PTC®

Extending PTC Integrity Lifecycle Manager with Custom Code: A Best Practices Guide

Author: **Scott Milton**
Revision: **1.1**

Change History

Date	Author	Version	Description
2014-07-21	Smilton	0.1	Draft for review
2014-08-21	Smilton	1.0	General Release
2015-02-28	Smilton	1.1	Pre/Post Event, computed field recommendations

Table of Contents

1. Introduction

PTC Integrity Lifecycle Manager provides a wealth of configuration options. However, due to the complexity of the development ecosystem and demands from the user base, there will inevitably be a desire for extending the core functionality with custom code. There can be a number of reasons for extending Lifecycle Manager including, but not limited to:

- Integration to other Applications
- Custom business logic
- Automation
- Improved Interface Efficiency, e.g., role or solution specific Interfaces

Due to the ratio of end users to administrators, a large amount of administrative development time should often be exchanged for a relatively modest saving in time for end users. In other words, an investment many hours of administrative time to save a few minutes of end user time is often still a viable exchange. This is because there will be many more end users performing the operation more frequently, so 500 end users each saving a few minutes a day may quickly add up to exceed the amount of time spent by the administrator in developing the time saving functionality. However, you should also consider the amount of time to maintain and upgrade any custom code, as well as any performance impacts it may have on the Lifecycle Manager server or other users.

Note: A brief update on naming. The product formerly known as “PTC Integrity” is now named “PTC Integrity Lifecycle Manager”, since PTC Integrity now refers to a family of software and systems engineering products. For brevity and clarity, this guide uses “Lifecycle Manager” as an abbreviation for the full name, “PTC Integrity Lifecycle Manager”.

This guide is meant to provide tips and best practices for the various coding interfaces provided by Lifecycle Manager. It cannot cover each topic exhaustively, so references to other guides are noted as applicable. It will also focus specifically on Lifecycle Manager functionality and will not repeat commonly understood best practices for the programming language you intend to use; you should familiarize yourself with those best practices and standards beforehand and consider this guide to be supplementary material. The functional areas discussed in this guide include:

- Trigger Scripts
- Custom Action Buttons
- Application Programming Interface (API)
- Automated Test Execution Framework (ATEF)
- Web Services
- Report Recipes
- Command Line Interface (CLI)

2. Trigger Scripts, Server-Side

Generally speaking, trigger scripts are relatively small code files hosted on the Lifecycle Manager server, written in JavaScript but using various Java Objects. They automatically execute when a particular event happens on the Lifecycle Manager server, a particular operation is triggered from an end user, or on a schedule.

For more details:	
Location	**Details**
Lifecycle Manager server Homepage, "Event Trigger Java Documentation" Link	Object Reference Guide
Server Administration Guide	General Trigger Details
Training Course	PTC offers a full training course, including exercises on this topic.

2.1. Trigger Attributes

Names and descriptions should follow a standard that is suitable for a given implementation or solution. The standard naming aids visual recognition and allows multiple administrators to more efficiently identify what items triggers will operate on. For instance, prefixing every trigger with the item type names it can operate on. This also helps administrators visually detect which sets of triggers may go together, and how to reorder them is necessary. If there are multiple solutions configured on the same Lifecycle Manager server, an acronym representing the solution would also be recommended. Finally, another suitable naming convention might include the functional aspect of the trigger, for example, Automation/Assignment, Enforcement/Data Validation, and E-mail Notifications.

Regardless of the naming convention you may choose, the relative position of triggers is important. The relative position displayed in the Administrative client is also the order in which the triggers execute, should their rule evaluate to true. It is entirely possible for an early run trigger to change the item data such that later triggers will not execute. For instance, a trigger that modifies the item state may result in the item "jumping over" one or more triggers that are later in the execution order. Generally speaking, you should have Automation/Assignment triggers executing first, followed by Enforcement/Data Validation and finally E-mail Notifications.

2.2. Trigger Design

An event will generally consist of the following sequence:

- Begin Transaction
- PRE-Event Triggers
- Commit-Transaction
- POST-Event Triggers

Keeping in mind this sequence, event triggers can broadly be broken down into three categories, Pre-Commit, Post-Commit and Scheduled.

- Pre-Commit event triggers are included in the same transaction as the triggering event. As the operation has not yet been committed to the database when the script executes, it allows a trigger author to both validate and modify data, as well as vetoing the event and rolling back the transaction, if desired. Since the changes have not been committed to the database when pre-event triggers are executing, this category of triggers are not suitable for external notifications (emails, generation of reports or charts, calling out to other servers of any type), as the transaction could still be aborted and rolled back by an error or another trigger script.
- Post-Commit event triggers run only after the transaction has been successfully completed, making them appropriate for logging, notifications, backing up data, etc. Data cannot be changed during a post-event trigger, as the transaction has already completed.
- Scheduled triggers are available in the Workflows and Documents area of Lifecycle Manager only. This category of trigger scripts will execute at a scheduled frequency and will start their own transaction when editing items, giving the trigger script author control over when to commit any changes to the database.

In general, it is better to have many small, “single purpose” trigger scripts, rather than a single monolithic script, assuming those small scripts can be reused elsewhere in the solution. Do not hardcode static argument values within your trigger script because it adversely affects reuse and maintainability. The downside with defining multiple triggers is the administrative overhead with maintaining, naming and editing each trigger, but this can be scripted, if necessary.

Structure the trigger code in a consistent manner from trigger script to trigger script. It is recommended that there be a structure that includes the following:

- Trigger Description block
- Parameter Definition block
- Function Definition block
- Main() block
- Variable Declaration block

Use the **try…catch…finally** control structure to trap errors, especially in the case where some sort of cleanup or disconnection is required even if there was an error.

```
try {
        // Code to be executed
} catch (error) {
        // Code to execute if there is an error
} finally {
        // Optional – always execute this code
}
```

Keep system security in mind as triggers run on the server under escalated permissions and have the potential to affect process integrity and file/system security. The actions taken by a trigger are logged under the user running the operation that initially triggered the event, or the “Run As” user in the case of a scheduled trigger.

Use the abortScript() method, available from the ScriptEnvironmentBean**,** to stop a trigger from executing when a script error occurs. This is usually used in a pre-event trigger to display a message to the user and prevent (veto) them from doing something. It can also be used to display error messages when debugging a trigger script, or to display trigger configuration errors.

You may write your own custom Java packages for frequently used methods and functions. However, be careful that the Java code handles all exceptions and does not inadvertently affect the stability of the Lifecycle Manager server. Furthermore, updates to such Java packages will almost always require a server restart.

An internal best practice for triggers that will be delivered from PTC is that any trigger that sends email notifications will first check to see if triggers have been disabled by setting the “disableEmail” environment variable to true in the “data/triggers/env.properties" file. It is recommended that any email notification triggers you write follow the same standard. This is mainly to allow for easy disabling or enabling of email trigger notifications from development and test environments to avoid inadvertently “spamming” end users from an inappropriate environment.

As a starting point, modify an existing trigger script that is close to what you need. Develop and test trigger scripts starting small, gradually adding on more logic and logging script information to the server.log as frequently as necessary to help debug.

Use an editor that supports JavaScript syntax highlighting to catch simple bugs.

API specific beans are available to facilitate running API commands in the event that the functionality that you need is not available in the provided beans. In general, the native beans should always be used where possible and API usage should be limited to reading data. In the event that data must be modified, ensure that the commands are not long running and will not potentially cause deadlocks. It is highly recommended that you contact PTC Technical Support or Global Services for additional advice before deploying these types of triggers to your production environment.

Finally, ensure all custom trigger scripts are stored in a separate Custom folder inside the <InstallDir>/data/triggers/scripts directory. This helps identify custom triggers from those default trigger scripts installed with the Lifecycle Manager server when the time comes to upgrade. It is also a good idea to keep the custom scripts under version control, for example, using the Configuration Management functionality of the Lifecycle Manager server. This allows for automatic recording of change history and easy rollback of trigger scripts.

If there are server-side Lifecycle Manager triggers that are using the API, do not use the same login for standalone API programs. If both the standalone API application user and the Trigger API user (e.g., the mksis.apiSession.defaultUser) are the same login, this can potentially cause a deadlock if the standalone API program causes the API based event trigger to fire.

2.3. Performance

The number of triggers has little bearing on the overall performance of the Lifecycle Manager server system, but instead the real performance metric has to do with the actual trigger script code being executed. Most trigger scripts within the process of the server should execute in a matter of milliseconds. It is best to avoid server-side triggers that have very long execution times, such as those that synchronously connect to remote systems which could timeout, or those scripts with long running commands that iterate over many objects, such as a configuration management checkpoint. This is because the longer the database transaction remains open the greater the opportunity for conflicts or deadlocks to occur. In other words, very long running trigger scripts will lock items and prevent them from being edited by other users or processes for an unacceptable amount of time. In pre-event triggers, these scripts will also force end users to wait for the script to complete before proceeding which will generally be interpreted as unacceptable performance or the system "locking up".

Similar to the above, it is also best to avoid running batch processes (i.e., shell/cmd) from server-side triggers. If you must execute something in batch, then determine a way to asynchronously execute the process (i.e., detached from the Lifecycle Manager server process). Running the process on a totally different machine is probably the ideal option, if possible. If you absolutely need to run a batch process on the Lifecycle Manager server, then you must read from both the stdout and stderr streams. Failure to do so, results in a situation where the batch process has so much output information that if you wait until the process is finished to read the stream, the process can never actually finish because the external stream is full. This causes the trigger to hang and ultimately the Lifecycle Manager server as well. Do not use the ScriptEnvironmentBean.spawn() function because it does not read the external streams. Instead use ProcessBuilderBean for better batch process execution.

2.4. Logging

Logging is extremely important for developing and debugging trigger scripts. There is built in functionality for printing to the standard server.log file and it can be configured on each message to print to various categories and severity levels. The trigger written can decide if they want to use the default DEBUG logging category, a custom category for each trigger script, or a custom category for all trigger scripts. A good mix of the three logging options can ideally be incorporated in all custom trigger scripts:

- A global DEBUG logging will provide information to support as to where exactly a problem might be occurring, by declaring that a script is starting or finishing, and any other major operations.
- Script level custom logging category can provide useful troubleshooting information while you are developing or debugging a trigger script.
- Parameter based trigger instance logging can be equally important when you have several instances of the same trigger executing, potentially including the user name of the current user who triggered the script.

2.5. Trigger-Type Specific Best Practices

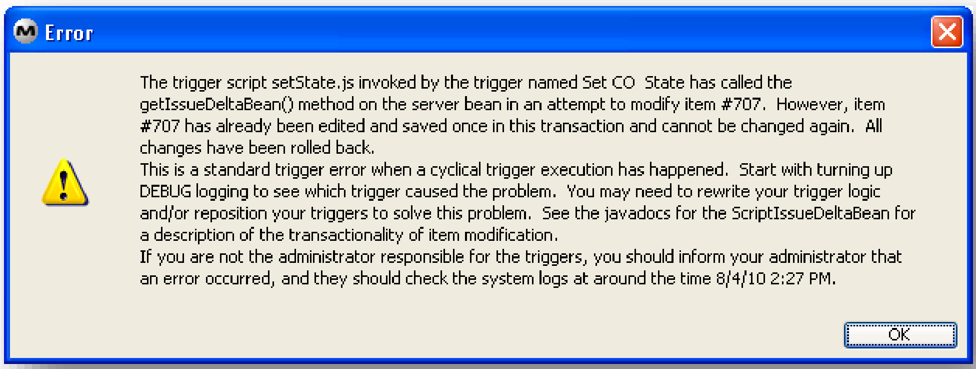
2.5.1. Configuration Management Triggers

- Determine what operation(s) the trigger needs to execute on and whether it should execute for all projects, or only some projects. After this has been determined, define it in the appropriate events file.
- The trigger resolution order will be the same as the order in which the scripts are listed for a particular operation.
- Avoid long running commands in trigger scripts, like checkpoint if possible.

2.5.2. Workflows & Documents Triggers, Rule Based

- When defining rules for triggers, start with an OR node as the root node. This provides flexibility to add additional rules to the same trigger without affecting the existing rule. Also ensure that trigger rules are well qualified. Always include the type in the rule if possible. Avoid creating rules that encompass all items, for example, Type == Defect. Always use the new values when comparing Type, ID, and other fields that set on the initial creation of the item. Since often times your trigger will not execute when new items are created, be advised that clearing the selection for the Pre and Post options does not necessarily inactivate the trigger. The rule will still be evaluated and if the rule evaluates to true, then any Assignments on the Assignment tab will be honored.
- Ensure that the decision whether or not to execute some specific action should be abstracted to the trigger rule (as much as possible), rather than handled with conditional structures inside the trigger script. Lifecycle Manager's ability for deciding whether or not a rule evaluates to true for any given delta is extremely fast, likely many times more so than the execution of interpreted trigger code, which is done through Java reflection.
- Since items cannot be modified in a post event. A technical workaround might be to use the Lifecycle Manager API to modify the item. Although this might seem to work in most cases, this practice should be avoided if possible, because control is not returned to the end user until after the post event has completed its execution. More importantly, trying to modify the same item that caused the delta can cause editing conflicts and potential deadlocks. In general, you should also avoid using the API to modify data in a pre-event trigger and use imIssueDeltaBean(s) instead.
- It is not allowed to have cascading item creation in multiple trigger scripts. As an example, consider two trigger scripts: one that creates a new item of type Defect, if an existing Task item is changed to the 'Review Failed' state and a second trigger, which creates a new Task, if a Defect is put to the 'Resolved' state. In this example, a Task can spawn a Defect and a Defect can spawn a Task. If an administrator is not careful with the initial states, it is entirely possible for an endless loop to be inadvertently created, populating the database with an endless alternating set of Defects and Tasks. In order to avoid this situation, the secondary item creation from the second trigger is detected and an exception is thrown. If you must have this functionality, incorporate all of the item creations in a single trigger script rather than having it compartmentalized in multiple scripts.
- When a pre-event trigger fires, it does so in the context of the single item whose edit fired the trigger. If the trigger script attempts to modify another item or create another item, the secondary item is added to a list of items that become part of the current transaction. Either all of the modifications made to all of the items must succeed or none of them will. This means that any pre-event triggers for the secondary item must also run, and they are subject to the same rules as the triggers running for the primary item. To ensure a pre-event trigger does not run more than once, when the pre-event trigger has completed, other secondary triggers are not permitted to obtain that Item again. Rule triggers are particularly

susceptible to secondary modification issues. As an example, consider two trigger scripts: one that creates a new item of type Defect, if an existing Task item is changed to the 'Review Failed' state and a second trigger, which edits a parent Task setting a resolved flag, if a Defect is put to the 'Resolved' state.   Ensure trigger code contains the necessary check to handle this exception or similar to the above item creation example, incorporate all of the item edits in a single trigger script rather than having it compartmentalized in multiple scripts.



The trigger script setState.js invoked by the trigger named Set CO State has called the getIssueDeltaBean() method on the server bean in an attempt to modify item #707. However, item #707 has already been edited and saved once in this transaction and cannot be changed again. All changes have been rolled back.
This is a standard trigger error when a cyclical trigger execution has happened. Start with turning up DEBUG logging to see which trigger caused the problem. You may need to rewrite your trigger logic and/or reposition your triggers to solve this problem. See the javadocs for the ScriptIssueDeltaBean for a description of the transactionality of item modification.
If you are not the administrator responsible for the triggers, you should inform your administrator that an error occurred, and they should check the system logs at around the time 8/4/10 2:27 PM.

- If the trigger is changing the state of an item:
  - Ensure the state transition is valid.
  - Ensure mandatory fields in the new state contain appropriate values.
  - Ensure open change packages are closed, if the new state does not allow open change packages.
- If your rule contains only one condition, **And** / **Or** nodes are not needed, but should generally be added anyways as this will make it easier to augment the rule later.
- If no rule is defined, the trigger runs every time any user performs an action on any item.
- Avoid using the walkHierarchy functions. However, if you must use them, rather than creating a ScriptIssueBean for each item returned, use ScriptServerBean.getIssueBeans() for an optimized processing. This is more efficient, but provides access to items in a read-only fashion.
- Finally, do not attempt to actually delete items in the rule-based trigger, as this is a time consuming command. Deleting items is generally not recommended, especially if they are used in a document, but if necessary, use a CLI script or scheduled trigger instead.

2.5.3. Workflows & Documents Triggers, Scheduled

- If one trigger takes longer to run than the next schedule trigger is set to run at, the next trigger is missed. When too many scheduled triggers are configured in a system, ensure they are sufficiently spaced apart in order to avoid their executions "stepping on" each other, as well as to keep the system performance from being negatively.
- If the Lifecycle Manager server is not running, triggers scheduled to run during that down time are skipped.
- All modified items from a scheduled trigger are part of a single transaction - if the trigger fails no items are modified.
- Scheduled triggers provide a minimum execution of 1 hour intervals. This might be too wide for some needs where a trigger needs to run every minute or second. Although you may stagger 6 instances of the same scheduled trigger to run on the hour, 10 minutes past the hour, and so on until 50 minutes past the hour to achieve a 10 minute execution interval.

You should not attempt to create any lower. Both Windows and UNIX operating systems provide down to the minute scheduled tasks and should be considered as a viable alternative. Generally such intervals are needed for integrations to other systems. Hence running the integration outside the framework for Lifecycle Manager server scheduled triggers is the recommended option.

- Scheduled triggers have also been used for data migrations. Although this might be desired given the nature of migrations in that you want a clean run of the migration, scheduled triggers may pose limitations. For instance, you cannot preserve the Created Date and Created User of the item. There is a limit to the number of items that can be uncommitted because the scheduled trigger has to commit all or nothing. Lastly, you cannot workaround mandatory fields. The "all or nothing" proposition might seem a good idea, however in large migrations this might hurt your timelines because you have to restart migrations every time you encounter an error. Consider using the 'im importissue' command to migrate data since it was specifically designed for this purpose.

2.5.4. Workflows & Documents Triggers, Other

- Time Entry triggers are useful to integrate time entered in Lifecycle Manager with your organization's corporate time keeping system. Modifications to time entries in MKS Lifecycle Manager can be integrated in real-time with other systems, but a scheduled trigger is another option if real-time performance is not needed. This allows for batch updates and mitigates the impact of connection overhead.

3. Trigger Scripts, Client-Side

These trigger scripts are only used for Configuration Management client operations.

For more details:	
Location	**Details**
Server Administration Guide	General Trigger Details, Environment Variables List
Training Course	PTC offers a full training course, including exercises on this topic.

Client-side event triggers should be configured if and only if they need to take actions directly on the client side, rather than the Lifecycle Manager server. They can be configured for pre and post events for member based operations only. They do not trigger on directory, project, sandbox or change package operations.

The client-side event trigger launches a command line program/script when triggered. The program/script needs to be available to the client environment and Lifecycle Manager does not automate their deployment. Unless additional OS level security, or a centralized location is used to protect the integrity of these programs/scripts, they should not be used for security or audit operations, as it is possible for end users to alter the program/script due to their existing in the client environment.

If using the Web service session handling is provided automatically by the Lifecycle Manager server. The Web service session uses a concurrent license, which is tracked through FlexNet.

The session that is created by the initial Web service request is reused for all subsequent requests to the server.

You do not need to disconnect from Web services since sessions time out automatically based on the Web client timeout property defined on the server.

Command Line Interface (CLI) or

Application Programming Interface (API), JAVA/ANSI C for the client side program/script, refer to their respective best practices entries.

4. Custom Action Buttons

Custom action buttons are very similar to client side event triggers and they use the same environment variables. Instead of being launched when a particular event happens, the script will be launched when the end user clicks on a toolbar button or a custom menu option. They can be used for both Configuration Management and Workflows & Documents functionality and are configured on individual viewsets.

For more details:	
Location	**Details**
Server Administration Guide	Environment Variables List

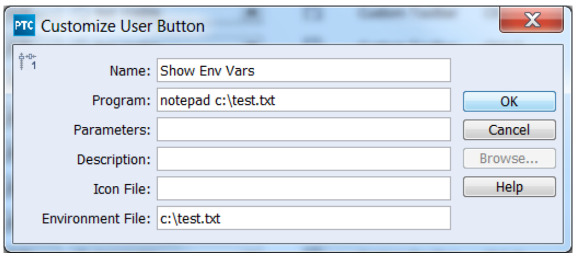
If you would like to use a batch file on windows to launch the custom button action, and would like to hide the black "DOS" window, consider using the Windows Script Host instead. Configure the program argument of the custom action button to be wscript.exe. In the custom action button script, add code similar to the following:

```
' Create a File Sys object
Set ObjFs = WScript.CreateObject("Scripting.FileSystemObject")

' Create a windows host scripting object
Set WshShell = WScript.CreateObject("WScript.Shell")

'Add the Command to run in the StrCmd variable, 0 will hide the window, true will wait for the command to complete before continuing
WshShell.Run StrCmd, 0, true
```

If you are unsure of what environment variables are populated on a particular view, configure a simple button that just prints the environment variables. When clicked, a button configured as follows will launch notepad.exe with the Lifecycle Manager populated environment variables.


Customize User Button

Name: Show Env Vars
Program: notepad c:\test.txt
Parameters:
Description:
Icon File:
Environment File: c:\test.txt

If using the Web service session handling is provided automatically by the Lifecycle Manager server. The Web service session uses a concurrent license, which is tracked through FlexNet.

The session that is created by the initial Web service request is reused for all subsequent requests to the server.

You do not need to disconnect from Web services since sessions time out automatically based on the Web client timeout property defined on the server.

Command Line Interface (CLI) or

Application Programming Interface (API), JAVA/ANSI C for the client side program/script, refer to their respective best practices entries.

5. Application Programming Interface (API), JAVA/ANSI C

For more details:	
Location	**Details**
Lifecycle Manager Server Homepage, "Java API Documentation" Link	JAVA Object Reference Guide
Lifecycle Manager Server Homepage, "ANSI API Documentation" Link	ANSI C Object Reference Guide
Integration Builders Guide	PDF Manual, General API Information and Command Certification List
Training Course	PTC offers a full training course, including exercises on this topic.

5.1. Command Certification

Virtually every command and command option on the server/client can be called through the API, but only certified commands can be expected to always produce predictable results. This list of all the certified commands is available in the Integration Builder Guide. Any uncertified commands are not guaranteed to work or to return predictable results, especially between upgrades.

5.2. Connectivity

There are two ways to connect to Lifecycle Manager via the API: a server integration point or local (client) integration point. Many server-to-server or simple tasks can be completed with a minimum of overhead by communicating directly with the Lifecycle Manager server using a server integration point: IntegrationPointFactory.createIntegrationPoint (host, port, 4, 11). This should be you default choice, unless it is otherwise determined that the local/client integration point is needed instead.

A local/client integration point allows 'remote control' of the Lifecycle Manager client, which then communicates to the Lifecycle Manager server. This provides access to the local file system the client is installed on, allows spawning the local GUI (with -g), and access to other client side only features such as sandboxes. If it is determined that you need a client side integration point instead of a server side, call: IntegrationPointFactory.createLocalIntegrationPoint(4,11) instead. The 4, 11 in the calls is the API major/minor version, which can be seen on the server homepage.

Maintain the connection as needed. When planning to make several calls to the API, there is no need to disconnect and re-connect between commands (even to change credentials). Once a session is established from the integration point, it will remain open for a default timeout period as set on the server. When planning to use a session repeatedly but sparingly, calling session.setAutoReconnect(true) on the newly created session allows it to idle timeout but transparently reconnect when called upon again.

Note: session.setTimeout() is actually "how long to wait for connection to be established" rather than an idle connection timeout.

If API based operation should show up in the history under a particular user account, but it is not convenient or possible to connect as that user, use impersonation instead. This allows a known administrator account to impersonate other accounts for the purposes of permissions and audit trails.

5.3. Performance

Reuse command runners in the same thread as appropriate. A CommandRunner object created from a session can be reused over and over again by feeding its .execute() method with Command objects. The execute method will wait for the command to complete before returning a Response unless you use .executeWithInterim(). If you need to execute commands concurrently, create a separate CommandRunner for each thread.

Use the appropriate command. Most Lifecycle Manager commands come in a "view" or "list" format (e.g. "viewissue" or "issues"). Both of these formats can accept multiple 'selections' (item IDs in the case of issues commands) e.g., "im viewissue 123 124 125" or "im issues 123 124 125".

Only the list ("issues") version can also accept a query or query definition. The viewissue command retrieves all field values for the supplied items, whereas the issues command returns only those values requested with the "--fields" parameter (or ID, Type, Summary, State by default). Thinking in terms of SQL, the "im viewissue" version of a command is similar to "SELECT * FROM items WHERE (ID = <id>)"; whereas "im issues" is "SELECT <desired columns> FROM items WHERE (ID or query matches)". In the vast majority of cases, it makes the most sense (best performance) to use the "im issues" command through the API. Only certain operations like item history require “im viewissue”.

Avoid iteration when possible. Use as few calls to the server as possible to retrieve the desired data. As an example, it is not desirable to fetch a list of item IDs via a query (e.g. "im issues --queryDefinition=<my query>") and then iteratively call "im viewissue #" or "im issues #" for each ID returned in the previous query. In most cases, it is more appropriate and vastly quicker to make a single call to retrieve all the needed data:

```
"im issues –fields=<list of desired fields> –queryDefinition=<my query def>"
```

This returns only the desired fields for all the requested items in one trip to the server and one call to the underlying database. If all of the data cannot be retrieved in a single call, (e.g. items and their related children are required), an efficient practice is to retrieve all the parent items in a single call, iterate the response in memory building a list of unique child item IDs, make a second call for all the children, and then match the items in memory as needed (rather than making a separate call to get child data for each parent). Alternatively, the second call could be a query that returns the children (by following the relationship to the parent and/or using appropriate criteria) rather than a list of IDs.

Clean up when finished. When each API component is no longer needed, call its .release() method to free any resources it occupies.

5.4. Simple Java Example

```
IntegrationPointFactory ipf = IntegrationPointFactory.getInstance();

IntegrationPoint ip = ipf.createIntegrationPoint(hostname, port, 4, 10);

Session sess = ip.createSession(user, pass);

// the above lines could be combined...    .getInstance().createIntegrationPoint(...).createSession(...)

CmdRunner cr = sess.createCmdRunner();

// set default host/credentials to use with command runner (if desired) with .setDefaultXXXX methods
Command cmd = new Command("im", "issues");

cmd.addOption(new Option("fields", "ID,Summary,Other Fields I Want"));

cmd.addOption(new Option("query", "username:Quick Query"));

Response res = cr.execute(cmd);

WorkItemIterator wii = res.getWorkItems();

While (wii.hasNext()) {
   WorkItem wi = wii.next();
      // do stuff with item
}

cr.release();
```

6. Automated Test Execution Framework (ATEF)

Lifecycle Manager provides a test execution framework that allows you to automate a test execution environment by integrating Lifecycle Manager Test Management functionality with an external testing tool. Using an adapter, you can use your third party testing tool to run test cases and have the results automatically sent back to Lifecycle Manager. The results can then be viewed and edited through the Lifecycle Manager Client.

For more details:	
Location	**Details**
Lifecycle Manager Server Homepage, "Test Management Execution API Documentation" Link	Object Reference Guide
Integrating MKS Lifecycle Manager 2009 for Automated Test Execution Guide	General details about the ATEF

If necessary, use the standard API commands in your agent scripts to run Lifecycle Manager commands.

Enable Agent logging on the Lifecycle Manager server when debugging ATEF agent scripts.

Use a combination of Agent scripts and Triggers to accomplish your goals. Some activities are easier to program as server side trigger scripts.

7. Web Services

For more details:	
Location	**Details**
Web Services Reference Guide	SOAP Reference protocol

Web service session handling is provided automatically by the Lifecycle Manager server. The Web service session uses a concurrent license, which is tracked through FlexNet.

The session that is created by the initial Web service request is reused for all subsequent requests to the server.

You do not need to disconnect from Web services since sessions time out automatically based on the Web client timeout property defined on the server.

8. Command Line Interface (CLI)

The command line interface allows you to quickly and easily automate commands that would be too burdensome to do in the GUI. It is also frequently used for *NIX clients, that do not have a display defined. Frequently run or Critical automated processes, such as automated build, should consider using the API instead of the CLI, mainly for more robust string handling exception handling.

For more details:	
Location	**Details**
CLI Reference for Server Administration	Command Reference Guide for Server Administration Commands
CLI Reference for Workflows and Documents	Command Reference Guide for W&D Commands
CLI Reference for Configuration Management	Command Reference Guide CM Commands

The Lifecycle Manager GUI is configured to retain the connection information of any particular view. This is very useful when using the same set of Lifecycle Manager servers, but can be burdensome when rapidly switching between servers, as Developers and Administrators are occasionally required to do. In this case, use the connection arguments (--hostname, --port, --user, --password) to execute the command on an alternate server to the default.

- If the client is already connected when specifying the same hostname, port and user, the password argument is not needed. This is especially useful for scripting custom action buttons.
- Adding the -g option will launch a graphical view of the command connected to the alternative server. If the view is persistent, such as an “im issues” view, it can be further manipulated in the GUI rather than back on the CLI.

As mentioned above during the API best practices, use the appropriate command. Most Lifecycle Manager commands come in a "view" or "list" format (e.g. "viewissue" or "issues"). Both of these formats can accept multiple 'selections' (item IDs in the case of issues commands)
e.g. "im viewissue 123 124 125" or "im issues 123 124 125".

Only the list ("issues") version can also accept a query or query definition. The viewissue command retrieves all field values for the supplied items, whereas the issues command returns only those values requested with the "--fields" parameter (or ID, Type, Summary, State by default). Thinking in terms of SQL, the "im viewissue" version of a command is similar to "SELECT * FROM items WHERE (ID = <id>)"; whereas "im issues" is "SELECT <desired columns> FROM items WHERE (ID or query matches)". In the vast majority of cases, it makes the most sense (best performance) to use the "im issues" command through the CLI. Only certain operations like item history require “im viewissue”.

As mentioned above during the API best practices, avoid iteration when possible. Use as few calls to the server as possible to retrieve the desired data. As an example, it is not desirable to fetch a list of item IDs via a query (e.g. "im issues --queryDefinition=<my query>") and then iteratively call "im viewissue #" or "im issues #" for each ID returned in the previous query. In most cases, it is more appropriate and vastly quicker to make a single call to retrieve all the needed data:
"im issues --fields=<list of desired fields> --queryDefinition=<my query def>"

This returns only the desired fields for all the requested items in one trip to the server and one call to the underlying database. If all of the data cannot be retrieved in a single call (e.g. items and their related children are required), an efficient practice is to retrieve all the parent items in a single call, iterate the response in memory building a list of unique child item IDs, make a second call for all the children, and then match the items in memory as needed (rather than making a separate call to get child data for each parent). Alternatively, the second call could be a query that returns the children (by following the relationship to the parent and/or using appropriate criteria) rather than a list of IDs.

9. Report Recipes

Report Recipes are used to render Lifecycle Manager data into html, xml, txt or other formats. Similar to JSP or ASP formats, certain tags will be replaced with actual data by the Lifecycle Manager server automatically when the report is run.

For more details:	
Location	**Details**
Server Administration Guide	General report recipe information
Training Course	PTC offers a full training course, including exercises on this topic.
Implementation Pattern: Server Side Report Filtering	Server side report filtering suggestions
Implementation Pattern: Client Side Report Filtering	Client side report filtering suggestions

Follow web design best practices to ensure an appealing display and do your best to ensure cross browser compatibility.

Do not directly modify existing report recipes provided by PTC. These recipes may be updated in future releases or you may need to refer to the original recipe in the future. To preserve existing report recipes for future use, create a copy of the recipe and modify the copy to suit your needs. You can move the original to backup location, if it is not being used.

Include comments in a report recipe, just as you would in any other custom code, to document the functions of tags in a report recipe. PTC recommends at a minimum including comments before each distinct block of tags to "name" them. If a report is very large, this can simplify navigation in the report recipe for you and other authors. For example:

```
<!–ChangePackageEntriesinfo–>
<!–Displaychangepackageentryheading–>
<tr> <tdclass="heading4">&cpentryheading</td> </tr>
```

Use the <%weburl%> tag to create URLs in report recipes. The <%weburl%> tag automatically resolves to http:// or https://, depending on whether SSL is enabled on the Lifecycle Manager Server. This prevents you from having to update report recipes whenever SSL is enabled or disabled.

- Recommended: <imgsrc="<%weburl%>logo.gif">
- Not recommended: <imgsrc="http://<%hostname%>:<%hostport%>/logo.gif">

Refrain from using hard-coded field names in report recipes if possible, use report parameters so that the recipe can be configured by the report creator and used for more than one purpose. If you know the only possible field that can be used is one of the default fields, you may use the <%builtinfieldname%> tag, so that reports will not break if the default field name is changed.

Use report recipes and not report templates to create reports. Template based reports are included for legacy purposes, and are not intended for new development. Report templates cannot be modified by the reporting wizard and also tend to contain hard-coded field names that, if renamed, cause the report to break.

10. Computed Fields

Generally speaking, computed fields are simple formulas that dynamically computed when a Workflows and Document Item is viewed, or on a predetermined schedule. They are not strictly speaking, custom code, but as custom code often interacts with computed fields, a few general suggestions with respect to performance have been included below. Please refer to the Lifecycle Manager Server Administration guide for additional details on computed fields, especially the section entitled “Best Practices and Key Consideration for Computed Expressions”.

Custom code authors need to keep in mind that dynamically computed fields are not stored in the database. As such, computed fields cannot be indexed to improve performance and should not be used as a query filter if at all possible.

- As a workaround consider creating an event trigger that either upon an event, or on a schedule, will execute a computation and update a standard field which has an editability rule of false.

Dynamically computed short text fields cannot be located with “all text field” search filter. To search for dynamically computed short text fields, create a query that includes a specific “field contains” comparison. If the query does not include additional filters, the query may not return optimal results.

11. For more information

For any further questions or details not covered in this guide, please contact PTC technical support.