Built from the ground up, with **Continuous Delivery** at its heart

- Deployment **should be a Snap**
- Visualize and **share your delivery process** with ease
- Parallelize builds for **faster feedback**
- Automatic **PR Integration & Branch Tracking**

@snap_ci | snap-ci.com

#ohsnap
AUTOMATE ALL
#ohsnap
THE THINGS
#ohsnap
TEST EVERYTHING THAT COULD BREAK
#ohsnap
NO SNOWFLAKES
Infrastructure management + deploy tested artifacts
FAIL FAST
Continuous Integration + trunk based development

snap

Build, test and deploy in the cloud.

CONTENTS

» About Continuous Integration
» Build Software at Every Change
» Patterns and Anti-patterns
» Version Control
» Build Management
» Build Practices and more...

Continuous Integration: Patterns and Anti-Patterns

By Paul M. Duvall

ABOUT CONTINUOUS INTEGRATION

Continuous Integration (CI) is the process of building software with every change committed to a project's version control repository.

CI can be explained via patterns (i.e., a solution to a problem in a particular context) and anti-patterns (i.e., ineffective approaches sometimes used to "fix" the particular problem) associated with the process. Anti-patterns are solutions that appear to be beneficial, but, in the end, they tend to produce adverse effects. They are not necessarily bad practices, but can produce unintended results when compared to implementing the pattern.

CONTINUOUS INTEGRATION

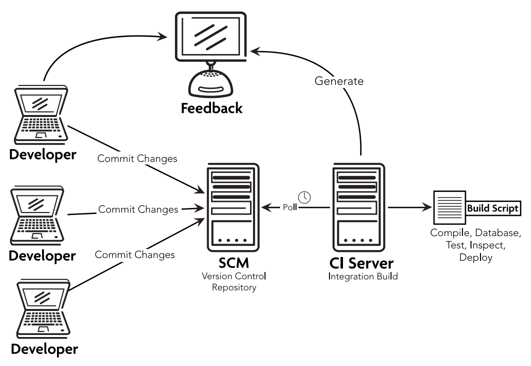
While the conventional use of the term Continuous Integration generally refers to the "build and test" cycle, this Refcard expands on the notion of CI to include concepts such as deployment and provisioning. The end result is learning whether you are capable of delivering working software with every source change.

Pattern	Run a software build with every change applied to the Repository.
Anti-Patterns	Scheduled builds, nightly builds, building periodically, building exclusively on developer's machines, not building at all.

BUILD SOFTWARE AT EVERY CHANGE

A CI scenario starts with the developer committing source code to the repository. There are four features required for CI:

- A connection to a version control repository
- An automated build script
- Some sort of feedback mechanism (such as email)
- A process for integrating the source code changes (manual or CI server)



The following table contains a summary of all the patterns covered in this Refcard:

PATTERN	DESCRIPTION
Private Workspace	Develop software in a Private Workspace to isolate changes.
Repository	Commit all files to a version-control repository.
Mainline	Develop on a mainline to minimize merging and to manage active code lines.
Branching Policy	Develop software within a system that utilizes multiple branches.
Task-Level Commit	Organize source-code changes by task-oriented units of work and submit changes as a Task-Level Commit.
Label Build	Label the build with unique name.
Automated Build	Automate all activities to build software from source without manual configuration.
Automated Build Slave Setup	Automate the setup of your build slaves. If that is not possible, reduce pre-installed tool dependencies to the bare minimum.
Binary Integrity	For each tagged deployment, use the same deployment package (e.g. WAR or EAR) in each target environment.
Dependency Management	Centralize all dependent libraries.
Consistent Directories	Create a simple, yet well-defined directory structure.
Template Verifier	Create a single template file that all target environment properties are based on.

snap
Build, test and deploy in the cloud.

@snap_ci | snap-ci.com

Built from the ground up, with **Continuous Delivery** at its heart

PATTERN	DESCRIPTION
Staged Deployments	Run automated deployments into different target environments.
Pre-Merge Build	Perform a Pre-merge Build before committing changes to the Repository.
Integration Build	Perform an Integration Build periodically, continually, etc.
Continuous Feedback	Send automated feedback from CI server to development team.
Expeditious Fixes	Fix build errors as soon as they occur.
Developer Documentation	Generate developer documentation with builds based on checked-in source code.
Independent Build	Separate build scripts from the IDE.
Single Command	Ensure all build and deployment processes can be run through a single command.
Dedicated Resources	Run builds on a separate dedicated machine or cloud service.
Externalize Configuration	Externalize all variable values from the application configuration into build-time properties.
Tokenize Configuration	Enter token values into configuration files and then replace during the Scripted Deployment.
Protected Configuration	Authorize only necessary team members to share files.
Scripted Database	Script all database actions.
Database Sandbox	Create a lightweight version of your database.
Database Upgrade	Use scripts and the database to apply incremental changes in each target environment.
Automated Tests	Write an automated test for each unique path.
Categorize Tests	Categorize tests by type.
Continuous Inspection	Run automated code analysis to find common problems.
Build Quality Threshold	Use thresholds to notify team members of code aberrations.
Automated Smoke Test	Script self-testing capabilities into Scripted Deployments.
Scripted Deployment	Write all deployment processes in a script.
Headless Execution	Securely interface with multiple machines without typing a command.
Unified Deployment	Create a single deployment script capable of running on different platforms and target environments.
Disposable Container	Automate the installation and configuration of Web and database containers.
Remote Deployment	Use a centralized machine or cluster to deploy software to multiple target environments.
Environment Rollback	Provide an automated Single Command rollback of changes after an unsuccessful deployment.
Continuous Deployment	Deploy software with every change applied to the project's version control repository.
Single-Command Provisioning	Run a single command to provision target environment.
Environment-independent	Separate the environment-specific configuration from the application.

PATTERNS AND ANTI-PATTERNS

VERSION CONTROL

The patterns in this section were originally described in the book Software Configuration Management Patterns (Addison-Wesley, 2003, Berczuk and Appleton), except for "Label Build":

PATTERN	DESCRIPTION
Private Workspace	Prevent integration issues from distracting you, and from your changes causing others problems by developing in a Private Workspace.
Repository	All files are committed to version-control repository—in the deployment context, all of the configuration files and tools.
Mainline	Minimize merging and keep the number of active code lines manageable by developing on a Mainline.
Branching Policy	The policy should be brief, and should spell out the "rules of the road" for the branch.

TASK-LEVEL COMMIT

Pattern	Organize source code changes by task-oriented units of work and submit changes as a Task Level Commit.
Anti-Patterns	Keeping changes local to developer for several days and stacking up changes until committing all changes. This often causes build failures or requires complex troubleshooting.

LABEL BUILD

Pattern	Label the build with unique name so that you can run the same build at another time.
Anti-Patterns	Not labeling builds, using revisions or branches as "labels."

```
<path id="svn.classpath">
  <fileset dir="${lib.dir}">
    <include name="**/*.jar" />
  </fileset>
</path>
<taskdef name="svn" classpathref="svn.classpath"
classname="org. tigris.subversion.svnant.SvnTask" />

<target name="create-tag-from-trunk">
  <svn username="jhancock" password="S!gnhere">
    <copy srcUrl="https://brewery-ci.googlecode.com/svn/
    trunk" destUrl="https://brewery-ci.googlecode.com/svn/
    tags/brewery-1.0.0" message="Tag created by jhancock
    on ${TODAY}" />
  </svn>
</target>
```

BUILD MANAGEMENT

AUTOMATED BUILD

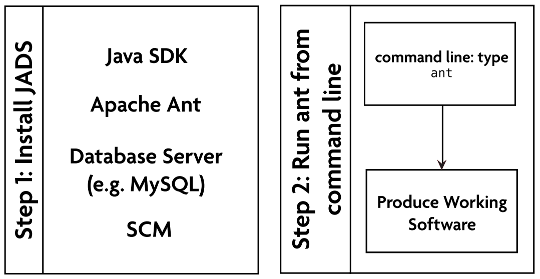
Pattern	Automate all activities to build software from source without manual configuration. Create build scripts that are decoupled from IDEs. Later, these build scripts will be executed by a CI system so that software is built at every change.
Anti-Patterns	Continually repeating the same processes with manual builds or partially-automated builds requiring numerous manual configuration activities.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<project name="brewery" default="all" basedir=".">
  <target name="clean" />
  <target name="svn-update" />
  <target name="all" depends="clean,svn-update" />
  <target name="compile-src" />
  <target name="compile-tests" />
  <target name="integrate-database" />
  <target name="run-tests" />
  <target name="run-inspections" />
  <target name="package" />
  <target name="deploy" />
</project>
```

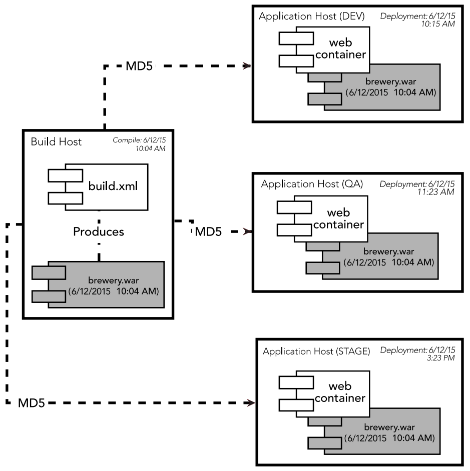
AUTOMATED BUILD SLAVE SETUP

Pattern	Automate the provisioning of your build slaves, if possible. Otherwise, reduce pre-installed tool dependencies to the bare minimum. Eliminate required environment variables from the Automated Build and Scripted Deployment.
Anti-Patterns	Requiring developer to define and configure environment variables. Require developer to install numerous tools in order for the build/deployment to work.



BINARY INTEGRITY

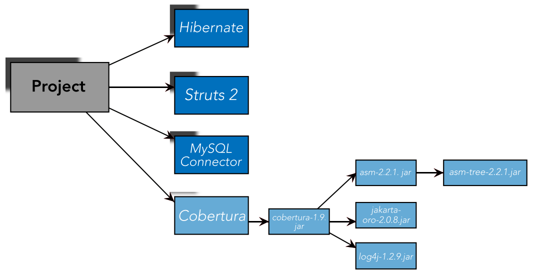
Pattern	Use the same deployment package (e.g. WAR or EAR) in each target environment for each tagged deployment.
Anti-Patterns	Separating compilation for each target environment on the same tag.


Build Host
Compile: 6/12/15 10:04 AM
build.xml
Produces
brewery.war (6/12/2015 10:04 AM)
MD5
MD5
MD5
Application Host (DEV)
Deployment: 6/12/15 10:15 AM
web container
brewery.war (6/12/2015 10:04 AM)
Application Host (QA)
Deployment: 6/12/15 11:23 AM
web container
brewery.war (6/12/2015 10:04 AM)
Application Host (STAGE)
Deployment: 6/12/15 3:23 PM
web container
brewery.war (6/12/2015 10:04 AM)

DEPENDENCY MANAGEMENT

Pattern	Centralize all dependent libraries to reduce bloat, classpath problems, and repetition of the same dependent libraries and transitive dependencies from project to project.
Anti-Patterns	Having multiple copies of the same JAR dependencies in each and every project. Redefining the same information for each project. Classpath hell!

Tools such as Ivy and Maven can be used for managing dependencies.

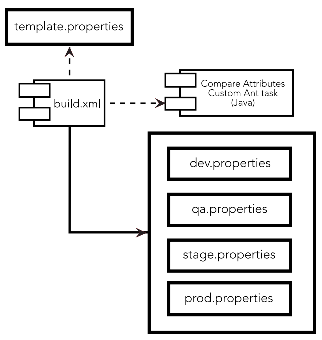


CONSISTENT DIRECTORIES

Pattern	Create a simple, yet well-defined directory structure to optimize software builds and increase cross-project knowledge transfer.
Anti-Patterns	Putting code, documentation, and large files in the same parent directory structure, leading to long-running builds.

TEMPLATE VERIFIER

Pattern	Create a single template file that all target environment properties are based on.
Anti-Patterns	Using manual verification or trial and error (when deployment fails, check the logs); or keeping files “hidden” on a machine.



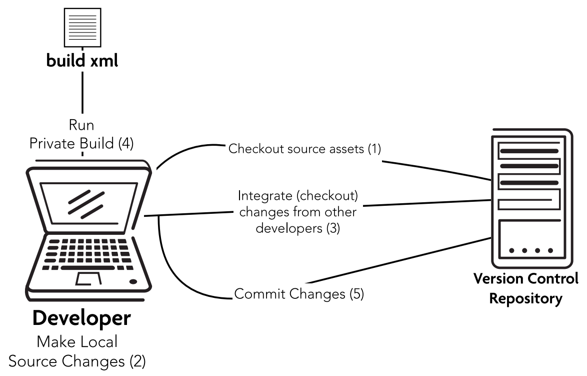
STAGED DEPLOYMENTS

Pattern	Run deployments into different target environments using the Remote Deployment pattern.
Anti-Patterns	Deploying directly to production.

BUILD PRACTICES

PRE-MERGE BUILD

Pattern	Verify your changes will not break the Integration Build by performing a Pre-merge Build—either locally or using Continuous Integration.
Anti-Patterns	Checking in changes to version-control repository without running a build on developer's workstation.


build xml
Run Private Build (4)
Checkout source assets (1)
Integrate (checkout) changes from other developers (3)
Commit Changes (5)
Version Control Repository
Developer
Make Local Source Changes (2)

INTEGRATION BUILD

Pattern	Ensure that your code base always builds reliably by doing an Integration Build periodically.
Anti-Patterns	"Works on My Machine" (WOMM). Continuous Compilation.

CONTINUOUS FEEDBACK

Pattern	Send automated feedback from CI server to development team.
Anti-Patterns	Sending minimal feedback, which prevents action from occurring. Receiving spam feedback, which causes people to ignore messages.
Examples	Email, RSS, SMS, X10, Monitors, Web Notifiers, Campfire, Slack, HipChat

EXPEDITIOUS FIXES

Pattern	Fix build errors as soon as they occur.
Anti-Patterns	Allowing problems to stack up (build entropy), causing more complex troubleshooting; some claim that "CI is the problem."

Fix broken builds immediately	Although it is the team's responsibility, the developer who recently committed code must be involved in fixing the failed build.
Run private builds	To prevent Integration failures, get changes from other developers by getting the latest changes from the repository, and run a full integration build locally, known as a Private Build.
Avoid getting broken code	If the build has failed, you will lose time if you get code from the Repository. Wait for the change or help the developer(s) fix the build failure and then get the latest code.

DEVELOPER DOCUMENTATION

Pattern	Generate developer documentation with builds (at appropriate intervals) based on checked-in source code.
Anti-Patterns	Manually generating developer documentation, periodically. This is both a burdensome process and one in which the information becomes useless quickly because it does not reflect the checked-in source code.

Automating your documentation's generation will help you keep it up-to-date and thereby make it more useful for your software's users.

SCHEMASPY

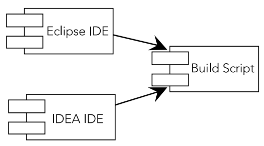
```
<property name="reports.dir" value="${basedir}" />
<java jar="schemaSpy_3.1.1.jar" output="${reports.dir}/
output.log" error="${reports.dir}/error.log" fork="true">
  <arg line="-t mysql" />
  <arg line="-host localhost" />
  <arg line="-port 3306" />
  <arg line="-db brewery" />
  <arg line="-u root" />
  <arg line="-p sa" />
  <arg line="-cp mysql-connector-java-5.0.5-bin.jar" />
  <arg line="-o ${reports.dir}" />
</java>
```

Note: 'Integration Build' is also from Berczuk and Appleton's book Software Configuration Management Patterns (Addison-Wesley, 2003, Berczuk and Appleton)

BUILD CONFIGURATION

INDEPENDENT BUILD

Pattern	Separate build scripts from the IDE. Create build scripts that are decoupled from IDEs. Later, these build scripts will be executed by a CI system so that software is built at every change.
Anti-Patterns	Relying on IDE settings for Automated Build. Build cannot run from the command line.



SINGLE COMMAND

Pattern	Ensure all build and deployment processes can be run through a single command. This makes it easier to use, reduces deployment complexities and ensures a Headless Execution of the deployment process. Deployers, or headless processes, can type a single command to generate working software for users.
Anti-Patterns	Requiring people to enter multiple commands and procedures during the deployment process, such as copying files, modifying configuration files, restarting a server, setting passwords, and other repetitive, error-prone actions.

Single-command deployment execution using Ant:

```
ant-Dproperties.file=$USERHOME/projects/petstore/
properties/dev-install.properties deploy:remote:install
```

DEDICATED RESOURCES

Pattern	Run builds on a separate dedicated machine or cloud service.
Anti-Patterns	Relying on existing environmental and configuration assumptions (can lead to the “but it works on my machine problem”).

When creating an integration build machine consider the following:

Recommended system resources	Increase hardware resources for an integration build machine rather than wasting time waiting for slow builds.
All software assets in the version control repository	See the Repository pattern.
Clean environment	CI process removes any code dependencies on the integration environment. Automated build must set test data and any other configuration elements to a known state.

EXTERNALIZE CONFIGURATION

Pattern	Externalize all variable values from the application configuration into build-time properties.
Anti-Patterns	Hardcoding these values, manually, for each of the target environments, or using GUI tools to do the same.

Example properties that are external to application-specific files:

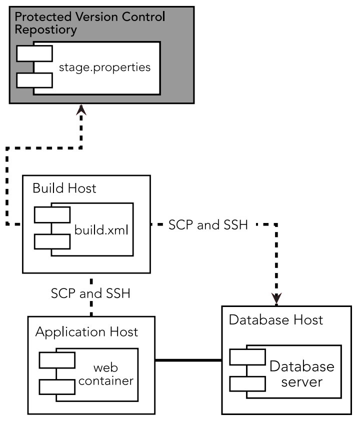
```
authentication.type=db application.url=http://${tomcat.
server.hostname}:${tomcat.server. port}/brewery-webapp
database.type=mysql
database.server=localhost
database.port=3306
database.name=mydb
database.user=myuser!
database.password=mypa$$! database.
url=jdbc:mysql://${database.server}:${database.
port}/${database.name}
tomcat.server.hostname=localhost
tomcat.server.name=default
tomcat.web.password=pa$$123!
tomcat.cobraorb.port=12748
```

TOKENIZE CONFIGURATION

Pattern	Enter token values into configuration files and then replace them during the Scripted Deployment based on Externalized Configuration properties checked into Repository.
Anti-Patterns	Entering target-specific data into configuration files in each environment.

PROTECTED CONFIGURATION

Pattern	Allow only authorized team members to share files. Share files using the repository.
Anti-Patterns	Files are managed on team members’ machines or stored on shared drives accessible by authorized team members.



DATABASE

SCRIPTED DATABASE

Pattern	Script all database actions.
Anti-Patterns	Migrating a database manually and late in the development cycle (this is painful and expensive).

Script all DDL and DML so that database changes can be run from the command line. Use a version-control repository to manage all database-related changes (i.e. refer to the pattern).

```
<target name="db:create" depends="filterSqlFiles"
description="Create the database definition">
  <sql driver="com.mysql.jdbc.Driver" url="jdbc:mysql://
  localhost:3306/" userid="root" password="root"
  classpathref="db.lib.path" src="${filtered.sql.dir}/
  database-definition.sql" delimiter="//" />
</target>
```

DATABASE SANDBOX

Pattern	* Create a lightweight version of your database (only enough records to test functionality) * Use this lightweight DML to populate local database sandboxes for each developer * Use this data in development environments to expedite test execution
Anti-Patterns	Sharing development database.

Give each developer, tester, or test user a separate database instance. Install a lightweight database server in each user's test environment (e.g. MySQL, Personal Oracle), which can be installed on the user's private workstation, on a shared test server, or on a dedicated "virtual server" running on a shared server.

DATABASE UPGRADE

Pattern	Use scripts and the database to apply incremental changes in each target environment, which provides a centrally managed and scripted process to applying incremental changes to the database.
Anti-Patterns	Manually applying database and data changes in each target environment.

Running a custom SQL file from a LiquiBase change set:

```
build.xml
<updateDatabase changeLogFile="db.change.xml" driver="org.
apache.derby.jdbc.EmbeddedDriver" url="jdbc:derby:brewery"
username="" password="" dropFirst="true"
classpathref="project.class.path" />

db.change.xml
<changeSet id="1" author="phenry">
  <sqlFile path="insert-data.sql" />
</changeSet>
```

TESTING AND CODE QUALITY

AUTOMATED TESTS

Pattern	Write an automated test for each unique path.
Anti-Patterns	Not running tests, no regression tests, manual testing.

Examples	*A Simple Unit Test*
	`public void setUp() {`
	`beerService = new BeerDaoStub();`
	`}`
	
	`public void testUnitGetBeer() {`
	`Collection beers = beerService.findAll();`
	`assertTrue(beers != null && beers.size()`
	`> 0);`
	`}`
	
	Running a Unit Test in Ant
	`<junit fork="yes" haltonfailure="true"`
	`dir="${basedir}" printsummary="yes">`
	`  <classpath refid="test.class.path" />`
	`  <classpath refid="project.class.path"`
	`  />`
	`  <formatter type="plain" usefile="true"`
	`  />`
	`  <formatter type="xml" usefile="true" />`
	`  <batchtest fork="yes"`
	`  todir="${logs.junit.dir}">`
	`    <fileset dir="${test.unit.dir}">`
	`      <patternset refid="test.sources.`
	`      pattern" />`
	`    </fileset>`
	`  </batchtest>`
	`</junit>`

CATEGORIZE TESTS

Pattern	Categorize tests by type (your builds become more agile, tests can be run more frequently, and tests no longer take hours to complete).
Anti-Patterns	Not categorizing tests—tests take hours to run, leading to excessive wait times and increased expense.

CONTINUOUS INSPECTION

Pattern	Run automated code analysis to find common problems. Have these tools run as part of continuous integration or periodic builds.
Anti-Patterns	Reviewing through long, manual code reviews, or not reviewing code at all.

EXAMPLES:

CheckStyle

```
<taskdef resource="checkstyletask.properties"
classpath="${checkstyle.jar}" />
<checkstyle config="${basedir}/cs-rules.xml"
failOnViolation="false">
  <formatter toFile="${checkstyle.data.
  file}" type="xml" />
    <fileset casesensitive="yes"
    dir="${src.dir}" includes="**/*java" />
</checkstyle>

<xslt taskname="checkstyle"
in="${checkstyle.data.file}"
out="${checkstyle.report.file}"
style="${checkstyle.xsl.file}" />
```

BUILD QUALITY THRESHOLD

Pattern	Notify team members of code aberrations such as low code coverage or high cyclomatic complexity. Fail a build when a project rule is violated. Use continuous feedback mechanisms to notify team members.
Anti-Patterns	Reviewing through lengthy manual code reviews. Learning of code quality issues later in the development cycle.

```
<module name="CyclomaticComplexity">
  <property name="max" value="10" />
</module>
```

AUTOMATED SMOKE TEST

Pattern	Script self-testing capabilities into Scripted Deployments.
Anti-Patterns	Verifying deployments by running through manual functional tests that do not focus on deployment-specific aspects. No deployment tests are run.

The table below describes examples of the types of test that might be run as part of a Deployment Test smoke suite:

EXAMPLE TEST TYPE	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 email message from the application.
Web service	Use a tool like SOAP API to submit a Web service and verify the output.
Web container(s)	Verify all container services are operating correctly.

EXAMPLE TEST TYPE	DESCRIPTION
Lightweight Directory Access Protocol (LDAP)	Use the application to authenticate via LDAP.
Logging	Write a test that writes a log using the application's logging mechanism.

DEPLOYMENT

SCRIPTED DEPLOYMENT

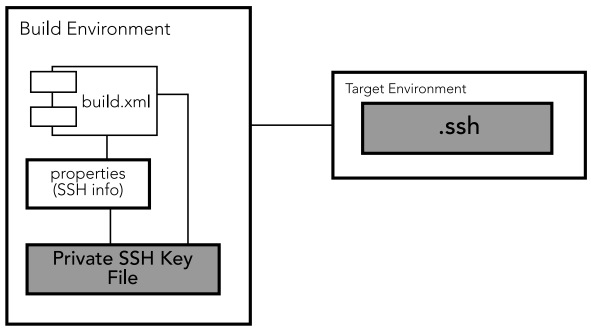
Pattern	Write all deployment processes in a script.
Anti-Patterns	Manually installing and configuring a Web container. Use of the GUI-based administration tool provided by the container to modify the container based on a specific environment.

```
<available file="@{tomcat.home}/server/@{tomcat.server.
name}/bin" property="tomcat.bin.exists" />
<if>
   <isset property="tomcat.bin.exists" />
<then>
   <echo message="Starting tomcat instance at @{tomcat.
   home} with start_tomcat" />
   <exec executable="@{tomcat.home}/server/@{tomcat.server.
   name}/bin/start_tomcat" osfamily="unix" />
</then> <else>
   <echo message="Starting tomcat instance at @{tomcat.
   home} with startup.sh" />
   <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>

   <exec executable="sh" osfamily="unix" dir="@{tomcat.
   home}/bin" spawn="true">
      <env key="NOPAUSE" value="true" />
      <arg line="startup.sh" />
   </exec>
   <exec osfamily="windows" executable="cmd" dir="@{tomcat.
   home}/ bin" spawn="true" >
   <env key="NOPAUSE" value="true" /> <arg line="/c
   startup.sh" />
   </exec>
   <sleep seconds="15" />
   </else>
</if>
```

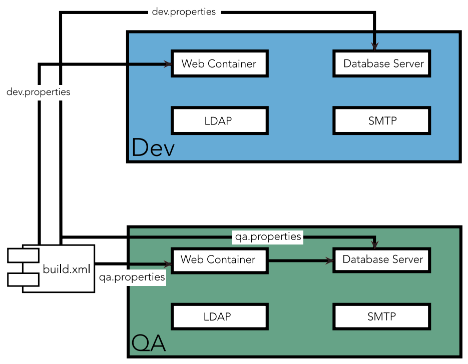
HEADLESS EXECUTION

Pattern	Interface securely with multiple machines without typing a command.
Anti-Patterns	People manually access machines by logging into each of the machines as different users; then they copy files, configure values, and so on.



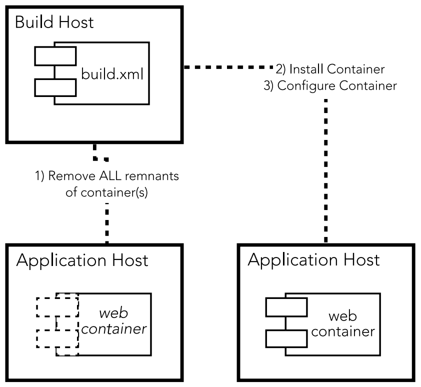
UNIFIED DEPLOYMENT

Pattern	Create a single deployment script capable of running on different platforms and target environments.
Anti-Patterns	Some may use a different deployment script for each target environment or even for a specific machine.



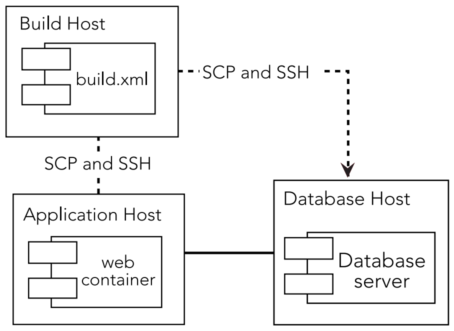
DISPOSABLE CONTAINER

Pattern	Automate the installation and configuration of Web and database containers by decoupling installation and configuration.
Anti-Patterns	Manually install and configure containers into each target environment.



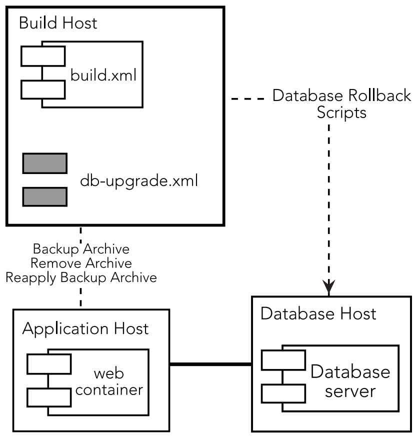
REMOTE DEPLOYMENT

Pattern	Use a centralized machine or cluster to deploy software to multiple target environments.
Anti-Patterns	Manually applying deployments locally in each target environment.



ENVIRONMENT ROLLBACK

Pattern	Provide an automated Single Command rollback of changes after an unsuccessful deployment.
Anti-Patterns	Manually rolling back application and database changes.



CONTINUOUS DEPLOYMENT

Pattern	Deploy software with every change applied to the project's version control repository.
Anti-Patterns	Deploying periodically. Manual deployments. Manual configuration of target environments.

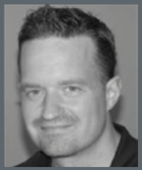
SINGLE-COMMAND PROVISIONING

Pattern	Run a single command or click a button to provision target environment.
Anti-Patterns	Numerous manual and error-prone steps, often performed by other teams, leading to delays and target environment inconsistencies making errors difficult to troubleshoot.

ENVIRONMENT-INDEPENDENT BUILDS

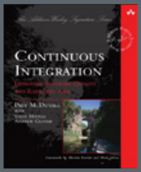
Pattern	Separate the environment-specific configuration from the application deliverable.
Anti-Patterns	Saving off preconfigured images whose configuration has not been automated.

ABOUT THE AUTHOR



Paul M. Duvall is the CEO of Stelligent, a firm that helps clients create production-ready software every day. A featured speaker at many leading software conferences, he has worked in virtually every role on software projects: developer, project manager, architect, and tester. He is the principal author of Continuous Integration: Improving Software Quality and Reducing Risk (Addison-Wesley, 2007) and a 2008 Jolt Award Winner. Paul contributed to the UML 2 Toolkit (Wiley, 2003), wrote a series for IBM developerWorks called "Automation for the People," and contributed a chapter to No Fluff Just Stuff Anthology: The 2007 Edition (Pragmatic Programmers, 2007). He is passionate about automating software development and release processes and actively blogs on IntegrateButton.com and TestEarly.com.

RECOMMENDED BOOK



For any software developer who has spent days in "integration hell," cobbling together myriad software components, Continuous Integration: Improving Software Quality and Reducing Risk illustrates how to transform integration from a necessary evil into an everyday part of the development process. The key, as the authors show, is to integrate regularly and often using continuous integration (CI) practices and techniques.

BUY NOW

Some of the concepts and material in this Refcard were adapted from:

- **Continuous Integration: Improving Software Quality and Reducing Risk**, by Paul M. Duvall (Addison-Wesley, 2007) - http://www.amazon.com/gp/product0321336380/?tag=integratecom-20
- **IBM developerWorks series Automation for the people**, by Paul Duvall - http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=automation+people

CREDITS:
Editor: G. Ryan Spain | Designer: Yassee Mohebbi | Production: Chris Smith | Sponsor Relations: Brandon Rosser | Marketing: Chelsea Bosworth | Reviewer: Andrew Phillips

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0 $7.95