some steps often depends on external influences, it is not always easy to prioritize the steps. Keep in mind, that breaking down the project into little steps is crucial, because this makes it easier to track down any problems that you might encounter.

The Existing Application and Environment

In the last chapter we introduced the various software components that make up a PHP system, and we have learned how they interact. Now it is time to a look at your existing PHP environment and application to understand how the different components are integrated. This information will be the basis for defining the target system later.

While it is obvious which operating system is used, you should find out about the exact version number and which patches, hot fixes and service packs are installed. PHP is to a large extent independent from these operating system details, but it is generally a good idea to use a similar operating system installation and configuration on the target system, unless of course you plan to migrate to another operating system or operating system version.

It is not always easy to find out about the operating system architecture and processor of your system. On Windows, you can run `systeminfo` at the command line to find out which processor your computer has. On Unix, use `cat /proc/cpuinfo` to display information about the processor and system architecture. If you plan to migrate to another operating system, choose one that is optimized for your system's architecture (see Chapter 3).

To find out about the MySQL version being used, call the MySQL executable at the command line, using the switch `-version`:

```
> mysqld-nt --version

mysqld-nt  Ver 5.0.27-community-nt-log for Win32 on ia32
           (MySQL Community Edition (GPL))
```

This example shows the call and result on Windows; on Unix the executable file is usually just called `mysql`. Alternatively, you can use a tool like phpMyAdmin to display information about the version and configuration of the database. Then, find out which configuration files your databases uses. MySQL usually uses a configuration file `my.ini` or `my.cnf`. Compare these to the default configuration files coming with the MySQL distribution to find out about where the active configuration differs. When installing the target system, you will have to decide whether to keep the default settings or use the configuration settings of your existing system.

To find out about the web server version, and display additional information, again call the executable file with additional command line switches. For Apache, this is the file `httpd` on Unix and `Apache.exe` on Windows:

```
> Apache -v

Server version: Apache/2.0.59
Server built:   Jul 27 2006 15:55:03

> Apache -V

Server version: Apache/2.0.59
Server built:   Jul 27 2006 15:55:03
Server's Module Magic Number: 20020903:12
Server loaded:  APR 0.9.12, APR-UTIL 0.9.12
Compiled using: APR 0.9.12, APR-UTIL 0.9.12
Architecture:   32-bit
Server compiled with....
 -D APACHE_MPM_DIR="server/mpm/winnt"
 -D SERVER_CONFIG_FILE="conf/httpd.conf"
 -D SUEXEC_BIN="/apache/bin/suexec"
 [...]
```

The first call displays the server version, while the second call shows additional information about the options Apache has been compiled with. This tells you which thread model and MPM Apache uses, and which configuration file is used.

The web server's configuration file will tell you how PHP has been integrated into the web server (see Chapter 3). In newer Apache versions, the configuration is often split into various files that are included into `http.conf`.

For information about PHP, run this code in the web server:

```
<?php

  phpinfo();

?>
```

You will not only see the exact PHP version number, but also the server API (SAPI) PHP uses. The line `Loaded Config File` tells you which configuration file PHP is using, and the line Thread Safety tells you whether PHP is thread-safe. Normally, PHP should be thread-safe.

The phpinfo() output also shows which PHP extensions are used and how they are configured. To display a shorter list of PHP extensions, run

```
> php -m
```

at the command line. Keep in mind that the PHP configuration at the command line need not be the same as the PHP configuration in the web server. PHP can not only be configured in php.ini, but also in httpd.conf, .htaccess files and with ini_set() at script runtime. To find out about the configuration PHP is actually running in, use the function ini_get_all() (see Chapter 7).

Many migration problems are related to different PHP configuration of the existing system and the target system, and some PHP applications modify the PHP configuration at runtime, possibly only for certain directories.

In Chapter 3, we listed the most important PHP configuration settings. You can compare the configuration your existing PHP runs in with the default configuration file php.ini-recommended that comes with your PHP distribution.

The following program displays information about the user id, group and user name of the user PHP and thus the PHP scripts are running under.

```
<?php

  var_dump(getmyuid());
  var_dump(getmygid());
  var_dump(get_current_user());

?>
```

You should run this program in the web server and not at the command line, because at the command line your operating system user name will be displayed.

Alternatively, you can create and save a file from a PHP program. The owner of this file is the user executing the PHP program. In addition to the file owner, access rights are very important, especially on Unix (see Chapter 3).

Now, search all directories in PHP's include_path for code that is used by your application. You will have to take this code into account when migrating the system.

To find out about all the external dependencies of your applications, search the code for the PHP function calls listed in Chapter 8.

Another kind of external dependencies are remote services your application might use. To list these, you must search your code for all file access function calls and all places where HTTP, XMLRPC, or SOAP clients are used.

There is a simple trick to get rid of include_path dependencies in your application. Copy the contents of all directories listed in the include_path in a subdirectory of your application's root directory, for example lib. Then, at the beginning of your PHP program, add:

```
ini_set('include_path', '.' . PATH_SEPARATOR . 'lib'.);
```

This will adjust include_path accordingly. Now your application has no more dependencies on PHP code outside the your document root (the www, "htdocs", or "wwwroot" directory).

Now you can start analyzing which PHP extensions your application requires. A good starting point is PHP_CompatInfo, which analyzes PHP source code and tries to figure out which PHP extensions it requires (see Chapter 7). PHP_CompatInfo, however, does not take classes into account, but only function calls, so when the object-oriented API of a PHP extensions is used, PHP_CompatInfo does not list the dependency.

The Target System

You have now gained a first overview of your existing system. Next we will use this information to define the target system we migrate onto.

In most cases, the latest version of all software components (operating system, database, web server, and PHP) will be used, but in some cases when there are compatibility problems between certain components, you might have to choose an older version.

You should start by listing the latest release version numbers of all used software components. If there are no known problems in getting them to work together, this list of versions should reflect your ideal target system. Should problems turn up later, you might have to revisit this list and choose a different version of one or more components to work around the problem. The goal is to find a set of software components that work together in a stable and reliable way.

If your project allows it, you should use PHP as an Apache2 module. You should not use CGI or FastCGI, except if your application is supposed to run on a shared hosting server. If you run PHP as CGI or FastCGI for good reason, you need to have a close look at all the file accesses of your application, and which access rights files are created with, otherwise your application might not work correctly on the target system.

Most PHP applications use third party code like PEAR classes, or frameworks. When migrating, you must make sure that third party code runs on the target system. There might be a more current version in use than the one your application is using. If that is the case, you need to make sure that the API has not changed before upgrading to the newer version. If the API of a component has changed, you must adjust your code accordingly.

In Chapter 3 we learned how to output a list of PHP extensions installed on the existing system. Compare this list with the minimum requirements `PHP_CompatInfo` has created for you. Now you have to find out whether all required extensions still exist for the current PHP version. Chapter 3 shows an overview of PHP extensions that are no longer maintained, and lists possible alternatives.

Not all PHP extensions can be used together. It is not possible, for example, to use xdebug with any Zend extension. If you have problems with a certain PHP extension, you can consider replacing it by an external program or by PHP code. PHP code, of course, will always be slower than a PHP extension.

{index target system!reducing complexity of}}Try to reduce the complexity of your target system. At least while migrating, you should leave out all components that are not really required. This includes PHP accelerators, optimizers, proxies, and security measures like firewalls. This reduces the amount of possible sources of errors. It does make your application slower and less secure, of course, but makes finding and fixing of bugs much easier. After the migration, when the system has been stabilized, you can add the complexity again step by step.

Let us assume that you have the following existing system:

- Windows 2000
- PHP 4.3.11

- Apache 1.3.39
- PHP is installed as Apache module
- the PHP extensions mysql, Session, and XML are used
- MySQL 4.0.12
- Turck MMCache 2.4.6

Windows 2000 is rather old. Since I (like many others) would not recommend using Windows Vista in production, you should consider using Windows XP, or even better switching to Unix. This would also save you the license cost of a Windows Server operating system.

PHP 4.3.11 is very old. PHP 4.4 fixes a lot of problems, especially with reference handling, so programs running on PHP 4.3 might not work properly on PHP 4.4. It might be a good idea to upgrade PHP to 4.4 as a first step, and see how the application behaves on a test system. This allows you to rule out differences in reference handling as a source of errors when upgrading to PHP 5.

Apache 1.3.39 is (at the time of writing) the latest available version of the 1.3 series, but the newer 2.0 and even 2.2 series have been available for quite a while now. Consider upgrading Apache to version 2.2.6 but keep in mind that the Apache configuration has changed quite a bit since 1.3, so you will probably not be able to use your 1.3 configuration file for Apache2 without modifications.

The mysql and Session PHP extensions are enabled in PHP 4 and 5 by default, so there should be no problems with these. The XML extension in PHP 5, however, is based on another library than in PHP 4, which also means that the extension API is different in PHP 5. You will have to adapt your PHP code for the new extension (see Chapter 8). Depending on the application, you could also consider using the XMLReader or XMLWriter extension.

MySQL 4.0 is rather old as well. The current version is 5.0.45, which supports stored procedures and triggers (see Chapter 3). Upgrading MySQL, however, is usually pretty seamless, so you will probably not experience considerable problems. You will have to decide whether to use Unicode support of the database, and convert your data accordingly (also in Chapter 3).

Turck MMCache is no longer under active development, as the author, Dmitry Stogov, now works for Zend. Consequently, Turck MMCache is not compatible with current PHP versions. Besides the fact that on the target system no PHP bytecode cache should be used until you have stabilized your software and environment, it is probably a good idea to use the Alternative PHP Cache (APC) in the future. APC is open source and developed by various well-known core PHP developers, so that compatibility to future PHP versions should be ensured.

Planning the Migration

To break down the migration into several smaller steps, you should now try to identify separate code modules of your application. If the code is already modular, this is very easy. If not, try to use files, classes, commands or screens as the criteria to break down the application into modules. Every third-party-class, -library and -component should become a module of their own.

The goal of breaking down the application into modules is to be able to migrate the application module by module. The better the isolation between the modules, the easier they can be tested when migrating. If the application is not well structured, you can consider improving the design by refactoring prior to migrating the code (see Chapter 5).

Now list each module of your application. For each module, you can choose one of the following three migration paths:

- Keep the existing code and solve the problems that turn up when migrating one by one
- Clean up the code by refactoring before migrating
- Replace the module by another implementation, and test to make sure that the application's behavior did not change

The decision about which path to choose is not always final. Sometimes, problems turning up while migrating the code might make you change your decision. An example would be deciding to reimplement a module that does not work on the target system after having spent some time trying to fix bugs without success.

Now you must prioritize migrating the modules. If the application has several layers, you should work from bottom to top since this makes it easier to isolate any occurring errors.

Third-party code is usually easy to migrate, since in most cases you only need to update to the latest version. When the API has changed, either modify the application code or write an adapter that translates the old API calls to the new API of the third-party component. I would recommend starting to migrate libraries and third-party components first, and test them in isolation.

Replacing implementations as mentioned above can be a useful tool to make migration easier. You can consider removing logging code, and use xdebug instead when you need a log file (see Chapter 7). Code based on the XML extension in PHP 4 or another extension that is not maintained any more, must be rewritten for another extension, unless you can find an alternative PHP implementation.

Migrating an application is a good reason to use more third-party code. Consider replacing custom libraries and classes by code from mainstream frameworks. This means that you will have to adapt your code, but reduces the amount of code you will have to maintain in the future.

Keep in mind, however, that most current frameworks require at least PHP 5. The same holds true for current versions of PHPUnit. If you choose to use such code, you cannot run your application on PHP 4 any more, which is one of the reasons why you will need two test systems, as we will see later.

When migrating your own code, you should also work bottom-up and start with the main functionality of the application. Focus on the success cases rather than the obscure edge cases. You will have to test edge cases at some point, but try to resist the temptation of testing more and more edge cases at first.

Before you can start migrating the PHP code of your application, you have to decide how to you want to migrate the environment. When you migrate the whole environment in one big step, you do not know where to relate potential errors to. It may thus be better to migrate the environment in a series of smaller steps, each of which is can be a full migration cycle.

When you plan to migrate to another operating system, you should do so in a separate step. Chapter 3 explains the most important differences between the major operating system families. Also as seen in Chapter 8, when you switch the database vendor, you should do so in a separate migration step as well. Upgrading to a newer version of the same database is usually less problematic. Migrating the web server is usually not a big issue as well, so you can migrate the web server when migrating PHP itself.

It can be a good idea to migrate the whole environment except PHP in one step, fix the problems that turn up, then migrate PHP and adjust the PHP configuration on the target system. Many problems that occur while migrating are due to different configuration rather than the different PHP version used.

I would recommend working towards running your application in a default configuration (see Chapter 5), since this configuration has been tested the most and ensures maximum portability.

To sum it up, the example system given above could be migrated using the following steps:

- Deactivate Turck MMCache
- Migrate from Windows to Linux, still using PHP 4.3
- Migrate to MySQL 5
- Migrate to PHP 4.4 on Apache 2.2
- Migrate to PHP 5.2
- Activate APC

Chapter 5

The Migration

"A bird does not sing because it has an answer. It sings because it has a song."
—(Chinese Proverb)

Now we will start to migrate the PHP application. You should start by setting up two test systems, to be able to test your application in the existing environment and in the target environment in parallel. By comparing the test results on both test systems, you can track down possible sources of errors.

The next step is identifying relevant test cases. Start off by creating a list of main features of your application. Usually, each feature will need several test cases, to test the business logic for the normal case and the most important errors.

To successfully test the application, you will need test data. It is not always easy to identify and create test data, especially when you work with sensitive data like credit card or cell phone numbers.

Usually, there are not too many automated tests for existing applications. As you probably will not want to repeatedly test your application by hand, you should create automated tests.

To reduce the amount of necessary code changes, and thereby also reduce the probability of introducing bugs while migrating the code, you might want to refactor the code. In practice, you will probably not refactor code until you encounter problems when migrating.

Migrating the code comprises several steps. First of all, you need to make sure whether the existing code always works as expected or if there are errors under certain conditions. I would recommend fixing these errors first, so that you can differentiate between existing errors and those introduced in the migration process.

Now is the time to replace or reimplement application modules. Sometimes, it is less work to reimplement a module than spending hours tracking down bugs and trying to fix them.

When you run the application (or certain parts or modules of it, respectively) on the target system, you can encounter various problems that you have to fix one by one. Common errors are name conflicts, case problems, and problems related to differences between the PHP versions.

First up, you need to fix the errors that prevent the code from actually being executed. Next, you should modify the code so that no notices, warnings, and errors are displayed. As every PHP error message indicates a potential problem of the code, so you are taking great risk of running into trouble later when you ignore them.

Now your application should run in the target environment without runtime errors. This does not mean, however, that the application always works as expected, or works as it does in the old environment. You must now try to find the all bugs that do not lead to PHP errors on the target system.

Another optional, but strongly recommended step is to make your application work in a default PHP configuration. This makes it more portable and will also make the next migration easier.

To make sure that the application does not only work as expected on the test system, but also on the live system, you must perform integration testing. While we have tried to keep the test system as simple as possible, to reduce the amount of possible error sources, you should now test the system with all components to increase performance and security added to it.

The best way to migrate a live system is not to touch the existing system, but install the new environment on new hardware. Switching from the existing production system to the newly installed system can be done in two ways: either you switch at once, or you run both systems in parallel for a while and gradually migrate.

Preparations

To successfully migrate, you need two independent test systems. This allows you to compare the test results on both systems. The first test system represents the existing system your application currently runs on, while the second test system represents the target system you are migrating to.

You should never use live systems for testing, as you will have to tweak the system configuration, which is likely to cause problems with your live application. By testing on a live system, you risk downtime and corruption or even loss of data.

Consider using virtual machines as test systems. Virtual machines are slower than physical hardware, but for the tests this should not really matter. An advantage of using virtual machines is that you can store snapshots of the installed systems, and instantly rollback to a saved snapshot if you have run into serious trouble.

The First Test System

When installing the first test system, you should leave out all unnecessary complexity like PHP accelerators, caches, and security measures like firewalls, unless of course they are required by the application. You should also not use any proxy servers, as they do not always play nice with rapidly changing files, and sometimes serve stale content, which can lead to hard-to-reproduce errors on the client.

If you use a virtual machine as test system, you might be able to convert the existing physical live system into a virtual machine. Some software vendors offer tools for that. As a fallback, you can always install the test system from scratch. In that case, make sure that you use the same versions of all software components. Any deviation could lead to errors or result in altered behavior of the application.

If you cannot set up a test system that resembles the existing system, for example because you have no more access to the software components that were used, you can try cloning the existing live system. To make this work, you will need to use identical hardware for the test system.

Cloning systems is not always an easy task. When cloning Windows, for example, you must change the system identifier (SID), at least when you plan to run both systems in the same network. Quite often, the system configuration of the cloned system has to be adapted, for example because the MAC address of the network adapter has to be changed.

The Second Test System

The second test system is usually easier to install, as you will use the latest version of all existing software components in most cases. Make sure to meticulously document the installation of the test system. I would recommend reinstalling the test system based on the documentation you have created, to make sure the documentation is accurate. You will later have to install the live system based on this documentation, so spending more time in getting the installation process right and documenting it will definitively pay off on the long run.

The second test system should be kept as simple as possible. While testing, performance and security are not really important. After you have installed both test systems, you should store all source codes and binary installation files that you have used in your version control system. This will allow you to reinstall the exact same system at any point in the future, if need be.

Testing

Before you start testing, make sure that all PHP error messages, warnings and notices are displayed. Activate the `php.ini` setting `display_errors` on the test system (see Chapter 8 for more information) and configure PHP to display all errors by setting `error_reporting` to the value 2147483647. This ensures that all PHP errors are displayed, even in future PHP versions.

Now, check whether your application defines a custom error handler (see Chapter 8 for more information). If it does, you should deactivate this error handler by commenting out the `set_error_handler()` call, or make sure that the custom error handler does handle and output every error message. Also, remove all `@-operators` preceding function and method calls from the code, as they suppress display of error messages.

You should consider installing the PHP extension *xdebug*. xdebug displays the call stack along with every error message, and optionally even displays the current values of local and superglobal variables (see Chapter 7 for more information). This can greatly simplify tracking down bugs.

Configure both test systems as described and make sure that the command line PHP configuration does not differ from the PHP configuration in the web server.

Current PHP versions display `E_STRICT` or `E_DEPRECATED` errors when deprecated code constructs are used. These errors do usually not indicate to altered or even undesired program behavior, but point out a future migration problem. If you are short on time, you can ignore the `E_STRICT` errors for now. I would not recommend doing so, though, as you only push the pending problems ahead.

Chapter 8 explains the errors that commonly occur when migrating PHP code. Use the index of this book when you encounter an error message on migration.

Finding Relevant Test Cases

It is impossible to thoroughly test an application. As we have seen in Chapter 2, even a small application has too many possible execution paths to test them all. What's more, some error conditions like lost database connections or a hard disk that ran out of space are hard to simulate.

Focus on testing the application's functionality rather than too many obscure error conditions. Start by creating a list of the application's features. For a typical (simplified) Web 2.0 application, this list could look like this:

- Login
- Logout
- Create user account
- Forgot password
- Edit user profile
- End membership
- Create article
- Edit article

We will have to test all these basic features of the application. If one of these features does not work, the application is not usable. In the context of this example, we refer to article as any piece of content a user generates, be it an article, a blog entry, or a comment.

Most features will require multiple tests. To test the login, for example, you will need to write at least two tests, one to make sure that login with valid credentials work, another one to make sure that login with invalid credentials does not work.

When creating an article, there are several preconditions that must be met (and tested). Let us assume users can only create articles when the administrator has given them the right to do so. We further assume that articles containing a link must be moderated, and that commenting an article must be explicitly enabled by the article's author.

This leads to various scenarios that need to be tested when creating an article:

- A user is allowed to create an article, and creates an article
- A user is not allowed to create an article, and tries to create an article
- A new article without links is displayed without moderation
- A new article with links must be moderated before it is displayed
- A user comments an article that allows comments
- A user tries to comment an article that disallows comments

To fully test the application, we would also have to test any other possible execution path, or at least all the combinations allowed by the application. If, for example, a user is not allowed to create articles, there is no question whether the comment contains links.

In practice, you will probably try to create test cases for the six listed scenarios. Bear in mind, however, that you are leave possible execution paths untested, which could result in an undetected error. Should such an error show up, you should create an additional test case.

Creating Test Data

To test software, test data is required. When testing the login process, at least one user account with a valid password must exist. To comment an article, this article must be present. Ideally, you can test with live data. You should never use the live database to test, though, but create a test database from a dump of the live database.

If the business logic is separated from data access in your application, you can create test data by API calls, and will not have to worry about SQL or the details of your database structure. The advantage of this solution is that you can recreate the test database a needed, even when the database structure has changed.

If you migrate an application without clear separation of business logic and data access, you will have to use the application as a whole to create test data. Consider using Selenium (see Chapter 7) to automate the creation of test data.

Keep in mind that testing with synthetic test data will usually result in different errors than when working with live data. Live data are not always consistent and correct, even when they fulfill all database constraints. An example would be stored form input that has been validated by an older version of the application that might have used less strict validation rules than the current version does.

There are also non-technical reasons why live data cannot be used for testing. Sometimes, privacy concerns make testing with real data impossible. If third parties are involved in the migration, there might also be legal issues with granting them access to real customer data like credit card numbers.

In some cases, the data can be made anonymous, but this is not always easy. If the application uses MX lookups to make sure that email addresses are valid, you cannot use fake email addresses for testing. When the application sends SMS or ringtones, you will probably need valid cell phone numbers, at least for end-to-end testing. Using mock objects (see Chapter 3) does help, but does not solve the legal problems of testing with live data.

Creating Tests

Let as assume our fictitious object-oriented Web 2.0 application has a class `Article` in its business logic class. The controller `CreateArticle` creates an article using input data from a `Request` object.

To test this part of the application, you can either create functional tests or unit tests. Unit tests are usually easier to create and maintain, since they do not need a complex test fixture. Functional tests require the full application with a test database containing sensible test data. This makes functional tests more complex and more difficult to manage.

The following pseudo code shows how a unit test for the test case described above could be implemented:

```
<?php

  function ArticleTest extends PHPUnit_Framework_TestCase
  {
    protected function setUp()
    {
      $this->user = new User('John Doe');
      $this->user->setPassword('*****');
      $this->user->allowComments(true);
    }

    public function testCreateArticleAllowed()
    {
      $this->article = new Article($this->user);
      $this->assertTrue($this->user->has Article($this->article));
    }

    public function testCreateArticleForbidden()
    {
      $this->user->allowComments(false);

      try
      {
        $this->article = new Article($this->user);
        $this->fail('Could create an article though disallowed');
      }

      catch (CannotCreateArticleException $e) {}
    }
  }

?>
```

In this example, we assume that data storage is decoupled from the business logic so that we do not have to deal with storing the created objects.

In the first test, we check whether a user can create an article. The assertion checks whether the newly created article is linked to the user.

In the second test, we disallow creating an article, then try to create one. We expect an exception to occur. Should this not be the case, the test has failed.

Unit tests with PHPUnit are not limited to object-oriented code. Testing a procedural application requires a little more effort to set up and tear down the test environment, but testing basically works in the same way.

The following example shows pseudo code of the first test case for a procedural application:

```
<?php

  function ArticleTest extends PHPUnit_Framework_TestCase
  {
    protected function setUp()
    {
      $this->dbConn = db_connect(...);
      restore_database($this->dbConn, 'default_testdb.sql');
      create_user($db_conn, 'John Doe', '*****', ...);
    }

    protected function tearDown()
    {
      delete_database($this->dbConn);
      unset($this->dbConn);
    }

    public function testCreateArticle()
    {
      $this->assertTrue(create_article($this->dbConn, 'John Doe'));
    }
  }

?>
```

In this example, we assume that the function `create_article()` creates an article in the database. The database connection is passed to the function as a parameter. The function returns *true* when the article has been successfully created.

If you do not want to rely on the return value of `create_article()` for your test, you can actually load the article from the database in `testCreateArticleAllowed()`.

Before the test begins, we create the test environment in the `setUp()` method. By calling `restore_database()`, we recreate the test database from a SQL dump that has been created from an initialized database (see Chapter 3).

When the test is done, the method `tearDown()` is called. This method deletes the test database (or its contents, respectively), so that we can fill it with new test data for the next test that is executed.

You should try to avoid making unit tests dependent from the database. The database does not only make the tests rather slow, but also prevents you from running multiple tests in parallel, except if you use a separate database for each user or test runner.

If the SQL statements your application uses are portable, you can consider using SQLite for testing. SQLite is an embedded database that requires no installation or configuration. By default, SQLite is part of every PHP 5 installation.

Refactoring

If you are lucky, your existing PHP code will work on the target system without any changes. In most cases, however, there are various problems to be solved when migrating to a new environment. Solving these problems usually means making changes to the PHP source code.

If the application contains similar or identical code in multiple places, you will have to repeatedly make the same or very similar changes to the code. Since any code change bears the risk of introducing an error, it is a good idea to try to minimize the number of necessary changes by refactoring the code so that you only need to make each change once.

Refactoring can help you make existing code more readable and easier to maintain. Keep in mind that refactoring is not adding new features. Ideally, automated tests should be in place to make sure that the refactored code still works as expected. In practice, most existing applications do not have automated test, so that you might have to test manually, unless you create automated tests as you refactor.

Eliminate Redundant Code

Redundant code does increase the size of your code base and forces you to make the same changes more than once. If you encounter identical or similar code, create a function or method.

The following example shows how you can refactor redundant code to a function. As always, the example is greatly simplified, and, as it is, would be vulnerable to an SQL-Injection attack:

```
<?php

  [...]

  $q = $db->query('SELECT quantity FROM data WHERE
                item=\'' . $some_item . '\'');
  $quantity = $q->current(SQLITE_NUM);
  $quantity = $quantity[0];

  [...]

  $q = $db->query('SELECT quantity FROM data WHERE
                item=\'' . $_GET['item'] . '\'');
  $db_result = $q->current(SQLITE_NUM);
  $number = $db_result[0];

  [...]

?>
```

Let us create a function that takes the name of the article as parameter:

```
<?php

  function get_quantity($aItem)
  {
    $db = new SQLiteDatabase('report.sqlite');
    $q = $db->query('SELECT quantity FROM data WHERE [...]
    $result = $q->current(SQLITE_NUM);
    return $result[0];
  }

?>
```

Now we can replace the redundant code by a function call:

```
$quantity = get_quantity($_GET['item']);
```

Shorten Code Blocks

Eliminating redundant code is a first important step towards a better structured application. In the second refactoring step, you should shorten code blocks.

There are no fixed rules as to how long one code block should be. As a rule of thumb, one function or method should fit on the screen without scrolling.

Another good rule is that every programmer must be able to understand what a code block does within 30 seconds. If it takes longer than that, the code is too long or not self-explaining.

Separate Different Concerns

Long code blocks usually deal with different concerns. At the application level, presentation, program logic and data storage should be separated. At the function and method level, different functionalities like add, modify and delete should be separated. Separated concerns make programming easier, since the programmer can focus on one single aspect.

The following example from an inventory management system mixes presentation, logic, and data access:

```
<?php

  $q = $db->query('SELECT * from data ORDER BY ' . $order);
  print '<table>';

  print '<tr><td><b><a href="?sort=item">Item</a></b></td><td><b>
         <a href="?sort=barcode">Barcode</a></b></td><td><b>
         <a href="?sort=shelf">Shelf</a></b></td><td><b>
         <a href="?sort=quantity">Quantity</a></b></td></tr>';

  while($q->valid())
  {
    $r = $q->current(SQLITE_ASSOC);

    print '<tr>';
    $first = true;

    foreach ($r as $key => $value)
    {
      if ($first)
      {
        $item = $value;
        $first = false;
      }

      print ’<td>’ . $value . ’</td>’;

      if (’quantity’ == $key) $sum += $value;
    }

    print ’<td> [<a href="?sort=’ . $order . ’&action=add&item=’ .
           $item . ’">+</a>]</td><td>[<a href="?sort=’ . $order .
          ’&action=remove&item=’ . $item . ’">-</a>]</td>’;

    print ’</tr>’;
    $q->next();
  }

  print ’<tr><td></td><td></td><td>Total:</td><td><b>’ . $sum .
        ’</b></td></tr></table>’;

?>
```

To separate the different concerns, we encapsulate the data access in a function:

```
<?php

  function get_data($aSort)
  {
    $db = new SQLiteDatabase(’report.sqlite’);

    $q = $db->query(’SELECT * from data ORDER BY ’ . $aSort);

    $result = array();

    while($q->valid())
    {
      $result[] = $q->current(SQLITE_ASSOC);
      $q->next();
    }

    return $result;
  }

?>
```

The business logic in our example is not really complex, as only the total number of items is calculated. Nevertheless we encapsulate this business logic in another function:

```
<?php

  function get_total()
  {
    $db = new SQLiteDatabase('report.sqlite');

    $q = $db->query('SELECT SUM(quantity) from data');
    $result = $q->current(SQLITE_NUM);

    return $result[0];
  }

?>
```

Now we can modify the original code to use both functions. The remaining code is only presentational, so we have clearly separated all three concerns:

```
<?php

  print '<table>';

  print '<tr><td><b><a href="?sort=item">Item</a></b></td><td><b>
         <a href="?sort=barcode">Barcode</a></b></td><td><b>
         <a href="?sort=shelf">Shelf</a></b></td><td><b>
         <a href="?sort=quantity">Quantity</a></b></td></tr>';

  foreach (get_data() as $r)
  {
    print '<tr>';
    $first = true;

    foreach ($r as $key => $value)
    {
      if ($first)
      {
        $item = $value;
        $first = false;
      }
```

```
   print '<td>' . $value . '</td>';
  }

  print '<td>[<a href="?sort=' . $order . '&action=add&item=' .
      $item . '">+</a>]</td><td>[<a href="?sort=' . $order .
     '&action=remove&item=' . $item . '">-</a>]</td>';

  print '</tr>';
 }

 print '<tr><td></td><td></td><td>Total:</td><td><b>' . get_total() .
    '</b></td></tr></table>';

?>
```

Migrating

Now we are getting serious. Two test systems are available, one a mirror of the existing system, the second one a (possibly simplified) version of the target system. Install your application in both test systems and run it. If you have no automated tests, consider installing Selenium IDE (see Chapter 7), a Firefox plugin allowing you to perform capture and replay testing in Firefox.

Instead of recording one big test case, you should break down the tests into a number of small test cases. You can use PHPUnit to run Selenium tests, and the build automation tool phing (see Chapter 7) to run all tests automatically, whether they are plain unit tests or functional Selenium tests.

In essence, migration is about answering the question, whether the application still works as expected after changes have been made to the code. There are three kinds of possible errors.

The first kind are syntactical errors that prevent PHP from compiling the application's source code to executable code. You should run a lint check (as discussed in Chapter 7) on all PHP files of the application to make sure the PHP code still compiles in the target environment.

The second kind are runtime errors. As we all know, various errors can occur at runtime, even though the code is syntactically correct. An example for a runtime error would be trying to instantiate a non-existing class.

The third kind are errors in the program logic that account for spurious results. Such errors can occur when the behavior of certain PHP constructs has changed between releases. Since they usually do not trigger PHP error messages, they can be hard to find.

The best way of finding these logical errors is to compare the program results of both test systems. When working with floating point calculations, the results may not be exactly the same, but be within a certain bandwidth. Another possibility is to compare trace logs (see chapter 7) created on both systems.

Fixing Existing Bugs

When migrating, the first step should be to fix any existing bugs in the application. This is not absolutely necessary, but a good idea to do, as it makes it easier for you to find any errors related to the migration.

Test the application in the first test system, which represents the existing environment. Write unit tests so that at least the main application functionality is covered by unit tests. Though it can be tempting to write test cases for rare edge cases, you should put the focus on the main functionality.

If your application still runs on PHP 4, you cannot use the current version of PHPUnit for testing. You could use an old version of PHPUnit, at least for simple tests, or use SimpleTest, which also runs on PHP 4.

In addition to the unit tests, you can use Selenium to test the application as whole. Tests that were recorded with Selenium IDE can be replayed with PHPUnit, on either of the test systems. Together with Selenium, you can use PHPUnit on a PHP 5 system to run tests on a PHP 4 system.

Before starting with the actual migration, you should fix all pending bugs and problems in the application. The code should not emit any errors, warnings or notices at all (see Chapter 8). If you plan to replace certain modules of the application, there is of course no need to fix anything there right now.

If you decide not to fix anything now, you should document any existing problems, to make it possible to distinguish between these and problems that are caused by the new environment, or have been introduced while migrating the code.

Replacing Modules

By now you probably have a list of modules that you want to replace with different implementations. Now is the time to replace these modules one by one. If, at any point in the future, you feel that you should replace another module rather than spending time fixing bugs, do so.

When replacing components, the replacements will probably have a different API. You can either modify the code of your application to use the new API, or write an adapter that provides your application with the existing API, and translates these API calls to use the new component internally. I would suggest modifying the application code if there are only few calls to the component's API, and write an adapter if the component is extensively used.

Fixing Syntax Errors

The most obvious error messages when migrating are the fatal errors that occur at compile time. If the code used to work fine in the existing environment, all problems can be attributed to the change of the environment.

As already mentioned, run a lint check to find syntax errors (see Chapter 7 for more information). Depending on the PHP configuration, the lint check will also display `E_STRICT` errors. To automate the lint check, you can use phing and create a fileset comprising of all PHP source files.

Fixing All PHP Error Messages

The PHP source code should compile without errors now. This does not mean, however, that the code does not output any error messages. Since every PHP message indicates a potential problem that could subsequently lead to errors, you should not ignore any of these messages.

There are some exceptions to the rule, though. Some PHP messages are generated in rare edge cases, like the PHP warning that no connection to a database is possible. In this case, preventing this message would probably not be worth the effort. Do not use the `@` operator, though, since it could suppress other and more important error messages as well. In practice, making sure that your code does not omit any PHP messages in the normal case should be sufficient.

Always process messages in the order they occur, since often one problem subsequently leads to more errors. Fixing the original problem will often remove not only one, but multiple PHP messages. Remember to always rerun the tests after modifying the code.

I would recommend adding an assertion to all Selenium tests that checks for any PHP messages in the result HTML. Since you should have configured your test systems to display all messages as part of the HTML output, all PHP errors should always be displayed, thus leading to a failed test.

Fixing Logical Errors

Unfortunately, a program that does not omit any PHP errors, warnings, or notices, does not necessarily work as expected. In various situations, like a regular expression that is too complex, no PHP error message is displayed, though the program will continue with a bogus result.

Compare tests on both test systems. Using Selenium, you can store each HTML page the application has created and compare them, or create a trace log of your Selenium tests and compare these.

You can even consider using the test results of the first test system as expected results for the tests on the second test system. This way, you can not only compare HTML pages, but also serialized computational results, or even database content.

Depending on the application and the data, there might be certain deviations. As already mentioned, results of floating point calculations can differ from system to system. In that case, set a tolerance level of 0.000000001, for example, and ignore any deviations that are smaller. Other examples for allowed differences would be timestamps or random data.

When migrating PHP 4 code that uses references, you should carefully test whether the application's behavior has not changed. Contrary to common belief, references in PHP do not necessarily save memory, but can have the opposite effect. See Chapter 8 for more information.

Normalizing the PHP Configuration

Your application should now work flawlessly on the target system, but still use the PHP configuration of the existing system. If possible, you should now adapt your application so that it runs in a default PHP configuration.

This last migration step is optional, but I would strongly recommend taking the time to go this additional mile. Try to run your application in the `php.ini-recommended` configuration, which should by default be distributed with PHP (at least on Windows). Running an application in the default configuration ensures maximum compatibility to other PHP environments.

You should at least make your application independent from the following `php.ini` settings:

- ze1_compatibility_mode
- allow_call_time_pass_reference
- register_globals
- magic_quotes_gpc
- magic_quotes_runtime
- magic_quotes_sybase
- register_long_arrays
- safe_mode
- short_open_tags

All these settings will be removed in PHP 6, so that you can complete the first step of your migration to PHP 6 today. Refer to Chapter 3 for more information about the individual `php.ini` settings listed here.

Migrating the Production System

When you have gained enough confidence that your application runs flawlessly in the new environment, it is time to put the migrated application into production.

There are different approaches for this. Either you have an additional server that replaces the existing live server, or you have to update the live system, which I would not recommend.

When special or complex hardware is used for the live system, though, it is not possible to buy an identical second system. When you update the live system, make sure you have a fallback position in case the update fails. Before you start, make sure that it is possible to restore the system backup, otherwise you might have to reinstall the system from scratch in case the update fails.

Now you must develop an update procedure. Document each step, ideally by working on a virtual machine with a configuration similar to the existing live system. Store snapshots so that you can rollback to a defined state in case anything goes wrong (see Chapter 7 for more information).

Having updated the test system, test the application in the new environment. If no problems show up, you should repeat the whole update according to the documentation you have created, to make sure the procedure works for you.

Usually, you should not care about performance and security on the test systems, to reduce complexity and thus the number of potential sources of errors. When migrating a live system that works under high load, with a large number of users, or with special security requirements, it may still be necessary to deal with performance and security from day one. Sometimes there are non-technical reasons like organizational or legal rules that require a certain level of security.

To make sure that the application does not only work on the test system, but also on the live system, an integration test is required. To do this, add the required measures to increase security and performance to the second test system and test the application again. When you use Selenium tests, it is particularly easy to just run the tests again. There are some limitations to Selenium tests that are described in Chapter 7.

A very important aspect to focus on in the integration test is the backup and restore process. Without a working backup and restore process, you are sooner or later very likely to run into trouble.

When updating an existing live system, you will have to accept downtime for your application. If you cannot afford any downtime, you will have to use a second server hardware. Another advantage of a second server is that the system on which you run your integration test matches the live system, as opposed to running integration tests on a test system that is similar to the live system. The latter case always bears the risk new problems surface when you go live.

When two systems are available, there are two ways of going live. You can either switch from the old system to the new system at a certain point. This requires transferring the data between systems. To do this, write a migration script and test it thoroughly, before you actually go live.

As we have already seen in Chapter 3, it is not always easy to create a consistent database backup. The same holds true for a consistent migration of all data in a database. If some downtime is acceptable, you can create a consistent database backup, then transfer the data to the new database or convert the existing database accordingly.

If absolutely no downtime is acceptable, you can consider running the old and new version of the application in parallel on the same database instance. Of course, this will only work if you have decided to migrate the database separately from the application itself.

The second possibility of going live with the new system is by gradual migration. In this case, the old and new system run in parallel for a certain time. From a technical viewpoint, is it much more difficult to migrate the data from the old to the new system, but you can migrate without downtime.

Depending on the application, you could consider directing all new customers to the new server, while existing customers still work on the old server. You can even offer to let the user choose when their account and data is migrated to the new server, which makes the system unavailable for some time, but just for this one user.

Sometimes, both application versions are run in sync, which means that all user actions in one application are replayed in the other application. In this case, every application has their own, independent database. The advantage is that you can continue working with just one of the servers at any time. In practice, however, it can be extremely difficult to keep the data kept in both applications in sync.

When migrating web sites, there is a very elegant way of migrating. Keep the existing version untouched and install the new version on a second server. Then, change the DNS records to point to the new system. Until this DNS update has been propagated throughout the Internet, users will either be directed to the old or new system, but they will not experience any downtime. To speed up distribution of the updated DNS records, you can reduce the TTL (time to live) of the DNS records prior to making the DNS update.

This approach, though, does not work in every case. Since you cannot predict which server a user session will be directed to, you must make sure that the data of both servers is in sync when working with sessions.

Finishing the Migration

You are almost done. Your PHP application runs in production in the new environment. Make sure that during the first days, the developers and administrators are always reachable. They can usually fix all the smaller problems that occur in the first days of production rather quickly. Without proper attendance, however, a small problems can turn into a major catastrophe.

If the system is stable and has been running for several days without problems, you can start to relax. Keep in mind, though, that there is always the odd error that just sits there waiting to surface at some point in the future. Depending on the application and the frequency of usage, it can be weeks or even months until this happens. Or, if you are lucky, it will never happen.

One of your most important tasks now is to make sure that the backup actually works. Restore the system from the backup on a separate test system and run a full test of the application. Do not forget to document the restore procedure, so that any of your colleagues can restore the system in case of a catastrophe, even when you are not available.

If you have left out complexity from the system, you can now start adding complexity to the system again. Work in small steps and make one change to the system at a time, otherwise you will not be able to determine the source of a problem. After making a change, allow some time to see whether the system still works flawlessly, and to straighten out potential problems. When you are sure that everything works fine, make the next change.

Chapter 6

After the Migration

“Be not afraid of growing slowly, be afraid only of standing still.” —*(Chinese Proverb)*

What’s Next?

Now that your application has been migrated, and runs smoothly in the new environment, you should allow yourself some time to reflect upon what you have learned, and how to apply this to future projects.

There are some basic principles that you should keep in mind when programming. Doing so will help you avoid common migration problems. Remember that the question is not if another migration is due, but when it is due.

Modular Programming

Regardless of whether you are programming object-oriented, procedural, or hybrid, a modular application is always easier to test. Modular applications also greatly simplify debugging. Changes to the code tend to be local, and do not affect the rest of the application. If the API of a module remains unchanged even though you changed the implementation, you have reached your goal.

Establish Coding Guidelines

The more developers are involved in a project, the more important coding guidelines (and sticking to them) becomes. The goal is to have the code of a project look and feel the same, regardless of which programmer has written it.

Coding guidelines do not only specify how source code should be formatted, but also define naming schemes, disallow certain code constructs, and specify a set of PHP extensions that the project relies on.

To establish coding guidelines, you do not have to reinvent the wheel. You can reuse existing coding guidelines that are available on the Internet[1]. These documents are a good starting point for defining your own rules. You can ask your favorite search engine for "php coding standard" or "php coding guidelines" and will find a cornucopia of material.

Defensive Coding

It is really easy to write a PHP program. Writing a good PHP program that is stable and portable to other PHP systems, is likely to run in future PHP versions, is not always easy. Using many hacks in the code is not a sign of quality, even though some programmers think different. There may be a time and place for innovative and crazy hacks, but they should definitely not appear in a business application. You should always code defensively.

It is better to check once too often whether all input and parameters are present, all objects have the correct class, and data in the database is consistent. Of course, you ultimately have to strike a balance, otherwise your business logic vanishes between all the plausibility and error checks.

Often, programmers argue that their programs are optimized for speed, and thus do not perform extensive input and parameter checking. Since a few lines of PHP code will make the program just a few microseconds slower, you are usually better

[1] Examples of coding standards include Squiz.net: MySource Matrix Coding Standards, 2006, (http://matrix.squiz.net/__data/assets/file/0008/431/coding_standards.pdf), Zend Inc.: Zend Framework PHP Coding Standard, 2007, (http://framework.zend.com/manual/en/coding-standard.html), or PEAR Documentation Group: Coding Standards, 2007, (http://pear.php.net/manual/de/standards.php)

off in having the application run a little slower than spending more time hunting and fixing bugs. Developer time is more expensive than computing time.

Defensive coding also means that you should not use undocumented features. Many migration problems occur because programmers relied on undocumented features. It is not guaranteed that they are still available in future PHP versions, or that PHP's behavior is still the same. Changing the behavior of undocumented features is not bad intent by the PHP developers, but simply a result of the fact that no test exists for that feature, so nobody (except the one relying on the feature) will ever notice that something has changed in a new PHP version.

Do Not Be The First

When you do bleeding edge development, you will always face technical problems that you probably have to solve by yourself. Partially, the open source world exists because users gain (sometimes painful) experience with a product, and then pass their experience on to the community. When the community process works, this experience helps to make the product better, so that everybody benefits.

If you are not forced to do bleeding edge development, though, you should stay behind a little, so that you can benefit from the know-how and experience others have gained. Especially in the open source environment, development processes are very dynamic, so time schedules and feature lists should always be taken with a grain of salt.

For example, namespaces have originally been announced for PHP 5, but did not make it into the final release. PHP 6 has originally been announced for end of 2006, then for end of 2007, then PHP 5.3 took its place, which originally had been announced for the beginning of 2008, but has still not been released at the time of this writing.

Unfortunately, there are many examples of developers that relied on announced features or preview versions, only to realize later that they would have to change their plans. To keep the technical risk of your project low, you should consider to not always use the latest software version, but wait to give others some time to find and fix errors before you encounter them.

Continuous Refactoring

Even the best programmers will not come up with perfect solutions and ideal software architectures in every project. Many design problems can be accounted to the ever-changing requirements, of course, but even in a perfect world, systems and programmers evolve.

If you read code that you have written a while ago, you will often see pieces that could be improved. When refactoring, you change existing code to make it easier to read and maintain, but without changing the visible behavior. By constant refactoring, you can keep your code fresh, improve the design over time and thus adapt the application to a changing environment. Do not wait until the problems accumulate, but fix them in small steps, one at a time.

To make sure that the application still works as expected after making changes to the code, you should use automated tests that you rerun after every code change.

Agile Migration

In recent years, more and more so-called agile methods are used in software development. In contrast to the cumbersome and bureaucratic classical process models, agile methods do not use many rules and formalities. The main difference between agile methods and the V model or waterfall model is that small iterations are used. Long concept, design, implementation and test phases are replaced by short iterations. In each iteration, a relatively small set of features is developed.

This approach can also be used when migrating systems. Instead of letting a system age over time ("never touch a running system"), you should migrate the system in smaller steps, in defined intervals. Just like when you are programming, smaller steps make it much easier to find the source of problems and bugs.

It can be a good idea to not use the exact same PHP version on all development systems. That way, you will quickly find out about pending problems when the application behaves differently on two systems. You will effectively migrate the system while you are developing or maintaining it. This will make the next big migration much less frightening for you.

Chapter 7

Tools

“When building a high tower, spend a lot of time at the foundation.” —*Chinese Proverb*

Tools Make All the Difference

To develop software, good tools are required to be productive and to automate recurring tasks. Using the right tools will greatly increase your productivity. This chapter deals with tools you might find useful when migrating. Though there are usually more than one tool for each task, we will focus on just one tool in this book for reasons of brevity. You may feel that different tools fit into your development process and environment more smoothly. Choose whatever works for you.

Tools are not an end in themselves. You may not need all the tools and techniques we introduce in this chapter, but you should still take the time to read through this chapter to familiarize yourself with the described tools and methods.

Version Control

The most important tool that no developer should ever work without is a version control software. Even when you work on a project as the only developer, version control is absolutely essential. If you already use version control, like most developers, feel free to skip this section.

Those of you that don't use version control should start off tomorrow morning by introducing version control. There are various products, some of them commercial, some free software. If you are not forced to go with a commercial solution for some political reasons, you should choose from one of the free alternatives.

The old bull of version control is CVS, which natively runs on Unix only. Subversion is a newer and better version control software that runs on Unix and Windows alike. Subversion is easy to handle, and was in fact created to be a successor to CVS. Using Subversion is similar to working with CVS, but Subversion does not have most of the main weaknesses of CVS.

Basically, Subversion provides one big versioned directory tree, making it much easier than CVS to rename and move files without losing their history. Subversion has been production-ready for quite some years now, and is successfully being used in many projects. Under `http://subversion.tigris.org/project_packages.html` you can download the Subversion source code or binaries for various operating systems.

One of the main benefits of version control is that you can roll back changes to files at any time, putting your project in a stable and defined state again. By comparing two versions of the same file, you can easily find out about the changes you have made, and undo them selectively if required. Sometimes, it is better to throw away an hour of work in a minute than to spend hours trying to fix a bug one has introduced on the way.

Command Line Tools

An integrated development environment (IDE) or a text editor usually comes with a powerful set of tools to automate repeated tasks when developing software. Sometimes, however, it can be useful to use tools outside the IDE.

Sending HTTP Requests and Downloading Files

The Web is based on the HTTP protocol. All browsers use HTTP to communicate with the web servers, but sometimes it is useful to be able to send a HTTP or HTTPS request without a browser. The command line tool `wget` enables to you do this, which also allows for downloading files from the command line, or starting a program on a web server.

The same effect, of course, can also be achieved by a short PHP script:

```
file_get_contents('http://www.example.com');
```

To make `file_get_contents()` work for remote files, you must activate `allow_url_fopen` in `php.ini`. Since PHP 4.0.5, `file_get_contents()` even supports redirects. The problem with redirects, however, is that the URL of the downloaded document does not match the URL that has originally been requested. To overcome this limitation, you must use a full-blown HTTP client.

`wget` should be available by default on most Unix systems. On Windows, you can download wget from `http://users.ugent.be/˜bpuype/wget`. This version of `wget` also supports HTTPS.

Search Files and Directories

Especially when migrating, you will have to search one or more files for certain strings or regular expressions. The command line tool `grep` allows you to search files and even directory trees. Keep in mind that `grep` might fall into an endless loop when you recursively search directories that contain symlinks to a parent directory.

On Unix, `grep` should be available by default. Windows users can download grep from `http://gnuwin32.sourceforge.net/packages/grep.htm`. You need the two ZIP archives `grep-2.5.1a-2-bin.zip` and `grep-2.5.1a-2-dep.zip`. Unpack both archives in the same directory, and add the bin subdirectory to the system path.

To search files, use

```
> grep -r clone *

index.php:  $a = clone $b;
test.php:  // clone the object
```

This command lists all files in the current directory and its subdirectories containing the word clone. Following each file name, an excerpt from the file is shown.

Replacing in Files

While grep is great for searching, you cannot replace strings in the file. Automated replacing in files is always a little dangerous, as you can easily damage files by replacing too much. Source code may not compile any more after a careless replace operation. Always run a syntax check of the files after you have replaced any strings (for more information, see the section entitled "Syntax Check" later in this chapter).

To replace strings in files at the command line, you can use sed. On Unix, sed should be available by default, while Windows users will have to download it from http://gnuwin32.sourceforge.net/packages/sed.htm. sed does not only support replacing of text, but also understands regular expressions. For complex replace operations, multiple rules can be combined to a sed program. These programs are a bit hard to read, though.

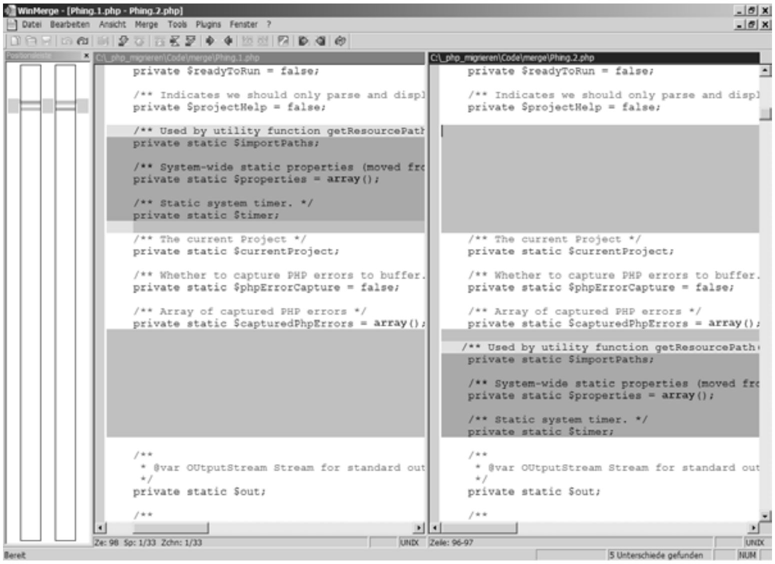
I would therefore recommend using a text editor or the IDE to replace in files, or write a small PHP program. You could even use the PHP function token_get_all() to have the PHP parser tokenize the source code, making it easier to find out where to replace.

Comparing Files and Directories

When migrating a PHP application, you will repeatedly make modifications to the PHP source code. Sometimes, it is useful to be able to visualize the differences between to versions of a file. Every version control software contains a diff program to compare different files. A graphical diff tool is especially handy, since differences between the compared files are color highlighted.

Windows users can use Winmerge, available at http://winmerge.org/downloads/index.php. Winmerge also supports comparing directories, and can be configured to ignore case or whitespace differences. The functionality can be extended by plugins, so that even certain binary file formats can be compared. Optionally, Winmerge also recognizes blocks of code that have been moved. This is very helpful when you have refactored the code.

Unix users can use Meld, a tool with a feature set very similar to Winmerge. Meld is available at http://meld.sourceforge.net and is written in Python, so you do not have to compile the source code, but need Python and Gnome libraries to run it.


WinMerge - [Phing.1.php - Phing.2.php]
Datei Bearbeiten Ansicht Merge Tools Plugins Fenster ?
Positionsleiste
C:_php_migrieren\Code\merge\Phing.1.php
C:_php_migrieren\Code\merge\Phing.2.php
Ze: 98 Sp: 1/33 Zchn: 1/33
UNIX
Zeile: 96-97
UNIX
Bereit
5 Unterschiede gefunden
NUM

```
    private $readyToRun = false;

    /** Indicates we should only parse and displ
    private $projectHelp = false;

    /** Used by utility function getResourcePath
    private static $importPaths;

    /** System-wide static properties (moved frc
    private static $properties = array();

    /** Static system timer. */
    private static $timer;

    /** The current Project */
    private static $currentProject;

    /** Whether to capture PHP errors to buffer.
    private static $phpErrorCapture = false;

    /** Array of captured PHP errors */
    private static $capturedPhpErrors = array();

    /**
     * @var OUtputStream Stream for standard out
     */
    private static $out;

    /**
```

```
    private $readyToRun = false;

    /** Indicates we should only parse and displ
    private $projectHelp = false;

    /** The current Project */
    private static $currentProject;

    /** Whether to capture PHP errors to buffer.
    private static $phpErrorCapture = false;

    /** Array of captured PHP errors */
    private static $capturedPhpErrors = array();

  /** Used by utility function getResourcePath
    private static $importPaths;

    /** System-wide static properties (moved frc
    private static $properties = array();

    /** Static system timer. */
    private static $timer;

    /**
     * @var OUtputStream Stream for standard out
     */
    private static $out;

    /**
```

Figure 7.1

Validating (X)HTML Files

Web pages are written in HTML or XHTML. When a HTML is displayed in a browser, a DOM tree is created from the parsed HTML code. This DOM tree represents the HTML page and can be modified at runtime by Javascript.

There is not only one HTML standard. Even when we ignore really old HTML versions, there are still HTML 3.2, HTML 4.01 which comes in the three variants Strict, Frameset, and Transitional, and the two XHTML versions 1.0 und 1.1. To make the transition from HTML to XHTML easier, XHTML 1.0 also exists in three variants Strict, Frameset, and Transitional, while XHTML 1.1 only exists in Strict.

A basic problem in the early days of HTML was that no means existed for telling the browser which HTML version a given document was written in. In 1997, with HTML 4.0, the so-called Document Type Definition (DTD) has been introduced to tell the browser the HTML or XHTML version of the document.

When rendering HTML, browsers are extremely forgiving instead of presenting the user with an error message. This makes the web very usable, since you can still see a part of its content with older browsers, even when certain features are not available to you.

There is a fundamental difference between HTML and XHTML. Since XHTML is XML, a document must always be valid, otherwise the browser will not be able to parse it. HTML documents can be parsed regardless of how many errors they contain. If a browser encounters invalid XHTML, it will fall back to HTML mode to display the page.

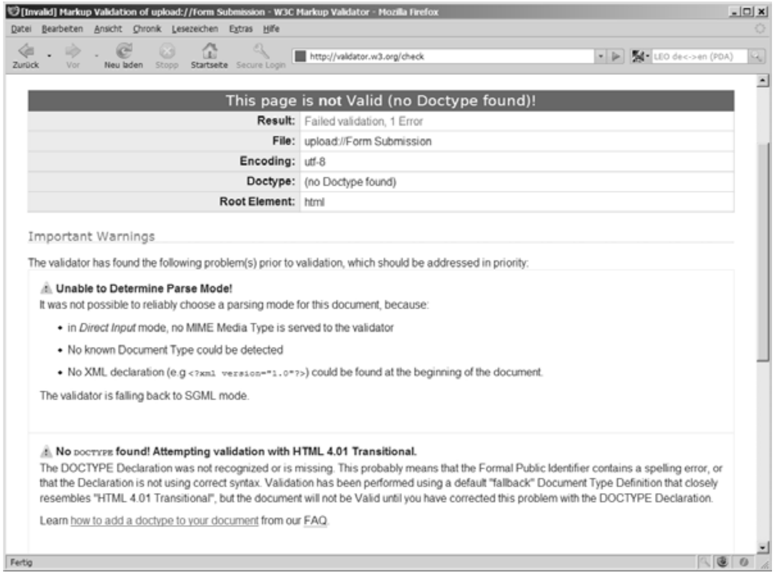
Especially when working with XHTML, you should ensure that every page of your site is valid.

The W3C Validator

The World Wide Web Consortium (W3C), the entity responsible for the HTML standards, offers a free online service to validate (X)HTML pages, available at `http://validator.w3.org`. You can use the validator in three ways. To validate a document that is available on the Internet, specify its URI. For documents in private networks, you can upload the document or paste the source code into a text field.

The document must contain a Document Type Declaration, telling the validator which (X)HTML standard the document is supposed to conform to. The first line of the document should be one of the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.
    org/TR/html4/loose.dtd">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Strict //EN" "http://www.w3.org/TR/
    html4/strict.dtd">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR
    /html4/frameset.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-strict.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.
    org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR
    /xhtml1/DTD/xhtml1-frameset.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/
    DTD/xhtml11.dtd">
```


[Invalid] Markup Validation of upload://Form Submission - W3C Markup Validator - Mozilla Firefox
Datei
Bearbeiten
Ansicht
Chronik
Lesezeichen
Extras
Hilfe
Zurück
Vor
Neu laden
Stopp
Startseite
Secure Login
http://validator.w3.org/check
LEO de<->en (PDA)
Fertig

This page is not Valid (no Doctype found)!

Result:	Failed validation, 1 Error
File:	upload://Form Submission
Encoding:	utf-8
Doctype:	(no Doctype found)
Root Element:	html

Important Warnings

The validator has found the following problem(s) prior to validation, which should be addressed in priority:

⚠ **Unable to Determine Parse Mode!**
It was not possible to reliably choose a parsing mode for this document, because:

- in *Direct Input* mode, no MIME Media Type is served to the validator
- No known Document Type could be detected
- No XML declaration (e.g `<?xml version="1.0"?>`) could be found at the beginning of the document.

The validator is falling back to SGML mode.

⚠ **No DOCTYPE found! Attempting validation with HTML 4.01 Transitional.**
The DOCTYPE Declaration was not recognized or is missing. This probably means that the Formal Public Identifier contains a spelling error, or that the Declaration is not using correct syntax. Validation has been performed using a default "fallback" Document Type Definition that closely resembles "HTML 4.01 Transitional", but the document will not be Valid until you have corrected this problem with the DOCTYPE Declaration.

Learn how to add a doctype to your document from our FAQ.

Figure 7.2

If no DTD is given, the validator will assume a HTML 4.01 document, which does not always yield the desired results. Figure 7.2 shows the result of a validation.

HTML Tidy

The W3C Validator is a very useful tool, but Internet access is required to use it. HTML Tidy as a local tool is an alternative to the W3C validator. HTML Tidy is free software and available for all major operating systems. You can download it from http:*tidy.sourceforge.net/#binaries. Windows users can download a binary distribution from http:*www.paehl.com/open_source/?HTML_Tidy_for_Windows.

Tidy does not only check your (X)HTML files for errors, but also corrects common errors automatically, and optionally even beautifies HTML code by correctly indenting it. This can be particularly useful for PHP programmers, since HTML code that has been generated by PHP is usually not very readable. When migrating, I would advise you to work with indented HTML code, since you will sooner or later have to look at the source code when you try to fix a bug.

Let us first have a look at how you can fix errors in the HTML code with Tidy. Running the following code will output a list of command line options:

```
> tidy -help
```

To display a list of all possible configuration settings, use the following line of code:

```
> tidy -help-config
```

The full list with explanations is also available at the "HTML Tidy Configuration Options Quick Reference" (`http://tidy.sourceforge.net/docs/quickref.html`). Since there are so many configuration settings, you should consider writing a configuration file, instead of adding a long list of command line switches to each and every Tidy call. For a first test, though, we will run Tidy in the default configuration.

Let us create a really buggy HTML page:

```
<html>
<h1>Heading</h2>
<p>First paragraph.
<p><b>Second<b> paragraph</p>
<p>Third paragraph</b></p>
</html>
```

To have Tidy clean up this file, we start Tidy with the file name as parameter and redirect the output to another file:

```
> tidy test.html > tidy_test.html

line 1 column 1 - Warning: missing <!DOCTYPE> declaration
line 2 column 1 - Warning: inserting implicit <body>
line 2 column 1 - Warning: missing </h1> before </h2>
line 2 column 16 - Warning: discarding unexpected </h2>
line 5 column 4 - Warning: <b> is probably intended as </b>
line 6 column 18 - Warning: discarding unexpected </b>
line 2 column 1 - Warning: inserting missing 'title' element
Info: Document content looks like HTML 3.2

7 warnings, 0 errors were found!
```

Since the file has no DTD, Tidy has guessed the HTML standard the file adheres to. Also, Tidy has found and corrected quite some errors. Isn't it amazing how many error such a small file can contain? The result will be something like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<meta name="generator" content=
           "HTML Tidy for Windows (vers 15 August 2007), see www.w3.org">
<title></title>
</head>
<body>
<h1>Heading</h1>
<p>First paragraph.</p>
<p><b>Second</b> paragraph</p>
<p>Third paragraph</p>
</body>
</html>
```

If we run Tidy with the additional command line switch `-indent`, our HTML file would have been indented as well:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
  <head>
    <meta name="generator" content=
          "HTML Tidy for Windows (vers 15 August 2007), see www.w3.org">
    <title></title>
  </head>
  <body>
    <h1>
      Heading
    </h1>
    <p>
      First paragraph.
    </p>
    <p>
      <b>Second</b> paragraph
    </p>
    <p>
      Third paragraph
    </p>
  </body>
```

```
</html>
```

The Tidy PHP extension

Thanks to John Coggeshall, HTML Tidy is also available as a PHP extension. Since PHP 5, that extension is included with PHP by default, if the configuration switch `-with-tidy` has been used. There is also a Tidy extension for PHP 4, using an older Tidy version with less functionality.

On Windows, the Tidy extension is distributed as a DLL that you must enable. This is done by adding the following line of code to `php.ini`:

```
extension=php_tidy.dll
```

Make sure the `extension_dir` setting points to the `ext` subdirectory of your PHP directory, otherwise the DLL will not be found. As always, you will have to restart the web server after modifying `php.ini`.

The Tidy extension adds some new functions to PHP that allows you to check and format HTML code, but also offer you access to Tidy's parse tree. This can be useful to access individual HTML elements when testing.

Tidy can be used together with output buffering. When you register Tidy as a handler for the output buffer, the created HTML code will automatically be cleaned up by Tidy before it is being sent to the browser. Since the browser always receives valid HTML code, this means that less display errors will occur.

```
<?php
  ob_start('ob_tidyhandler');
?>
<p>Sloppy coded <p> illegally nested <b>formatted<b> HTML.
```

The browser will receive:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">
```

```
<html>
<head>
<title></title>
</head>
<body>
<p>Sloppy coded</p>
<p>illegally nested <b>formatted</b> HTML.
</body>
</html>
```

Sometimes, you might not want to modify the HTML code sent to the browser. You can also use Tidy to just report the errors, without modifying the HTML code. To achieve this, turn on output buffering without registering Tidy as a handler:

```
<?php

  ob_start();

?>
```

At the end of your program, add the following piece of code:

```
<?php

  $tidy = new tidy;
  $tidy->parseString(ob_get_contents());

  $tidy->CleanRepair();
  $tidy->diagnose();

  var_dump($tidy->errorBuffer);

?>
```

This program instantiates a Tidy object and passes it the contents of the output buffer. By calling the `CleanRepair()` method, Tidy processes that HTML code. Now we must call the `diagnose()` method so that we can output all error messages and an error count:

```
line 1 column 1 - Warning: missing <!DOCTYPE> declaration
line 1 column 39 - Warning: <b> is probably intended as </b>
line 1 column 1 - Warning: inserting missing 'title' element
Info: Document content looks like HTML 3.2
3 warnings, 0 errors were found!
```

Tidy analyzes the page, but does not modify the output buffer, so the created HTML will still be sent to the browser. Alternatively, you can discard the output buffer and output the HTML code Tidy has generated:

```
<?php

  ob_end_clean();
  print $tidy->value;

?>
```

Validating CSS Files

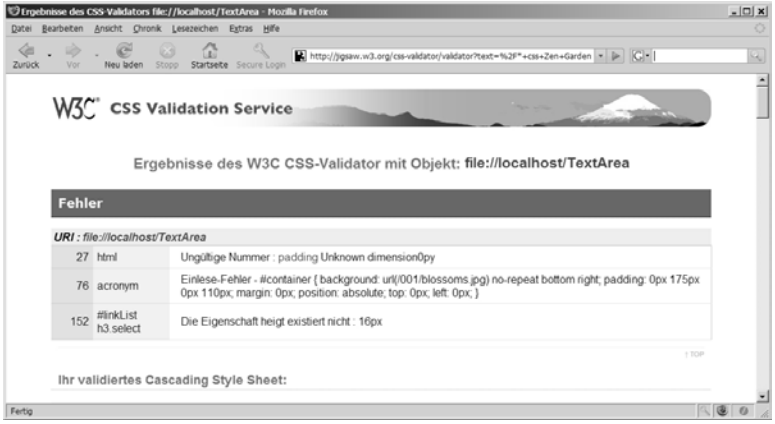
When tracking down display errors, you should make sure that the CSS the page is using is valid. The W3C also provides us with a validator for CSS. This service, *Jigsaw*, ensures that CSS files are syntactically correct. Just like with the HTML Validator, you can specify a URI, upload a CSS file or paste the CSS code into a text field.

Keep in mind that CSS still needs valid (X)HTML for the page to display correctly, so you should always also validate the HTML code.

Figure 7.3 shows the result of a CSS validation. To make the output more interesting, I have put in some errors. As you can see, the validator shows the wrong selectors including the line number with a short description of each error.

Validating XML Files

As previously pointed out, XML files must always be well-formed (which, roughly, means syntactically correct). A well-formed XML document must have one root element, all tags must be correctly nested, and all elements with content must be enclosed into an opening and closing tag, for example.


Ergebnisse des CSS-Validators file://localhost/TextArea - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
Zurück Vor Neu laden Stopp Startseite Secure Login
http://jigsaw.w3.org/css-validator/validator?text=%2F*+css+Zen+Garden
CSS Validation Service
↑ TOP
Fertig

Ergebnisse des W3C CSS-Validator mit Objekt: file://localhost/TextArea

Fehler

URI : file://localhost/TextArea

27	html	Ungültige Nummer : padding Unknown dimension0py
76	acronym	Einlese-Fehler - #container { background: url(/001/blossoms.jpg) no-repeat bottom right; padding: 0px 175px 0px 110px; margin: 0px; position: absolute; top: 0px; left: 0px; }
152	#linkList h3.select	Die Eigenschaft heigt existiert nicht : 16px

Ihr validiertes Cascading Style Sheet:

Figure 7.3

XML parsers cannot process XML files that are not well-formed. Unfortunately, the error messages that XML parsers provide us with are usually hard to understand.

xmllint

The command line tool `xmllint` checks whether XML files are well-formed. Please note that `xmllint` does not check whether a given XML file adheres to a certain DTD. Think of `xmllint` as a syntax checker for XML. `xmllint` is part of `libxml2` and free software under MIT license. You can download `libxml2` from `ftp://xmlsoft.org/libxml2`.

Let us run `xmllint` on a file that contains some errors:

```
> xmllint test.xml
file:///test.xml:11: parser error : Opening and ending tag mismatch:
                                    versions line 11 and version
    <versions>2.0</version>
                           ^
file:///test.xml:43: parser error : Opening and ending tag mismatch:
                                    command line 41 and action
      </action>
               ^
file:///test.xml:50: parser error : expected ’>’
    </actions>
```

```
            ^

file:///test.xml:122: parser error : Opening and ending tag mismatch:
                                     actions line 21 and header
  </header>
           ^
file:///test.xml:226: parser error : Premature end of data in tag header
                                     line 8
</package>
          ^
```

The list of error messages usually looks scary at first sight. Most errors follow-up errors, though, that will disappear when you have fixed the initial error, so make sure you always work your way from down the first error message, and then rerun `xmllint` to see how many errors remain.

PHP

You can also use PHP to check XML whether files are well-formed. To do this, instantiate the built-in class `DOMDocument` and either load an XML document or pass it as a string:

```
<?php

  $doc = new DOMDocument();
  $doc->loadXML(’<root></roots>’);

?>
```

As you can see, that small XML snipped is not well-formed, so PHP will complain:

```
Warning: DOMDocument::loadXML(): expected ’>’
         in Entity, line: 1 in test.php on line 3

Warning: DOMDocument::loadXML(): Extra content at the end of the document
         in Entity, line: 1 in test.php on line 3
```

As you can see, the PHP error message much is less readable than the error messages of `xmllint`.

Static Analysis of JavaScript Files

JavaScript code, especially in AJAX times, is often stitched together from various snippets and included in the HTML page itself, either by PHP or by the programmer. To avoid having to test the whole application in the browser just to find out about a JavaScript error, you can use a JavaScript lint tool.

jsl

JavaScript Lint (jsl) will perform a static code analysis of JavaScript code. This means that the code is being inspected for syntax error and various other problems without executing it. JavaScript Lint is freely available under the Mozilla Public License. You can download it from `http://www.javascriptlint.com`, either in source code or as an executable file for Windows or Mac OS X.

Compiling JavaScript Lint on Unix is very easy:

```
> wget http://www.javascriptlint.com/download/jsl-0.3.0-src.tar.gz
> tar xfz jsl-0.3.0-src.tar.gz
> cd jsl-0.3.0/src
> make -f Makefile.ref
```

After compiling, the executable file `jsl` can be found in the `Linux_All_DBG.OBJ` subdirectory.

Alternatively, you can check your JavaScript code online on the project website. This only works with the JavaScript Lint default configuration, though.

When using JavaScript Lint locally, you should write a configuration file. You can run the following to output a template configuration file:

```
> jsl -help:conf > lint_config
```

Adapt this file to your needs, then run the following:

```
> jsl -conf lint_config -process test.js

JavaScript Lint 0.3.0 (JavaScript-C 1.5 2004-09-24)
Developed by Matthias Miller (http://www.JavaScriptLint.com)
```

```
test.js
test.js(9): lint warning: missing semicolon
    var getZip = function(query) {
 ...^

test.js(28): SyntaxError: syntax error
        oACDS.queryMatchContains = ;
 ................................^

1 error(s), 1 warning(s)
```

As you can see, JavaScript Lint detects common errors in JavaScript code. This includes a missing semicolon, missing or illegally nested curly braces, nested comments, leading zeroes for numbers, which would cause them to be interpreted as octal numbers, or a missing break statement in a case switch.

To run a lint check on JavaScript code generated by PHP, you can store the generated HTML, then remove everything except the Javascript part, and run JavaScript Lint on the file.

JSLint

An alternative lint tool for JavaScript, JSLint by Douglas Crockford, is available at `http://www.jslint.com/lint.html`. This tool also offers online check right at the web site, but you can also run JSLint locally. Since JSLint is written in JavaScript itself, you need the Java-based JavaScript engine Rhino, which is free software developed by the Mozilla foundation. You can download it from `http://www.mozilla.org/rhino/download.html`.

Having downloaded Rhino, unpack the archive. To run Rhino, you must call Java. Then, run JSLint inside Rhino. Last but not least, you will have to specify the file name of the JavaScript file to check, resulting in the following command line:

```
> java -jar js.jar jslint.js test.js

Lint at line 7 character 23: Missing semicolon.
myDataTable = null

Lint at line 28 character 36: Expected an identifier and instead saw ';'.
oACDS.queryMatchContains = ;
```

```
Lint at line 28 character 36: Stopping, unable to continue. (34% scanned).
```

Firefox Extensions

Firefox should be the first choice browser for developers today. It is available on all major platforms, and many interesting plugins are available for it. Some extensions should not be missing on any development system.

Webdeveloper

The Webdeveloper extension by Chris Pederick is the Swiss army knife for web developers. The easiest way to install a Firefox extension is to download and save it locally. To do this, right-click the Download link on `http://chrispederick.com/work/web-developer`. The file `web-developer.xpi` will be downloaded. Most `xpi` files work across platforms. To install the extension, just drag and drop the `xpi` file into your browser window.

The Webdeveloper extension will provide you with a toolbar and a context menu. In the *Disable* menu, for example, you can disable certain browser features like Java and JavaScript. In the *Cookies* menu, you can view, edit, delete and even create cookies. In the *CSS* menu you can selectively deactivate CSS style sheets, change the media type of add a custom style sheet. The function *View Style Information* is very useful to display all active CSS selectors for an element of the page that you select by just clicking on it.

The *Form* menu allows you to modify HTML forms, which includes making password fields readable, filling in deactivated form fields, or removing the length restrictions on fields. To make testing easier, you can convert `GET` requests to `POST` requests and vice versa in the *Convert Form Methods* menu.

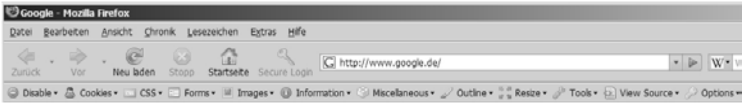


Figure 7.4

In the *Images* menu, you can suppress display of images, view image information like file size, or images size in pixels. The function *Find Broken Images* is really useful. It makes Webdeveloper automatically check whether all images the page links to exist.

The *Information* menu allows you to display additional information about page elements by tool tips, for example the HTML ID of elements, their CSS class, anchors, link targets, title attributes, keyboard shortcuts, or even the size of each displayed block.

The *Tools* menu allows you to validate HTML and CSS, and all page links. Webdevloper uses the online validators described earlier in this chapter for this.

Firebug

The Firefox extension Firebug by Joe Hewitt makes debugging JavaScript code almost fun. To install this plugin, download it from `https://addons.mozilla.org/de/firefox/addon/1843`, then drag and drop it into your browser window.

Since Firebug slows down your browser quite a bit, it is deactivated by default. You can enable Firebug by clicking the round gray icon as shown in Figure 7.5. When you right-click this icon, a context menu will appear that allows you to define certain URLs Firebug should be activated for. This way, you can use Firebox for development and testing without it getting in your way when you are surfing the Internet.

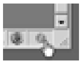


Figure 7.5

Firebug displays the HTML source code of the current page in a tree view, allowing you to expand and collapse block as required. The same holds true for the DOM tree that the browser creates from the HTML source code. In the CSS stylesheet display, you can selectively enable and disable individual rules.

When you click *Inspect* and move the mouse pointer over a page element, Firebug will display the relevant portion of the HTML source code, the relevant CSS rules, additional layout information and the node in the DOM tree.

By clicking on *Edit*, you can edit the HTML and CSS code in place and will directly view the result in the browser window. On the *Console* tab, you can click *Profile* to activate the JavaScript profiler, which will provide you with statistics about the computing time the JavaScript calls on the page have required.

In the *Option* menu on the right, you can enable JavaScript warnings, CSS warnings, XML errors and even XMLHTTP requests and their server responses. In the *Net* tab below the menu bar, you will see an overview of the load times of each page element. Elements that have been loaded from the browser cache will be displayed in a different color.

PHP's Own Means

Since you can not only run PHP in the web server, but also at the command line, PHP is also well suited as a platform-independent script language to automate tasks.

The PHP Configuration

Configuring PHP is a far more complex issue than you might think at first glance. Besides the fact that the configuration file is being searched for in various locations in the file system (see Chapter 3), some PHP configuration settings can also be changed at runtime and in other configuration files like `httpd.conf` or `.htaccess`. This allows you to adapt the PHP configuration to virtual hosts, applications or even individual directories.

It is therefore not easy to predict the PHP configuration a certain script will run in just by looking at the configuration files. Especially when migrating this is very important, because many migration problems are actually caused by differences in the PHP configuration.

When your application has only one entry point, for example `index.php`, the configuration will be the same throughout the application. If there is more than one entry point, the application might run in different configurations depending on the entry point. This can be a source of problems when testing the code.

To illustrate this, let us imagine the two files `index.php` and `.htaccess` in the root directory, and the files `sub.php` and another `.htaccess` file in the subdirectory inc:

```
.htaccess
index.php
inc/.htaccess
inc/sub.php
```

As long as sub.php is being included from index.php, the configuration settings in .htaccess in the root directory will be in effect. If you call inc/sub.php directly, the configuration settings in inc/.htaccess will be in effect.

Many PHP applications rely on .htaccess configuration, and contain .htaccess files or create them during the installation process. Quite often, the Apache module mod_rewrite is used to map search engine-friendly URLs to the actual URLs the application uses.

Users without administrative rights can sometimes only use .htaccess files to modify the PHP configuration, as they have no write access to php.ini and cannot restart the web server.

Keep in mind that you will see the CLI configuration when you execute php with the command line switch -i. If you use the browser to access a PHP script calling phpinfo(), you will see how PHP is configured when running in the web server. This configuration can differ from the command line configuration.

The easiest way of finding out about the current PHP configuration is to list the configuration at script runtime. You can use the built-in functions ini_get() and ini_get_all() to retrieve one or all configuration settings. Please note that the function get_cfg_var() will always return the configuration value that is configured in php.ini.

ini_get_all() returns a large array like shown below:

```
array(230) {
  ["allow_call_time_pass_reference"] => array(3)
  {
    ["global_value"] => string(0) ""
    ["local_value"]  => string(0) ""
    ["access"]       => int(6)
  }

  ["allow_url_fopen"] => array(3)
  {
    ["global_value"] => string(1) "1"
    ["local_value"]  => string(1) "1"
    ["access"]       => int(4)
  }

  [...]
```

As the `ini_get_all()` output shows, combinations of these values are also possible. The value 6, for example, is the binary combination of the values 2 and 4. This means that the respective setting can be changed per directory and system-wide.

This array also holds information about where individual settings can be changed. Table 7.1 shows where the configuration settings can be changed:

Constant	Value	Remark
PHP_INI_USER	1	Can be modified with ini_set() at runtime
PHP_INI_PERDIR	2	Can be modified in php.ini, .htaccess, or httpd.conf
PHP_INI_SYSTEM	4	Can only be set in php.ini and httpd.conf
PHP_INI_ALL	7	Can be modified everywhere

On Windows, all `PHP_INI_USER` configuration settings can also be configured through the registry. Since this is not possible on Unix, I would not recommend putting PHP configuration settings into the registry.

Whenever possible, you should put all configuration settings into `php.ini` or `httpd.conf`. It is also possible to make settings for a virtual host or a directory in `httpd.conf`, so you do not necessarily have to use `.htaccess` files.

Syntax Check

Until PHP 5.0.5, PHP had a built-in function `php_check_syntax()`, which would check the syntax of the PHP source code passed in as a string. This function has been removed from PHP and replaced by a command line switch, making it easier for IDEs to run syntax checks on PHP files.

By calling PHP with the `-l` switch, you can run a syntax check:

```
> php -l test.php

No syntax errors detected test.php
```

This so-called lint check will not execute the code, but only check for syntax errors. This means that only errors that occur at compile time can be detected. If you have set `error_reporting` accordingly in `php.ini`, `E_STRICT` errors will also be displayed by the lint check.

A lint check will not output any PHP runtime errors or warnings, obviously. This makes the lint check a good initial quality assurance measure, but it does not replace a full test of the file.

To display `E_STRICT` errors when syntax checking the code, you can also use an additional command line switch:

```
> php -d error_reporting=8191 -l test.php
```

See the section titled "New And Modified Error Messages" in Chapter 8 for an explanation of the value *8191*.

Together with a build automation tool like `phing`, lint checks can be fully automated. See later in this chapter for more information.

Prepend and Append Files

The two `php.ini` settings `auto_prepend_file` and `auto_append_file` specify files that PHP will automatically run prior to or after running the requested file. Make sure whether your application relies on `prepend` or `append` files, as you will have to make them available on the target system as well.

You must use the full, absolute path to your prepend and append files in `php.ini`, otherwise PHP will try to find them relatively to the current working directory, which may not work.

```
auto_prepend_file = c:/www/prepend.php
auto_append_file = c:/www/append.php
```

If one of the two files does not exist, PHP will output a rather cryptic warning:

```
Warning: Unknown: failed to open stream: No such file or directory in Unknown on
     line 0
```

```
Fatal error: Unknown: Failed opening required 'prepend.php' (include_path='.;C:\
    php\pear') in Unknown on line 0
```

It is not easy to relate this error to a problem with prepend and append files, since `line 0` is mentioned. To make this error message easier to understand, you should name the prepend and append files as shown in the example.

I would not recommend relying on prepend and append files, but rather include the files explicitly when required. This documents the dependency in the source code. When migrating, however, prepend and append files can sometimes be useful. You can use them to perform additional tasks, without modifying the source code of your application. Some useful additional tasks can be logging of activities and errors, activating a certain PHP configuration, or forcing the application to use a certain error or exception handler.

PEAR Components

When PHP 4 was still in development, Stig Bakken founded the PEAR (PHP Extension and Application Repository) project inspired by the CPAN (Comprehensive Perl Archive Network). PEAR is primarily a collection of open source software components for PHP, but also a great infrastructure for installing and managing components, the so-called PEAR packages. Since version 1.4, which was released in 2005, PEAR supports a channel concept that enables everybody to set up a PEAR server. When installing a package, PEAR can automatically resolve the dependencies by downloading and installing any additional required packages.

On Linux, PEAR should be part of every PHP installation, unless PHP has been configured with the `-without-pear` option. On Windows, PEAR is usually not installed, but must be installed manually. This is rather easy, because PEAR is being deployed as a self-extracting PHP archive.

To find out whether PEAR is already installed on your system, run the following command at the command line:

```
> pear version

PEAR Version: 1.6.0
PHP Version: 5.2.3
```

```
Zend Engine Version: 2.2.0
Running on: Windows NT MOPANI 5.1 build 2600
```

If you see an error message instead of output similar to the one printed above, either PEAR is not installed or not configured properly, so that the `pear` command is not in the system's search path.

To install PEAR, download `go-pear.phar` from `http://pear.php.net`. Probably, the source code of the file will be displayed in the browser, so you must choose *Save As* to save the file locally. Please note that Internet Explorer rewrites any HTML code when saving it, making the PHP program unusable, so either a decent browser or `wget` to download `go-pear.phar` (as discussed earlier in this chapter).

On Windows, copy `go-pear.phar` to the directory containing the `php.exe` file and add this directory to the system search path. To install PEAR, just execute the Phar archive with PHP:

```
> php go-pear.phar

Are you installing a system-wide PEAR or a local copy?
(system|local) [system] :

Below is a suggested file layout for your new PEAR installation.  To change
     individual locations, type the number in front of the directory.  Type 'all'
      to change all of them or simply press Enter to accept these locations.

 1. Installation base ($prefix)        : C:\php
 2. Temporary directory for processing : C:\php\tmp
 3. Temporary directory for downloads  : C:\php\tmp
 4. Binaries directory                 : C:\php
 5. PHP code directory ($php_dir)      : C:\php\pear
 6. Documentation directory            : C:\php\pear\docs
 7. Data directory                     : C:\php\pear\data
 8. Tests directory                    : C:\php\pear\tests
 9. Name of configuration file         : C:\WINDOWS\pear.ini
10. Path to CLI php.exe                : C:\php
```

The installer will ask some questions, but usually you can just accept the suggested default values. The PEAR installer will modify `php.ini` and add the PEAR installation directory to PHP's `include_path`.

Having completed the installation, you can run the following to list all installed packages:

```
> pear list

INSTALLED PACKAGES, CHANNEL PEAR.PHP.NET:
=========================================
PACKAGE          VERSION STATE
Archive_Tar      1.3.2   stable
Console_Getopt   1.2.3   stable
PEAR             1.6.1   stable
Structures_Graph 1.0.2   stable
```

If PEAR is already installed on your system, or you want to update the installation later, just run the following:

```
> pear upgrade PEAR

WARNING: channel "pear.php.net" has updated its protocols, use "channel-update
    pear.php.net" to update
Did not download optional dependencies: pear/XML_RPC, use --alldeps to download
    automatically
pear/PEAR can optionally use package "pear/XML_RPC" (version >= 1.4.0)
downloading PEAR-1.6.2.tgz ...
Starting to download PEAR-1.6.2.tgz (297,794 bytes)
 .................................................done: 297,794 bytes

upgrade ok: channel://pear.php.net/PEAR-1.6.2
```

After updating PEAR, you should also update the channel protocols. This is mandatory when you were updating from a version earlier than 1.4.0:

```
> pear channel-update pear.php.net

Updating channel "pear.php.net"
Update of Channel "pear.php.net" succeeded
```

PHP_Compat

PHP_Compat is a PEAR package that provides you with PHP-level implementations of new built-in PHP functions and constants. Using PHP_Compat, you can use the new functions available in a certain PHP version in an older PHP version.

This package is no replacement for a migration, but can help you prepare the code for a migration, or ensure backwards compatibility of code that otherwise requires a newer PHP version. Please note that PHP_Compat does not provide you with any object-oriented features.

Installing PHP_Compat is very easy, since there are no dependencies to other PEAR packages:

```
> pear install PHP_Compat

downloading PHP_Compat-1.5.0.tgz ...
Starting to download PHP_Compat-1.5.0.tgz (44,133 bytes)
 ...........done: 44,133 bytes

install ok: channel://pear.php.net/PHP_Compat-1.5.0
```

The easiest way to use PHP_Compat is to load all functions and constants that are available in a certain PHP version:

```
<?php

  require_once 'PHP/Compat.php';

  PHP_Compat::loadVersion('5.0.0');

?>
```

Alternatively, you can load individual functions:

```
<?php

  require_once 'PHP/Compat.php';

  PHP_Compat::loadFunction('file_get_contents');

?>
```

PHP_Beautifier

One of the biggest annoyances when programming is source code that is not consistently formatted and indented. This makes the code hard to read and understand. Any development project should set and enforce strict coding guidelines that regulate indentation, where to put whitespace, and also define a consistent naming scheme. The goal is to write self-explanatory code that looks the same, regardless of which team member has actually written the code.

Fortunately, we do not have to reformat existing source code manually, but can have the computer do the job for us. The PEAR package PHP_Beautifier automatically formats source code, supporting various styles that can be customized. You can also create your own style if necessary.

PHP_Beautifier is currently available in version 0.1.13 and still a beta release, though the package has been under development for three years now. This version number should not keep you from using PHP_Beautifier, though. The development continues, albeit slowly.

PHP_Beautifier requires PHP 5 to run, but can format PHP 4 and PHP 5 source code. Installation is simple, but requires to additional command line switches because there are dependencies to other PEAR packages, and the release is not yet marked as stable. To automatically install all dependencies, we need the `-alldeps` switch, while the `-force` switch is required to install a beta version:

```
> pear install --alldeps --force PHP_Beautifier

WARNING: failed to download pear.php.net/PHP_Beautifier within preferred state "
     stable", will instead download version 0.1.13, stability "beta"
downloading PHP_Beautifier-0.1.13.tgz ...
Starting to download PHP_Beautifier-0.1.13.tgz (47,528 bytes)
 ............done: 47,528 bytes

install ok: channel://pear.php.net/PHP_Beautifier-0.1.13
```

To start the application, call `php_beautifier`, which is a batch file on Windows and a shell script on Unix. To see a list of all supported options, run the following code:

```
> php_beautifier --help
seine Liste der üuntersttzen Schalter und Optionen ausgeben:
```

```
Usage: php_beautifier [options] <infile> <out>
         <infile> and/or <out> can be ’-’, which means stdin/stdout.
         you can use ? and ∗ for batch processing
         <out> can be a dir (ending with ’/’ or a real dir)
               or a file (without ’/’)
         multiple ins and one out = merge all files in one output

Options:
     --input             or -f <file>    input file - default: stdin
     --output            or -o <out>     output dir or file
                                         - default: stdout
     --indent_tabs       or -t <int>     indent with tabs
     --indent_spaces     or -s <int>     indent with spaces - default
     --filters           or -l <fil_def> Add filter(s)
     --directory_filters or -d <path>    Include dirs for filters
     --compress          or -c <type>    Compress output
     --recursive         or -r           Search in subdir recursively
     --help              or -?           display help/usage
                                         (this message)

Filter definition:
     --filters "Filter1(setting1=value1,setting2=value2) Filter2()"
```

In PHP_Beautifier, source code is formatted through filters. You can parameterize and chain filters, or write custom filters. Existing filters are the `ArrayNested` filter that indents multi-dimensional arrays, `IndentStyles` that supports various indentation styles, `Lowercase` to convert all control structures to lower case, and `NewLines` that allows you to insert new lines into the source at certain points.

For fully automated formatting, there are the `Default` and `Pear` filter. While `Default` indents source code in C’s Kernighan and Ritchie style (also called K&R style), the `Pear` filter will format source code to adhere to the PEAR coding guidelines.

Since PHP_Beautifier will modify your source code, you should always keep a backup of the original files when beautifying the code, just in case something goes wrong. Having formatted the source code, you should run at least a lint check (as discussed earlier in this chapter) to make sure no errors were introduced.

To test PHP_Beautifier, we run it on a short piece of code that is really, really unformatted:

```
<?php
```

```
 function test ($aParameter)
{
  $a = 4;
    $b=5;

    if ($a == $aParameter)
  {
     var_dump('Hello world'); }

   }

?>
```

To format the source code, we call PHP_Beautifier at the command line and pass the name of the file for beautify and the name of the target file as parameters:

```
> php_beautifier ugly.php clean.php
```

The result is:

```
<?php
function test($aParameter) {
    $a = 4;
    $b = 5;
    if ($a == $aParameter) {
        var_dump('Hello world');
    }
}
?>
```

By default, PHP_Beautifier will use four blanks to indent code and put the opening curly braces on the same line, rather than the next line. You can use additional filters to modify the result. Let us use the IndentStyles filter to have the opening curly braces on the next line:

```
> php_beautifier --filters "IndentStyles(style=bsd)" ugly.php clean.php
```

Now the result is source code formatted in the BSD style rather than the K&R style:

```
<?php
function test($aParameter)
{
    $a = 4;
    $b = 5;
    if ($a == $aParameter)
    {
        var_dump('Hello world');
    }
}
?>
```

To improve readability, you can have the NewLines filter insert new lines before or after certain tokens. For a full list of tokens, see [PHP 2008-3]:

```
"NewLines(before=T_CLOSE_TAG:T_CLASS,after=T_OPEN_TAG)"
```

PHP_CodeSniffer

PHP_CodeSniffer is a PEAR package that analyzes PHP source code and detects violations of a given coding style. As we already mentioned above, every project should have coding guidelines, and PHP_CodeSniffer is the tool that helps you enforce these guidelines.

Among the checks PHP_CodeSniffer can perform are:

- Is the indentation correct?
- Are the line endings consistent?
- Are all constants in upper case?
- Are all method names in camel case?

PHP_CodeSniffer requires at least PHP 5.1 to run, but can process code written for older PHP versions. While PHP_Beautifier modifies the existing source code, PHP_CodeSniffer just outputs a list of warnings and errors, and expects the programmer to fix them.

Again, installation is very simple:

```
> pear install PHP_CodeSniffer

downloading PHP_CodeSniffer-1.0.0.tgz ...
Starting to download PHP_CodeSniffer-1.0.0.tgz (203,616 bytes)
 ........................................done: 203,616 bytes

install ok: channel://pear.php.net/PHP_CodeSniffer-1.0.0
```

To start PHP_CodeSniffer, run `phpcs`, which is a batch file on Windows and a shell script on Unix. Using the `-help` switch, you can output a list of supported command line options:

```
> phpcs --help

Usage: phpcs [-nwlvi] [--report=<report>] [--standard=<standard>]
    [--config-set key value] [--config-delete key] [--config-show]
    [--generator=<generator>] [--extensions=<extensions>]
    [--ignore=<patterns>] [--tab-width=<width>] <file> ...

        -n            Do not print warnings
        -w            Print both warnings and errors (on by default)
        -l            Local directory only, no recursion
        -v[v][v]      Print verbose output
        -i            Show a list of installed coding standards
        --help        Print this help message
        --version     Print version information
        <file>        One or more files and/or directories to check
        <extensions>  A comma separated list of file extensions to check
                      (only valid if checking a directory)
        <patterns>    A comma separated list of patterns that are used
                      to ignore directories and files
        <standard>    The name of the coding standard to use
        <width>       The number of spaces each tab represents
        <generator>   The name of a doc generator to use
                      (forces doc generation instead of checking)
        <report>      Print either the "full", "xml", "checkstyle",
                      "csv" or "summary" report
                      (the "full" report is printed by default)
```

PHP_CodeSniffer can process a file, or a while directory including its subdirectories. You can use the `-extensions` switch to specify the file extensions to analyze. To exclude files or directories from the check, you can specify a pattern in the `-ignore` switch. This is very useful to ignore `.svn` or `.cvs` subdirectories, for example.

Running `phpcs -i` will show a list of all available coding standards. Currently, there are MySource, PEAR, PHPCS, Squiz, and Zend. By default, PHP_CodeSniffer uses the PEAT standard that puts a strict focus on source code format.

The MySpace and Squiz coding standards are very strict and usually complain quite a lot. They will recognize commented out code, and enforce type safe comparisons, for example. When functions in the global namespace are found, PHP_CodeSniffer will suggest to move them to a static class.

Let us run PHP_CodeSniffer on an example program that we coded deliberately sloppy, so that we see some output. Let us assume that the file `test.php` is located in the `src` subdirectory:

```
> phpcs --standard=Squiz src

FILE: src\test.php
--------------------------------------------------------------------------
FOUND 33 ERROR(S) AND 2 WARNING(S) AFFECTING 1091 LINE(S)
--------------------------------------------------------------------------
    1 | ERROR   | End of line character is invalid; expected "\n" but
                  found "\r\n"
    2 | ERROR   | You must use "/**" style comments for a file comment
   20 | ERROR   | Expected "if (...) {\n"; found "if (...)\n{\n"
   22 | ERROR   | Line indented incorrectly; expected at least 4 spaces,
                  found 3
   27 | WARNING | Line exceeds 85 characters; contains 147 characters
   38 | ERROR   | Missing class doc comment
   61 | ERROR   | You must use "/**" style comments for a function comment
   77 | ERROR   | Expected "if (...) {\n"; found "if (...)\n      {\n"
   91 | WARNING | Equals sign not aligned with surrounding assignments;
                  expected 4 spaces but found 1 space
  174 | ERROR   | Method name "RSS::tag_open" is not in camel caps format
  189 | ERROR   | Break statement indented incorrectly; expected 13
                  spaces, found 12
  198 | ERROR   | Closing brace must be on a line by itself
  228 | ERROR   | Expected 1 space after comma in function call; 2 found
  270 | ERROR   | @see tag comment indented incorrectly.
                  Expected 4 spaces but found 1.
  271 | ERROR   | Missing @package tag in class comment
  291 | ERROR   | Parameters must appear immediately after the comment
  368 | ERROR   | Doc comment var "error" does not match actual variable
                  name "$error_code" at position 1
```

```
 398 | ERROR   | Expected 1 space before variable type
 443 | ERROR   | The variable names for parameters $message (1) and
                 $code (2) do not align
 529 | ERROR   | Missing comment for param "$message" at position 1
 598 | ERROR   | Function name "XML_RPC_entity_decode" is invalid;
                 consider "XML_RPC_Entity_decode" instead
 599 | ERROR   | Space after opening parenthesis of function call
                 prohibited
 604 | ERROR   | Function name "file_get_contents" is prefixed with a
                 package name but does not begin with a capital letter
 606 | ERROR   | Perl-style comments are not allowed. Use "// Comment."
                 or "/* comment */" instead.
 620 | ERROR   | Constants must be uppercase; expected
                 'PLUGIN_20C' but found 'PLUGIN_20c'
 623 | ERROR   | Class name must begin with a capital letter
 653 | ERROR   | "include" is a statement, not a function;
                 no parentheses are required
 668 | ERROR   | File is being unconditionally included;
                 use "require" instead
 721 | ERROR   | Expected "} elseif (...) {\n"; found
                 "}\n    elseif (...) {\n"
 723 | ERROR   | Space before closing parenthesis of
                 function call prohibited
 793 | ERROR   | Space found before comma in function call
 805 | ERROR   | Closing brace indented incorrectly;
                 expected 0 spaces, found 1
 875 | ERROR   | Expected "foreach (...) {\n"; found "foreach(...) {\n"
 882 | ERROR   | Missing @return tag in function comment
 913 | ERROR   | Method name "plugin_api::remove_plugin_instance"
                 is not in camel caps format
------------------------------------------------------------------------
```

If you want to focus less on the formatting and indentation aspect, I would recommend running PHP_Beautifier first, then PHP_CodeSniffer. Since PHP_CodeSniffer does not check for syntax errors, you should run a lint check (as discussed earlier in this chapter) before you run the sniffer.

PHP_CodeSniffer can be extremely helpful when migrating code, since you can write your own coding standard that will point out potentially problematic code.

PHP_CompatInfo

The PEAR package PHP_CompatInfo allows you to find out which PHP version and which PHP extensions are required for by a certain application.

Like all other PEAR packages, installation is simple:

```
> pear install PHP_CompatInfo

downloading PHP_CompatInfo-1.5.1.tgz ...
Starting to download PHP_CompatInfo-1.5.1.tgz (122,434 bytes)
 ..........................done: 122,434 bytes

install ok: channel://pear.php.net/PHP_CompatInfo-1.5.1
```

To start PHP_CompatInfo, run `pci`, which is a batch file on Windows and a shell script on Unix. Again, using the `-help` switch, you can output a list of supported command line options:

```
> pci --help

Usage: pci.php [options]

  -d  --dir (optional)value                Parse DIR to get its
                                           compatibility info ()
  -f  --file (optional)value               Parse FILE to get its
                                           compatibility info ()
  -v  --verbose (optional)value            Set the verbose level (1)
  -n  --no-recurse                         Do not recursively parse files
                                           when using --dir
  -if --ignore-files (optional)value       Data file name which contains
                                           a list of file to ignore
                                           (files.txt)
  -id --ignore-dirs (optional)value        Data file name which contains
                                           a list of directory to ignore
                                           (dirs.txt)
  -in --ignore-functions (optional)value   Data file name which contains
                                           a list of php function to
                                           ignore (functions.txt)
  -ic --ignore-constants (optional)value   Data file name which contains
                                           a list of php constant to
                                           ignore (constants.txt)
  -ie --ignore-extensions (optional)value  Data file name which contains
                                           a list of php extension to
```

```
                                          ignore (extensions.txt)
-iv --ignore-versions values(optional)   PHP versions - functions to
                                          exclude when parsing source
                                          code (5.0.0)
-h  --help                                Show this help
```

PHP_CompatInfo works well in the default configuration, though there are many configuration options. Most of the options should only be required if you work with rarely used PHP extensions. Let us try out PHP_CompatInfo:

```
> php compat.php -d test/src

+----------------+---------+------------+------------------+
| Path           | Version | Extensions | Constants/Tokens |
+----------------+---------+------------+------------------+
| [...]\*        | 5.1.0   | mysql      | protected        |
|                |         | pcre       | public           |
|                |         | zlib       | interface        |
|                |         | mcrypt     |                  |
+----------------+---------+------------+------------------+
```

As we can see, this application requires at least PHP version 5.1, because the tokens `protected`, `public`, and `interface` were found in the source code. Based on the function calls found in the code, PHP_CompatInfo has determined that the PHP extensions `mysql`, `pcre`, `zlib` and `mcrypt` are required.

Please note that when the PHP source code contains syntax errors, PHP_CompatInfo will deliver bogus results. I recommend running a lint check before running PHP_CompatInfo (as discussed earlier in this chapter). Also, it is important to know that all extensions that are enabled by default in PHP will not be listed as required. Among these are the Mail extension, or SPL, for example.

PHP_CompatInfo does not take object-oriented code into account. When a program makes use of the XMLReader extension, for example, by instantiating an object with `new XMLReader()`, XMLReader will not be listed as a dependency. If your application uses the object-oriented API of one or more PHP extensions, you cannot rely on the output of PHP_CompatInfo any more.

Virtual Machines

One of the big trends in computer science these days is virtualization. Virtual systems do not run on real hardware, which makes them more flexible, because they are independent from the actual host system used. And, of course, to install virtual machines you do not need to buy additional hardware, but can create your own test lab by installing a set of virtual machines that you run as needed, on just one or a few host systems.

Various virtualization products are available today, some of them are free and open source software, some of them are commercial products. Luckily, some commercial vendors today also offer free versions of their virtualization software.

Microsoft, for example, gives away the Windows version of VirtualPC for free. VirtualPC simulates a standard PC, but officially only runs Windows operating systems as guests, even though it should be possible to run Linux as well. VirtualPC is also available for Macintosh computers, but not for free.

The free software XEN, developed at Cambridge University, requires either processors with virtualization support, or the operating system kernel of the guest system has to be adapted to XEN. Obviously, this is not possible with commercial closed-source operating systems.

Another product, Bochs, is released under LGPL and emulates x86 and IA32 processors. With Bochs, it would be possible to install Windows on a PowerPC processor. Emulation, however, is very processor-intensive, so guest systems in Bochs are rather slow.

Other virtualization products are Plex86 and QUEMU, which both are free. VirtualBox is dual-licensed and available in a free and a commercial version. Another commercial vendor is Parallels. One of the best known vendors of virtualization solutions, however, is VMware Inc., based in Palo Alto, California.

VMWare

VMware offers various commercial and free virtualization products. VMware Workstation, though not a free product, is a great tool for developers, that is definitely worth its license fee. VMware Workstation is available in a Windows and a Unix version and supports most well-known operating systems as guests. For Apple users, there is a product called VMware Fusion, which allows you to install Windows or

Linux as guest operating systems on a Mac. Unfortunately, VMware Fusion only runs on x86-compatible architectures.

The number of virtual machines you can create is only limited by the available storage space. Every virtual machine uses quite some space, as its hard disk is being represented by a file in the host computer's file system. The number of virtual machines you can run in parallel is limited by the amount of RAM your host computer has, because as soon as the operating system starts swapping, performance is almost down to zero.

VMware offers to free products, namely the VMware Player and the VMWare Server. With the player, you can run virtual machines, but you cannot create or modify them. While, at first glance, VMware server may seem to be a free alternative to VMware Workstation, it is lacking the snapshot and cloning features that are particularly important for development.

We will have a closer look at VMware Workstation, and see how this product can be used to replace a large hardware-based test lab. You can download the software from `http://www.vmware.com/download/ws`. A free 30-day evaluation copy is available when you register on the website and request a license key by email.

Installing A Virtual Machine

To work with a virtual machine, it has to be created first. To do this, click *New* in the *File* menu and select *Virtual Machine* from the flyout menu. In most cases, you should choose to create a typical virtual machine, except if you need it to be compatible to older VMware versions.

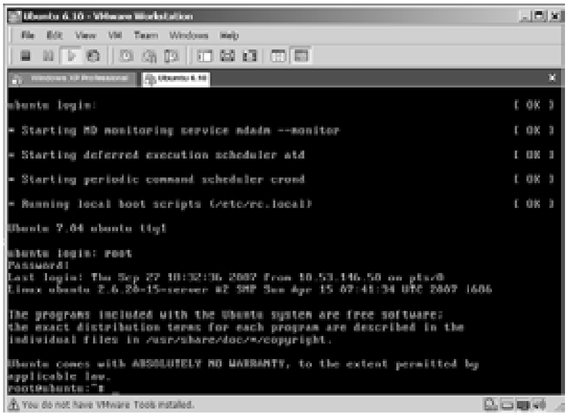
In the second step you can choose the type of the guest system. The hardware of the virtual machine will be chosen based on what guest operating system you have chosen. The range of supported guest systems reaches from nostalgic Windows 3.1 to experimental support for 32bit and 64bit versions of Windows Vista. In addition, common Unix and Linux distributions are supported, as well as Sun Solaris, Novell Netware, FreeBSD, and even MS-DOS.

After choosing a name for the virtual machine and specifying the directory where the virtual machine is to be stored, you have to decide which type of networking to use. VMware supports various types of networking, with bridged networking usually being the best choice, as it allows the guest system to directly use the host system's network adapter. To all other computers in the network, the virtual machine seems to be physically connected to the network using its own IP address.

The next step is an important decision. You must decide on the size of the virtual hard disk. In most cases, 4 GB or 8 GB are suggested, if you use Windows Vista as the guest system, this will not suffice. Once you have chosen the size of the hard disk, you cannot change it any more, so it is better to be generous instead of realizing that, after the guest system is installed and configured, there is not enough disk space to install the required applications.

When the virtual machine has been created, click the Power On button (this is the button with the well-known Play symbol). You will now see the BIOS self test, probably followed by an error message that no operating system is installed. Now you can install an operating system inside the virtual machine like you would on real hardware.

To access CDs and DVDs, you can either use the hardware drive of the host system, which is mapped to the drive of the virtual machine, or map the virtual machine's drive to an ISO image. This makes it possible to work with CDs and DVDs on virtual machines even when the host system has no physical optical disk drive.


Ubuntu 6.10 - VMware Workstation
File Edit View VM Team Windows Help
Windows XP Professional
Ubuntu 6.10
You do not have VMware Tools installed.

```
ubuntu login:                                                [ OK ]

* Starting MD monitoring service mdadm --monitor             [ OK ]

* Starting deferred execution scheduler atd                  [ OK ]

* Starting periodic command scheduler crond                  [ OK ]

* Running local boot scripts (/etc/rc.local)                 [ OK ]

Ubuntu 7.04 ubuntu tty1

ubuntu login: root
Password:
Last login: Thu Sep 27 10:32:36 2007 from 10.53.146.50 on pts/0
Linux ubuntu 2.6.20-15-server #2 SMP Sun Apr 15 07:41:34 UTC 2007 i686

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
root@ubuntu:~# _
```

Figure 7.6

Working With Snapshots

One of the big advantages of VMware Workstation over other virtualization products is the sophisticated snapshot management. At any point in time, you can store a snapshot of the guest system which included its current state. Snapshots are much easier to handle than full backups, and are smaller, since only changes (and not the full hard disk) are stored in the snapshot.

With snapshots, you can rollback any changes you have made to the system immediately by just restoring the last snapshot. Snapshots are ideally suited to test installations or run software tests on different operating systems and in different configurations. Once you have automated your tests, you can execute them in multiple virtual machines without additional human effort.

When migrating, I would recommend to set up a virtual machine that represents the current system and one that represents the target system. Though virtual machines are a little slower than real hardware, the snapshot management will make your migration much easier.

Test Tools

The most important aspect of migrating PHP code is probably testing. Only by testing can you make sure that migrated code works as expected on the target system. To do this, you can compare the test results of the old system with the test results on the target system.

Tests must be repeated very often. In theory, you must repeat all tests after each code modification, to make sure that you did not introduce a bug. The effort of manually repeating the same tests over and over, however, is clearly not bearable, so you must automate the tests.

In a perfect world, you would have a suite of tests that cover the whole application's functionality. In reality, especially for old applications, few or even no tests at all exist. To be fair, one has to take into account that years ago the tests tools that are available today did not exist, so the infrastructure to write sophisticated tests was just not there.

Today, a number of great test tools exist, and a migration gives you a good reason to get into automated testing. There are two kinds of tests. You can either test the whole application, as if it was used by a human user. These kinds of tests are called system tests or functional tests. They test the HTML user interface of the application. It is not sufficient, though, to statically analyze the HTML code generated by the application. As modern web application make intensive use of JavaScript on the client side, you will also want to execute and test the JavaScipt code. That is why system tests need a relatively complex environment, especially when you want to test edge cases.

The second type of tests are unit tests that test parts of the application in isolation. A part of the application can be a class, a module, or a whole layer. Unit tests require a less complex environment and are therefore easier to handle.

To thoroughly test your code, you will have to combine both test types. It is important to note that tests can never prove that an application is bug-free. A test can only prove that a bug is present.

Unit Tests with PHPUnit

The first unit test tools originate in the Smalltalk world. They became famous when Erich Gamma and Kent Beck, two of the founding fathers of Extreme Programming, ported the software from Smalltalk to Java. Today, there is a unit test framework for almost every programming language. They are often being referred to as xUnit frameworks.

As we had already mentioned in chapter 2, an automated test basically means comparing the program output with a pre-calculated value. Any deviations outside the specified tolerances will make a test fail. The xUnit framework provides us with an infrastructure for automated tests. You can run a number of tests, independently from each other. The framework creates a fresh environment, the so-called *fixture*, for each test.

The best known xUnit framework for PHP is PHPUnit, written by Sebastian Bergmann. PHPUnit used to be part of the PEAR project, but is today developed as an independent project and can be installed from PHPUnits own channel server. Make sure that you do not install PHPUnit from the PEAR channel server:

```
> pear install phpunit

WARNING: "pear/PHPUnit" is deprecated in favor of
```

```
         "channel://phpunit.de/PHPUnit"
```

As you can see, the PHPUnit version in PEAR is outdated, but will still be installed. The first PHPUnit version was written in PHP 4 and should not be used any more, since the functionality is rather limited. To install the current version of PHPUnit, you must first register the PHPUnit channel, then install the software:

```
> pear channel-discover pear.phpunit.de

Adding Channel "pear.phpunit.de" succeeded Dis-
covery of channel "pear.phpunit.de" succeeded

> pear install phpunit/phpunit

Did not download optional dependencies: pear/Image_GraphViz, pear/Log, use --
    alldeps to download automatically
phpunit/PHPUnit can optionally use package "pear/Image_GraphViz" (version >=
    1.2.1)
phpunit/PHPUnit can optionally use package "pear/Log"
phpunit/PHPUnit can optionally use PHP extension "pdo"
phpunit/PHPUnit can optionally use PHP extension "pdo_mysql" phpunit/PHPUnit
can optionally use PHP extension "pdo_sqlite" phpunit/PHPUnit can optionally
use PHP extension "xdebug" (version >= 2.0.0) downloading PHPUnit-3.1.8.tgz
...
Starting to download PHPUnit-3.1.8.tgz (116,960 bytes)
 .......................done: 116,960 bytes
install ok: channel://pear.phpunit.de/PHPUnit-3.1.8
```

You can install all optional dependencies by using the command line switch –alldeps. This will install a number of PEAR packages that you do not need for basic testing. Please note that PHPUnit requires PHP 5, which prevents you from running unit tests in the existing environment when you are migrating a PHP 4 application.

PHPUnit is a command line tool. We have already learned that PHP can be configured differently at the command line and inside the web server, so make sure that the tested code does not behave differently in production just because of a different PHP configuration.

Let us write a simple PHPUnit test:

```
<?php
```

```
    require_once 'PHPUnit/Framework.php';

    class FooTest extends PHPUnit_Framework_TestCase
    {
      public function testAdd()
      {
        $foo = new Foo;

        $this->assertEquals(3, $foo->add(1, 2));
        $this->assertEquals(2, $foo->add(-2, 4));
        $this->assertEquals(1, $foo->add(0, 0));

      }
    }

    class Foo
    {
      public function add($a, $b)
      {
        return $a + $b;
      }
    }

  ?>
```

By convention, the names of test classes always end in `Test`. The test method names start with `test`. When calling PHPUnit at the command line and passing the name of the `test` class as parameter, all test methods of this class will automatically be executed. Keep in mind that the `test` class must load all required production code and classes to test.

In the above example, we have deliberately made a mistake: the expected result of adding 0 and 0 is definitely not 1. When running the tests, we expect two assertions to pass, while one assertion should fail:

```
> phpunit FooTest

PHPUnit 3.2.15 by Sebastian Bergmann.

F

There was 1 failure:
```

```
1) testAdd (FooTest)

FooTest.php:13

FAILURES!!!

Tests run: 1, Failures: 1, Errors: 0, Incomplete Tests: 0..

Time: 0 seconds
```

PHPUnit supports a large set of assertions. A full list is available at `http://www.phpunit.de/pocket_guide/3.2/en/api.html#api.testcase`. In our example, we test a very simple method that adds two values. For more complex code, it is not always easy to find appropriate input values. The key to success is finding a small set of representative test cases.

First up, always test the main functionality. It may be tempting to create test cases for edge cases, but you can quickly get carried away in creating too many tests for rare exceptions, rather than testing the application's functionality.

The more unit tests exist for your application, the more confidence in your code you will gain when they still work on the target system. Any failed test indicates a problem that has been caused by the migration.

Most existing applications do not have unit tests. In that case, write a unit test for every problem you encounter when migrating. The test should be written so that it fails and exposes the bug. Then, fix the code and rerun the tests. As soon as the test passes, you know that you have fixed the bug. Not only will you gradually build up a set of unit tests, but you also will quickly find out when a bug that was already fixed suddenly reappears.

Though PHPUnit is object-oriented, unit tests are not only suited for object-oriented code. You can test procedural code as well. If the code has a many dependencies on global variables, though, it can be tedious to set up an environment for each test.

Generally, the more modular your application is, the easier testing becomes. Ideally, the tested components should have no dependencies. Objects that interact with databases, for example, can be replaced by dummy objects, the so-called mock objects.

To automate a number of PHPUnit tests, you can use the build automation tool phing, which is described in this Chapter.

System Tests with Selenium

A system test tests the application through its user interface. Even when all components of the application are well tested, there is no guarantee that all components will work together without problems. The NASA probe Climate Orbiter is a sad example for this: after a course correction that went wrong, the probe incinerated in the Mars atmosphere. The problem was that one software module used the metric system, while another module used the imperial system. When passing values between these modules, they were misinterpreted, which lead to the loss of Climate Orbiter.

Today, the best approach to testing web application is to run it in a real browser and remote control this browser. An example for a system test of a web application could be:

- We try to access the confidential area of a website. We expect either a login screen or an error message to appear
- We enter invalid credentials to the login form. We expect an error message with the question, whether the user wants to retrieve a lost password
- We enter valid credentials and expect a success message, or to view the protected page that was originally requested
- We reload the page to make sure that we still have access. We expect to see the same page again without an error message
- We click Logout and expect to see the login form again
- We click on “Back” and reload the page. We expect to see an error message or the login page, but not the protected page

As you can see, just as with unit tests, there are certain assertions that must be fulfilled for each step of the test.

Let us now use the free software Selenium to automate our system tests. The core of Selenium (in due course, called Selenium Core) is written in JavaScript that is embedded into a page in an IFrame and remote controls the website. This is technically not easy, and as we will see, there are still some issues with this due to the browser's security restrictions.

Selenium supports most common browsers, but usually only in newer versions. On Windows, Internet Explorer, Firefox and Opera are supported. On Linux, Konqueror and on Mac, Camino and Safari are supported as well. A full list of supported browsers including version numbers is available at [Openqa 2008].

There are two versions of Selenium. The Selenium IDE is a full-blown integrated development environment for automated tests distributed as a Firefox plugin. With Selenium IDE, you can record your browser interaction and reply it later. You can also add assertions to the tests to make sure the application behaves as expected.

The second Selenium product is Selenium Remote Control (Selenium RC). Selenium RC is Java software and has its own web server that acts as a proxy to the web server to test. This proxy allows you to run system tests directly from PHPUnit. Since a proxy is used, some security restrictions of the browser, namely the same origin policy that usually prevents JavaScript from different URLs to be executed in a page, can be circumvented. This allows us to test remote systems.

The preferred way of using Selenium is to record tests with Selenium IDE, then use PHPUnit to run these tests. In conjunction with a build automation tool like phing, system tests can be fully automated.

An important precondition for system tests is a defined initial state of the application, also called *test fixture*. The database must exist and be populated with sensible default values that may depend on each test case.

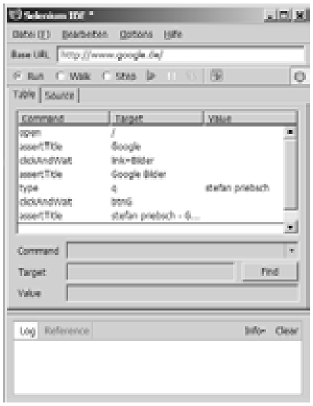
To isolate the tests from each other, you should install the application in the test's `setUp()` method and put it into a defined state, for example by restoring a backup test database, or by sending HTTP requests to login to the application.

Setting up a test environment for each system test can be tedious and makes testing slow, but it is still better to automate the tests instead of running them manually over and over again.

Capture & Replay Tests with Selenium

To record tests, we must install Selenium IDE first. As previously mentioned, Selenium IDE is a Firefox plugin which you can download from `http://www.openqa.org/selenium-ide/download.action`. Since you are not installing the plugin from an official Mozilla page, you may have to add `openqa.org` to the list of sites that are allowed to install software.

After restarting Firefox, the *Extra* menu has a new entry, *Selenium IDE*. Click on this menu entry to start Selenium IDE. A new window as shown in Figure 7.7 will open. By default, Selenium records your browser interactions. There is a small catch, though: Selenium has not recorded the URL of the page that you were already viewing. You can also not type the URL into the address bar of the browser, since Selenium has no read access to it. The same holds true for the *Forward* and *Back* buttons of your browser, so you should not use them while recording tests.



Base URL: http://www.google.de/

● Run ○ Walk ○ Step

Command	Target	Value
open	/	
assertTitle	Google	
clickAndWait	link=Bilder	
assertTitle	Google Bilder	
type	q	stefan priebsch
clickAndWait	btnG	
assertTitle	stefan priebsch - G...	

Command:
Target:
Value:

Figure 7.7

Make sure that the URL you are viewing matches the *Base URL* in the Selenium window. Now, click a link that takes you to the page to test. If the base URL does not match the URL you were viewing, replaying the tests is likely to fail, as the recorded interactions will be sent to the wrong page.

You cannot change the base URL during a test. The reason is a security feature in JavaScript, namely the same origin policy. You can also not switch between HTTP and HTTPS in a test, since this also counts as a change of URL, even when you access the same page with both protocols.

To insert assertions, select them from the drop down menu and enter the required parameters. The assertions allow you to test virtually any aspect of the HTML page (or the DOM tree, respectively).

The button to stop recording is not easy to find. To replay a recorded test, stop the recording, choose *Run*, and click the icon with the *Play* symbol.

Automated System Tests with the Proxy

Tests that have been recorded with Selenium IDE can be run by PHPUnit in conjunction with a Java Runtime Environment. Before we begin, you should make sure that you have a current Java Runtime Environment (JRE) installed:

```
> java -version

java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b06)
Java HotSpot(TM) Client VM (build 1.6.0_02-b06, mixed mode, sharing)
```

When this chapter was written, Java 1.6.0_02 was the latest version. This version is marketed by Sun as Version 6 Update 2. If you have an older Java version, you should update it, except when working with Java-based software that specifically requires an older Java version.

Windows users can download Java from [Sun 2008], on most Unix systems, Java should be installed by default. If this is not the case, you can either install Java with your system's packet manager, or download binaries from the Sun website.

Now you can install Selenium RC. Download the software from `http://www.openqa.org/selenium-rc/download.action` and unpack the zip archive. Always use the latest available version of Selenium RC. Since Selenium RC is Java software, no installation is required. You can directly run Selenium from the `selenium-server.jar` archive that is located in the `selenium-server-0.9.2` directory. Let us start the server:

```
> java -jar /path/to/selenium-server-0.9.2/selenium-server.jar
```

```
11:35:40.953 INFO - Java: Sun Microsystems Inc. 1.6.0_02-b06
11:35:40.984 INFO - OS: Windows XP 5.1 x86
11:35:41.000 INFO - v0.9.2 [2006], with Core v0.8.3 [1879]
11:35:41.562 INFO - Version Jetty/5.1.x
11:35:41.578 INFO - Started HttpContext[/selenium-server/driver, /selenium-
    server/driver]
11:35:41.593 INFO - Started HttpContext[/selenium-server,/selenium-server]
11:35:41.593 INFO - Started HttpContext[/,/]
11:35:41.734 INFO - Started SocketListener on 0.0.0.0:4444
11:35:41.734 INFO - Started org.mortbay.jetty.Server@5ac072
```

The Selenium server listens to port 4444, so you may have to configure your firewall accordingly. Leave the console window open, otherwise you will end the server.

To run Selenium RC on a server without a GUI, you can use the virtual X server xvfb to start the browsers. Keep in mind that you do not need Selenium RC to work with Selenium IDE, but only to run tests from PHPUnit.

To do this, record a test in Selenium IDE, then save it and export it to the PHPUnit format by choosing *Export Test As* in the *File* menu and selecting *PHP - Selenium*. When this chapter was written, Selenium IDE 0.8.7 was the most recent version that unfortunately creates PHPUnit tests for an older PHPUnit version.

You can solve this problem by patching Selenium IDE with a modified `selenium-ide.jar` which you can download from my blog at [Priebsch 2007]. Copy this file into the `extensions{a6fd85ed-e919-4a43-a5af-8da18bda539f}\chrome` subdirectory of your Firefox profile directory. This patch has also been submitted to the Selenium developers, so hopefully the next Selenium release will include this patch.

When exporting a test case, the class name defaults to `Example`. Export the test to a file `Example.php` so that you can run the test directly:

```
phpunit Example
```

If the file name does not match the class name (without `.php`, obviously), you can specify the file name and path as second parameter:

```
phpunit Example path/to/the/testfile.php
```

Let us have a look at a test exported from Selenium IDE:

```
<?php

require_once ’PHPUnit/Extensions/SeleniumTestCase.php’;

class Example extends PHPUnit_Extensions_SeleniumTestCase
{
  function setUp()
  {
    $this->setBrowser("*firefox");
    $this->setBrowserUrl("http://localhost/");
  }

  function testMyTestCase()
  {
    $this->deleteCookie("symfony", "/");

    $this->open("/symfony/web/login");
    $this->assertEquals("the page title", $this->getTitle());
    $this->assertTrue($this->isTextPresent("Please log in:"));

    $this->type("username", "admin");
    $this->type("password", "the password");
    $this->click("commit");
    $this->waitForPageToLoad("30000");

    $this->assertEquals("the page title", $this->getTitle());
    $this->assertTrue($this->isTextPresent("User: admin"));
    $this->assertTrue($this->isTextPresent("There are no files"));

    $this->assertEquals("the page title", $this->getTitle());
    $this->assertTrue($this->isTextPresent("Select a file to upload"));
    $this->assertEquals(">e-novative> dox", $this->getTitle());

    $this->type("file", "C:\www\symfony\selenium\testfile.txt");
    $this->click("commit");

    $this->click("//a[@onclick=\"return confirm(’Delete file?’);\"]");
    $this->waitForPageToLoad("30000");

    $this->click("link=Edit Profile");
    $this->waitForPageToLoad("30000");

    $this->assertEquals("the page title", $this->getTitle());
    $this->type("firstname", "first");
    $this->type("lastname", "last");
    $this->type("company", "company");
    $this->type("email", "user@host.com");

    $this->click("commit");
    $this->waitForPageToLoad("30000");

    $this->assertEquals("the page title", $this->getTitle());
  }
}

?>
```

This example is a test of an application that has been created using the Symfony framework. Now let us run this test in PHPUnit. First, start the Selenium RC server, then run the tests:

```
> phpunit Example

PHPUnit 3.2.15 by Sebastian Bergmann.

.

Time: 10 seconds

OK (1 test)
```

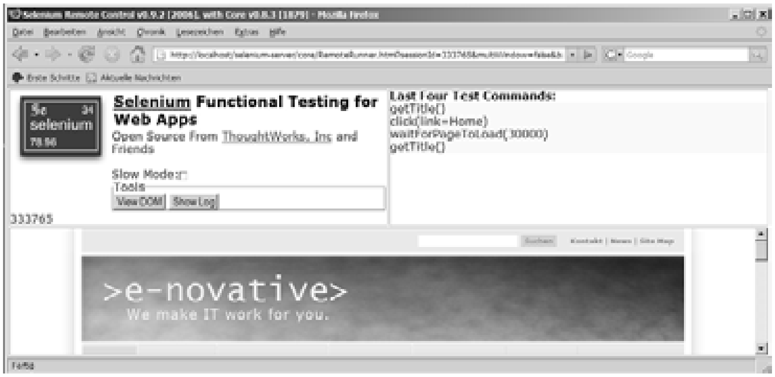
While the test is running, a browser window appears, so you can watch the test progress (see Figure 7.8). In our example, the test has completed successfully.

We have already mentioned that there are several general issues with Selenium tests due to security restrictions in the browsers. It is not possible to upload a file, for example, because the browser would have to access the local filesystem to read the file. For a very good reason, this is not allowed.

Some of these limitations can be circumvented by running Firefox in *Chrome* mode, and Internet Explorer in *HTA* mode. In Firefox, the Chrome mode makes file uploads possible, for example.

To run Firefox in Chrome mode, you must modify the `setUp()` method of your test:

```
$this->setBrowser("*chrome");
```



```
Last Four Test Commands:
getTitle()
click(link=Home)
waitForPageToLoad(30000)
getTitle()
```

Selenium Remote Control v0.9.2 [2006], with Core v0.8.3 [1879] - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
Erste Schritte
Aktuelle Nachrichten
Se 34 selenium 78.96
Selenium Functional Testing for Web Apps
Open Source From ThoughtWorks, Inc and Friends
Slow Mode:
Tools
View DOM
Show Log
333765
Suchen
Kontakt | News | Site Map
>e-novative>
We make IT work for you.
Fertig

Figure 7.8

Selenium does not support HTTP authentication. HTTP authentication asks for username and password, and again it would be a serious security risk if the browser was allowed to access the credentials by JavaScript. According to RFC 3986 (`http://tools.ietf.org/html/rfc3986`), username and password can also be submitted via the URL. Theoretically, you could make HTTP authentication work for PHPUnit tests, but some modern browsers like Internet Explorer do not support RFC 3986 any more - for security reasons, as you may have guessed. Since most web applications use a HTML-based login form instead of HTTP authentication, this should not be an issue. If your application uses HTTP authentication, you can always disable it for the Selenium tests.

Program Analysis and Debugging

The PHP extension xdebug written by Derick Rethans is a Swiss army knife for PHP developers. It supports tracing, profiling, debugging and creation of code coverage statistics. In this section, we will focus on the features that can be useful when migrating code. xdebug should be installed on every development system. It will definitely make the migration easier for you.

Installation

The xdebug installation differs between the different operating system families. On Unix, you can try install xdebug with the following code:

```
> pecl install xdebug
```

This command will not work on all systems, though, so you might have to compile xdebug by yourself (please adapt the paths to your system):

```
> wget http://xdebug.org/link.php?url=xdebug202
> tar -xzf xdebug-2.0.2.tgz
> cd xdebug-2.0.2
> phpize
> ./configure --enable-xdebug --with-php-config=/usr/bin/php-config
> make
> cp modules/xdebug.so /usr/lib/apache2/modules/xdebug.so
```

Installation is even easier for Windows users. Download the correct DLL for your PHP version from `http://pecl4win.php.net/ext.php/php_xdebug.dll` and activate xdebug by adding the following entry to `php.ini`:

```
zend_extension_ts="c:\php\ext\php_xdebug.dll"
```

The shown entry is valid on a Windows system, on Unix you would specify the full path to `xdebug.so`. Depending on the thread model of your web server (or PHP, respectively), you must load xdebug either as a thread-safe or non-thread-safe extension. In the latter case, use `zend_extension` instead of `zend_extension_ts`. For more information about thread models, see chapter 3.

Now let us check whether xdebug has been loaded correctly:

```
> php -m

[PHP Modules]
bcmath
bz2
[...]
xdebug
```

```
[...]
xsl
zlib

[Zend Modules]
Xdebug
```

As we can see, xdebug appears twice in the list, once in the PHP Module section, and again in the Zend Module section.

Useful Features

When xdebug is activated, it provides you with some very useful small features that you will not want to miss after you worked with them for a while.

First of all, the `var_dump()` output is beautified. For example, xdebug improves the formatting, and outputs size and data type of each value. Second, along with each error message, xdebug shows a stack trace, and can optionally display additional information. In complex programs, a stack trace can help you to better understand the program, since you can see which function and method lead to the error.

To protect you from crashing PHP, xdebug prevents endless recursion. As soon as the stack depth reaches a limit that has been configured in `php.ini`, the program is stopped. Please note that this feature will not protect you from endless loops, but only from endless recursion. You can add protected against endless loops at PHP level by introducing a counter and stopping the program one this counter has reached a certain level. This kind of protection can be useful while developing or migrating, but should be deactivated or removed in production code.

Tracing

For complex, object-oriented applications, it is very difficult to understand the control flow of a program. Many programs write important decisions to log files, but the problem is that when debugging, quite often the relevant information is not in the log file.

With xdebug, you can create a trace log of the whole application. The trace log contains all function calls, optionally with parameters and return values. Instead of trying to add logging commands to the application in the right spots, create a full trace log and use tools like grep or your favorite IDE or text editor to find the interesting parts of the log file.

To have xdebug write a trace log, you must add some settings to php.ini:

```
xdebug.auto_trace=On
xdebug.trace_output_dir=c:\log
```

These settings must appear after the setting that loads xdebug, otherwise they will not be recognized. The specified log directory must exist and be writeable for the user PHP runs as.

With these settings, xdebug will create a minimal trace log. By adding some settings to php.ini, you can create a trace log with more information:

```
xdebug.show_mem_delta=On
xdebug.collect_params=4
xdebug.collect_return=On
```

Now the log file will contain information about the memory consumption, the parameters passed to each function, and the return values. Do not forget to restart your web server once you have changed php.ini, otherwise the settings will not be effective.

Let us have a look at the trace log of a simple recursive factorial program:

```
TRACE START [2007-10-26 12:30:04]
    0.0133      55704      +48     -> fac($x = 7) test.php:8
    0.0133      55840     +136       -> fac($x = 6) test.php:13
    0.0134      56096     +256         -> fac($x = 5) test.php:13
    0.0134      56352     +256           -> fac($x = 4) test.php:13
    0.0134      56632     +280             -> fac($x = 3) test.php:13
    0.0135      56912     +280               -> fac($x = 2) test.php:13
    0.0135      57192     +280                 -> fac($x = 1) ...
    0.0135      57472     +280                   -> fac($x = 0) ...
                                                 >=> 1
                                               >=> 1
                                             >=> 2
                                           >=> 6
                                         >=> 24
                                       >=> 120
                                     >=> 720
```

```
                                  >==> 5040
    0.0140      57648     +176     -> xdebug_stop_trace()test.php:26
    0.0140      57648
TRACE END   [2007-10-26 12:30:04]
```

Since xdebug has to write one line to the log file for each function call, the application will be really slow when tracing is enabled. Therefore you should only enable tracing when you really need it. Never trace an application on a production system.

Debugging

As its name suggests, xdebug is a full-blown remote PHP debugger. If PHP runs in debugging mode, you can pause the program execution at any time. As soon as an error or exception occurs, program execution is automatically paused. When the program is paused, you can inspect the current variable values and the call stack on the client, and even modify variables. Then, you can choose to continue the program or run the application step by step.

Ecplise PDT, the free PHP plugin to Eclipse, contains a debug client that supports the Zend debugger and xdebug. Since Ecplise is Java software, it should run without installation on most available systems. You can download Eclipse PDT from `http://www.eclipse.org/pdt/downloads`. Choose the *All-in-One* package for your platform, unless you already use Eclipse.

Now some configuration is required to set up debugging with xdebug. On the server, we must enable the xdebug debugger and specify the DNS name or IP address of the debug client:

```
xdebug.remote_enable=On
xdebug.remote_host="localhost"
xdebug.remote_port=9000
xdebug.remote_handler="dbgp"
```

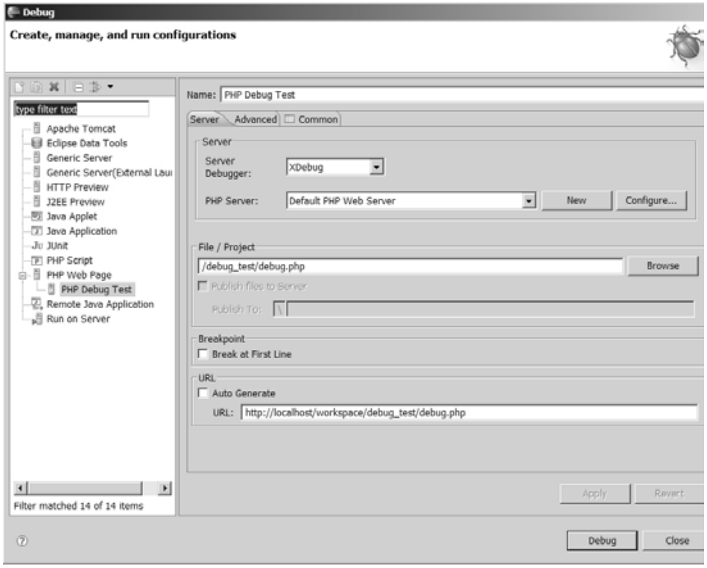
For our example, we are using a local Apache PHP-enabled web server, so the debug server and debug client are on the same system.

Now we must configure Eclipse PDT since by default, if is configured to use the Zend debugger. Click *Preferences* in the Window menu and expand the PHP subtree on the left. Now, change the PHP debugger setting to xdebug and click *Apply* to save the changes.

Next, we create a debug configuration. Choose *Open Debug Dialog* from the *Run* menu and create a new debug configuration by double-clicking *PHP Web Page.*

You will see a window with three tabs, namely *Server*, *Advanced*, and *Common.* Make sure that xdebug is selected as *Server Debugger.* In the *File/Project* field you must enter the path to the script you want to debug, relative to the workspace. If you debug remotely, you will need an identical version of the source code on the server and on the client.

To enable Eclipse to map URLs to local filenames, the URL field must show the correct URL of the script you just entered. If the value is not correct, unselect *Auto Generate* and fix it. Click *Apply* again to save the changes. Now, your window should look like the one in Figure 7.9.



Debug

Create, manage, and run configurations

Name: PHP Debug Test

Server Debugger: XDebug

PHP Server: Default PHP Web Server

File / Project: /debug_test/debug.php

□ Publish files to Server

Publish To: \

□ Break at First Line

□ Auto Generate

URL: http://localhost/workspace/debug_test/debug.php

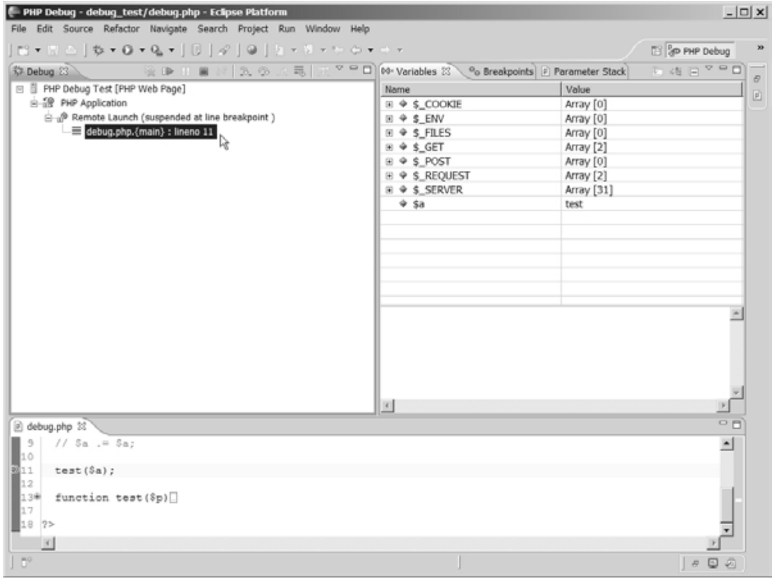
Filter matched 14 of 14 items

Figure 7.9

Now you can start a debug session by selecting the *Debug* entry from the *Run* menu. Make sure that no firewall between the debug client and server is blocking the communication on port 9000.

Figure 7.10 shows the debug view of an example program. On the right, you see the current variable values. At the bottom, the source code of the executed file is shown. The line that was executed last is highlighted.

By default, Ecplise pauses program execution at the first line of a PHP script. You can disable this feature by unselecting the setting *Break at First Line* in the *Breakpoint* section of the debug configuration.



PHP Debug Test [PHP Web Page]
PHP Application
Remote Launch (suspended at line breakpoint)
debug.php.{main} : lineno 11

Name	Value
$_COOKIE	Array [0]
$_ENV	Array [0]
$_FILES	Array [0]
$_GET	Array [2]
$_POST	Array [0]
$_REQUEST	Array [2]
$_SERVER	Array [31]
$a	test

```
// $a .= $a;

test($a);

function test($p)□

?>
```

Figure 7.10

To change the value of a variable, click it and enter a new value. To set a breakpoint, right-click a line in the source code and choose *Toggle Breakpoints* from the context menu. You can also set conditional breakpoints by choosing *Set Condition* from the context menu. Enter the condition as a normal PHP expression, using the program variables as needed.

Remote debugging can be very useful to find bugs in complex software without making any changes to the source code. This is also called non-invasive debugging.

Code Coverage

Code coverage tells you how much of your code is being tested. A code coverage of 100% is not realistic, though, since there are usually edge cases that are very complicated to test. If your application contains code to handle such edge cases, testing this code is often not worth the effort.

On top of that, 100% code coverage does not mean that the code has been thoroughly tested, as the following example shows:

```
<?php

  ...

  if ($a)
  {
    print '1';
  } else {
    print '2';
  }

  if ($b)
  {
    print '1';
  } else {
    print '2';
  }

?>
```

This example has four possible execution paths, as table 7.2 shows. By testing the two execution paths `$a = true`, `$b = false` and `$a = false`, `$b = true` full code coverage is reached, though two execution paths still remain untested:

a	b	Result
true	true	11
true	false	12
false	true	21
false	false	22

xdebug only collects the raw code coverage statistics. To aggregate and visualize them, you can use phing, which we will introduce in the next section.

Build Automation

When developing software, there are quite some tasks that must be repeated many times. A good IDE will help you to automate tasks like code quality assurance, running tests, creating API documentation, or creating a software release in the form of a `.tar.gz` or ZIP archive, or even a PEAR package.

It can be a good idea, though, to automate these tasks independently from the IDE. This will enable you perform the tasks not only on one development machine, bus also on a central build server, for example.

Since we are doing PHP, why not choose a PHP-based tool to automate builds? The free software phing is based on Apache Ant, which is written in Java. phing just requires some additional PEAR packages, XML and XSL support in PHP. No additional runtime environment is requires, and phing can be extended in PHP, should the need arise. Thanks to phing, build processes can be portable and work across platforms.

Installation

Installing phing is easy and works cross-platform. The phing project runs their own PEAR channel server, so installing phing is a matter of two command line commands.

To install phing, you need a working PEAR environment of version 1.4 or newer. If an older version is present, you should upgrade PEAR as described earlier in this chapter. First, we must make the phing channel server known to PEAR:

```
> pear channel-discover pear.phing.info

Adding Channel "pear.phing.info" succeeded
Discovery of channel "pear.phing.info" succeeded
```

Now you can install phing from this channel server. The easiest way to do this is by adding the command line switch `-alldeps`. This tells the PEAR installer to automatically download and install all PEAR packages phing depends on. Since there is a problem with the PHPUnit installation (as discussed earlier in this chapter), you should install PHPUnit manually before installing phing:

```
> pear install --alldeps phpunit/phpunit

[...]

> pear install --alldeps pear.phing.info/phing

Failed to download pear/VersionControl_SVN within preferred state "stable",
     latest release is version 0.3.1, stability "alpha", use "channel://pear.php.
     net/VersionControl_SVN-0.3.1" to install
WARNING: "pear/PHPUnit2" is deprecated in favor of "channel://phpunit.de/PHPUnit
     "
WARNING: "pear/DB" is deprecated in favor of "pear/MDB2"
phing/phing can optionally use package "pear/VersionControl_SVN" (version >=
     0.3.0alpha1)
phing/phing can optionally use package "pear/Xdebug" (version >= 2.0.0beta2)
pear/PHPUnit2 can optionally use PHP extension "xdebug"
pear/Log can optionally use PHP extension "sqlite"
downloading phing-2.3.0beta1.tgz ...
Starting to download phing-2.3.0beta1.tgz (397,990 bytes)
 ................................................done: 397,990 bytes
```

Code Quality Assurance

PHP is not a compiled language like C or Java. PHP scripts are compiles before they are executed, but this happens right before the script is being executed. In the classical compiled languages, the compiler plays an important tole in quality assurance, because you cannot just deploy source code that does not compile. In PHP, programs are not compiled before they are executed, so you can deliver broken code and will not even notice until the customer tries to run the code.

Fortunately, PHP can run a lint check on the source code when called with the `-l` command line switch (as discussed earlier in this chapter). In conjunction with phing, it is very easy to run a PHP lint check.

The phing build process is driven by a XML file that defines various so-called targets. The build file should be named `build.xml`. The individual targets can depend on each other, which allows you, for example, to only create a release when all tests have passed.

```
<?xml version="1.0"?>

<project name="Test" default="build" basedir=".">
  <target name="clean">
    ...
  </target>

  <target name="build" depends="clean">
    ...
  </target>

</project>
```

To call phing, specify the target to execute at the command line:

```
> phing build

Buildfile: build.xml

www > clean:

www > build:

BUILD FINISHED

Total time: 2.6946 seconds
```

Inside the target, a number of tasks can be executed. Each task is being represented by an XML tag. There are various tasks, ranging from simple file operations like `copy` or `rename` to complex operations like running tests, generating API documentation, or creating a compressed archive or PEAR package.

To select files to be processed, you must define a *fileset*. When defining filesets, you have a great deal of flexibility, as wildcards can be used to select any combination of individual files or whole directories. To recursively select all files with a name that ends in `.php`, for example, you can use:

```
<fileset dir="src" id="php_files">
  <include name="*.php"/>
  <include name="**/*.php"/>
</fileset>
```

The two stars refer to the current directory and its subdirectories. It is a little awkward that phing does not include files from the root directory `src` when specifying `**/*Test.php`. This is why you have to add `*Test.php` as well.

To run a lint check on all files of a fileset, use the `phplint` task:

```
<target name="lint">
  <phplint>
    <fileset refid="php_files"/>
  </phplint>
</target>
```

When called, the lint target yields the following result:

```
> phing lint

Buildfile: build.xml

www > lint:

  [phplint] index.php: No syntax errors detected
  [phplint] .php: No syntax errors detected
  [...]

BUILD FINISHED

Total time: 12.3692 seconds
```

Code quality assurance is not limited to PHP lint checks. Using the `xmllint` task, you can run a lint check on an XML file. The undocumented jslint task allows you to run a JavaScript Lint check if the tool jsl is available.

If you use a Zend IDE, you can use the analyze task to run the Zend Code Analyzer directly from phing. Additional quality assurance can be put in place by calling external programs with an exec task. Keep in mind that this will make your build process dependent on external tools, though.

Test Automation

Using PHPUnit, phing supports running automated tests. The advantage of using phing is that you can define a fileset that contains all test cases to execute, and phing will take care of the rest. You do not have to deal with defining a *TestSuite* in PHPUnit.

By convention, the names of test classes always end in `Test`, so it is easy to define a fileset that contains all unit tests of our application:

```
<fileset dir="src" id="unit_tests">
  <include name="*Test.php"/>
  <include name="**/*Test.php"/>
</fileset>
```

We assume that both source code and tests are located in the `src` directory and its subdirectories. The two asterisks in the second include tag refer to all subdirectories. It is a small oddity of phing that the test files in the given root directory are not included by this tag. That is what we need the first include tag for.

```
<target name="unit_tests">
  <phpunit2 printsummary="true">
    <batchtest>
      <fileset refid="unit_tests"/>
    </batchtest>
  </phpunit2>
</target>
```

Though the phing tags are still named `phpunit2`, phing does work with PHPUnit3. Let us run the tests:

```
> phing unit_tests

Buildfile: build.xml

www > unit_test:

 [phpunit2] Tests run: 2, Failures: 0, Errors: 1, Time elapsed: 0.01 sec
 [phpunit2] Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.04 sec
 [...]

BUILD FAILED
```

```
build.xml:305:39: One or more tests failed

Total time: 29.2344 seconds
```

If you add a formatter tag to the phpunit2 task, phing will write all test statistics to a file. You can then use the phpunit2report task to create a nice HTML report from this data.

If you use PHPUnit to run system tests as well, you should create a separate target systemtest. Often, system tests are kept in a different directory, while unit tests are kept in the same directory where the production code is kept.

When creating a software release, you probably want to stop the build when a test fails. When migrating, you will probably want to run all tests to get a full report of all the failures and errors.

The haltonfailure and haltonerror attributes of the phpunit2 task determine whether the build is stopped, when a tests fails, or an error occurs. Both attributes default to false. When looking at test results, be warned that one failed tests often accounts for consecutive faults. You should therefore always fix the first reported error first, then rerun all tests.

Code Coverage Statistics

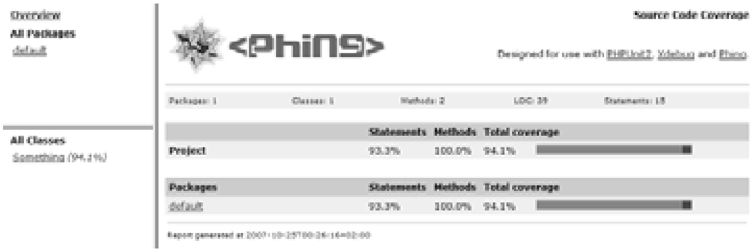
Together with xdebug and PHPUnit, phing can create a code coverage statistics report. The basic idea is to create code coverage statistics of the application code while running all unit tests. We will need two filesets to do this:

```
<fileset dir="src" id="php">
  <include name="*.php"/>
  <include name="**/*.php/"/>
  <exclude name="*Test.php"/>
  <exclude name="**/*Test.php"/>
</fileset>

<fileset dir="src" id="tests">
  <include name="*Test.php"/>
  <include name="**/*Test.php"/>
</fileset>
```

The first fileset contains the PHP files of the application, while the second fileset contains all unit tests. Now we configure phing to track code coverage statistics for the php fileset, run the unit tests, and create a report from the raw data:

```
<target name="code_coverage">
  <mkdir dir="coverage_db"/>
  <mkdir dir="coverage_result"/>

  <coverage-setup database="coverage_db/coverage.db">
    <fileset refid="php"/>
  </coverage-setup>

  <phpunit2 codecoverage="true">
    <batchtest>
      <fileset refid="tests"/>
    </batchtest>
  </phpunit2>

  <coverage-report outfile="coverage_db/coverage.xml">
    <report todir="coverage_result"/>
  </coverage-report>

</target>
```

To avoid a mix-up of old and new files, you should delete the subdirectories coverage-db and coverage-result at the beginning of the code_coverage target or write a clean target that code_coverage depends on.



Packages: 1   Classes: 1   Methods: 2   LOC: 39   Statements: 15

	Statements	Methods	Total coverage
Project	93.3%	100.0%	94.1%

Packages	Statements	Methods	Total coverage
default	93.3%	100.0%	94.1%

Report generated at 2007-10-25T00:26:16+02:00

Figure 7.11

Chapter 8

Migrating PHP Code

“Wise men learn by other men’s mistakes, fools by their own.” —*(Chinese Proverb)*

Chapter Overview

This chapter deals with the changes that must be made to PHP code when migrating. New PHP versions not only contain new features, but also bugfixes that might result in altered behavior of PHP functions or language constructs.

The PHP developers are working hard to keep backwards compatibility with older PHP versions. Sometimes, however, it is necessary to break backwards compatibility, for example when security problems have to be fixed.

After migrating the PHP environment, the PHP source code of the application to be migrated must be modified. Not only must the application run without error messages, but it must also behave identically on the target platform.

As we have already seen in Chapter 3, newer PHP versions can be configured to behave like older versions, which usually causes most trouble when migrating. The best idea is to make your application work in the default PHP configuration. This will save you trouble in the long run, because by keeping the existing configuration, you just push the pending problems farther ahead.

While the individual problems mentioned in this chapter are not too severe by themselves, ignoring them can lead to a chain of events that ultimately results in hard to find errors. This is why you should pay attention to even small details when migrating.

Case Sensitivity

In the third chapter, we learned that some operating systems are case sensitive, while others are case insensitive. Windows is case insensitive, which means that it does not distinguish between upper and lower case letters (in file names, at least). Unix is case sensitive, which is the reason why URLs are, at least in part, case sensitive as well.

PHP has a very pragmatic approach to case sensitivity, meaning that some aspects are case sensitive, while others are case insensitive. Fortunately, this behavior does not depend on the operating system PHP runs on, except for the file names, where PHP obviously relies on operating system functions, thus inheriting the operating system's case sensitivity here.

When working with strings, PHP is case sensitive by default, but you can force case insensitivity by using case insensitive string handling functions like `stristr()` instead of `strstr()`, for example. Another approach to force case insensitivity would be to convert all strings to upper or lower case when processing them. This can be useful when comparing strings, for example.

Variables

Variable names are always case sensitive, regardless of which operating system PHP runs on. The following example shows that PHP treats variable names with differing case as two distinct variables:

```
<?php

  $test = 3;
  $Test = 4;

  var_dump($test);
  var_dump($Test);

?>
```

This program creates the following output:

```
int(3)
int(4)
```

You should set strict guidelines as to which case to use for variable names, otherwise you risk inadvertently creating a new variable when you actually want to refer to an existing variable. Such errors are usually hard to find.

In most cases, you will not have to deal with case issues when migrating PHP code. Still, it is important to know how PHP behaves, and which problems you can expect, especially when migrating code that you have not written by yourself.

Constants

It is interesting to observe that constants are also case-sensitive, though by convention they are usually written in upper case letters. You should stick to this convention to make it easier to distinguish variables and constants.

The following example shows that PHP will treat constant names with differing case as two constants:

```
<?php

  define('TEST', 3);
  define('test', 4);

  var_dump(TEST);
  var_dump(test);

?>
```

This program generates the following output:

```
int(3)
int(4)
```

When defining a constant, you can use an optional boolean parameter that forces the constant to be case insensitive:

```
<?php

  define('TEST', 3, true);
  define('test', 4, true);

  var_dump(TEST);
  var_dump(test);

?>
```

In this case PHP will issue a notice when running the above program, as you are trying to redefine an already existing constant:

```
Notice: Constant test already defined in test.php on line 4

int(3)
int(3)
```

Unfortunately, many programmers configure PHP so that `E_NOTICE` errors are not displayed. In that case, they might not realize that the `define()` in line 4 had no effect and might end up asking themselves why the constant has a wrong value at a completely different point in the program.

PHP has some built-in constants. These are case sensitive as well, which means that you always have to capitalize them, otherwise you are referring to an undefined constant:

```
<?php

  var_dump(PHP_EOL);
  var_dump(php_eol);

?>
```

This program creates the following output:

```
string(2) "
"
Notice: Use of undefined constant php_eol - assumed 'php_eol' in test.php on
     line 6
```

```
string(7) "php_eol"
```

Again, PHP shows a notice informing you that an undefined constant has been used. This, again, shows the importance of always displaying all error messages, warnings, and notices.

This example shows another behavior typical for PHP: whenever an undefined constant is used, PHP creates a temporary constant at runtime. This happens, for example, when you output a string without quoting it. In the above example, the constant `php_eol` will be defined, containing the string “php_eol”.

Should you inadvertently use a temporary constant, you will really run into trouble when a constant of the same name is defined at some point in the future. PHP will then use the existing constant instead of creating the temporary constant, as the following example shows:

```
<?php

  define('Hello', 'Goodbye');

  var_dump(Hello);

?>
```

Instead of greeting you, the program will now output the following:

```
string(7) "Goodbye"
```

This is a very good example how a small and seemingly harmless problem that has been ignored for too long can become a serious problem in the end.

As a rule of thumb, a program should by default never emit any errors, warning or notices. Every message indicates a potential problem, that might surface when migrating the code.

Magic Constants

PHP knows the so-called magic constants `__LINE__`, `__FILE__`, `__FUNCTION__`, `__CLASS__` and `__METHOD__`. Starting with PHP 5.3, the magic constants `__DIR__` and

__NAMESPACE__ will be available. The names of magic constants always begin and end with two underscores. Magic constants are no real constants, but hold a value that changes depending on the execution state of the program. __LINE__, for example, always holds the number of the currently executed line of code, and __FUNCTION__ holds the name of the currently executed function.

You can use magic constants just like normal constants, but have to keep in mind that they are always case insensitive, as opposed to normal constants.

```
<?php

  var_dump(__LINE__);
  var_dump(__line__);

?>
```

This program generates the following output:

```
int(5)
int(6)
```

While __FUNCTION__, __METHOD__ and __CLASS__ always returned lower case characters in PHP 4, in PHP 5 these magic constants return the names as they have been defined. To keep PHP 4's behavior, use strtolower() to convert the result to lower case characters. It is convenient to embed the magic constant into a function:

```
<?php

  function get_current_classname()
  {
    return strtolower(__CLASS__);
  }

?>
```

In the long run, though, you should consider modifying your code to not expect only lower case characters, especially when mapping class names to file names.

Functions and Methods

Function and method names in PHP are always case insensitive. This means that PHP does not differentiate between `test()` and `Test()`, as the following example shows:

```
<?php

  function test()
  {
    var_dump(3);
  }

  function Test()
  {
    var_dump(4);
  }

  test();
  Test();

?>
```

This program fails with a fatal error at compile time:

```
PHP Fatal error:  Cannot redeclare test() (previously declared in test.php:5) in
     test.php on line 11
```

Unfortunately, this allows for sluggish programming:

```
<?php

  function test_function()
  {
    var_dump('test function');
  }

  test_function();
  Test_Function();
  TEST_FUNCTION();

?>
```

The program output shows that PHP calls the same function, regardless of the capitalization in the function call:

```
string(11) "test function"
string(11) "test function"
string(11) "test function"
```

Try to keep capitalization strict and consistent, even though PHP does not really enforce it.

Classes

Just like with function and method names, PHP also treats class names in a case insensitive way:

```
<?php

  class ClassName
  {
    ...
  }

  class className
  {
    ...
  }

  $a = new ClassName;
  $b = new className;

?>
</code php>

This program fails with a fatal error at compile time:

<code>
Fatal error: Cannot redeclare class className in test.php on line 8
```

Again, this encourages sluggish programming, because you can write class names in varying case and still refer to the same class name:

```
<?php

  class Test
  {
    public function __construct()
    {
      var_dump('class Test');
    }
  }

  $test = new test;
  $test = new Test;
  $test = new TEST;

?>
```

PHP instantiates the same class three times:

```
string(10) "class Test"
string(10) "class Test"
string(10) "class Test"
```

When mapping class names to file names, case can become an issue, since the underlying operating system might be case sensitive or case insensitive. Please refer to Chapter 3 for more information.

Files

PHP treats file names either case sensitive or case insensitive depending on the operating system PHP runs on. On Unix, file names are case sensitive, while they are case insensitive on Windows. This means that on Unix `myFile.txt` and `myfile.txt` refers to two different files, whereas on Windows both names refer to the same file.

You should never use file names that only differ in capitalization. Though this is possible and perfectly valid on Unix, you will run into trouble once you try to copy these files to Windows.

When migrating from Windows to Unix, you should make sure that your application finds all required files on the target system. This means not only checking all include and require statements, but also any function that can be used to access a file, like `fopen()` or `file_get_contents()`.

Let us try to access an existing file `test.txt` from a PHP program:

```
<?php

  $string = file_get_contents('test.TXT');

?>
```

This program will work without problems on Windows, but fail on Unix:

```
Warning: file_get_contents(test.TXT): failed to open stream: No such file or
     directory in test.php on line 3
```

Also, migrating from Unix to Windows is not always without problems related to file names, because Unix allows more special characters in file names than Windows does. Unix basically allows every special character except the directory separator (which is the forward slash `/`). Windows is more restrictive and does not allow a number of special characters. For details please refer to Chapter 3.

When you use `include`, `require`, `include_once`, or `require_once`, also read the section entitled "Dynamically Loaded Code" later in this chapter.

Name Conflicts

Computers are not very intelligent. A human that reads source code can resolve ambiguities intuitively, but computers cannot. Names and identifiers must always be unambiguous so that the computer can execute a program.

Many program languages support namespaces to make identifiers unique. Since PHP does not support namespaces until version 5.3, there is a lot of potential for name conflicts in existing PHP code.

Reserved Keywords

Reserved keywords solve a special purpose in PHP. When PHP translates the source code to something executable, the tokenizer parses the code and translates source code text into tokens. To find its way through the source code, the parser uses a number of reserved keywords as anchor points.

If a reserved keyword appears as an identifier (i.e. class, function or method name, or constant) in the source code, the parser gets confused and fails to compile the PHP program.

Though it is possible to define constants with a keyword as their name, because the identifier is enclosed in quotes, accessing this constant does not work, as the following example shows:

```
<?php

  define('case', 3);

  var_dump(case);

?>
```

The output is:

```
Parse error: syntax error, unexpected T_CASE, expecting ')' in test.php on line
    5
```

As you can see, the error occurs in line 5, where we try to output the constant, and not in line 3, where we have defined it.

Even a namespace does not protect you from conflicts with reserved keywords. You can, for example, not call a method `new()`, even though the method name is protected by the namespace of the class it is defined in:

```
<?php

  class Test
  {
    function new()
    {
      ...
    }
  }

?>
```

Running this program leads to a fatal error on compilation:

```
Parse error: syntax error, unexpected T_NEW, expecting T_STRING in test.php on
    line 5
```

Over time, more keywords are being added to PHP. If the PHP developers choose to use an identifier you are using as a new keyword, you will have to rename this identifier to make your program compile again.

Reserved Keywords:

- __CLASS__, __FILE__, __FUNCTION__, __LINE__, __METHOD__
- and, array(), as
- break
- case, class, const, continue
- declare, default, die(), do
- echo(), else, elseif, empty(), enddeclare, endfor, endforeach, endif, endswitch, endwhile, eval(), exit(), extends
- for, foreach, function
- global
- if, include(), include_once(), isset()
- list()
- new
- or
- print()
- require(), require_once(), return()
- static, switch
- unset()

- var
- while
- xor

Keywords reserved in PHP 4 only:

- cfunction
- old_function

New reserved keywords in PHP 5

- __DIR__
- __NAMESPACE__
- abstract
- catch, clone
- exception
- final
- goto
- interface, implements
- namespace
- php_user_filter, public, private, protected
- this, throw, try
- use

When parsing a PHP file fails because a keyword is used as an identifier, an error message with a token name is displayed:

```
Parse error: syntax error, unexpected T_FOREACH, expecting T_STRING in test.php
    on line 5
```

Please note that the ::-operator has the rather cryptic Hebrew token name T_PAAMAYIM_NEKUDOTAYIM. For a full list of tokens please refer to http://www.php.net/tokens. Since the above mentioned error occurs at compile time, it is rather easy to spot and fix.

To avoid name conflicts with keywords, you should prefix all class, function and constant names. For method names, using a prefix is not very nice, though:

```
<?php

  function fooSomething()
  {
    ...
  }

  class fooTest
  {
    public function fooNew()
    {
      ...
    }
  }

?>
```

So, instead of prefixing method names, you should use composite names consisting of at least two words. Using a verb and a noun like doFoo(), registerFoo() or importFoo() will also make the purpose of the method clearer.

Functions

For obvious reasons, user functions in PHP may not conflict with the names of built-in PHP functions. The problem is that PHP extensions can register function names as well, so you never really know which function names will be taken up by a certain PHP installation.

Luckily, most authors of PHP extensions stick to the rule of prefixing the function names with the extension name followed by an underscore. You should do the same with your user functions in the global namespace: use a unique prefix to avoid name conflicts. Choosing a prefix boils down to some guessing, though, because you can never know which PHP extensions might be activated in the PHP installation your application runs on. Refrain from using obvious and speaking names like `imap`, `ldap`, `odbc`, or `mysql`, as these are very likely to be names of PHP extensions.

As we have already seen, the number of internal PHP functions will differ between PHP installations. The following program will output a list of all existing PHP functions.

```
<?php

  var_dump(get_defined_functions());

?>
```

This list is split into two parts. The first, internal, shows all functions registered by the PHP code and PHP extensions, while the second one, user, shows the user functions defined by the PHP script. On my development system with some activated PHP extensions, over 1500 functions are listed.

You can also list all functions registered by a certain PHP extension:

```
<php

  var_dump(get_extension_funcs('imap'));

?>
```

Classes, Interfaces and Exceptions

As we know, the support for object oriented programming in PHP was rudimentary until the release of PHP 5. In PHP 4, objects were arrays with additional functions, while PHP 5 supports all object-oriented concepts like objects, access modifiers, interfaces, and exceptions.

Current PHP extensions use more and more object-oriented APIs, which means that an increasing number of built-in interfaces and classes are available. All built-in classes and interfaces are part of the global namespace, so name conflicts with class and interface names are possible. Since the registered classes and interfaces can differ between PHP installations, so you should use the following script to list them:

```
<?php

  var_dump(implode(’, ’, array_merge(get_declared_classes(),
       get_declared_interfaces())));

?>
```

Since exceptions are objects as well, they are listed by `get_declared_classes()`. On my PHP installation, the above script lists 128 names, some of which have a great potential for name conflicts:

- ArrayObject
- Countable
- DateTime
- Directory
- DOMElement
- DOMDocument
- FindFile
- Iterator
- Reflection
- Serializable
- tidy
- Traversable

When PHP runs on Windows, the COM extension which is enabled by default registers the classes variant, com, and dotnet.

The Standard PHP Library (SPL) defines quite a few iterators like RecursiveIterator, FilterIterator, EmptyIterator, or AppendIterator and a hierarchy of exceptions like LogicException, OutOfBoundsException, DomainException, or RuntimeException.

When using names for iterators or exceptions you should always use a prefix as mentioned above, or use the namespaces that became available with PHP 5.3. When using namespaces, you have to clearly distinguish whether you are referring to a built-in class or a class or your namespace, as the following example shows:

```
<?php

  namespace foo;

  class Exception
  {
    ...
  }

  $test = new Exception;
  var_dump($test);

  $test = new ::Exception;
  var_dump($test);

?>
```

In this example, we define a class Exception in our namespace foo. Since there is a built-in class Exception in the global namespace, we should always fully qualify the class name Exception when using it. While Exception refers to the Exception class in our foo namespace, ::Exception refers to the built-in class Exception:

```
object(foo::Exception)#1 (0) {
}

object(Exception)#2 (6) {
  ["message":protected]=>
  string(0) ""
  ["string":"Exception":private]=>
  string(0) ""
  ["code":protected]=>
  int(0)
  ["file":protected]=>
  string(8) "test.php"
  ["line":protected]=>
  int(12)
  ["trace":"Exception":private]=>
  array(0) {
  }
}
```

Fortunately name conflicts with built-in classes are easy to find, since any attempt to reuse an existing name will lead to a fatal error at compilation time:

```
Fatal error: Cannot redeclare class Exception in test.php on line 4
```

When running a lint check on your target system, you should find out about all conflicting class names, and resolve any issues by renaming classes. Keep in mind, though, that the first fatal error makes PHP stop compilation, so after you have fixed the error you have to repeat the lint check to make sure that there are no more errors in the same file.

When no more name conflicts occur at compile time, there is still a potential for name conflicts at run time, for example when you use a variable name to store a dynamically created class name. Please refer to the “Dynamic Calls” section later in this chapter for more information.

Changing a class name in PHP code is not always easy. Sometimes, it is sufficient to rename the class name with search and replace in all files of the application, but when the code has external dependencies, you may also need to repeat the same renaming other projects.

Constants

PHP itself and PHP extensions define built-in constants. Recently, PHP extensions are using class constants more and more, but there is still a large number of constants in the global namespace.

Some of the predefined constants in every PHP installation are:

- NULL
- boolean TRUE
- boolean FALSE
- the PHP version number PHP_VERSION
- the end-of-line character(s) PHP_EOL
- the Euler number M_E
- the mathematical constant M_PI

Like with functions and classes, you can never exactly tell which constants will be available in a certain PHP installation. The following program lists all constants available in your PHP installation:

```
<?php

  var_dump(get_defined_constants());

?>
```

On my system, this program lists 1170 constants!

Unfortunately, name conflicts with built-in constants are not as easy to find as name conflicts between functions or classes. If you try to use an already defined constant, PHP will show a warning that you are trying to redefine an existing constant:

```
<?php

  var_dump(PHP_SAPI);
  define('PHP_SAPI', "CGI");
  var_dump(PHP_SAPI);

?>
```

The program generates the following output:

```
string(3) "cli"
Notice: Constant PHP_SAPI already defined in test.php on line 4
string(3) "cli"
```

An existing constant cannot be overwritten, but PHP only shows a notice, so it is rather easy to overlook this error, especially when PHP is not configured to display all error messages. Also, please note that a lint check will not be sufficient to display the above notice, you will have to execute the program.

It can get even worse. If the critical `define()` statement is in one execution path of the program, a single test may not be sufficient to detect the problem. You would have to test all possible execution paths of the program, like the following example defining a constant depending on the value of `$a` shows:

```
<?php

  $a = false;

  if ($a)
  {
    define('PHP_EOL', 'nonsense');
  }

  var_dump(PHP_EOL);

?>
```

Fortunately, code like this should not occur too frequently. When migrating such code, either move the `define()` out of the branch or add an else branch with another `define()` statement.

PHP also allows for accessing constants by specifying their name as a string:

```
<?php

  var_dump(constant('PHP_EOL'));

?>
```

When you dynamically create constant names, to access them by `constant()`, you can run into similar problems like with dynamically created class names. Since you cannot find these names with search and replace, you must check every constant() call in your code.

To minimize the risk of name conflicts, you should not define constants in global namespace, but wrap a class around them, as the following example shows:

```
<?php

  class MyConstants
  {
    const EOL = "\n";
    const HELLO_WORLD = "Hello World";
  }

  var_dump(MyConstants::HELLO_WORLD);

?>
```

All constants are now protected by the namespace of the class MyConstants. Another advantage of using class constants is that you can modify constants without changing existing code by creating a subclass of MyConstants.

If you run into name conflicts with constants on migration, you should consider introducing a static class that holds all your constants. This will avoid any further name conflicts in the future.

Magic Constants, Functions, and Methods

As we have already seen earlier in this chapter, PHP has some magic constants like __FILE__ or __CLASS__. In addition, there are magic functions and methods as well, that PHP automatically calls at certain points in time.

Examples for magic methods are the constructor and destructor methods __construct() and __destruct() mentioned in the “Objects” section of this chapter. Other magic methods include __call(), __sleep() or __clone().

All magic constructs begin with two underscores. You should never use two underscores for non-magic names in a PHP program. Should you find these names when migrating, rename them throughout the whole application.

Variables

Just like in all common programming languages, functions and methods have their own scope in PHP. All variables within this scope that are not explicitly declared as global are local variables.

A local variable is only valid inside the scope it has been defined in. Variables of the same name can exist in other scopes without any conflicts. Let us look at an example:

```
<?php

  $test = 'global $test value';
  var_dump($test);

  do_something();
  var_dump($test);

  function do_something()
  {
    $test = 'local $test value';
    var_dump($test);
  }

?>
```

This program generates the following output:

```
string(18) "global $test value"
string(17) "local $test value"
string(18) "global $test value"
```

The first `var_dump()` call will output the global variable `$test`, while the second `var_dump()` call inside the `do_something()` function refers to the local variable `$test`. In the third call the value of the global variable `$test` is output again, since the function has already ended and the local variable `$test` does not exist any more.

Since functions and methods are usually not too long, name conflicts with local variables are rare. The problem are variables in global namespace. Since the global namespace spans multiple source files and all libraries as well, there is a big potential for name conflicts.

To list all variables registered in the global namespace, run the following program:

```
<?php

  var_dump(array_keys($GLOBALS));

?>
```

Add this at the bottom of your program, for example in an append file (see Chapter 7), to make sure you actually see all defined global variables. Keep in mind that the amount of global variables a program registers can be different depending on the execution paths.

The solution to avoid name conflicts, again, is to prefix all global variable names. In some configurations, though, PHP will automatically create a non-prefixed global variable from input the script receives.

Keep in mind that also libraries that you use can define global variables. If they are not prefixed, name conflicts between different libraries can occur.

Besides the problem of name conflicts, you should avoid using global variables at all, because you cannot control access to then, which means that they can be overwritten at any time.

Use constants instead to store values that do not change while the program executes. Constants, though, do not allow storing non-scalar values like arrays or objects. To work with non-scalar objects, you can create an object with non-public members, providing only accessor methods to read, but not write the values.

You can list all variables in the current scope with

```
<?php

  var_dump(get_defined_vars());

?>
```

This list includes all global variables that were imported into the scope with a global declaration.

Depending on the PHP configuration, various associative arrays are present in the global namespace that hold input data and environment variables. In current PHP versions, also the so-called superglobal variables are present. The name of a superglobal variable consists of capital letters and starts with an underscore (see the section on "Superglobals" later in this chapter).

The following script provides you with a better overview of global and superglobal variables:

```
<?php

  var_dump(array_keys(get_defined_vars()));

?>
```

On my system, this program generates the following output:

```
Array
(
    [0] => GLOBALS
    [1] => argv
    [2] => argc
    [3] => _POST
    [4] => _GET
    [5] => _COOKIE
    [6] => _FILES
    [7] => _SERVER
)
```

Components and Libraries

As PHP applications grow more complex, code from different authors is integrated. While it is rather easy to avoid name conflicts when all code has been written by the same author, the probability of name conflicts grows when code by different authors is integrated.

As we have already mentioned, prefix all names in global namespace to make them unique, or use namespaces, which however raises your minimum required PHP version to 5.3. When using namespaces, name conflicts can be resolved without modifying existing code by defining aliases, but it will probably take some time until existing libraries and components will make thorough use of namespaces.

Keep in mind that library PHP code you use does not only consist of the classes and functions that make up the public API. To reduce complexity of the public API, there might be additional classes and functions that are hidden from public view, yet are present and can cause potential name conflicts. Since this "private" code can be changed without changing the public API, there is a potential for name conflicts whenever you use a new release of a component or library.

Quite often, PHP programs load code at runtime by include or require. If a name conflict with an existing class or function occurs, usually an error message is generated:

```
Fatal Error: Cannot redeclare class ...
```

This error does not occur right away in all cases, though. Many PHP applications use autoload. The autoload handler is a magic function that is called when a non-existing class is instantiated. The autoload handler then tries to load the class. The advantage of using autoload is that you need no include statements on top of all your PHP files, because all classes and their dependencies are automatically loaded when they are required.

Most programmers put each class into one separate file. The autoload function must map the class name to a function name. If applications use the PEAR naming scheme, the file name can be calculated from the class name by replacing underscores with forward slashes and appending `.php` to the name. The class `PEAR_SomePackage_Exception`, for example, would be mapped to the file name `PEAR/SomePackage/Exception.php`.

When your application uses autoload, name conflicts will not always result in a fatal error. Let us assume your application uses a class `File`, which is already loaded. When some library also uses a (different) class `File`, autoload is not triggered because the class is already defined. Yet, an instance of the wrong `File` class will be created. Since the class name matches, this error will only show when you try to call a non-existing method or when a type hint fails:

```
<?php

  test(new File);
  function test(FileInterface $aFile)
  {
    ...
  }

?>

<?php

  // Your application

  interface FileInterface
  {
    ...
  }

  class File implements FileInterface
  {
    public function load()
    {
      ...
    }
  }

?>

<?php

  // The library

  class File
  {
    public function read()
    {
      ...
    }
  }

?>
```

Depending on the execution order, this program will either work or fail. In the latter case, you will see an error message like the following:

```
Catchable fatal error: Argument 1 passed to test() must be an instance of
    FileInterface, instance of File given, called in test.php on line 3 and
    defined in test.php on line 5
```

In this case, PHP even tells you where the class in question has been defined. It can get even worse, though:

```
Fatal error: Call to undefined method File::load() in test.php on line 4
```

In this case, PHP does not tell you where `File` has been defined, so you must search the whole source code for any definitions of a `File` class. Should the "wrong" `File` class have a method of the same name, but with different parameters, PHP will complain about wrong or missing parameters. It can be hard to find the source of this errors.

Processing Input Data

Every program processes input data and creates output data. PHP programs typically process HTTP requests that can transmit parameters as URL parameters, for example. The result generated by the PHP program is usually a HTML page.

When PHP programs are executed at the command line rather than in a web server, input data also is available in the `$argv` array. The variable `$argc` contains the number of parameters passed to the PHP program. By definition, `$argv[0]` is always the name of the called PHP script.

If PHP runs in a web server, which should be the usual case for a web application, there are various possibilities of accessing input data.

Registering Global Variables

The history of the `php.ini` setting `register_globals` reflects, in a way, the whole history of PHP. The first PHP versions were created to be more user-friendly than CGI programming with PERL, so accessing input data was made as easy as possible. That is why PHP programs used to automatically create a global variable for each `GET`, `POST`, and cookie input.

```
<?php

  // File: test.php

  var_dump($page);
  var_dump($user);

?>
```

When called via the URL `test.php?page=products&user=Stefan`, this program would generate the following output:

```
string 'products' (length=8)
string 'Stefan' (length=6)
```

Today, you can configure whether PHP registers global variables with the `php.ini` setting `register_globals`. Until PHP 4.2 `register_globals` was *On* by default, since then it is recommended to keep it *Off*. Unfortunately, this recommendation is not always being adhered to. Even today, some programs require `register_globals` to be *On*. In PHP 6, `register_globals` will be removed.

Automatically creating global variables simplifies programming at first glance, but as programs grow complex, disadvantages pile up. What happens if an input variable is overwritten, just because the program uses a global variable of the same name? How can I tell which variables a program defines, just by looking at it from the outside?

In conjunction with sloppy coding, `register_globals` can account for security problems, as the following simplified example shows:

```
<?php

  // File: test.php

  if ($username == $usr && $password == $pwd)
  {
    ...

    if ($username == 'admin') $admin = true;
  }
```

```
    ... viel Code ...

    if ($admin)
    {
      ...
    }

  ?>
```

When an attacker calls the URL `test.php?admin=1`, he will be treated as administrator, though he is not even logged in to the system. The reason for this is the URL parameter "admin" that leads to creation of the global `$admin` variable in the above program. This security problem could easily be solved by initializing all used global variables, or not using global variables at all, but still many PHP applications had similar problems in the past.

There is a simple solution to make a program work with deactivated `register_globals` setting. Just explicitly register the appropriate input as global variables. This can be done with the PHP function `import_request_variables()`, which has been available from PHP 4.0.7. You can even specify the data sources you want to create global variables from:

```
  <?php

    // File: test.php

    import_request_variables('GPC');

    var_dump($page);
    var_dump($user);

  ?>
```

The `G` stands for `GET` data, `P` for `POST` data, and `C` for cookie data. The order is important, as data registered later overwrites already registered data. By changing the order of data sources, the behavior of the application can change as well, so you should use the value of the `variables_order` setting in `php.ini`.

While using `import_request_variables()` is not the recommended way of solving the `register_globals` problem, it can act as a quick workaround to make an older program work in a current PHP environment.

Even in PHP 4, there have always been alternatives to using `register_globals`, namely global and superglobal variables, as explained below.

Long Input Arrays

PHP registers global variables for input data from the various sources. Instead of accessing a global variable, one accesses an array element. This makes it much easier to separate input from variables used by the program itself.

PHP defines the following global arrays:

- $HTTP_GET_VARS
- $HTTP_POST_VARS
- $HTTP_COOKIE_VARS
- $HTTP_SERVER_VARS
- $HTTP_ENV_VARS
- $HTTP_POST_FILES

The disadvantage of using these global arrays is that they must be declared as “global” in every scope. To overcome this limitation, the *superglobal variables* have been created.

Since PHP 5.0 you should not use the long input arrays any more, since they can be deactivated with the `php.ini` setting `register_long_arrays`. Thus it is not guaranteed that the long arrays are always present in every PHP installation. What is more, the long input arrays will not be available in PHP 6 any more.

Fortunately, it is not a big deal to modify a program using long input arrays. All you have to do is to replace all occurrences of the above listed variable names by the respective superglobal arrays (see the next section in this chapter for more information). While doing this, you can remove all global declarations of the long arrays as well.

For performance reasons, it is recommended to disable registering the long input arrays when you are planning to use them. This saves you some computing time on every PHP request.

Superglobal Variables

As pointed out in the previous section, introducing the superglobal variables to PHP was an evolutionary improvement. Since most programs access input data in various places, it is better not to have to declare input variables as global in every scope.

PHP defines eight superglobal variables, namely `$_GET`, `$_POST`, `$_COOKIE`, `$_SESSION`, `$_REQUEST`, `$_ENV`, `$_FILES` and `$_SERVER`.

The superglobal variable `$_REQUEST` combines input data from various sources. When working with sensitive input data, you should prefer `$_POST` over `$_GET` or `$_REQUEST`, because `GET` parameters can be manipulated very easily, and show up in the server log files and the browser history.

Superglobal variables are the preferred way of accessing input data in PHP. If your application uses other means of accessing input, you should consider modifying the code to use superglobal when migrating.

An even better idea is to create a class `Input` that is passed the superglobals as it's created, and is then used throughout the application to access input data. This has the advantage of decoupling your application from the actual HTTP request, which makes the program much easier to test. The `Input` object is also a good place to preprocess and filter input data to make sure that the application only works with sanitized variables.

Magic Quotes

Magic quotes are another feature that has been introduced into PHP to make a programmer's life a little easier. The idea is to automatically escape backslashes, single and double quotes and the NULL character in strings with a backslash.

This is done to avoid problems when using the strings in SQL statements, where a string containing an unescaped quote can easily lead to an illegal SQL statement, and potentially even an SQL injection attack. The same problem exists when using unescaped strings as arguments to command line calls.

The following example program illustrates the problem:

```
<?php

  // File test.php

  $sql = "SELECT * FROM test WHERE username='" . $_GET['username'] . "'";
  var_dump($sql);

?>
```

When the URL `test.php?username=O'Hara` is called with disabled magic quotes, the program will output:

```
string 'SELECT * FROM test WHERE username='O'Hara'' (length=42)
```

This SQL statement is syntactically incorrect, and can thus not be executed by the database. Magic quotes exist to resolve this issue, but the problem is that magic quotes are configurable, and do not behave the same on all PHP systems. An application cannot rely on the fact that input is always escaped (or not escaped).

This means that programs relying on magic quotes are vulnerable by SQL injection attacks if magic quotes are disabled. If, on the other hand, an application relies on magic quotes to be turned off, and explicitly escapes special characters, backslashes start to pile up in the strings. You may have seen web pages displaying multiple backslashes due to sloppy PHP programming.

To make a program portable, its behavior must be made independent from the PHP configuration. This means we have to check whether magic quotes are enabled or disabled and escape or unescape the strings accordingly.

As already mentioned earlier in Chapter 3, the various configuration settings relating to magic quotes are among the features that will be removed in PHP 6. If your application relies on magic quotes, you must modify the code when migrating.

magic_quotes_gpc

The `php.ini` setting `magic_quotes_gpc` determines whether PHP will automatically escape `GET`, `POST`, and cookie input. Since this escaping takes place before the actual PHP script starts, you cannot change `magic_quotes_gpc` at script runtime. The default value is *On*.

Today most database drivers in PHP use prepared statements that automatically take care of proper parameter escaping. This allows you to work with unescaped data in your application, which makes outputting HTML code easier.

To make an application independent from the magic_quotes_gpc, use the function get_magic_quotes_gpc() to check the setting at runtime. Depending on the result, either use addslashes() or stripslashes() on the input data:

```
<?php

  if (get_magic_quotes_gpc())
  {
    $_GET = unquote($_GET);
    $_POST = unquote($_POST);
    $_COOKIE = unquote($_COOKIE);
  }

  function unquote($aValue)
  {
    if (is_array($aValue))
    {
      return array_map('unquote', $aValue);
    } else {
      return stripslashes($aValue);
    }
  }

?>
```

Since stripslashes() only works on strings, but not on arrays, we have written a function unquote() which operates on strings and arrays. This function is recursive and can thus also handle nested arrays.

In PHP 4 magic quotes are also applied to the contents of the superglobal variable $_ENV, that contains the operating system's environment variables. You should check whether your application works with $_ENV and if necessary, change the code accordingly.

The setting magic_quotes_gpc will be removed in PHP 6.

magic_quotes_runtime

The setting magic_quotes_runtime determines whether data read from external sources like databases or flat files is automatically escaped. The default value is *Off*. You can disable magic_quotes_runtime at script runtime by calling set_magic_quotes_runtime(0). The function get_magic_quotes_runtime() allows you to retrieve the current value of magic_quotes_runtime.

Since magic_quotes_runtime will also be removed in PHP 6, you can use the following code to safely check the magic_quotes_runtime setting:

```
<?php

  if (function_exists('set_magic_quotes_runtime'))
  {
    set_magic_quotes_runtime(0);
  }

?>
```

Whether this function will be removed in PHP 6, or just modified to always return "false", has not been finally decided at the time of this writing.

magic_quotes_sybase

The SQL standard requires single quotes to be escaped by another single quote. Escaping single quotes with a backslash is non-standard behavior, but still supported by many databases. Early MySQL versions did not support SQL-compatible escaping, but only escaping with backslashes. Though today most databases support escaping single quotes by another single quote, database specific escaping is used quite often.

By enabling the php.ini setting magic_quotes_sybase, the behavior of magic_quotes_gpc changes to SQL-standard-conforming escaping of single quotes. Double quotes, backslashes, and NULL characters will not be escaped.

Please note that the name of the configuration setting is misleading, since it does not only apply to Sybase databases, but to all data and databases.

Just like the other configuration settings dealing with magic quotes, magic_quotes_sybase is problematic, because you cannot rely on a certain behavior across different PHP installations. To disable magic_quotes_sybase at runtime, use

```
<?php

  ini_set(’magic_quotes_sybase’, false);

?>
```

To explicitly escape a string according to SQL standard, you can use the following string replacement code:

```
<?

  $string = str_replace(’\’, ’\’\’’’, $string);

?>
```

Most of the time, the database driver takes care of escaping the data nowadays, especially when prepared statements and/or PDO is being used. Most database drivers offer functions for database-specific escaping, like mysql_real_escape_string() for MySQL, or sqlite_escape_string() for SQLite.

If you plan to undo the magic quoting of GET, POST, and cookie data, you should do so before turning magic_quotes_sybase off. This is because magic_quotes_sybase also modifies the semantics of the stripslashes() function: when magic_quotes_sybase is activated, stripslashes() will replace two single quotes with one single quote.

Accessing POST Data

As we have already learned, sensitive form data should always be transmitted by a POST request. When uploading files from the browser to the server, POST requests are used as well. PHP will automatically decode the binary data that is transmitted base64-encoded. After decoding the data, the uploaded file is made available to the PHP script in a temporary directory. Using the function move_uploaded_file(), this file can be copied to another directory for further processing.

Usually PHP developers do not have to deal with the technical details of a POST request. In special cases, however, it can be necessary to access the original POST data sent by the browser. PHP will make this data available in the global variable `$HTTP_RAW_POST_DATA` if the `php.ini` setting `always_populate_raw_post_data` is enabled. `$HTTP_RAW_POST_DATA` will only be filled with the original `POST` data when the mime type of the `POST` request was not multipart/form-data, which is exactly the mime type of requests used to upload files.

This means that you cannot access uploaded files via `$HTTP_RAW_POST_DATA`. You can always use the input stream to read the original POST data independently from `php.ini` settings:

```
<?php

  $raw_post_data = file_get_contents('php://input');

?>
```

The Script Name in $PHP_SELF

PHP programs often need to output links. Especially applications that tunnel all requests through one single PHP script will frequently have to construct self-referencing URLs.

In older PHP versions until PHP 4.1.0 there was a global variable `$PHP_SELF` containing the name of the currently executed script, relative to the web server's document root directory. If PHP runs in CLI mode, `$PHP_SELF` contains the absolute path of the currently executed PHP file.

If you need to know the full path of the currently executed script, you can use the magic constant `__FILE__`, that contains the script's full absolute path, or, starting with PHP 5.3, "__DIR__", that contains the directory of the running script.

Since PHP 4.1.0, `$PHP_SELF` is no longer available when `register_globals` is deactivated in `php.ini` (as discussed previously in this chapter). You can still read the same value from the `$_SERVER` superglobal variable.

If you want a quick fix without modifying the code of your application, just define a global variable `$PHP_SELF` in your application when it is not yet defined:

```
<?php
```

```
  if (!isset($PHP_SELF))
  {
    $PHP_SELF = $_SERVER['PHP_SELF'];
  }

?>
```

A better solution, however, would be to replace all occurrences of `$PHP_SELF` in the source code by `$_SERVER['PHP_SELF']`.

Error Handling

According to Murphy's Law, everything that can go wrong will eventually go wrong. Though this law was originally stated back in 1949, it definitely applies to computers. When programming, you cannot rely on everything to just work, but always have to take into account that things can and will go wrong. There are many sources of errors: users make wrong entries, the database contains nonsense values, the hard disk runs out of memory, or a whole system fails. A good program should at least handle the most common errors.

As a matter of fact, error handling can account for up to 90% of the programming effort. Especially applications requiring lots of user input require many plausibility checks. These include not only simple checks for mandatory inputs or field length, but also more complex checks like validity of a domain name or an email address.

Even when all fields have been validated successfully, this does not mean that there will be no more error when processing the data. Additional information required from the database might be missing, or the data to store may already be present in the database.

We can distinguish between errors that occur at compile time, and errors that occur at runtime. A typical error that occurs at compile time is a syntax error:

```
Parse error: syntax error, unexpected $end in test.php on line 9
```

Basically, an error message like this tells you that the PHP parser cannot compile the PHP source code to executable code, because the program does not follow the grammar rules of PHP. These errors might be a little annoying, but are usually very easy to fix.

Errors that occur at runtime are much harder to fix, since they might occur only in certain situations, depending on the input data and the state of the program.

PHP supports custom error handlers that allow you to process any runtime errors (see the section entitled "Returning References" later in this chapter). It is not possible, though, to handle compile-time errors with a custom error handler, since they occur before execution of the PHP program even starts.

Suppressing Errors With @

Too many programmers configure PHP not to display all error classes, or use the `@` operator to selectively suppress the error for certain function calls. Suppressing errors can be useful in certain situations, but as programs grow complex, ignoring errors leads to other hard to find errors in most cases, and should thus be avoided.

Please note that the `@` operator suppresses even critical errors that would otherwise end program execution. I often see code using the `@` operator to suppress warnings when non-existing files are read:

```
<?php

  var_dump(file_get_contents('non-existing file'));

?>
```

If the file does not exist in the current directory, the program issues a warning:

```
Warning: file_get_contents(non-existing file): failed to open stream: No such
    file or directory in test.php on line 3

bool(false)
```

Since the file could not be read, `file_get_contents()` returns a boolean false. Depending on the program context, a missing file is not always an error, though, so many programmers use `@` to suppress the warning:

```
<?php
```

```
  var_dump(@file_get_contents('non-existing file'));

?>
```

A better solution would be to first check whether the file exists with `file_exists()`, and then read the file. You might even have to check whether the file is readable with `is_readable()`, because an existing file need not necessarily be readable by every user.

Removing `@` in this example comes at the cost of two additional file system accesses, which can be rather slow depending on which operating system and file system is being used.

Still, I would not recommend using `@`, which is rather slow, by the way, because it uses a kind of `eval` internally. When you encounter `@` statements in the code you are migrating, you should remove them. This also makes debugging the code easier, since you will probably see additional warnings or error messages when there is a problem with your code. Keep in mind, that good code should not emit any warnings or notices, at least by default.

On production systems, you should always disable error display by setting `display_errors` to *Off* in `php.ini`. You can also do that with `ini_set()` at runtime, so that your program is independent from `php.ini` (see Chapter 7).

Storing the Previous Error Message

To ease error handling in procedural code, PHP stores the last error messages in the variable `$php_error` when `track_errors` is enabled in `php.ini`.

Though `$php_errormsg` is a global variable, the stored error message is only accessible within the scope the error occurred in. Quite often, `$php_errormsg` is used to store database error messages and later access them.

Today, PHP offers a much better mechanism to handle non-fatal errors, namely exceptions. With exceptions, you do not have to handle an error right where it occurs, but can handle it where it makes most sense in the context of your program. Another advantage of exceptions is that they make it very easy to attach arbitrary environment information to the actual error message. Instead of relying on `$php_errormsg`, you should use exceptions instead, as track_errors is disabled in most PHP installations anyway.

When run code that relies on $php_errormsg on a PHP with track_errors disabled, you will see error messages, as you are trying to access a non-existing variable. If you use $GLOBALS['php_errormsg'] instead of $php_errormsg, you will see a notice as you are trying to access a non-existing array element. A quick workaround is to activate track_errors at runtime:

```
ini_set('track_errors', 'On');
```

In the long run, you should adapt your code to use exceptions. Please note that using exceptions does not necessarily mean that you must write object-oriented code. You can use exceptions in procedural code just as well. For more information about exceptions, please refer to Chapter 8.

Configuring Error Display

A great feature of PHP is that error messages are displayed as part of the HTML output, because the position of the error message can help track down the error. While this error display is extremely useful at development time, you should always suppress error display on production systems.

Besides the fact that does not make a good impression on the user when PHP error messages screw up the page layout, displaying error messages can in fact be a serious security breach. Error messages do not only provide a potential attackers with information about path and file names, but can also inspire them to SQL injection attacks by displaying SQL statements.

You can turn on and off error display by the php.ini setting display_errors. The setting error_reporting determines which error messages PHP will output. You should always set error_reporting to the highest possible value to make sure that you see all errors, warnings, and notices your application generates.

If your application does not only output HTML pages, but also other content like images of PDF files, you should disable display_errors for these content types. Otherwise, a single PHP error message will make the file unusable. Add the following snippet of code on top of all files that generate non-HTML content to selectively suppress error display.

```
ini_set('display_errors', 'Off');
```

When migrating, I would recommend also using output buffering to make sure that no output or error messages prevent the application from sending HTTP headers later.

New And Modified Error Messages

Any error message, warning, or notice indicates a potential migration problem. Therefore it is not a good idea to suppress any error messages. The available types of error messages differ between PHP versions and new versions may add additional error classes, or reclassify existing error messages to another error class. Table 8.1 shows an overview over the PHP error classes.

In PHP 5, a new error class `E_STRICT` has been introduced. Error messages of this class indicate the use of old and deprecated code constructs. This includes calling non-static methods statically, assigning the result of a new statement by reference, or using `is_a()` instead of `instanceOf`. Since `E_STRICT` error messages occur at compile time, you cannot suppress them by setting `error_reporting` to a lower value at script runtime.

Constant	Value	Description
E_ERROR	1	A fatal error that occurs at runtime. Program execution ends.
E_WARNING	2	Warnings that occur at runtime and point to potential errors in the code. This includes, for example, the warning that a file to read has not been found. Program execution continues.
E_PARSE	4	A parse error that occurred at runtime. Since the script could not be compiled, it cannot be executed
E_NOTICE	8	Indicates a potential problem at runtime. This includes, for example, uninitialized variables. Program execution continues.

Constant	Value	Description
E_CORE_ERROR	16	Fatal error, encountered by the PHP core while starting PHP. Like with E_ERROR, program execution is stopped.
E_CORE_WARNING	32	Warning by the PHP core while starting PHP. Like E_WARNING, program execution continues.
E_COMPILE_ERROR	64	A fatal error in the Zend Engine at compile time. Like with E_ERROR, program execution is stopped.
E_COMPILE_WARNING	128	Warning by the Zend Engine. Like with E_WARNING, program execution continues.
E_USER_ERROR	256	A fatal error, generated by the user by calling trigger_error(). Like with E_ERROR, program execution is stopped.
E_USER_WARNING	512	A warning, generated by the user by calling trigger_error(). Like with E_WARNING, program execution continues.
E_USER_NOTICE	1024	A notice generated by the user by calling trigger_error(). Like with E_NOTICE, program execution continues.
E_STRICT	2048	Warnings indicating PHP code using old and deprecated code constructs. This error class exists since PHP 5.0.0. E_STRICT errors occur at compile time, the program is still executed.

Constant	Value	Description
E_RECOVERABLE_ ERROR	4096	An error occurring at runtime, triggered for example by a failed type hint. Since PHP is not put into an undefined state, program execution can continue if the error is handled by a custom error handler. If the error is not handled, program execution is stopped. Available since PHP 5.2.0.
E_ALL	8191	All error classes except E_STRICT. In PHP 6, E_STRICT will be part of E_ALL.

PHP 5.3 will have a new error class, E_DEPRECATED. These errors are generated when deprecated code constructs are used that will be removed in future PHP versions, for example passing references at runtime (see the section entitled “Passing References at Runtime” later in this chapter).

The value of error_reporting is a bit field, so you can combine the various error classes by boolean operators.

```
error_reporting(E_WARNING | E_NOTICE);
error_reporting(E_ALL & ~E_WARNING);
```

Please note that the constants are only valid in PHP source code and php.ini. When you configure error_reporting in httpd.conf or .htaccess, you will have to use the numeric values instead.

If you want to make sure that all errors will be displayed in all PHP versions, use the highest possible value 2147483647. This value sets all bits in a signed 32 bit word, and thus ensures that all errors will be displayed, even in future PHP versions.

Please note that in older PHP versions, the value of E_ALL used to be 2047, or 6143, respectively.

Custom Error Handlers

PHP allows writing a custom error handler to handle errors that occur at runtime. This error handler completely replaces PHP’s internal error handling and is independent from the php.ini setting error_reporting.

The error handler is invoked on every PHP error and must handle them appropriately:

```
<?php

  set_error_handler('error_handler');

  trigger_error('User-generated error', E_USER_ERROR);

  function error_handler($aNumber, $aString, $aFile, $aLine)
  {
    var_dump($aNumber);
    var_dump($aString);
    var_dump($aFile);
    var_dump($aLine);
  }

?>
```

In this example program, an error is triggered, then handled by the error handler. The program will output

```
int(256)
string(20) "User-generated error"
string(53) "test.php"
int(7)
still running
```

The error handler must differentiate between the various error classes, and is also responsible for quitting the program when a fatal error has occurred. This implies that custom error handlers can also be used to suppress errors, warnings, and notices. When migrating, you should consider deactivating custom error handlers to make sure that no PHP errors are suppressed.

If you use xdebug (see Chapter 7), you must use the function `xdebug_get_function_stack()` to display additional information, since using a custom error handler deactivates xdebug's extended error display.

Sometimes, it is useful to generate certain errors to test error handler or logging mechanisms. You can trigger an `E_NOTICE` by accessing an uninitialized variable:

```
<?php

  var_dump($undefined);

?>
```

To trigger an E_WARNING, you can use a scalar value as argument to a foreach loop:

```
<?php

  $test = 'test';

  foreach ($test as $item) {}

?>
```

Triggering an E_ERROR is particularly easy, since you just have to call a non-existing method or function:

```
<?php

  undefined_function_name();

?>
```

It is also rather easy to trigger an E_STRICT error. Keep in mind, though, that E_STRICT errors occur at compile time, and thus cannot be handled by a custom error handler inside the PHP program.

```
<?php

  class Test {}

  $foo =& new Test;

?>
```

When migrating to a PHP version that adds new error classes, you must adapt your error handler to handle these as well. The best way to ensure that all errors are handled is by adding a default case to the error handler's switch statement.

Exceptions

Along with all the new object-oriented features, PHP 5 has also introduced exceptions. Exceptions allow the developer to handle an error not necessarily where it has occurred, but where it fits best into the program flow. To make error handling more efficient, exceptions allow passing arbitrary environment information along with the actual error message.

It is not recommended to use empty exception handlers, as this is another way of suppressing error messages:

```
<?php

  try
  {
    ...
  }

  catch (Exception $e)
  {
  }

?>
```

Ignoring errors like this is bad programming style and should be avoided. Make sure that the code you migrate does not contain empty catch blocks. If you find empty catch blocks, either remove them along with the `try` statement, or rethrow the exception so that either another catch block handles the exception or a fatal error occurs because of an uncaught exception.

References

PHP originally was a procedural scripting language. Originally, objects in PHP 4 existed to make accessing arrays and complex data structures a little easier. Still, many programmers started to extensively use object-orientation in PHP, causing the PHP developers to add full support for object-oriented programming in PHP 5.

Looking back, OOP support in PHP 4 was extremely rudimentary. PHP 4 copied objects instead of working by reference, as one intuitively might expect. To make

OOP code in PHP 4 work, one had to use & rather extensively to force PHP to create reference instead of a copy.

PHP 5, by default, treats objects by reference. This enables us to pass around objects and create multiple references to one and the same object. As soon as an object is not referenced any more, it will automatically be deleted.

The differences in reference handling is probably the one change between PHP 4 and PHP 5 that causes the most trouble when migrating. Basically, you will have fix every object access in the code.

Sometimes, developers use references to save memory. The idea is to simply create a reference instead of copying data. Due to the way PHP internally works, however, using references can have the opposite effect, as core developer Sara Golemon explains in her blog entry[1]. The bottom line is that you should trust PHP to work efficiently instead of extensively using references.

The PHP 4 Compatibility Mode

To make the migration from PHP 4 to PHP 5 easier, PHP 5 has a compatibility mode that can be enabled by the `php.ini` setting `zend.ze1_compatibility_mode`. If enabled, PHP 5 treats objects like PHP 4, which means that, by default, objects are copied. The default value is *Off*, and probably no PHP 5 program will work properly when the compatibility mode is enabled.

I would strongly recommend not using the compatibility mode, except perhaps for limited amount of time when running tests. Rumors are that the compatibility mode does not work properly anyway, since many modifications that have been made to PHP just ignore the `zend.ze1_compatibility_mode` setting. If you really need PHP 4 behavior, use PHP 4.

Should you encounter code that runs in compatibility mode, you must disable this setting, then modify the code so that the result is the same like when running in compatibility mode.

[1] Golemon, Sara: You're being lied to, 2007, `http://is.gd/u3W`

Creating A Reference

To create an object of a certain class, the new statement is used. It expects one argument, namely the name of the class to instantiate. It is also possible to use a variable for the class name, making object creation in PHP very flexible.

Usually, an object (reference) created by new will be assigned to a variable:

```
<?php

  $a = new Test;

?>
```

PHP 4 would actually create the object, then create a copy, and discard the original object since no reference to it exists any more. This is unnecessary effort, thus in PHP 4 you should explicitly create a reference:

```
<?php

  $a =& new Test;

?>
```

$a now holds another reference to the new object. It is also possible to write

```
<?php

  $a = &new Test;

?>
```

Explicitly creating a reference is no more necessary in PHP 5, and will result in an E_STRICT error:

```
Strict Standards: Assigning the return value of new by reference is deprecated
     in test.php on line 3
```

Please note that the code will still be executed. At first glance, it does not seem critical to copy an object right after creating it, then deleting the original. In fact, the problem starts when in the object constructor a reference to the object itself (`$this`) is passed to another object. As soon as a copy of the object is created, two instances of the same object exists. The resulting errors can be very hard to find.

If you do not want to create a copy of the object for good reason (see the section entitled "Copying Objects" later in this chapter), you can replace `=&` with `=` in PHP 5 without altering the program's behavior. Please note that the modified program will probably not work properly in PHP 4 any more.

Passing References

A function or method can have one return value at most, but can accept more than one parameter. Since every function has its own scope, modifications made to the parameters will not affect the outer scope. When you pass a reference to a function, the scope isolation is breached. By modifying a variable inside a function, the value also changes in the outer scope.

```
<?php

  $a = array(1, 2, 3, 4);
  var_dump($a);

  test($a);
  var_dump($a);

  function test(&$a)
  {
    unset($a[0]);
  }

?>
```

This program will output

```
array(4) {
  [0] => int(1)
  [1] => int(2)
  [2] => int(3)
```

```
    [3] => int(4)
  }

  array(3) {
    [1] => int(2)
    [2] => int(3)
    [3] => int(4)
  }
```

As we can see, the function has changed the array `$a` in the global scope. This is called a side effect.

When working with objects, side effects are desired, and usually are implied when calling a method. When working with scalar values and objects, you should only rely on side effects when this prevents you from copying large amounts of data, or when you need to pass back certain elements of a data structure, for example a DOM tree.

Since PHP 5 treats objects by reference, you do not have to explicitly create a reference to an object any more. In fact, you can remove any surplus address operators (&) in the source code.

If you pass data to a function by reference, you can circumvent the limitation of just one return value. In that case, you deliberately use a side effect to pass back data from the function.

When references are not used to prevent copying of data, or to pass back data, you should not use references. By isolating the different scopes from each other, you can save a lot of time searching for bugs when inadvertently changing a reference inside a function.

Returning References

To return a reference from a function or method, you must add an ampersand before the function name in the function declaration:

```
<?php

  class Test
  {
    protected $foo = ’foo’;

    function &returnReference()
    {
      return $this->foo;
    }
  }

  $test = new Test;

  $var =& $test->returnReference();
  $var = ’bar’;

  var_dump($test);

?>
```

The program will output:

```
object(Test)#1 (1) {
  ["foo:protected"] => &string(3) "bar"
}
```

As we can see, the protected member `$foo` in the class `Test` has been changed through the reference in `$var`, and without using the proper accessor method! Please note that to make this example work, you must assign the return value by reference as well.

Code like this is definitely not good programming style. I would strongly recommend avoiding such hacks, as the code is extremely hard to read, and even harder to debug.

It is generally only possible to pass variables by reference, not expressions. If you try to pass an expression by reference, newer PHP versions will issue a warning:

```
<?php

  function &return_reference()
  {
    return 42;
  }

  var_dump(return_reference());

?>
```

This program will output:

```
Notice: Only variable references should be returned by reference in test.php on
    line 5

int(42)
```

The official PHP documentation[2] contains various examples of problematic code that (ab)uses references. In real projects, such code should be rare, but should you encounter a reference mess when migrating, it is probably best to modify the code to work without references.

When existing code uses references, consider replacing them with objects. Since objects are always treated by reference, you will not have to worry about one ampersand more or less.

Passing References At Runtime

Older PHP versions allowed the programmer to decide at runtime whether to pass a parameter by value or by reference:

```
<?php

  $a = array('imagine a lot of data here');
  var_dump($a);
  test(&$a);
  var_dump($a);

  function test($x)
  {
    $x = array_merge($x, $x);
  }

?>
```

This program looks deceptively easy. Please note that `test()` contains no return statement, and the function's return value is not re-assigned to `$a`.

[2] PHP Documentation Group: PHP: Changes in reference handling, 2008, http://www.php.net/manual/en/migration51.references.php

Depending on the `php.ini` setting `allow_call_time_pass_reference` this program will issue a `E_WARNING` at compile time, making this error impossible to suppress, even if you use the following code at runtime:

```
error_reporting(E_NONE)
```

This warning will always be displayed when `error_reporting` in `php.ini` includes `E_WARNING`:

```
Warning: Call-time pass-by-reference has been deprecated; If you would like to
     pass it by reference, modify the declaration of [runtime function name]().
          If you would like to enable call-time pass-by-reference, you can set
               allow_call_time_pass_reference to true in your INI file in test.php
                on line 14
```

The solution to this problem is not to set `allow_call_time_pass_reference` to *Off*, because this setting will be removed in PHP 6. Move the ampersand to the function declaration instead:

```
<?php

  $a = array('imagine a lot of data here');
  var_dump($a);
  test($a);
  var_dump($a);

  function test(&$x)
  {
    $x = array_merge($x, $x);
  }

?>
```

Now the parameter `$x` will always be passed to `test()` by reference, so it is clear that any modifications you make to `$x` inside `test()` will affect the outer scope.

Copying Objects

While PHP 4 copied objects by default, you must use the `clone` statement to create a copy of an object in PHP 5, since an assignment with = would just create another reference to the same object.

```
<?php

  $a = new Test;
  $b = $a;
  $c = clone $a;

  var_dump($a);
  var_dump($b);
  var_dump($c);

  class Test
  {
    ...
  }

?>
```

The program will output the following

```
object(Test)#1 (0) {
}
object(Test)#1 (0) {
}
object(Test)#2 (0) {
}
```

As we can see, = has created another reference to `$a`, while `clone` has created a copy of the object. If you migrate PHP 4 code or code that runs in compatibility mode (see earlier in this chapter), you must replace every occurrence of = by a `clone` statement and remove the ampersand from all =& assignments in turn.

When cloning an object, you might also have to clone all related objects, depending on the business logic. You can implement the magic method `__clone()` to do this.

Magic Constants

Since PHP 4.0.2 the magic constant `__FILE__` contains the absolute path of the currently executed file. In older PHP versions, in some situations `__FILE__` would contain the relative path. Since you will probably not encounter a PHP installation older than 4.0.2, this problem should not affect you any more.

There is another catch, though. On Windows, `__FILE__` returns a path containing backslashes as directory separator:

```
<?php

  var_dump(__FILE__);

?>
```

On Unix, this program might output:

```
string(21) "/home/Steve/test.php"
```

while on Windows, the result could be:

```
string(30) "C:\www\migration\code\test.php"
```

On Windows, the path does not only contain a drive letter, but also the backslash as directory separator. As long as you only work on Windows, this is not a problem, as long as you escape each literal backslash with another backslash in strings.

To keep your application portable so that it also runs on Unix without modifications, you should replace the backslashes by forward slashes, so that you can rely on paths to always contain forward slashes as directory separator.

The following example shows you can encapsulate `__FILE__` in a function:

```
<?php

  $filename = get_script_filename(__FILE__);

  function get_script_filename($aFilename)
  {
      if ("\r\n" == PHP_EOL) return str_replace("\\","/", $aFilename);
      return $aFilename;
    }

?>
```

Instead of using `__FILE__`, you now use `get_script_filename()`. If PHP runs on Windows, the function replaces backslashes by forward slashes. As already mentioned above, the literal backslash must be escaped by another backslash in the `str_replace()` statement.

It is important to only replace backslashes on Windows, since backslashes are allowed characters in filenames on Unix. To detect the operating system, we use a trick. By looking at the value of the built-in constant `PHP_EOL`, we can quickly detect Windows operating systems. The constant `PHP_OS` returns a string that is not always easy to parse, so using `PHP_EOL` is a lot easier and more reliable.

Altered Behavior of PHP Functions

As PHP evolves, the behavior of some PHP functions is altered, mostly to correct errors of the past. Though the changes do not affect every PHP program, it is a good idea to use the wrapper functions shown below:

array_merge()

The function `array_merge()` merges one or more arrays. While PHP 5 expects all parameters passed to the function to actually be arrays, PHP 4 silently converts non-array parameters to arrays:

```
<?php

  $a = array(1, 2, 3);
  $b = 4;

  var_dump(array_merge($a, $b));

?>
```

On PHP 4, this program outputs:

```
array(4) {
  [0] => int(1)
  [1] => int(2)
  [2] => int(3)
  [3] => int(4)
}
```

On PHP 5, the program issues a warning, and the result is empty:

```
Warning: array_merge(): Argument #2 is not an array in test.php on line 3

NULL
```

On PHP 4, the second parameter could even be empty, as the next example shows:

```
<?php

  var_dump(array_merge(array(1, 2, 3), NULL));

?>
```

On PHP 4, this program outputs

```
array(3) {
  [0] => int(1)
  [1] => int(2)
  [2] => int(3)
}
```

On PHP 5, again, a warning is issued and the result is empty:

```
Warning: array_merge(): Argument #2 is not an array in test.php on line 3

NULL
```

To avoid altered application behavior when migrating, you can write a wrapper function and replace all occurrences of `array_merge()` by a call of the wrapper function.

Since `array_merge()` can take a variable amount of parameters, this wrapper function is a little complicated:

```
<?php

  function array_merge_compat()
  {
    $num = func_num_args();

    if (0 == $num) return NULL;
    if (1 == $num) return func_get_arg(0);

    $result = func_get_arg(0);

    for ($i = 1; $i < $num; $i++)
    {
      $arg = func_get_arg($i);

      if (NULL == $arg) continue;

      if (is_array($arg))
      {
        $result = array_merge($result, $arg);
      } else {
        $result = array_merge($result, array($arg));
      }
    }

    return $result;
  }

?>
```

ip2long()

The function `ip2long()` converts an IP address given in the common format of four numbers separated by a dot to a 32 bit number. On PHP 4, the function would return *-1* instead of *false* if the string passed in was no valid IP address. PHP 5 returns *false* instead. So let us try to convert an invalid IP address:

```
<?php
```

```
  var_dump(ip2long('127.0.0.'));

?>
```

In PHP 4, this program outputs:

```
int(-1)
```

while in PHP 5, the usual error code is returned:

```
bool(false)
```

Replace any checks for *-1* in your application by a type-safe comparison with false:

```
<?php

  if (false === ip2long(...)) ...

?>
```

strrpos()

The function `strrpos()` finds the last occurrence in a string. While PHP 4 only searched for a single character, PHP 5 will search for the whole string instead of its first character.

```
<?php

  $haystack = 'hello world';
  $needle = 'lo';

  var_dump(strrpos($haystack, $needle));

?>
```

In PHP 4, the output is:

```
int(9)
```

In PHP 5, the output is:

```
int(3)
```

To make sure that the application's behavior is not altered, you should secure every call to `strrpos()` with a `substr()` statement returning the first character of the string to search for. Again, let us write a wrapper function:

```
<?php

  function strrpos_compat($aHaystack, $aNeedle, $aOffset = 0)
  {
    return strrpos($aHaystack, substr($aNeedle, 0, 1), $aOffset);
  }

?>
```

strripos()

Similar to `strrpos()`, `strripos()` will find the last occurrence in a string, but perform a case-insensitive search. Like with `strrpos()`, PHP 5 will look for the whole string, not only the first character.

Let us write another wrapper function to make sure the application's behavior is not altered:

```
<?php

  function strripos_compat($aHaystack, $aNeedle, $aOffset = 0)
  {
    return strripos($aHaystack, substr($aNeedle, 0, 1), $aOffset);
  }

?>
```

strtotime()

The function `strtotime()` tries to parse a date description in English and converts the result to a Unix timestamp. PHP 4 would return *-1* to indicate an error, while PHP 5 will return *false*.

Just like with *ip2long()*, you should modify any checks for *-1* in conjunction with calls to *strtotime()* accordingly.

Classes

Over time, PHP has become more and more strict for OOP code. Depending on which PHP version you are migrating from, errors might occur on the target system, even though the code used to work fine on the original system.

Static Methods And Dynamic Calls

A class in PHP can not only contain members and methods bound to an object instance, but also static members and methods. Static members and methods do not require an object instance to be used. You can view static members as semi-global variables inside a class.

Static methods are often used to group library functions together in a class. To call a static method, you must specify the class name followed by a double colon and the method name:

```
Classname::methodName();
```

In a static method, the special variable `$this` is not defined, since no object instance exists. In older PHP versions, dynamic methods could be called statically as well:

```
<?php

  class Test
  {
    public function foo()
    {
      ...
    }
  }

  Test::foo();

?>
```

Newer PHP versions will issue an `E_STRICT` warning:

```
Strict standards: Non-static method Test::foo() should not be called statically
    in test.php on line 11
```

In PHP 6, statically calling dynamic methods will result in a fatal error.

Abstract Private Methods

In PHP versions from 5.0.0 until 5.0.4 abstract methods could be declared as private. Since an abstract method must be implemented in a subclass, but on the other hand a private method is not available in the subclass, abstract private methods do not make much sense. Current PHP versions will fail with an error if you try to define an abstract private method:

```
Fatal error: Abstract function Test::foo() cannot be declared private in test.
    php on line 7
```

When you encounter this error when migrating, either move the method to the subclass, declare it as protected, or do not declare it as abstract.

Abstract Static Methods

In PHP 5.0 and 5.1 it was possible to declare abstract methods as static. Since a static method is bound to a class, and an abstract method must be implemented in a subclass, it does not make much sense to declare an abstract method as static as well.

```
<?php

  abstract class Test
  {
    abstract static function bar();
  }

?>
```

Current PHP versions will complain with an `E_STRICT` error:

```
Strict standards: Static function Test::bar() should not be abstract in test.php
     on line 5
```

If you encounter this error, either move the static method to the subclass, or do not declare it as abstract.

Modified Method Signature in Derived Classes

Until PHP 5.0 it was possible to modify the signature of a method in a subclass. The signature of a method is the number and type of its parameters, and whether parameters are to be passed by reference or by value.

Newer PHP versions require signatures of derived classed to be compatible to the function signature in the base class. “Compatible” means that you can add additional parameters, for example:

```
<?php

  class Test
  {
    function doSomething($aParameter)
    {
      return array('one', 'two');
    }
  }

  class ModifiedTest extends Test
  {
    function doSomething($aParameter, $aSecondParameter = '')
    {
      return array('one', 'two', 'three');
    }
  }

  $test = new ModifiedTest;
  var_dump($test->doSomething(3, 5));

?>
```

This program works and outputs:

```
array(3) {
  [0]=>
  string(3) "one"
  [1]=>
  string(3) "two"
  [2]=>
  string(5) "three"
}
```

If the method in the subclass has fewer parameters, an `E_STRICT` error occurs:

```
<?php

  class Test
  {
    function doSomething($aParameter, $aSecondParameter = '')
    {
      return array('one', 'two');
    }
  }

  class ModifiedTest extends Test
  {
    function doSomething($aParameter)
    {
      return array('one', 'two', 'three');
    }
  }

  $test = new ModifiedTest;
  var_dump($test->doSomething(3, 5));

?>
```

This program will output:

```
Strict standards: Declaration of ModifiedTest::doSomething() should be
    compatible with that of Test::doSomething() in test.php on line 19

array(3) {
  [0]=>
  string(3) "one"
  [1]=>
  string(3) "two"
  [2]=>
  string(5) "three"
}
```

Sometimes, you can avoid these error messages by changing the parameter order. If this is not possible, consider passing certain parameters to the object constructor or a special accessor method instead of to the method itself. This requires you to check whether all required parameters are actually set. If this is not the case, you can throw an exception:

```
$foo must be set before calling bar(). Use the setFoo() method.
```

This may not be good style, but sometimes helps you to avoid making changes to the inheritance tree or the parameter order.

Objects

As "real" OOP was introduced with PHP 5, there are some differences in object handling between PHP 4 and PHP 5.

Constructor

When objects are created, they must be initialized. A magic method, the constructor, is automatically called when an object is instantiated. In PHP 5, constructor methods were named like the class:

```
<?php

    public function Test()
    {
      var_dump('Old style Test constructor');
    }
  }

  class myTest extends Test
  {
    public function myTest()
    {
      parent::Test();

      var_dump('Old style myTest constructor');
    }
  }

  $test = new myTest;

?>
```

This program will output:

```
string(26) "Old style Test constructor"
string(27) "Old style myTest constructor"
```

When changing the inheritance tree, calls of the parent constructor must be changed. This is unnecessary work and rather error-prone. Since PHP 5 constructors are always named `__construct()`, so you can always call the parent constructor by `parent::__construct()`, even when the base class changes at some point in the future.

PHP still supports the old constructors to ensure backwards compatibility. Mixing both constructor types, however, will earn you an `E_STRICT error`, and only the new constructor will be executed:

```
<?php

  class Test
  {
    public function __construct()
    {
      var_dump('New style Test constructor');
    }
    public function Test()
    {
      var_dump('Old style Test constructor');
    }
  }

  $test = new Test;

?>
```

This program will output:

```
Strict Standards: Redefining already defined constructor for class Test in test.
    php on line 10

string(26) "New style Test constructor"
```

A migration is the perfect time to change your code to use all new constructors to remove one more potential source of errors.

Destructors

Nothing lasts forever. Like anything else, PHP objects cease to exist at some point. When this happens, another magic method, the destructor `__destruct()` will automatically be called. The destructor is the right place to close database connections and files, or persist the object.

PHP 4 did not support destructors, but some code like PEAR uses simulated destructors by registering a shutdown handler and keeping track of all instantiated objects. If you must migrate code that uses simulated destructors, you should modify the code to use the proper magic destructor function. This may change you application's behavior, though, as it is not guaranteed that the destructors are called in the same order.

In PHP 4, you could destroy objects by setting `$this` to *NULL*:

```
<?php

  class Test
  {
    function destruct()
```

```
      {
         $this = NULL;
      }
    }

    $test = new Test;
    $test->destruct();
    var_dump($test);

  ?>
```

In PHP 4, this program outputs:

```
  NULL
```

In PHP 5, redefining the `$this` it is not allowed any more (see the section on “Name Conflicts with $this” later in this chapter), so the result is a fatal error:

```
  Fatal error: Cannot re-assign $this in test.php on line 7
```

Redefining Class Constants

In PHP 5.0, you could redefine class constants:

```
  <?php

    class Test
    {
      const foo = ’foo’;
      const foo = ’bar’;
    }

    var_dump(Test::foo);

  ?>
```

Since PHP 5.1, this is a fatal error:

```
  Fatal error: Cannot redefine class constant Test::foo in test.php on line 6
```

To fix this problem, remove the first declaration, or move the second declaration to a subclass. It is still possible to change an existing declaration in a subclass, but not in the same class.

instanceOf instead of is_a()

The function `is_a()` checks whether an object has a certain class, or is derived from this class:

```
<?php

  class Test
  {
    ...
  }

  class ModifiedTest extends Test
  {
    ...
  }

  $t1 = new Test;
  $t2 = new ModifiedTest;

  var_dump(is_a($t2, 'Test'));

?>
```

In PHP 5 the language construct `instanceOf` has been introduced, which should be used instead of `is_a()`. When you use `is_a()` in current PHP versions, an `E_STRICT` error will be displayed:

```
Strict standards: is_a(): Deprecated. Please use the instanceof operator in test
    .php on line 18
```

To fix this problem, replace all occurrences of `is_a()` by `instanceOf`:

```
<?php

  interface Testable
  {
    ...
  }

  class Test implements Testable
  {
    ...
  }

  class ModifiedTest extends Test
  {
    ...
  }

  $t1 = new Test;
  $t2 = new ModifiedTest;

  var_dump($t1 instanceof Test);
  var_dump($t2 instanceof Testable);

?>
```

Keep in mind that - contrary to the `is_a()` function - the class name must not be quoted. The program will output:

```
bool(true)
bool(true)
```

Name Conflicts with $this

Inside an object instance, `$this` refers to the object instance. Though it is possible to write procedural code that uses `$this` as a parameter, you should only use this variable name in the object-oriented context. It will make your code much easier to read.

```
<?php

  $test = new Test;
  $test->foo(3);
  class Test
  {
    function foo($this)
    {
      var_dump($this);
    }
  }

?>
```

In PHP 4, this program executes without an error and outputs

```
int(3)
```

In PHP 5, the program quits with a fatal error:

```
PHP Fatal error: Cannot re-assign $this in test.php on line 5
```

The solution to this problem is simple: rename `$this` to avoid the name conflict.

Redefining $this

Older PHP versions allowed redefining `$this` inside a method. This effectively changed the class of the object instance:

```
<?php

  class Something {}

  class Test
  {
    function doTest()
    {
      $this = new Something;
    }
  }

  $test = new Test;
  $test->doTest();
  var_dump($test);
```

```
?>
```

In PHP 4, this program works and outputs:

```
object(something)(0) {
}
```

In PHP 5, the program quits with a fatal error:

```
Fatal error: Cannot re-assign $this in test.php on line 9
```

Instead of redefining `$this`, create another object with new and return a reference to it.

Comparing Objects

There are two ways to compare objects. When using the == operator, PHP will check whether both object have the same class and the same members. Using the strict comparison ===, PHP will check whether both object references actually point to the same object instance.

The non-strict comparison with == has been changed in PHP 5.2 to recursively compare all members. When encountering circular references, PHP can fall into an endless loop. Fortunately, this will be detected:

```
<?php

  class Test
  {
    public $test;
  }

  $t1 = new Test;
  $t2 = new Test;

  $t1->test = $t2;
  $t2->test = $t1;

  var_dump($t1 == $t2);
```

```
?>
```

In PHP 5.2, this program will output the following error message:

```
Fatal error: Nesting level too deep - recursive dependency? in test.php on line
    13
```

In older PHP 5 versions, or PHP 4, the program will work perfectly and output:

```
bool(false)
```

A simple solution to this problem, if the application logic allows it, is to replace == by ===. If this is not possible, you may have to create `compare()` method that compares the relevant members, but avoids the endless recursion.

Dynamic Calls

PHP is a very dynamic language. Since PHP is not compiled like Java, there are various interesting possibilities to generate names of classes or functions at runtime.

Names created at runtime are difficult to change. You cannot find them easily by searching through the source code, thus there is a high probability of forgetting one or more occurrences when you change a name. So when migrating, there are some code constructs you should watch out for. They may need extra care.

System Calls

Calling external programs from PHP is very powerful, but creates a dependency of the application to an external program. This external program must be present on the target system, possibly in a certain path, and version and language should be identical to make sure the output stays the same (see Chapter 3 for more information).

There are various possibilities of executing an external program. Table 8.2 shows an overview.

PHP Command	Explanation
‘ (Backtick Operator)	Runs an external program, just like in Unix shell scripts
exec()	Runs an external program
passthru()	Runs an external program and displays its output
shell_exec()	Runs an external program in a shell and passes back its output as a string. Running an external program in a shell has the advantage that the usual environment variables are available to it.
system()	Runs an external program and displays its output
popen()	Connects to a started process or program. The connection can be used to read data from the program, or send data to the program
proc_open()	Like popen(), but allows a greater degree of control over the executed program, as popen() only allows setting the working directory and environment variables.

When migrating, you should search your application for all of these PHP commands and check the external dependencies they create. Since executing commands with unescaped parameters can be risky, you should take extra caution not to open up a security hole while migrating.

Class Names

Object-oriented applications instantiate objects at runtime. The names of the classes to instantiate are often calculated at runtime based on user input, or the state of the application. PHP makes this very easy by allowing you to specify the class name as a variable in the new statement:

```
<?php

  $class = ’Test’;
  $obj = new $class;

?>
```

This program behaves identically to the following program:

```
<?php

  $obj = new Test;

?>
```

In real-life code, to implement plugins, statements like the following are used:

```
<?php

  $class = 'Plugin' . $_GET['command'] . 'Command';
  $obj = new $class;

  $obj->execute();

?>
```

Please note that this is a minimal example without any security. The point to be made is that by using dynamically created class names, it becomes impossible to change the name of a class just by replacing all occurrences of the old name in the source code. Instead, you will have to look at each new statement and figure out which class names it uses.

Renaming a class can result in modified commands or URLs. If that happened, I would suggest creating a wrapper class that maps the old command or URL to the new one. This hides the renamed class from the user interface, and existing tests and documentation will not have to be adapted.

The call_user_func() Family

The `call_user_func()` family of functions is another example for the very dynamic nature of PHP. These functions allow you to call a function or method, passing the name as a parameter. This allows you to generate the name of a function to call at runtime, which can sometimes replace a rather long switch statement.

Another possibility is to store function or method names in variables. Also watch out for these kinds of statements when migrating code:

```
$function();

$object->$method();
```

Dynamic calls with `call_user_func()` and `call_user_func_array()`, respectively, will always pass parameters by value. It is not possible to pass in references, thus you cannot affect the outer scope as shown earlier in this chapter from dynamically called functions.

To call a function, pass its name as the first parameter:

```
<?php

  function doSomething()
  {
    var_dump('doing something');
  }

  call_user_func('doSomething');

?>
```

This program will output:

```
string(15) "doing something"
```

For historic reasons, the two methods `call_user_method()` and `call_user_method_array()` to call methods exist. Since PHP 4.1, these methods are deprecated, however.

```
<?php

  class Test
  {
    public function doSomething()
    {
      var_dump('doing something');
    }
  }
  $test = new Test;
  call_user_method('doSomething', $test);

?>
```

This program will output:

```
Strict standards: Function call_user_method() is deprecated in test.php on line
    12

string(15) "doing something"
```

You can easily fix this by replacing `call_user_method()` with `call_user_func()`. Use an array as the first parameter, with the first element being an object reference and the second element the name of the method to call:

```
<?php

  class Test
  {
    public function doSomething()
    {
      var_dump('doing something');
    }
  }

  $test = new Test;
  call_user_func(array($test, 'doSomething'));

?>
```

As expected, this program will not issue an `E_STRICT` error any more:

```
string(15) "doing something"
```

You can call static methods as well, by using a class name instead of an object reference as the first array element:

```
<?php
```

```
  class Test
  {
    public static function doSomething()
    {
      var_dump('doing something');
    }
  }

  call_user_func(array('Test', 'doSomething'));

?>
```

If you try to call a non-static method statically, an `E_STRICT` error will be generated, as we discussed earlier in this chapter.

Dynamically Loaded Code

Most PHP applications do not only consist of one source file, but of a number of files. This makes managing the source code easier, since each file can be edited independently from the others.

For object-oriented code, it is recommended to put each class into one file. This allows for selective loading of classes, for example with an autoload handler.

In PHP 4, you could load function definitions by include or require multiple times without an error. In PHP 5, trying to include or require a class or function that has already been defined leads to a fatal error:

```
Fatal error: Cannot redeclare class Test in test.php on line 4
```

For functions, the error message will tell you in which file the function has been previously defined:

```
Fatal error: Cannot redeclare test() (previously declared in test.php:3) in test
     .php on line 28
```

If you encounter this error, you should reorganize the code so that included files only contain code inside functions and classes, and always load them with `include_once` or `require_once`. This way, PHP will ensure that every file is loaded only once.

Should you work with code outside functions and classes, as it is often the case with templates, make sure not to define any functions or classes in these files. Then, you can load the code with `include` and `require` as often as you need, without running into danger of a fatal error due to already defined functions or classes. It should be noted that this is not really good programming style. I would recommend encapsulating the code in a function, load this function once, and call it multiple times.

When PHP code uses new language features like interfaces, classes must be defined before they are used. Though PHP programs are compiled before they are executed, the parser cannot resolve this type of forward references, as the following example shows:

```
<?php

  class Test implements Testable
  {
    ...
  }

  interface Testable
  {
    ...
  }

  $test = new Test;
  var_dump($test);

?>
```

This program works fine:

```
object(Test)#1 (0) {
}
```

Let us now move the new statement before the class definition:

```
<?php

  $test = new Test;
  var_dump($test);
```

```
    class Test implements Testable
    {
      ...
    }

    interface Testable
    {
      ...
    }

  ?>
```

Now the program fails:

```
  Fatal error: Class 'Test' not found in test.php on line 2
```

To avoid this error, you must stick to the “define before use” rule. An easy way to do this is to use `autoload` and put each class and interface into an individual file. PHP will then take care of loading all required code in the correct order.

eval()

The `eval()` function allows you to execute PHP code that is stored in a string. Quite often, the function is being referred to as “evil eval”, because executing arbitrary code at runtime can be a big security risk.

In fact, `eval()` statements should not be necessary at all. You can always create a file with the PHP code to execute, and just include this file. This does not resolve the security issues, but you can at least avoid the risk of an attacker managing to execute arbitrary code by injecting it into your application.

Hard-coded `eval()` statements have no advantage over regular PHP code. By using output buffering, which can also be layered, you can usually solve all problems more efficiently.

Little Beastlinesses

This chapter lists various problems that you can encounter when migrating PHP code. In the given examples, the problems by themselves may not look too scary, but in the context of a big applications, small problems quickly become major show-stoppers that keep you busy with debugging for far too long.

unset() and Strings

The function `unset()` can delete existing variables or array elements. To delete array elements, specify their key:

```
<?php

  $a = array(1 => 'one', 2 => 'two');
  unset($a[1]);
  var_dump($a);

?>
```

This program will output:

```
array(1) {
  [2] => string(3) "two"
}
```

The same syntax allows read access of single characters of a string. The string index is also called offset in that case:

```
<?php

  $string = 'hello world';

  var_dump($string[0]);
  var_dump($string[3]);

?>
```

This program will output:

```
string(1) "h"
string(1) "l"
```

It has never been possible, though, to delete characters from a string using `unset()`. While PHP 4 had silently ignored the attempt, PHP 5 bails out with a fatal error:

```
<?php

  $string = 'hello world';
  unset($string[0]);
  var_dump($string);

?>
```

In PHP 4, this program outputs:

```
string(11) "hello world"
```

In PHP 5, it fails:

```
Fatal error: Cannot unset string offsets in test.php on line 5
```

To avoid this, do not use `unset()` on a string. You can use the following check to do this:

```
if (!is_string($var)) unset($var[0]);
```

Errors When Sending HTTP Headers

HTML output created by PHP running in a web server consists of two parts, the HTTP headers and the HTML body. Once a PHP script has started to create output, no more HTTP headers can be sent, unless output buffering is used. If output was already started, trying to send a header will result in a warning:

```
Warning: Cannot modify header information - headers already sent by (output
     started at test.php:50) in test.php on line 43
```

This warning can be triggered by any PHP function that send a HTTP header. These are:

- session_start()
- setcookie()
- setrawcookie()
- header()

A single blank character, for example following the closing PHP tag ?> in an included file, is enough to ruin your change of sending headers. Also, whitespace before the opening PHP tag <?php or in auto prepend files, or any PHP error message, warning or notice, will do the trick.

Output is even started when your source code is UTF-8 encoded and has a byte order mark (see Chapter 3 for more information).

To avoid these problems, use output buffering as described earlier in this chapter. This also has the advantage that you can discard the whole output buffer and send an error page if a fatal error has occured. Output buffering cannot fix a UTF-8 BOM, since it has been output before the PHP script even starts.

Date and Time Functions

When your application uses date functions, you must add the php.ini setting date.timezone in php.ini, set an environment variable TZ, or add a call to date_default_time_zone() to your script. If no timezone is set, the script will issue a warning and try to guess the correct time zone:

```
<?php

  var_dump(date('dmY'));

?>
```

Without a timezone set in php.ini or in the environment variable TZ, this program will output:

```
Strict standards: date(): It is not safe to rely on the system's timezone
                  settings. Please use the date.timezone setting, the TZ
                  environment variable or the date_default_timezone_set()
```

```
                    function. In case you used any of those methods and you
                    are still getting this warning, you most likely
                    misspelled the timezone identifier. We selected
                    'Europe/Paris' for '1.0/no DST' instead in
                    test.php on line 3
```

The same warning will be output when you try to set a non-existing timezone. The PHP manual lists (at `http://www.php.net/manual/de/timezones.php`) all supported time zones, which you can also view by calling `timezone_identifiers_list()`. Where I live, I use:

```
date.timezone = Europe/Berlin
```

in `php.ini`, or call

```
date_default_time_zone('Europe/Berlin')
```

when initializing the application.

Modulo Division

As we all know, a division by zero is undefined. The *Modulo* operator calculates the remainder of a division. When modulo dividing by zero, the remainder is undefined as well. Older PHP versions just returned the error code *false*:

```
<?php

  var_dump(10 % 0);

?>
```

If you continue the program without checking the result, hard-to-find follow-up errors can occur, so newer PHP versions will issue a warning:

```
Warning: Division by zero in test.php on line 3

bool(false)
```

Take care not to mix up a numeric *0* with *false*:

```
<?php

  $result = 10 % 0;

  var_dump($result == 0);

?>
```

The program will output:

```
Warning: Division by zero in test.php on line 3

bool(true)
```

As we can see, PHP can not distinguish between *0* and *false* when using the == comparison operator. You must use the type-safe comparison operator === instead:

```
<?php

  $result = 10 % 10;

  var_dump($result == 0);
  var_dump($result == false);

  var_dump($result === 0);
  var_dump($result === false);

?>
```

In this example, the remainder is *0*. The program will output:

```
bool(true)
bool(true)
bool(true)
bool(false)
```

As you can see, only the type-safe comparison yields a correct result.

Wrong Parameter Count In Function Calls

A function in PHP expects a certain number of parameters. PHP also supports optional parameters that have a default value in the function declaration. When the optional parameter is not specified when calling the function, the default value is used.

Optional parameters have to be at the end of the parameter list. Obviously, you can not leave out the first one or two parameters and expect PHP to magically guess whether the given value is the first, second, or third function parameter.

```
<?php

  test('Hello World');
  test('Hello World', true);
  test('Hello World', false);

  function test($aParameter, $aOptional = true)
  {
    var_dump($aOptional);
  }

?>
```

In the first call, the default value true is used for the second parameter:

```
bool(true)
bool(true)
bool(false)
```

A function can have an arbitrary number of parameters. Even built-in functions use optional parameters, so that you rarely see an error message when you specify too few parameters in a function call. The problem is that when calling a built-in function with a wrong parameter count, a warning is issued, and the program continues without executing the function, thus assuming *NULL* as return value:

```
<?php

  var_dump(substr('hello world'));
  var_dump(substr('hello world', 6, 5));
  var_dump(substr(’hello world’, 6, 5, 7));

?>
```

The program outputs:

```
Warning: Wrong parameter count for substr() in test.php on line 3

NULL
string(5) "world"

Warning: Wrong parameter count for substr() in test.php on line 5

NULL
```

The first call has not enough, the third call too many parameters. As you can see, PHP will warn us, but continue the program. This is another example for the fact that you should always configure PHP to display all error classes.

Type-Converting Integer Values

PHP is a dynamically typed language. Other than languages with strong typing, PHP does not need to know the type of a variable when to creating it. PHP will interpret the variable as a different data type according to the situation.

In some cases, automated conversion is not that easy. While it is obvious that the string ’123’ matches the integer value 123, the question arises whether ’ 123’ and ’123 ’ should also be interpreted as 123.

The various PHP versions are not equally strict when it comes to these type conversions. While PHP 5.0 and 5.1 will silently return false when implicit conversion of an integer value fails, current PHP versions generate an `E_NOTICE`, and even convert integers with leading or trailing whitespace. An example is the `date()` function that converts a date given as Unix timestamp:

```
<?php

  var_dump(date(’d.m.Y’, ’   120000000000   ’));

?>
```

The program will output:

```
Notice: A non well formed numeric value encountered in test.php on line 3

string(10) "16.10.1961"
```

Should you see notices like this in your application, consider using `trim()` to remove the extra whitespace. If PHP completely fails to convert the value to an integer, the following warning will be issued:

```
Warning: date() expects parameter 2 to be long, string given in test.php
         on line 3
```

In that case, `date()` will return a boolean false instead of a string.

Empty Objects

The way PHP treats empty objects has changed between PHP 4 and PHP 5. While an object without properties was considered empty by PHP 4, PHP 5 disagrees with that:

```
<?php

  class Test
  {
  }

   $test = new Test;
   var_dump(empty($test));

?>
```

In PHP 4, this program outputs:

```
bool(true)
```

In PHP 5 the result is:

```
bool(false)
```

This difference can be explained by the fact that objects in PHP 4 were basically arrays. Should your program rely on PHP 4's behavior, you can add an `isEmpty()` method to your object that uses reflection to check whether there are any members:

```
<?php

  public function isEmpty()
  {
    $refl = new ReflectionClass($this);
    return sizeof($refl->getProperties()) == 0;
  }

?>
```

$this, Delegation, and Static Calls

In conjunction with `$this` and static method calls, a rather tricky error can occur:

```
<?php

  class Test
  {
    public function delegate()
    {
      Foo::test();
    }
  }

  class Foo
  {
    public function test()
    {
      var_dump($this);
    }
  }


  $test = new Test;
  $test->delegate();
```

```
?>
```

This program will output:

```
Strict standards: Non-static method Foo::test() should not be called statically,
     assuming $this from incompatible context in C:\_php_migration\Code\
    this_and_static.php on line 7

object(Test)#1 (0) {
}
```

If you have configured PHP to not display `E_STRICT` errors, which is the case in most default PHP configurations, you will sooner or later wonder why `$this` inside the `Foo` class resolves to an object of the class `Test`. If you try to call another method in `Foo::test()`, a fatal error will occur:

```
<?php

  ...

  class Foo
  {
    public function test()
    {
      var_dump($this);
      $this->doTest();
    }

    public function doTest() {}
  }

  ...

?>
```

This program will lead to a fatal error, because the non-existing method `doTest()` in `Test` instead of `Foo` is called:

```
Fatal error: Call to undefined method Test::doTest() in test.php on line 18
```

Since the method test() in Foo does not know it has been called from another class, we cannot even blame it for using $this. As we know, $this is not defined inside static methods, since there is not object instance.

In PHP 6, it will not be possible to call non-static methods statically, as we discussed earlier in this chapter; the example presented above being the reason for this decision.

Outputting Objects and the Magic __toString() Method

The magic __toString() method is called to convert an object to a string. This allows for printing objects directly by print '$object, instead of calling a render() or print() method. Older PHP versions used to output the object identifier when implicitly converting an object to a string. Some applications rely on this behavior to write log files.

Until PHP 5.2 __toString() was only called when the object was directly output by print or echo. Since PHP 5.2, the magic method __toString() is called whenever the object is used as a string.

If no __toString() method is present, PHP will not convert the object to a string any more, but output a catchable fatal error that can be handled by a custom error handler:

```
Catchable fatal error: Object of class Test could not be converted to string in
    test.php on line 16
```

It is forbidden to throw exceptions in a __toString() method. Trying to do so will result in a fatal error:

```
Fatal error: Method Test::__toString() must not throw an exception in test.php
    on line 18
```

PHP Extensions

Due to the large number of available PHP extensions, it is impossible to provide you with a full overview of all changes and potential migration problems. This chapter only lists changes in the most commonly used PHP extensions. Fortunately, migration problems with these PHP extensions are rare.

mysql and mysqli

PHP and MySQL are like a married old couple. Both are at least in part responsible for the other's success. Due to license issues, the MySQL client library is not being distributed with PHP 5 any more. While this may sound shocking at first glance, it is in reality not a big issue. You will just have to download and install the client library by yourself. If you do not compile PHP by yourself, you will in most cases not even notice the difference.

On Windows, the MySQL client library is available as a `DLL` file in the PHP directory. Either add this directory to the system's search path, or copy the `DLL` to a directory that is contained in the search path.

The classic MySQL extension in PHP is `mysql`, which provides you with the `mysql_` functions that are used by most existing applications that do not use database abstraction. The newer `mysqli` extension ("i" stands for improved) also depends on the MySQL client library, and allows using new features like nested queries or Unicode support, which is part of MySQL since version 4.1. If you do not need the new features, you can still use the `mysql` extension to access MySQL databases, though.

To avoid licensing issues in the future, MySQL has created a new, open source client library `mysqlnd`. `mysqlnd` can be used as a replacement for the old client library `libmysql` and will probably be the only supported client on the long run. Due to the large existing userbase, though, `libmysql` will also be supported for the next years.

SPL

Since PHP 5.2.1, the method `getFilename()` of the `SPLFileObject` class does not return the full path of a file, but only the file name. To retrieve the full name, use the method `getPathname()`. If you only need the path without the filename, use `getPath()`.

Tidy

We have already introduced the Tidy extension in Chapter 7. PHP 5 contains a newer Tidy version than PHP 4. When migrating from PHP 4, you must adapt your code to the new Tidy API.

Tokenizer

Before a PHP program is executed, the source code is converted to a stream of tokens. The Tokenizer extension allows you to access this stream of tokens. This makes applications that analyze PHP code, like PHP_CodeSniffer described in Chapter 7, possible.

As PHP evolves, new parser tokens are being added. Sometimes, existing tokens are being removed, like `T_ML_COMMENT`, which represents multiline comments. This token was available in PHP 4, but was never actually used, and thus has been removed in PHP 5.

When migrating software using Tokenizer, make sure that you do not require parser tokens of older PHP versions, and support the parser tokens of newer PHP versions. A full list of parser tokens is available at [PHP 2007-9].

XML

When PHP 4 was released, XML was relatively young, thus the libraries supporting XML were not too sophisticated yet. As a result, XML support in PHP 4 was not DOM-compliant.

For PHP 5, the XML support has been completely rewritten. The XML extension in PHP 5 is based on a different library than the PHP 4 extension, so their API is different. The same holds true for the XSL extension in PHP 5.

If your application makes use of the old XML or XML extensions, you will have to adapt the code to the new extensions. Due to the large number of differences, you will probably have to rewrite the part of the application that handles XML.

By comparing the results of the old code and the results of the new code, you can find out whether the application's behavior has altered due to the change. Keep in mind that the new DOM extension by default works with UTF-8 characters. Refer to Chapter 3 for more information.

If you only have to read XML data, you can consider using the SimpleXML extension. As soon as you want to make modifications to the DOM tree, SimpleXML is not the best choice any more.

If you work with very complex XML documents, the XMLReader extension can be useful. Instead of creating a DOM tree from the whole XML file, the individual XML tags are processed on after each other. This allows for processing large amounts of XML data while consuming less memory.

Index