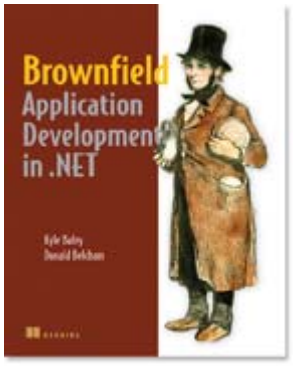
MANNING PUBLICATIONS

Build Artifacts

Excerpted from



Brownfield Application Development in .NET

EARLY ACCESS EDITION

Kyle Baley and Donald Belcham
MEAP Release: March 2008
Softbound print: June 2009 (est.) | 550 pages
ISBN: 1933988711

This article is taken from the book Brownfield Application Development in .NET from Manning Publications. As part of a chapter on utilizing continuous integration in your Brownfield project, this segment discusses the creation of artifacts during the build process, including controlling the contents of your final compiled assemblies and creating folders for individual releases. For the table of contents, the Author Forum, and other resources, go to http://manning.com/baley/.

When working on an automated build process, you will create a number of artifacts that will be used in the process. At a minimum you will have a build script file. You may also end up with supporting files that contain scripts for database creation and population, templates for config files for different testing environments and others. These are files that you want to have in source control. Because these files are related only to the build process, we recommend that you organize them in an isolated area, as shown in Figure 1.

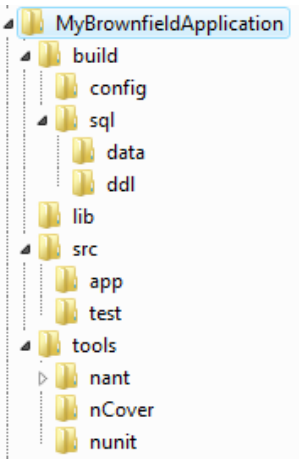


Figure 1: Example of isolating the build artifacts that will exist on your project.

In the solution and Version Control System folder structure we recommend a folder specifically for build artifacts. We also recommend that you create a project within your Visual Studio solution to hold these files.

As you can see in our sample folder structure, we have created subdirectories to store different types of files. The config folder would store any configuration files that will be included in your release package. This would also include any templates that you have that are used for the creation of configuration files.

NOTE:

Templates are a great way to add environment based configuration values to your project's automated build process. Create a separate configuration template for each environment and include in it anything that is environment specific. For example, include your database connection string (you do have different databases for each environment right?), logging configurations, and third-party service endpoints. Once you have your environment specific template files, you can use your build-scripting tool to gather the appropriate values depending on the environment that you're building for.

We also show a separate *sql* folder which includes folders for *data* and *ddl*. The purpose of the *sql* folder is to have a repository for any Sql scripting artifacts that you may have for the creation of your release. The *data* and *ddl* scripts are separated primarily for file management reasons. The *ddl* folder would hold all scripts needed to create a database, its tables, their keys and relationships and any security entries needed. The *data* folder contains all the scripts for priming the database. The scripts in the *data* folder should not be adding data to the database for the sole purpose of enabling integration tests. Priming a database for testing should be handled through separate means.

The level of difficulty when creating your build script can be reduced based on the directory structure that you have created for your project. We recommend having five folders, as shown in Figure 1: Build, Doc, Lib, Src, and Tools. That recommendation is partially based on good organization of the files, concepts and tools. It's also partially based on the need to make the build script as simple as possible.

By segmenting the source code from the test code (the app and test folders in Figure 1), we can make use of the command line compiler (csc.exe or vbc.exe) to easily script the building of the entire application or test suite into one assembly with just one simple NAnt build task. Like we mentioned before, you could use the solution build method to create a minimal build script, as shown in Listing 1.

Listing 1: A Minimal Build Script

```
<csc target="library" output="${compiled.assembly}" debug="true">
  <sources>
    <include name="${dir.project.app}\**\*.cs"/>
  </sources>
  <references>
    <include name="${thirdparty.nhibernate}"/>
  </references>
</csc>
```

Using csc.exe or vbc.exe, as is shown in Listing 1, offers you the ability to control the contents of your final compiled assemblies at a much more granular level than using the solution file. Making this transition allows you to view the solution structure that appears in the Solution Explorer in Visual Studio as a file organizational tool. This takes the pressure away from trying to squeeze your logical or physical layering into the solution/project way of thinking.

The lack of solution/project alignment doesn't mean that we've abandoned structure altogether. Instead we are placing the files into folder structures that make hierarchical sense when we are working in the editor. We still must align our compilation of these files with the deployment needs of our application and environment. Depending on your situation, this can allow you to compile your application into one assembly or executable per physical deployment location.

There is one thing to be aware of when moving to a project/solution structure favoring a scripted compilation. It is possible that you will create a structure that will no longer allow for direct running and debugging in the IDE. With Visual Studio this would mean that you may no longer be able to hit F5 and step into code with the debugger. For some developers this will be a major friction point in their personal development process (code, debug, verify, fix, repeat). For others, the transition to relying on automated tests to do their debugging and verification will be

more fluid. We recommend that, even if your code will debug in the IDE, you work on having the development team build confidence and comfort in using the automated tests instead of debugging for verification.

Challenge Your Assumptions: Solution Explorer is not a designer for layering your application

When we first started to learn logical and physical layering in our applications we were taught to distinguish the separate layers by creating separate projects in our Visual Studio solution. We ended up with solutions that have projects called <*something*>.Data, <*something*>.UI, <*something*>.Business, and so on. While this approach has worked, it has also constrained us to a mentality that the IDE, and more specifically Solution Explorer is a tool for designing the logical and physical layering of our applications.

Another side effect is that everything in Visual Studio begins to slow down as we add more projects to the solution. We have worked on projects where there are 50, 60 and hundreds of projects in one solution file. Without fail, developers on those projects complained about performance opening the solution and compiling projects.

Instead of suffering this in the name of “we’ve always done it that way”, we suggest that you look at the Solution Explorer as a tool for organizing files. Create projects when code concerns require you to, but don’t let your logical and physical layering dictate them. Instead let namespacing and folder structures in one project delineate the layers.

When it comes time to compile your application, use a build scripting tool, like nAnt and MSBuild, to compile the raw code files into assemblies as you would like to deploy them. The results may include many files for each physical layer that you are deploying to. Also consider the possibility that creating one file per physical layer may be the simplest option available without harming the execution of the application.

The concept of building all of the code files into one assembly is an aggressive one. You must be very aware of the environmental constraints your application will face. You must also take the time to ensure that compilation into a structure that differs from the one that appears in the Visual Studio Solution Explorer will be successful.

If you are thinking that this drastic difference in structure and compilation could cause significant and painful integration, you could be right. You could be right if you are not using the automated build script to continually integrate the code (in this case integration is the act of compiling) on a frequent and continual basis. This, above all other things, proves the worth of making an automated build script that can be run easily, frequently and locally by developers. If that build script can quickly provide positive and negative feedback to the developer, problems resulting from the difference in compilation and structure between the script and the IDE will be addressed often and before they become a problem for the entire team.

Although not build artifacts, your build script should create two additional folders in your local source tree. When compiling the code into assemblies and executables, you will need to put those compiled files. Having your script create a *compile* folder is a good way to keep the results of the build process separate from the raw code and build artifacts. The *compile* folder also can be used as the working directory for the automated testing.

The second folder that you may want to have your build process is a *release* folder, as shown in Figure 2. This is a local staging area for the creation of the items needed in a release. If you’ve gotten to the point where you are automating your releases, this folder will be used as the master for the release. To create the release you will pull compiled assemblies from the “compile” folder and consolidate them into the “release” folder in the same structure that will be needed for deployment. If you are deploying to multiple servers, or to clients and servers, you will probably want to have the script create distinct sub folders inside of “release” for each of those deployments.

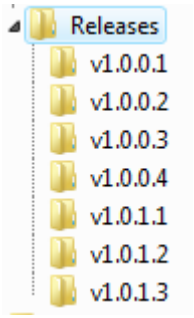


Figure 2: Example of creating a release archive.

Neither the *compile* nor the *release* folders should be added to your Version Control System. If anything requires version based archiving it is the contents of the *release* folder. This should be done by moving the files to a folder, identifiable with the build's version number, in file system storage. Not all builds are ones that should be stored for long term archiving. Because frequent builds can create dozens of "release" archives per day, you may want to purge versions that are older than a certain point in time. This is fine, but you should always make sure to keep the most recent release to the acceptance testing environments. It's probably even a good habit to keep the release that was deployed to that environment prior to that as well.

From the start of creating your automated build scripts, through the files that will be required for compilation and on to the archiving of the final releasable assemblies, there are a number of things that will be considered build artifacts. Maintaining a flexible and reliable folder structure sets the foundation for allowing you to achieve a clean, efficient, and maintainable build process all while increasing the speed of the feedback loop.