Technologia i rozwiązania

PHP5

Narzędzia dla ekspertów

Osiągnij wyższy poziom zaawansowania w PHP!

- Jak tworzyć efektywny, łatwy w utrzymaniu kod PHP?
- Jak automatycznie tworzyć dokumentację techniczną?
- Jak debugować aplikacje z Xdebug?

Helion

Dirk Merkel

PACKT

PHP 5. Narzędzia dla ekspertów

Autor: Dirk Merkel
Tłumaczenie: Jarosław Dobrzański
ISBN: 978-83-246-2860-5
Tytuł oryginału: Expert PHP 5 Tools
Format: 170×230, stron: 450

Osiągnij wyższy poziom zaawansowania w PHP!

- Jak tworzyć efektywny, łatwy w utrzymaniu kod PHP?
- Jak automatycznie tworzyć dokumentację techniczną?
- Jak debugować aplikację z Xdebug?

Język PHP to ulubione środowisko wielu programistów tworzących aplikacje i strony internetowe. Jego wykorzystanie pozwala na błyskawiczne osiągnięcie efektów, a nauka nie przysparza trudności. Trudno wskazać moment, w którym PHP zdobył tak ogromną popularność. Chwilami można odnieść wrażenie, jakby w sieci był obecny od zawsze. Piąta wersja tego języka zawiera wszystko to, co powinien posiadać nowoczesny język programowania – możliwość programowania obiektowego, wsparcie dla formatu XML oraz rozbudowane mechanizmy wejścia-wyjścia. PHP 5 może z powodzeniem konkurować z „dużymi" rozwiązaniami, dostępnymi od lat na rynku aplikacji internetowych.

Niniejsza książka to pozycja przeznaczona dla programistów, którzy znają już podstawy tego języka. To unikalny podręcznik, dzięki któremu nauczysz się tworzyć efektywny, profesjonalny i łatwy w utrzymaniu kod. W trakcie lektury zdobędziesz wiedzę na temat systemów kontroli wersji, testów jednostkowych, szkieletów aplikacji oraz narzędzi wspomagających proces debugowania. Ponadto dowiesz się, w jaki sposób tworzyć dokumentację z wykorzystaniem phpDocumentor, jak wybrać najlepszy szkielet aplikacji oraz wdrożyć aplikację w środowisku produkcyjnym. Dzięki tej książce osiągniesz wyższy poziom zaawansowania w programowaniu w języku PHP!

- Standardy pisania kodu PHP
- Opracowywanie własnych standardów
- Przygotowanie profesjonalnego środowiska programistycznego
- Dokumentowanie kodu za pomocą phpDocumentor
- Zarządzanie kodem źródłowym i jego wersjami
- Debugowanie aplikacji
- Szkielety aplikacji PHP
- Testy jednostkowe – tworzenie niezawodnego kodu
- Wdrażanie aplikacji
- Projektowanie aplikacji z wykorzystaniem UML
- Proces ciągłej integracji

Ta książka pomoże Ci stać się lepszym programistą!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 8

Wdrażanie aplikacji

Po zakończeniu pisania aplikacji i zadbaniu o to, by inwestorzy zatwierdzili jej odbiór, przychodzi czas na jej wdrożenie. W zasadzie w tym momencie powinniśmy już mieć za sobą kilkukrotne jej wdrożenie i cały proces powinien być w mniejszym lub większym stopniu zautomatyzowany.

Większość projektów, w które byłem ostatnio zaangażowany, skorzystało na tym, że aplikacja była wielokrotnie wdrażana w różnych środowiskach, takich jak programistyczne, testowe i docelowe. Automatyzacja tego procesu umożliwia szybkie uruchamianie kolejnych egzemplarzy aplikacji.

Jest to nie tylko dobry sposób na wyeliminowanie wszelkich potencjalnych problemów przy wdrożeniu, ale także duży krok w kierunku poprawienia produktywności nowych programistów. Jeżeli proces wdrożenia został zoptymalizowany i dobrze udokumentowany, nowi członkowie zespołu programistycznego nie będą musieli poświęcać wiele czasu na utworzenie własnego środowiska programistycznego. Zamiast tego mogą wykonać kilka prostych kroków, aby uzyskać aplikację działającą i gotową do dalszego rozwoju.

Cele i wymagania

Zastanówmy się, jakie powinny być nasze cele podczas wdrażania lub aktualizowania aplikacji. Innymi słowy, co jest miarą sukcesu w tym procesie? Może nam przyjść do głowy twierdzenie, że to, jak dobrze działa aplikacja, jest konsekwencją tego, jak dobrze przeprowadzone zostało wdrożenie. To jednak byłoby mylące. Na tym etapie nie interesuje nas już projekt funkcjonalny, programowanie ani testowanie. Działamy przy założeniu, że dysponujemy w pełni funkcjonującą aplikacją, która musi zostać wdrożona. To, czy aplikacja będzie działać zgodnie z oczekiwaniami, może, ale nie musi być naszym problemem i nie ma nic wspólnego z samym jej wdrażaniem.

Powstaje więc pytanie, na jakiej podstawie ustalić, czy wdrożenie przebiegło pomyślnie? Do czego powinniśmy dążyć, tworząc plan wdrożenia? Jak można się spodziewać, mam kilka przemyśleń na ten temat.

Po pierwsze, wdrożenie powinno odbyć się szybko. Wszyscy ci, którzy wdrażają aplikacje ręcznie, będą zaskoczeni, ile z tego, co robią, można zautomatyzować przy odpowiednim planowaniu. Kiedy wdrażamy aplikację po raz pierwszy, zwykle nie musimy się spieszyć i możemy upewnić się, że wszystko działa, jak trzeba, zanim powiadomimy klienta o tym, iż stała się dostępna. Oczywiście szybkie wdrożenie staje się o wiele ważniejsze, kiedy aktualizujemy aplikację, będącą już w stałym użyciu. W takiej sytuacji naszym celem powinno być zminimalizowanie lub najlepiej uniknięcie wszelkich niedogodności lub przerw w dostawie usług świadczonych przez aplikację. Właśnie wówczas szybkość staje się istotna.

Drugim celem wdrożenia jest jego pełna odwracalność. Najlepiej zacząć traktować aktualizacje aplikacji jako transakcję znaną z systemów bazodanowych. Jeżeli coś się nie powiedzie w trakcie wdrożenia, powinniśmy być w stanie cofnąć wszystkie wykonane dotychczas kroki i przywrócić aplikację dokładnie do stanu sprzed rozpoczęcia wdrożenia. O ile wydaje się to sensowne i logiczne, czasami niełatwo osiągnąć to w praktyce.

Weźmy na przykład sytuację, kiedy musimy zmodyfikować istniejącą tabelę bazy danych. Może to wymagać uruchomienia kilku zapytań, aby zmienić strukturę tabeli i pomanipulować danymi. Jeżeli popełnimy błąd lub pojawi się jakiś nieprzewidziany problem, to jak cofniemy zmiany? Jeżeli mamy plan, będziemy mogli przywrócić bazę z kopii zapasowej, którą zrobiliśmy przed rozpoczęciem wdrażania. Jednak w zależności od rozmiaru bazy ładowanie wszystkich danych do wszystkich tabel może potrwać dobre kilka minut. Możemy więc przywrócić z kopii zapasowej tylko naruszoną tabelę, ale do tego potrzebujemy specjalnego narzędzia. Jeżeli zrobiliśmy kopię zapasową w formie monolitycznego pliku zrzutu, znalezienie jednej tabeli i przywrócenie jej może być trudne. Inną opcją jest uruchamianie specjalnie przygotowanych zapytań cofających zmiany, które doprowadziły do problemu. W tym przypadku zapytania te trzeba będzie przygotować wcześniej i nie będziemy mieć na to czasu w sytuacji kryzysowej przy dużych naciskach na szybkie rozwiązanie problemu.

Jak widać, jest wiele sposobów na to, by uczynić aktualizacje odwracalnymi. Konieczne jest dokładne planowanie, aby proces był odwracalny na każdym etapie.

Podsumujmy nasze wymagania dotyczące wdrożenia lub aktualizacji:

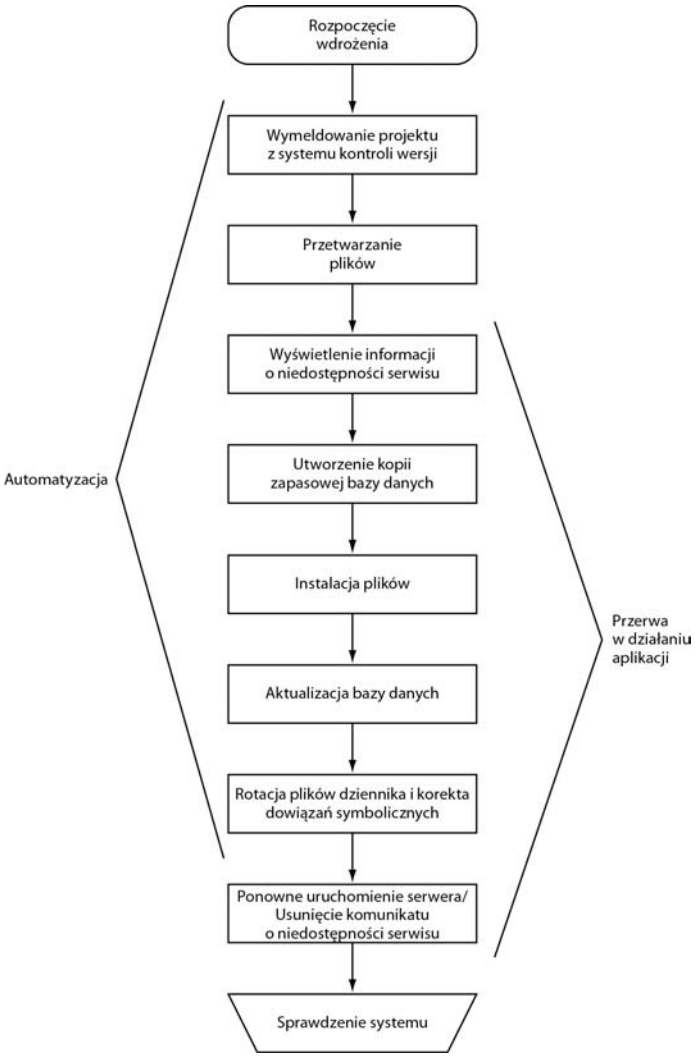
1. Szybkość i automatyzacja w celu minimalizacji niedogodności dla użytkowników i błędów ludzkich.
2. Pełna odwracalność wszystkich czynności.

Powtórzyłem to w punktach, ponieważ dalsze rozważania poświęcone planom wdrożenia, poszczególnym czynnościom i pisaniu kodu, który pozwoli to zrealizować, będą osnute wokół konsekwencji tych wymogów. To na ich podstawie będziemy oceniać, na ile udało się osiągnąć cel.

Wdrażanie aplikacji

W poprzednim punkcie wspominaliśmy o możliwości odwracania poszczególnych kroków w trakcie wdrażania, ale co właściwie kryje się pod pojęciem kroku? Przyjrzyjmy się kilku czynnościom, które są typowe podczas wdrażania lub aktualizowania aplikacji. W dalszej części rozdziału przejdziemy do implementacji tych zadań i próby ich maksymalnego zautomatyzowania.

Poniższy diagram ilustruje kroki w procesie wdrażania aplikacji lub serwisu internetowego. Oczywiście w zależności od aplikacji możemy jakieś kroki dodać, a jakieś zignorować, ale kroki tu pokazane stanowią dobrą podstawę w przypadku większości wdrożeń.



Zanim omówię szczegółowo każdy z powyższych kroków, zastanówmy się nad ogólnym przebiegiem procesu ukazanego na diagramie. Zaczynamy od wymeldowania całego projektu z systemu kontroli wersji. W celu dostosowania się do środowiska, w którym aplikacja będzie wdrażana, niektóre pliki, a w szczególności pliki konfiguracyjne będą musiały zostać zmodyfikowane. Następnie w istniejącym serwisie internetowym publikujemy komunikat informujący użytkowników o trwających pracach konserwacyjnych. Po zainstalowaniu plików, dokonaniu odpowiednich zmian w bazie danych i rotacji plików dziennika usuwamy komunikat o przerwie w funkcjonowaniu serwisu, restartujemy serwer i ponownie udostępniamy stronę użytkownikom. Na koniec dokonujemy inspekcji serwisu, aby sprawdzić, czy aplikacja działa poprawnie.

Wymeldowywanie plików i wysyłanie ich na serwer

Zakładając, że nasz projekt rezyduje w jakimś systemie kontroli wersji, będziemy musieli najpierw „wydobyć" go stamtąd w takiej postaci, która będzie nadawać się do wdrożenia na docelowy serwer. Jeżeli jako systemu kontroli wersji używamy narzędzia Subversion, omówionego w innym rozdziale tej książki, musimy wykonać polecenie `export`, aby uzyskać kopię pozbawioną wszystkich metadanych, jakie Subversion przechowuje zwykle w plikach ukrytych w każdym z katalogów projektu. W innych systemach kontroli wersji może obowiązywać inna terminologia i inne polecenia, ale idea pozostaje taka sama. Konieczne jest utworzenie kopii projektu pozbawionej metadanych i wszelkich innych plików, które nie są przeznaczone do wdrożenia na serwer.

Czasem jednak będzie nam zależeć na takim wdrożeniu, w ramach którego wdrożony kod źródłowy pozostaje powiązany z repozytorium. Posługując się terminologią Subversion, możemy wymeldować kod do lokalnej lub zdalnej kopii roboczej. Korzyści z takiego podejścia są dwojakie. Po pierwsze możemy używać informacji z repozytorium do przyspieszenia przyszłych procesów aktualizacji. Możemy na przykład zaktualizować wymeldowaną wersję do bieżącej wersji z repozytorium jednym poleceniem. Po drugie, jeżeli nasz proces wdrożeniowy obsługuje wymeldowania z repozytorium, można dzięki niemu szybko tworzyć środowiska programowania i w ten sposób poprawiać produktywność nowych członków zespołu. Z tego względu proces, który opracujemy w tym rozdziale, będzie umożliwiał zarówno wdrożenie docelowe, jak i programistyczne.

Prawdopodobnie będziemy musieli również zmienić jeden z kilku plików konfiguracyjnych, wprowadzając takie ustawienia, które odpowiadają środowisku docelowemu aplikacji. Bywa, że trzeba ustawić w ten sposób katalog główny projektu, parametry uwierzytelniające dla bazy danych, bazowy adres URL itp.

Warto w tym momencie powrócić na chwilę do rozdziału poświęconego szkieletom aplikacji, ponieważ stworzona tam przykładowa aplikacja została specjalnie zaprojektowana tak, by działać w różnych środowiskach z użyciem stosownych sekcji w pliku konfiguracyjnym. Takie rozwiązanie przewiduje istnienie zautomatyzowanego procesu wdrożeniowego, obsługującego różne środowiska.

Dysponując wdrażaną wersją projektu, możemy przesłać go na serwer. Zwykle następuje to za pośrednictwem protokołów FTP, SFTP lub SCP (SSH). Zalecam korzystanie z któregoś z bezpiecznych wariantów, aby nie eksponować danych uwierzytelniających.

Jeżeli pracujemy na instalacji rezydującej na serwerze docelowym, możemy połączyć dwa kroki, czyli eksportowanie i wysyłanie plików na serwer, wykorzystując fakt, że wszystkie nowoczesne systemy kontroli wersji obsługują operacje zdalne. Wystarczy wymeldować projekt z repozytorium wprost na docelowy komputer.

Wyświetlanie informacji o niedostępności serwisu

Zanim naciśniemy metaforyczny guzik „wyłączający" serwis, powinniśmy powiadomić użytkowników, że prowadzimy prace konserwacyjne na serwerze. Ja sam zwykle kieruję użytkowników do standardowej strony HTML informującej o trwających pracach i sugerującej ponowną próbę połączenia za kilka minut.

Jeżeli dobrze zaplanujemy i zautomatyzujemy aktualizacje, przerwa w dostępności usługi dla użytkowników powinna być minimalna lub wręcz żadna. W tym momencie może paść pytanie, po co wobec tego w ogóle wstawiamy informację o niedostępności serwisu. Otóż informacja ta nagle może stać się bardzo istotna w sytuacji, gdy coś przeszkodzi nam w płynnej realizacji dobrze zaplanowanego wdrożenia. W takiej chwili będziemy desperacko próbować ustalić źródło problemu i jak najszybciej go wyeliminować. Dużą ulgą będzie wówczas to, że nie musimy przejmować się informowaniem użytkowników i zamiast tego możemy skupić się na tym, co naprawdę wymaga naszej uwagi, czyli na rozwiązywaniu problemu.

Po zakończeniu aktualizacji strona informująca o przerwie musi zostać usunięta. To również należy zautomatyzować, ponieważ jest to coś, o czym naprawdę nie chcielibyśmy zapomnieć.

Aktualizacja i instalacja plików

Kiedy pliki są już na serwerze, musimy udostępnić je dla użytkowników. Zastępowanie istniejących plików nowymi nie jest zgodne z naszą zasadą odwracalności. Można by zmienić nazwy plików i przenieść je do katalogu ze starymi plikami, po czym zastąpić oryginały nowymi wersjami, ale istnieje szybsza metoda.

W systemach, które dopuszczają stosowanie dowiązań symbolicznych (zwanych też **aliasami** lub **skrótami**), dobra praktyka polega na utworzeniu dowiązania symbolicznego wskazującego katalog zawierający pliki aplikacji. Gdy przychodzi czas aktualizacji, wystarczy przekierować dowiązanie na inny katalog, a nowe pliki staną się z miejsca dostępne.

Aktualizacja schematu i zawartości bazy danych

Większość współczesnych aplikacji korzysta z jakiejś formy bazy danych, wśród których prym wiedzie MySQL z uwagi na swoją popularność. Podczas pierwszego wdrożenia aplikacji zwykle jesteśmy także odpowiedzialni za prawidłowe utworzenie i udostępnienie bazy danych. W przypadku aktualizacji istniejącej instalacji musimy znaleźć jakiś sposób modyfikacji schematu i zawartości bazy. Najczęściej polega to na utrzymywaniu pliku tekstowego, do którego programiści dodają zapytania wprowadzające w bazie zmiany wymagane przez zaktualizowany kod. W trakcie wdrażania administrator bazy lub programista jest wówczas odpowiedzialny za wykonanie tych zapytań w tym czasie, gdy wykonywane są wszystkie inne zadania związane z wdrożeniem.

Mimo że takie podejście może się sprawdzać w przypadku aktualizacji lub tworzenia baz, nie spełnia ono postawionych wcześniej wymogów. Otóż nie umożliwia łatwej odwracalności, chyba że z góry to zaplanujemy. Poza tym wdrażając kolejne aktualizacje, napotykamy kolejne zmiany w bazie danych i wówczas przywrócenie poprzednich stanów staje się coraz bardziej skomplikowane.

Jedno z rozwiązań może stanowić narzędzie umożliwiające definiowanie, zarządzanie i wykonywanie krokowych zmian w bazie danych. Owe krokowe zmiany zwane są migracjami, a narzędzie, za którego pomocą będziemy zarządzać migracjami, to DbDeploy. Ponieważ każdemu krokowi aktualizującemu bazę odpowiada krok wstecz, powracający do poprzedniej wersji, możliwe jest przeskakiwanie od stanu do stanu bazy w dowolnym kierunku. Co więcej, zmiany można aplikować w sposób zautomatyzowany, co zaspakaja obydwa nasze wymogi dotyczące udanego wdrożenia.

W szczegóły programu DbDeploy zagłębimy się w dalszej części rozdziału, przy okazji definiowania migracji bazy danych na potrzeby naszego przykładu.

Rotacja plików dziennika i aktualizacja dowiązań symbolicznych

W zależności od tego, gdzie i w jaki sposób nasza aplikacja przechowuje informacje, konieczna albo przynajmniej zalecana może być rotacja plików dziennika. Przede wszystkim trzeba się upewnić, czy serwer może zapisywać do plików dziennika, co nie jest pewne w przypadku, gdy pliki dziennika znajdują się w podkatalogu naszej aplikacji i uaktywniliśmy je na serwerze docelowym, edytując dowiązanie symboliczne.

Bywa, że do świeżo wdrożonej aplikacji trzeba skopiować inne biblioteki lub aplikacje wspomagające. Ja na przykład utrzymuję interfejs WWW do czytania poczty, który jest dostępny z poziomu katalogu głównego dokumentów serwera mojej strony WWW. Nie jest on jednak częścią projektu w repozytorium Subversion i trzeba go skopiować lub przekierować dowiązanie symboliczne po każdym wdrożeniu nowej wersji strony.

Weryfikacja wdrożonej aplikacji

Krok ten może się wydawać oczywisty, ale jednocześnie jest to coś, o czym nie wypada zapomnieć. Powinniśmy sprawdzić, czy wszystko, co zostało wdrożone, działa tak, jak oczekujemy. W tym jednym kroku musimy dopuścić pewne odstępstwo od założonych wymogów. Co prawda możemy i chcemy pewne testy zautomatyzować, np. testowanie nagłówków odpowiedzi HTTP, ale niektóre rzeczy po prostu trzeba sprawdzić ręcznie. Czasami najprostszym sposobem na sprawdzenie, czy wszystko działa, jest otwarcie przeglądarki i skorzystanie z aplikacji tak, jak będą korzystać z niej użytkownicy. Dzięki temu z miejsca możemy wykryć wiele poważnych problemów.

Automatyzacja procesu wdrożenia

Skoro wiemy już, co chcemy osiągnąć, przejdźmy do omawiania narzędzi, za których pomocą dokonamy implementacji i automatyzacji naszego planu wdrożenia. Jest kilka pomniejszych narzędzi, które pomogą nam wykonać to zadanie, ale głównym narzędziem pozwalającym wszystko zautomatyzować i wykonującym większość zadań jest Phing.

Phing

Phing to system konsolidacji projektów. Nazwa to rekurencyjny akronim, którego pełne brzmienie w języku angielskim to *Phing Is ot Gnu Make* (Phing to nie GNU Make). Phing umożliwia wykonywanie różnych zadań związanych z konsolidacją oprogramowania. Szczególnie dobry jest w automatyzacji zadań, która, jak można wnioskować po dotychczasowej lekturze tego rozdziału, jest dla nas bardzo ważna.

Co prawda twórcy narzędzia Phing aktywnie zaprzeczają wszelkim związkom pomiędzy tym programem a narzędziem *make*, ale można bezpiecznie stwierdzić, że *make* leży w jakiejś części u podstaw Phing. Phing został jednak bardziej oparty na narzędziu *Ant*, które jest najczęściej stosowanym systemem konsolidacyjnym w języku Java.

W naszym przypadku przewaga Phing nad narzędziem *Ant* polega na tym, że obsługuje on różne zadania specyficzne dla programowania w PHP. Poza tym Phing jest napisany w PHP, co ułatwia programistom tego języka rozszerzanie funkcjonalności systemu.

Phing jest sterowany tzw. celami (*targets*) zdefiniowanymi w pliku XML. Cele są w gruncie rzeczy działaniami wykonywanymi przez Phing. Plik XML definiujący owe cele i zależności między nimi zwykle ma nazwę *build.xml*. Cele składają się z kolei z jednego lub kilku zadań. Więcej o celach i zadaniach będzie w dalszej części rozdziału.

Takie rozwiązanie ułatwia odrębne wykonywanie któregoś ze zdefiniowanych celów z automatyczną obsługą zależności. Przykładowe cele zdefiniowane przez użytkownika to:

- *create-skeleton* — tworzy katalogi potrzebne na serwerze.
- *checkout-site* — wymeldowuje projekt z systemu Subversion.
- *update-db* — wykonuje wstępnie zdefiniowane zapytania w celu aktualizacji struktury i zawartości bazy danych.

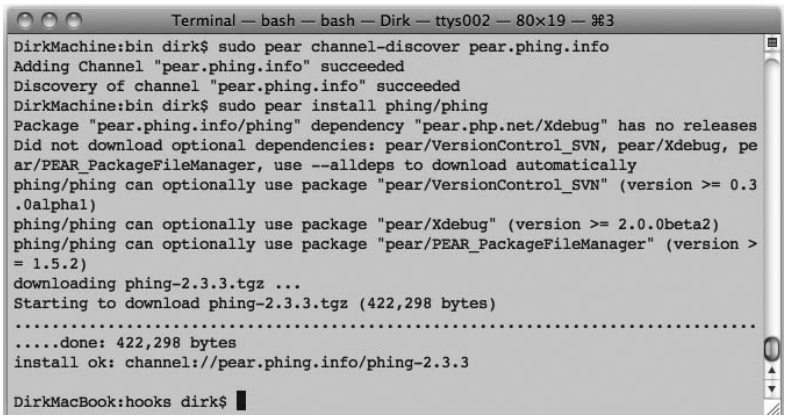
Powyższych przykładowych celów użyjemy też między innymi w naszym projekcie.

Instalacja narzędzia Phing

Narzędzie Phing można zainstalować na kilka sposobów. Najprostszy i najbardziej bezbolesny sposób polega na wykorzystaniu instalatora Pear. Z repozytoriów i narzędzia Pear korzystaliśmy już wiele razy w tej książce. Przyczyna jest prosta — to działa i stało się ogólnie przyjęte do tego stopnia, że większość narzędzi, które tutaj omawiam, można w ten sposób pobrać i zainstalować.

Zamiast od razu uruchamiać narzędzie *pear*, chciałbym zwrócić uwagę, jak świetny przykład stanowi ono w niniejszym rozdziale. Chwila zastanowienia i okazuje się, że wpisując `pear install phing/phing`, robimy dokładnie to, czemu poświęcony jest ten rozdział — wdrażamy (instalujemy) aplikację, a konkretnie Phing. Innymi słowy, zależnie od typu aplikacji rozprowadzanie jej za pośrednictwem kanału Pear może stanowić jeszcze jedno podejście do wdrażania.

Teraz możemy przejść do praktyki i zainstalować Phing za pomocą Pear. Oto przebieg i rezultat instalacji Phing z wiersza poleceń:



Tak naprawdę są tu tylko dwa polecenia. Pierwsze, `pear channel-discover pear.phing.info`, informuje instalator Pear, że pod adresem *pear.phing.info* znajduje się repozytorium Pear. Drugie polecenie, `pear install phing/phing`, instaluje pakiet o nazwie *phing* (drugie wystąpienie „phing") poprzez kanał o nazwie *phing* (pierwsze wystąpienie „phing").

Inną metodą instalacji Phing jest bezpośrednie wymeldowanie go z repozytorium CVS projektu. Zaletą jest to, że otrzymujemy w ten sposób najświeższą i najlepszą bazę kodu, w tym nieopublikowane jeszcze poprawki i ulepszenia. Z tej właśnie metody musimy skorzystać, jeżeli chcemy włożyć własną pracę w rozwój projektu, ponieważ będziemy wówczas mogli zatwierdzać zmiany z powrotem do repozytorium. Oczywiście zakładając, że otrzymamy uprawnienia do dokonywania zmian.

Podstawowa składnia i struktura pliku

Plik konsolidacyjny zawiera kod XML definiujący wszystkie działania i cele dostępne dla użytkownika. Zgodnie z konwencją plik ten otrzymuje nazwę *build.xml*. Jeżeli jednak korzystamy z opcji `–buildfile [nazwa_pliku]` w wierszu poleceń, możemy zmienić tę nazwę na dowolną inną. W naszym przykładzie pozostaniemy przy przyjętej konwencji nazewniczej.

Przyjrzyjmy się ogólnej strukturze pliku konsolidacyjnego Phing. Poniższy szkielet takiego pliku nie definiuje żadnych działań. Jego celem jest jedynie zilustrowanie podstawowej struktury takich plików. Wraz z postępem pracy nad naszym przykładem będziemy uzupełniać kolejne części tego pliku, zmierzając do opracowania w pełni zautomatyzowanego procesu wdrożeniowego.

```
<l?xml version="1.0"?>

<project name="nazwaProjektu" description="Opcjonalny opis pliku
➥konsolidacyjnego." default="nazwaCeluDomyślnego">
    <property name="jakaśWłaściwośćGlobalna" value="wartość" override="true" />
    <type>
        <!-- globalna definicja typu -->
    </type>

    <target name="nazwaCeluDomyślnego" depends="celPomoniczy" description="Opis
     ➥domyślnego zadania.">
        <property name="właściwośćLokalna" value="wartość" override="true" />
        <type>
            <!-- lokalna definicja typu -->
        </type>

        <task>
            <!-- definicja zadania -->
        </task>
        <!-- następne zadania (opcjonalnie) -->
    </target>

    <target name="celPomocniczy" description="Opis celu pomocnicznego.">
        <task>
            <!-- definicja zadania -->
        </task>
```

```
        <!-- następne zadania (opcjonalnie) -->
    </target>
</project>
```

Jako programista zapewne znasz dobrze format XML, dlatego omówię szkielet pliku konsolidacyjnego bardzo krótko. Wydaje mi się też, że najlepiej zacząć omawianie hierarchii znaczników od wewnątrz.

Zadania

Sercem pliku konsolidacyjnego są zadania ujmowane w znacznikach `<task>`. Znaczniki te odpowiadają wprost działaniom. To tutaj wykonywana jest cała rzeczywista praca. Znaczniki `<task>` można traktować jako najdrobniejszą jednostkę wykonywanych działań. Dokumentacja Phing definiuje te podstawowe zadania jako te, które są niezbędne, by skonsolidować projekt. Dla odróżnienia zadania opcjonalne to te, które nie są niezbędne dla skonsolidowania projektu. Moim zdaniem rozróżnienie to jest nieco sztuczne, szczególnie w związku z faktem, że PHP to język interpretowany, co oznacza, że proces konsolidacji nie zawiera w sobie fazy kompilacji.

Oto przykładowe zadania:

- `CopyTask` — kopiuje pliki lub grupy plików albo katalogów z jednego miejsca w systemie plików w inne, z możliwością zmiany nazwy.
- `ForeachTask` — przechodzi przez listę i umożliwia ujęcie jednego lub kilku zadań w pętli i wykonanie każdego z nich dla każdego elementu z listy.
- `InputTask` — prosi użytkownika o wprowadzenie danych, z których można skorzystać przy wykonywaniu następnych zadań.

A oto przykłady zadań opcjonalnych:

- `SvnExportTask` — eksportuje projekt z repozytorium Subversion do lokalnego katalogu.
- `ZipTask/UnzipTask` — dwa uzupełniające się zadania tworzące archiwum ZIP z grupy plików lub rozpakowujące pliki z istniejącego archiwum.
- `PHPUnit2Task` — uruchamia przypadki testowania lub zestawy testów za pośrednictwem systemu testującego PHPUnit2.

Zadania przypominają nieco funkcje w tym sensie, że mogą przyjmować argumenty. Gdy tylko zaczniemy tworzyć nasz przykładowy skrypt konsolidacyjny do wdrażania aplikacji, zobaczymy praktyczne przykłady zadań.

Zamiast podawać listę wszystkich dostępnych zadań wraz z możliwymi opcjami, odsyłam Czytelników do bardzo dobrze napisanej dokumentacji online, zawierającej najświeższą i najlepszą listę zadań wraz z opisami, dostępnej pod adresem *http://phing.info/docs/guide/*.

Wreszcie, jeżeli zestaw zadań udostępniany przez Phing nie zaspakaja naszych wymagań, możemy bez problemu dodawać własne zadania. Jako narzędzie open source Phing jest z założenia rozszerzalny. Fakt, iż został napisany w PHP, potencjalnie ułatwi większości Czytelników tej książki dopisywanie kodu własnych zadań. Tak naprawdę dodanie własnego zadania w formie klasy przyjmującej argumenty i wykonującej pożądane operacje jest zaskakująco proste.

Cele

Cele (*targets*) to logicznie powiązane grupy zadań. Zadania grupowane są w formie celów, aby osiągnąć określony rezultat. Możemy na przykład mieć cel o nazwie `backup-db`, który grupuje zadanie tworzące kopię zapasową bazy danych, zadanie kompresujące otrzymany plik zrzutu bazy oraz zadanie przesyłające kopię poprzez FTP do miejsca, w którym zwykle przechowujemy kopie zapasowe.

Zadania zawarte pomiędzy otwierającym i zamykającym znacznikiem `<target>` są wykonywane w kolejności występowania. Cele mają trzy atrybuty — są to `name` (nazwa), `description` (opis) oraz `depends` (powiązania). Dzięki atrybutowi `name` możliwe jest wykonanie danego celu z wiersza poleceń. Oto przykładowe wywołanie celu `upgrade-db` w domyślnym pliku konsolidacyjnym *build.xml*:

```
$ phing upgrade-db
```

Nazwa celu jest opcjonalna w powyższym wywołaniu i jeżeli nie zostanie podana, wykonywany jest domyślny cel zdefiniowany w znaczniku `<project>`, który omówiony zostanie za chwilę.

Atrybut `description` znacznika `<target>` zawiera krótkie podsumowanie działań wykonywanych w ramach celu.

Wreszcie atrybut `depends` pozwala wskazać inne cele, które muszą zostać wykonane przed danym celem. Phing śledzi, które z celów zostały już wykonane, i automatycznie wywołuje cele, które są konieczne, aby spełnić ten wymóg. W przedstawionym wcześniej przykładowym szkielecie pliku *build.xml* cel o nazwie `domyślnaNazwaCelu` jest uzależniony od celu `celPomocniczy`. Jeżeli wywołamy cel `domyślnaNazwaCelu`, Phing zadba o to, by `celPomocniczy` został wykonany wcześniej. W atrybucie `depends` można podać więcej niż jeden cel zależny, oddzielając je przecinkami. Podobnie jak zadania, cele są wykonywane w kolejności występowania.

Właściwości i plik właściwości

W terminologii narzędzia Phing właściwości to odpowiedniki zmiennych. Można definiować je w globalnej przestrzeni nazw lub w lokalnej dla określonego celu. Globalne definicje właściwości muszą następować poza obrębem znaczników `<target>`, a definicje lokalne w obrębie znacznika `<target>`, którego mają dotyczyć.

Nieco dalej pojawi się kilka globalnych definicji właściwości i typów. Właściwości to w gruncie rzeczy zmienne, z których większość nie zmienia wartości w trakcie wykonywania skryptu. Są jednak również właściwości tworzone dynamicznie i używane przez skrypt konsolidacyjny do zachowania stanu w obrębie celu lub pomiędzy wykonaniem poszczególnych celów.

Właściwości są definiowane i używane w pliku *build.xml* w następujący sposób:

```
<property name="svn.url" value= "https://${svn.server}/home/svn/${svn.project}"
➥override="true" />
```

W tym przykładzie definiujemy właściwość o nazwie `svn.url`. Wartość przypisywana tej właściwości to adres URL, który z kolei jest konstruowany z kilku ciągów tekstowych i dwóch zdefiniowanych wcześniej właściwości: `svn.server` i `svn.project`. Jak widać, aby posłużyć się wartością przypisaną do danej właściwości, należy użyć składni z symbolem dolara, po którym następuje nazwa właściwości w nawiasach klamrowych: `${`*`nazwa_właściwości`*`}`.

Możliwe jest (i bardzo wygodne) przechowywanie właściwości w odrębnych plikach, zawierających wyłącznie pary nazwa-wartość. Pliki te są zgodne z konwencją nazewniczą, nakazującą, by nazwa kończyła się przyrostkiem *.properties*. Oto przykład prostego pliku właściwości:

```
# Subversion
svn.server=waferthin.com
svn.proto=https://

# ... definicje wielu innych właściwości ...

# ustawienia i parametry uwierzytelniające dla bazy danych
db.server=localhost
db.user=root
db.password=psstdonttell
db.name=state_secrets
```

Jak widać, składnia jest bardzo prosta. Wartości są przypisywane nazwom właściwości za pomocą znaku równości i w każdym wierszu musi występować tylko jedna para nazwa-wartość. Importowanie takiego pliku właściwości jest możliwe dzięki atrybutowi `file` zadania `property`:

```
<property file="propfile.properties"/>
```

To wystarczy, by ustawić wszystkie właściwości wymienione w pliku *propfile.properties* dla przestrzeni nazw, w której występuje zadanie `property`.

Używanie plików właściwości ma co najmniej dwie zalety. Po pierwsze, dzięki niemu plik *build.xml* staje się krótszy i bardziej przejrzysty. Składnia XML jest dość rozwlekła, więc utrzymywanie właściwości w odrębnym pliku poprawia czytelność i ułatwia zrozumienie samego pliku konsolidacyjnego. Po drugie, pliki właściwości wprowadzają kolejny poziom abstrakcji, podobnie jak centralny plik lub obiekt konfiguracyjny dodaje poziom abstrakcji do aplikacji PHP. Aby wdrożyć aplikację gdzieś indziej, wystarczy dokonać edycji pliku właściwości bez naruszania pliku *build.xml*.

Rozwijając tę ideę, możemy zapewnić obsługę różnych środowisk. Jak zobaczymy później w naszym przykładzie, możemy po prostu wskazać Phing środowisko, w jakim ma nastąpić wdrożenie, a reszta ustawień będzie realizowana poprzez dołączenie pliku właściwości odpowiadającemu danemu środowisku. Bardzo często spotyka się pliki właściwości o nazwach w stylu *dev.properties*, *staging.properties* lub *production.properties*, odzwierciedlające środowisko, dla którego konfigurowany jest proces konsolidacji lub wdrożenia.

Typy

Typy mogą reprezentować dane bardziej złożone niż właściwości. Na przykład typ może być odnośnikiem do plików w danym katalogu, którego nazwa musi pasować do podanego wyrażenia regularnego. Oto przykład typu `fileset`, który zawiera odnośniki do wszystkich plików *.properties* w katalogu *build* projektu, poza plikiem o nazwie *deprecated.properties*.

```
<fileset dir="${project.home}/build" >
    <include name="*.properties" />
    <exclude name="deprecated.properties" />
</fileset>
```

Oto wbudowane typy Phing:

- `FileList` — uporządkowana lista plików w systemie plików. Pliki nie muszą istnieć w systemie plików.
- `FileSet` — nieuporządkowana lista plików, które istnieją w systemie plików.
- `Path / ClassPath` — służy do reprezentowania zbiorów ścieżek do katalogów.

Dokładny opis funkcjonalności i atrybutów tych typów można znaleźć w dokumentacji narzędzia Phing.

Filtry

Jak sugeruje nazwa, filtry pozwalają filtrować i przekształcać w jakiś sposób zawartość pliku. Gdy pisałem tę książkę, było dostępnych 14 głównych filtrów, pozwalających wykonywać tak różnorodne działania, jak:

- rozwijanie właściwości w pliku,
- usuwanie znaków przejścia do następnego wiersza, komentarzy w wierszu lub komentarzy PHP,
- usuwanie lub dodawanie wierszy w pliku w zależności od ich lokalizacji w danym pliku lub usuwanie zawartości wiersza.

Filtry muszą być zawarte pomiędzy znacznikami otwierającym i zamykającym `filterchain`. Nasz plik konsolidacyjny w podsekcji *mappers* również zawiera przykład zastosowania filtru. W dalszej części rozdziału zobaczymy jeszcze jeden przykład zastosowania filtrów w celu zmiany zawartości jednego lub kilku plików.

Mapery

O ile filtry operują na zawartości pliku, mapery działają podobnie, ale na nazwach plików. Obecnie istnieje w Phing pięć podstawowych maperów, które pozwalają wykonywać na ścieżkach i nazwach plików następujące operacje:

- `FlattenMapper` — usuwa katalogi z podanej ścieżki, pozostawiając jedynie nazwy plików.
- `GlobalMapper` — przemieszcza pliki, nie zmieniając ich nazw.
- `IdentityMapper` — nie zmienia niczego.
- `MergeMapper` — zmienia kilka plików tak, by miały tę samą nazwę.
- `RegexpMapper` — zmienia nazwę plików, posługując się wyrażeniami regularnymi.

Oto przykład zmiany nazw plików szablonów na nazwy rzeczywistych plików PHP z wykorzystaniem filtru `expandproperties` oraz zmiany nazw plików za pomocą filtru `GlobalMapper`:

```
<copy todir="/includes">
    <filterchain>
        <expandproperties />
    </filterchain>
    <mapper type="glob" from="*.php.tpl" to="*.php"/>
    <fileset dir="templates">
        <include name="*.php.tpl" />
    </fileset>
</copy>
```

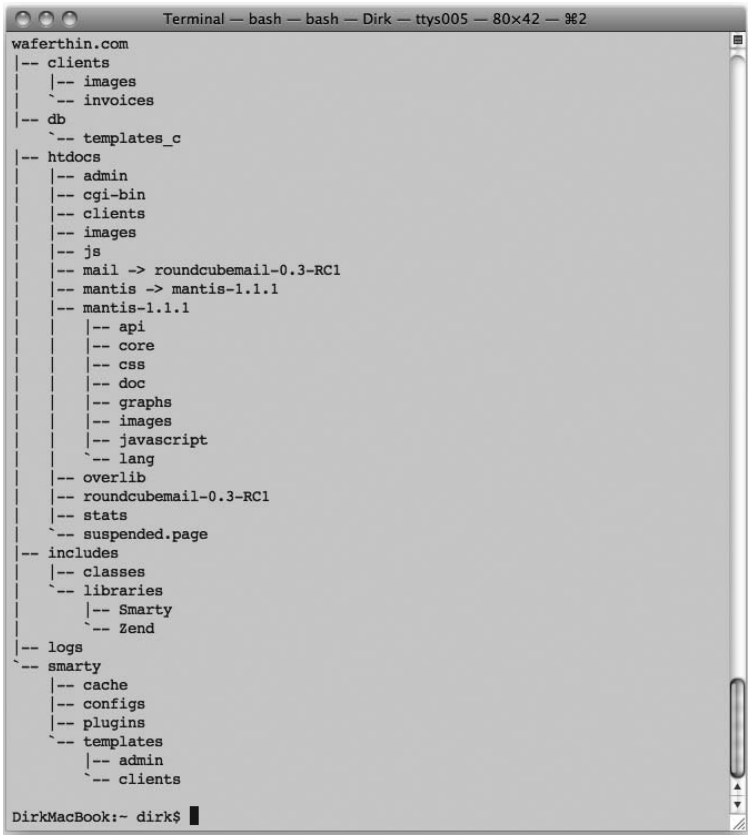
Jak zwykle, pełna lista wszystkich filtrów i maperów oraz dokładne opisy ich zastosowania i atrybutów są dostępne w doskonałej dokumentacji online narzędzia Phing.

Znacznik project

Najbardziej zewnętrzny znacznik to znacznik `<project>`, który zawiera atrybuty definiujące nazwę projektu, jego opis oraz nazwę celu, jaki ma być wykonywany domyślnie. Jak się za chwilę przekonamy, zawsze istnieje możliwość nakazania Phing wykonania celu innego niż zdefiniowany tutaj domyślny. Poza tym Phing korzysta z nazwy projektu, przekazując informacje użytkownikom.

Wdrażanie serwisu

Spróbujmy teraz wykorzystać właśnie zdobytą wiedzę na temat zadań, celów, właściwości, typów, filtrów, maperów oraz projektów i utworzyć plik konsolidacyjny, który płynnie wdroży aktualizację serwisu internetowego. Utworzymy także kilka szablonów i danych, które pozwolą nam dowolnie aktualizować i cofać aktualizacje bazy danych. Zamiast eksperymentować z czyjąś stroną, zdecydowałem się zautomatyzować wdrożenie mojej własnej strony internetowej *waferthin.com*. Oto struktura katalogów wdrożonego serwisu:



Separowanie zewnętrznych zależności

Sensowne wydaje się odseparowanie zewnętrznych zależności, które nie rezydują w naszym systemie kontroli wersji, od reszty projektu. Są to zwykle pliki i katalogi, które niekoniecznie muszą być aktualizowane za każdym razem, gdy przeprowadzamy wdrożenie. Separując te zależności, nie będziemy musieli martwić się o to, że przypadkowo je nadpiszemy lub uszkodzimy. W przypadku mojej strony jest kilka katalogów oraz plików, które zostały po prostu skopiowane na serwer podczas pierwotnej ręcznej instalacji, takich jak Zend Library, Mantis (narzędzie do śledzenia problemów) i RoundCube (przeglądarkowy czytnik e-maili). Katalogi te będą musiały zostać albo przeniesione ze starej wersji serwisu do nowej, albo zastąpione dowiązaniem symbolicznym. Z tego samego powodu katalog *logs* będzie musiał być przeniesiony poza katalog projektu. Po ukończeniu naszego skryptu konsolidacyjnego i udanym wdrożeniu serwisu przyjrzymy się, jak zmieniła się jego struktura w porównaniu ze stanem sprzed wdrożenia.

Tworzenie skryptu konsolidacyjnego

Zacznijmy od utworzenia prostego skryptu konsolidacyjnego. Na szczęście cele dzielą skrypt na łatwiejsze do ogarnięcia części. Będziemy tworzyć kolejno po jednym celu, do momentu gdy wszystkie elementy tej „układanki" będą gotowe i możliwe stanie się wdrożenie serwisu jednym poleceniem.

Środowisko i właściwości

Zwykle pracuję na lokalnej kopii projektu, ale na koniec zdalnie wdrażam wersję testową i ostateczną. Tutaj każdy programista może preferować inny sposób pracy, ale niemal wszyscy spotkamy się z sytuacją, kiedy musimy wdrożyć tę samą aplikację w wielu różnych środowiskach i serwerach. Przydałoby się, aby skrypt Phing był na tyle elastyczny, by uwzględniał owe różne wymagania w sposób nieangażujący użytkownika. Ponieważ większość, jeżeli nie wszystkie czynności, jakie trzeba wykonać, jest przy każdym wdrożeniu taka sama, napiszemy skrypt pozwalający wdrażać aplikację w różnych środowiskach poprzez prostą zmianę kilku właściwości, takich jak nazwa domeny, ścieżka do projektu na serwerze, ustawienia bazy danych itp.

Typowe rozwiązanie tego problemu polega na utworzeniu plików właściwości, które odpowiadają różnym środowiskom, które chcemy obsłużyć. Następnie możemy utworzyć cele ładujące odpowiednie pliki właściwości lub wręcz pytające użytkownika, z którego pliku właściwości należy skorzystać.

Oto plik *dev.properties*, zawierający ustawienia dla wdrożenia wersji mojej strony w środowisku programistycznym na moim lokalnym komputerze:

```
# wdrożenie
site.fqdn=dev.waferthin.com
site.fqdn.secure=dev.secure.waferthin.com
site.home=/Users/dirk/Sites/${site.fqdn}
site.root=/Users/dirk/Sites/${site.fqdn}/${site.fqdn}

# system Subversion
svn.bin=/usr/bin/svn
svn.fqdn=svn
svn.user=dirk
svn.repo=/svn/
svn.proto=https://
svn.project=waferthin.com/trunk
svn.password=donttellanybody

# ustawienia połączenia z bazą i parametry uwierzytelniające
db.user=root
db.password=itsasecret
db.name=waferthin
db.fqdn=localhost
```

```
db.port=3306
db.bin=/usr/local/mysql/bin/mysql
db.backup.dir=${site.home}/backups

# lokalizacja pliku dziennika aplikacji
log=${site.home}/logs/waferthin.log

# moduł szablonów Smarty
smarty.templates_dir=${site.root}/smarty/templates
smarty.compile_dir=${site.root}/smarty/templates_c
smarty.configs_dir=${site.root}/smarty/configs
smarty.cache_dir=${site.root}/smarty/cache
smarty.plugins_dir=${site.root}/smarty/plugins
smarty.plugins2_dir=${site.root}/includes/libraries/Smarty/plugins
smarty.force_compile=true

# zewnętrzne narzędzia
extern.apachectl=/usr/sbin/apachectl
extern.sudo=/usr/bin/sudo
extern.ln=/bin/ln
extern.mysqldump=/usr/local/mysql/bin/mysqldump

# biblioteki
zend_dir=/usr/local/lib/php/Zend
```

Jak widać, plik składa się z sześciu sekcji definiujących następujący podział logiczny:

1. Właściwości z przedrostkiem `site.` odnoszą się do lokalizacji na serwerze, w jakiej ma zostać wdrożony serwis.
2. Właściwości z przedrostkiem `svn.` odnoszą się do dostępu do repozytorium Subversion przechowującego kod źródłowy.
3. Właściwości z przedrostkiem `db.` odnoszą się do parametrów połączenia i parametrów uwierzytelniających umożliwiających połączenie z bazą danych.
4. Właściwości z przedrostkiem `smarty.` odnoszą się do konfiguracji modułu szablonów Smarty.
5. Właściwości z przedrostkiem `extern.` odnoszą się do lokalizacji zewnętrznych plików wykonywalnych, wymaganych przez skrypt konsolidacyjny.
6. Właściwości `log` i `zend_dir` służą do zachowania jeszcze innych zewnętrznych zależności poprzez utworzenie dowiązań symbolicznych (więcej na ten temat w dalszej części rozdziału).

Mam też podobne pliki dla środowiska docelowego (*prod.properties*) oraz testowego (*test.properties*). Wszystkie trzy pliki znajdują się w tym samym katalogu co plik *build.xml*. Po zaimplementowaniu obsługi plików właściwości i wielu różnych środowisk możemy dodawać dowolną liczbę środowisk wdrożeniowych poprzez utworzenie stosownych plików właściwości.

Zacznijmy teraz od utworzenia pliku *build.xml*, który na razie inicjalizuje jedynie środowisko:

```
<?xml version="1.0"?>
<project name="waferthin.com" description="Realizuje utrzymanie i wdrożenie
↪serwisu waferthin.com." default="deploy">

    <!-- Inicjalizuje datownik, który będzie używany przy nadawaniu nazw różnym plikom
    ↪i katalogom. -->
    <tstamp/>

    <target name="deploy" depends="get-env,create-skeleton,svn-export,
     ↪stamp-config,disp-maint,backup-db,deploy-db,publish-site"
     ↪description="Wdraża serwis na serwer WWW i wykonuje niezbędne zadania
     ↪konsolidacyjne i aktualizacyjne.">
    </target>

    <target name="get-env" description="Pobiera środowisko, do jakiego ma
     ↪nastąpić wdrożenie.">
        <!-- Czy środowisko zostało już ustawione? -->
        <if>
            <not>
                <isset property="environment" />
            </not>
            <then>
                <!-- Prosi użytkownika o wybranie środowiska z listy obsługiwanych
                ↪środowisk. -->
                    <input propertyname="environment" validargs="dev,test,prod"
                    ↪promptChar=":">Podaj środowisko </input>
            </then>
        </if>

        <!-- Sprawdza, czy istnieje plik właściwości dla danego środowiska. -->
        <available file="${environment}.properties" property="env_prop_exists"
        ↪type="file" />
        <if>
            <equals arg1="${env_prop_exists}" arg2="true" />
            <then>
                <!-- Odczytuje pliki właściwości. -->
                <property file="${environment}.properties"/>
            </then>
            <else>
                <!-- Przerywa konsolidację i ukazuje komunikat dotyczący błędu. -->
                <fail message="Nie znaleziono pliku właściwości dla wybranego
                 ↪środowiska (${environment}.properties)" />
            </else>
        </if>
    </target>
```

```
    <target name="deploy-dev" description="Wdraża serwis w środowisku
     ↪programowania.">
        <property name="environment" value="dev" override="true" />
        <phingcall target="deploy" />
    </target>

    <target name="deploy-prod" description="Wdraża serwis w środowisku
     ↪docelowym.">
        <property name="environment" value="prod" override="true" />
        <phingcall target="deploy" />
    </target>

    <target name="deploy-test" description="Wdraża serwis w środowisku
    ↪testowym.">
        <property name="environment" value="test" override="true" />
        <phingcall target="deploy" />
    </target>
</project>
```

Znacznik `<project>` zawiera opis celów pliku *build.xml* oraz identyfikuje cel `deploy` jako domyślny. Jedyną interesującą rzeczą związaną z celem `deploy` jest jego atrybut `depends`, który w tym przypadku informuje Phing, że wcześniej wykonany musi zostać cel `get-env`. Przyjrzyjmy się więc celowi `get-env`, który jak na razie jest jedynym celem wykonującym jakieś konkretne zadania.

Oto, co się stanie, gdy uruchomimy wstępną wersję pliku *build.xml* z wiersza poleceń:

```
Terminal — bash — bash — Dirk — ttys002 — 80×17 — ⌘3
DirkMacBook:~ dirk$ phing -f build_1.xml
Buildfile: /Volumes/Documents/Work/WaferThin/Packt/Book/Code/chapter8/build_1.xm
l

waferthin.com > get-env:

Enter environment (dev,test,prod): dev
 [property] Loading /Volumes/Documents/Work/WaferThin/Packt/Book/Code/chapter8/d
ev.properties

waferthin.com > deploy:


BUILD FINISHED

Total time: 2.9911 seconds
DirkMacBook:hooks dirk$
```

W pliku występują również cele `deploy-dev`, `deploy-test` i `deploy-prod`. Ustawiają one właściwość definiującą środowisko na `dev`, `prod` lub `test` w zależności od tego, czy wdrażamy aplikację odpowiednio w środowisku programowania, docelowym lub testowym, po czym wywołują cel `deploy`. Dzięki temu możliwe jest wdrażanie aplikacji w każdym z tych środowisk bez konieczności wpisywania jego nazwy ręcznie.

Szkielet katalogów

Sposób, w jaki wdrażamy naszą aplikację, zakłada istnienie określonej struktury katalogów. Jeżeli wdrażamy aplikację po raz pierwszy, musimy utworzyć katalogi, które będą potrzebne w następnych krokach. Jeżeli dokonujemy aktualizacji, wciąż musimy utworzyć wszelkie katalogi, których wcześniej brakowało.

Oto cel, który zajmuje się tworzeniem katalogów:

```
<!-- Tworzy katalogi; żadne istniejące katalogi nie zostaną nadpisane. -->
    <target name="create-skeleton" description="Tworzy podstawową strukturę
    ↪katalogów dla serwisu.">
        <mkdir dir="${site.home}" />
        <mkdir dir="${site.home}/build" />
        <mkdir dir="${site.home}/backups" />
        <mkdir dir="${site.home}/logs" />
        <mkdir dir="${site.home}/tmp" />
    </target>
...
```

Zadanie `mkdir` tworzy katalog określony za pomocą atrybutu `dir`.

Eksportowanie i wymeldowywanie z Subversion

Teraz przyszła pora na pobranie kodu z systemu kontroli wersji. Jako że w książce tej skupiliśmy się na Subversion jako przykładzie takiego systemu, będziemy tu trzymać się tego przykładu. Jeżeli jednak nasz projekt rezyduje w CVS, Git, Perforce lub jakimkolwiek innym systemie, opisane tu kroki będą wyglądać bardzo podobnie. Tak się składa, że Phing ma pewne wbudowane zadania opcjonalne, pozwalające na interakcję z Subversion. Jeżeli jednak korzystamy z nieco mniej popularnego typu repozytorium, możemy utworzyć własne zadanie Phing lub użyć zadania `ExecTask`, które pozwala uruchamiać pliki wykonywalne w wierszu poleceń.

Oto fragment pliku *build.xml* definiujący cel `svn-export`:

```
...
<target name="svn-export" description="Eksportuje pliki serwisu z Subversion do
↪lokalnego katalogu docelowego.">

        <!-- Konstruuje poprawne URL dla Subversion -->
        <property name="svn.url" value="${svn.proto}
        ↪${svn.fqdn}${svn.repo}${svn.project}" override="true" />

        <!-- Czy hasło dostępu do Subversion zostało podane w pliku właściwości? -->
        <if>
            <not>
                <isset property="svn.password" />
            </not>
```

```
            <then>
                <!-- Prosi użytkownika o podanie hasła dostępu do Subversion. -->
                <input propertyname="svn.password" promptChar=":">Podaj hasło
                 ↪dla użytkownika ${svn.user}, aby pobrać projekt
                 ↪${svn.project} z repozytorium Subversion
                 ↪${svn.fqdn}${svn.repo}</input>
            </then>
        </if>

        <!-- Wymeldowuje projekt do środowiska programowania. -->
        <if>
            <equals arg1="${environment}" arg2="dev" />
            <then>
                <echo>Wymeldowywanie z svn zostało rozpoczęte...</echo>
                <svncheckout svnpath="${svn.bin}"
                    repositoryurl="${svn.url}"
                    todir="${site.root}.${DSTAMP}${TSTAMP}"
                    username="${svn.user}"
                    password="${svn.password}" />
            </then>
            <!-- Eksportuje projekt na potrzeby wdrożenia. -->
            <else>
                <echo>Eksport svn został rozpoczęty ...</echo>
                <svnexport svnpath="${svn.bin}"
                    repositoryurl="${svn.url}"
                    todir="${site.root}.${DSTAMP}${TSTAMP}"
                    username="${svn.user}"
                    password="${svn.password}" />
            </else>
        </if>
    </target>
```

Na początku za pomocą zadania `property` konstruowany jest prawidłowy ciąg URL dla Subversion wskazujący nasz projekt, po czym zostaje on zapisany we właściwości `svn.url`.

Następnie sprawdzamy, czy właściwość `svn.password` została ustawiona. Dobra praktyka nakazuje nie wpisywać haseł do plików właściwości, ale przerywa to pełną automatyzację. Nasze rozwiązanie obsługuje obydwie możliwości — jeżeli nie podano w pliku wartości `svn.password`, Phing poprosi użytkownika za pośrednictwem znacznika `inputTask` o ręczne wpisanie hasła.

Jeżeli nie chcemy za każdym razem wpisywać nazwy użytkownika i hasła SSH, zawsze możemy zainstalować swój publiczny klucz SHH na serwerze, na którym rezyduje repozytorium Subversion, i zmodyfikować plik *build.xml* tak, by nie prosił o podanie parametrów uwierzytelniających.

Zastosowany sposób pobrania kodu z repozytorium zależy od tego, co mamy zamiar z nim zrobić. Użyliśmy tutaj instrukcji warunkowej `if-then-else`, ponieważ wymagane kroki są nieco inne w przypadku środowiska programowania. Jeżeli pracujemy w środowisku programowania,

dokonujemy wymeldowania z Subversion za pomocą zadania `svncheckout`, które pozwoli nam zatwierdzić zmiany z powrotem do repozytorium. Jeżeli z kolei wdrażamy aplikację, zastosujemy zadanie `svnexport`, usuwające wszelkie dane, które w strukturze katalogów przechowuje na własne potrzeby system Subversion.

Tworzenie plików na podstawie szablonów

Każdy serwis lub aplikacja zawiera jakieś dane konfiguracyjne i istnieje wiele różnych sposobów przechowywania tych informacji i udostępniania ich na potrzeby aplikacji. Można się spotkać ze stosowaniem na potrzeby konfiguracji plików właściwości, plików XML lub globalnych zmiennych PHP. W moim serwisie korzystam z klasy `Config` zdefiniowanej w pliku *Config.php*, gdzie ustawienia konfiguracyjne są przechowywane albo jako stałe klasy albo jako prywatne właściwości statyczne. Normalnie oznaczałoby to konieczność ręcznej edycji takiego pliku, aby ustawić parametry odpowiednie dla środowiska, do którego następuje wdrożenie. Skoro jednak staramy się zautomatyzować proces wdrożeniowy, musimy znaleźć jakieś inne rozwiązanie.

Rozwiązanie to będzie polegać na utworzeniu szablonu pliku *Config.php*, na którego podstawie tworzony będzie plik *Config.php* dostosowany do danego środowiska. Oto kilka pierwszych wierszy szablonu pliku *Config.php*:

```
class Config
{
    // ustawienia i parametry uwierzytelniające dla bazy danych
    const DB_VENDOR = 'mysql';
    const DB_HOSTNAME = '${db.fqdn}';
    const DB_PORT = ${db.port};
    const DB_USERNAME = '${db.user}';
    const DB_PASSWORD = '${db.password}';
    const DB_DATABASE_NAME = '${db.name}';

    // lokalizacja pliku dziennika aplikacji
    const LOG_FILE = '${log}';
    ...
```

Bardzo łatwo rozpoznać powyższe bloki, które zostaną zastąpione wartościami przypisanymi do stałych klasy, ponieważ są to po prostu właściwości Phing. Następujący fragment pliku *build.xml* pobiera szablon pliku *Config.php* i zastępuje owe bloki wartościami, jakie reprezentują, pochodzącymi wprost z pliku właściwości.

```
<target name="stamp-config" description="Zapełnia klasę Config.php
↪właściwościami konfiguracyjnymi.">
        <copy todir="${site.root}.${DSTAMP}${TSTAMP}/includes/classes">
          <filterchain>
            <expandproperties />
          </filterchain>
```

```
            <fileset dir="${site.root}.${DSTAMP}${TSTAMP}/config/templates">
              <include name="Config.php" />
            </fileset>
          </copy>
        </target>
```

Zadanie `copy` przenosi szablon *Config.php* do podkatalogu *includes/classes* katalogu zdefiniowanego jako katalog główny aplikacji. W zadaniu tym jednak dzieje się jeszcze kilka innych rzeczy wartych omówienia.

Są tam dwa zagnieżdżone znaczniki. Pierwszy to zadanie `filterchain`, które pozwala przetwarzać kopiowane pliki. W tym przypadku za pomocą zadania `expandproperties` zastępujemy wszystkie bloki reprezentujące właściwości ich wartościami. Zadanie `fileset` pozwala tworzyć listy plików poprzez wyłączanie i włączanie różnych plików na podstawie różnych kryteriów, takich jak wyrażenia regularne, dopasowujące pliki do ścieżki lub nazwy pliku. W naszym przypadku lista zawiera tylko jeden plik, *Config.php*, który dołączamy na podstawie nazwy.

Strona z komunikatem o niedostępności serwisu

W tym momencie zakończyliśmy wszystkie kroki przygotowawcze związane z aktualizacją strony. Na potrzeby następnych działań musimy zadbać o to, by użytkownicy odwiedzający stronę nie zakłócali procesu aktualizacji. Stąd też konieczne jest poinformowanie użytkowników o tymczasowej niedostępności serwisu poprzez przekierowanie całego ruchu na specjalną stronę, która pełni taką właśnie funkcję. W moim serwisie jest to strona *maintenance.html* w głównym katalogu publicznie dostępnej ścieżki */htdocs/*.

Korzystam z Apache w roli serwera WWW, który pozwala tworzyć pliki konfiguracyjne dedykowane dla konkretnego katalogu, zwykle nazywane *.htaccess*. Aby opisywana tu metoda zadziałała, należy się upewnić, czy stosowanie plików *.htaccess* jest uaktywnione. Poniższy kod wymaga także włączenia na serwerze modułu *mod_rewrite*, za którego pomocą modyfikowane jest żądanie URL i przekierowywana jest przeglądarka użytkownika. Krótko mówiąc, tworzymy lokalny plik konfiguracyjny Apache, który za pomocą *mod_rewrite* tymczasowo przekierowuje wszystkie żądania na stronę *maintenance.html*.

```
<target name="disp-maint" description="Eksportuje pliki serwisu z Subversion do
➥lokalnego katalogu docelowego.">
        <!-- Sprawdza, czy plik .htaccess już istnieje. -->
        <available file="${site.root}/htdocs/.htaccess"
        ➥property="htaccess_exists" type="file" />
        <if>
            <equals arg1="${htaccess_exists}" arg2="true" />
            <then>
                <!-- .htaccess istnieje; zmienia jego nazwę. -->
                <move file="${site.root}/htdocs/.htaccess"
                    tofile="${site.home}/htdocs/.htaccess.bck"
                    overwrite="false" />
            </then>
```

```
        </if>

        <!-- nowy plik .htaccess na potrzeby komunikatu o przerwie w dostępności serwisu -->
        <echo file="${site.root}/htdocs/.htaccess" append="false">
            Options +FollowSymlinks
            RewriteEngine on
            RewriteCond %{REQUEST_URI} !/maintenance.html$
            RewriteCond %{REMOTE_HOST} !^127\.0\.0\.1
            RewriteRule $ /maintenance.html [R=302,L]
        </echo>
    </target>
```

Powyższy kod zamiast od razu tworzyć plik *.htaccess*, najpierw sprawdza, czy plik taki już istnieje. Jeżeli istnieje, zmienia jego nazwę za pomocą zadania `move`. Następnie za pomocą zadania `echo` z atrybutem `file` zapisuje niezbędne dyrektywy Apache w nowo utworzonym pliku *.htaccess*.

Kopia zapasowa bazy danych

Ponieważ zablokowaliśmy użytkownikom dostęp do serwisu, możemy mieć pewność, że baza danych, z której korzysta aplikacja, nie będzie używana. Jeżeli serwis, który wdrażamy, ma jakieś zautomatyzowane zadania, korzystające z bazy danych, najprawdopodobniej będziemy musieli tymczasowo je wyłączyć.

Następnym krokiem jest sporządzenie kopii zapasowej bazy danych. Mimo że narzędzie, z którego korzystamy do migracji, obsługuje możliwość aktualizowania i cofania aktualizacji do dowolnej wersji, dobrą praktyką jest tworzenie kopii zapasowej bazy zawsze, gdy coś ulega zmianie. Na szczęście mamy procedurę, która tworzy kopię zapasową bazy oraz całego serwisu.

Oto fragment kodu tworzący kopię zapasową bazy danych:

```
<target name="backup-db" description="Tworzy kopię zapasową bazy danych przez
↪jej aktualizacją.">
        <!-- Czy hasło do bazy zostało podane w pliku właściwości? -->
        <if>
            <not>
                <isset property="db.password" />
            </not>
            <then>
                <!-- Prosi użytkownika o podanie hasła do bazy danych. -->
                <input propertyname="db.password" promptChar=":">Podaj hasło
                ↪użytkownika ${db.user} dla bazy ${db.name}</input>
            </then>
        </if>

        <!-- Wykonuje zewnętrzne polecenie mysqldump, aby utworzyć kopię zapasową bazy
        ↪danych. -->
```

```
        <exec command="${extern.mysqldump} --quick --password=${db.password} —
        ➥user=${db.user} ${db.name} > ${db.name}.${DSTAMP}${TSTAMP}.sql"
            dir="${db.backup.dir}"
            escape="false" />

        <!-- kompresja pliku zrzutu bazy -->
        <zip destfile="${db.backup.dir}/${db.name}.${DSTAMP}${TSTAMP}.sql.zip">
            <fileset dir="${db.backup.dir}">
                <include name="${db.name}.${DSTAMP}${TSTAMP}.sql" />
            </fileset>
        </zip>
        <!-- Usuwa oryginalny plik zrzutu, aby zaoszczędzić miejsce. -->
        <delete file="${db.backup.dir}/${db.name}.${DSTAMP}${TSTAMP}.sql" />
    </target>
```

Zaczynamy od sprawdzenia, czy hasło do bazy danych zostało podane w pliku właściwości. Jeżeli nie, użytkownik jest proszony o jego wpisanie w trybie interaktywnym z wiersza poleceń. Rozwiązanie to powinno wydawać się już znajome, ponieważ podobne zostało zastosowane przy pobieraniu hasła do systemu Subversion.

Następnie za pomocą zadania `exec` uruchamiane jest zewnętrzne polecenie, konkretnie narzędzie `mysqldump`, eksportujące schemat i zawartość bazy danych do pliku tekstowego. Plik ten jest kompletnym obrazem stanu bazy i może być użyty do przywrócenia bazy dokładnie do stanu z chwili jego utworzenia. Ponownie do bazy pliku dołączany jest datownik, aby było wiadomo, kiedy dokładnie został utworzony.

Atrybut `command` zadania `exec` zawiera polecenie, jakie ma zostać wykonane w wierszu poleceń po przejściu do katalogu wskazanego atrybutem `dir`. Atrybut `escape` to wartość logiczna, która precyzuje, czy znaki specjalne powłoki systemowej mają zostać poprzedzone znakiem ucieczki przed wykonaniem polecenia. Opis innych atrybutów obsługiwanych przez zadanie `exec` można znaleźć w dokumentacji.

Pliki zrzutu bazy danych są po prostu plikami tekstowymi i jako takie doskonale nadają się do kompresji, która pozwoli zaoszczędzić miejsce na dysku. Na szczęście Phing udostępnia zadanie kompresujące pliki za pomocą algorytmu ZIP. Podobnie jak wcześniejsze zadanie `copy`, zadanie `zip` zawiera w sobie znacznik `fileset` określający, jakie pliki mają zostać włączone do archiwum. W naszym przypadku kompresji poddajemy jeden plik.

Wreszcie po skompresowaniu pliku zrzutu bazy danych możemy usunąć oryginalny (nieskompresowany) plik, używając zadania `delete`. Co prawda zadanie `delete` obsługuje kilka innych atrybutów, ale tutaj używamy jedynie atrybutu `file` wskazującego plik przeznaczony do usunięcia.

Warto też zwrócić uwagę, że katalog, w którym przechowujemy kopie zapasowe, to jeden z katalogów utworzonych wcześniej w celu `create-skeleton`.

Migracje bazy danych

Po utworzeniu kopii zapasowej bazy danych możemy zastosować wszelkie zmiany do jej schematu i zawartości. W tym celu Phing udostępnia bardzo przydatne zadanie `dbdeploy`. Pozwala ono utworzyć pliki zawierające zmiany w bazie danych. W plikach tych wstawia się kod SQL potrzebny do zaktualizowania bazy oraz kod SQL potrzebny, by wycofać wprowadzone zmiany. Te dwie sekwencje kodu SQL są oddzielone sekwencją `-- //@UNDO`.

Nazwa pliku powinna opisywać jego działanie. Musi także zaczynać się od liczby, która wskazuje na kolejność, w jakiej poszczególne pliki migracji mają być przetwarzane. Pliki o niższym numerze są wykonywane wcześniej.

Aby „pamiętać", która migracja została zastosowana, narzędzie `dbdeploy` wymaga własnego mechanizmu śledzenia:

```
CREATE TABLE `changelog` (
`change_number` bigint(20) NOT NULL,
`delta_set` varchar(10) NOT NULL,
`start_dt` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
↪CURRENT_TIMESTAMP,
`complete_dt` timestamp NULL DEFAULT NULL,
`applied_by` varchar(100) NOT NULL,
`description` varchar(500) NOT NULL,
PRIMARY KEY (`change_number`,`delta_set`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Aby wygenerować tabelę `users`, utworzyłem plik *db/deltas/1-create-users.sql* o następującej zawartości:

```
CREATE TABLE `users` (
`id` int(11) unsigned NOT NULL AUTO_INCREMENT,
`login` varchar(50) NOT NULL,
`password` varchar(100) NOT NULL,
`email` varchar(100) DEFAULT '',
`active` tinyint(1) NOT NULL DEFAULT '1',
`date_modified` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
↪CURRENT_TIMESTAMP,
`date_added` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
PRIMARY KEY (`id`),
UNIQUE KEY `unique_login` (`login`)
) ENGINE=MyISAM AUTO_INCREMENT=4 DEFAULT CHARSET=latin1;
-- //@UNDO

DROP TABLE IF EXISTS `users`;
```

Instrukcja `CREATE TABLE` reprezentuje aktualizację, a `DROP TABLE` wycofanie tej aktualizacji — tworzy to dwukierunkową ścieżkę migracyjną.

Zadanie `dbdeploy` nie wykonuje zapytania SQL, tylko je tworzy. To dlatego potrzebne jest zadanie `exec`, które wykona wygenerowane zapytanie aktualizujące za pośrednictwem klienta tekstowego MySQL. Oto kod celu aktualizującego bazę danych:

```
<target name="deploy-db" description="Uruchamia migracje SQL aktualizujące
↪schemat i zawartość bazy danych.">
        <!-- Ładuje zadanie dbdeploy. -->
        <taskdef name="dbdeploy"
        ↪classname="phing.tasks.ext.dbdeploy.DbDeployTask"/>

         <!-- Generuje SQL aktualizujący bazę danych do najnowszej migracji. -->
        <dbdeploy url="mysql:host=${db.fqdn};dbname=${db.name}"
            userid="${db.user}"
            password="${db.password}"
            dir="${site.root}.${DSTAMP}${TSTAMP}/db/deltas"
            outputfile="${site.home}/build/db-upgrade-${DSTAMP}${TSTAMP}.sql"
            undooutputfile="${site.home}/build/
            ↪db-downgrade-${DSTAMP}${TSTAMP}.sql" />

        <!-- Wykonuje kod SQL za pomocą tekstowego klienta mysql. -->
        <exec
            command="${extern.mysql} -h${db.fqdn} -u${db.user} -p${db.password}
            ↪${db.name} &lt; ${site.home}/build/db-upgrade-
            ↪${DSTAMP}${TSTAMP}.sql"
            dir="${site.home}/build"
            checkreturn="true" />
    </target>
    ...
```

Udostępnianie serwisu

To już prawie koniec. Wymeldowaliśmy i zmodyfikowaliśmy serwis, utworzyliśmy kopię zapasową bazy i dokonaliśmy jej aktualizacji. Pozostało teraz „przełączyć" się ze starej wersji serwisu na nowo utworzoną. Tym zajmuje się cel `publish-site`:

```
<target name="publish-site" description="Aktywuje nową wersję serwisu
↪i restartuje serwer Apache, aby uaktywnić wszystkie zmiany.">
        <!-- Dowiązanie symboliczne do zewnętrznej biblioteki. -->
        <exec command="${extern.ln} -s ${zend_dir}"
            dir="${site.root}.${DSTAMP}${TSTAMP}/includes/libraries"
            escape="false" />
        <!-- Usuwa dowiązanie symboliczne do aktywnej kopii serwisu. -->
        <delete file="${site.root}" />
        <!-- Dowiązanie symboliczne do najnowszej wersji serwisu. -->
        <exec command="${extern.ln} -s ${site.fqdn}.${DSTAMP}${TSTAMP}
        ↪${site.fqdn}"
            dir="${site.home}"
            escape="false" />
```

```
    <!-- Przeprowadza płynny restart serwera Apache, aby uwzględnić wszystkie zmiany.
    ↪Nowa wersja serwisu staje się dostępna!!! -->
    <exec command="${extern.sudo} ${extern.apachectl} graceful"
    ↪escape="false" />
</target>
```

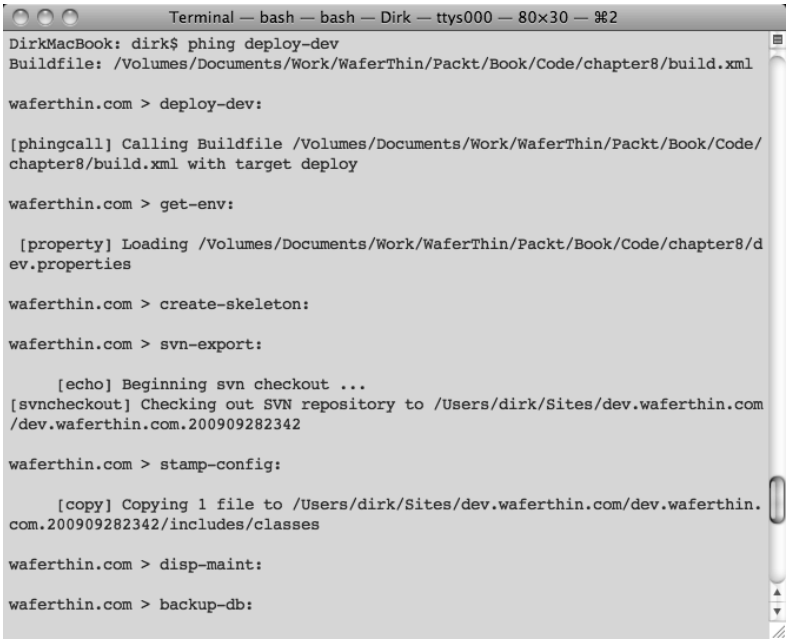
Na początku, ponownie za pomocą zadania `exec`, tworzymy dowiązanie symboliczne do kopii szkieletu Zend, która jest wymagana, by serwis mógł działać. Potem zadaniem `delete` usuwamy dowiązanie symboliczne, które wskazuje na poprzednią wersję serwisu. Następnie, znowu zadaniem `exec`, tworzymy nowe dowiązanie symboliczne do wersji serwisu, która właśnie została wymeldowana z Subversion i przygotowana do wdrożenia.

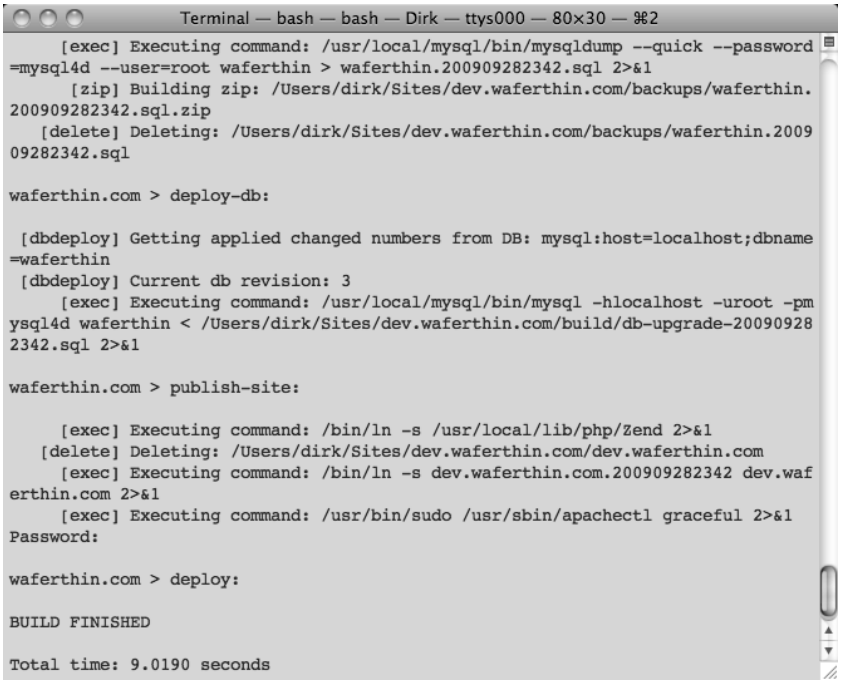
W ostatnim kroku nakazujemy serwerowi Apache przeładowanie plików konfiguracyjnych, aby upewnić się, że wszystkie zmiany zostaną od razu zastosowane. Ponieważ serwer Apache nie działa w przypadku mojego użytkownika, muszę zastosować polecenie `sudo`, które spowoduje wyświetlenie prośby o podanie hasła administratora w celu wykonania tej czynności.

Ostateczna wersja pliku konsolidacyjnego

Skoro skonstruowaliśmy poszczególne cele, możemy je połączyć w finalną wersję pliku *build.xml*. Kompletny listing można zobaczyć w dołączonych do książki przykładach kodów z tego rozdziału.

Po uruchomieniu tego pliku w środowisku programowania z wiersza poleceń otrzymuję następujący rezultat, który rozbiłem na dwa zrzuty, aby wszystko pomieścić.

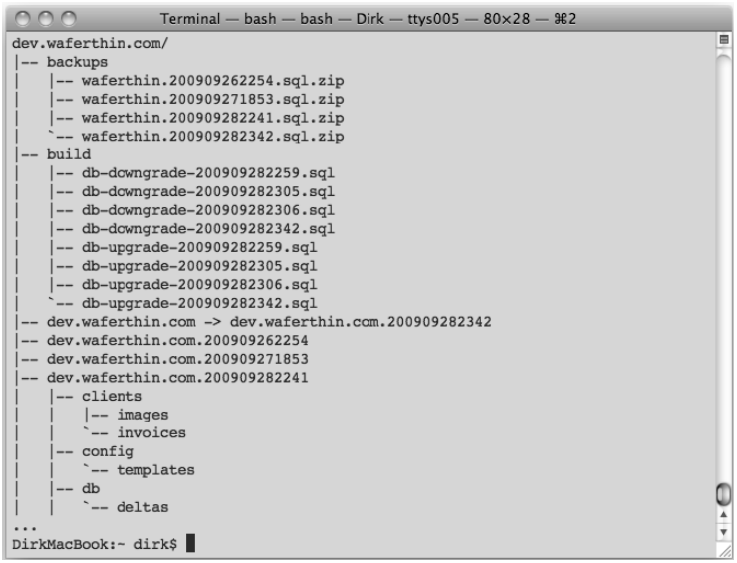


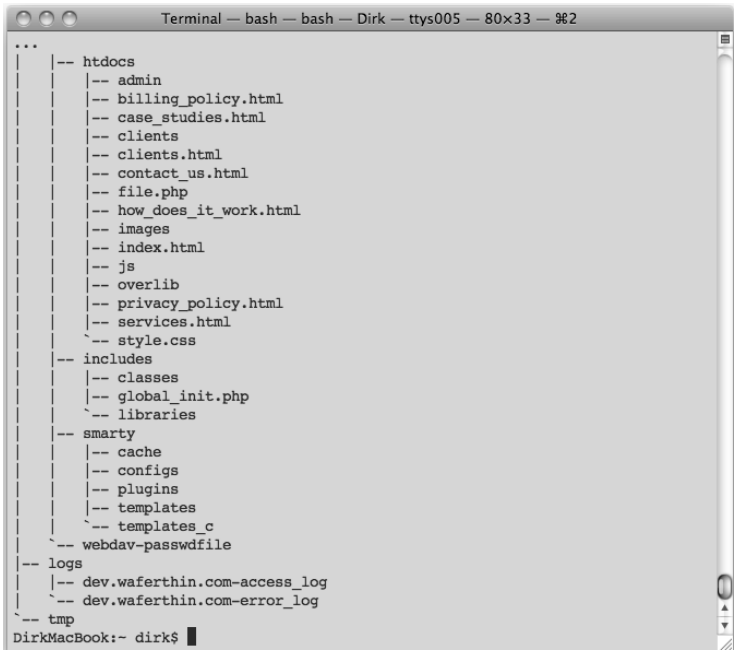


Całkiem niezły wynik. W nieco ponad dziewięć sekund udało mi się wymeldować projekt z Subversion, utworzyć kilka plików z szablonów, wstawić stronę z informacją o niedostępności serwisu, utworzyć kopię zapasową bazy i zaktualizować bazę, utworzyć różne dowiązania symboliczne i katalogi, po czym zrestartować serwer WWW. Wliczam w to czas poświęcony na wpisanie hasła administratora przed restartem serwera.

Spójrzmy teraz na zmienioną strukturę katalogów serwisu. Możemy cofnąć się o parę stron i porównać go do stanu sprzed zmiany jego struktury w celu ułatwienia aktualizacji.

Aby dowieść, jak łatwo możemy teraz wdrażać serwis, dołączyłem w listingu skrypty aktualizujące bazę danych i cofające jej aktualizację (*db-upgrade.xxxx.sql* i *db-upgrade.xxxx.sql*), a także główne katalogi wcześniej wdrożonych aplikacji, które zostały teraz zarchiwizowane (*dev.waferthin.com.xxxx*). Niestety, listing jest tak długi, że muszę podzielić go na dwa zrzuty ekranowe (rysunki na następnej stronie).





Katalog u szczytu hierarchii, *dev.waferthin.com*, zawiera teraz następujące podkatalogi:

- *backups* — zawiera archiwa bazy danych sprzed aktualizacji.
- *build* — zawiera skrypty SQL do krokowej aktualizacji (odtwarzania) bazy danych.
- *dev.waferthin.com.YYYYMMDDHHMM* — kod źródłowy serwisu z datownikiem.

- *dev.waferthin.com* — dowiązanie symboliczne wskazujące na bieżącą wersję serwisu.
- *logs* — pliki dzienników.
- *tmp* — tymczasowy katalog na potrzeby manipulacji plikami.

Wdrożenie aplikacji na serwer testowy i docelowy jest równie proste. O ile mamy zainstalowane narzędzie Phing, musimy jedynie skopiować plik *build.xml* oraz pliki właściwości dla danego środowiska na komputer docelowy i uruchomić odpowiedni cel.

Cofanie aktualizacji

Po udostępnieniu nowej wersji serwisu musimy sprawdzić, czy wszystko działa zgodnie z oczekiwaniami. Ale co, jeżeli zauważymy jakieś problemy? Jeżeli jest to coś, czego nie jesteśmy w stanie naprawić od razu, pozostaje wycofanie aktualizacji i przywrócenie poprzedniej wersji. Przy takiej, a nie innej strukturze serwisu sprowadza się to do edycji dowiązania symbolicznego, tak aby zaczęło wskazywać na starszą wersję serwisu, i zrestartowania serwera WWW. Dodatkowo, w zależności od tego, jakie aktualizacje zostały zastosowane dla bazy danych, możemy także uruchomić skrypt odwracający zmiany w bazie. I to wszystko — w ciągu kilku sekund możemy przełączać się pomiędzy dwoma wersjami lub większą liczbą wersji tego samego serwisu.

Podsumowanie

Mam nadzieję, że lektura tego rozdziału pozwala spojrzeć w nowym świetle na zagadnienie wdrażania aplikacji. Mimo że wdrożenie jest często ostatnim krokiem procesu produkcji oprogramowania, trzeba o nim myśleć już od początku. Połowa sukcesu to zorganizowanie plików serwisu w taki sposób, który ułatwi automatyzację wdrożenia.

Zaczęliśmy ten rozdział od próby zebrania wytycznych, które pozwolą mierzyć sukces wdrożenia. Proces, który opracowaliśmy, spełnia obydwa postawione cele — jest w pełni zautomatyzowany, aby zminimalizować błędy ludzkie oraz czas, przez który serwis jest niedostępny lub nie w pełni funkcjonalny.

W trakcie automatyzowania procesu wdrożenia poznałeś narzędzie Phing, dowiedziałeś się, jak pomaga ono automatyzować wszelkiego rodzaju zadania. Tutaj zastosowaliśmy Phing do wdrożenia serwisu, ale możliwości tego programu są znacznie większe. Możemy tworzyć nim cele wykonujące wszelkiego rodzaju zadania konserwacyjne na kodzie lub serwisie. Przykładem może być synchronizacja plików i katalogów oraz generowanie dokumentacji za pomocą programu phpDocumentor. Lista zastosowań programu Phing nie ma końca.