Will the Circle Be Unbroken? Continuous Integration and Visual FoxPro

Eric Selje
Salty Dog Solutions, LLC
Madison WI
Voice: 708-4NaClK9
Twitter: EricSelje
Email:Eric@SaltyDogLLC.com

This whitepaper discusses improving the Software Development Lifecycle for Visual FoxPro developers. We'll begin by reviewing the typical development cycle, and show ways to improve and automate our current process using a Continuous Integration server and open source tools from VFPX such as Automated Build, FoxUnit, and SubFox.

A Typical Day in the Life of a VFP Programmer

Eric (his name has been changed to protect his identity) is a typical Visual FoxPro developer. Eric got on board very early with the Fox. He learned very quickly in the late 80s that dBase was great but FoxBase was far better. Eric was very excited to when FoxPro came out, and when Visual FoxPro 3.0 arrived in 1995, his development careeer changed forever.

But in some ways Eric's Visual FoxPro career hadn't changed much since. New version of Visual FoxPro arrived regularly (up until recently), but the basic way he created applications was consistent throughout.

The Project Manager was always Visual FoxPro's command center for development. When Eric wanted to work on an application, he'd fire up Visual FoxPro, open the project in the Project Manager, and do what needed doing. Testing involved running the application and verifying it "did what it was supposed to do." Version control meant hopefully remembering to backup the application's source code directory. Deploying meant building an EXE and copying to someplace where the users could get at it, and likely meant kicking everybody out of the application so the update could be copied. It happened so often the users didn't think twice about it, assuming this was just the way it had to be. Eric too thought this was the way all programmers did it. Perhaps you do too?

It doesn't have to be this way.

Eric's slow realization of this truth came when he left the insular world of Visual FoxPro development and was exposed to what other programmers did. He started hearing words he didn't understand, like "Git" when there were no horses around, and "Mercurial" when there were no running backs. He started seeing applications that would automatically update themselves while the user was still running them, and "Nightly Builds" dropping onto servers at times when all the developers should long have been home in bed.

When Eric finally discovered the powerful concept of "Continuous Integration," his mind was blown. This was a concept that would once again change the way he developed forever – *if* he could get it all to work in Visual FoxPro.

A True Story

It’s the middle of the night. Your phone rings once, twice…on the third ring you pick it up. It’s your boss (or best client, whichever is applicable to you).

Groggily you say “Hello?” while hearing a sigh from your tossing spouse.”

“Hey!” you hear, causing you to pull the phone two inches away from your ear. “All of our nightly reports are crashing! Fix it!”

Now slightly awake, you see it’s 3 a.m. The reports just kicked in and immediately failed. You try to get it together enough to recollect what you may have done that day that would have caused this failure, and you’re actually hoping it was your fault because if it was another developer then you’re going to have to murder someone on your team and then have to go through the trouble of finding a replacement.

Eventually your synapses recall that one small optimization you made to the function that most of the reports use. Yes, that’s it. Easy fix…I’ll just put that change back. Since your company doesn’t reliably use version control, you’ll hopefully remember what the code looked like before and put it back right.

You fire up the laptop, make the change, build it, log onto your VPN, call the office, wait while everyone is out of the app, and copy the new EXE to the three or four different places it needs to go for everything to run smoothly until you get to your desk in… only 3 hours?!

That “one little fix” took very little time to actually fix but quite a bit of time to build and deploy. You *hope* it works, but you’ll find out if it failed when the phone rings again just as you’re drifting back to sleep.

I imagine that story makes you cringe. Just writing it started stirrings of PTSD inside of me. Now let’s investigate how Visual FoxPro developers like Eric can use Continuous Integration so that none of us will ever have to suffer through an experience like that again.

What is Continuous Integration?

At the most basic level, Continuous Integration (CI) means letting a tool, a Continuous Integration Server, automate the day-to-day tasks that you have been doing manually. That means automatically running the tests to ensure your code is good, building the EXE from the project’s source code, and copying the EXE to a place where others can get at it.

Some Visual FoxPro developers have concocted a collection of scripts and batch files to do much of this mundanity, but a true CI Server is a comprehensive, integrated tool with hooks you can tap into in order to consolidate those disparate scripts under one management tool. A CI server gives you the extra functionality of creating status reports that can be emailed and web dashboards that allow you to quickly ascertain the status of all of your projects at a glance.

How much would you pay for this amazing Continuous Integration awesomeness? Well, you *could* pay a lot as there are commercial products available. On the other hand you could not pay anything at all. Let's look at some of our options.

Continuous Integration Servers

There are quite a few CI servers available[1], but here are some that are probably most likely to be used by Visual FoxPro developers:

Commercial Offerings

Team Foundation Server (TFS): This is Microsoft's official offering. It's much more than just a CI server, it is also the successor to Visual SourceSafe for version control, has a facility for project task list tracking, and reporting. This is commonly used in Microsoft-only shops and, in my experience, rarely used elsewhere.

TeamCity: This program from JetBrains, the Resharper folks, is the most common name I hear when I ask which Continuous Integration product other developers are using. It is built to be language-agnostic so it works well with .Net, Java, Ruby, etc., and with plug-ins can be modified to work with Visual FoxPro as well. While I've listed here as Commercial, it's actually free for "small teams."

Free and Open Source

CruiseControl: This open-source framework was the first one I used because Markus Winhard had already written a configuration file for Automated Build. The original CruiseControl was written in Java but was ported over to the .Net framework in a fork called CruiseControl.Net.

Jenkins: Jenkins is my current open-source favorite. It's very simple to install, has a very small footprint on my development machine, and is infinitely configurable. Jenkins is the tool I'll be using in my session and for this whitepaper.

This is not intended to be a comprehensive list. Please do some research before deciding which CI server is right for you.

When you've chosen and installed your Continuous Integration server and integrated all the plug-ins you need for your project, you will have a complete system that allows you to

[1] http://en.wikipedia.org/wiki/Continuous_integration

deploy software in a repeatable, efficient manner that decreases errors and increases your productivity.

This diagram shows the continuous integration cycle. The cycle begins when the developers at the top check in their source control to the central repository. That causes the CI Server to run Unit Tests, Integration Tests, Functional Tests, Code Analysis, and whatever else you've plugged in. If the tests fail, the check-in will be rolled back, but if they succeed the project will be compiled into an EXE. If the build passes (no compilation errors), it can proceed to deploying the final product to wherever it eventually goes. All along the way reports can be generated and sent to the project team to apprise them of the status of the project.

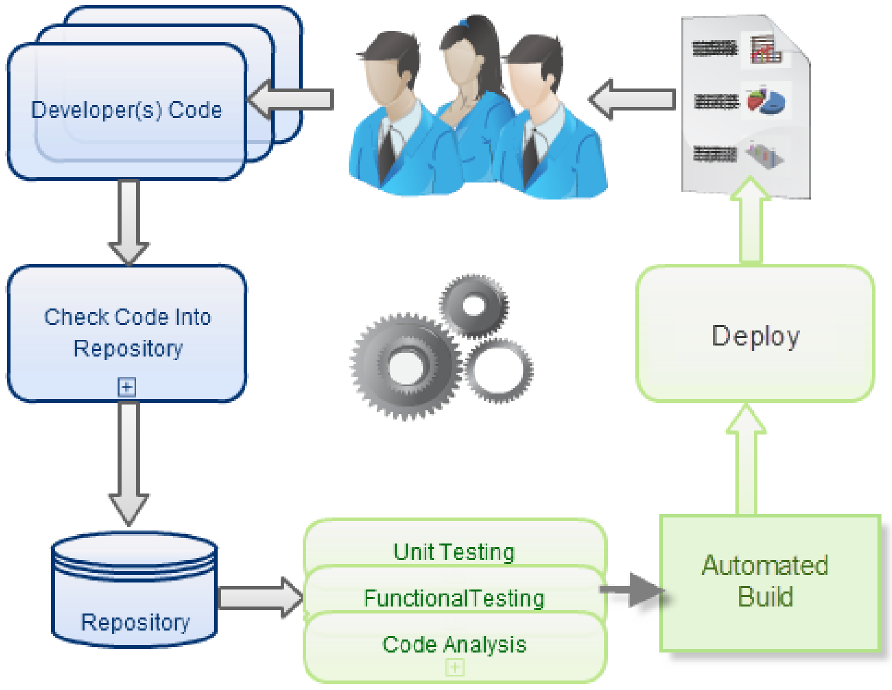


Figure 1: The Continuous Integration Cycle

Ideally the tests pass, the build is created, the executable is deployed, the client is updated, the developers are notified, and the cycle starts again. The question then becomes, to borrow a title from a classic, "Will the Circle Be Unbroken?"

Setting up a Continuous Integration Server

To start, let's set up a CI Server. This can be on your development machine (which is the way I do it), a Windows server on your network, or a virtual machine in "the cloud." It really doesn't matter where it's physically located, but if it's on your development machine you are limiting the scale to just one developer. If you think there's a chance you'll ever involve more than one developer, or you feel more comfortable putting your eggs in different baskets, consider putting the CI Server somewhere else.

You might imagine that a CI service needs to be running on a powerful server somewhere but this definitely not the case. It can run on any machine that has access to the source control as well as any additional tools required to do its job such as the Build tool. It does have to be a Windows machine though, because Visual FoxPro projects are Windows-only and require a Windows machine to build them.

Step 1: Use Source Control

Version control, while not *absolutely* necessary, is preferred because CI servers have the ability to monitor repositories of the most popular types, and usually a source code check-in is the trigger that fires the entire CI process. Hopefully by now you've begun taking advantage of version control! Even if you don't work with other developers, you need to be doing this.

If you don't have an offline repository yet, just search the web for any number of free/cheap ones. In this figure I setup a free repo on **ProjectLocker** and joined my test project to it in less than five minutes.

Serial Killer

One quirk of Visual FoxPro is that much of the source code is stored in binary files (actually VFP tables themselves) rather than text files. Binary files are problematic for version control systems because it's rather difficult to "diff" them. The solution for VFP developers then as always been to *serialize* the table, i.e. turn it into an equivalent ASCII text file.

Projects that have been "Joined to Source Control" using Visual FoxPro's built-in functionality automatically get their binaries serialized via the native SCCTEXT.prg. This is an inferior program and has long been superseded by any number of projects, including VFPX's **SubFox** and **SCCTEXTX**, Christoph Wollenhaupt's **TwoFox**, or Toni Feltman's **F1Utils**.

If you prefer to do version control from the Windows filesystem, you can use the fine **Tortoise** tools for Subversion or Mercurial. Tortoise integrates right into Windows Explorer. Tortoise doesn't serialize the binaries for you automatically though, so you will need to be sure to take care of that manually using one of the tools mentioned above. SubFox does have some nice hooks into Tortoise that automates the process.

The problem with all of these tools however is that invoking the serializer/deserializer is a *manual* process[2], and with Continuous Integration we need things to run *automatically*. Whichever of those tools you use, some customization will be necessary in order to get them to run from the Windows shell in order to be integrated into Continuous Integration.

For this reason, I recommend that you do check in the binaries into your version control system along with the serialized version. That way they won't need to be deserialized when you run an automated build, saving you this manual step.

The exception to this rule is the PJX/PJT files: Do not check those into version control. Automated Build, in the next step, will recreate the PJX file from the "PJM" file that's generated. Just be sure to check in your PJM file!

Step 2: Install Automated Build

This is the utility that takes your source control and creates an EXE out of it. Visual FoxPro developers have always had to do this manually by clicking on the Build button in the Project Manager. Now we also have **Automated Build,** another VFPX tool, to compile EXEs from the command line.

Automated Build was written by Markus Winhard. It's actually just one PRG, BuildProject.prg along with a few supporting files. See the full documentation on VFPX, but the installation is basically just copying the files needed to a folder.

Tip: You might be tempted to create an EXE so you run Automated Build without needing a full VFP deployment; However Automated Build uses the command BUILD PROJECT, which is unsupported in the VFP Runtime. You'll have to copy the minimum files needed to run the full Visual FoxPro onto your CI server. This means you'll need a fully licensed copy of VFP for your CI server if it's a separate machine than your development machine.

If you use CruiseControl.Net as your CI server, Automated Build has sample configuration plug-in files that make it very easy to inject AutomatedBuild into your CI Cycle. For Jenkins we'll have to configure things ourselves as well, which we will do later when we set up a real job.

GenPJM

BuildProject takes as a parameter the name of a PJM file, which is the serialized version of the project's PJX/PJT file. If your project is joined to a source control provider, the PJM is created automatically for you. If not, you will need to generate that PJM manually. A recent addition to the Automated Build project on VFPX is GenPJM.prg, which will serialize your project's binary into a PJM file that Automated Build can use.

[2] SubFox's hooks do deserialize binaries automatically when you check out files from the Tortoise interface, so it may have potential to from a command line, but this needs further investigation into the source code. The other tools could have wrappers written that allow them to do it as well.

Tip: Call GenPJM from a ProjectHook whenever the project is modified, saved, or built so you don't have to remember to do it manually. Remember, one goal of CI is to automate as much as possible!

BuildProject recreates the PJX file from the PJM, compiles the source code, and creates output that the Continuous Integration server can parse and act on.

Are we done?

You *could* stop right there and just have a CI server that automates builds periodically or whenever you check in source control. That alone would probably save you a lot of time and effort. Let's not stop there - let's add a couple more things to really create a professional development environment.

Step 3: Set up Unit Testing Framework

(Optional, but Desirable)

Unit testing is often the first tool fired in the CI cycle, occurring right after new source code is checked into the repository. A good CI server can be configured to proceed to a build only after the source code passes the unit tests, or rollback a check-in that causes unit tests to fail and send the developer that tried to check in the failing code a detailed report of the results.

Visual FoxPro developers have the excellent **FoxUnit** framework to allow them to interactively create and run unit tests while they're developing, but that doesn't help us in an automated environment. Now that FoxUnit has also moved to VFPX, an effort is underway to expand its functionality to allow tests to be run from the command line so they can be integrated into a Continuous Integration tool.

John Miller wrote an excellent article for CoDE Magazine that details the work you need to do in order to make FoxUnit work with Team Foundation Server. Only slight modification is needed to customize it to the Continuous Integration framework of our choice, Jenkins.

And More

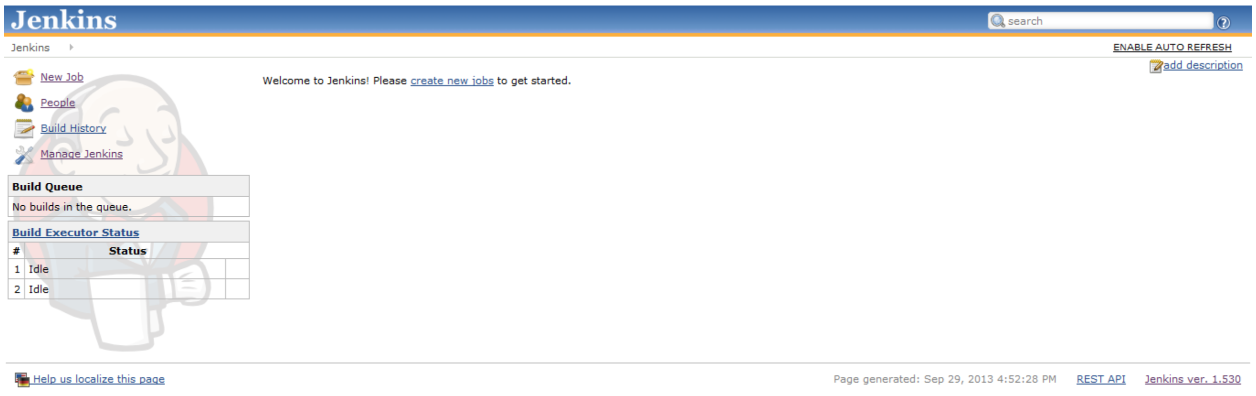
Continuous Integration Servers have "hooks" that allow you to plug-in any number of 3rd party tools. Properly modified, it would be possible to integrate tools into your software development lifecycle that check for "code smells", plug into your issue tracking system or project management, deploy the software to FTP servers or Content Delivery Networks, and so much more.

Final Step: Set up a Continuous Integration Framework

Finally we have arrived at the most necessary step of all in order to implement Continuous Integration, setting up the CI Server itself. I'm going to use *Jenkins*, a popular open-source Continuous Integration framework. You can read all about the interesting history of Jenkins here. Installing Jenkins on Windows is trivial, just download the .ZIP file and run the SETUP.EXE on the machine that will serve as your Continuous Integration server.

After installing Jenkins, you'll have a new service running on that machine which is setup to run automatically when the machine starts. This is likely what you want but you can flip it to start manually if you'd rather.

The service monitors a port on your machine, which by default is 8080. Fire up a browser and navigate to the servername:8080 and you'll see the basic Jenkins dashboard (**Figure 2**).



Welcome to Jenkins! Please create new jobs to get started.

Build Queue
No builds in the queue.

Build Executor Status

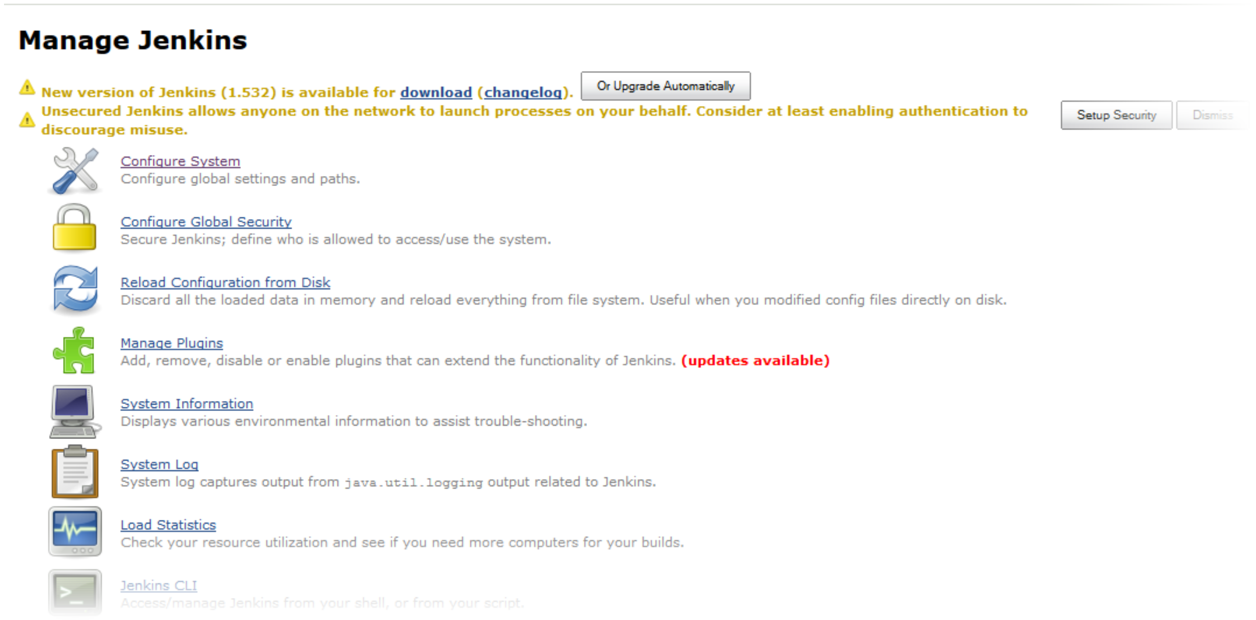
#	Status
1	Idle
2	Idle

Help us localize this page

Page generated: Sep 29, 2013 4:52:28 PM REST API Jenkins ver. 1.530

Figure 2: Jenkins Dashboard

Start by configuring Jenkins to your liking by clicking the "Manage Jenkins" link to get the configuration page (**Figure 3**). Notice how Jenkins can check back with its home server and lets you know when it is out of date. You can update with the click of a button. Nice.



Manage Jenkins

New version of Jenkins (1.532) is available for download (changelog). Or Upgrade Automatically

Unsecured Jenkins allows anyone on the network to launch processes on your behalf. Consider at least enabling authentication to discourage misuse. Setup Security Dismiss

Configure System
Configure global settings and paths.

Configure Global Security
Secure Jenkins; define who is allowed to access/use the system.

Reload Configuration from Disk
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly on disk.

Manage Plugins
Add, remove, disable or enable plugins that can extend the functionality of Jenkins. (updates available)

System Information
Displays various environmental information to assist trouble-shooting.

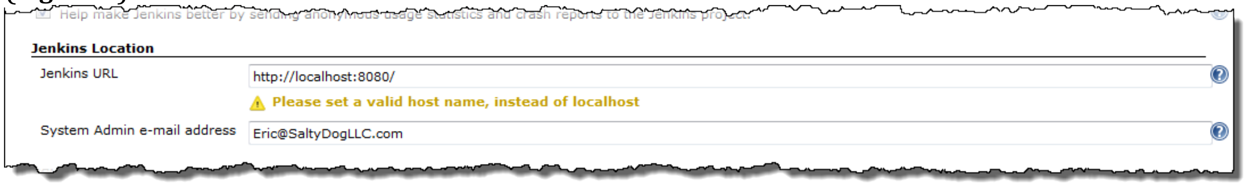
System Log
System log captures output from `java.util.logging` output related to Jenkins.

Load Statistics
Check your resource utilization and see if you need more computers for your builds.

Jenkins CLI
Access/manage Jenkins from your shell, or from your script.

Figure 3: Jenkins Management Screen

On the “Configure System” screen, you’ll want to verify/change which port you’re using for Jenkins (**Figure 4**), as well as set up the email settings in order to receive notifications (**Figure 5**).

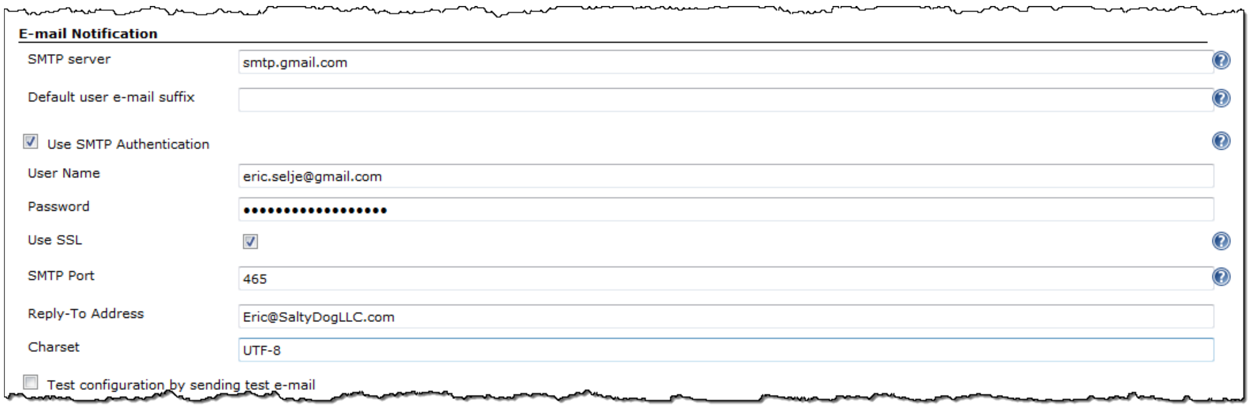


Help make Jenkins better by sending anonymous usage statistics and crash reports to the Jenkins project.

Jenkins Location

Jenkins URL	http://localhost:8080/
	⚠ Please set a valid host name, instead of localhost
System Admin e-mail address	Eric@SaltyDogLLC.com

Figure 4: Configure Jenkins Port Number



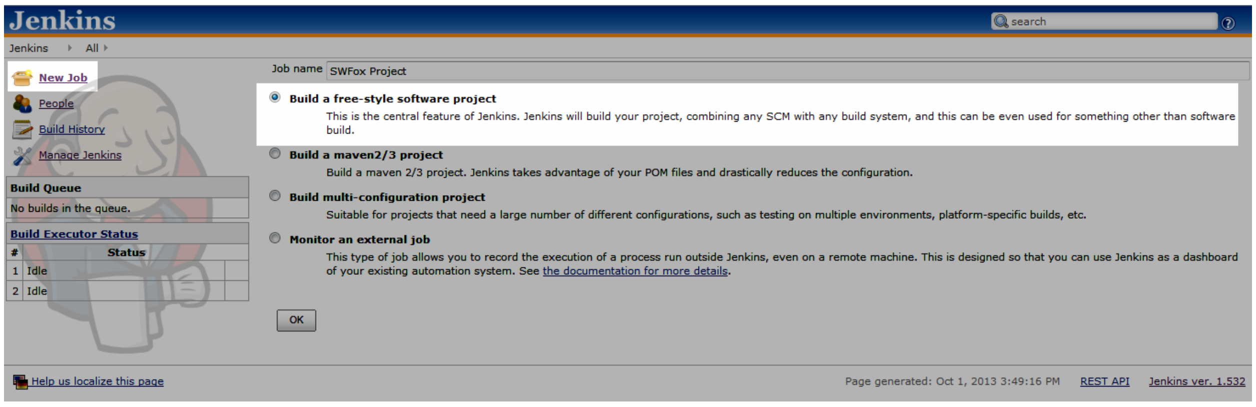
E-mail Notification

SMTP server	smtp.gmail.com
Default user e-mail suffix	
✓ Use SMTP Authentication	
User Name	eric.selje@gmail.com
Password	••••••••••••••••••
Use SSL	✓
SMTP Port	465
Reply-To Address	Eric@SaltyDogLLC.com
Charset	UTF-8
☐ Test configuration by sending test e-mail	

Figure 5: Configure Jenkins to send general EMail Notifications

Creating a Job in Jenkins

After Jenkins is configured we can proceed to create our first “job”. Jobs are somewhat analogous to “Scheduled Tasks” in Windows (or cron jobs in *nix), in that they’ll run on a schedule or when a specific event happens.



Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Job name: SWFox Project

(●) **Build a free-style software project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

( ) **Build a maven2/3 project**
Build a maven 2/3 project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

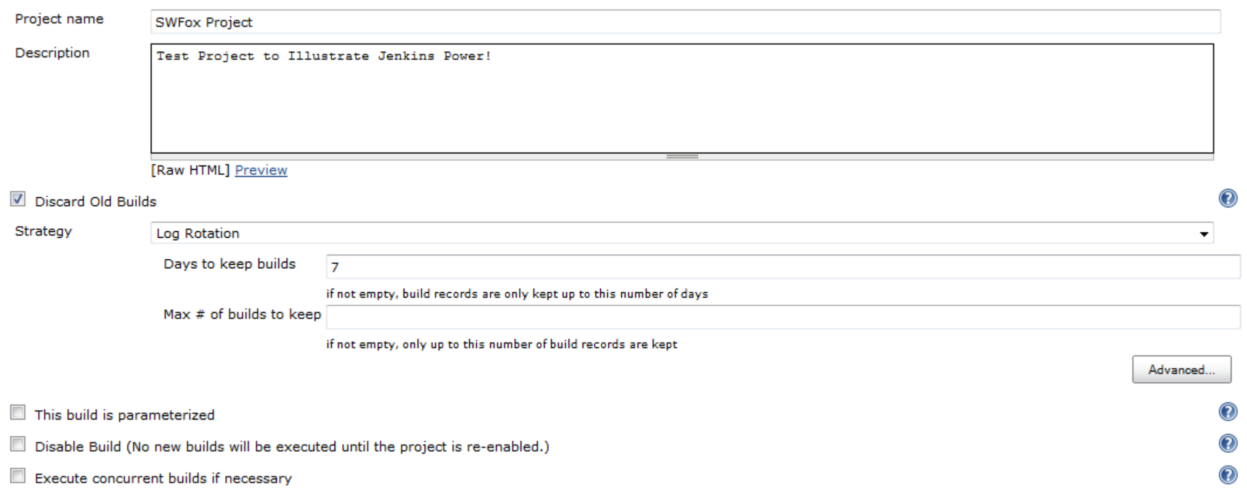
( ) **Build multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

( ) **Monitor an external job**
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See the documentation for more details.

Page generated: Oct 1, 2013 3:49:16 PM REST API Jenkins ver. 1.532

Figure 6: Creating a new "Job" in Jenkins

For our job we’ll create a “free-style” project, giving us the most flexibility since we use a non-standard setup. First we name and describe our project, and perhaps give it a few other options like how many old builds to keep around (**Figure** 6: Name and Describe the Job).



Project name: SWFox Project

Description: Test Project to Illustrate Jenkins Power!

[Raw HTML] Preview

☑ Discard Old Builds

Strategy: Log Rotation

Days to keep builds: 7

if not empty, build records are only kept up to this number of days

Max # of builds to keep:

if not empty, only up to this number of build records are kept

☐ This build is parameterized

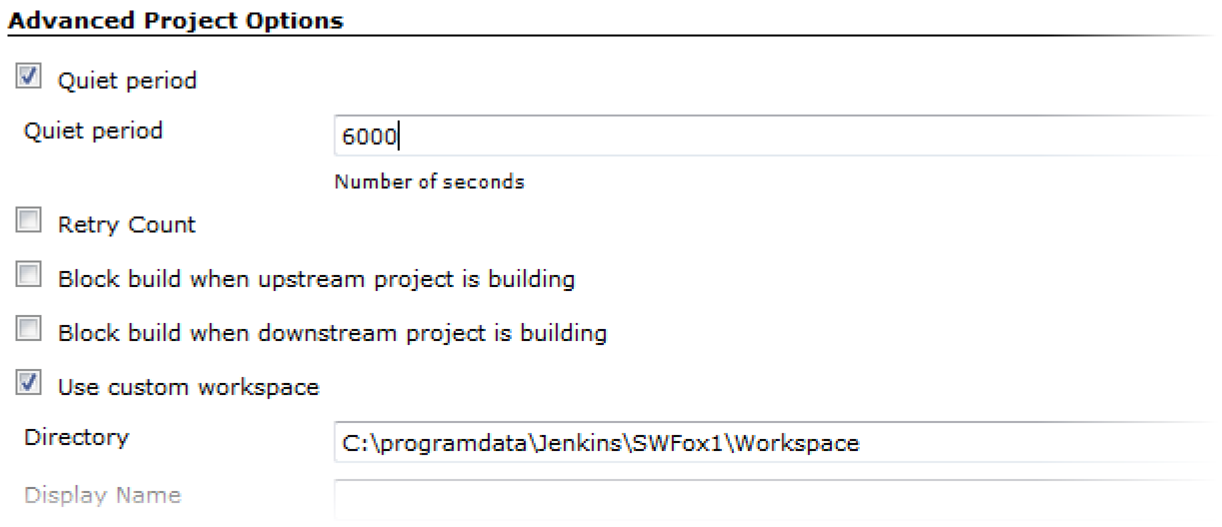
☐ Disable Build (No new builds will be executed until the project is re-enabled.)

☐ Execute concurrent builds if necessary

Figure 7: Name and Describe the Job

Next we can set even more options (**Figure** 7: Advanced job options), like a "Quiet Period", which sets an interval between each build. This is useful when you have quite a few developers submitting code a lot, to prevent the build from constantly firing every time a new submission comes in. In Figure 7 I've throttled Jenkins to at most do one build every 10 minutes (6,000 seconds).

By default, the "workspace" for a build is in a folder underneath the default Jenkins install folder (e.g. C:\Program Files (x86)\Jenkins\jobs\<project name>\workspace). You might want it to be somewhere else though, especially on Windows machines so you don't succumb to the virtualization issues that come with writing files under C:\Program Files. I prefer creating my workspaces under C:\ProgramData.



Advanced Project Options

☑ Quiet period

Quiet period: 6000

Number of seconds

☐ Retry Count

☐ Block build when upstream project is building

☐ Block build when downstream project is building

☑ Use custom workspace

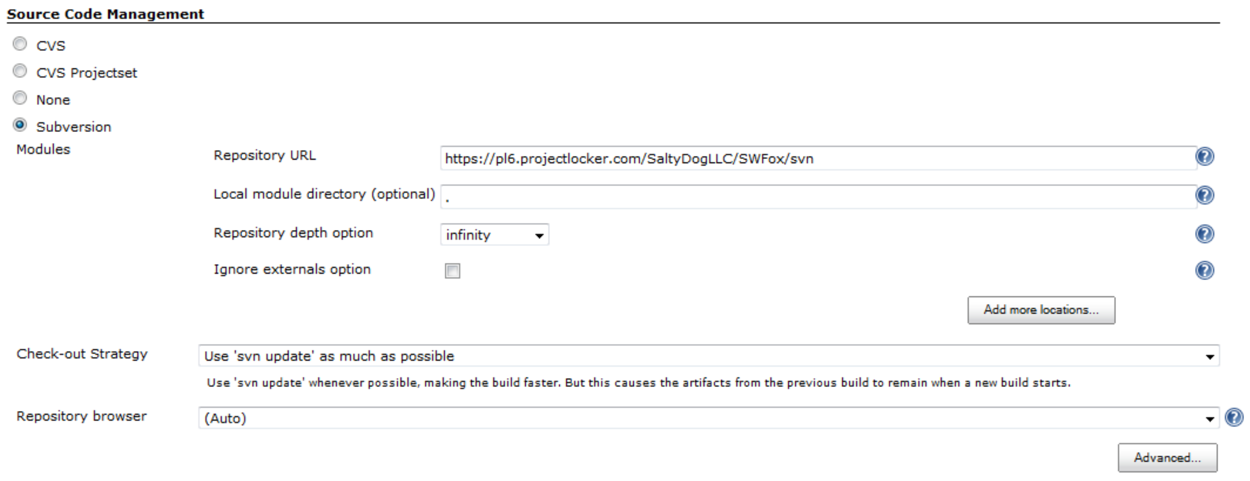
Directory: C:\programdata\Jenkins\SWFox1\Workspace

Display Name:

Figure 8: Advanced Job options

Configuring Your Source Control

The next step is to tell Jenkins where your source code is checked in (**Figure** 8). Out of the box (I guess that expression doesn't really apply these days) Jenkins supports CVS and Subversion, but there are plug-ins for Mercurial, Git, and probably just about any other version control system out there.


Source Code Management

- CVS
- CVS Projectset
- None
- [✓ Subversion]

Modules

Repository URL: https://pl6.projectlocker.com/SaltyDogLLC/SWFox/svn

Local module directory (optional): .

Repository depth option: infinity

Ignore externals option: ☐

Check-out Strategy: Use 'svn update' as much as possible

Use 'svn update' whenever possible, making the build faster. But this causes the artifacts from the previous build to remain when a new build starts.

Repository browser: (Auto)

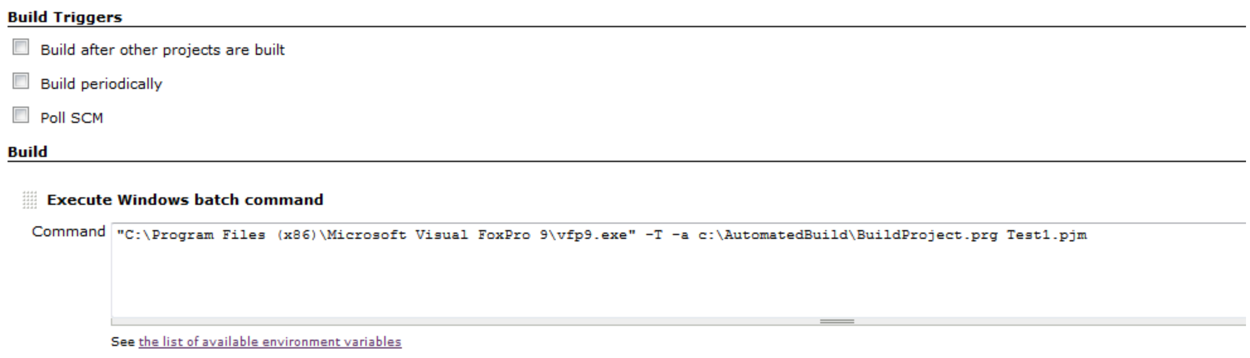
Figure 9: Source Code Management

Note again that linking your CI Server to your source control isn't absolutely necessary. If you choose not to link them, you will lose the ability to do a build automatically when source code is checked in. That may not be an issue for you if you'd rather just do a build on a regular schedule or create a build manually. If that's the case just select "None" for source control and you can set your development folder as your Workspace. You will lose a lot of scalability doing things this way, but it might be all you need.

If you checked your binaries (VCX, SCX, FRX, etc) into source control as I suggested, you now have an extra step: You have to deserialize the text files back into binaries before you can do the build because VFP unfortunately can't build based on the serialized version (which is unfortunate really, because one would think that internally it's *reserializing* the binaries before it does its compilation anyway). There are a variety of ways to do that depending on how they were serialized but most would involve writing a wrapper around whichever technique you used to serialize in the first place. *There is a lot of opportunity for some VFPX volunteer work to be done here.*

Set Up the Build

The most important step is to actually build the EXE (or DLL) from our project. That means telling Jenkins when to build and how to do it. Using the Build Triggers (**Figure 10**), you can instruct Jenkins to poll the Source Control repository if you set one up in the previous step, build periodically using a very flexible syntax, or build after other Jenkins projects get successfully built, which is great for projects that are dependent on other projects.


Build Triggers
Build after other projects are built
Build periodically
Poll SCM
Build
Execute Windows batch command
Command
See the list of available environment variables

```
"C:\Program Files (x86)\Microsoft Visual FoxPro 9\vfp9.exe" -T -a c:\AutomatedBuild\BuildProject.prg Test1.pjm
```

Figure 10: Scheduling an Automated Build

You don't have to schedule the build at all. Anytime you want to build you can fire up the Jenkins dashboard and click on the "Schedule Build" icon. That will queue up a build to begin immediately.

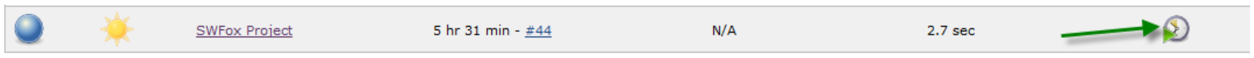


Figure 11: Build manually anytime by queueing up a build with this button

Tip: Hover over that icon and look at the address bar on the bottom of the browser. Because Jenkins is web-based and has a well-written API, you can use that same URL to invoke a build *(e.g. http://(CI Server):8080/job/(Project)/build?delay=0sec)*. Just add a button to your VFP Developer Toolbar (which is very easy if you use Thor) that calls that URL (simple with a RUN /N CURL.exe or either WestWind IP Stuff or VFP2C32) and you can have a build fire at the push of a button.

The last step in scheduling a build is telling Jenkins how to do the build (again, **Figure 10**). Here we're using the full version of VFP (because we need to use the BUILD PROJECT command) to call automated build (BuildProject.prg). The syntax for BuildProject is

```
BuildProject.prg PJMFile [ManifestType, EXE Name]
```

PJMFile is the name of the serialized PJX file created with either GenPJM or our source control tool if you've joined your project to source control.

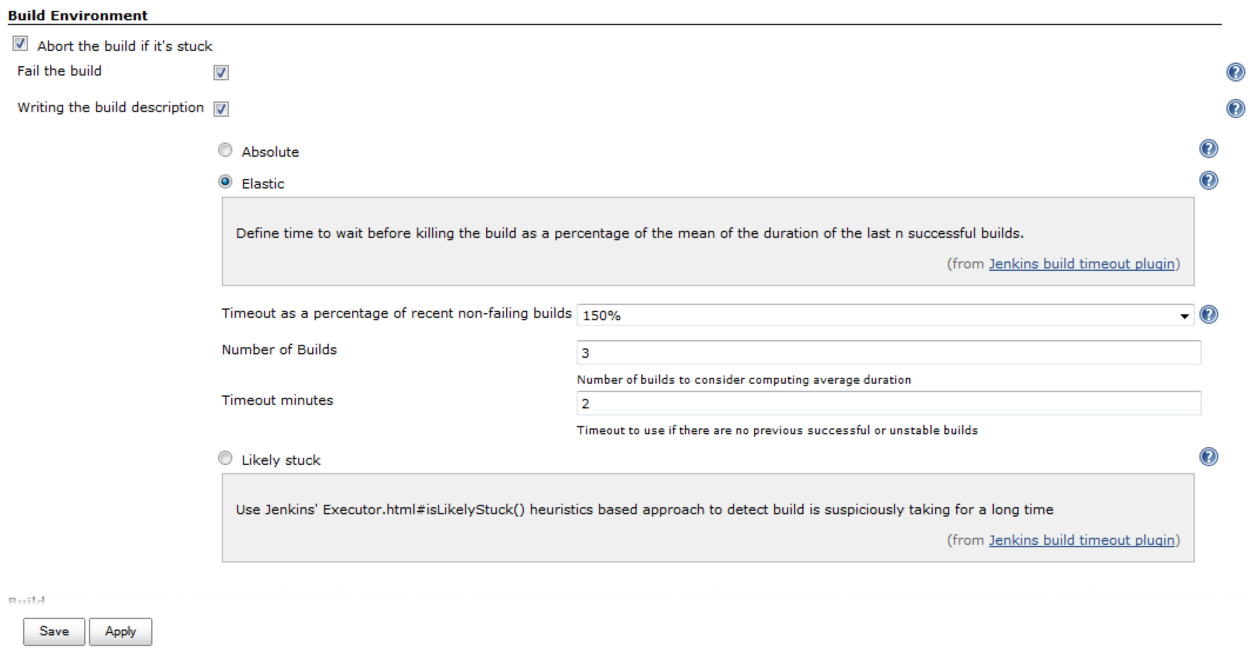
ManifestType is either ADMIN or USER to indicate the execution level (default is USER).

Once you've got your Build action described, go ahead and test it by clicking on the icon to queue up a build immediately (**Figure 11**). You should see the "Last Success" time update automatically and a pretty sun icon to indicate all is well. Clouds would indicate things aren't going so well. Clearly the developers of Jenkins aren't goth.

Failed Builds

If an Automated Build fails, perhaps because of a syntax error or a missing file or anything else that can cause a build to fail, it currently just sits there and doesn't exit properly. Markus is aware of this problem and we've spent some time trying to figure out quite why it's happening, but it turns out there's a nice workaround in Jenkins for this so it doesn't break things at all.

If you go back to the root of the dashboard and choose “Manage Jenkins” and then “Manage Plugins” (or type http://<CIServer>:8080/pluginManager/? into your browser’s address bar) you can select from over 300 plug-ins currently available for Jenkins. One of them is called the “Jenkins Build Timeout Plugin” and it’s just what we need. Install that plugin to add a new option on the Job Configuration page called Build Environment (**Figure 12**).



Build Environment

[✓] Abort the build if it's stuck

Fail the build [✓]

Writing the build description [✓]

○ Absolute

● Elastic

Define time to wait before killing the build as a percentage of the mean of the duration of the last n successful builds.

(from Jenkins build timeout plugin)

Timeout as a percentage of recent non-failing builds	150%
Number of Builds	3
	Number of builds to consider computing average duration
Timeout minutes	2
	Timeout to use if there are no previous successful or unstable builds

○ Likely stuck

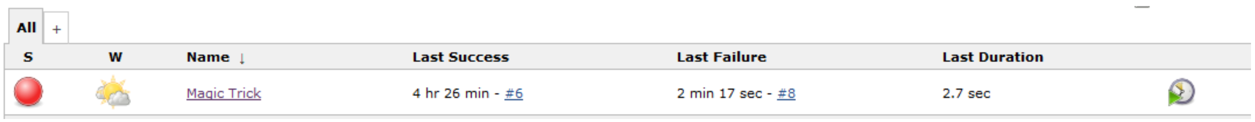
Use Jenkins' Executor.html#isLikelyStuck() heuristics based approach to detect build is suspiciously taking for a long time

(from Jenkins build timeout plugin)

Figure 12: New options available after the Jenkins Build Timout Plugin is installed

The plugin is nicely configurable, allowing us to specify an absolute time to abort the build, an elastic time based on how long previous builds took, or a “likely stuck” time based on some algorithms built into the plugin. Here I’ve chosen to say “Abort the build if it takes more than 150% of the time that the last 3 builds took; if there haven’t been 3 builds yet then abort after 2 minutes.”

Now if I introduce an error into my code and try to build, I’ll get a cloudy icon indicating all is not well, but Jenkins will not hang. You can find the .ERR file in the workspace folder to see what’s causing the problem.



S	W	Name ↓	Last Success	Last Failure	Last Duration
●		Magic Trick	4 hr 26 min - #6	2 min 17 sec - #8	2.7 sec

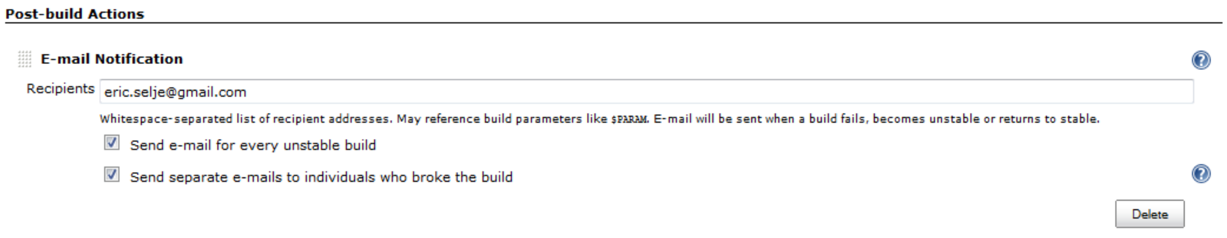
Figure 13: A Failed Build

Adding Build Actions

Now that your build is scheduled and executing properly, you can add more build actions. The new actions can be sequenced either before or after the actual build by dragging the actions around the screen. There are over 300+ plugins available currently for Jenkins, but very few are applicable to Visual FoxPro. Though we are a bit limited as compared to some of the other development tools right now, though perhaps we can use VFPX to get our tools up to snuff. Here are some ideas for build actions:

E-Mail Notification

Jenkins includes functionality to send emails when a build fails (**Figure 14**). This is useful when you've scheduled your builds to run automatically and aren't keeping your eyes on the dashboard.



Post-build Actions

E-mail Notification

Recipients: eric.selje@gmail.com

Whitespace-separated list of recipient addresses. May reference build parameters like $PARAM. E-mail will be sent when a build fails, becomes unstable or returns to stable.

- [✓] Send e-mail for every unstable build
- [✓] Send separate e-mails to individuals who broke the build

Figure 14: Send email when a build fails

Unit Testing

Ideally you'll want to run your unit tests before you actually build, and abort the build process if any of the unit tests fail. But FoxUnit, the de facto unit testing framework for Visual FoxPro, was not created with Continuous Integration in mind. It's not callable from the command line to execute unit tests, and it cannot create the types of "console applications" necessary to write to STDOUT, so we're in a bit of a quandary.

Fortunately John Miller has started us down a road that will take care of this problem. Check out the code in the article referenced above. I haven't quite gotten it working perfectly yet myself and so I can't write as much as I'd hoped to on this topic, but I would welcome any contributions to the FoxUnit project in this direction.

Code Analyst

This VFPX tool runs a "sniff test" on the code to ensure it conforms to standards and is well-written, based on built-in rules and rules you create. Current rules included with the project check your code for problems such as:

- Are there any RETURN statements within a WITH /ENDWITH block?
- Are you using CTOD() anywhere? (This doesn't internationalize well)
- Does each function have a RETURN?
- And much more, plus you can write your own rules.

Like FoxUnit, Code Analyst will need a wrapper in order to make it usable from a CI Server, but we have the talent in our community to do this.

Deployment Tool

This is the tool that copies your EXE to wherever it needs to go. It may be as simple as copying it onto a shared folder or as complex as creating a variety of Setup executables for different platforms, uploading them to a secure FTP site, and updating a web page with links that points to the latest versions.

Continuous Integration servers usually come with functionality to deploy your EXE in the most common ways, but you may have unique needs that require you to use something special.

For example you might run an InnoSetup script that bundles your files into a SETUP.EXE with auto-updating ability. That would be nice and fairly easy to integrate into CI, but it's beyond the scope of this whitepaper. See Rick Borup's whitepaper from a few years ago on using InnoSetup with Visual FoxPro, and then just add the Inno script as a Build Step after the project is built to create the SETUP file.

Once you've include a deployment step into your CI cycle you've gone beyond Continuous Integration into the realm of Continuous Deployment. This is a logical step to automate and it creates some more possibilities and issues, such as where to deploy nightly builds vs. stable builds.

And More

Other development environments may include other checks to ensure code integrity before a project is deployed – tools for which no native VFP solution exists. That means there's room for creative folks like you construct the tools, or customize an existing tool to do the job for us. These tools might include:

- **Integration Testing**: These are tests that make sure that your code works well with others, like the database, the web services, the mail server, etc. I'm not aware of any frameworks that are made specifically for FoxPro but many are adaptable. It would also be acceptable, though perhaps not "pure," to write FoxUnit tests that are actually not strictly unit tests but also integration tests.

- **Functional Testing**: These are tests for the User Interface of your system, rather than the business logic which is what Unit Testing is aimed at.

- **Load Testing**: Tests how your application would fare when many users are hitting it at once. These sorts of tests spin up multiple instances of your app, often on multiple machines, to simulate real-world conditions. It's amazing how some apps that work fine when the developer is testing things don't perform as well under load.

- **Coverage Testing**: These tests simulate paths through your code to see which code is getting nailed and which code may not be touched at all. They perform something similar to what would happen if you SET COVERAGE TO a file in the beginning of your application, ran it all day through every code path possible, and then ran the coverage analyzer afterwards.

- **So Much More**: Take a look at the plugins page of Jenkins to see the plethora of options available to you. Want to Tweet when your build succeeds? Fire a growl notification? Make an entry in a database? Run a StyleCop on your HTML code? There's so much more you can do with an extensible CI framework like Jenkins.

Conclusion

Attempting to integrate your Visual FoxPro solutions into a Continuous Integration process exposes the limitations of Visual FoxPro: the inability to create console apps, the binary source code, and the lack of a native deployment tool create difficulties for the developer. However given some patience and a few workarounds, we can take advantage of Continuous Integration.

I've attempted to use as many VFPX projects in this session as possible, exposing opportunities for improving the utilities. I implore you, dear reader, to get involved. Throughout this whitepaper I've indicated several places where VFP could use a little more help to make Continuous Integration useful and easy for every programmer.

I hope the concept of Continuous Integration, and the specifics of how to implement it into your Visual FoxPro Software Development Lifecycle, are much clearer to you for having read this whitepaper and/or attended my Southwest Fox session. This is a necessary process for serious developers to integrate into toolbox, because like any good tool it will make you more efficient and reduce errors.

Happy coding, and may your continuous integration cycle forever be unbroken.

Dedication

More details about the individual VFPX tools used in this whitepaper are available in the recently published book *VFPX: Open Source Treasure for the VFP Developer*. Many thanks to the authors of those tools for creating the parts we needed to put together this entire system.

John Miller was way ahead of his time, writing about pretty much this very topic way back in 2007. I came across his article when I was writing about Automated Build for the book mentioned above, but ironically had already forgotten about it when I was writing this. It was while rereading my own chapter in the VFPX book that I stumbled across my reference to his earlier article. This whitepaper restates, probably less elegantly, most of the same points he made way back then. For more information about this topic I encourage you to read his article.