Integrating Build Forge with ClearQuest

Using the Basic ClearQuest/BuildForge Integration

Kristofer A. Duer

January 26, 2009

INTRODUCTION 3

BASIC CLEARQUEST INTEGRATION 4

CLEARQUEST REQUIREMENTS 4
BuildTracking 4
DeploymentTracking 4
BUILD FORGE REQUIREMENTS 4
CQ_DBNAME 4
CQ_DBSET 4
CQ_USER 5
CQ_PASSWORD 5
CQ_RELEASE_NAME 5
CQ_RECORD_ID 5
HIDE_PASSWORD 5
BFCQDEBUG 5
HOW IT WORKS 5
Instantiation 5
Completion 6

STEPS TO INTEGRATE EXAMPLES 8

ClearQuest Steps 8
Build Forge Steps 10
Verification Steps 11

BFCQBUILD.PL 14

BREAKDOWN 14
DIFFERENT FUNCTIONS 15
Main Body 15
ValidateInfo() 16
RecordExists() 18
CheckSubmit() 18
ValidateNextState() 19
CheckReleaseRecord() 20
SubmitBuild() 20
UpdateBuild() 21

CUSTOMIZING THE BUILD FORGE – CLEARQUEST BASE INTEGRATION 22

CUSTOMIZING BFCQBUILD.PL 22
Getting the Tag Field When Using .retag In A Build 22
CUSTOMIZING THE CLEARQUEST BUILD RECORD 24
HOW IT ALL FITS TOGETHER 25

APPENDIX A – BFCQBUILDPL. SCRIPT 28

APPENDIX B - MODIFIED BFCQBUILD.PL 35

Introduction

This white paper introduces the Build Forge with ClearQuest integration capabilities for you to leverage. After reading this white paper you will be familiar with how Build Forge integrates with ClearQuest. This paper will also show a couple of techniques to customize the integration in your environment. This customization will allow you to take advantage of more functionality in both Build Forge and ClearQuest.

This integration grants Build Forge creating and updating of build records inside of ClearQuest. This can be a powerful tool that can help your organization keep accurate records of builds, what went into them and the results of these builds in a ClearQuest format.

The integration is available for version 7.0 and better for both Build Forge and ClearQuest.

This paper does not discuss the ClearQuest Adaptors available. The main goal is to illustrate how Build Forge uses the base ClearQuest integration to give end-users a better idea of how and when to employ it in their environments.

Basic ClearQuest Integration

The basic Build Forge/ClearQuest integration involves only these two products. You can use this integration to capture basic information about the build.

The integration looks for specific environment variables in the project's environment to initiate the script. When Build Forge sees these variables it will then use the ClearQuest API that is called from the Perl script **bfcqbuild.pl** to manage the integration.

This script is located in:

- Microsoft Windows: %BF_HOME%\integration
- UNIX and Linux: $BF_HOME/Platform/integration

This script then connects to ClearQuest and enters a new build record with the build information added to it throughout the course of the build. At the end the record is flipped to retired and the step's results and log web link are posted into the record.

ClearQuest Requirements

The ClearQuest database will require packages to be applied to the database being used for the integration – **BuildTracking** and **DeploymentTracking**. The ClearQuest client on the machine where the engine is installed will require the connection set to the ClearQuest database being used for the integration. The code for the integration is a combination of some set up and logic within the Build Forge engine as well as the cqbuild.pl script in the integration directory. <u>This integration only works with ClearQuest 7.0 or higher.</u>

<u>BuildTracking</u>

This package contains the record type **BTBuild** that is used for capturing the base ClearQuest integration information.

<u>DeploymentTracking</u>

This package contains the **DTRelease** record type used to hold the releases that have current BTBuild records. The release is captured in the project environment variable <u>CQ_RELEASE_NAME</u>.

Build Forge Requirements

There are a few environment variables Build Forge requires to be available in the project's environment group. These environment variables prompt a special routine within the engine to create, update, and add data to a ClearQuest record.

CQ_DBNAME

This is the name of the ClearQuest database where the packages BuildTracking and DeploymentTracking are applied.

CQ_DBSET

This is the database set, or connection string to use. If this is not used then the ClearQuest API will use the default database set with the version string of the ClearQuest version, for example 7.0.0. This can be omitted if you are using either the default connection set for Build Forge, or there is only one connection set specified for ClearQuest.

CQ_USER

This is the name of the ClearQuest user who has access to this database for record creation and updates.

CQ_PASSWORD

This is the password of the ClearQuest user specified for the CQ_USER environment variable. This variable is typically set to assign hidden, suppress display to prevent any Build Forge user from having access to the variable during job starts.

CQ_RELEASE_NAME

This specifies a release to be captured under the Release field on the BTBuild record and a release is captured in the DTRelease record type.

CQ_RECORD_ID

This specifies a Build Record ID. This will not typically be used in the project environment. If this variable does not exist the script will then create the Build Record. If it does exist then the build will add the step results to the current record. This variable is created during a build run by the system to deal with restarts and step log processing.

HIDE_PASSWORD

The ClearQuest password for the CQ_PASSWORD variable will be hidden by the script it this option is enabled when running in ClearQuest debug mode. This is a security measure and should always be employed in the project environment. The value can be anything as the script only checks to see if the variable exists.

BFCQDEBUG

This enables debugging within the cqbuild.pl script. The debugging information is for the Build Forge engine only. The data is not sent to the build log.

How it Works

The build for the Build Forge-ClearQuest integration passes through these three main phases:

1) Instantiation
2) Step Update
3) Completion
4) Retirement

These phases are responsible for creating, modifying, and recording build information into a BTBuild record type. In addition to this the release specified will also be created for the DTRelease record type.

An adapter can influence whether or not a build is started, or cancelled. In the same way an adapter can influence whether a build record is created or modified through the use of adapter links. A build and corresponding ClearQuest build record will not be created in the event of a failed adapter step while using an adapter link. This allows for source adaptors to control when the build fires, keeping unneeded builds out of the Build Forge database as well as keeping empty build records out of the ClearQuest database.

Instantiation

This phase is entered in one of two ways:

1) A new build
2) A restart

A new build will create a new build record, and assign the id to the CQ_RECORD_ID variable. This variable will be available in the build environment and can be used by notifications, or other steps within this project.

A restart of a build takes advantage of the existing CQ_RECORD_ID variable to reference the currently available build record and add the additional step log runs to it. In each case the instantiation phase will modify the record state to "Submitted" and the Start Time field will be updated to reflect the current date and time of the engine machine. The field values are dynamically produced based on the build and project CQ_ variables. These variables are created internally and not run on an agent machine. As a result the build variables and Build Forge specific variables – BF_... – are not available to this script.

Breakdown of initial field values:

FIELD	DEFAULT VALUE	DESCRIPTION
id	Next available ClearQuest record id number	This field is auto incremented and controlled internally by ClearQuest.
State	Submitted	A new or restarted build will set the State field to "Submitted"
Build Start Time	Current date and time of the engine machine	This value is updated with the default value for any new or restarted build
ReleaseName	Value in the CQ_RELEASE_NAME project environment variable	The release name is used to organize builds for different releases within the ClearQuest database.
Build ID	The BF_TAG variable upon project start	This value is static and cannot be adjusted dynamically. A .retag will change the build tag within Build Forge; however, ClearQuest will use the initial tag for the project.
Build Web URL	The web link for the build referenced in this build record	The URL is built using the system setting or the hostname of the engine machine. This item is then plugged into the link in the following format: http://<server>/ /fullcontrol/index.php?mod=jobs&action=edit&bf_id=<Build ID in BF> If you are using a non-default port make sure the Console URL option in Admin > System has the server:port format for this link to work properly.
Build Log	- Build Step Summary -	The default heading is entered into this field, and step results are added below.

Completion

Completion performs the final tasks of the build for the ClearQuest build record. In this phase the build record captures whether the build passed or failed. The step results are posted to the Build Log tab. This phase can have one of three outcomes:

1) Completed
    a. This result represents a build which passed as well as a build which passes with warnings.
2) Failed
    a. This result represents a build which failed.

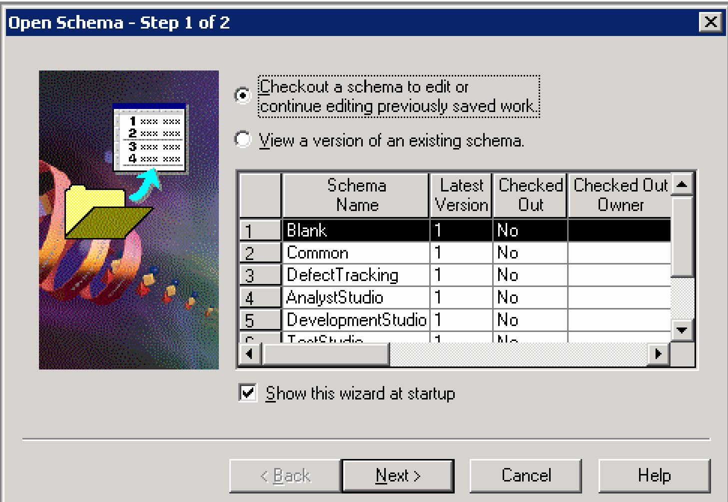
3) Retired
   a. This result occurs when a build is purged from the Build Forge system.

Steps to Integrate Examples

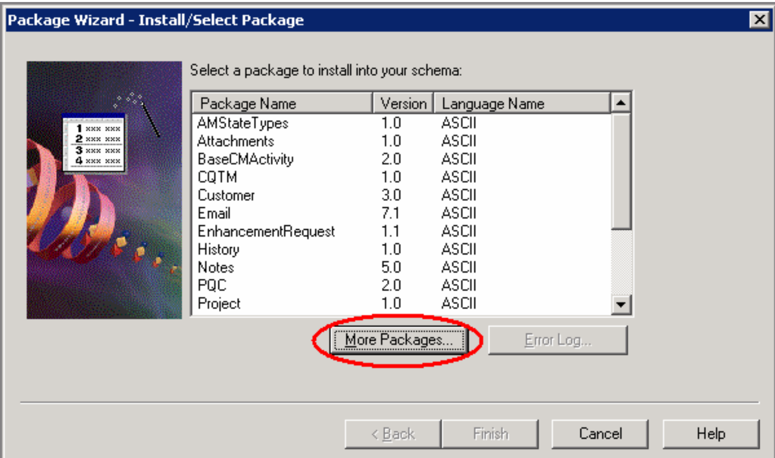
This process runs through the steps needed to create a Build Forge-ClearQuest integrated environment. We will start with ClearQuest requirements first. Next we will look at what is needed on the Build Forge side. Finally we will put it all together in a verification step to ensure the integration is running and the ClearQuest record is created.

ClearQuest Steps

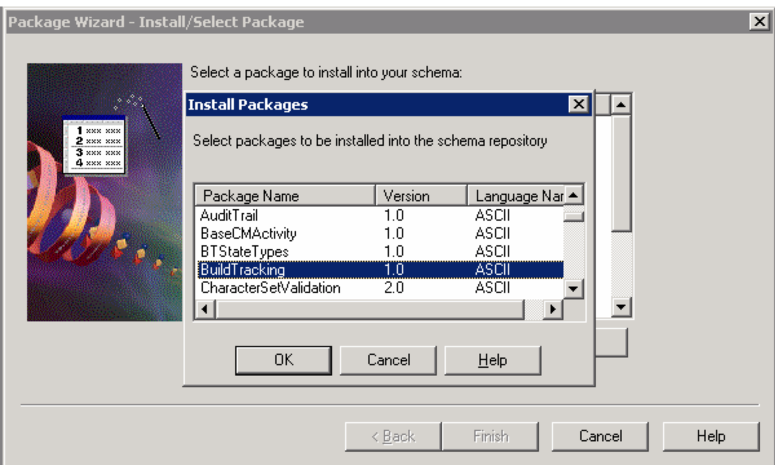
1) Modify a schema currently being used, or a new one to be associated with the Build Forge-ClearQuest integration database. Open the IBM ClearQuest Design Tool and select a schema to modify.

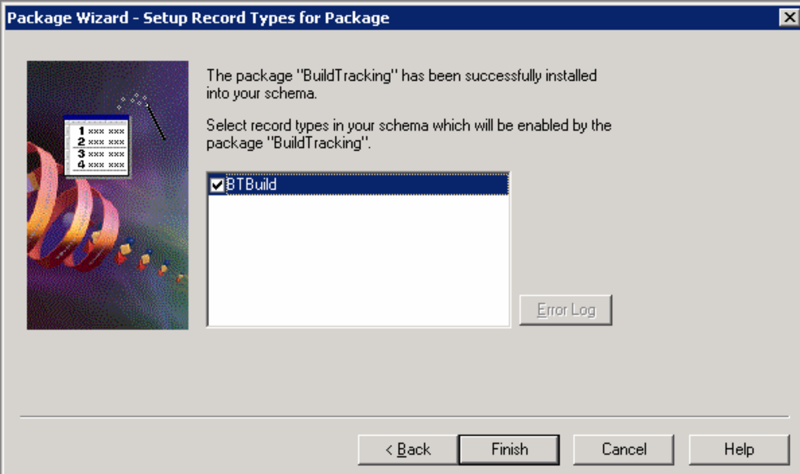


2) Open the Package Wizard and select the "More Packages" button.

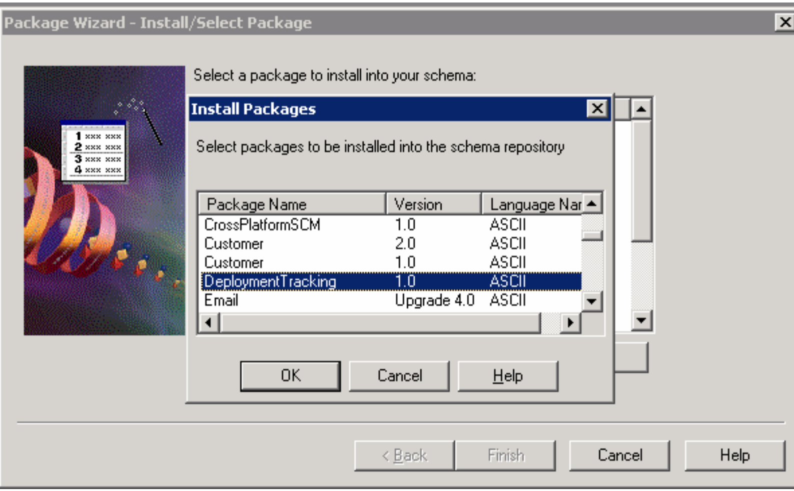


3) Select the BuildTracking package and apply it to this schema

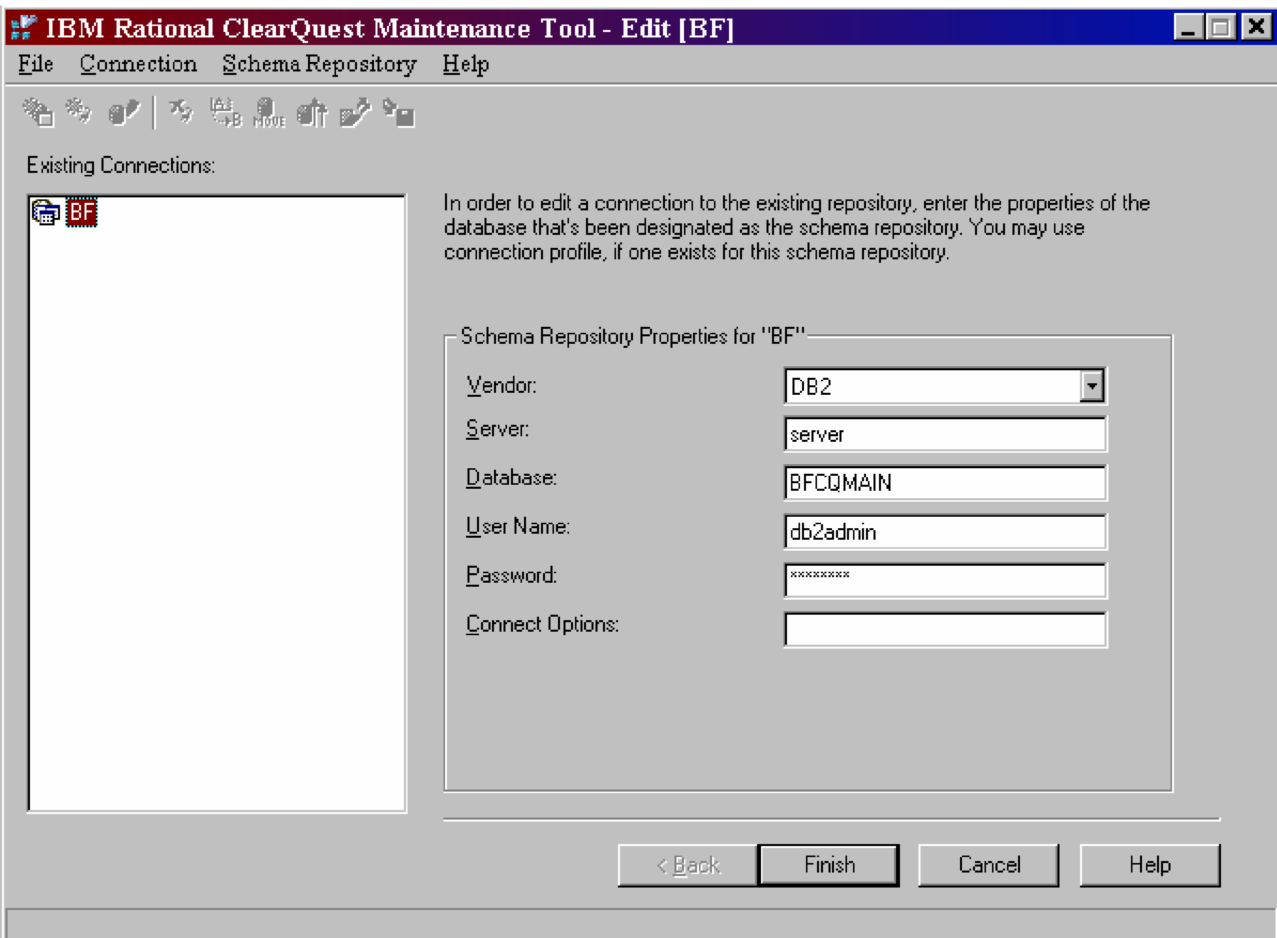




4) Perform the same actions to add the DeploymentTracking package to the same schema just modified.



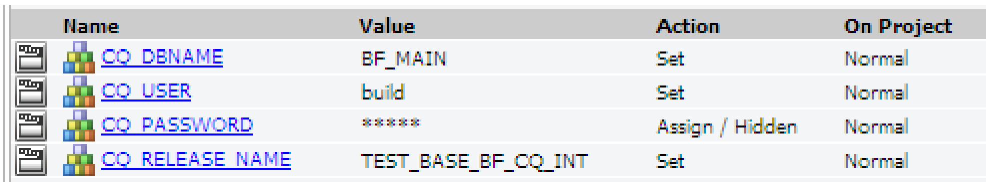
5) Add the connection to ClearQuest for this database if it does not already exist using the ClearQuest Maintenance tool.



You now have a ClearQuest database with the appropriate packages needed for the base integration. The next steps are implementing the ClearQuest project level environment variables in Build Forge.

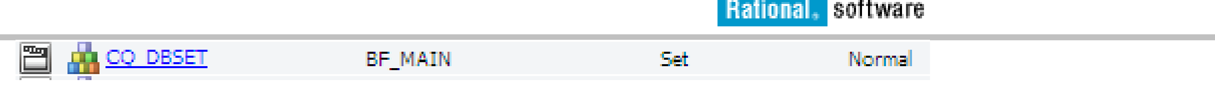
Build Forge Steps

1) Add as a minimum these four environment variables to an environment group:



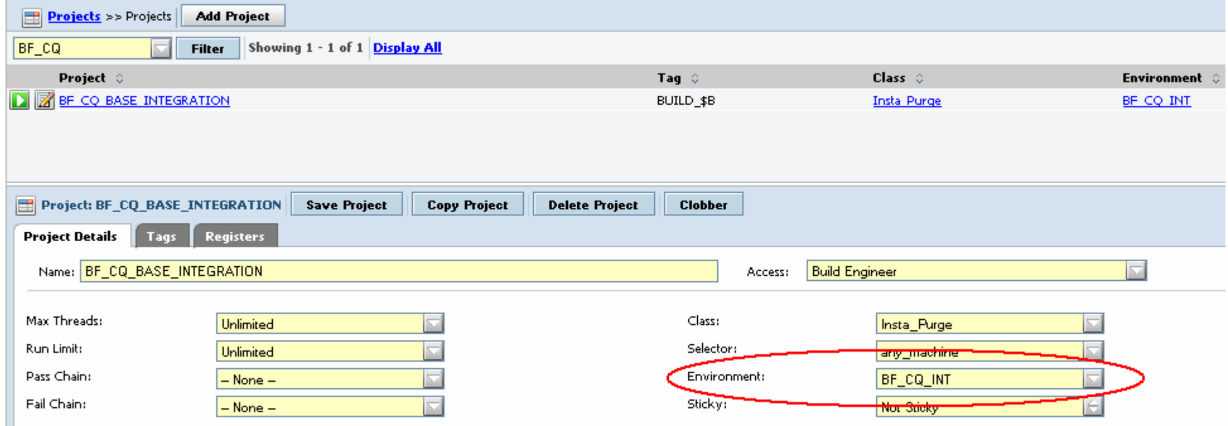
Name	Value	Action	On Project
CQ_DBNAME	BF_MAIN	Set	Normal
CQ_USER	build	Set	Normal
CQ_PASSWORD	*****	Assign / Hidden	Normal
CQ_RELEASE_NAME	TEST_BASE_BF_CQ_INT	Set	Normal

a. If you are using either multiple database connections, or the connection for the Build Forge database is not the ClearQuest default then specify an additional variable – CQ_DBSET.

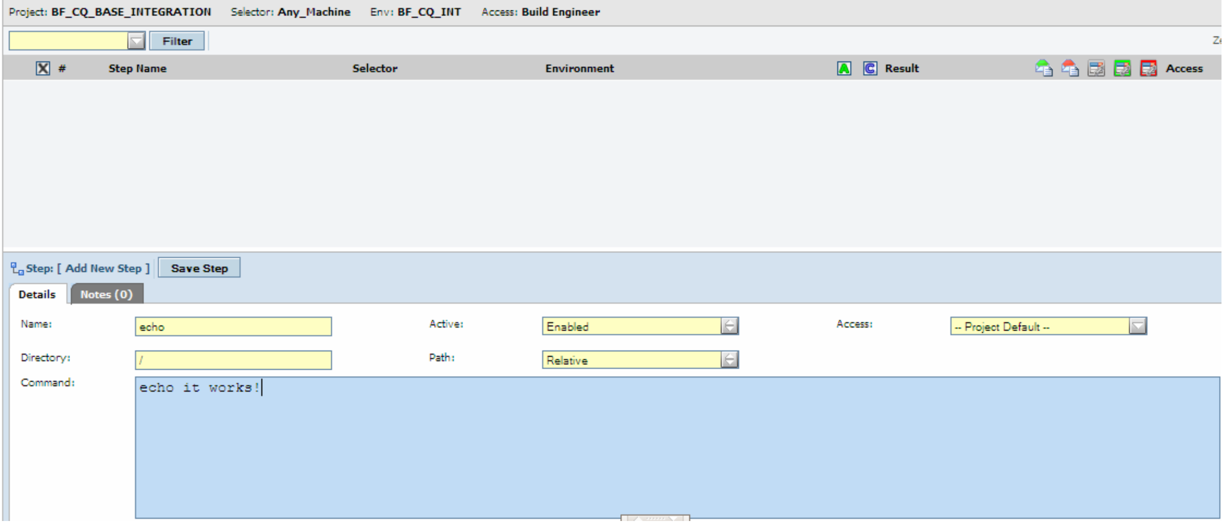


b. Enable HIDE_PASSWORD for security purposes

2) Create a new test project and add this environment to it



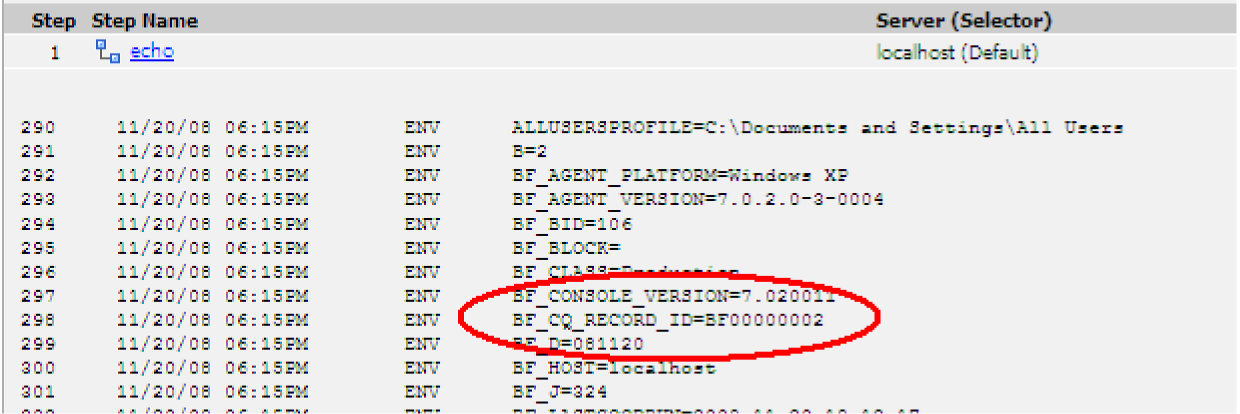
3) Add a sample step – this initial run is only a test to ensure the integration is working



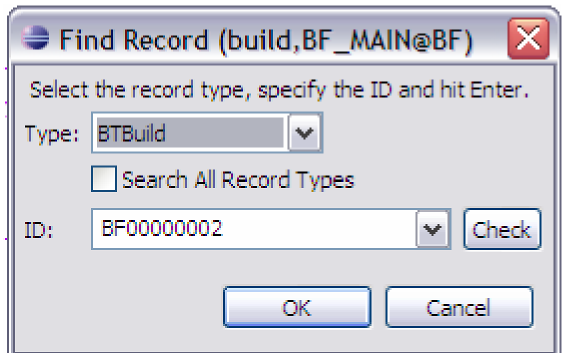
Verification Steps

The final phase in the setup is verification. The build for the Build Forge-ClearQuest integration passes through these two main phases:

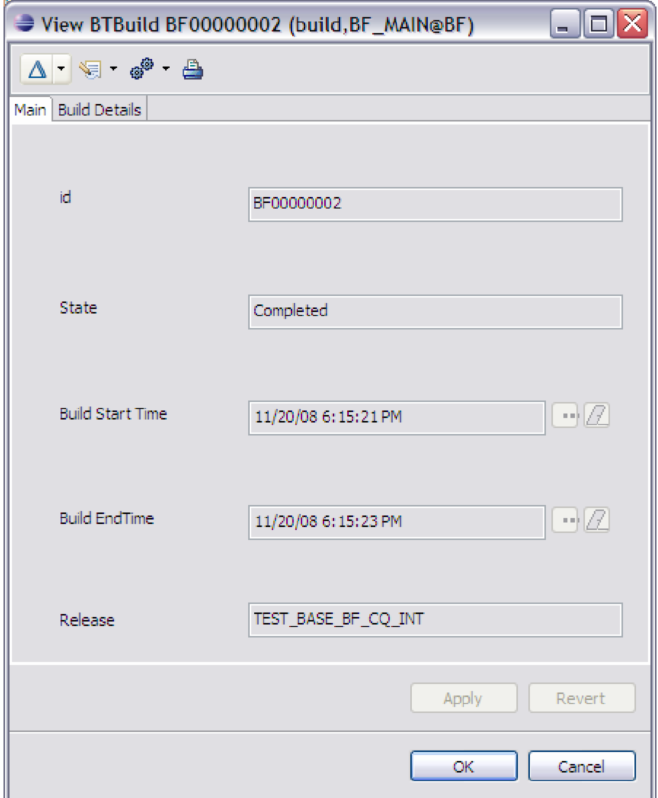
1) Run the build – The step log has a new variable BF_CQ_RECORD which contains the ClearQuest record ID which was created and tied to this build.

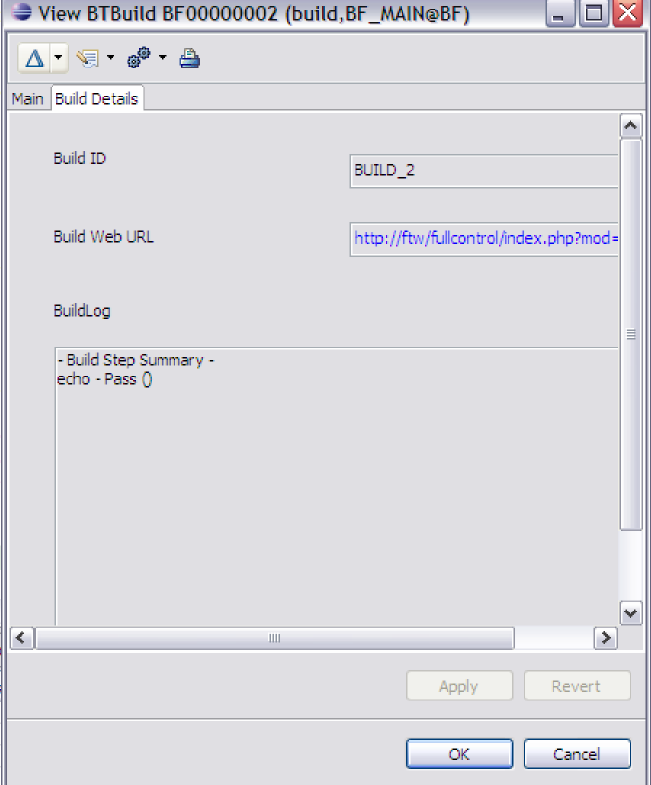


2) Search for the ClearQuest Record



3) Review the record for the various fields which are filled in





bfcqbuild.pl

The bfcqbuild.pl script is executed at various times through out the build lifecycle. This is the entry point given for the integration between Build Forge and ClearQuest. The script is filled with various functions and methods which all fit together to create and update ClearQuest build records. The script is handed a task in the form of a command line argument which it then uses to determine what should be done for this particular task.

Breakdown

The ClearQuest build record has four valid states. Each state can only transition into a particular next state. Below is a list of the valid states and the transitions each state can travel through.

1) Submitted
    a. Completed
    b. Failed
    c. Submit
    d. Retire
2) Completed
    a. Retire
3) Failed
    a. Submit
    b. Completed
    c. Retire
4) Retired

The list shows us only failed builds can be restarted as the script stands now. A passed build with or without warnings will be considered completed. As a result they cannot be re-run after completion. The only transition these builds have left is upon the purge process of the build record in Build Forge when the record transitions into Retired. A failed build can be resubmitted, or restarted to get a completed, or pass status. These transitions are enforced inside of the cqbuild.pl script, not in the ClearQuest record. This allows for changing the behavior of these different states should the end user so desire. For example the Completed state can be modified in the ValidateNextState() function to allow for restarts.

There are four main tasks, or command line arguments, a build will send to this script:

1) Submit
    a. This request occurs upon a new build instantiation. The first step of a project will prompt the creation of a ClearQuest build record. Adapter link steps occur prior to the first step which results in the ClearQuest build record getting created only upon a pass of the adapter link step. A restart will also be entered as a Submit request. The only difference is a build record will exist forcing a different path down the logic of the script.
2) Complete
    a. This captures the step result information once the project completes and places it into the Build Details > BuildLog box in the record. In addition it sets the state of the build to Completed.
3) Failure
    a. This captures the step result information once the project completes and places it into the Build Details > BuildLog box in the record. In addition it sets the state of the build to Failed. A project which passes with warnings will be considered Completed.
4) Retire
    a. This request is made upon a purge of the build. This will set the record state to Retired.

Different Functions

Main Body

The main body of the script controls the program flow. This main body is responsible for parsing the argument and starting the right chain reaction of methods for arriving at the desired result. This piece starts off by grabbing the request from the engine and filling the $ReqState variable with this value. This will be the entry point into the rest of the logic. The script then takes this variable and submits it to the ValidateInfo() function. This function will tell us if the request is a valid request. This function call represents the first time the script will attempt to connect to ClearQuest through the ClearQuest API.

The next step assuming the request is valid is the check to see if the record exists using the RecordExists() function call. This call will return the state of the ClearQuest record used by the build. If this record does not exists then ClearQuest will return undef, and subsequently the $CurState variable will be undef. This is an important piece which will allow us to go through the rest of the flow whether this is a new build or a restart.

The next phase is to evaluate the return we received from RecordExists(). If the variable $CurState is undef then this is a new build. The logic flow will travel to the CheckSubmit() function which will fill the $ReqState variable with "Submit". If this is a current build, i.e. a restart, then the ValidateNextState() function will be called. If the next state allows for a resubmit\restart then $ReqState will be filled with "Resubmit". Finally if the build does not have the next valid state according to the current record state then $ReqState will be filled with "CantChange". The $ReqState variable now holds the key for the rest of the script and the logic path it will follow.

The next phase checks to see if $ReqState is "CantChange". If it is then the build fails immediately. If it is "Submit" CheckReleaseRecord() is called followed immediately by SubmitBuild(). The CheckReleaseRecord() function looks for a DTRelease record for the release specified in the CQ_RELEASE_NAME environment variable. If it does not exist one is created and the ClearQuest build record is linked back to the release record. The SubmitBuild() function creates the new build record for the release. There are four environment variables used by this function sent to it by the engine: TAG, CQ_RELEASE_NAME, BUILD_URL, BUILD_LOG_TEXT. If this is a restart or a build completion the UpdateBuild() function is called. This function fills in any missing data upon build completion such as the step results and sets the final state of the record based on the final status of the build – Completed, Failed or Retired.

```
# Scope of our application
{
	my $ReqState = shift || "Submit";

	my $CQSession;
	my $CurState = "";
	eval { $CQSession = ValidateInfo($ReqState); };
	Error("Could not connect to ClearQuest - Invalid Initialization Data",$@) if($@);

	# See if this is a new submission, or a resubmit.

	# Get the current state if we have one...

	eval {$CurState = RecordExists($CQSession); };
```

```
        Debug("Curstate Is: $CurState");

        if($ReqState eq "Submit" && !$CurState) {
                $ReqState = CheckSubmit($CurState);
                Debug("Reqstate Now: $ReqState");
        }
        else {
                $ReqState = ValidateNextState($CurState,$ReqState);
                Debug("ReqState Now: $ReqState");
        }


        if($ReqState eq "CantChange") {
                Error("Cannot change build record.","The build record is in a Retired state.");
        }

        # Do our action.
        if($ReqState eq "Submit") {
                eval {CheckReleaseRecord($CQSession);};
                Error("Could not create Release Record in ClearQuest",$@) if($@);
                eval {SubmitBuild($CQSession);};
                Error("Could not submit build record to ClearQuest",$@) if($@);
        }
        else {
                eval {UpdateBuild($CQSession,$ReqState);};
                Error("Could not update build record in ClearQuest",$@) if($@);
        }

        # Drop our session, don't care if we error here, as we're leaving.
        eval { CQSession::Unbuild($CQSession); };

        exit 0;
} # Scope
```

ValidateInfo()

ValidateInfo() takes one argument - $ForState. This argument is given to the script directly from the Build Forge engine and can be one of these values – Submit, Failure, Complete, Retire. A restart of a failed build or new build will use Submit, a completed build which passes or has warnings will use Complete, a build which fails will use Failed, and a purge event will use Retire.

All of the states require certain environment variables to be present. These environment variables are made by the Build Forge engine and passed down to the script.

1) All states
    a. CQ_DBNAME
        i. The name of the Build Forge database given to ClearQuest
    b. CQ_USER
        i. The username who has access to the CQ_DBNAME database
2) Submit
    a. CQ_RELEASE_NAME

i. The DTRelease record's Release Name which all build records associated with this Build Forge environment group will use

b. TAG
   i. Internal ClearQuest integration only initial build tag of the project. The build record will only reflect this initial value – any .retags will not be reflected in the build record
c. BUILD_URL
   i. The URL to the Build Forge management console for this particular build
d. BUILD_LOG_TEXT
   i. This variable holds the step results information for the build

3) Failure
   a. CQ_RECORD_ID
      i. The ClearQuest build record ID. This variable is created initially by the engine and then looked for by the engine upon restart, and build completion. This value should never be modified via .bset.
   b. BUILD_LOG_TEXT
4) Complete
   a. CQ_RECORD_ID
   b. BUILD_LOG_TEXT
5) Retire
   a. CQ_RECORD_ID

If any of these environment variables are missing the script will die, and kill the project run. The function uses the ClearQuest API call $CQSession->UserLogon() to get a logon token to operate on the ClearQuest build records.

```
sub ValidateInfo {
	my ($ForState) = @_;
	my @RequiredInfo = ("CQ_DBNAME","CQ_USER");

	# Check our entry data...
	if($ForState =~/Submit/i) {
		push @RequiredInfo,("CQ_RELEASE_NAME","TAG","BUILD_URL","BUILD_LOG_TEXT");
	}
	elsif($ForState =~ /Failure/i) {
		push @RequiredInfo,("CQ_RECORD_ID","BUILD_LOG_TEXT");
	}
	elsif($ForState =~ /Complete/i) {
		push @RequiredInfo,("CQ_RECORD_ID","BUILD_LOG_TEXT");
	}
	elsif($ForState =~ /Retire/i) {
		push @RequiredInfo,("CQ_RECORD_ID");
	}
	foreach my $Required (@RequiredInfo) {
		unless($ENV{$Required}) {
			die "The required environment parameter [$Required] for operation [$ForState]
was not set";
		}
	}

	# Create a connection to ClearQuest.
	my $CQSession = CQPerlExt::CQSession_Build();
	my $CQ_Password = $ENV{'CQ_PASSWORD'} || "";
```

```
        my $DB_Set = $ENV{'CQ_DBSET'} || "";
        Debug("Connecting To ClearQuest using the following Credentials:");
        Debug("USER:$ENV{'CQ_USER'}") if($ENV{'CQ_USER'});
        Debug("PASSWORD:$CQ_Password") if($CQ_Password && !$ENV{'HIDE_PASSWORD'});
        Debug("DBNAME:$ENV{'CQ_DBNAME'}") if($ENV{'CQ_DBNAME'});
        Debug("DBSET:$DB_Set") if($DB_Set);
        $CQSession->UserLogon($ENV{'CQ_USER'},$CQ_Password,$ENV{'CQ_DBNAME'},$DB_Set);
        return $CQSession;
} # ValidateInfo
```

RecordExists()

The RecordExists() function checks to see if a build record exists. If this check fails the $StateRec variable will be undef and will return this to the caller. If successful the record state is returned to the caller. This state can be one of the following:

1) Submitted
   a. Represents a new build or a restart of a failed build
2) Completed
   a. Represents a passed or passed with warnings build
3) Failed
   a. Represents a failed build
4) Retired
   a. Represents a purged build

This function will use the ClearQuest API calls $CQSession->GetEntity(),$Entity->GetFieldValue and $StateRec->GetValue.

```
sub RecordExists {
        my ($CQSession) = @_;
        my $RecId = $ENV{'CQ_RECORD_ID'};
        my $Entity = undef;
        if($RecId) {
                $Entity = $CQSession->GetEntity("btbuild",$RecId);
        }
        my $StateRec = $Entity->GetFieldValue("State");
        return $StateRec->GetValue;
} # RecordExists
```

CheckSubmit()

Currently the only purpose of this function is to return "Submit" for a new build record. In the current release of the script the check to get to this function looks for both the $ReqState variable being in "Submit" and $CurState being undef. As a result $State will always be undef and will return "Submit" for new builds.

```
sub CheckSubmit {
        my($State) = @_;
        if($State =~ /Retired/i) {
```

```
                return "CantChange";
        }
        elsif($State =~ /(Completed|Failed)/i) {
                return "Resubmit";
        }
        elsif($State =~ /Submitted/i) {
                return "NoChange";
        }
        return "Submit";
} # CheckSubmit
```

ValidateNextState()

The ValidateNextState() function is used to enforce policy inside the script. This is where restarts of completed passed builds are prevented. The function receives two arguments - $CurState and $ReqState. $CurState represents the current state the build record is in, while $ReqState represents the state you would like to transition to, based on the build in Build Forge be it a start, restart or purge. The flow goes through the four states and fills the array @ValidStates with the valid next states for this particular build. The final piece fails out if the desired state to move to is not allowed by the current state. For example a completed passed build will have a current state of Completed which allows for one transition state – Retired. A restart will be prevented due to this function.

```
sub ValidateNextState {
        my($CurState,$ReqState) = @_;
        my @ValidStates = ("Submit","Resubmit","Retire","Complete","Failure");
        my $Valid = 0;
        foreach my $State (@ValidStates) {
                if($ReqState =~ /^$State$/i) {
                        $Valid = 1;
                }
        }
        Error("Invalid state requested","State [$ReqState] is not recognized as a valid state for a
ClearQuest Build record. Valid states are [".join("],[",@ValidStates)."].") unless($Valid);
        @ValidStates = ();
        $Valid = 0;
        if($CurState =~ /Failed/i) {
                @ValidStates = ("Resubmit","Retire","Submit");
                if($ReqState =~ /Submit/i) {
                        $ReqState = "Resubmit";
                }
        }
        elsif($CurState =~ /Completed/i) {
                @ValidStates = ("Retire");
        }
        elsif($CurState =~ /Retired/i) {
                @ValidStates = ();
        }
        elsif($CurState =~ /Submitted/i) {
                @ValidStates = ("Complete","Failure","Submit","NoChange");
                if($ReqState =~ /Submit/i) {
                        $ReqState = "NoChange";
```

```
		}
	}
	foreach my $State (@ValidStates) {
		if($ReqState =~ /$State/i) {
			$Valid = 1;
		}
	}
	Error("Invalid state requested","Current state is [$CurState], requested next state as
[$ReqState], valid nextstate(s) are [".join("],[",@ValidStates)."].") unless($Valid);

	return $ReqState;
} # ValidateNextState
```

CheckReleaseRecord()

The CheckReleaseRecord() function either returns the release record object if it exists, or creates a new one. This function initially uses the ClearQuest API call $CQSession->GetEntity() to get the release record. If it does not exist then the variable $ReleaseRec will not be a reference and the first return will not process. It will then take in the CQ_RELEASE_NAME variable and create a new release record inside of ClearQuest. This release record is a container for all of the build records for a particular release.

```
sub CheckReleaseRecord {
	my($CQSession) = @_;
	my $ReleaseRec = undef;
	eval {$ReleaseRec = $CQSession->GetEntity("dtrelease",$ENV{'CQ_RELEASE_NAME'});};
	return if(ref($ReleaseRec) eq "CQEntity");
	# Need to create the release record...
	my $CreateReleaseRec = $CQSession->BuildEntity("dtrelease");
	$CreateReleaseRec->SetFieldValue("release_name",$ENV{'CQ_RELEASE_NAME'});
	$CreateReleaseRec->Validate();
	my $CommitResult = $CreateReleaseRec->Commit();
	if($CommitResult eq "") {
		Log("Release Record Created:\n$ENV{'CQ_RELEASE_NAME'}");
	}
	else {
		Error("Could not create Release Record.",$CommitResult);
	}
}
```

SubmitBuild()

The SubmitBuild() function is only called if the build record does not already exist. It will run through and create the new build record for the new Build Forge build.

```
sub SubmitBuild {
	my($CQSession) = @_;
	my $BuildRec = $CQSession->BuildEntity("btbuild");
	$BuildRec->SetFieldValue("start_datetime",genTimeStamp());
	$BuildRec->SetFieldValue("build_system_id",$ENV{'TAG'});
```

```
        $BuildRec->SetFieldValue("releasename",$ENV{'CQ_RELEASE_NAME'});
        $BuildRec->SetFieldValue("build_system_url",$ENV{'BUILD_URL'});
        $BuildRec->SetFieldValue("buildlog",$ENV{'BUILD_LOG_TEXT'});
        my $RecordId = $BuildRec->GetDisplayName();

        # Validate and Commit.
        $BuildRec->Validate();
        $BuildRec->Commit();
        Log("Build Record Created:\n$RecordId");
} # SubmitBuild
```

UpdateBuild()

The UpdateBuild() function is responsible for handling capturing the completed build's details such as the step results, and restart results. This is the function which can be modified to add different information to the build record. We go over some basic customization possibilities in the final section Customizing the Build Forge – ClearQuest Base Integration.

```
# UpdateBuild
#
# Update an already existing Build Record.
sub UpdateBuild {
        my($CQSession,$Action) = @_;
        my $BuildRec = $CQSession->GetEntity("btbuild",$ENV{'CQ_RECORD_ID'});
        $CQSession->EditEntity($BuildRec,"modify");
        $BuildRec->SetFieldValue("build_system_url",$ENV{'BUILD_URL'})
if(exists($ENV{'BUILD_URL'}));
        $BuildRec->SetFieldValue("buildlog",$ENV{'BUILD_LOG_TEXT'})
if(exists($ENV{'BUILD_LOG_TEXT'}));
        if($Action =~ /(NoChange|Resubmit)/i) {
                $BuildRec->SetFieldValue("start_datetime",genTimeStamp());
        }
        elsif($Action !~ /Retire/i) {
                $BuildRec->SetFieldValue("end_datetime",genTimeStamp());
        }
        $BuildRec->Validate();
        $BuildRec->Commit();

        if($Action=~/(Resubmit|Failure|Complete|Retire)/i) {
                $CQSession->EditEntity($BuildRec,$Action);
                $BuildRec->Validate();
                $BuildRec->Commit();
        }
        Log("Build Record Updated:\n$ENV{'CQ_RECORD_ID'}");
} # UpdateBuild
```

Customizing the Build Forge – ClearQuest Base Integration

There are a couple of customization entry points provided within the script. The engine is hard coded to send across only a few ClearQuest specific environment variables. These will be the only variables available for the life of the script regardless of what environment variables are available to the build.

1) CQ_DBNAME
2) CQ_USER
3) CQ_RELEASE_NAME
4) TAG
5) CQ_DBSET
6) CQ_PASSWORD
7) BUILD_URL
8) BUILD_LOG_TEXT
9) CQ_RECORD_ID

Be sure to back up the current bfcqbuild.pl script before any modifications are made.

Customizing bfcqbuild.pl

The BUILD_URL variable contains at the end of it the build id of the associated build. This build id can then be used to gain access to a vast array of data regarding the build through the Build Forge API. The format of the URL is:
http://<server>/fullcontrol/index.php?mod=jobs&action=edit&bf_id=<build id>
For example:
http://server/fullcontrol/index.php?mod=jobs&action=edit&bf_id=130

The following regex can be used to get the build id from the URL:

```
$ENV{'BUILD_URL'} =~ /.*?bf_id\=(\d*)/;
my $bid = $1;
```

This variable holds the key to getting other build information from Build Forge. The best place to modify the bfcqbuild.pl script is within the UpdateBuild() function. This is the last function run against any build.

Below are a couple of examples of getting different data and updating the current ClearQuest record type with information.

Getting the Tag Field When Using .retag In A Build

The .retag command changes the tag dynamically, however the variable $TAG will not contain this new value. The build id can be used to retrieve this value using the Build Forge API. For example in the UpdateBuild() function we have:

```
# UpdateBuild
#
# Update an already existing Build Record.
sub UpdateBuild {
	my($CQSession,$Action) = @_;
...
```

Add a new function call to UpdateBuild():

```
my $tag = BF_tag();
```

This function will be responsible for getting the Build Forge build object for the build. This build object will in turn give a window into a great deal of information.

```
sub BF_tag {

        my $conn = bf_conn();
        $ENV{'BUILD_URL'} =~ /.*?bf_id\=(\d*)/;
        my $bid = $1;
        my $build = BuildForge::Services::DBO::Build->findById($conn, $bid);
        my $build_tag = $build->getTag();
        my $build_result = $build->getResult();
        return $build_tag;

}

sub bf_conn {

        #Update with the @INC path for the BuildForge Perl API client for your system
        use lib "C:/Perl/site/lib";
        use BuildForge::Services;

        my $hostname = 'localhost';
        my $user = 'build';
        my $pass = 'build';
        my $conn = new BuildForge::Services::Connection($hostname);
        $conn->authUser($user, $pass);
        return $conn;

}
```

We are able to get the end result of a build at the end as well. The script is run upon project completion allowing for the latest result, such as warning, to be retrieved. We will use these two items during the final phase of the customization to add the actual build tag changed from a .retag, and introduce a new state – Warning – which will allow restarts.

```
sub bf_result {
        $ENV{'BUILD_URL'} =~ /.*?bf_id\=(\d*)/;
        my $conn = bf_conn();
        my $bid = $1;
        my $build = BuildForge::Services::DBO::Build->findById($conn, $bid);
        my $build_result = $build->getResult();
        return $build_result;
}
```

Items to be aware of during this customization:

1) The Build Forge Perl client will need to be installed
2) The Perl/site/lib directory will needed to be added via use lib pragma
    a. The cqperl utility is used to run the scripts and this typically does not place the local Perl @INC directories in the directories searched for a module. The use BuildForge::Services

piece will fail if cqperl cannot find the module. The BuildForge module folder is placed in the perl/site/lib directory.

3) The ClearQuest record will require modification. The changes you make are yours to support. This shows a possibility only using the built in BTBuild package type.

Customizing the ClearQuest Build Record

We will make two changes for the purposes of this document. The possibilities beyond these are pretty wide open.

1) Make a new field to hold the new build tag.

The current BTBuild record tag field is only writable upon record creation. This will not allow for adjustment of the build tag later in the build process which is what is required to update the new .retag build tag onto the ClearQuest record.

1) Open the ClearQuest Designer
2) Open the schema with the BTBuild package applied to it
3) Add a new field – Build_Tag

- Blank, version 8
  - Schema Properties
  - Record Types
    - BTBuild
      - Fields
      - States and Actions
        - State Transition Matrix
        - Actions
        - Behaviors
      - Forms
      - Record Scripts
    - BTBuildcustom
    - DTApproval
    - DTDeployment
    - UCMUtilityActivity
  - Record Types - Stateless
  - Record Type Families
  - Global Scripts
  - Dynamic List Names

Field Name	Type	Default Value	Permission	Value Changed	Validation
dbid	DBID				
is_active	INT				
id	ID				
State	STATE				
version	INT				
lock_version	INT				
locked_by	INT				
ratl_mastership	REFERENCE				DEFAULT
history	JOURNAL				
is_duplicate	INT				
unduplicate_state	SHORT_STRING				
record_type	RECORDTYPE				
Build_System_URL	SHORT_STRING				
Build_System_ID	SHORT_STRING				
BuildLog	MULTILINE_STRING				
Start_DateTime	DATE_TIME				
End_DateTime	DATE_TIME				
BTBuild	REFERENCE_LIST				DEFAULT
ReleaseName	REFERENCE				DEFAULT
Build_Tag	SHORT_STRING				

This field will hold the new build tag from the

2) Add a new state - Warning

The Completed state is currently a catch all for builds which pass, or pass with warnings. There may be a desire to restart builds with warnings and adding a new state will help with this goal.

1) Add a new Action – Warning

Action Name	Type	Access Control	Initialization
Submit	SUBMIT	All Users	
Import	IMPORT	All Users	
Modify	MODIFY	All Users	
Complete	CHANGE_STATE	All Users	
Retire	CHANGE_STATE	All Users	
Failure	CHANGE_STATE	All Users	
BTBase	BASE	All Users	PERL
ReSubmit	CHANGE_STATE	All Users	
Warning	CHANGE_STATE	All Users	

a.

2) Add a new state – Warning – and set the same transitions as Failed

a.

To \ From	Submitted	Completed	Retired	Failed	Warning
Submitted				ReSubmit	ReSubmit
Completed	Complete				
Retired		Retire		Retire	Retire
Failed	Failure				
Warning					
6					

3) Modify the Completed state to also include Warning

a.

To \ From	Submitted	Completed	Retired	Failed	Warning
Submitted				ReSubmit	ReSubmit
Completed	Complete				
Retired		Retire		Retire	Retire
Failed	Failure				
Warning		Warning			
6					

The final function will only be called when the build is completed. Setting the completed state with the transition ability into warning will allow for the bfcqbuild.pl modifications to transition into warning should the actual result be a warning.

How it all fits together

The additional functions and added ClearQuest BTBuild modifications can now be taken advantage of. Modify the UpdateBuild() and ValidateNextState to make use of the new functions added above.

```
sub UpdateBuild {
        my($CQSession,$Action) = @_;

        my $BuildRec = $CQSession->GetEntity("btbuild",$ENV{'CQ_RECORD_ID'});
        $CQSession->EditEntity($BuildRec,"modify");
        $BuildRec->SetFieldValue("build_system_url",$ENV{'BUILD_URL'})
if(exists($ENV{'BUILD_URL'}));
        $BuildRec->SetFieldValue("buildlog",$ENV{'BUILD_LOG_TEXT'})
if(exists($ENV{'BUILD_LOG_TEXT'}));

        ################################
        #Grab the new Build Tag
        ################################
        my $tag = BF_tag();
        ################################
        #Update the new record field build_tag with the new build tag
        ################################
        $BuildRec->SetFieldValue("build_tag",$tag);

        if($Action =~ /(NoChange|Resubmit)/i) {
                $BuildRec->SetFieldValue("start_datetime",genTimeStamp());
        }
        elsif($Action !~ /Retire/i) {
                $BuildRec->SetFieldValue("end_datetime",genTimeStamp());
        }
```

```
        $BuildRec->Validate();
        $BuildRec->Commit();

        if($Action=~/(Resubmit|Failure|Complete|Retire)/i) {

                ################################
                #Get the build result
                ################################
                my $result = bf_result();
                $CQSession->EditEntity($BuildRec,$Action);
                $BuildRec->Validate();
                $BuildRec->Commit();

                ################################
                #If the result is either WARNING_FILTER or WARNING_FAILED_STEP
                #Update the build record state to Warning
                ################################
                if ($result =~ /warn/i) {
                        my $ret = $CQSession->EditEntity($BuildRec,'Warning');
                        $BuildRec->Validate();
                        $BuildRec->Commit();
                }
        }


        Log("Build Record Updated:\n$ENV{'CQ_RECORD_ID'}");
} # UpdateBuild

sub ValidateNextState {
        my($CurState,$ReqState) = @_;
        my @ValidStates = ("Submit","Resubmit","Retire","Complete","Failure");
        my $Valid = 0;
        foreach my $State (@ValidStates) {
                if($ReqState =~ /^$State$/i) {
                        $Valid = 1;
                }
        }
        Error("Invalid state requested","State [$ReqState] is not recognized as a valid state for a
ClearQuest Build record. Valid states are [".join("],[",@ValidStates)."].") unless($Valid);
        @ValidStates = ();
        $Valid = 0;
        if($CurState =~ /Failed/i) {
                @ValidStates = ("Resubmit","Retire","Submit");
                if($ReqState =~ /Submit/i) {
                        $ReqState = "Resubmit";
                }
        }
################################
#Add a Warning valid state
################################

        if($CurState =~ /Warning/i) {
                @ValidStates = ("Resubmit","Retire");
```

```
        if($ReqState =~ /Submit/i) {
                $ReqState = "Resubmit";
        }
    }
    elsif($CurState =~ /Completed/i) {
        @ValidStates = ("Resubmit","Retire","Submit");
    }
    elsif($CurState =~ /Retired/i) {
        @ValidStates = ();
    }
    elsif($CurState =~ /Submitted/i) {
        @ValidStates = ("Complete","Failure","Submit","NoChange");
        if($ReqState =~ /Submit/i) {
                $ReqState = "NoChange";
        }
    }

    foreach my $State (@ValidStates) {
        if($ReqState =~ /$State/i) {
                $Valid = 1;
        }
    }
    Error("Invalid state requested","Current state is [$CurState], requested next state as
[$ReqState], valid nextstate(s) are [".join("],[",@ValidStates)."].") unless($Valid);

    return $ReqState;
} # ValidateNextState
```

Appendix A – bfcqbuildpl. script

```
#!/usr/bin/cqperl
#
# Copyright (c)2003-2006, BuildForge, Inc. All rights reserved.
# BuildForge, Inc. owns the copyright and other
# intellectual property rights in this software.
#
# Duplication or use of the Software is not permitted
# except as expressly provided in a written agreement
# between your company and BuildForge, Inc.
#
#
# bfcqbuild.pl
#
# Description:
#
# This application is an extension of the ClearQuest API,
# it is used to create and modify Build records inside of
# ClearQuest.
#
#
# Created by: rfuller
# Modification Date: 11:01 AM 12/12/2005
#
# Usage:
# cqperl bfcqbuild.pl <update_state>
#
# The following Parameters are set via the environment.
#
# CQ_DBNAME
#       The ClearQuest Database Instance Name
# CQ_DBSET
#       The Database set or connection string (default "")
# CQ_USER
#       The ClearQuest User
# CQ_PASSWORD
#       Corresponding user password (default "")
# CQ_RELEASE_NAME
#       The Release Name as identified by ClearQuest
# CQ_RECORD_ID
#       The CQ Build Record ID (if updating).
# TAG
#       The Build Tag for the build.
# BUILD_URL
#       A URL pointing to relevant build information
# BUILD_LOG_TEXT
#       An extract of interesting build log information
# HIDE_PASSWORD
#       Do not log the password
# BFCQDEBUG
#       Print out debug info
```

```
# Import Libraries
use CQPerlExt;
use strict;

# We don't want ClearQuest warnings reported in our output stream.
open (STDERR,"> nul");

# Scope of our application
{
	my $ReqState = shift || "Submit";

	my $CQSession;
	my $CurState = "";
	eval { $CQSession = ValidateInfo($ReqState); };
	Error("Could not connect to ClearQuest - Invalid Initialization Data",$@) if($@);

	# See if this is a new submission, or a resubmit.

	# Get the current state if we have one...

	eval {$CurState = RecordExists($CQSession); };
	Debug("Curstate Is: $CurState");

	if($ReqState eq "Submit" && !$CurState) {
		$ReqState = CheckSubmit($CurState);
		Debug("Reqstate Now: $ReqState");
	}
	else {
		$ReqState = ValidateNextState($CurState,$ReqState);
		Debug("ReqState Now: $ReqState");
	}


	if($ReqState eq "CantChange") {
		Error("Cannot change build record.","The build record is in a Retired state.");
	}

	# Do our action.
	if($ReqState eq "Submit") {
		eval {CheckReleaseRecord($CQSession);};
		Error("Could not create Release Record in ClearQuest",$@) if($@);
		eval {SubmitBuild($CQSession);};
		Error("Could not submit build record to ClearQuest",$@) if($@);
	}
	else {
		eval {UpdateBuild($CQSession,$ReqState);};
		Error("Could not update build record in ClearQuest",$@) if($@);
	}

	# Drop our session, don't care if we error here, as we're leaving.
	eval { CQSession::Unbuild($CQSession); };
```

```
        exit 0;
} # Scope

# ValidateNextState
#
# Make sure we're trying to set a valid state.
sub ValidateNextState {
        my($CurState,$ReqState) = @_;
        my @ValidStates = ("Submit","Resubmit","Retire","Complete","Failure");
        my $Valid = 0;
        foreach my $State (@ValidStates) {
                if($ReqState =~ /^$State$/i) {
                        $Valid = 1;
                }
        }
        Error("Invalid state requested","State [$ReqState] is not recognized as a valid state for a
ClearQuest Build record. Valid states are [".join("],[",@ValidStates)."].") unless($Valid);
        @ValidStates = ();
        $Valid = 0;
        if($CurState =~ /Failed/i) {
                @ValidStates = ("Resubmit","Retire","Submit");
                if($ReqState =~ /Submit/i) {
                        $ReqState = "Resubmit";
                }
        }
        elsif($CurState =~ /Completed/i) {
                @ValidStates = ("Retire");
        }
        elsif($CurState =~ /Retired/i) {
                @ValidStates = ();
        }
        elsif($CurState =~ /Submitted/i) {
                @ValidStates = ("Complete","Failure","Submit","NoChange");
                if($ReqState =~ /Submit/i) {
                        $ReqState = "NoChange";
                }
        }
        foreach my $State (@ValidStates) {
                if($ReqState =~ /$State/i) {
                        $Valid = 1;
                }
        }
        Error("Invalid state requested","Current state is [$CurState], requested next state as
[$ReqState], valid nextstate(s) are [".join("],[",@ValidStates)."].") unless($Valid);

        return $ReqState;
} # ValidateNextState

# SubmitBuild
#
# Create a new ClearQuest Build Record.
sub SubmitBuild {
        my($CQSession) = @_;
```

```
        my $BuildRec = $CQSession->BuildEntity("btbuild");
        $BuildRec->SetFieldValue("start_datetime",genTimeStamp());
        $BuildRec->SetFieldValue("build_system_id",$ENV{'TAG'});
        $BuildRec->SetFieldValue("releasename",$ENV{'CQ_RELEASE_NAME'});
        $BuildRec->SetFieldValue("build_system_url",$ENV{'BUILD_URL'});
        $BuildRec->SetFieldValue("buildlog",$ENV{'BUILD_LOG_TEXT'});
        my $RecordId = $BuildRec->GetDisplayName();

        # Validate and Commit.
        $BuildRec->Validate();
        $BuildRec->Commit();
        Log("Build Record Created:\n$RecordId");
} # SubmitBuild

sub CheckReleaseRecord {
        my($CQSession) = @_;
        my $ReleaseRec = undef;
        eval {$ReleaseRec = $CQSession->GetEntity("dtrelease",$ENV{'CQ_RELEASE_NAME'});};
        return if(ref($ReleaseRec) eq "CQEntity");
        # Need to create the release record...
        my $CreateReleaseRec = $CQSession->BuildEntity("dtrelease");
        $CreateReleaseRec->SetFieldValue("release_name",$ENV{'CQ_RELEASE_NAME'});
        $CreateReleaseRec->Validate();
        my $CommitResult = $CreateReleaseRec->Commit();
        if($CommitResult eq "") {
                Log("Release Record Created:\n$ENV{'CQ_RELEASE_NAME'}");
        }
        else {
                Error("Could not create Release Record.",$CommitResult);
        }
}


# UpdateBuild
#
# Update an already existing Build Record.
sub UpdateBuild {
        my($CQSession,$Action) = @_;
        my $BuildRec = $CQSession->GetEntity("btbuild",$ENV{'CQ_RECORD_ID'});
        $CQSession->EditEntity($BuildRec,"modify");
        $BuildRec->SetFieldValue("build_system_url",$ENV{'BUILD_URL'})
if(exists($ENV{'BUILD_URL'}));
        $BuildRec->SetFieldValue("buildlog",$ENV{'BUILD_LOG_TEXT'})
if(exists($ENV{'BUILD_LOG_TEXT'}));
        if($Action =~ /(NoChange|Resubmit)/i) {
                $BuildRec->SetFieldValue("start_datetime",genTimeStamp());
        }
        elsif($Action !~ /Retire/i) {
                $BuildRec->SetFieldValue("end_datetime",genTimeStamp());
        }
        $BuildRec->Validate();
        $BuildRec->Commit();
```

```
        if($Action=~/(Resubmit|Failure|Complete|Retire)/i) {
                $CQSession->EditEntity($BuildRec,$Action);
                $BuildRec->Validate();
                $BuildRec->Commit();
        }
        Log("Build Record Updated:\n$ENV{'CQ_RECORD_ID'}");
} # UpdateBuild

# RecordExists
#
# Check to see if our record exists and if this should be resubmitted, or other.
sub RecordExists {
        my ($CQSession) = @_;
        my $RecId = $ENV{'CQ_RECORD_ID'};
        my $Entity = undef;
        if($RecId) {
                $Entity = $CQSession->GetEntity("btbuild",$RecId);
        }
        my $StateRec = $Entity->GetFieldValue("State");
        return $StateRec->GetValue;
} # RecordExists

# CheckSubmit
#
# Check to see if a Submit can happen or if it's a resubmit...
sub CheckSubmit {
        my($State) = @_;
        if($State =~ /Retired/i) {
                return "CantChange";
        }
        elsif($State =~ /(Completed|Failed)/i) {
                return "Resubmit";
        }
        elsif($State =~ /Submitted/i) {
                return "NoChange";
        }
        return "Submit";
} # CheckSubmit

# ValidateInfo
#
# Make sure we have the correct env for our request.
sub ValidateInfo {
        my ($ForState) = @_;
        my @RequiredInfo = ("CQ_DBNAME","CQ_USER");

        # Check our entry data...
        if($ForState =~/Submit/i) {
                push @RequiredInfo,("CQ_RELEASE_NAME","TAG","BUILD_URL","BUILD_LOG_TEXT");
        }
        elsif($ForState =~ /Failure/i) {
                push @RequiredInfo,("CQ_RECORD_ID","BUILD_LOG_TEXT");
        }
```

```
        elsif($ForState =~ /Complete/i) {
                push @RequiredInfo,("CQ_RECORD_ID","BUILD_LOG_TEXT");
        }
        elsif($ForState =~ /Retire/i) {
                push @RequiredInfo,("CQ_RECORD_ID");
        }
        foreach my $Required (@RequiredInfo) {
                unless($ENV{$Required}) {
                        die "The required environment parameter [$Required] for operation [$ForState]
was not set";
                }
        }

        # Create a connection to ClearQuest.
        my $CQSession = CQPerlExt::CQSession_Build();
        my $CQ_Password = $ENV{'CQ_PASSWORD'} || "";
        my $DB_Set = $ENV{'CQ_DBSET'} || "";
        Debug("Connecting To ClearQuest using the following Credentials:");
        Debug("USER:$ENV{'CQ_USER'}") if($ENV{'CQ_USER'});
        Debug("PASSWORD:$CQ_Password") if($CQ_Password && !$ENV{'HIDE_PASSWORD'});
        Debug("DBNAME:$ENV{'CQ_DBNAME'}") if($ENV{'CQ_DBNAME'});
        Debug("DBSET:$DB_Set") if($DB_Set);
        $CQSession->UserLogon($ENV{'CQ_USER'},$CQ_Password,$ENV{'CQ_DBNAME'},$DB_Set);
        return $CQSession;
} # ValidateInfo

# genTimeStamp
#
# Generate a CQ-friendly timestamp
sub genTimeStamp {
        my @Time=localtime();
        my $Time = sprintf("%d/%d/%d %d:%02d:%02d
%s",($Time[4]+1),$Time[3],($Time[5]+1900),($Time[2] > 12 ? ($Time[2]-12) : ($Time[2] == 0 ? 12
: $Time[2])),$Time[1],$Time[0],($Time[2] > 11 ? "P" : "A"));
        return $Time;
} # genTimeStamp

# Log
#
# Generic Logging
sub Log {
        my($String) = @_;
        print "LOG: $String\n";
} # Log

# Debug
#
# Generic Debugging
sub Debug {
        my($String) = @_;
        print "DEBUG: $String\n" if($ENV{'BFCQDEBUG'});
} # Debug
```

```
# Error
#
# Application Exit/Failing
sub Error {
        my ($ErrShort,$ErrLong) = @_;
        print "ERROR: $ErrLong\n";
        print "$ErrShort\n";
        exit 1;
} # Error

# END
```

Appendix B - Modified bfcqbuild.pl

This script has the additions discussed in the Customizing the Build Forge – ClearQuest Base Integration section. The modifications to the ClearQuest BTBuild record will also be required to take advantage of the modifications.

```
#!/usr/bin/cqperl

# Import Libraries
use CQPerlExt;
use strict;

# We don't want ClearQuest warnings reported in our output stream.
open (STDERR,"> nul");

# Scope of our application
{
	my $ReqState = shift || "Submit";

	my $CQSession;
	my $CurState = "";
	eval { $CQSession = ValidateInfo($ReqState); };
	Error("Could not connect to ClearQuest - Invalid Initialization Data",$@) if($@);

	# See if this is a new submission, or a resubmit.

	# Get the current state if we have one...

	eval {$CurState = RecordExists($CQSession); };
	Debug("Curstate Is: $CurState");

	if($ReqState eq "Submit" && !$CurState) {
		$ReqState = CheckSubmit($CurState);
		Debug("Reqstate Now: $ReqState");
	}
	else {
		$ReqState = ValidateNextState($CurState,$ReqState);
		Debug("ReqState Now: $ReqState");
	}


	if($ReqState eq "CantChange") {
		Error("Cannot change build record.","The build record is in a Retired state.");
	}

	# Do our action.
	if($ReqState eq "Submit") {
		eval {CheckReleaseRecord($CQSession);};
		Error("Could not create Release Record in ClearQuest",$@) if($@);
		eval {SubmitBuild($CQSession);};
		Error("Could not submit build record to ClearQuest",$@) if($@);
```

```
        }
        else {
                eval {UpdateBuild($CQSession,$ReqState);};
                Error("Could not update build record in ClearQuest",$@) if($@);
        }

        # Drop our session, don't care if we error here, as we're leaving.
        eval { CQSession::Unbuild($CQSession); };

        exit 0;
} # Scope

sub ValidateNextState {
        my($CurState,$ReqState) = @_;
        my @ValidStates = ("Submit","Resubmit","Retire","Complete","Failure");
        my $Valid = 0;
        foreach my $State (@ValidStates) {
                if($ReqState =~ /^$State$/i) {
                        $Valid = 1;
                }
        }
        Error("Invalid state requested","State [$ReqState] is not recognized as a valid state for a
ClearQuest Build record. Valid states are [".join("],[",@ValidStates)."].") unless($Valid);
        @ValidStates = ();
        $Valid = 0;
        if($CurState =~ /Failed/i) {
                @ValidStates = ("Resubmit","Retire","Submit");
                if($ReqState =~ /Submit/i) {
                        $ReqState = "Resubmit";
                }
        }
################################
#Add a Warning valid state
################################

        if($CurState =~ /Warning/i) {
                @ValidStates = ("Resubmit","Retire");
                if($ReqState =~ /Submit/i) {
                        $ReqState = "Resubmit";
                }
        }
        elsif($CurState =~ /Completed/i) {
                @ValidStates = ("Resubmit","Retire","Submit");
        }
        elsif($CurState =~ /Retired/i) {
                @ValidStates = ();
        }
        elsif($CurState =~ /Submitted/i) {
                @ValidStates = ("Complete","Failure","Submit","NoChange");
                if($ReqState =~ /Submit/i) {
                        $ReqState = "NoChange";
                }
        }
```

```
        foreach my $State (@ValidStates) {
                if($ReqState =~ /$State/i) {
                        $Valid = 1;
                }
        }
        Error("Invalid state requested","Current state is [$CurState], requested next state as
[$ReqState], valid nextstate(s) are [".join("],[",@ValidStates)."].") unless($Valid);

        return $ReqState;
} # ValidateNextState

# SubmitBuild
#
# Create a new ClearQuest Build Record.
sub SubmitBuild {
        my($CQSession) = @_;
        my $BuildRec = $CQSession->BuildEntity("btbuild");
        $BuildRec->SetFieldValue("start_datetime",genTimeStamp());
        $BuildRec->SetFieldValue("build_system_id",$ENV{'TAG'});
        $BuildRec->SetFieldValue("releasename",$ENV{'CQ_RELEASE_NAME'});
        $BuildRec->SetFieldValue("build_system_url",$ENV{'BUILD_URL'});
        $BuildRec->SetFieldValue("buildlog",$ENV{'BUILD_LOG_TEXT'});
        my $RecordId = $BuildRec->GetDisplayName();

        # Validate and Commit.
        $BuildRec->Validate();
        $BuildRec->Commit();
        Log("Build Record Created:\n$RecordId");
} # SubmitBuild

sub CheckReleaseRecord {
        my($CQSession) = @_;
        my $ReleaseRec = undef;
        eval {$ReleaseRec = $CQSession->GetEntity("dtrelease",$ENV{'CQ_RELEASE_NAME'});};
        return if(ref($ReleaseRec) eq "CQEntity");
        # Need to create the release record...
        my $CreateReleaseRec = $CQSession->BuildEntity("dtrelease");
        $CreateReleaseRec->SetFieldValue("release_name",$ENV{'CQ_RELEASE_NAME'});
        $CreateReleaseRec->Validate();
        my $CommitResult = $CreateReleaseRec->Commit();
        if($CommitResult eq "") {
                Log("Release Record Created:\n$ENV{'CQ_RELEASE_NAME'}");
        }
        else {
                Error("Could not create Release Record.",$CommitResult);
        }
}


# UpdateBuild
#
# Update an already existing Build Record.
```

```
sub UpdateBuild {
	my($CQSession,$Action) = @_;

	my $BuildRec = $CQSession->GetEntity("btbuild",$ENV{'CQ_RECORD_ID'});
	$CQSession->EditEntity($BuildRec,"modify");
	$BuildRec->SetFieldValue("build_system_url",$ENV{'BUILD_URL'})
if(exists($ENV{'BUILD_URL'}));
	$BuildRec->SetFieldValue("buildlog",$ENV{'BUILD_LOG_TEXT'})
if(exists($ENV{'BUILD_LOG_TEXT'}));

	################################
	#Grab the new Build Tag
	################################
	my $tag = BF_tag();
	################################
	#Update the new record field build_tag with the new build tag
	################################
	$BuildRec->SetFieldValue("build_tag",$tag);

	if($Action =~ /(NoChange|Resubmit)/i) {
		$BuildRec->SetFieldValue("start_datetime",genTimeStamp());
	}
	elsif($Action !~ /Retire/i) {
		$BuildRec->SetFieldValue("end_datetime",genTimeStamp());
	}
	$BuildRec->Validate();
	$BuildRec->Commit();

	if($Action=~/(Resubmit|Failure|Complete|Retire)/i) {

		################################
		#Get the build result
		################################
		my $result = bf_result();
		$CQSession->EditEntity($BuildRec,$Action);
		$BuildRec->Validate();
		$BuildRec->Commit();

		################################
		#If the result is either WARNING_FILTER or WARNING_FAILED_STEP
		#Update the build record state to Warning
		################################
		if ($result =~ /warn/i) {
			my $ret = $CQSession->EditEntity($BuildRec,'Warning');
			$BuildRec->Validate();
			$BuildRec->Commit();
		}
	}


	Log("Build Record Updated:\n$ENV{'CQ_RECORD_ID'}");
} # UpdateBuild
```

```
# RecordExists
#
# Check to see if our record exists and if this should be resubmitted, or other.
sub RecordExists {
        my ($CQSession) = @_;
        my $RecId = $ENV{'CQ_RECORD_ID'};
        my $Entity = undef;
        if($RecId) {
                $Entity = $CQSession->GetEntity("btbuild",$RecId);
        }
        my $StateRec = $Entity->GetFieldValue("State");
        return $StateRec->GetValue;
} # RecordExists

# CheckSubmit
#
# Check to see if a Submit can happen or if it's a resubmit...
sub CheckSubmit {
        my($State) = @_;
        if($State =~ /Retired/i) {
                return "CantChange";
        }
        elsif($State =~ /(Completed|Failed)/i) {
                return "Resubmit";
        }
        elsif($State =~ /Submitted/i) {
                return "NoChange";
        }
        return "Submit";
} # CheckSubmit

# ValidateInfo
#
# Make sure we have the correct env for our request.
sub ValidateInfo {
        my ($ForState) = @_;
        my @RequiredInfo = ("CQ_DBNAME","CQ_USER");

        # Check our entry data...
        if($ForState =~/Submit/i) {
                push @RequiredInfo,("CQ_RELEASE_NAME","TAG","BUILD_URL","BUILD_LOG_TEXT");
        }
        elsif($ForState =~ /Failure/i) {
                push @RequiredInfo,("CQ_RECORD_ID","BUILD_LOG_TEXT");
        }
        elsif($ForState =~ /Complete/i) {
                push @RequiredInfo,("CQ_RECORD_ID","BUILD_LOG_TEXT");
        }
        elsif($ForState =~ /Retire/i) {
                push @RequiredInfo,("CQ_RECORD_ID");
        }
        foreach my $Required (@RequiredInfo) {
                unless($ENV{$Required}) {
```

```
                        die "The required environment parameter [$Required] for operation [$ForState]
was not set";
                }
        }

        # Create a connection to ClearQuest.
        my $CQSession = CQPerlExt::CQSession_Build();
        my $CQ_Password = $ENV{'CQ_PASSWORD'} || "";
        my $DB_Set = $ENV{'CQ_DBSET'} || "";
        Debug("Connecting To ClearQuest using the following Credentials:");
        Debug("USER:$ENV{'CQ_USER'}") if($ENV{'CQ_USER'});
        Debug("PASSWORD:$CQ_Password") if($CQ_Password && !$ENV{'HIDE_PASSWORD'});
        Debug("DBNAME:$ENV{'CQ_DBNAME'}") if($ENV{'CQ_DBNAME'});
        Debug("DBSET:$DB_Set") if($DB_Set);
        $CQSession->UserLogon($ENV{'CQ_USER'},$CQ_Password,$ENV{'CQ_DBNAME'},$DB_Set);
        return $CQSession;
} # ValidateInfo

# genTimeStamp
#
# Generate a ClearQuest-friendly timestamp
sub genTimeStamp {
        my @Time=localtime();
        my $Time = sprintf("%d/%d/%d %d:%02d:%02d
%s",($Time[4]+1),$Time[3],($Time[5]+1900),($Time[2] > 12 ? ($Time[2]-12) : ($Time[2] == 0 ? 12
: $Time[2])),$Time[1],$Time[0],($Time[2] > 11 ? "P" : "A"));
        return $Time;
} # genTimeStamp

# Log
#
# Generic Logging
sub Log {
        my($String) = @_;
        print "LOG: $String\n";
} # Log

# Debug
#
# Generic Debugging
sub Debug {
        my($String) = @_;
        print "DEBUG: $String\n" if($ENV{'BFCQDEBUG'});
} # Debug

# Error
#
# Application Exit/Failing
sub Error {
        my ($ErrShort,$ErrLong) = @_;
        print "ERROR: $ErrLong\n";
        print "$ErrShort\n";
        exit 1;
```

```
} # Error

sub BF_tag {

        my $conn = bf_conn();
        $ENV{'BUILD_URL'} =~ /.*?bf_id\=(\d*)/;
        my $bid = $1;
        my $build = BuildForge::Services::DBO::Build->findById($conn, $bid);
        my $build_tag = $build->getTag();
        my $build_result = $build->getResult();
        return $build_tag;

}

sub bf_result {
        $ENV{'BUILD_URL'} =~ /.*?bf_id\=(\d*)/;
        my $conn = bf_conn();
        my $bid = $1;
        my $build = BuildForge::Services::DBO::Build->findById($conn, $bid);
        my $build_result = $build->getResult();
        return $build_result;
}


sub bf_conn {

        ################################
        #Update with the @INC path for the BuildForge Perl API client for your system
        ################################
        use lib "C:/Perl/site/lib";
        use BuildForge::Services;

        ################################
        #Use Build Forge environment variables if desired. You can then both hide the API user's
        password
        #and use different credentials for different teams by associating a different user and pass
        #with the ClearQuest environment the team is using.
        ################################
        my $hostname = 'localhost';
        my $user = 'build';
        my $pass = 'build';
        my $conn = new BuildForge::Services::Connection($hostname);
        $conn->authUser($user, $pass);
        return $conn;

}

# END
```