Using Ant to Solve Problems Posed by Frequent Deployments

Steve Shaw
Intelliware Development Inc
1709 Bloor St W., Suite 200
Toronto, Ontario
M6P 4E5, Canada
+1 416 762 0032
steve@intelliware.ca

Abstract. Deploying a system in an agile development environment presents its own set of challenges. If it is true that development cycles are short and that every release is as small as possible, then we are going to be releasing software much more frequently than with other methodologies. Related to this is the concept that the system is never finished, and deployments have to occur throughout the lifetime of the system. This paper examines some of the problems posed by this type of deployment environment and suggests how Ant can be used to solve them. An appendix describes concrete solutions to problems encountered on a real-life medium-sized project.

1. Introduction

This paper describes problems that were encountered during the development of a web-based system, but an attempt has been made to generalize wherever possible. Solutions are suggested, with the understanding that no solution is going to be appropriate for all, or even most, environments.

2. Ant is my very best friend

The Jakarta Project[1] creates a variety of open-source Java tools. Among these is Ant, a build tool. It is easy to use, relatively easy to extend, and provides enough functionality out of the box to handle many build issues. Most of the solutions suggested, and all of the concrete solutions presented, rely on the use of Ant.

Ant has several attractive features. All Ant scripts are written as XML, which is quickly becoming the lingua franca of distributed service development and as such is often familiar to developers. It is designed to be platform independent, so the little (but aggravating) things like differing file separators become unimportant. It's fully extensible with a little bit of Java knowledge. And it boasts an array of features that cover most of the tasks that a build and deploy process may need.

Intelliware has been using Ant as a build management tool for over a year. It is the core component of the deployment solution that Intelliware is currently using. There are Ant targets[2] to handle just about every aspect of the build and deploy process. Use of Ant allows for evolutionary change to the deploy process: most of the scripts and targets used for the deploy process grew out of the build targets that were defined early in the projects. These same scripts, with some slight modifications, are also used in the production environment to start services and set up resources. Ant has also proven useful in other situations when platform independence and flexibility are required: code integration, setting up batch jobs, and integration testing.

3. Issues

Many of the issues described in this paper apply to any kind of software deployment[3]. They become more important in an agile environment because of the frequency of deployment. In traditional development environments, deployments occur less frequently, and difficulties can be handled by manual intervention.

[1] More information on Ant and other Jakarta projects can be found at `jakarta.apache.org`.
[2] An Ant target is a set of tasks that are executed together. An Ant task is a single unit of work, either built-in or user-defined. Built-in tasks include `javac`, to compile Java code, and `junit`, to run JUnit test suites.
[3] Deployment is an overloaded term. It is used in this paper to refer to the process of distributing executables, setting up resources, and performing other tasks that are necessary to install a piece of software.

3.1 Different types of deployment

A robust deployment strategy allows for different flavours of deployments. This is especially true for test-centric development methodologies: it is often desirable to deploy the system to development machines, as well as to the nightly build[4] machine. Each type of deployment target poses different challenges.

Development machine. When running an entire system on a developer's machine, it becomes necessary to run each component (or a dummied-out version of that component) on a single machine. Actually doing the deployment should not be a problem, since the development machines will likely have a full set of tools. In fact, the "deployment" in this case can be as simple as building the application directly on the machine.

Nightly build machine. The nightly build machine (or machines) will likely contain the full set of development tools. Whether this machine is identical to the developers' workstations will depend on project needs and target environments. Another variable is whether the nightly build targets a single machine, or whether various services are spread across different targets in order to more accurately depict the eventual production environment. It can be easier to run unit tests against processes running on a single machine, but more robust integration testing might call for the use of multiple machines.

Everything else. Other types of deployments can be thought of as "real deployments". In these cases, the application is usually not built from scratch, but is distributed in some compiled format – JAR files, for example. It is unlikely (and undesirable) that these targets have the full set of development tools available for use. Whenever possible, the components should be compiled and packaged in a single spot.

Often the production environment calls for a multi-machine deployment. This type of deployment can be configured in such a way that changing one set of properties allows for different deployment targets. For example, the same deployment script could be used to distribute the application to three different sets of servers: one for end-user testing, one for pre-production staging, and a final set of production servers. Only the properties file driving the deployment needs to change. This file indicates the location of each component and updates resource directories accordingly.

Suggestion:

- When defining the deployment process, draw clear lines between the responsibilities of different agents. Where does the "build" end and the "deploy" begin? Or are they part of the same process? As the production environment becomes more complicated, these clear delineations of responsibility will make evolution of the system much easier.

3.2 Platform dependencies

It is not unusual for the eventual target environment of a system to differ from the development environment. The most common difference is operating system, although it is possible that such things as database managers and other system resources will also be different.

These platform differences have to be addressed during development, and are usually handled by using as many platform-agnostic tools and resources as possible. There are also implications during the deployment process: each problem mentioned here will have to be solved for each target platform.

Suggestion:

- Use Ant. It is specifically designed to be platform independent. If there isn't an Ant task that does what is needed, write a custom task or use the exec task to kick off a system command (different commands can be specified for different operating systems).

[4] The "nightly build" refers to the regularly-scheduled build, often run at night to reduce the impact on users of the system. Also called the daily build. It is common for the daily/nightly build script to be run more than once a day as required by the development team.

3.3 Defining system resources

One of the first problems with the deployment of any system is the definition of system resources. Examples of the resources that may need to be defined are databases, message queues, and resource directories such as JNDI.

A joy of working on a small agile project is that individual developers often have the authority to change the resource definitions to meet their needs, without having to schedule resource drops. The other developers will simply pick up the changed resource definitions on their next integration. As long as a working system can be built from scratch, the integration is considered to be successful.

Transient resources. Resources such as message queues[5] and resource directories can be handled without difficulty. These are "transient" resources because they can usually be torn down and redefined without affecting the state of the system (assuming the system is not currently active[6]). It is easy to arrange for the deploy process to destroy the existing resource and recreate it to current specifications during each build.

Databases. It is more difficult to handle databases, because the data they contain represent the current state of the system. Deploying a new database schema to development machines is simple enough – it's usually sufficient to drop the tables, recreate them, and repopulate them with a set of reasonable initial data.

However, a database on a testing or production platform cannot always be dropped and repopulated without regard to the data that it contains. More complex methods of updating a database schema while maintaining the integrity of the data must be utilized. This is not a trivial problem, as the database may have gone through several sets of changes since the last time a release was deployed to the target platform. Database changes that involve changing existing data are awkward. As Kent Beck[1] wrote about this and similar problems, "If preproduction weren't so dangerous, you'd keep from going into production forever".

Suggestions:

- Ant has a built-in task for handling JDBC calls, so database setup can be handled simply by coding SQL as CDATA[7] in the Ant scripts. An alternative is to create a database command script as supported by a specific database manager, but this approach is less portable. (However, it would allow access to vendor-specific functions not usable through JDBC – which may or may not be a good thing.)
- Write custom Ant tasks to define resources. An example is the `bindname` task developed at Intelliware to handle JNDI registration. The task implements different JNDI solutions based on Ant properties.
- Use `exec` tasks for everything else. For example, an `exec` task can be used to define JMS queues. The task invokes the vendor-supplied batch or shell script used to define, delete and register JMS resources.

Outstanding problem:

- The issue of dealing with database changes while maintaining the integrity of data is not one that can be dealt with easily. Minimizing the number of post-release database changes is desirable, but often unrealistic given the evolutionary nature of agile development practices. Traditional deployments will often deal with this sort of issue by creating data migration scripts, which is not onerous if database changes are made once a year. When releasing something every six weeks, however, a little bit of automation might be called for.

3.4 Remote deployments and authentication

In the case of a multi-machine deployment, the application has to somehow distribute the components of the application to the target machines. This is not usually a huge problem on a local network, as there are a number of solutions that will work across different operating systems[8].

[5] It can be argued that message queues are not transient resources, especially if they have persistent messages. In that case, changes to queue definitions behave much as changes to database definitions – special effort must be made to ensure the integrity of the persistent messages is maintained.

[6] Deploying a new release of a system that must be constantly available presents challenges outside the scope of this paper.

[7] Character data included inline in the XML file.

[8] Samba, FTP, and a shared cvs server are three different ways to solve this particular problem.

The distribution of components becomes more difficult once the application leaves the development floor. The mechanisms used in the lab are often not applicable in production, where the deployment process has firewalls, subnets and DMZs[9] with which to contend. For example, using FTP or the Unix `rcp` (remote copy) commands to handle deployments on the local network might be acceptable, but both utilities are insecure and not appropriate for transferring sensitive data over a public network.

Related to this is the problem of authentication. If the application has components on a publicly-accessible machine (as is necessary for a web application), it is imperative that only trusted peers are able to update the components on that machine. This limits the choice of tools to use when deploying to remote machines.

Suggestion:

- Ant is not of much direct use here. Creating a task is one approach to handling these issues – but it is difficult to ensure that authentication security is sufficient. It may be more secure to rely on OS-specific remote authentication procedures, which are bound to be more thoroughly tested than homegrown code.

  On Unix/Linux, use `scp` and `ssh` (secure versions of `rcp` and `rsh`) whenever remote authentication and security are important. Use the Ant `exec` task to kick off the Unix commands, and substitute the Windows equivalent when necessary.

Outstanding problem:

- Is there a Windows solution for remote deployment and authentication? There are versions of `scp` and `ssh` that are available for Windows, but it would be preferable to use a native solution.

3.5 Deployment frequency and timing

The increased frequency of deployments is the primary challenge in deployment planning for agile methodologies. Problems that are simply annoyances on other projects can become important issues on agile projects as more developer hours are spent manually handling them. As well, agile developers are used to automating everything – it seems wrong when manual intervention is called for.

Another issue is that of the timing of deployments. Different types of deployment are going to occur at different intervals. The application may be built and "deployed" several times per day on each developer's machine, and should occur at least once per day on the nightly build machine.

Other deployments have to be scheduled more carefully. For instance, changing the testers' deployment environment without warning is bound to result in irreproducible bugs and angry testers. It is also undesirable to deploy to a production environment while the system is under heavy load, even if a backup is in place to handle user requests while the primary system is being upgraded.

Suggestions:

- When dealing with very frequent, low-impact deployments (e.g, on nightly build machines), have an scheduled automatic process kick off the build/deployment.
- Other kinds of deployments are best initiated manually after developers, testers and users have agreed that it's time for a new version of the system.

3.6 Stopping and starting services

Monolithic systems are largely a thing of the past. Today's systems usually consist of a variety of components. The systems are flexible and easier to maintain, but are more complex. Part of this complexity is the issue of starting and stopping services that act as components of the system. These "services" could be anything that has to be shut down before the new code is dropped, and restarted afterwards. Examples include web and application servers, and parts of the system that run in their own process space.

[9] Demilitarized zone, referring to the subnet(s) that exist between the safe local network and the unsafe Internet.

This is another example of a task that is relatively straightforward when the system is deployed onto a single target machine but becomes more difficult when the components are spread across several machines. This type of task will also vary from platform to platform.

Suggestions:

- Again, Ant's `exec` task is useful for starting up any services that the system depends on. Alternatively, the `java` task might be applicable in some circumstances[10].
- When starting or stopping services on remote machines, use the same kind of process that was set up for remote deployment. The same argument applies: secure system utilities are likely to be more secure than homemade solutions.

3.7 Deployment monitoring and testing

With an agile development process in place, production deployments may be performed every few weeks. It is important that the development team is satisfied that the deployed system is working properly. This problem can be solved with the same practice that solves many other problems: more testing.

Hopefully the unit tests and integration tests that are run against the nightly build will catch any bugs before they move into production, but that cannot be guaranteed. It is likely that these tests are not suitable for running against a live production system.

Before tests can be run against a deployed system, the deployment must have completed successfully. This implies that a monitoring system must be in place. When the deployed system resides on a single machine, it is a trivial matter to ensure that the deployment has completed. It can be difficult to track the progress of a deployment across multiple machines, especially if individual processes are being done asynchronously.

The tests that are run against the deployed production system are bound to be different than unit or integration tests. It is important that the tests do not modify the behavior or state of the system. Of course, it is difficult to ascertain whether a test has worked if it does not affect the state of the system.

Suggestions:

- Track the success of the deployment carefully. If a deployment fails, then the development team should be made aware of it as soon as it is appropriate – whether through a log message, email or page.
- Create a series of lightweight tests that act strictly as a sanity check. Such "smoke tests" do not do anything except invoke the system components in the least intrusive way possible – e.g., make sure that all the web pages of a site are up, that all the servers are responding to pings, and that all services are responding to "heartbeat" maintenance messages. This level of testing should be sufficient to ensure that the application was deployed properly, while the other levels of testing performed during development ensure that the system performs as intended.

4. Conclusion

The major challenge presented by deploying in an agile development environment is that all of the traditional deployment problems are faced much more often. It's often not worth the effort to automate processes that only occur once a year, but when deployment occurs on a monthly basis, it is well worth automating the process as much as possible.

Ant has proven to be a valuable tool in implementing the build and deployment strategies at Intelliware. The combination of its ease of use, impressive range of features, and extensibility has contributed to the success of several projects.

[10] For example, starting a Java program. It's not rocket science.

5. Previous Work

Fowler and Foemmel[2] discuss continuous integration and the importance of automating it. Ant is mentioned as the tool of choice, in combination with JUnit and a reasonable amount of customization. The paper is more heavily focused on "why" rather than "how", but its conclusions and recommendations are compatible with those of this paper.

Hightower and Lesiecki[3] extensively describe the use of Ant and other open-source tools within agile projects. The book does not specifically address deployment issues, but does touch on them in passing. It also advocates the use of other open source tools, notably JMeter, JUnitPerf, and Cactus.

Loughran [4] provides an invaluable resource for anyone wanting to use Ant in a real-world situation. The paper has a short section on deployment, but its true value is in the strength of its common-sense suggestions.

6. References

[1] Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley (2000) p.135

[2] Fowler, Martin and Foemmel, Matthew. Continuous Integration. http://www.thoughtworks.com/library/Continuous Integration.pdf

[3] Hightower, Richard and Lesiecki, Nicholas. *Java Tools for Extreme Programming*. Wiley Computer Publishing (2002)

[4] Loughran, Steve. Ant in Anger. http://jakarta.apache.org/ant/ant_in_anger.html

7. Appendix: Anatomy of a Deployment Solution

This section outlines the solutions developed for a specific project at Intelliware. The deployment of this system encountered a variation on every problem outlined in this paper.

The application consists of:

- a web application, including servlets and JSPs, distributed as a WAR.
- a web services application, including servlets, distributed as a WAR.
- a number of independent asynchronous processes, distributed as JAR files.
- support and common code, bundled as JAR files.

Supporting this application:

- a small database (approximately 40 tables).
- the various (transient) messages queues use by the asynchronous processes.

Other considerations:

- development occurs on Windows workstations. Each machine is able to run the entire system locally.
- the nightly build runs on a Linux box.
- integration testing is spread across the build machine and several other Linux servers.
- the production target is Linux, using a variable number of servers depending on budget, load, and failover requirements. The servers are on a subnet with limited visibility to the development network.

7.1 Evolution: the different types of deployment

The beginning. When the project started, the team knew that they'd have to handle multi-platform deployment in the future, even though they didn't have any hardware yet. They borrowed a single Linux box from another project and used it as their nightly build machine. That was the point in the project where the core Ant scripts were developed: scripts to define the database, define the JMS queues, and run the build.

In the early days, all the services ran on a single machine. The “deployment” process was pretty easy – the script would simply start the services as necessary on the Linux box using shell scripts.

Maturity. A full Linux build environment was set up. The deployment process only had to change a bit. The primary build machine would kick off the build process on the other servers. In this case, each machine would check the source out of CVS and compile it, stop and start its own web servers, and define the JMS queues on each machine. The database server was on an entirely separate machine by this time, and the database definition and initial data load was performed by a database client on the build machine.

The obvious drawbacks to this type of deployment – redundant compiles and testing, and the fact that each server had a complete code drop and resource build even though they only needed a subset of the functionality – were acceptable in the development environment. It was even desirable that this sort of thing was happening on the development servers, as it gave developers the chance to run the entire system on each server, making it possible to cordon off a server or two for isolated testing.

Production. Several known issues became unacceptable when the system moved to production. The team modified the build process into distinct parts – compile, unit test, jar, deploy/distribute, and integration test.

The deploy/distribute phase has three parts. First, on the build machine, the necessary files (JARs, WARs, and whatever else is needed) are copied into subdirectories. Each subdirectory represents one machine – one of the web servers, or the services machine, for example. Next, these directories are distributed to the remote machines by use of `rcp` (the Linux/Unix remote copy command). Finally, `rsh` (remote shell) is used to kick off a shell script to handle the deployment of files on each machine locally (e.g. putting the web apps in a web server directory). The local shell script is also responsible for starting the services on each machine.

Once this setup was in place for the build machines, the deploy/distribute script was adapted to take the JARs from the build machine and perform deployment to production, with a different machine specified for each function. There was an immediate problem – the production subnet was tightly secured, and the local network could only communicate with a gatekeeper machine on that subnet. Furthermore, `rsh` and `rcp` were deemed to be too insecure to use on the production machines.

The deploy process was further modified. Now all of the files are copied to the gatekeeper machine, and a script run on that machine is responsible for deploying the components to the other machines on the production network. As well, all file copying and process invocations are now handled by `scp` and `ssh`, which required a little more setup on each machine.

Today, three different kinds of deployment are done. The local developers run a “local build and deploy” on their workstations when they want to ensure that changes they made do not affect the system. The local build compiles the code, runs unit tests, deploys the code locally, starts the support services, and runs the integration tests locally. (In practice, the local build is only used when changes have been made that cannot be fully tested by the unit tests: these usually are composed of changes that are made to non-code resources (e.g., JSPs) or code changes that cross several subsystem boundaries.)

The second kind of deployment is the nightly build. The build machine checks the current source out of CVS, compiles it, and runs all the unit tests. If the unit tests run correctly, the code is jarred and distributed to the appropriate machines. Resources are built or defined on the machines as they need to be, the database is rebuilt, and services are stopped and started as necessary. Integration tests are run, and the build is deemed a success if all the integration tests run properly.

The final kind of deployment is used for moving the system to the testing, staging and production environments. These deployments are started manually. JARs from a known “good” build are deployed to the appropriate machines. Resources are redefined as necessary. Services are started automatically, and no integration tests are run. Since the target machines are not directly accessible from the local network, all of the deployment and services management is handled through the “gatekeeper” machine as described above.

The difficulty of merging existing database data with a new schema is still handled semi-manually – a pair of developers will examine the differences between the current schema and the new one, and use scripting tools to assist in creating the database migration script.

All three types of deployment are handled by one set of scripts that evolved over the lifetime of the project. The scripts are generally modularized by function instead of by platform or type of deploy; this allows us to keep all of the “start service” logic in one script, for example.

7.2 Things that aren’t being done well… yet

Monitoring system. The deployment process is not being monitored. Because some of the processes are kicked off asynchronously by remote shell commands, it is difficult to know when the deployment is complete. Similarly, there are no automated tests that are suitable for running against the deployed system – someone has to test the system manually after a deploy.

Windows deployment. There is no allowance made for a remote or distributed deployment on Windows. It is unlikely the system will ever be deployed to that environment, so this capability has not been a priority.