

# Creating a P4DTG Plugin: Integrating Perforce and Defect Trackers

Alan H. Teague, Perforce Software

## Overview

The Perforce Defect Tracking Gateway (P4DTG) links fix information in a Perforce server to defects in a defect tracking system (DTS). P4DTG uses the Perforce jobs system to associate source code changes with defects. When a developer submits a changelist associated with the specific job or jobs that it fixes, the fix information is propagated to the associated DTS. When a QA or Technical Support engineer adds a new defect to the DTS, it is propagated to Perforce as a job. Developers can then search for applicable jobs using either the DTS or Perforce Jobs.

## The Architecture of P4DTG

P4DTG has two core components: the *configuration tool* and the *replication engine*. Both programs load shared libraries called *plugins* to communicate with Perforce servers and DTS servers. Using the P4DTG Software Developer Kit (SDK), you can create plugins to integrate your defect tracker with Perforce. This paper provides an overview of the process of developing a P4DTG plugin.

The following diagram shows the relationships among the processes and files used by P4DTG.

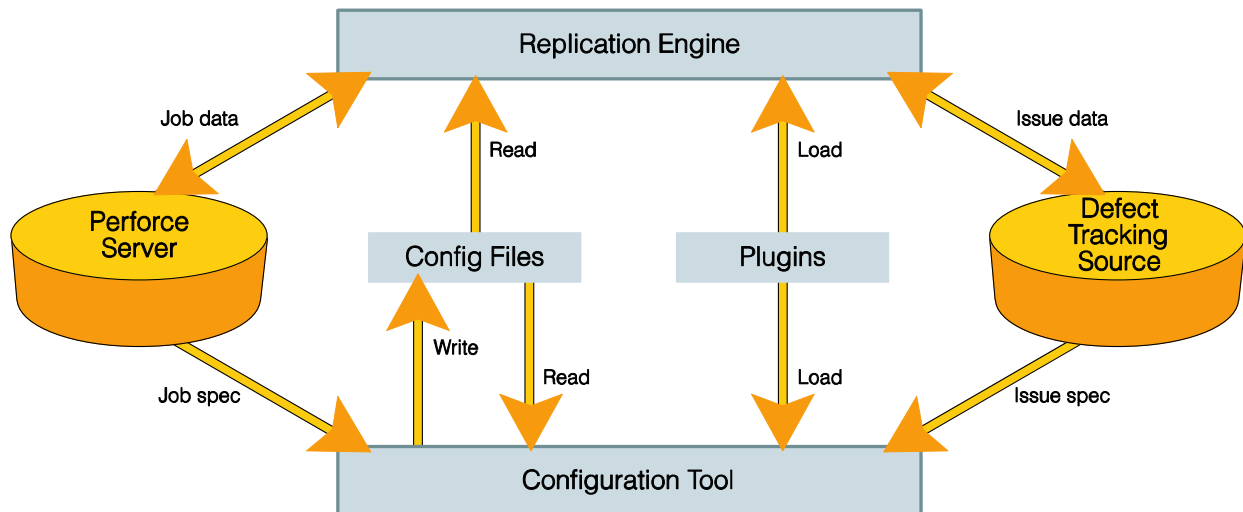


Figure 1: Architecture of the Perforce Defect Tracking Gateway

## The Configuration Tool

The configuration tool is a GUI that creates and edits configuration files, which contain the details about how data is replicated. The configuration tool uses plugins to obtain

the metadata from Perforce servers and DTS servers. The metadata specifies the field names and data types for job and defect data.

## The Replication Engine

The replication engine uses the plugins to verify the replication mappings and to update jobs and defects during replication. A specific instance of the replication engine links one Perforce server to one DTS server. The DTS server may be another Perforce server running Jobs as a primary defect tracking system.

## Configuration Files

Three types of configuration files contain the details for each replication instance:

- **Data source files:** contain the connection details for a specific Perforce server or DTS server, referred to as *data sources* in the configuration tool. A data source can contain one or more *projects*. A project is a set of defects having a common structure and a unique set of defect identifiers. Perforce jobs are treated as a single project.
- **Gateway mapping files:** contain field-to-field mappings that associate job data with defect data for a specified pair of data sources, typically one Perforce server and one DTS server.
- **Status files:** contain the current status and settings for an instance of the replication engine. These settings define the starting time for replication and record when the last replication cycle occurred.

These configurations are written to XML files that are stored on the machine running the configuration tool and replication engine. The format of these files is platform-independent.

## The Replication Process

When a replication engine is invoked, it performs the following steps:

1. The engine evaluates every defect in the DTS that was entered or modified between the specified start date and the current system date and time.
2. The engine makes a single pass through newly updated defects (skipping any updates originated by the dedicated user associated with the engine itself.) checking for updates that occurred during the first step.
3. Finally, the engine goes into a loop, polling for user changes made to defects or jobs, replicating them as specified by the mappings. This cycle repeats until the replication engine is turned off or encounters a fatal error.

## Replication Errors

Most errors that the replication engine encounters are not fatal. For non-fatal errors, the job is marked with the error and the replication engine skips that job and its associated defect during subsequent replication cycles. Non-fatal errors are typically one of two cases: (1) a change made in Perforce violates the workflow in the DTS and the DTS

rejects the change; or (2) a **select** field on either side contains old values that are no longer valid and are not mapped. To fix these types of error, correct the originating issue or job and then clear the `DTG_ERROR` field in the associated Perforce job. (When correcting jobs, do not connect using the Perforce user that is assigned to the replication engine). The corrected job and issue will be checked and replicated during the next replication cycle.

More serious errors require you to stop the replication engine, correct the errors and then start the engine with the **force** flag set. This flag verifies that all of the jobs and issues have been replicated.

## ***Developing Plugins***

For each type of defect tracker, the replication engine requires a plugin. The plugin enables the engine to communicate with defect tracker servers by correctly formatting required data fields and by defining required methods. (The engine also uses a plugin to communicate with Perforce servers.) Plugins enable P4DTG to be extended to support a variety of defect trackers without requiring changes to the replication engine or the configuration tool.

The following sections tell you how to create plugins.

### **The Structure of a Plugin**

Each plugin for P4DTG must provide a predefined set of functional interfaces, opaque types, and specialized data types. Both the functional interfaces and the specialized data types are C-based to prevent compiler/operating system issues in interacting with a shared library.

The **functional interfaces** are hierarchical, with specific functions requiring results from successful calls to outer-level functions. P4DTG requires a plugin to define the following levels of interfaces:

- **Library-level interfaces** exercise the plugin specific functions and do not require a connection to any specific server.
- **Server-level interfaces** are the functions that are available after a base connection to a DTS server has been established.
- **Project-level interfaces** are the functions that are available after a connection to a specific project on a connected server has been made.
- **Defect-level interfaces** are the functions that are available after a specific defect from a specific project on a connected server has been retrieved from the DTS server.

The **opaque types** are essential to the functional interfaces. These types encapsulate the data structure that the plugin uses to identify a connection, a project in a connection, and a specific defect in a project. Both the configuration tool and replication engine use the connection and project objects. Only the replication engine uses the defect objects.

The **specialized data types** contain lists of identifiers (project, job and defect names), dates, field definitions and field name/value pairs. The SDK provides utility functions to assist in the creation, modification and deletion of these types. By using these utility functions, you can avoid the problems caused by mixing calls to the memory allocator functions **malloc** and **new**.

## Designing the Plugin

When designing a plugin, consider the following:

- What level of integration is possible?
- What type of workflow is enforced by the DTS? How does that workflow determine which fields must be replicated from Perforce to the DTS? Does your workflow permit the defect tracker to accept defects originating as Perforce jobs?
- How are fields to be mapped? Which fields are to be read-only?
- How should the plugin handle connection authentication?
- Which operating systems and platforms will be supported?

## Levels of Integration

The type of API provided by the DTS mainly determines the level of integration. A full-functionality, TCP-based API provides the best level of integration. If a DTS does not have an external API but the underlying database used by the DTS does, a limited but useful integration is possible. If a DTS uses an interpreted API, a wrapper is required to integrate it into the plugin.

**Important:** If a plugin is interacting directly with the database, you are bypassing the business logic contained in the DTS. In this case, ensure that the plugin does not update data in ways that violate the workflow enforced by the DTS. To preserve the workflow, do not enable the plugin to create defects by replicating Perforce jobs, or to change the state of defects based on changes to Perforce jobs.

If your plugin supports a defect tracker that supports multiple types of databases, you can encapsulate all of the supported databases using a single plugin (handling the differences internally), or create one plugin for each type of underlying database.

## Data Mapping

P4DTG supports the following abstract data types for defect fields:

- **WORD**: single-word text fields
- **LINE**: single-line text fields
- **TEXT**: multiple-line text fields
- **SELECT**: fields that are single selections from a fixed list of values.
- **DATE**: date/time fields

The plugin must also provide conversion between **DATE** and the DTS's standard textual date/time representation.

## Authentication

P4DTG supports simple authentication using username and password. More sophisticated authentication can be built into your plugin. Because the replication engine is a background process, the input needed by this type of authentication must be stored outside of P4DTG, for example, in a file in a known location.

## *The P4DTG Software Developer's Kit (SDK)*

The SDK for P4DTG contains a complete C++ framework for building new plugins. This framework is located in the `mydts` directory. You can use it as the basis for plugin development.

The testing tool, `p4dtg-test`, located in the `test` directory, provides a command line interpreter for accessing all of the functional interfaces provided by a plugin. This tool enables you to build and test your plugin incrementally.

Two example plugins are included in the `p4jobs` and `bugs` directories.

- `p4jobs` contains the plugin for Perforce Jobs and shows the use of a DTS's API for building a plugin.
- `bugz` contains a sample plugin for Bugzilla and demonstrates how a plugin can interact with the database that underlies a DTS.

The `include` directory contains the header files that define the specific functional signatures for a plugin. The `share` directory contains functions that are useful for building new plugins.

## MyDTS Framework Layers

There are three layers in the framework:

- The **exported C functions** for the shared library map the exported interface to the MyDTG class layer. `DTG-mod-cpp.cc` contains the functions for this mapping. This layer should not require any changes for implementing your plugin other than modifying the name of the trace file used for protocol level debugging of the plugin. Most plugins will never need to be debugged at this level.
- The **MyDTG class layer** is composed of three classes: `MyDTG`, `MyDTGProject` and `MyDTGDefect`. Instances of these three classes map to the three opaque types passed back and forth through the shared library interface. These classes provide the base logic for how each of these types of objects behaves, including the calls to the MyDTS layer and smart caching of intermediate steps.
- The **MyDTS class layer** encapsulates communication with a specific DTS. This class requires the most modifications when you are developing a new plugin.

## Error Handling

Plugins communicate errors back to P4DTG using the `DTGError` structure. This structure contains both the error message and a flag that indicates whether the

connection may continue to be used. After a connection is made to a server, P4DTG checks for server warnings. To inform the user of any issues related to using an older version of the server, plugins can use the `dt_get_server_warnings` interface. This interface is coded in step 3 below.

## Coding the Plugin

To code your plugin, perform the following steps.

### Step 1: Compile the framework

Compile the MyDTS framework and the test tool. Both of these executables can be compiled without changing any of the code they contain. After you successfully compile them, run the test tool and issue the LL (Load Library), LT (Library Test), and LF (Library Free) commands. If the module is correctly built, it will successfully load, pass the tests, and be released.

### Step 2: Specify plugin name and define date conversion

You must modify the following code:

```
static struct DTGDate *MyDTG::extract_date( const char *date_string );
static char *MyDTG::format_date( struct DTGDate *date );
static const char *MyDTG::get_name( struct DTGError *error );
static const char *MyDTG::get_module_version( struct DTGError *error );
```

Change the strings that specify the plugin name and version, and code the conversion to and from the `DATE` format used by the DTS. After you make these changes, invoke the test tool and issue the LI, LE, LN, and LV commands (See Appendix A for a listing of the test tool commands and required arguments).

### Step 3: Code connection logic

You must modify the following code:

```
MyDTG::MyDTG( const char *server, const char *user, const char *pass,
              const struct DTGField *attrs, struct DTGError *error );
MyDTG::~MyDTG();
struct DTGStrList *MyDTG::list_projects( struct DTGError *error );
char *MyDTG::get_server_warnings( struct DTGError *error );
```

Code the logic required for:

- establishing a connection to a specific DTS
- retrieving a list of projects available for that connection

After you make these changes, invoke the test tool and issue the LC, SF, SL, and SW commands.

## Step 4: Code project logic

You must modify the following code:

```
MyDTGProj *MyDTG::get_project( const char *project, struct DTGError *error );
struct DTGDate *MyDTG::get_server_date( struct DTGError *error );
MyDTGProj::MyDTGProj( MyDTG *dt, const char *project, struct DTGError *error );
MyDTGProj::~MyDTGProj();
```

Code the logic required for:

- connecting to a specific project
- returning server version and date information

After you make these changes, invoke the test tool and run the SP, SD, and PF commands.

## Step 5: Define data type mappings

You must modify the following code:

```
struct DTGFieldDesc *MyDTGProj::list_fields( struct DTGError *error );
```

If the underlying API supports queries that return details about the structure of defect data, you can code logic that retrieves that data and maps each retrieved field definition to the appropriate data type and more importantly to whether the field is writable. One field must be designated as the field that contains the date and time that the defect was last changed. Optionally you can designate fields that contain the name of the user who made the last change and that contain the unique identifier for the issue.

Your plugin can include special-case logic for specific fields, as required by your workflow.

If the API does not support queries that return details about the structure of defect data, your plugin can either hard-code the mappings or use a local file containing the mappings.

After you make these decisions and implement this function, invoke the test tool and issue the PS command.

## Step 6: Test plugin with configuration tool

After all of the above steps have been completed, you can test your plugin using the P4DTG configuration tool. Use this tool to verify that you can configure a data store using your plugin.

## Step 7: Detect changed issues

You must modify the following code:

```
struct DTGStrList *MyDTGProj::list_changed_defects( int max_rows,
    struct DTGDate *since,
    const char *mod_date_field,
    const char *mod_by_field,
```

```
    const char *exclude_user,  
    struct DTGError *error );
```

Code the logic that retrieves the list of defects that have changed at or after a specified date and time.

After coding this function, run the test tool and issue the PL command.

### **Step 8: Define replication retrieval logic**

You must modify the following code:

```
MyDTGDefect *MyDTGProj::get_defect( const char *defect,  
    struct DTGError *error );  
  
MyDTGDefect *MyDTGProj::new_defect( struct DTGError *error );  
MyDTGDefect::MyDTGDefect( MyDTGProj *proj, const char *defect,  
    struct DTGError *error );  
  
MyDTGDefect::~MyDTGDefect();  
  
struct DTGField *MyDTGDefect::get_fields( struct DTGError *error );  
char *MyDTGDefect::get_field( const char *field, struct DTGError *error );
```

Next, code logic for:

- retrieving specified defects
- creating new defects

You can use the test tool to help you develop this logic incrementally. The PD, PN, DF, DL, and DR commands invoke these interfaces.

### **Step 9: Define replication update logic**

You must modify the following code:

```
void MyDTGDefect::set_field( const char *field, const char *value,  
    struct DTGError *error );  
  
char *MyDTGDefect::save( struct DTGError *error );
```

Finally, code logic for the actual updating of data, as follows:

- changing specific defect fields
- saving the current defect

Each DTS updates by either submitting the entire record or by submitting only changed fields. The plugin must encapsulate the behavior used by the specific DTS. You can use the test tool to help you implement this logic incrementally. The PW and PS commands invoke these interfaces.

### **Step 10: Test your plugin!**

After you have coded and tested the preceding logic, be sure to test your plugin for memory usage and leaks, using your preferred testing tools.

## **Conclusion**

P4DTG enables you to connect your defect tracker to Perforce. Using the SDK, you can create your own plugins for proprietary defect trackers or those that Perforce does not support directly. When creating your plugin, be sure that its logic and data mappings support your workflow. Use the P4DTG test tool to develop incrementally, testing as you go.

## Appendix A: P4DTG Test Tool Commands

To test your plugin, invoke `p4dtg-test`, located in the `test` directory. The following table lists valid commands. To list the commands from the command line, specify the “H” option. To free any allocated objects and exit the program, specify the “Q” option.

Type	Option	Function Tested	Remarks
Defect	DF	<code>defect_free(defectID)</code>	Unsets defectID
	DL	<code>defect_get_fields(defectID)</code>	
	DR field	<code>defect_get_field(defectID, field)</code>	
	DS	<code>defect_save(defectID)</code>	
	DW name value	<code>defect_set_field(defectID, name, value)</code>	
Library	LC server user pass	<code>dt_connect(server, user, pass)</code>	Returns a dtID. If no password, specify NULL.
	LE<cr>datestring	<code>extract_date(datestring)</code>	Format of datestring: yyyy/mm/dd-hh:mm:ss
	LF	<code>delete DTGModule()</code>	Unloads the library
	LI yyyy mm dd hh mm ss	<code>format_date(date)</code>	
	LL path	<code>new DTGModule(path)</code>	
	LN	<code>dt_get_name()</code>	
	LT	<code>DTGmodule::test()</code>	
	LV	<code>dt_get_module_version()</code>	
Project	PA fix	<code>proj_describe_fix(projID, fix)</code>	Perforce plugin only
	PC defect	<code>proj_list_fixes(projID, defect)</code>	Perforce plugin only
	PD defect	<code>proj_get_defect(projID, defect)</code>	Returns a defectID
	PF	<code>proj_free(projID)</code>	Unsets projID
	PL n d datef userf user	<code>proj_list_changed_defects(projID, ...)</code>	n: max rows to return d: modified since date yyyy/mm/dd/hh/mm/ss datef: ModifiedDate field name userf: ModifiedBy field name user: Exclude user
	PN	<code>proj_new_defect(projID)</code>	Returns a defectID

	PQ n qual	proj_find_defects( projID, ...)	Perforce plugin only n:max rows to return qual:qualification to use on query
	PS	proj_list_fields(projID)	
Server	SD	dt_get_server_date(dtID)	
	SF	dt_free(dtID)	Unsets dtID
	SL	dt_list_projects(dtID)	
	SP project	dt_get_project(dtID, project)	Returns a projID
	SV	dt_get_server_version(dtI D)	
	SW	dt_get_server_warnings(dt ID)	