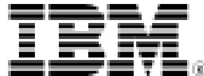


developerWorks.

Automation for the people: Deployment-automation patterns, Part 2

More patterns for one-click deployments

Skill Level: Introductory

Paul Duvall (paul.duvall@stelligent.com)
CTO
Stelligent

10 Feb 2009

Java™ deployments are often messy, error-prone, and manual, leading to delays in making software available to users. In Part 2 of this two-part article, automation expert Paul Duvall expands on a collection of key patterns for developing a reliable, repeatable, and consistent deployment process capable of generating one-click deployments for Java applications.

Deployment is yet another aspect of software creation that lends itself well to automation. Automated deployments reap the benefits of a reliable, repeatable process: improved accuracy, speed, and control. Part 1 of this two-part article describes eight deployment-automation patterns. In this installment, I expand the discussion to cover seven more equally beneficial approaches to deployment:

> **About this series**
> As developers, we work to automate processes for users; yet, many of us overlook opportunities to automate our own development processes. To that end, *Automation for the people* is a series of articles dedicated to exploring the practical uses of automating software development processes and teaching you *when* and *how* to apply automation successfully.

- **Binary Integrity**, which ensures the same artifact is promoted throughout target environments
- **Disposable Container**, which puts a target environment into a known

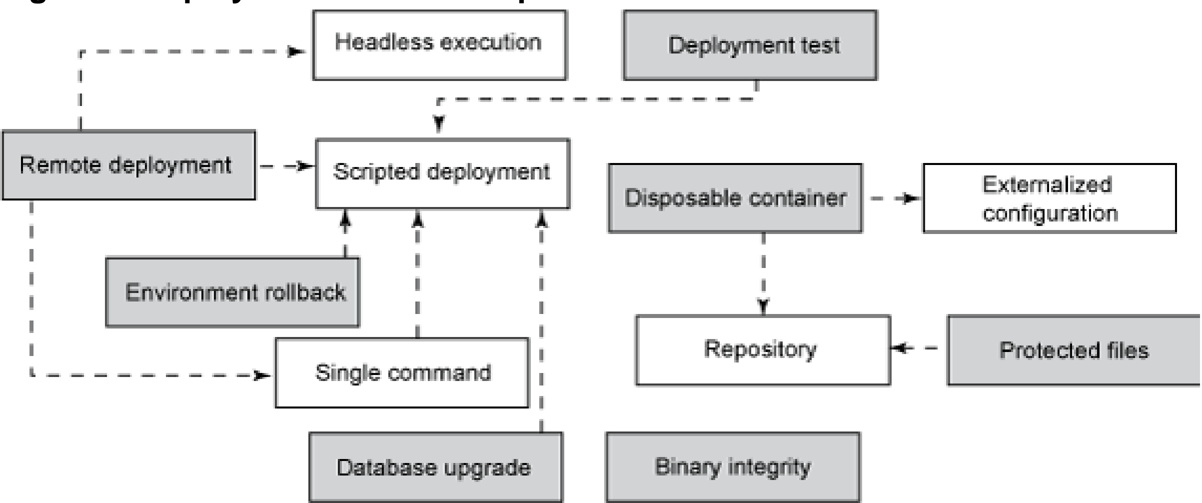
© Copyright IBM Corporation 1994, 2008. All rights reserved.

state to reduce deployment errors

- **Remote Deployment**, which ensures that deployments can interface with multiple machines from a centralized machine or cluster
- **Database Upgrades**, which provides a centrally managed and scripted process to applying incremental changes to the database
- **Deployment Test**, which uses pre- and post-deployment checks to verify the application is working as expected based on recent deployment
- **Environment Rollback**, which rolls back application and database changes if a deployment fails
- **Protected Files**, which controls access to certain files used by a build system

Figure 1 illustrates the relationships among the deployment patterns covered in this article (the unshaded patterns were covered in Part 1):

Figure 1. Deployment-automation patterns



These seven additional deployment-automation patterns build upon the first eight to help you create one-click deployments.

Compile once, deploy to many environments

Name: Binary Integrity

Pattern: For each tagged deployment, the same archive (WAR or EAR) is used in each target environment.

Antipatterns: Separate compilation for each target environment on the same tag.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

After numerous debates with colleagues on this topic, I've firmly come down on the side of *compile once, deploy to many target environments* rather than *compile and package in every target environment*. For instance, the deployment artifact produced from a Java Web deployment is the Web archive (WAR) or enterprise archive (EAR) file. This archive should be checked into the version-control repository and tagged one time — such as in the DEV environment.

Figure 2 illustrates the *compile once, deploy to many* philosophy as the same brewery.war generated on the build machine is deployed to each of the target environments:

Figure 2. The same Web archive deployed to different target environments

Ant provides a `checksum` task — using the Message-Digest algorithm 5 (MD5) hashing algorithm — to ensure the file that was compiled and packaged on the build machine is the same one being deployed to each of the target environments.

Some will argue that although the artifact may be the same, the deployment

© Copyright IBM Corporation 1994, 2008. All rights reserved.

configuration is different for each target environment. That is, when you use a Single-Command, Scripted Deployment, many automated processes can alter the application's output, regardless of whether it's the same archive. This is true; however, you could spend needless hours trying to troubleshoot a problem because the software was compiled and packaged using a different JDK version on the STAGE environment than was run in the QA environment. And opportunities for failure arise when the JARs from a centralized dependency-management repository (such as Ivy or Maven) that were used in DEV are different from those in the staging environment. These risks convince me that in order to ensure binary integrity, I must compile and package once so that I can deploy to many environments.

Make deployments cheap with disposable containers

Name: Disposable Container

Pattern: Automate the installation and configuration of Web and database containers by decoupling installation and configuration.

Antipatterns: Manually install and configure containers into each target environment.

In an earlier *Automation for the people* installment, "Continuous Integration anti-patterns, Part 2," you learned why cleaning up a "polluted" environment helps prevent falsely positive or negative builds. The Disposable Container reduces many of the problems that can occur when you rely on persistent containers. The Disposable Container pattern is based on two principles: *completely remove all container components* and *separate the container installation from its configuration*. This seems like a radical concept to some, particularly systems engineers, because it no longer assumes that containers should be managed, and obfuscated, by a separate team, never to be touched by developers or others. However, considering the common and costly problems that occur during deployments, it can be an area where all team members can realize the most benefit.

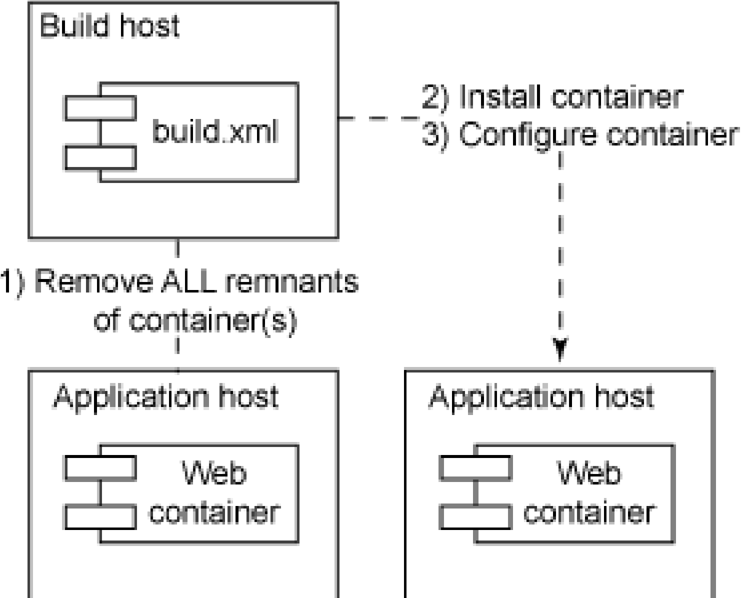
One-click deployments

Many times I've met with teams that have told me "Yep, we've got automated deployments." When I ask a few simple questions — such as "Are you able to type a single command (such as `ant`) to generate a working software application?" — the response usually goes something like: "Yes, once you install and configure the Web container..." or "Yes, once you set up the database." My definition for a truly automated deployment is that you should be able to start with a clean machine, install the Java platform and Ant (there are ways to eliminate this step as well), and then type a *single command* to get a working software application. If you can't, it's not "one-click," and costly human bottlenecks will occur in the deployment process.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

The Disposable Container pattern, shown in Figure 3, is grounded in a philosophy that *everything* should be in the *system* — (using the Repository pattern covered in Part 1) — not in someone's head:

Figure 3. Removing and installing container(s) during deployment



The Ant script in Listing 1 downloads the Tomcat ZIP from the Internet, removing any container remnants from previous deployments, then extracts, installs, and starts Tomcat:

Listing 1. Scripted Deployment, in Ant, that removes, reinstalls, starts, and configures a container

```
<!-- Check to see if Tomcat is running prior to this -->
...
<exec executable="sh" osfamily="unix" dir="${tomcat.home}/bin" spawn="true">
  <env key="NOPAUSE" value="true" />
  <arg line="shutdown.sh" />
</exec>
<delete dir="${tomcat.home}" />
<get src="${tomcat.binary.uri}/${tomcat.binary.file}"
  dest="${download.dir}/${tomcat.binary.file}" usetimestamp="true"/>
<unzip dest="${target.dir}" src="${download.dir}/${tomcat.binary.file}" />
<exec osfamily="unix" executable="chmod" spawn="true">
  <arg value="+x" />
  <arg file="${tomcat.home}/bin/startup.sh" />
  <arg file="${tomcat.home}/bin/shutdown.sh" />
</exec>
<xmltask source="${appserver.server-xml.file}"
  dest="${appserver.server-xml.file}">
  <attr path="/Server/Service[@name='${s.name}']/Connector[${port='${c.port}']"
    attr="proxyPort"
        value="${appserver.external.port}"/>
    <attr path="/Server/Service[${name='${s.name}']/Connector[${port='${c.port}']"
        attr="proxyName"
        value="${appserver.external.host}"/>
</xmltask>
```


© Copyright IBM Corporation 1994, 2008. All rights reserved.

```
<!-- Perform other container configuration -->
...
<echo message="Starting tomcat instance at ${tomcat.home} with startup.sh" />
<exec executable="sh" osfamily="unix" dir="${tomcat.home}/bin" spawn="true">
  <env key="NOPAUSE" value="true" />
  <arg line="startup.sh" />
</exec>
```

By putting an environment into a known state and deploying containers in a controlled manner, you reduce many common deployment errors that are the cause of most deployment pain.

Running commands in multiple external environments

Name: Remote Deployment

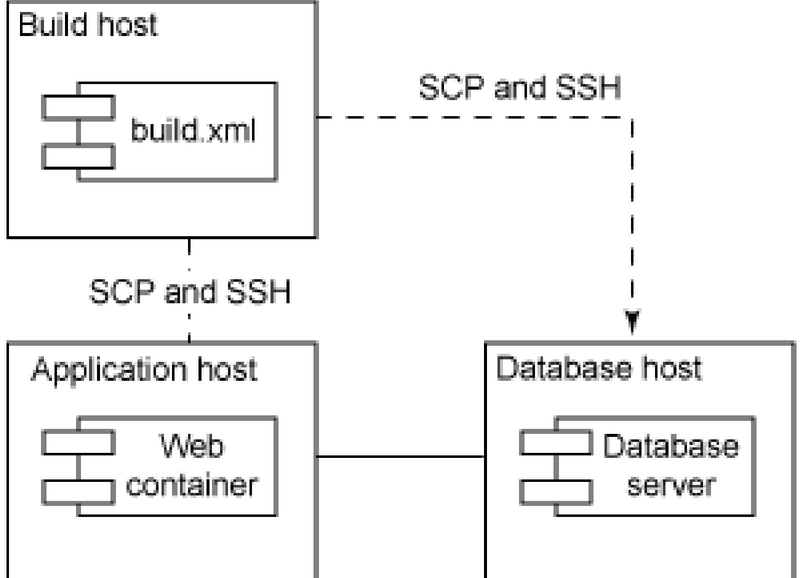
Pattern: Use a centralized machine or cluster to deploy software to multiple target environments.

Antipatterns: Manually applying deployments locally in each target environment.

Once database and Web containers have been installed, getting a deployment to run on a developer's *workstation* is usually rather trivial. However, the difference between development and production is vast. If an organization has multiple projects and different target environments (for instance, testing or staging environments), there's often a need to manage deployments centrally from a single environment: a machine or a cluster. Quite often, teams use a build server to manage deployments between each of these target environments. In Part 1, I covered the Headless Execution pattern, in which you use public and private keys so that you don't need to log in manually to each machine. Remote Deployment, illustrated in Figure 4, relies on the Headless Execution, Single Command, and Scripted Deployment patterns to make it easy to deploy to remote machines:

Figure 4. Build-management server to multiple environments

© Copyright IBM Corporation 1994, 2008. All rights reserved.



To deploy software from a centralized build server remotely, you need to use mechanisms for securely copying and running commands remotely. The two mechanisms I'll illustrate use Secure Copy (SCP) and Secure Shell (SSH). From a Scripted Deployment, as shown in Listing 2, a Web archive that was generated on a centralized build machine is remotely copied to a target environment:

Listing 2. Securely copying a war file from one machine to another

```
<target name="copy-tomcat-dist">
  <scp file="${basedir}/target/brewery.war"
  trust="true"
  keyfile="${basedir}/config/id_dsa"
  username="bobama"
  passphrase=""
  todir="pduvall:G0theD!stance@myhostname:/usr/local/jakarta-tomcat-5.5.20/webapps" />
</target>
```

After the WAR file is securely copied to the remote target environment, I can use something like the `SSHExec` task in Java Secure Channel to run any SSH commands — remotely from the central build machine. An alternative approach is to `ssh` to the remote environment and run any commands locally. This lessens the back-and-forth remote traffic and can reduce deployment times.

Putting database and data into known state

Name: Database Upgrade

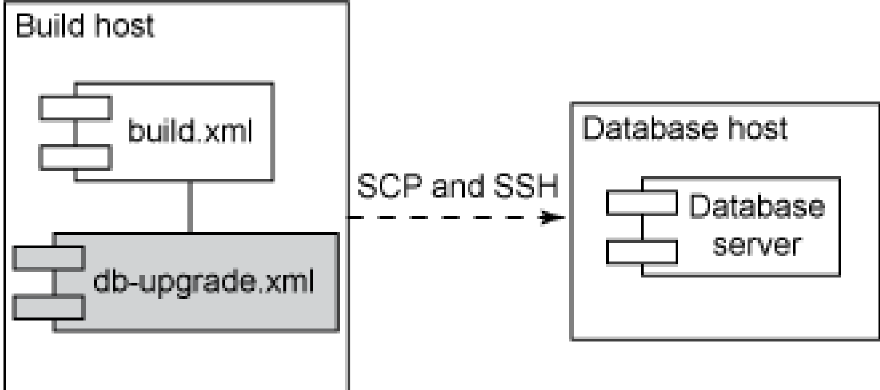
Pattern: Use scripts and database to apply incremental changes in each target environment.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Antipatterns: Manually applying database and data changes in each target environment.

In Figure 5, you see an example of using automated scripts to update the database as part of a Scripted Deployment:

Figure 5. Automatically applying incremental database updates



In an earlier *Automation for the people* installment, "Hands-free database migration," I covered the need to apply incremental database changes in an automated fashion. Like any other part of a Scripted Deployment, the database upgrade scripts are checked into the repository.

LiquiBase (see Resources) is a tool for applying incremental changes to a database so that the same change is applied into each target environment as part of the Scripted Deployment. In Listing 3, an SQL script is called as part of the LiquiBase changelog. This changelog (defined in XML) is then called by the Scripted Deployment (which is implemented in a build-scripting tool — such as Ant).

Listing 3. Running a custom SQL file from a LiquiBase change set

```
<changeSet id="1" author="jbiden">
  <sqlFile path="insert-distributor-data.sql"/>
</changeSet>
```

There's quite a bit more to learning and applying automated database upgrades, but the idea is to perform the upgrades as part of the Scripted Deployment so that all database changes are in the *system*, not a written procedure or in someone's head.

Smoke-testing deployments

Name: Deployment Test

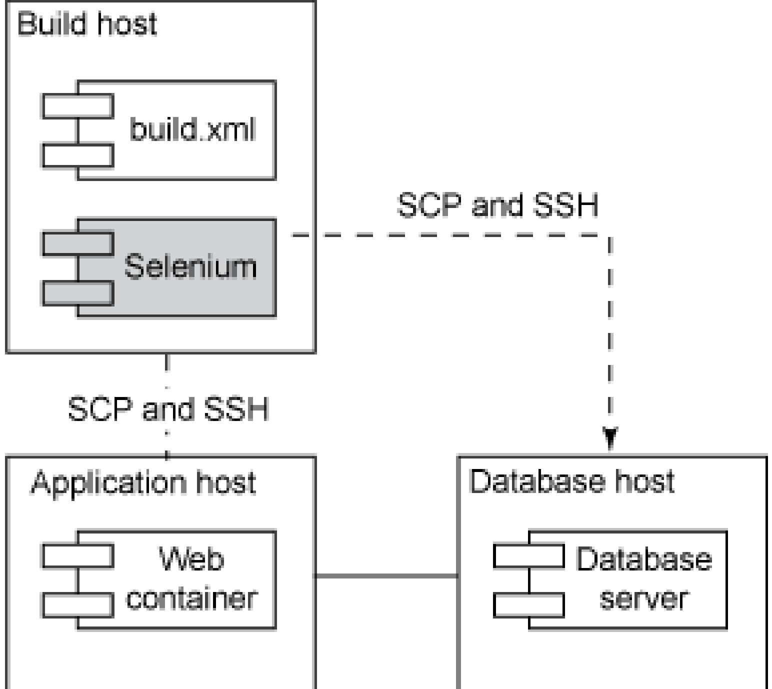
Pattern: Script self-testing capabilities into Scripted Deployments.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Antipatterns: Deployments are verified by running through manual functional tests that don't focus on deployment-specific aspects.

Figure 6 illustrates an example of running deployment tests before and after a deployment:

Figure 6. Running functional deployment tests against application



In Listing 4, I'm using Ant to perform predeployment tests to verify I'm using the correct tool versions. In the Scripted Deployment, the script can check for existing ports in use (which would cause the Web container deployment to fail), verify a connection to the database, and check if containers have been started, along with a host of other internal deployment tests.

Listing 4. Running predeployment checks to ensure deployment efficacy

```
<condition property="ant.version.success">
  <antversion atleast="${ant.check.version}" />
</condition>
<antunit:assertPropertyEquals name="ant.version.success" value="true" />
<echo message="Ant version is correct." />
<echo message="Validating Java version..."/>
<condition property="java.major.version.correct">
  <equals arg1="${ant.java.version}" arg2="${java.check.version.major}" />
</condition>
<antunit:assertTrue message="Your Java SDK version must be 1.5+. \
  You must install correct version.">
  <isset property="java.major.version.correct"/>
</antunit:assertTrue>
```

A more extensive deployment test can ensure that the application's *functionality* is

© Copyright IBM Corporation 1994, 2008. All rights reserved.

correct. By writing *deployment-specific* automated functional tests using a tool such as Selenium for Web applications or Abbot for client applications, you can verify the deployment changes have been properly applied. You can think of these tests as *smoke tests*: you need to test only the functionality that is affected by the deployment. For example, Table 1 shows ways you can use Selenium and other tools for a Web application:

Table 1. Deployment tests

Deployment test	Description
Database	Write an automated functional test that inserts data into a database. Verify the data was entered in the database.
Simple Mail Transfer Protocol (SMTP)	Write an automated functional test to send an e-mail message from the application.
Web service	Use a tool like SoapAPI to submit a Web service and verify the output.
Web container(s)	Verify all container services are operating correctly.
Lightweight Directory Access Protocol (LDAP)	Using the application, authenticate via LDAP.
Logging	Write a test that writes a log using the application's logging mechanism.

Automated tests aren't just for testing user functionality. By creating a suite that focuses on deployment tests, you can verify the efficacy of the deployment, reducing downstream errors and development costs.

Rolling back all deployment changes

Name: Environment Rollback

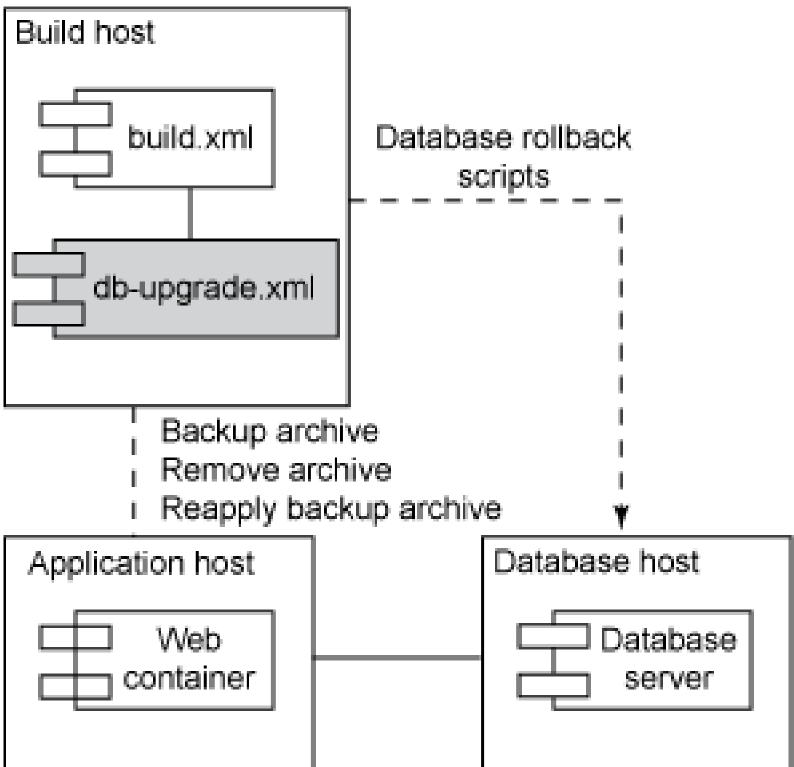
Pattern: Provide an automated Single Command rollback of changes after an unsuccessful deployment.

Antipatterns: Manually rolling back application and database changes.

Figure 7 illustrates rolling back database changes — using Database Upgrade — along with the automation processes for rolling back a Web deployment:

Figure 7. Rolling back deployment changes

© Copyright IBM Corporation 1994, 2008. All rights reserved.



Whether or not you're automating deployments, it's nice to have a way to roll back changes when a deployment goes wrong. In some cases, erroneous changes can result in a system outage costing an organization millions of dollars. To perform an Environment Rollback, you need to get the target environment back into the state it was in before the deployment. To do this, you essentially need a rollback script for every change. A Web deployment often requires more changes to roll back. One example of an Environment Rollback is to copy the archive (for example, a WAR file) prior to the deployment and provide rollback database scripts for each change. You also need to reapply the configuration changes applied to the Web container.

Listing 6 demonstrates an example of providing a rollback statement for each roll-forward statement using LiquiBase. I'm adding a new table called `brewery` while providing a corresponding `dropTable` rollback statement.

Listing 6. Provide rollback process when applying incremental data upgrades

```
<changeSet id="rollback-database-changes" author="bobama">
  <createTable tableName="brewery">
    <column name="id" type="int"/>
  </createTable>
  <rollback>
    <dropTable tableName="brewery"/>
  </rollback>
</changeSet>
```

This simple example is meant to be illustrative, not to trivialize rollback. Reverting to

© Copyright IBM Corporation 1994, 2008. All rights reserved.

a previous deployment is often a complex and time-consuming process to perform (and to automate). The time you invest in writing rollback scripts should be proportionate to the cost of the deployment failure.

Protect information from prying eyes

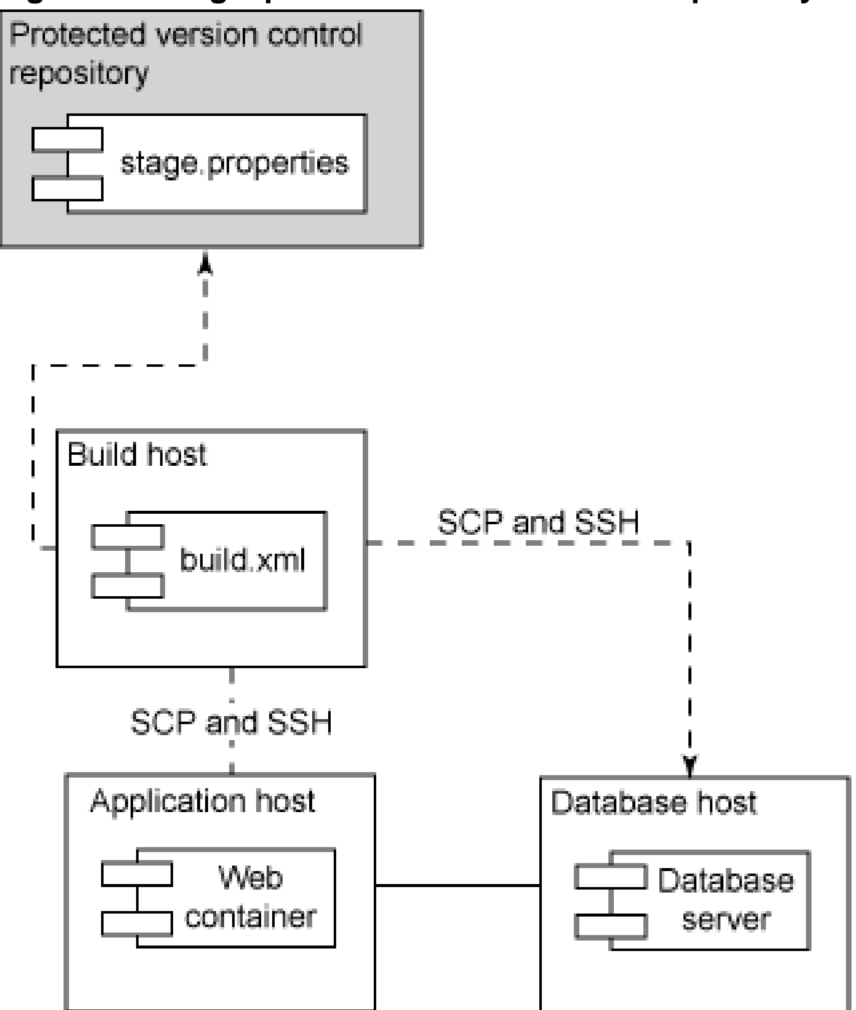
Name: Protected Files

Pattern: Using the repository, files are shared by authorized team members only.

Antipattern: Files are managed on team members' machines or stored on shared drives accessible by authorized team members.

Figure 8 shows a protected version-control repository used to host files that only authorized people or systems should access:

Figure 8. Using a protected version-control repository for sensitive files



© Copyright IBM Corporation 1994, 2008. All rights reserved.

In some cases, not all team members should have access to environment-specific data. However, keeping this information separate from the deployment scripts may prevent the scripts from executing. When discussing the Headless Execution pattern, I described using SSH keys with the Java Secure Channel tool to copy files and run remote commands securely without a human needing to enter commands. The properties you've used Externalized Configuration for are likely to contain data that not all team members should see. A technique I've used to ensure Headless Execution while protecting the data in the .properties files from prying eyes is to check these files into a protected repository.

In Listing 7, I'm configuring an Apache-hosted Subversion repository to deny access to all for a certain directory and then explicitly adding certain users:

Listing 7. Protecting a Subversion repository using Apache with Subversion

```
<DirectoryMatch "^/.*/(\.svn)/">
  Order deny,allow
  Deny from all
  Allow bobama,jbiden,hclinton
</DirectoryMatch>
```

By protecting access to the Subversion repository, a Scripted Deployment can access the properties as one of the allowed users without being prompted for a password, providing Headless Execution in a manner as defined by the SSH keys.

The one-click deployment

I've cataloged several more deployment-automation patterns than the 15 I've documented in this two-part article, but these 15 probably address 80 percent of the deployment situations I've encountered. Each of these patterns is intended to help get you to a literal one-click/single-command deployment for each and every target environment. I wish you many painless deployments!

That's all I wrote

This is my last article in the *Automation for the people* series. It's been a tremendous adventure sharing my experiences with you for more than two years. My goal with this series has always been to show how and why to automate myriad software development processes so that you're spending more time on interesting problems rather than futzing with repetitive, error-prone activities. In the series, I've demonstrated how to automate code reviews in order to refactor appropriately, upgrade an application database incrementally, apply Continuous Integration practices and tools, run automated tests with every change, generate GUI installers, create one-click deployments, automate the generation of developer documentation, perform dependency management, utilize version-control repositories, and use

© Copyright IBM Corporation 1994, 2008. All rights reserved.

various build scripts and tools effectively. I hope you've enjoyed reading the series as much as I've enjoyed writing it.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Resources

Learn

- "Deployment-automation patterns, Part 1" (Paul Duvall, developerWorks, January 2009): Patterns for one-click deployments.
- *Software Configuration Management Patterns* (Stephen Bercuzk and Brad Appleton, Addison-Wesley Professional, 2003): The authors describe the Repository pattern and many other software configuration management patterns.
- Patterns of Deployment: HP Labs' SmartFrog wiki includes a catalog of many deployment patterns.
- *Continuous Integration: Improving Software Quality and Reducing Risk* (Paul Duvall, Steve Matyas, and Andrew Glover, Addison-Wesley Signature Series, 2007): Chapter 8, *Continuous Deployment*, demonstrates examples of incorporating deployment into an automated process.
- *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps* (Mike Clark, The Pragmatic Programmers, 2004): Enjoy pragmatic, automatic, unattended software production that's reliable and accurate every time.
- *Release It!: Design and Deploy Production-Ready Software* (Michael T. Nygard, The Pragmatic Programmers, 2007): If you're a developer and don't want to be on call at 3 a.m. for the rest of your working life, this book will help.
- "Speed deployment with automation" (Paul Duvall, IBM developerWorks, January 2008): Leverage automation to move software through different environments quickly.
- "Continuous Integration anti-patterns, Part 2" (Paul Duvall, IBM developerWorks, March 2008): Make your life with CI easier by learning what not to do.
- "Hands-free database migration" (Paul Duvall, IBM developerWorks, August 2008): Use LiquiBase to manage database changes.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

- Ant: Download Ant and start building software in a predictable and repeatable manner.
- JSch: Download Java Secure Channel for secure communication.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

- Selenium: Download Selenium to perform deployment-specific functional testing.
- LiquiBase: Download LiquiBase to begin performing automated database migrations.
- Abbot: Download Abbot to begin performing deployment-centric functional testing.

Discuss

- Improve Your Code Quality discussion forum: Regular developerWorks contributor Andrew Glover brings his considerable expertise as a consultant focused on improving code quality to this moderated discussion forum.
- Accelerate development space: Regular developerWorks contributor Andrew Glover hosts a one-stop portal for all things related to developer testing, Continuous Integration, code metrics, and refactoring.
- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Paul Duvall
Paul Duvall is the CTO of Stelligent, a consultancy that helps clients delivery production-ready software. He is the co-author of the Addison-Wesley Signature Series book, *Continuous Integration: Improving Software Quality and Reducing Risk* (Addison-Wesley Professional, 2007; Jolt Award 2008 winner). He also contributed to the *UML 2 Toolkit* (Wiley, 2003) and the *No Fluff Just Stuff Anthology* (Pragmatic Programmers, 2007).

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

© Copyright IBM Corporation 1994, 2008. All rights reserved.