DGP Maintenance and Support

Distributed software systems require multiple different types of maintenance and support work, which can be divided into three broad categories: code, infrastructure, and users. DGP code maintenance can be further broken down into server code maintenance and client application code maintenance.

At a high level, DGP's architecture behaves as a two-tier thin client/fat server system in which the client application logic and server API logic are independent of each other thanks to the generic API messages they use for communication. The independence of the client applications from the server tier means that basically any type of client application, using any desired combination of operating system and programming language can be used to build them. The only requirement is that the client application must be able to create API request messages and read API response messages, both of which are nothing more than structured text (XML fragments). It also means that maintenance and deployments of client application code are totally independent of the web service APIs thanks to the immutable append-only deployments and the "feature toggle" functionality of the data-driven RBAC security subsystem.

Server API code maintenance is covered in the Evolution documentation, and security maintenance is covered in the DGPDrive admin UI user guides. The other type of server tier maintenance work is to patch the infrastructure servers, operating systems, app servers, network appliances, etc. In addition, maintenance includes performing periodic database server maintenance, etc.

Client Code Maintenance

Both the native client applications and the web service APIs for the DGPDrive prototype have been built using the full .NET Framework 4.8.1, which is considered to be part of Windows itself. No installation of an application is necessary. Deployment consists of copying and pasting a folder to the storage of the destination computer. The same immutable append-only deployment process is used for both the client app and the web service APIs. A parent directory represents the application, and each release folder (named for the date it was created) is added as a new subfolder below the parent directory. Editing the shortcut to point to the path of the new executable file (which is actually an assembly in .NET) allows the new version of the application to be run. If any problems occur, pointing the physical path back to the previous executable, plus deleting the folder of the failed new version constitutes a "rollback".

Server Code Maintenance

Thanks to the immutable append-only conventions which prevent breaking changes to backward compatibility, plus the ability to use the RBAC security system as a feature toggle mechanism, deployment of new versions of the web service APIs can be done incrementally on each server, in each location. In other words, the system tolerates the fact that the servers within a location, and between locations are not all exactly the same version during the maintenance process. New methods can be “switched on” in the security system of each environment after all the servers in each location have been upgraded.

DGP has been designed in multiple ways to eliminate code maintenance work within a software system, and make other necessary types of maintenance as easy as possible, with no maintenance window downtime for the system as a whole. The most important convention used to achieve this objective in practice is “immutable append-only”. This pattern is used at many different levels throughout the architecture, design and deployment of DGP systems. The reason it works so well is because it eliminates the problems caused to a software system whenever changes/additions are merged into existing code, data, and infrastructure.

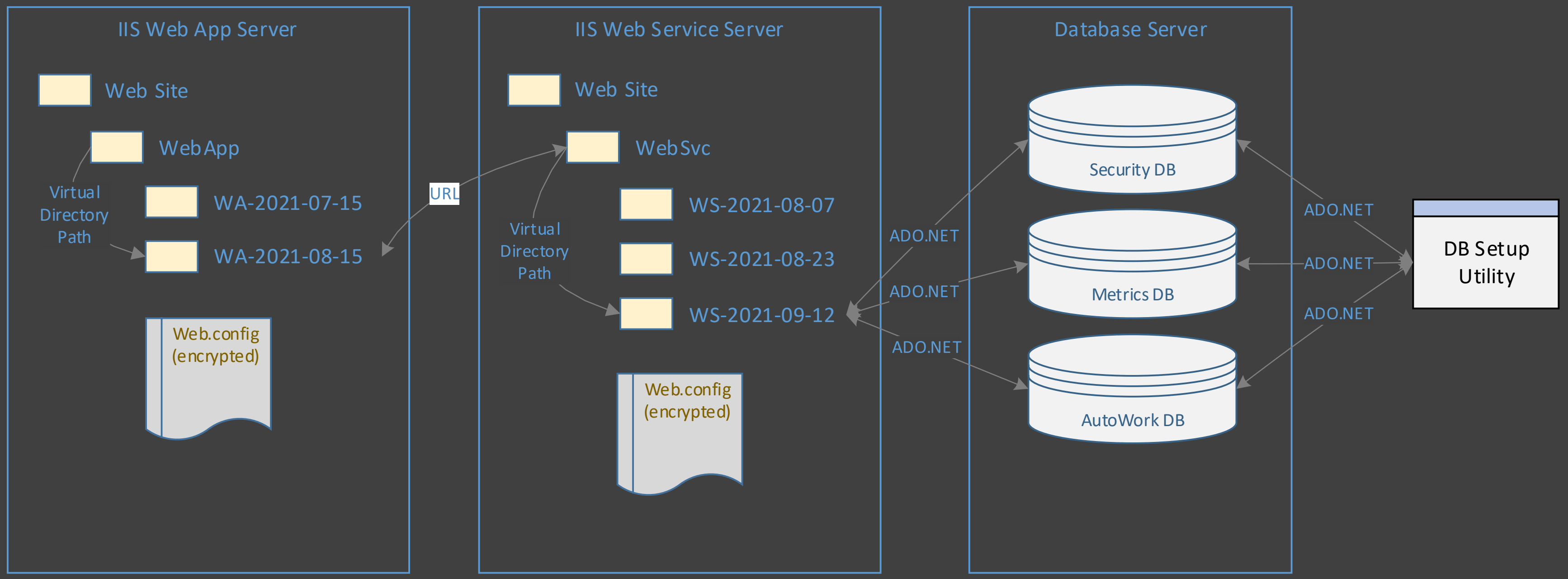
Instead of merging these changes into an existing system, every current version of code, data and infrastructure are preserved unchanged, while modifications are added (appended) to the system alongside them. The only exceptions to this rule are bug fixes to existing methods and/or data that have been found to have flaws that were not detected during testing. The effects of these immutable append-only conventions on software systems are significant. Every addition can be considered to be its own miniature prototype, allowing the code, data and infrastructure to be easily rolled back to its previous “known-good” state whenever one of these experimental prototypes fails. This helps to significantly improve the overall quality and reliability of DGP systems.

This immutable append-only convention used within the code is also what makes it possible for a system to require zero code maintenance work. Once deployed into production, each method, method test file, and method documentation page become immutable, and therefore require no work to maintain them going forward. As a result, the only development work for a DGP system consists of the creation of new extensions, enhancements and additions, along with an occasional bug fix for any flaws that make it past testing. Also, by following this convention each existing method becomes more and more “hardened” and reliable over time through constant use. The importance of this practice for reducing the total cost of systems while also significantly improving their overall quality and reliability cannot be overstated.

Software deployment process:

1. Use the DGP DBSetup utility application to apply immutable append-only changes to the database schemas and core data. The process to do so is idempotent, testing each action before it is executed to see if each action has already been completed.
    a. In general, additions made to the database schemas can be difficult (but not impossible) to roll back, so these additions are designed so that they are unobtrusive and do not cause any problems for backward compatibility. Ideally, this eliminates the need for any of these changes to be rolled back in the first place.
    b. Optionally, replica core data is also synchronized by the DBSetup utility as if it had been replicated and merged from a non-existent master database. Security data is the most common type of core data to be synchronized.
2. A new version of the web service APIs are deployed as a version folder, added to the collection of similar folders below the parent application folder.
    a. The web.config file of the new version folder must be edited for each specific environment. After the first such deployment, the easiest way to do this is by copying a web.config file from a previous version folder.
    b. The physical path of the application's virtual directory is then edited to use the new version folder. Rollbacks consist of editing the path to point back to the previous version folder if necessary. The failed version folder is then deleted.
3. New added functionality cannot be accessed by any users until it has been set up in the data-driven RBAC security system and assigned to a role. Generally, new API methods will initially be assigned to a tester role. The new methods will then be tested using the API test harness application and other applications. After passing the tests in a given environment, the new methods can be be assigned to additional roles for more widespread use.
4. Client applications:
    a. Web applications – web applications are deployed using a similar mechanism as the web services.
    b. Native applications – native apps also have multiple version folders below a parent application folder, but only require that any shortcuts be edited to point to the physical path of the executable in the new version folder.

The RBAC security system also provides help with deployments of new API methods. Once a new API method is initially deployed in an environment, no user is authorized to use it. It must be added to a role, and users assigned to that role before any user is authorized by the security system to call a specific method. New methods will at first only be added to test roles, and after successful testing in an environment, can then be added to other roles accessible to more users.



Server Infrastructure Maintenance

A redundant DGP system will usually have 2 locations, a primary and a backup. The Backup location can be taken offline for maintenance while the Primary location continues to function as normal. All nodes within the offline Backup location can then be taken offline to be patched, updated and maintained while all users continue using the Primary location with no interruptions.

When the maintenance for all servers is complete, the Backup location is brought back online, and allowed some time to synchronize data with the Primary. Once that data synchronization is complete, the Primary and Backup designations of the two locations can be reversed. This will gradually force client applications to switch from the old Primary to the new Primary location. Once that switch is complete, then the newly designated Backup location can be taken offline for maintenance, and the maintenance process repeated. Once that is completed, the Backup location is brought back online, but remains as the Backup going forward. In this way, the two locations alternate between Primary and Backup each time the maintenance process is performed.

Another option is to maintain the nodes in a location without taking the entire location itself offline. This is accomplished by removing single servers from their respective farm or cluster, performing the maintenance process (patches, upgrades, etc.), and then adding the server back into the farm/cluster when finished, sequentially doing the same work to all servers in the location, one at a time. This is much more difficult than designating an entire location as offline for maintenance, and will generally not be necessary.