



# Cycle Bundle for Blue Yonder Warehouse Management

**User Guide**  
v3.1.0



# Table of Contents

<b>Important Information .....</b>	<b>4</b>
<b>Project Structure .....</b>	<b>4</b>
<i>Overview .....</i>	<i>4</i>
<i>Terminology .....</i>	<i>5</i>
<i>Directories .....</i>	<i>5</i>
<i>Test Short Codes .....</i>	<i>6</i>
<b>Test Case Structure .....</b>	<b>8</b>
<i>Overview .....</i>	<i>8</i>
<i>Feature Header .....</i>	<i>8</i>
<i>Background .....</i>	<i>9</i>
<i>After Scenario .....</i>	<i>10</i>
<i>Scenario Outline .....</i>	<i>11</i>
<i>Test Case Inputs .....</i>	<i>12</i>
<b>Environment Setup and Initial Data Values .....</b>	<b>13</b>
<i>Environment Determination .....</i>	<i>13</i>
<i>Environment CSV Files .....</i>	<i>13</i>
<i>Initial Data Values .....</i>	<i>14</i>
<i>Overrides .....</i>	<i>14</i>
<i>Environment Changes for CI Pipelines .....</i>	<i>15</i>
<b>Test Case Execution Workflow .....</b>	<b>15</b>
<b>How to Get Started .....</b>	<b>16</b>
<i>Setting Up the Test Environment .....</i>	<i>16</i>
<i>WMS Users .....</i>	<i>19</i>
<i>Enabling and Executing the First Test .....</i>	<i>19</i>
<b>Mobile Application Testing .....</b>	<b>20</b>
<b>Regression Testing .....</b>	<b>21</b>
<i>Regression Tests .....</i>	<i>21</i>
<i>Playlists .....</i>	<i>21</i>
<b>Test Case and Utility Details .....</b>	<b>22</b>
<i>Test Case Specifications .....</i>	<i>23</i>

<b>Using Datasets .....</b>	<b>24</b>
<i>Load dataset .....</i>	<i>24</i>
<i>Cleanup dataset .....</i>	<i>24</i>
<b>Customizations .....</b>	<b>25</b>
<i>Overview .....</i>	<i>25</i>
<i>Directory Load Path .....</i>	<i>25</i>
<i>Importing Files .....</i>	<i>25</i>
<i>Scripts and Datasets .....</i>	<i>26</i>
<i>Adding Additional Customization Levels .....</i>	<i>27</i>
<b>Environment CSV File Details .....</b>	<b>27</b>
<b>Dynamic Data .....</b>	<b>29</b>
<i>Overview .....</i>	<i>29</i>
<i>Dynamic Data Setup .....</i>	<i>29</i>
<i>Instruction Types .....</i>	<i>30</i>
<i>Enabling Dynamic Data .....</i>	<i>30</i>
<i>Using Dynamic Data in Test Case Input .....</i>	<i>30</i>
<i>Setting Test Input Values .....</i>	<i>31</i>
<i>Returning Multiple Values .....</i>	<i>31</i>
<i>Retry Logic .....</i>	<i>32</i>
<i>Examples of Dynamic Data Instructions .....</i>	<i>32</i>
<b>Pre and Post Validations .....</b>	<b>35</b>
<i>Overview .....</i>	<i>35</i>
<i>Enabling Validations .....</i>	<i>35</i>
<i>Validation Setup .....</i>	<i>35</i>
<i>Input Parameters for Validations .....</i>	<i>36</i>
<i>Validation Results .....</i>	<i>37</i>
<i>Validation Example .....</i>	<i>38</i>
<i>Validation of Integrator Transactions .....</i>	<i>40</i>
<b>Tag Usage .....</b>	<b>41</b>
<b>Standard Wait Time Variables .....</b>	<b>41</b>
<i>Wait/Within Time Values .....</i>	<i>41</i>

# Important Information

---

This section contains important information about the Cycle Bundle, please review prior to use:

- Because of security changes in MOCA environment variable handling, **Cycle version 2.7.1 (or greater) is required** for use with **Cycle Bundle v3.1.0**
- For existing users of the Bundle only, because of security changes in MOCA environment variable processing, **all Custom MSQL scripts/datasets will need to be modified**. Also, **Test Cases and Utilities will need to remove CycleScript steps that set and use MOCA environment variables**. Please see the document *“Cycle Bundle Upgrade Notes for Blue Yonder WMS”* for more details.
- For existing users of the Bundle only, **all modified Test Cases, Utilities, Playlists, Test Specifications, and other data** that has both a Base and Custom directory should be **placed in their associated Custom directory and not in the Base directory**. This helps preserve the contents of the Base directory and greatly assists in the upgrade process.
- Pre/Post Validation and Dynamic Data base functionality has been implemented, but this functionality **has not been integrated into Bundle Test Cases** at this time.
- Execution of the Bundle and its Test Cases **will interact with your configured WMS**. The Bundle should **ONLY be run in non-production environments** (only in test instances with regular instance snapshots being generated).
- Sometimes **datasets can hang** due to MOCA commands on the WMS or network conditions taking longer than 30 seconds.
- All **new Test Case directories** that you create for your own new Test Cases should be added to the **Test Cases/Custom** directory and not the Test Cases/Base directory.
- Volume Testing is not supported directly with the v3.1.0 release of the Cycle Bundle.
- Important upgrade Information for this release is contained in the document *“Cycle Bundle Upgrade Notes for Blue Yonder WMS”* at the root level of the Bundle. Please **read prior to using the Bundle** for important information on upgrading from a prior release of the Bundle.
- Important information about this release is contained in *“Cycle Bundle Release Notes for Blue Yonder WMS.pdf”* at the root level of the Bundle. Please **read prior to using the Bundle**.

## Project Structure

---

### Overview

The Cycle Bundle contains the complete directory structure and files for your **Blue Yonder 2019.1.1 and/or 2020.1.1** testing projects. The Cycle Bundle can be installed locally or on a shared directory and accessed through the Cycle application. Tryon Solutions recommends creating a version-controlled repository to track changes against the Cycle Bundle relative to your projects.

See the [Cycle User Manual](#) for detailed descriptions of the fundamental Cycle concepts mentioned in this guide. It is highly recommended that you read the Release Notes document (*Cycle Bundle Release Notes for Blue Yonder WM*) prior to use of the Bundle for important information about this release.

# Terminology

General terminology that will be used in this document includes:

- **Feature File** – A Cycle Feature file is a physical file that contains test Scenarios. Within the Bundle, it allows for the implementation of either a Test Case or a Utility. All Feature files end in a “.feature” extension.
- **Scenario** – A collection of CycleScript code that encapsulates a logical set of steps. Those steps may implement a component of a Test Case (Scenario Outline, Background, After Scenario) or a component of a Utility (a Utility Scenario).
- **Test Case** – A Test Case is an instance of a Feature file, that in the implementation of the Bundle, contains a single Scenario (Scenario Outline), a single Background, and a single After Scenario. The Test Case will implement and verify test specifications associated with the Test Case. Scenario Outlines, Background, and After Scenarios are discussed in greater detail later in this document.
- **Utility** – A Utility is another instance of a Feature file. Utilities are organized by a functional area (i.e. Inventory or Picking) and are broken down into Terminal and Web versions depending on the type of interface they are interacting with. A Utility contains one or many Scenarios (called Utility Scenarios) that implements a logical piece of functionality (i.e. Terminal Login or Web Login) that can be called from Test Cases or other Utility Scenarios. Utility Scenarios are re-usable meaning they can be called by multiple test cases or other Utility Scenarios and have a well-defined interface (in terms of its inputs and outputs).
- **Playlist** – Playlist is a collection of Test Cases that can be run sequentially via Cycle. All playlists will have a “.cycplay” file extension.
- **Test Case Inputs** – CSV files that contain the inputs to Test Cases.
- **Datasets** – Collection of MOCA SQL scripts (MSQL) used to setup and/or cleanup data in the WMS. MSQL files have a “.msql” file extension.
- **Environments** – Set of configuration files to initialize and define the environment and the input to test cases.

## Directories

The following directories are included in the Cycle Bundle. Directories may have sub-directories for Base and Custom. These directories allow for customizations of the Base Bundle product (referenced later in this document).

1. **Data:** Contains data and configuration files that will be used by many different Test Cases
  - **Interfaces:** Contains XML files used by Integrator Test Cases.
  - **Locators:** Contains CSV files storing native application or web screen locator values.
  - **Serial Numbers:** Contains files with serial numbers for a specific serial number type.
  - **Dynamic Data:** Contains example CSV files used in the dynamic selection of test case inputs.
2. **Datasets:** Contains MOCA SQL (MSQL) load and cleanup files used by Test Cases in the Cycle Bundle.
3. **Documentation:** Contains documentation for the Cycle Bundle.
  - **FILE:** Cycle Bundle User Guide for Blue Yonder WM 2019.1.1 and 2020.1.1.pdf
  - **Test Case and Utility Details and Inputs:** Text documents that contain detailed information about Test Cases and Utilities features
  - **Test Specifications:** Word documents representing Test Specifications broken down by functional area.
4. **Environments:** Contains the configuration files used to determine the environment for a test case and initializing the Cycle Test Case to execute against the environment.

5. **Playlists:** Contains Cycle playlists of Test Cases either grouped by functional area or set to run a logical set of test cases.
6. **Scripts:** Contains the MSQF Files and Groovy subdirectories. These subdirectories contain the MSQF and Groovy scripts used by the Test Cases and Utilities.
7. **Test Case Inputs:** Contains the CSV files which serves as the input parameters to Test Cases.
  - **Samples:** Examples for initially setting up Test Cases in your environment. These are provided for reference **ONLY** as these are the inputs used when running the Bundle Test Cases at Tryon. Inputs will **need to updated** to reflect your WMS configuration **prior to use**.
  - **Samples/Details:** Contains CSV files to serve as another layer of input parameters for Test Cases.
8. **Test Cases:** Contains the user executable Test Cases broken down by function areas.
9. **Utilities:** Contains Utility Feature files consisting of Utility Scenarios. Starting in this release, Utilities have been broken out into **Terminal, Web, Mobile, and API** sub-directories. The appropriate imports have been modified so no changes to Test Cases are required for these location changes. Examples include:
  - **Environment.feature:** reading in testing environment parameters relative to your WMS environment under test
  - **Import Utilities.feature:** contains Scenarios to import dependencies
  - **Terminal/Terminal Utilities.feature:** contains login/logout Scenarios for the Blue Yonder Terminal, among others.
  - **Web/Web Utilities.feature:** contains login and logout Scenarios for the Blue Yonder Web, among others.
  - **Mobile/Mobile Utilities.feature:** contains login and logout Scenarios for the Blue Yonder Mobile Applications, among others.
10. **Test Case Validations:** Contains examples of CSV files used in pre/post validations.
11. **FILE:** WMS-BUNDLE-3.1.0.cycproj
  - File that contains the Cycle project definition and settings
12. **FILE:** Cycle Bundle Release Notes for Blue Yonder WMS.pdf
  - Important information about this release, please review before using the Cycle Bundle.
13. **FILE:** Cycle Bundle Upgrade Notes for Blue Yonder WMS.pdf
  - Important information about upgrading to this release, please review before using the Cycle Bundle.

## Test Short Codes

A short code is used in Test Cases and their associated Test Case Inputs CSV file. The short code's format is:

- **<BASE|CUSTOM>-<FUNCTIONAL\_AREA>-XXXX**
  - **BASE or CUSTOM** specifies if the test case is part of the standard Bundle or if it was developed as a modification or addition to the Bundle.
  - NOTE:** Any user customizations should be stored in the Custom directories to maintain the integrity of the Cycle Bundle
  - **FUNCTIONAL\_AREA** denotes a specific functional area that best maps to the functionality the test case is testing. Some examples include:
    - PCK – Picking
    - INV – Inventory
    - INT – Integration

- RPL - Replenishment
  - BLD – Pallet Build
  - YRD – Yard
  - RCV – Receiving
  - CNT – Counting
  - LDG – Loading
  - SHP – Shipping
  - WAV – Waves and Allocations
  - PRD – Production
  - DD – Dynamic Data (currently used for example purposes)
  - VAL – Pre and Post Validation (currently used for example purposes)
- XXXX is a 4-digit code that uniquely names the test case within its functional area. Usually each test case is an increment of 10 higher from the prior leaving room for test case expansion.
    - If the code is in range 0010-1000, this denotes that it is a **Terminal** based test case.
    - If the code is in range 1010-2000, this denotes that it is a **Blue Yonder Web** based test case.
    - If the code is in range 2010-3000, this denotes that this is an **Integrator** test case.
    - If the code is in range 4010-4000, this denotes that this is a **Blue Yonder Windows Native Application** test case. Note, currently there are not Native Application test cases delivered in the Cycle Bundle.
    - If the code is in range 4010-5000, this denotes that this is a **Mobile Application** test case.
    - If the code is in range 5010-6000, this denotes that this is a **Flow test case using Web and Terminal Interfaces**. Flow test cases represent end-to-end Scenarios and ones that use more than one user interface (Web and Terminal).
    - If the code is in range 6010-6100, this denotes that this is a Flow test case using **Web and Mobile Application Interfaces**. Flow test cases represent end-to-end Scenarios and ones that use more than one user interface (Web and Mobile Application).
    - If the code is in range 8010-9000, this denotes that this is a **Blue Yonder WMS API** based test case.

**Short codes are used in Test Cases in the following way:**

- Test Case file names contain the short code followed a descriptive name for the test (which includes if the test case is interacting with the Terminal or the Web). An example is:
  - BASE-INV-0030 Terminal Inventory Adjustment Increase.feature
- Test Case Scenario Outlines include a reference to its associated CSV file in the “/Test Case Inputs” directory. For example:
  - CSV Examples: Test Case Inputs/BASE-INV-0030.csv

# Test Case Structure

## Overview

Each Test Case in the Cycle Bundle will include the following sections in the following order:

1. **Feature Header**
2. **Background**
3. **After Scenario**
4. **Scenario Outline**
5. **Test Case Inputs**

An explanation of each Test Case section follows.

## Feature Header

Each Feature file includes a header of [comments](#) that contains the relevant WMS and Cycle versions, assumptions, and usage instructions. An example of the Feature Header is as follows:

```
#####
# Copyright 2020, Tryon Solutions, Inc.
# All rights reserved.  Proprietary and confidential.
#
# This file is subject to the license terms found at
# https://www.cycleautomation.com/end-user-license-agreement/
#
# The methods and techniques described herein are considered
# confidential and/or trade secrets.
# No part of this file may be copied, modified, propagated,
# or distributed except as authorized by the license.
#####
# Test Case: <Name of the Test Case>
#
# Functional Area: <Functional Area being tested>
# Author: <Your Name>
# Blue Yonder WMS Version: Please Consult Release Notes
# Test Case Type: Regression
# Blue Yonder Interfaces Interacted With: Terminal, Mobile, MOCA, Web, Integrator, API
#
# Description:
# This Test Case generates a receipt via MSQL and receives it.
#
# Input Source: Test Case Inputs/<csv file containing test data>
# Required Inputs:
#     parameter1 – Description of parameter 1
```



```

#      parameter2 – Description of parameter 2
# Optional Inputs:
#      parameter3 – Description of optional parameter 3
#      parameter4 – Description of optional parameter 4
#
# Assumptions:
#      - Assumption one
#      - Assumption two
#
# Notes:
# - Test Case Inputs (CSV) - Examples:
#      Example Row: <description of first example row>
#      Example Row: <description of second example row>
# - Usage note 1
# - Usage note 2
#
#####

```

## Background

The [Background](#) (one per Test Case) is a specific type of [Scenario](#) that runs prior to every test execution example.

The following is an example of a Background Scenario.

**NOTE:** The sequence of steps is critical to supporting the configuration of environment settings and customization levels. Lines 76 through 84 should never be changed except for modifying line 81 to import different files and line 82 to change the “test\_case” value:

```

76 Given I "setup the environment"
77     Then I assign all chevron variables to unassigned dollar variables
78     And I import scenarios from "Utilities/Base/Environment.feature"
79     When I execute scenario "Set Up Environment"
80
81     Given I execute scenario "Terminal Receiving Imports"
82
83     Then I assign "BASE-RCV-0010" to variable "test_case"
84     When I execute scenario "Test Data Triggers"
85
86 And I "load the dataset"
87     Then I assign "Receiving" to variable "dataset_directory"
88     And I execute scenario "Perform MOCA Dataset"

```

The Background will perform the following actions:

1. Convert all the test arguments that are initially read in and stored as chevron variables to standard cycle dollar variables.  
**NOTE:** With Bundle v3.1.0, the prior call to "I assign all chevron variables to MOCA environment variables" has been removed as part of the changes in MOCA environment variable processing (see the "*Cycle Bundle Upgrade Notes for Blue Yonder WMS*" document)
2. Import "Utilities/Base/Environment.feature", which contains Scenarios to determine and set up the environment.
3. Execute "Set Up Environment" to determine the environment and configure the Cycle test to run against the environment
4. Import any needed Utility scenarios. The Import Scenarios are stored in "/Utilities/Base/Import Utilities.feature".
5. Assign the "test\_case" variable and run the "Test Data Triggers" Scenario
6. Set which dataset to use and run the MOCA Dataset

## After Scenario

The [After Scenario](#) (one per Feature file) runs after every example of the test case. The After Scenario is responsible for environment clean-up such as closing browsers, exiting terminals, and cleaning up regression data.

The call to *Test Completion* performs the logout calls from a Web, Mobile, and Terminal perspective and will be used for future expansion activities that occur when a test case completes.

The following is an example After Scenario. Note that the first 2 lines should be in all test cases in the exact sequence shown:

```

85 After Scenario:
86 #####
87 # Description: Logs out of the interface and cleans up the dataset
88 #####
89
90 Given I "perform test completion activities including logging out of the interfaces"
91     Then I execute scenario "Test Completion"
92
93 And I "cleanup the dataset"
94     Then I assign "Receiving" to variable "cleanup_directory"
95     And I execute scenario "Perform MOCA Cleanup Script"

```

## Scenario Outline

The Scenario Outline is made up of the CSV Examples which provides the test data for a test case and the Cycle steps needed to perform the test case. The two main types of Scenarios included are:

1. **CSV Examples:** This contains a reference to the CSV file that holds the test case data. All CSV files for this are located in the "/Test Case Inputs" directory.
2. **Cycle Steps:** Following the CSV Examples is one to many blocks of cycle steps that perform the actions of the test case. These actions will include activities like signing onto a terminal or web site, receiving inventory, allocating order, loading trucks, and verifying activities. Most activities will be accomplished by calls to Utility Scenarios provided by the Bundle to interact with the WMS system. Some steps may be assigned variables needed within the utilities and validation steps.

```

102 |
103 @BASE-RCV-0010
104 Scenario Outline: BASE-RCV-0010 Terminal Inbound Receiving
105 CSV Examples: Test Case Inputs/BASE-RCV-0010.csv
106
107 Given I "execute pre-test scenario actions (including pre-validations)"
108     And I execute scenario "Begin Pre-Test Activities"
109
110 Then I "login to the Terminal"
111     When I execute scenario "Terminal Login"
112
113 When I "open the Receipt, process workflow, receive each receipt line individually, and deposit to location"
114     Given I "open the Receipt"
115         Then I execute scenario "Terminal LPN Receiving Menu"
116         And I enter $trknum in terminal
117
118     And I "check to see if we have a workflow and process the workflow"
119         Then I execute scenario "Terminal Process Workflow"
120
121     When I "perform Receiving (non-ASN)"
122         Then I execute scenario "Terminal Non-ASN Receiving"
123

```

## Test Case Inputs

Almost all test cases require some input values to run. Test cases get the needed input values from the CSV file referenced in the CSV Examples line of the Scenario Outline. The Bundle stores Test Case Input CSV files in the *"/Test Case Inputs"* directory. The name of the CSV file must match the value on the CSV Examples line. As a standard the CSV file is given the name of the Test Case's "short code" (reference earlier).

The format of a Test Case Input CSV file is a single heading row containing all the input's names. The remaining rows in the file are sets of data (examples). Each row represents one example of the test. When the test case is executed, it will read the first row of data, perform the entire test process (including environment setup, data setup, and data cleanup). This process is then repeated for each data row in the file.

Below is a sample Test Case Input CSV file:

```

wh_id,prtnum,untqty,stoloc
WMD1,SHAMPOO,48,G2-230-A
WMD1,SOAP,40,M4-541-C

```

**NOTE:** Test Case Inputs and examples are provided in the Test Case Inputs/Samples directory of the Cycle Bundle. These are provided for reference **ONLY** as these are the inputs used when running the Bundle Test Cases at Tryon. Inputs will **need to updated** to reflect your WMS configuration **prior to use**.

# Environment Setup and Initial Data Values

---

The Bundle is designed to support test cases executing in many different test environment configurations. When a test case is executed, a set of simple configure files are evaluated to determine the environment for the test and then configure the test to run within the environment.

## Environment Determination

Every test case must be run against a specific environment. An environment will specify the WMS instance that will be tested. The environment can be determined by one of two ways. Either a test case specifies the environment by populating the \$environment cycle variable within the test params or by reading the Environment.csv located in the /Environments directory.

The Environment.csv file must have the following format:

```
environment
<<Environment Name>>
```

The following is an example Environment.csv:

```
environment
WMST_1
```

## Environment CSV Files

Once the environment for the test execution is determined, Cycle will read in a set of CSV files containing initial data values for the test. Each file read in will contain a list of variable/value pairs. The values in these files will be used by Cycle to perform the test execution.

Cycle will read in the following six CSV files in the sequence listed below. The warehouse specific files are only used when the Warehouse Id (**wh\_id**) is defined as a test case input value.

1. Environment and CI (Continuous Integration) Override file
2. Environment and Warehouse specific Override file
3. Environment specific Override file
4. Warehouse specific All Environments Override file
5. All Environment Override file
6. Warehouse specific Environment file
7. Environment file (**mandatory**)

See Section “*Environment CSV File Details*” below for more information on each of these Environment files.

## Initial Data Values

The initial data values for a variable can be setup either by being defined within the test parameters or by being loaded from one of the environment CSV files. Once a variable is assigned an initial data value the value will not be overridden by subsequent CSV files. The variable values may be changed by the test case logic itself. These variables are used by the datasets and within the operation of the test case.

Cycle Credentials and Connections are setup within the Cycle application's configuration panel – this will be demonstrated in a section further below.

The variables assigned within the environment CSV files include, but is not limited to:

Variable	Variable Description
wh_id	Warehouse Id
client_id	Client Id
moca_server_connection	MOCA server instance setup in Cycle "Connections"
devcod	Terminal Device Code
moca_credentials	MOCA User Name and Password Credential setup in Cycle "Credentials"
terminal_credentials	Terminal User Name and Password Credential setup in Cycle "Credentials"
username	User Name
mobile_credentials	Mobile Application User Name and Password Credential setup in Cycle "Credentials"
web_credentials	Web User Name and Password Credential setup in Cycle "Credentials"
ui_credentials	Native UI User and Password Credential setup in Cycle "Credentials"
browser	Browser being used for tests (Chrome or Edge)
terminal_server	<Server Name>:<Port>
terminal_protocol	Protocol to connect to Terminal (SSH or telnet)
web_ui	URL for WMS web portal
mobile_ui	URL for WMS Mobile Application Portal

## Overrides

The Environment Override CSV files are optional. The purpose of these files is to allow team members to share a single environment files, via source control or file copying, while still being able to execute test cases using different terminal IDs, user logins, or other data values the are specific to the individual tester. These files are normally used to override variable like Device Code, Vehicle Type, and Username. The override files are normally not shared between testers.

## Environment Changes for CI Pipelines

The following enhancements to more effectively integrate the Cycle Bundle with Jenkins Continuous Integration (CI) Pipelines (and specifically to have the ability to dynamically set runtime parameters that track to the WMS under test) have been integrated.

1. On the invocation of Cycle-CLI in a Pipeline with Cycle Playlists, functionality has added to set a Windows Environment variable (**BUNDLE\_CI\_ENVIRONMENT**) prior to the call to Cycle-CLI. For instance:
  - a. `Set-Item -Path Env:BUNDLE_CI_ENVIRONMENT -Value "${BUNDLE_CI_ENVIRONMENT_2020}";  
${WORKSPACE}\\cycle-extract\\cycle-cli.exe ....`

With the changes to the Cycle Bundle, this Environment variable and setting will trigger the Cycle Bundle to use the Environment variable value as the **Environment** (normally stored in the **Environments/Environment.csv** file) value and to specifically look for a CI override file in the Environment directory.

For example, if the Windows Environment variable **BUNDLE\_CI\_ENVIRONMENT** is set to **\$(BUNDLE\_CI\_ENVIRONMENT\_2020)** and that value is **WMS2020**, then the Bundle's environment setting (the WMS under test) will be set to **WMS2020** and the Cycle Bundle will look for a **Environments/WMS2020/WMS2020\_Environment\_Override\_CI.csv** file to get setting specific Bundle settings to the use with the execution of this Pipeline and this 2020 WMS instance.

The **WMS2020\_Environment\_Override\_CI.csv** file can contain (for example) specific WMS users to use, specific device codes to use, specific Internet Browsers to use, and any other specific parameter you want to override when using the Cycle Bundle in a Jenkins CI Pipeline.

**An example Override\_CI file would be:**

```
1  variable,value
2  devcod,MTFJENKINS
3  start_loc,MTFJENKINS
4  vehtyp,HAND
5  username,JENKINSUSER
6  browser,edge
7  mobile_devcod,WEBMTFJENKINS
8  mobile_start_loc,WEBMTFJENKINS
9  parallel_testing,TRUE
10
```

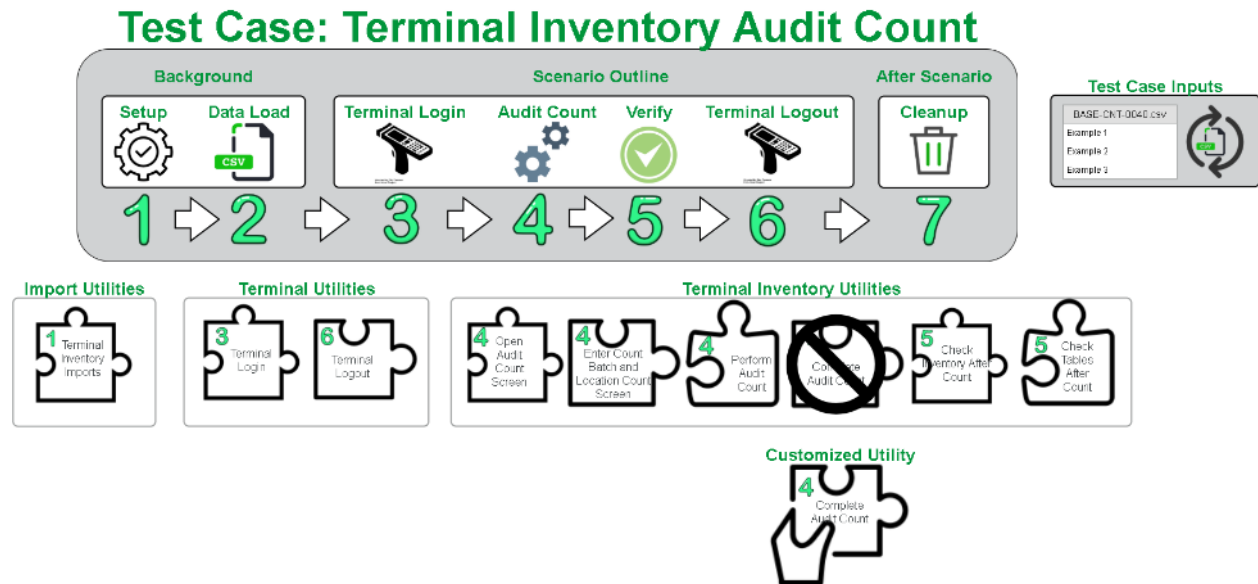
Note, the **parallel\_testing** flag is used to tell the Cycle Bundle not to attempt to kill associated Web Drivers at the end of Test Cases given other Test Cases (to other WMS instances) might be running on the same Cycle Client.

## Test Case Execution Workflow

The following diagram describes the execution workflow of a Bundle Test Case (i.e. Terminal Inventory Audit Count). Its components include:

- The grey boxes represent the Test Case including its components and its execution flow (denoted by each component's number).


- The bottom row of boxes represent the Utilities files and the Utility Scenarios contained within them. Utility Scenarios are also numbered, showing the flow/call from the Test Case.
- In the Terminal Inventory Utilities - Utility box, this example is showing a customization where the user is providing their own version of the Utility Scenario for Complete Audit Count (which would be included in the Utilities/Custom directory)



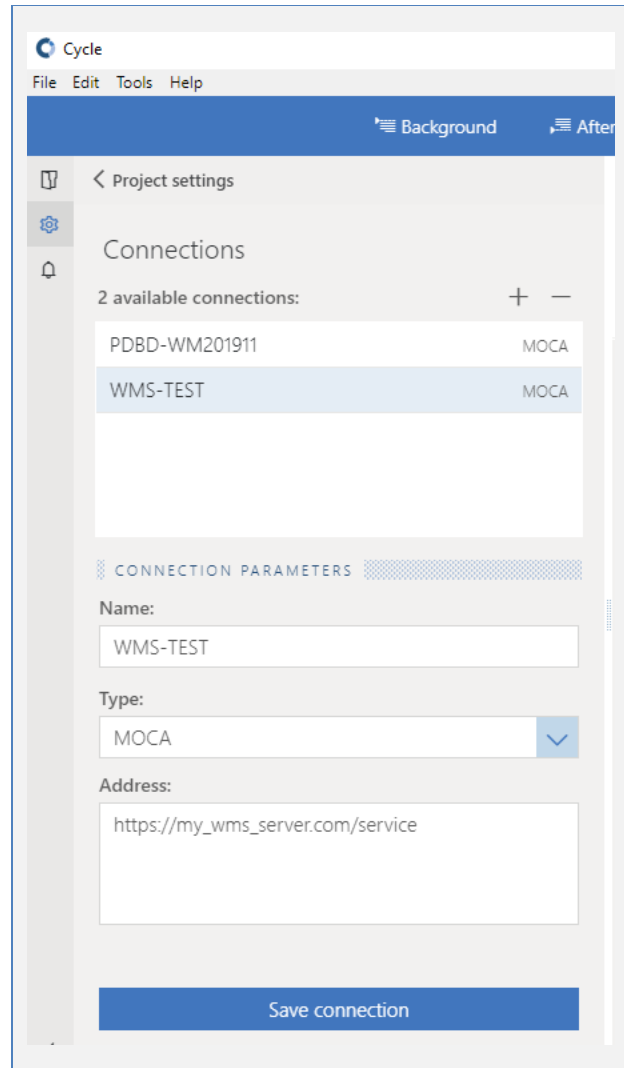
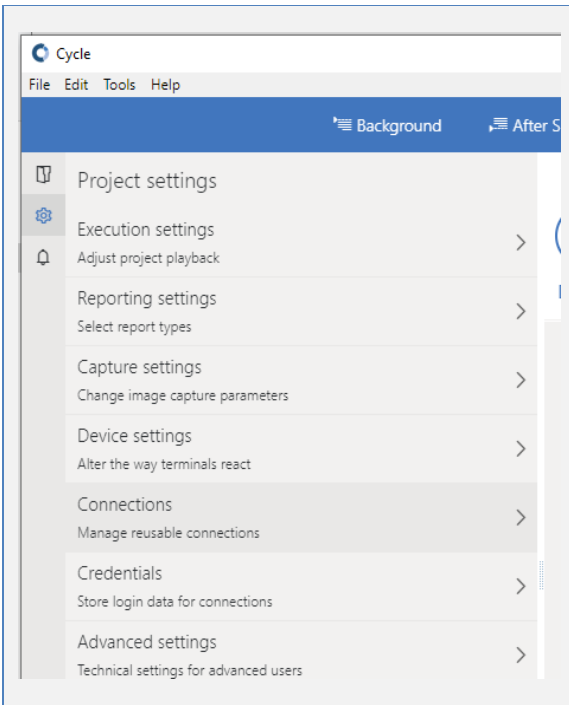
## How to Get Started

### Setting Up the Test Environment

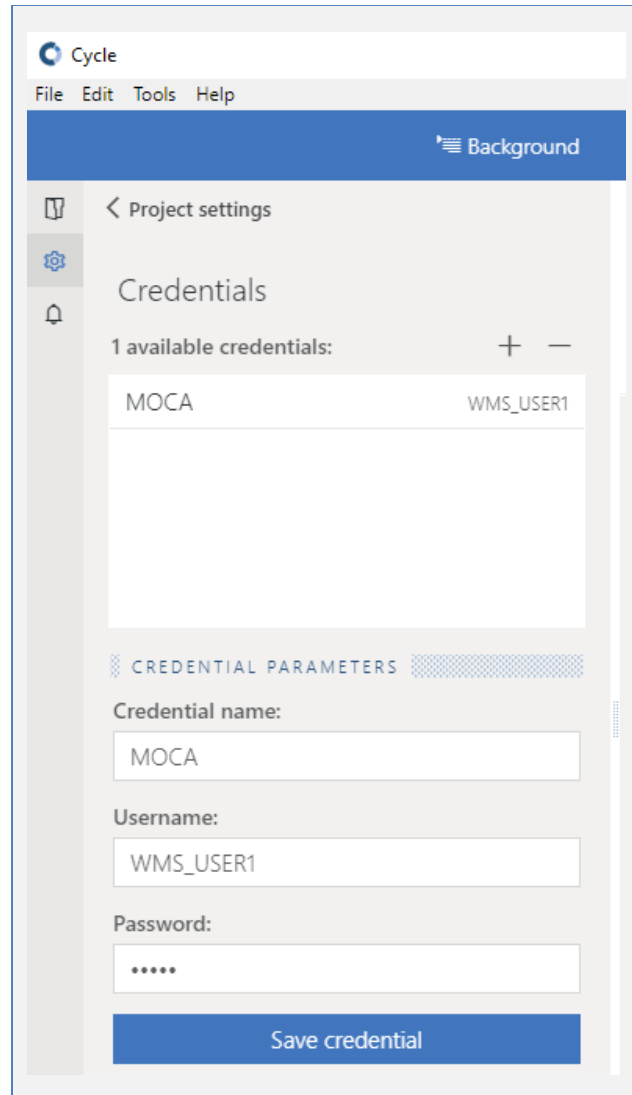
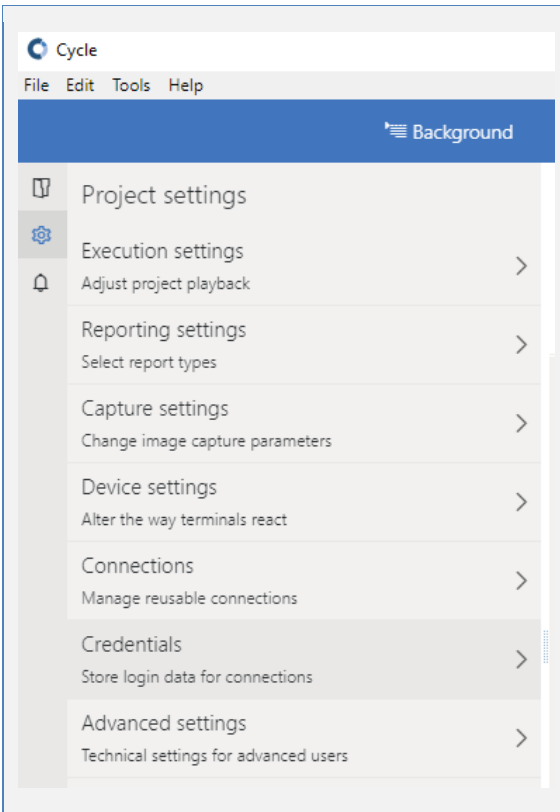
The first step of getting started testing with the Bundle is to configure the Bundle to connect and use a WMS system. Follow this step to get Cycle and an environment configured:

1. Define the WMS server in Cycle. This is done using the Connections option within the Projects setting Menu .





2. Set up the MOCA credentials for your WMS system. If you will use the same user and password to connect to MOCA, a Terminal, the Web Portal, and the Mobile Application, then you only need to add the credentials for that user.



3. Set up the environment configuration file. In Windows File Explorer, navigate to the Environments directory. Within this directory, create a new subdirectory with the name that you want to use for the environment.
4. Copy the Environment\_Template.CSV file from the Environments directory into the new subdirectory and rename the file using the format <Environment Name>\_Environment.csv.
5. Edit the new <Environment\_Name>\_Environment.csv file. Change the values within the file replacing the place holder values with values that are valid for the WMS system. Note that the value for moca\_server\_connection should match the Connection created in step 1 and the value for moca\_credentials, terminal\_credentials, web\_credentials and ui\_credentials should all match the Credential created step 2. An example of the <Environment\_Name>\_Environment.csv included below.
6. Validate the environment setup. Using Cycle, open the Feature file "Utilities/Base/Verify Environment.feature". Set the test to with these tags (that are applicable for the type of testing you are performing): @verify-MOCA, @verify-terminal, @verify-web-ui, @verify-mobile, @verify-api and Press the play button and validate if all of the test scenarios pass.

An Example <Environment\_Name>\_Environment.csv file:

1	variable	value
2	wh_id	WH1
3	src_wh_id	WH1
4	client_id	CLIENT A
5	devcod	RDT001
6	start_loc	RDT001
7	vehtyp	HAND
8	recovery_deploc	QA_HOLD
9	moca_server_connection	WMS-TEST
10	moca_credentials	MOCA
11	terminal_credentials	MOCA
12	username	USERNAME
13	web_credentials	MOCA
14	ui_credentials	MOCA
15	browser	Chrome
16	server	https://wms-test-moca.your-domain-xyzabc.com/service
17	terminal_server	wms-test-rf.your-domain-xyzabc.com:your-port
18	terminal_protocol	telnet
19	web_ui	https://wms-test.your-domain-xyzabc.com/
20	wms_inbound_directory	\$LESDIR/files/hostin
21		

## WMS Users

All test cases were developed and executed at Tryon Solutions with WMS users that have SUPER privileges/settings.

Some of the Inventory adjustment tests involving approvals require WMS users that have particular thresholds set to specifically trigger approval workflows. These users should ideally be represented in your WMS environment with the following characteristics/settings:

- INV\_ADJ\_USR
  - set adj\_thr\_unit = 1, adj\_thr\_cst = 1 in les\_usr\_ath table
- INV\_ADJ\_USR\_COST
  - set adj\_thr\_unit = 0, adj\_thr\_cst = 500 in les\_usr\_ath table

## Enabling and Executing the First Test

Once the configuration of the environment setup within the Bundle has been validated with Cycle, you can begin to set up test cases. The following instructions can be used to set up a test that performs receiving using the terminal. Please note, these steps are only required on new/fresh installs of the Cycle Bundle.

1. Create the Test Case Input CSV file. In Windows File Explorer copy the file “Test Case Inputs/Samples/BASE-RCV-0010.csv” to “/Test Case Inputs”.

2. Edit the Test Case Input CSV file. Using Notepad, Excel, or whatever application you prefer for editing CSV files, **open BASE-RCV-0010.csv and replace all the values in the second and subsequent rows with values that are valid and specific to your WMS environment.**
3. Open and execute the test. In Cycle, open the Test Case file “Test Cases/Base/Inbound/BASE-RCV-0010 Terminal Inbound Receiving.feature” and press the Play button to execute the test.

Repeat the steps for other test cases or add more rows in the Test Case Inputs CSV file to create more examples of the same test case.

## Mobile Application Testing

Testing of the Blue Yonder Android Mobile Application is supported in the v3.1.0 release of the Cycle Bundle via interaction with the Blue Yonder Mobile Application emulator. The emulator displays the mobile application in a standard Web interface enabling all standard Cycle Web steps to be utilized during automation development and execution.

Support has been added to the Cycle Bundle in the form of new Mobile Utilities and Test Cases. Mobile Test Cases leverage common datasets and MSQl scripts used for Terminal support. Utility Scenarios that are not specific to a user interface (Terminal or Mobile) are referenced out of the Terminal version of that Utility (for instance Scenarios to perform WMS validations via MOCA). Mobile Utilities can be found in the Utilities/Base/Mobile directory.

The support for Mobile in terms of overall Test Case automation coverage is equivalent to Terminal support (both in standard Test Cases and Flow Test Cases (interacting with both Web and Mobile interfaces) and in terms of Utilities). Test Specifications are available for Inventory and Inventory Count Test Cases and others will be completed for other functional areas in a follow-on Cycle Bundle release. A Cycle playlist for the execution of all Mobile tests has been developed and is located in *“Playlists/Base/All Mobile Test Cases.cycplay”*

Cycle Bundle support for Mobile begins with **2019.1.1.12 and continues with 2020.1.1**. These versions support interaction with the Mobile Application data elements at the top of the Mobile Application screen and automation support is based on this functionality.

Requirements for Mobile testing includes having a supported, configured and enabled Blue Yonder WMS instance with Mobile Support. It also requires having the appropriate RF devices setup and configured for Mobile support (**Vendor Name: WEBMTF, Terminal Type: Handheld**). Please consult your Blue Yonder documentation for configuring Mobile support with your Blue Yonder WMS instance.

The following are newly added Cycle Bundle configuration variables relative to Mobile support:

Variable	Variable Description
mobile_ui (required)	URL to access to the Mobile Emulator.
mobile_credentials (required)	Cycle Credential to use for the Mobile App, similar to credentials for Terminal and/or Web.
mobile_devcod	Terminal ID / Device Code to use when logging into the Mobile Interface (Terminal ID / RF device must be created in your WMS instance). If not set, will default to value of <b>devcod setting</b>
mobile_start_loc	Starting location to use on the Work Information screen. If not set, will default to value of <b>start_loc setting</b> .

<b>mobile_screen_orientation_setting</b>	Valid values are <b>Portrait</b> or <b>Landscape</b> and this controls the screen orientation if you are using the <b>DEMO</b> interface (URL ends in demo instead of login and is configured for your WMS instance). Only valid when logging into the Mobile Application.
<b>mobile_screen_size_setting</b>	Valid values are <b>Tablet</b> or <b>Handheld</b> and this controls the size mode if you are using the <b>DEMO</b> interface (URL ends in demo instead of login and is configured for your WMS instance). Only valid when logging into the Mobile Application.

**NOTE:** Since both Mobile and Web support use Cycle Web steps and a standard browser (**testing with Mobile has been completed with both Chrome and Edge**), interaction with the same browser for Mobile and Web not been utilized (and not been tested). Currently the Test Case must either be logged into Mobile **OR** Web (via Bundle Utilities) but not at the same time.

## Regression Testing

### Regression Tests

Each Test Case will serve as a regression test. Data for the test is loaded through datasets and then is cleaned up following execution. The data can be reused as environment or configuration changes take place to determine if the process is still performing as expected.

**NOTE:** Test Case Inputs are provided in the Test Case Inputs/Samples directory of the Cycle Bundle. These are provided for reference **ONLY** as these are the inputs used when running the Bundle Test Cases at Tryon. Inputs will **need to updated** to reflect your WMS configuration **prior to use**.

### Playlists

Cycle includes the ability to execute Feature files in Playlists, which automate the sequential execution of multiple Test Cases with a single run.

The Playlists directory in the Cycle Bundle contains Playlists for each functional area of the warehouse. Each Playlist executes all standard Test Cases for that functional area (regardless of user interface type).

Cycle Bundle Playlists included in the Bundle are listed below:

1. **BASE-BLD.cycplay** - Pallet Building Test Cases
2. **BASE-CNT.cycplay** - Cycle Counting Test Cases
3. **BASE-INT.cycplay** - Integration Test Cases
4. **BASE-INV.cycplay** - Inventory Test Cases
5. **BASE-LDG.cycplay** - Outbound Test Cases
6. **BASE-PCK.cycplay** - Picking Test Cases
7. **BASE-PRD.cycplay** - Production and Work Orders Test Cases
8. **BASE-RCV.cycplay** - Receiving Test Cases
9. **BASE-RPL.cycplay** - Replenishment Test Cases
10. **BASE-SHP.cycplay** - Outbound Shipping Test Cases
11. **BASE-WAV.cycplay** - Waving Test Cases

12. *BASE-YRD.cycplay* - Yard Management Test Cases
13. *Core Test Cases.cycplay* - Sub-set of Test Cases from each functional area
14. *Smoke Test Cases.cycplay* - Sub-set of Tests Cased from each functional area (shorter duration than CORE)
15. *All Test Cases.cycplay* - All Test Cases
16. *All Terminal Test Cases.cycplay* - All Terminal Test Cases
17. *All Web Test Cases.cycplay* - All Web Test Cases
18. *All Mobile Test Cases.cycplay* – All Mobile Test Cases

## Test Case and Utility Details

---

Included in the directory “*Documentation/Test Specifications*” directory (*Base and Custom*) are two files that document the Test Cases and Utilities code information included in the Bundle:

- “*Test Case Details and Inputs.txt*”
- “*Utility Details and Inputs.txt*”

Both of these files are generated from the actual CycleScript code from each of the Test Case and Utility Feature files. The data is parsed from the Header and Scenario comment blocks and the documentation is generated from this information.

**NOTE:** Anyone developing new code should first look through the “*Utility Details and Inputs.txt*” file to look for Scenarios that they can re-use within their new Test Case development.

All Test Cases are listed in the “*Test Case Details and Inputs.txt*” file. Each Test Case will include a header. An example is:

```
#####
Test Case Name: BASE-RCV-0010 Terminal Inbound Receiving
Test Case File: BASE-RCV-0010 Terminal Inbound Receiving.feature
#####
```

This will allow for a search in this file for the documentation on any Test Case included in the Bundle.

Test Case documentation will include information on the following:

- Test Case Description – Short description of the Test Case
- Test Case Author – Author of the Test Case
- Test Case Type – Type of Test Case - Regression
- Functional Area – WMS functional area that the Test Case interacts with
- Blue Yonder WMS Version – Blue Yonder WMS version
- Blue Yonder Interfaces Interacted with – Terminal, MOCA, Integrator, and/or WEB
- Test Case Dataset setup scripts – Dataset Setup Script directory (relative to */Datasets/Base/*)
- Test Case Dataset cleanup scripts - Dataset Cleanup Script directory (relative to */Datasets/Base/*)
- Test Case Input CSV – Pointer to the CSV file with inputs for this Test Case
- Test Case Required Inputs – list of all the required input parameters and descriptions
- Test Case Optional Inputs - list of all the optional input parameters and descriptions
- Test Case Assumptions – Assumptions relative to this Test Case
- Test Case Notes – Additional Notes relative to this Test Case

All Utilities and their associated Utility Scenarios are listed in the “*Utility Details and Inputs.txt*” file. Each Utility will include a header. **An example is:**

```
#####  
Utility Name: Terminal Inventory Utilities  
Utility File: Terminal Inventory Utilities.feature  
#####
```

All Utility Scenarios will have a header. **An example is:**

```
#####  
Utility Scenario Name: Terminal Inventory Transfer Undirected  
#####
```

This will allow for a search in this file for the documentation on any Utility and/or Utility Scenario included in the Bundle.

The Utility Feature file documentation will include information on the following:

- **Utility Description** – Short description of the Utility
- **Utility Author** – Author of the Utility
- **Utility Type** – Type of Utility - Utility
- **Functional Area** – WMS functional area that the Utility implements Utility Scenarios for
- **Blue Yonder WMS Version** – Blue Yonder WMS version
- **Blue Yonder Interfaces Interacted with** – Terminal, MOCA, Integrator, and/or WEB
- **Public Scenarios** – A list of each of the public/callable Utility Scenarios implemented in this Utility and a description.
- **Utility Assumptions** – Assumptions relative to this Utility
- **Utility Notes** – Additional Notes about this Utility

The Utility Scenario documentation will include information on each Utility Scenario contained in the Utility Feature file:

- **Description** - Short description of the Utility Scenario
- **MSQL Files** - List of MSQL scripts used in this Utility Scenario
- **Groovy Files** - List of Groovy scripts used in this Utility Scenario (optional)
- **API Endpoints** – List of API Endpoints used in this Utility Scenario (optional)
- **Required Inputs** - List of each of the required inputs to the Utility Scenario with a description
- **Optional Inputs** - List of each of the optional inputs to the Utility Scenario with a description
- **Outputs** - all variable effected by the Utility Scenario that will be available to the Utility Scenario caller

## Test Case Specifications

Included in the directory “*Documentation/Test Specifications/Base*” directory are the following directories that mirror the Test Cases directory. These directories contain Word documents documenting Bundle Test Case Specifications:

- API
- Inbound
- Inventory

- Inventory
- Production
- Outbound
- Yard

Test Case Specifications are Word documents that detail Test Cases in Bundle and provide information on the steps a test takes during execution as well as the expected result for each step (including screen shots). They also include information about setup, cleanup, and validation approaches for each test case.

These specifications are included in Word format so that if you add new tests or change existing test cases you have these as templates for your work. Please always make **sure to keep the original version of the Test Specifications in the Base directory**.

## Using Datasets

The Datasets directory in the Cycle Bundle contains regression datasets that load and cleanup data for all included Test Cases. It bears noting that, as of the Cycle 2.0 release, there is no need to have separate load and cleanup datasets for your SQL and MSQL initialization and termination routines, unless you have to run SQL and/or MSQL scripts out of the standard order.

In the Cycle Bundle, the mechanism used to load and cleanup data is MSQL files containing a combination of MOCA and local syntax. CSV files that perform direct table inserts were not used in the Cycle Bundle, but they can be used as well.

### Load dataset

The load dataset will be named reasonably based on functional area and test (such as “count\_bulk”), and will contain an MSQL script and/or CSV files as needed. Note that the loading MSQL script must be named *load\*.msql*. There can be more than one loading MSQL in the directory, but they need to be named appropriately in order to run in the proper sequence— for example: *load\_01\_inbound\_po.msql*, *load\_02\_create\_inventory.msql*, etc. Each CSV file will be named for the table it is to be loaded into. Distributed allocation datasets in the Bundle have 2 load (*load\_01.msql*, *load\_02.msql*) MSQL files to aid with performance.

The following steps should be used to load a dataset:

```

4
5 Given I assign "count_bulk" to variable "cleanup_directory"
6 And I execute scenario "Perform MOCA Dataset"
7

```

### Cleanup dataset

The cleanup dataset will be named *cleanup\*.msql* and contained in the same directory as the loading MSQLs for purging data from the Test Case/Scenario. Cleanup files run prior to load files when invoked with “Perform MOCA Dataset”. Cleanup files run standalone when invoked with the Step “Perform MOCA Cleanup Script”. This Step will run only the cleanup scripts in the subdirectory specified in the parameter.



The following steps should be used to run a cleanup:

```
4  
5 Given I assign "count_bulk" to variable "dataset_directory"  
6 And I execute scenario "Perform MOCA Cleanup Script"  
7
```

Wherever possible, datasets should be dynamic and should honor the variables present in the Scenarios from which they are run, including those inherited from *Environment.feature*.

Note that variables are available only in MSQL scripts and not in CSV files, therefore only static data should be loaded via CSV file. It can be assumed that *Environment.feature* will export its own functional variables such as *wh\_id* via the "Set Up Environment" Scenario.

## Customizations

---

### Overview

The Bundle has been designed to allow teams to override the standard base scripts with customer specific scripts. This Feature allows teams to modify the behavior of individual utilities, MSQL commands, data loads, and data cleanups without overwriting the objects delivered with the Bundle.

**NOTE:** All customizations should be stored in the Custom directories to maintain the integrity of the Cycle Bundle and to assist in the upgrade process.

### Directory Load Path

Customizations are supported by the implementation of a directory load path. When the Bundle loads or runs logic stored outside of the test case, the Directory Load Path is used to control the sequence in which the directories are searched to find the code. All the directories in the Bundle code set contain both Base and Custom subdirectories. The default load path for the Bundle is Custom, then Base. This means the Bundle will allow look in the Custom subdirectory before the Base subdirectory. Therefore, if a script "create\_part.msql" is ran, it will be loaded from either the "Scripts/MSQL\_Files/Custom" directory or the "Scripts/MSQL\_Files/Base" directory.

### Importing Files

All Bundle test cases import utilities files that contain the shared logic for working with the WMS. Most Utility files contain many Utility Scenarios. When importing utility files, the Bundle looks for files with matching names within the subdirectories listed in the Directory Load Path. The file search is done in reverse order. This means that files in the BASE directory are loaded before the files in the Custom directory. The result of loading in this manner is that any Scenarios loaded from the Custom subdirectory will override the same named Scenario loaded from the Base directory.

To make a customization to the “Terminal Outbound Trailer Complete Stop” Scenario within the *“/Utilities/Base/Terminal/Terminal Trailer Utilities.feature”* you would create a new file in */Utilities/Custom/Terminal* named *“Terminal Trailer Utilities.feature”* containing just the customized version of “Terminal Outbound Trailer Complete Stop”

```
4
5 Given I assign "Web Utilities.feature" to variable "import_file"
6 And I execute scenario "Perform File Import"
7
```

## Scripts and Datasets

Most Bundle test cases use a combination of MSQL scripts, Groovy scripts, MOCA Datasets scripts and MOCA Cleanup scripts. When any of these script types are run from the Bundle, the Directory Load Path is used to determine the search sequence used to find the script to execute. This means that a script in the Custom subdirectory will always be found first. Only the first script file found will be used.

To make a customization to the file *“/Scripts/MSQL\_Files/Base/create\_part.msql”* you would copy the script file to *“/Scripts/MSQL\_Files/Custom”*. Apply custom changes to the new version of the script file.

```
4
5 Given I assign "clear_device_context.msql" to variable "msql_file"
6 And I execute scenario "Perform MSQL Execution"
7
```

```
4
5 Given I assign "do_something.groovy" to variable "groovy_file"
6 And I execute scenario "Perform Groovy Execution"
7
```

```
4
5 Given I assign "Pallet_Building_Undirected" to variable "dataset_directory"
6 And I execute scenario "Perform MOCA Dataset"
7
```

```
4
5 Given I assign "Pallet_Building_Undirected" to variable "cleanup_directory"
6 And I execute scenario "Perform MOCA Cleanup Script"
7
```

## Adding Additional Customization Levels

It is possible that some projects may need more than a single level of customization. For instance, a project may have specialized code that applies to all of their site, but there may also be an extra set of customizations that apply to only 1 of the sites.

Many layers of customization can easily be supported by adding the subdirectories to hold the custom code and updating the Directory Load Path. Adding the new subdirectories can be accomplished using Windows File Explorer. Be sure to add your new subdirectories to every directory that contains the Base and Custom subdirectories. Change the Directory Load Path by assigning the `directory_load_path` variable within the `environment.csv` (most specific to least specific). When adjusting the `directory_load_path` variable, be sure that "Base" is always the final value. Also note that there should NOT be spaces between the values and commas in the `directory_load_path`.

## Environment CSV File Details

This section is an extension of "Environment Setup and Initial Data Values" section above. This section details the 6 different Environment CSV Override Files in the Bundle and additional detail on each.

1. **Environment and Pipeline CI specific Override file:**
  - If the Windows Environment variable **BUNDLE\_CI\_ENVIRONMENT** is set then the value of this OS Environment Variable will be used as opposed to looking in the `Environments/Environment.csv` file (or getting from the standard variable `$environment`)
  - If **BUNDLE\_CI\_ENVIRONMENT** is set it will look for an Override files in the directory
    - `/Environments/<Environment Name>` where **<Environment Name>** is the value of the **BUNDLE\_CI\_ENVIRONMENT** environment variable
  - This Override file must have the name
    - `<Environment Name>_Environment_Override_CI.csv`
  - This file is optional
2. **Environment and Warehouse specific Override file:**
  - This file contains a list of variables and values that will only override the main environment variables when a test case is executed for the specified environment and a specified warehouse.
  - This file must be located in the directory
    - `/Environments/<Environment Name>`
  - This file must have the name
    - `<Environment Name>_Environment_Override_<Warehouse Id>.csv`
  - This file is optional

3. **Environment specific Override file:**

- This file contains a list of variables and values that will only override the main environment variables when a test case is executed for the specified environment.
- This file must be located in the directory
  - /Environments/<Environment Name>
- This file must have the name
  - <Environment Name>\_Environment\_Override.csv
- This file is optional

4. **Warehouse specific All Environments Override file:**

- This file contains a list of variables and values that will override the main environment variables when a test case is executed for any environment but for the specified warehouse.
- This file must be located in the directory
  - /Environments/
- This file must have the name
  - Environment\_Override\_<Warehouse ID>.csv
- This file is optional

5. **All Environment Override file:**

- This file contains a list of variables and values that will override the main environment variables when a test case is executed for any environment.
- This file must be located in the directory
  - /Environments/
- This file must have the name
  - Environment\_Override.csv
- This file is optional

6. **Warehouse specific Environment file:**

- This file contains a list of variable and values with will be used when a test case is executed for the specified environment and warehouse.
- This file must be located in the directory
  - /Environments/<Environment Name>
- This file must have the name
  - <Environment Name>\_Environment\_<Warehouse Id>.csv
- This file is optional

7. **Environment file:**

- This file contains a list of variable and values with will be used when a test case is executed for the specified environment.
- This file must be located in the directory
  - /Environments/<Environment Name>
- This file must have the name
  - <Environment Name>\_Environment.csv
- This file is **mandatory**

# Dynamic Data

## Overview

Almost all test cases use examples from the test case input CSV file to provide data needed to perform the test. Some tests cases need a valid value from the WMS environment to perform a test, but a specific set value is not required. For instance, a test case that tests putting a storage location into error and resetting the location probably does not require a specific storage location. Usually any storage location, not in error, will be sufficient to perform the test. For test input values like this, and any value that can be derived at the time of test execution, the Bundle provides Dynamic Data functionality.

Dynamic Data is base Bundle functionality that allows the user to specify an instruction for deriving a data value at the time of test execution instead of a specific static data value. The dynamic data value is derived at the time the examples are read from the test case input CSV, so the dynamic data value is able to be used by the test's dataset logic.

➤ Please note that the use of Dynamic Data is optional and currently not utilized by Test Cases. It is recommended to first setup test cases using specific static data as Test Case Inputs (and specific to your WMS environment). Once test cases are successfully executing, then Dynamic Data can be implemented.

## Dynamic Data Setup

Dynamic Data instructions are setup in CSV files located within the Data/Dynamic Data directory. When Cycle searches for a dynamic data instruction it will read the CSV files located in the Dynamic Data directory *in the directory\_load\_path sequence*.

Dynamic Data CSV files must contain all of the following fields in the exact order shown below:

1. **value:** Key value, always starting with "?", that identifies the dynamic data instruction. This is the value that will be used in the Test Case Input CSV file.
1. **instruction\_type:** Defines the type of logic used to determine the value
2. **instruction:** Specific instruction used by Cycle to determine the value. The exact use varies by instruction type.
3. **where\_clause:** SQL "where" clause that can be reference from within MSQl scripts. This can include single column assignment or entire sub-queries.
4. **order\_by\_clause:** SQL "order by" clause that can be referenced from within MSQl scripts
5. **returned\_fields:** List of fields returned by the instruction
6. **retry\_value:** Reference to an additional dynamic data value that will be executed if the current instruction fails to return a value.
7. **comment:** Field used document the dynamic data instruction and usage.

Below is a sample Dynamic Data CSV fie:

```
value,instruction_type,instruction,where_clause,order_by_clause,return_fields,retry_value,comment
?prtnum:any,MSQl,dynamic data/get_any_part.msql,,prtnum,"prtnum,prt_client_id",,Returns any part and part client id
?prtnum:prompt,Prompt-String,Enter Part Number,,,prtnum,,Prompt user for Part
?lotnum:any,MSQl,dynamic data/get_any_lot.msql,,,lotnum,,Return a lot number for any lot
```

## Instruction Types

The Bundle supports many different methods of deriving dynamic data values. Each method is defined as an *Instruction Type*. The instruction types currently support include:

1. **Scenario:** Cycle will execute a CycleScript Scenario to determine the data value
  - o **Instruction:** The Scenario name that Cycle will execute
2. **MSQL:** Cycle will execute an MSQL script to determine the data value
  - o **Instruction:** The MSQL File that Cycle will execute
3. **SQL:** Cycle will execute a SQL script to determine the data value
  - o **Instruction:** The SQL File that Cycle will execute
4. **Prompt-String:** Cycle will prompt the user for a String value
  - o **Instruction:** The prompt to be displayed to the user
5. **Prompt-Integer:** Cycle will prompt the user for an Integer value
  - o **Instruction:** The prompt to be displayed to the user

## Enabling Dynamic Data

The Dynamic Data functionality within the Bundle can be switched on and off by adjusting the “dynamic\_data” Cycle variable. The variable must be set to TRUE to use Dynamic Data. The variable can be set for a single test by defining the value in the Test Case Input example or the variable can be set globally by defining the variable in the environment CSV files.

## Using Dynamic Data in Test Case Input

Once Dynamic Data is enabled, using a Dynamic Data value as part of Test Case Input is very simple. To use a Dynamic Data instruction instead of a set value for test, modify the Test Case Input csv by specifying the Dynamic Data value key.

The below example shows a Test Case Input file changed to use Dynamic Data. In the example, Dynamic Data is used to select a part number and to a specific the pallet quantity for the part number selected.

### **Original Input CSV:**

```
wh_id,prtnum,untqty,stoloc  
WMD1,SHAMPOO,48,G2-230-A
```

### **Updated Input CSV with Dynamic Data:**

```
wh_id,prtnum,untqty,stoloc,dynamic_data
```

```
WMD1,?prtnum:any,?untqty:pallet_qty_for_part,G2-230-A,TRUE
```

It is important to understand that the Dynamic Data values are processed from left to right within the Test Case Input CSV. This means that if a Dynamic Data instruction is dependent on a value already being set, and that needed value is also being selected using Dynamic Data, then the needed value must be listed before (to the left within the csv) the Dynamic data that requires the value. However, any values that are specified in the Test Case Input CSV are set before the Dynamic Data processing is executed, so those values can be used by all Dynamic Data instructions.

## Setting Test Input Values

The purpose behind using Dynamic Data instructions is to derive and set a value as a test case input. The Dynamic Data functionality handles returning values from a Scenario, MSQL, SQL, or Prompt so that the value is available for the test case in exactly the same manner as if a data value was specifically set. How the data is returned varies by instruction type:

1. **Scenario:** Cycle will use the first field value in the “returned\_fields” list as the data value. The value will be assigned to a Cycle variable with a name based on the Test Case Input field that invoked the Dynamic Data Instruction. If a MOCA connection is established, then a MOCA environment variable is also created.
2. **MSQL and SQL:** Cycle will use the first field value in the “returned\_fields” list to read data out of the first row of the current MOCA or SQL dataset. The value read from the dataset will be assigned to a Cycle variable with a name based on the Test Case Input field that invoked the Dynamic Data Instruction. If a MOCA connection is established, then a MOCA environment variable is also created.
3. **Prompt-String and Prompt-Integer:** Cycle will store the user’s response in a Cycle variable with a name based on the Test Case Input field that invoked the Dynamic Data Instruction. If a MOCA connection is established, then a MOCA environment variable is also created.

## Returning Multiple Values

In many situations it is useful to get more than one field value returned from a Dynamic Data Instruction. For example, when an instruction is finding a part number in the system, it is also useful to return the part’s client id. Dynamic Data supports returning multiple values from a Dynamic Data instruction by using the “returned\_fields” setting.

The “returned\_fields” setting is a comma separated list of field names that are expected to be returned from the call to an instruction. After the instruction is run, Cycle reads the “returned\_fields” list to determine what values should all be assigned to Cycle variables. The first value in the “returned\_fields” list is always assigned to a Cycle variable with a name based on the Test Case Input field that invoked the Dynamic Data Instruction. All subsequent fields are assigned to variables based on the field name in the “returned\_fields” list.

When using a Dynamic Data Instruction that returns multiple field values, usually you will have Test Case Input fields who’s value will be derived by an another field’s Dynamic Data Instruction. When this happens, you can leave the field blank in the test case inputs. Note that if a field is returned by more than one Dynamic Data Instruction, the last value derived will be the value that is set for the test.

## Retry Logic

When using Dynamic Data to select input values for a test case there may be situations in which there is more than one way to find a usable value. The Dynamic Data logic supports this need with the “retry\_value” field.

The “retry\_value” field can be used to specify the “value” field on another Dynamic data. When Dynamic Data is processed, the instruction for the dynamic data is run. If the instruction fails, then Cycle will evaluate if a “retry\_value” was defined. If a “retry\_value” was defined, then Cycle will find the definition of the new value and execute the next Dynamic Data instruction. This process will be repeated until either the Dynamic Data Instruction passes or there is no “retry\_value”

As example of how the retry logic might be use, consider the following Scenario. For a test case input we want Cycle to select a dock door for shipping. The preference is to select an empty dock door in the Shipping Dock area. However, if there are no doors available, then the test can use a door from the shared Shipping/Receiving Dock area. The following examples shows how this would look. Noticed that the “retry\_value” field in the first entry matches the “value” field in the second entry:

**value,instruction\_type,instruction,where\_clause,order\_by\_clause,return\_fields,retry\_value,comment**

?stoloc:ship\_door,MSQL,dynamic data/get\_shipping\_door.msql,,,stoloc, ?stoloc\_shipcecv\_door>Returns a shipping door location

?stoloc:shiprecv\_door,MSQL,dynamic data/get\_ship\_recv\_door.msql,,,stoloc,,Returns a door location from shipping/receiving area

## Examples of Dynamic Data Instructions

The following list shows examples of actual Dynamic Data Instructions. For each instruction type a CSV file entry that defines the instruction is shown followed by a table showing the same values in an easier to read format. The CycleScript and MSQL scripts to support the Dynamic Data Instructions are also shown.

1. **Scenario:** This instruction returns a value read from a CSV file.

**value,instruction\_type,instruction,where\_clause,order\_by\_clause,return\_fields,retry\_value,comment**

?ordnum:sample\_csv,Scenario,Get Value from CSV for Dynamic Data,data/Samples/sequence\_from\_file.csv,,next\_csv\_value, ,Returns the next value from the sequence\_from\_file.csv file

Column	Setting
value	?ordnum:sample_csv
instruction_type	Scenario
instruction	Get Value from CSV for Dynamic Data
where_clause	data/Samples/sequence_from_file.csv
order_by_clause	
return_fields	next_csv_value
retry_value	
comment	Returns the next value from the sequence_from_file.csv file



```

@wip @public
Scenario: Get Value from CSV for Dynamic Data
#####
# Description:
# This function will pull the next value from a single column
# CSV file. The row taken is removed.
# MSQL Files:
# None
# Inputs:
# Required:
#     where_clause - path and name of CSV file
# Optional:
#     None
# Outputs:
#     next_csv_value - value read from the csv
#####

Given I "read the next line from the file and assign it to value"
    Given I assign variable "csv_file" by combining $where_clause
    Then I verify file $csv_file exists
    Then I remove next value from $csv_file and assign to variable "next_csv_value"

```

2. **MSQL and SQL:** This instruction reads any valid part number from the WMS and returns the part and associated client id.

value,instruction\_type,instruction,where\_clause,order\_by\_clause,return\_fields,retry\_value,comment

?prtnum:lot\_controlled,MSQL,dynamic data/get\_any\_part.msql,lotflg=1 ,prtnum,"prtnum,prt\_client\_id",Returns any lot controlled part and prt\_client\_id

Column	Setting
value	?prtnum:lot_controlled
instruction_type	MSQL
instruction	dynamic data/get_any_part.msql

where_clause	lotflg=1
order_by_clause	prtnum
return_fields	prtnum,prt_client_id
retry_value	
comment	Returns any lot-controlled part and prt_client_id

```

1  /*
2  MSQL File: get_any_part.msql
3  Description:
4  Get a part (production and receivable) for specified warehouse and Client Id.  Built for use with Dynamic Data logic.
5  */
6  publish data
7  where wh_id = $wh_id
8      and prt_client_id = nvl($prt_client_id, nvl($client_id,'----'))
9      and dynamic_where = nvl($where_clause, "1 = 1")
10     and dynamic_order_by = nvl($order_by_clause, "1")
11     and dynamic_rownum = nvl($rownum, 1)
12 |
13 [select *
14     from prtmst
15     where wh_id_tmpl = @wh_id
16         and prt_client_id = @prt_client_id
17         and rcvflg = 1
18         and @dynamic_where:RAW
19         and rownum = @dynamic_rownum
20     order by @dynamic_order_by:RAW]

```

3. **Prompt-String and Prompt-Integer:** This instruction prompts the user to enter a part number.

value,instruction\_type,instruction,where\_clause,order\_by\_clause,return\_fields,retry\_value,comment  
?prtnum:prompt,Prompt-String,Enter Part Number,,,prtnum,,Prompt user for Part

Column	Setting
value	?prtnum:prompt
instruction_type	Prompt-String
instruction	Enter Part Number
where_clause	
order_by_clause	
return_fields	prtnum
retry_value	
comment	Prompt user for Part

# Pre and Post Validations

---

## Overview

Almost all test cases perform validation logic to confirm the results. Validations are used both to confirm an automated test completed all the required actions and to confirm that the responses of the system to the actions performed match the expected outcomes.

The test cases included in the Bundle contain some basic validations to confirm the test case ran as expected. However, the expected response to a test case's actions could vary greatly from system to system. For this reason, the definition and setup of validations is part of setting up a test case.

In addition to validations running after a test completes, it is sometime advantageous to run validations at the beginning of a test execution to confirm the system and data are properly setup for a test case before having Cycle execute a long running test case.

The Bundle includes functionality to add validations at the begin of a test and at the end of a test without requiring changes to be made to the base test case feature.

Pre and Post Validations are base Bundle functionality that allows the user to specify any number of validations to be performed at the time of test execution. Any number of both Pre-Validations and Post-Validations can be added.

➤ *Please note that the use of Pre and Post Validations is optional and currently not utilized by Test Cases.*

## Enabling Validations

The Validation functionality within the Bundle can be switched on and off by adjusting the “pre\_validations” and “post\_validations” Cycle variables. The variables must be set to **TRUE** to use the validation type. The variables can be set for a single test by defining the values in the Test Case Input example or the variables can be set globally by defining the variables in the environment CSV files.

## Validation Setup

Validations are setup in CSV files located within the Test Case Validations directory. The validation files need to be named using the test case's Short Code post fixed with “-Validatins.csv”. For example, the validation file for test case **BASE-INV-1050 Web Inventory Move.feature** would be in **PROJECT/Test Case Validations/BASE-INV-1050-Validations.csv**. When Cycle searches for a validation file it will search the CSV files located in the Test Case Validation directory *in the directory\_load\_path sequence*. Only the first matching file is used.

Validation CSV files contain all of the following fields:

1. **type:** Defines the type of validation.
  - PRE – Validation is performed at the start of the test.
  - POST – Validation is performed at the end of the test.
1. **instruction\_type:** Defines the type of logic used to determine the value
  - Scenario – CycleScript

- MSQL – MOCA Script
  - SQL – SQL Script
  - Prompt-String – Question prompted to user. Users response must match parameter\_1
  - Prompt-Integer - Question prompted to user. Users response must match parameter\_1
2. **instruction:** Specific instruction used by Cycle to perform determine the value. The exact use varies by instruction type.
    - Scenario – Cycle Scenario to be run
    - MSQL – MOCA Script to run
    - SQL – SQL Script to be run
    - Prompt-String – Question to be displayed to the user
    - Prompt-Integer - Question to be displayed to the user
  3. **parameter\_1:** Cycle variable to be used as input by the instruction. (optional)
  4. **parameter\_n:** Cycle variable to be used as input by the instruction where *n* is an increasing number (optional)

Below is a sample Validation CSV file:

```
type,instruction_type,instruction,parameter_1,parameter_2,parameter_3,parameter_4, parameter_5,parameter_6
PRE,MSQL,Validations/validate_amount_of_part_in_location.msql,$stoloc,$prtnum,500, , ,
PRE,MSQL,Validations/validate_location_contains_inventory_id.msql,$stoloc,CYC_EA_LOD5,,,,
POST,MSQL,Validations/validate_location_contains_inventory_id.msql,$stoloc,$prtnum,,,,
POST,Scenario,Validate Transaction was Created,INV-ADJ,$prtnum,,,,
```

## Input Parameters for Validations

Most validation logic cannot be performed without data being provided by the test case. The Bundle Validation logic provides a way to pass data from the test case being executed to the validation logic. The data passed may be a specific value or it may be a value stored in a Cycle variable.

Parameters for a Validation are defined in the test case validation file. Columns **parameter\_1** through **parameter\_n** can be defined in the CSV. Before the validation runs, Cycle will assign all the “parameter\_n” variables. If the value configured in the file begins with \$, then Cycle assumes the value is a Cycle variable and the value stored in the Cycle variable is assigned to the \$parameter\_n Cycle variable. If the value configured does not start with a \$, then the value configured is assigned directly to the \$parameter\_n Cycle variable.

For example, given the following Validation configuration:

```
type,instruction_type,instruction,parameter_1,parameter_2,parameter_3
PRE,MSQL,Validations/validate_amount_of_part_in_location.msql,$stoloc,$prtnum,500
```

Before the validation is performed the following parameters will be assigned.

1. **parameter\_1** – Assigned to the value stored in the \$stoloc Cycle variable

2. **parameter\_2** — Assigned to the value stored in the \$prtnum Cycle variable
3. **parameter\_3** — Assigned to the value 500

Many test cases will have more than one example in the Test Case Input CSV file. The expected result of a test case will often be different for each example in the input file. The Bundle Validation logic can support the varying results by adding a field to the Test Case Inputs to define the expected value for each example. The field added to the Test Case Inputs can then be used as one of the validation parameters fields.

For example, in the following Test Case Input CSV file the last field (final\_quantity) was added so that the value could be passed as parameter\_3 to the validation.

**Test Case Input CSV File:**

```
wh_id,prtnum,untqty,stoloc,final_quantity
WMD1,SHAMPOO,48,G2-230-A,96
WMD1,SOAP,100,G5-140-B,150
WMD1,TOOTHPASTE,500,E2-940-F,550
```

**Test Case Validation CSV File:**

```
type,instruction_type,instruction,parameter_1,parameter_2,parameter_3
POST,MSQL,Validations/validate_location_contains_inventory_id.msql,$stoloc,$prtnum,$final_quantity
```

## Validation Results

After running a validation instruction, the Bundle will evaluate the results to determine if the validation passed or failed. This evaluation done for each instruction type is as follows:

1. **Scenario:**
  - If the execution of the scenario fails, the validation fails
  - If the variable “validation\_status” is not set to TRUE, the validation fails. An error message can be returned in the \$error\_message variable.
2. **MSQL:**
  - If the execution of the MSQL does return a MOCA status of 0, the validation fails
  - If the field “validation\_status” within the returned dataset is not set to TRUE, the validation fails. An error message can be returned in the error\_message field of the dataset.
3. **SQL:**
  - If the execution of the SQL does not return a row count greater than 0, the validation fails.
4. **Prompt-String:**
  - If the text value enter by the user does not match parameter\_1, the validation fails
5. **Prompt-Integer:** Cycle will prompt the user for an Integer value
  - If the integer value enter by the user does not match parameter\_1, the validation fails

## Validation Example

The Bundle's prebuilt validation components are stored in the following locations

1. Pre Validation Scenarios - *project*/Utilities/Base/Pre Validation Utilities.feature
2. Post Validation Scenarios - *project*/Utilities/Base/Post Validation Utilities.feature
3. MSQL Files - *project*/Scripts/MSQL\_Files/Base/Validations

Below is an example of a Scenario validation instruction and an MSQL validation instruction.

```
Scenario: Validate Variable Value
#####
# Description: Compare two values and fail if not equal
# MSQL Files:
#   None
# Inputs:
#   Required:
#     parameter_1 - first value
#     parameter_2 - second value
#   Optional:
#     None
# Outputs:
#   None
#####

If I verify variable $parameter_1 is equal to variable $parameter_2
Else I assign variable "error_message" by combining "ERROR: Validation failed as
value (" $parameter_1 ") is not equal to (" $parameter_2 ")"
    Then I fail step with error message $error_message
EndIf
```

```

1  /*
2  MSQL File: validate_location_is_empty.msql
3  Description: This script confirms that a location is empty and there is no pending inventory.
4  This MSQL is designed to be used with the validation logic.
5  Inputs:
6      Required:
7          wh_id - Warehouse Id
8          parameter_1 - Location (parameter_1)
9      Optional:
10         None
11  Outputs:
12      validation_status - TRUE or FALSE
13      error_message - Error Message set when validation fails
14  */
15  publish data
16      where wh_id = $wh_id
17      and stoloc = $parameter_1
18  |
19  [select nvl(sum(untqty), 0) untqty
20      from inventory_view
21      where wh_id = @wh_id
22      and stoloc = @stoloc]
23  |
24  if (@untqty = 0)
25  {
26      [select nvl(sum(i.pndqty), 0) pndqty
27          from invsum i
28          where i.wh_id = @wh_id
29          and i.stoloc = @stoloc]
30  }
31  |
32  if (@pndqty = 0 and @untqty = 0)
33  {
34      publish data
35          where validation_status = 'TRUE'
36  }
37  else if (@untqty > 0 )
38  {
39      publish data
40          where validation_status = 'FALSE'
41          and error_message = "Location " || @stoloc || " is not empty"
42  }
43  else if (@pndqty > 0)
44  {
45      publish data
46          where validation_status = 'FALSE'
47          and error_message = "Location " || @stoloc || " has pending quantity"
48  }

```

## Validation of Integrator Transactions

Many test cases will trigger transactions to be sent to another system. The information sent is usually a key test result that tester want to validate. The Bundle's Post Validation functionality contains a set of pre-written Scenarios to perform this type of integrator transaction validation.

The following list describes each of the Post Validation Scenarios:

1. **Validate Transaction was Created** – This Scenario will confirm a transaction was created.
  - **parameter\_1** – Event Id
  - **parameter\_2** – Event Argument Data value
  - **parameter\_3** – Destination System (optional)
2. **Validate Transaction was Sent** – This Scenario will confirm a transaction was sent.
  - **parameter\_1** – Event Id
  - **parameter\_2** – Event Argument Data value
  - **parameter\_3** – Destination System (optional)
3. **Validate Transaction Field Value in XML** – This Scenario will confirm a field value within a transaction.
  - **parameter\_1** – XML tag for the field
  - **parameter\_2** – Expected value
  - **evt\_xml** – The XML representation of the transaction. This value is normally provided by configuring the “Validate Transaction was Created” validation to run before this validation runs.
4. **Validate Transaction Field Value List in XML** - This Scenario will confirm a field value of multiple fields within a transaction. The field and value are provided in sequential parameters.
  - **parameter\_1** – XML tag for the first field
  - **parameter\_2** – Expected value for the first field
  - **parameter\_3** – XML tag for the second field
  - **parameter\_4** – Expected value for the second field
  - **parameter\_n** – XML tag for the nth field
  - **parameter\_n+1** – Expected value for the nth field
  - **evt\_xml** – The XML representation of the transaction. This value is normally provided by configuring the “Validate Transaction was Created” validation to run before this validation runs.

Below is an example of a Test Case Validation CSV file that performs integrator transaction validations:

```
type,instruction_type,instruction,parameter_1,parameter_2,parameter_3,parameter_4, parameter_5,parameter_6
POST,Scenario,Validate Transaction was Created,INV-ADJ,$prtnum,,
POST,Scenario,Validate Transaction was Sent,INV-ADJ,$prtnum,,
POST,Scenario,Validate Transaction was Created,INV-ADJ,$prtnum,HOST,
POST,Scenario,Validate Transaction Field Value in XML,STKUOM,EA,,
POST,Scenario,Validate Transaction Field Value in XML,HLD_FLG,0,,
```



POST,Scenario,Validate Transaction Field Value List in XML,STKUOM,\$exp\_uom,HLD\_FLG,0

## Tag Usage

Scenarios in a Feature file should be tagged for the following purposes:

1. **@<shortcode>**: Use the “@shortcode” tag for every Test Case Scenario.
2. **@wip**: Use the “@wip” tag for any Utility Scenario.
3. **@public**: Any Scenarios which are intended to be called by other Test Cases or Utility Scenarios.
4. **@private**: Any Scenarios intended only to be called by other Utility Scenarios within the same Utility file.
5. **Other**: Other tag use is allowed, but other tags should not interfere with the above.

## Standard Wait Time Variables

The “Set Up Environment” Scenario in *Environment.feature* establishes a set of standard time variables. Those time variable values may be overridden in the environment setup if the common default times are too generous or lean.

The wait times are read in from a “*Wait Times.csv*” file located in the “Environments” subdirectory. The list of standard variables and accompanying values are as follows.

### Wait/Within Time Values

Variable Name	Value in Seconds
wait_short	1
wait_med	5
wait_long	10
max_response	30
screen_wait	2.5

Any wait time longer than 30 seconds should be manually coded. In cases where a “within” time longer than 10 seconds is needed, consider using a “Once” Step (e.g., during login).