Continuous Innovation at BASIS
Building and Testing in the Cloud

BASIS engineers are responsible for completing a status report template each week. It usually isn't difficult for me to fill in the first four sections and describe what I've worked on or elaborate on problems I've encountered or summarize meetings I attended; that's all pretty standard stuff for a status report. But those last two items - *Interesting Article Review* and *Innovative Ideas* - confront me every week. They are a standing challenge, like a gauntlet tossed at my feet. They are a rebuke for my complacency and resistance to change, a nagging reminder that I need to spend time thinking about new technology and new ideas.

You see, BASIS is constantly on the lookout for better ways of doing things. More than any other company I've ever worked for, BASIS strives to keep abreast of the latest technology and continually improve the software development process. Some of these technical innovations are plainly visible in our products (BUI, anyone?), while others are implemented "behind the scenes" and make us a stronger, more successful company. This article tells the story of one such improvement, something that indirectly benefits everyone who uses BASIS technology.

It Wasn't Broke, but We Had to Fix It

The Business BASIC language, as you might imagine, is a large and complex piece of software. It has thousands of source code files and associated libraries. A few years ago it took an equally complicated system to build it. We appointed a machine as our dedicated build server, a pile of arcane build scripts written in various languages, and a designated "Build Master" engineer who spent all of his time keeping everything running. This arrangement got the job done and worked well for decades. It was clear to everyone, however, that it had several inherent disadvantages.

By Mike Phelps
Software Engineer

- SLOW - Getting all the source files from our source code repository, compiling them and assembling them into the final downloadable and installable package took many hours every day. During a normal day, we could get only two builds at most.
- SERIAL - Only one build could run at a time. Quite often we needed to publish daily builds of more than one BBj® version at a time, such as a development build (representing the latest work found in the "mainline" version of our source code repository) and a release build (representing a numbered release version of BBj). The build system could not perform simultaneous builds, guaranteeing delays during our most hectic periods.
- MANUAL - The build system was not completely automated. A human being always had to press the button to start a build, and if the build failed, a human being (the Build Master) had to comb through the log files to determine the cause. If, late in the afternoon, an engineer checked in a source file containing a syntax error, we wouldn't find out until the morning of the next day that it caused the nightly build to fail. The Build Master had to do some manual detective work to find out what file caused the build to fail and who checked that file into the source code repository so that person could be tasked with fixing the problem.
- FALLIBLE - There was always the worrying potential for catastrophic failure. If our Build Master ever happened to get hit by a bus, it would have taken a long time for someone else to learn how to operate and maintain the system. Likewise, had our dedicated build server ever failed, we would have spent several painful days trying to get another machine properly configured to take its place. We had no immediately available backup server or personnel.

Obviously our software build methodology was ripe for a little innovation. Nothing was broken, but we wanted (needed) to fix it anyway, so under *"Innovative Ideas"* I described how 'continuous integration' (CI) could make our build process better.

Learning to Continuously Integrate

The term continuous integration refers to a software development practice where teams of programmers frequently synchronize with and commit their changes to a code base repository, then rebuild the code base after every change is committed. This takes the pain out of getting code from many different people to compile successfully, and helps catch bugs by providing much quicker feedback about build results. Continuous integration is closely associated with other development techniques such as 'continuous testing' and 'continuous deployment,' where the code is constantly tested and constantly delivered to the end users.

These techniques all imply a high degree of automation, efficiency, and speed; things our build system conspicuously lacked. It all sounded wonderful, but we realized that adopting this technology was going to be a tall order. It was not possible to convert a long-standing, deeply-entrenched-in-the-corporate-culture build system overnight. There were several prerequisite steps that must be taken.

First of all, the code base must be stored in a version control system (VCS) such as CVS, Subversion, Git, Mercurial, or one of many others. CI servers, the software application that takes charge of doing software builds, need a convenient single location from which to get the code to build, and need to work with today's popular version control systems. BASIS has always used a VCS and currently uses Subversion to maintain the Business BASIC code base.

Next, the software build must be automated, meaning that the code base should compile with some type of script executed by a command from a shell prompt. In its simplest form, a CI server is a constantly running program that listens to signals from a VCS repository. When the VCS announces that a file has been added or deleted or modified, the CI server checks its list of build projects and starts a software build by invoking the command associated with that project. The CI server does not build the software itself; it runs a command that starts the automated script which does the build. When the build is finished, the CI server intervenes once more to store the results in an archive and notify any interested parties about success or failure.

Our existing build system was already partially automated; we had a collection of scripts that we could start from a command-line shell. We had developed these scripts over many years, using whatever scripting technology the build system architects were most comfortable with at the time. We decided to start over from scratch using Ant (Another Neat Tool), a well-known and extremely flexible build scripting system written in Java. This rewrite effort brought much needed simplicity and consistency to the build process, but it was by far the most time-consuming and difficult part of the conversion to CI. Ant allowed us to store the build scripts in the same directory packages as the code they were meant to build, or in other words, each time it checked the source code out from the repository, the Ant build scripts came with it automatically. Ant was easy to learn and understand, and best of all, Ant was able to compile the BBj code base in less than ten minutes!

With the prerequisites out of the way, we were ready to begin using a CI server for the first time. We chose CruiseControl, the "grandfather" of CI build servers. We set it up on a spare Linux server and turned it on to building BBj whenever anyone committed a change. These builds were not complete in that they did not produce the installable product delivered to customers; they were meant only to give quick build success/failure feedback to the engineers. The result was fewer failed builds on the main build system.

We later converted to a new CI build server called Hudson in order to take advantage of its beautiful web-based control system. Instead of arranging for each automated build with hard-to-decipher XML files, we configured the complete system and every build project by filling out GUI forms on a web page. After that, enthusiasm for the CI conversion really began to take off. More engineers got involved in running and maintaining the build system since it was easy to comprehend and fun to operate.

This proved so successful that we gradually added more tasks to the CI build system, like documentation builds. While fleshing-out and expanding the CI build system, we continued to rely on our original build system to produce the final product. Testing involved comparing the CI results with the builds from the legacy system. When the results were the same, we knew we had succeeded.

Ascending to the Cloud

BASIS moved to the cloud in 2010. Cloud computing refers to the increasingly common practice of purchasing computing power from a commercial services provider instead of maintaining an in-house collection of servers. (The word "cloud" in this term means the offsite collection of computer resources and data storage typically shown in diagrams as a cloud symbol.) We contracted with Amazon Web Services and began re-hosting our operations from on-premise servers to Amazon Elastic Compute Cloud (EC2) instances. We moved our Subversion source code repository to a persistent cloud server that summer and then transitioned the build system later in the fall.

Hudson worked extremely well in the cloud, thanks to a special Amazon EC2 plugin developed by Kohsuke Kawaguchi (the original creator of Hudson at Sun Microsystems). Although our ground-based Hudson installation could handle multiple simultaneous builds, the small number of machines in our onsite server lab that we could use as slaves still limited us. During busy times the Hudson server would get progressively slower, while requested builds would pile up in a queue waiting for a slave machine to become free. The Hudson CI server in the cloud, on the other hand, was able to start as many slave instances as required to handle the load. We could do an unlimited number of different simultaneous builds, and then scale back by shutting down the slave instances when their work was finished and the crunch was over.

In early 2012, when the original author of Hudson and most of the project's developers had a disagreement with Oracle (the corporate sponsor who took over from Sun Microsystems) and decided to fork the code base, we followed them to their new rebranded CI server, called Jenkins. The change was painless.

The use of a CI server and the move to the cloud caused a revolution in our testing and deployment process as well. Jenkins allows "chaining" build projects together in various combinations. This means that we could break up long,

multi-step processes into smaller individual build projects, then link them together so that any given build step has a 'parent' predecessor build project and a 'child' successor build project. The 'child' build projects do not run if their 'parent' builds fail. We wrote Ant scripts to initiate various kinds of tests and then linked them in Jenkins to the software build projects whose results we designed to test. Each time a software build successfully completes, another build starts which retrieves the compiled code from the CI server's archive, assembles it into an install package, and uploads it to an Amazon S3 (Simple Storage Service) 'bucket' in the cloud which anyone at BASIS can find. After the software build and packaging/upload projects complete, the CI server instantiates new slave instances in the cloud and on premises, which correspond to all the platforms supported by Business BASIC, loads each slave with the packaged build results, and calls the associated Ant scripts to initiate testing. The entire process is totally automated and runs without human intervention. (Well, almost. A human being triggers the process by checking in a change to the code base.)

Living in the Future

We are doing things now that were in the realm of science fiction a few years ago.

- The open-source Selenium testing framework integrated very well with our CI server, allowing us to automate testing our BBj GUI interface classes and each individual method belonging to them. This kind of testing previously demanded hours of tedium from a human being sitting in front of a screen with a mouse and keyboard, meaning that it was not done very often.

- At BASIS, we run the administrative and operations side of our company using software written in Business BASIC (we 'eat our own dog food'). We have a development environment in which we test new code and a production environment where we use stable code. In the days before automation and the cloud, getting new code from the development environment over to the production environment was a somewhat iffy manual operation. The two environments were not identical; what seemed to work in development might not work in production. Now, the issue is resolved with some help from the cloud. When we want to deploy the latest software version to the production server, we invoke an instance of a server that is identically configured to the production server, run tests to verify everything works, and then use rsync (the Linux remote file synchronization utility) to copy all the new or altered files to the production server. Quick, easy, and as foolproof as it gets.

- Running a TechCon conference used to involve very expensive, time-consuming configuration of in-house and rented computer equipment. All the issues and costs involved with physical transportation, installation, networking, testing, maintenance, and takedown were our responsibility. The cloud has made that old level of effort seem incredible, even ridiculous. We can now prepare Amazon Machine Images representing a fully configured computer running MS Windows or Linux with all the necessary software pre-installed, then invoke as many instances of them as we want, whenever we want, from the presenter's personal laptop while the demonstration is in progress. We can travel light, but still take with us all the power we need!

Summary

Our new CI cloud-based build, testing, and delivery system is fast, easy to operate, almost infinitely scalable, and (we believe) disaster-proof. The payoff has been remarkable. Is it perfect, or for that matter, is it even finished? I think not, it's a continuous process. We always have a list of improvements to make, inefficiencies to iron out, and new features to implement. With the cloud, there are cost savings available when we tweak this or cut back on that or take advantage of special offers. In this business, nothing remains unchanged for long...except for the weekly BASIS engineering status reports, which still end with the twin challenges - *Interesting Article Review* and *Innovative Ideas*. ■

- For more information on continuous integration, read
  - Continuous Integration - Fowler, Martin. (2006). *Martin Fowler*. Retrieved 6 November 2012 from bit.ly/1k01VP
  - The Cornerstone of a Great Shop – Jared Richardson Methods & Tools, Spring 2006 from bit.ly/eXtyT9
- Tools mentioned in this article include

• Ant	ant.apache.org
• Amazon AWS	aws.amazon.com/documentation
• Hudson	wiki.eclipse.org/Hudson-ci
• Jenkins	jenkins-ci.org
• rsync	rsync.net
• Selenium	seleniumhq.org