# Update Processor

The Update Processor is a console-based utility that facilitates making changes to existing applications. It is used to distribute regular *code updates* (analytics updates, rule updates) to applications like STARS Sentinel and STARS Interceptor. It is also used to distribute patches and version upgrades to all applications in the STARS Solutions suite. Its most common uses involve executing SQL Statements on the client database servers and/or updating files on the client servers, such as JAR files containing executable source code.

### Distribution Package

To distribute such updates to clients, a package is created containing a number of important files. At the root of the package directory is the file used to initiate the update, called **runUpdate**. This file comes in two varieties: a .bat (batch script) file for Windows and a .sh (shell script) file for UNIX. There are small discrepancies between them but the basic purpose is the same. Clients will first update the corresponding **setUpdate** file (which comes in the same two system-dependent versions) and then execute runUpdate, which will trigger the UpdateProcessor itself by invoking its JAR file via the command line.

## Execution Code Flow

#### CodeUpdate.java

The primary driver class in the Update Processor is the **CodeUpdate** class. Despite its name, this class is used for both code updates and migrations, and the codeupdate package name is the primary package of the UpdateProcessor framework. This class extends the more general DIStepJob class, which is found within the *fwa-persistence* package. On execution, this class prepares the environment by initializing key classes like the singleton MigrationInfo class, and prompting for user input as needed.

#### DriverFileProcessor.java

With the environment prepared, CodeUpdate invokes the **DriverFileProcessor** class to begin the actual work. This class steps through the driver files, evaluating each line to determine whether or not the specified action needs to be run.

There are a number of reasons that a line in a driver file will be skipped. If the corresponding **AppDomain** (corresponding to database schemas, e.g. “SINT”=”Sentinel”) doesn’t exist in the current database environment then its actions will be skipped. For patch steps, it will skip the step if that specific patch has previously been applied, or if a newer version number of the product exists (e.g. it will not install patch 18.4 if any updates from version 19.x have already been installed).

Update\_log

For code updates, the DriverFileProcessor will additionally check a special table in CORE called UPDATE\_LOG to see if the current line (or a line with the same text) has ever been applied in the past; if so, the current line will be skipped. For migrations, the UPDATE\_LOG table is not checked, though it is still updated. Skipping lines based on the UPDATE\_LOG table entries can be toggled off by passing “-r true” as a command-line parameter when invoking the Update Processor.

## Driver File Format

Driver Files are a custom file format. They are stored as standard ASCII text files and typically use a .txt file extension. The logic for processing these files correctly is found in the DriverFileProcessor class.

#### Comment Lines

Within these files, lines beginning with “--” or “//” are understood to be comment lines, and they are not processed. Note that the comment symbols must be the very first characters on the line; it is not possible to append these to the same line as uncommented code. Additionally, lines beginning with “/\*” are recognized as (potentially) multi-line comments, and all subsequent lines are ignored until the matching “\*/” delimiter is found.

#### Step Class Invocation

A typical line in a Driver File begins with the name of a class to execute, usually ending in the word “Step” (e.g. “CodeUpdateStep”). For historical reasons the full package name is sometimes specified, though just the class name itself is sufficient.[[1]](#footnote-1) Any of the classes in the UpdateProcessor framework that implement the IStep interface may be specified. These classes have been designed to be invoked by driver files.

#### Passing Parameters

Most step classes expect one or more parameters when invoked. To pass these from a driver file, follow the step class name with a comma, then include each parameter also delineated by commas. Leading or trailing whitespace is trimmed, and values may optionally be wrapped in quotation marks (“). Using quotes makes it possible to pass parameters that include commas.

Putting it all together, see the following example: it invokes the ExecuteSqlFileStep with two parameters. Note that the string “Creating tables, as needed” is a single parameter because, though it contains a comma, it is wrapped in quotes.

ExecuteSqlFileStep, Create\_tables, "Creating tables, as needed"

#### AppDomain Prefix

When using the UpdateProcessor for migrations, typically the updates to many different products is distributed as a single distribution. In this case it is useful to specify the AppDomain (product schema) for steps so that products that are not installed on a particular environment are skipped over. Rather than repeat this on every line, it is possible to specify the relevent AppDomain for a particualr step, and then the Update Processor will remember this domain and use it for all lines within the specified step. This can be done by specifying the target AppDomain name at the start of a line, followed by a colon, followed by the standard procedure as outlined above: the class name and its parameters.

SSNL: ExecutePatchFileStep, DataUpdate, “Sentinel Update”

Consider the preceding example. In this case, “SSNL” (Sentinel) indicates the target AppDomain. In this case, the specified driver file (called “DataUpdate”) will be invoked if and only if Sentinel is installed on the target machine. Additionally, within the DataUpdate driver file, it will remember that SSNL is the current AppDomain, so this will not have to be restated for its steps.

It is possible to nest these domain invocations, so that within the example outlined before, the DataUpdate driver file may have a call to “CORE: ExecuteSqlFile, UpdateTables”. This follows the same procedure: the specified sql file (“UpdateTables”) will be invoked if CORE is installed, and within that file, it will recall that CORE is the active domain. When that file is complete, execution of the current file (“DataUpdate”) will proceed, with SSNL as the active domain as before.

## Step Classes

There are scores of step class files available for use, so it is recommended to browse the UpdateProcessor package (i.e. the codeupdate package) to see them all. Some classes are of particular importance though, especially for migrations; these will be outlined here in greater detail. It is possible to implement an entire migration package using only these classes.

### ExecuteSqlFileStep

This general-purpose step is used to execute a specific SQL file. It only requires one parameter: the name of the SQL file to execute. This name *may* include a file extension, but it is recommended to specify only the filename itself without the extension. This enables you to use the extension to distinguish database types: see below for details.

Directory

[todo]

File Extension

If the specified name of the SQL file does not correspond to an exact filename – e.g., if it was specified without a file extension – then the step class will automatically search for a file with the given filename whose file extension is “.ora.sql” if running on an Oracle environment, or “.mss.sql” if running with MS-SQL.[[2]](#footnote-2) Failing this, it will use a file with the given name whose extension is simply “.sql”. Note that it will *never* invoke a file like \*.mss.sql if running on Oracle, or vise-versa.

In this way, it is possible to write multiple versions of a file for those that require it; while if a single SQL file is compatible with both Oracle and MS-SQL, then it may be given the simple “.sql” extension, and it will be invoked on both environments.

1. The full package name of the class is automatically appended so long as the class exists in one of the auto-loaded packages specified in codeupdate.PackageClassLoader. [↑](#footnote-ref-1)
2. Support for a Teradata-specific file extension such as “.ter.sql” file extension has not yet been added but it is recommended that this feature be included in the future. [↑](#footnote-ref-2)