



DIAVOLO VB

**Setup and servicing sub-system
An overview**



Diavolo update and service sub-system is an evolution of its Rainbow Portal ancestor. While RP update system was simple but very effective, Diavolo aims to introduce a new, more advanced sub-system which will become the basis for further enhancements.

The Rainbow Portal update system

RP update system was very simple but it was incredibly effective. It was based on comparing RP code version with database version and then apply a list of SQL scripts and install new modules. It had many limits as, for example, it could lead to partial updates and couldn't easily perform complex upgrades.

New update sub-system

New update system has been introduced as of Diavolo 3.0.0.1883 (release version 1883). It can then successfully upgrade Rainbow Portal v1.6 installations as well as v2.0.0.1881 (release compiled for .NET 2.0, the basis for Diavolo).

The basis of this new system aren't scripts anymore. Rather, the basis are so called *upgrade objects*, that is separate compiled DLLs which can perform atomic upgrades. Such objects can perform way more complex upgrades and updates than usual SQL scripts which Rainbow Portal was doing. In fact, *upgrade objects* are meant to perform much more than simply executing SQL scripts.

From SQL scripts to upgrade objects : perform complex operations

To further emphasize the difference between the old and new system, just imagine that a new upgrade should be applied to current portal. Imagine that you need to install a new module which access a remote service (weather, feeds, maps, whatever...) and imagine that module requires a key to be requested and then used during remote calls.

What would it take to install that by using RP? Database should be updated, then user should visit service website, request key, go back to portal, configure module to use key and then display it. Pretty normal.

However, an *upgrade object* could just upgrade your database, connect to remote server, request a key, install module and set key inside database then validate service functionalities and complete. A much more complex scenario.

Each object is an atomic operation

While SQL Scripts can succeed, fail or partially succeed, exposing database to dangers of difficult rollbacks, each upgrade object is an atomic operation. If main method used to start upgrade will fail, Diavolo will rollback any change to database and offer object itself a way to recover other operations (for example, cancel a subscription of a remote service). This makes more difficult to leave your database in a non-consistent state. Additionally, *upgrade objects* can also perform pre- and post-upgrade operations because Diavolo will invoke specific methods for such events. Pre-



upgrade events are perfect to check conditions which are needed to successfully complete current upgrade for example by checking availability of a remote service or checking that database is consistent and ready to be upgraded.

Post-upgrade events are perfect to finalize operations and to perform clean-ups.

This design will ensure that almost any scenario will be supported.

Upgrades released as DLL libraries

To use the new system, upgrades and updates will be released as individual DLL libraries. This will led to many advantages:

- each library is a self contained upgrade and will include all data and logic needed to perform upgrade;
- DLLs can be signed to verify their sources: this can provide additional security in a distributed, open-source scenario and could provide additional value for developers;
- upgrade libraries will be stored inside *bin/setup* folder and thus will be separated from other libraries. That will make it easier to upload and remove when upgrade has been executed;
- upgrade objects will include a unique ID which will prevent to apply the same update more than once;
- libraries can choose to allow partial updates, which would normally not be allowed.

Moreover, Diavolo will allow a more proactive upgrade procedure since portal will not look for updates until admins choose to do that by setting a value in *web.config*. This way, admins will have more control over update procedure and portal will gain speed improvements since will not look for updates until necessary.

Friendly to developers

Up until now, RP developers had little chance to exploit update system as this was reserved to main system. Problem was to start an automatic update, you had to change code version and doing so would break compatibility with future RP releases.

Diavolo aimed to change this by introducing an update system which is friendly to developers and allows ISV (independent software vendors) to build their own application based on Diavolo while still being confident that they will be treated as first-class citizens and they will be able to apply updates which will be released later.

Diavolo *upgrade objects* introduce *Minor Release Version*, a code which is dedicated to custom portals and thus to developers. While *Release Version* is used by Diavolo to keep track of system changes, *minor release version* can be used by developers to service their custom portals while still benefiting of all infrastructure service like auto-discovery, pro-active updating and security.



Upgrade objects can be built to deliver custom updates and portal administrators will be offered to apply patch the very same way they would be when system patches are available. Furthermore, custom updates will enjoy a strict control over versions core Diavolo components use. For example, patches can be tied to specific minimum minor version, in order to ensure that database and the whole system will not be non-functional after applying them. Moreover, minor release patches can be tied to specific minimum major release version, thus ensuring that custom patches will only be applied to compatible Diavolo installations. That not only gives access to update infrastructure but ensure predictability over patches and lower chances that a patch might fail to be applied.

Developers will also be able to provide upgrade objects to solve complex custom situations. For example, developers could provide libraries to convert custom installations to standard installations in order to achieve compatibility with future enhancements. All these kinds of scenarios are supported and the most important thing is developers are now allowed to service their custom installations the very same way we will service Diavolo.

That will effectively allow to develop custom application based on Diavolo, while still being sure to be able to evolve custom extensions while still maintaining compatibility with main Diavolo codebase and allow main codebase to evolve. This will dramatically enhance the usefulness of our platform. This infrastructure will also dramatically improve user experience when dealing with updates as, in most cases, users will only need to drop&run such patches.

Custom upgrade objects could also be signed to ensure their source, something very important in a open-source context when dealing with businesses or institutions, for example.

Note: best practice suggest that minor release must not affect system objects, thus not changing the core of Diavolo. If developers follow that rule of thumb, they are guaranteed not to have problems in applying both system and custom patches.

Auto-discovery of updates: the drop&run paradigm

One of the most interesting features of new update system is the ability to drop upgrade libraries inside the *bin/setup* folder and have Diavolo auto-discover available patches. Users can simply drop upgrade objects (i.e. DLLs) inside that folder and Diavolo will present a list of available patches upon running portal or by a dedicated admin module.

That represents a dramatic evolution for portal administration as most users are not skilled enough to perform complex operations. A self-contained patch object, especially if it can be auto-discovered upon upload, would make upgrading and updating your portals much easier. And size of such libraries is not a concern as after installation, those files can be deleted from server.

Focus on database consistency and multiple update scenarios

As previously described, part of the high value of this new system is the ability to support many kinds of scenarios and prevent most common errors which led to problems in portals. The most common problem is database consistency which is a result of a failed update and that's the most



dangerous thing which could happen because it will leave the database in a mixed state where the new codebase cannot properly operate and rolling back to previous version is not so simple (unless you have a full backup of your db you could restore). Users are usually not able to cope with this scenario and that causes many problems and portal downtime. Also, current RP users have to be a few experienced in database administration to be able to properly operate a portal, at least to manage upgrades and some other little problems (for example, trimming down monitoring table when it's enabled).

This led to a sharp contrast with other CMS solutions , especially those based on PHP, where user friendliness is better and where users are usually only required to drop new PHP pages in specific folders and enable new extensions to enjoy new features.

Our new system would allow to plan delivery of updates for any portal or for a specific portal (say, v1881 or 1884 and so on), thus reducing the danger of upgrading from incompatible version. Moreover, we allow custom patches (minor release) to follow the same schema so a developer could offer a patch to be applied to minor release 85 or fail if minor release is less than that. But developers could also require that at least major release 1881 to be installed in tandem with minor release 85, thus requiring that a 2.0.0.1881 (minor release 85) to be installed before applying that patch. Scenarios are endless.

We would also like to emphasize how such mechanism allows developers to create custom and specific evolution roadmaps for their applications based on Diavolo. For example, we usually provide an upgrade from Rainbow Portal v1.6.0.1879 or v2.0.0.1881 to Diavolo 3.0.0.1883, that is a path from last and up-to-date RP version to first Diavolo version. However, community and/or single developers could also provide an upgrade object to service RP v1.5.0 and put that portal in pair with release 2.0.0.1881. Or a developer could apply a patch to turn his/her highly-modified custom RP-based portal in pair with Diavolo 3.0.0.1883 in order to regain compatibility with our releases and exploit the ability to provide custom patches to evolve their modifications.

Our goal is to give users less headaches and let developers do the dirty job, which is why they've been paid for ;-)

By introducing a simple but effective tracking system, we provide developers easier and predictable ways to do their dirty job thus, in the end, providing a value to both users and developers. Constrains can be introduced to exactly manage different situations and provide mechanism to recover and solve most problems in a disconnected way. We also aimed to provide developers a simple way to provide users more value by introducing ways to manage both our needs and their needs.

Finally, we also provide a way to build custom solutions on a proven, business-friendly, open-source-based codebase while still supporting different situations and evolution. Developers can now simply plug their skills into our platform and focus on developing extensions, if that's what they really want.