



**stonebranch**

Universal Controller 7.2.x

Universal Extension

© 2022 by Stonebranch, Inc. All Rights Reserved.

1. Universal Extension .....	3
1.1 Universal Extension for Universal Controller .....	4
1.1.1 Creating a Universal Extension .....	11
1.2 Universal Extension for Universal Agent .....	12
1.3 How-To/Getting Started for Universal Extension Development .....	29
1.3.1 Introduction .....	30
1.3.2 Development Environment Set-Up .....	32
1.3.3 Universal Extension API .....	34
1.3.4 Task Entry Point .....	48
1.3.5 Dynamic Choice Field .....	77
1.3.6 Dynamic Update and Output Only Fields .....	86
1.3.7 Dynamic Command .....	94
1.3.8 In-Process Dynamic Commands .....	103
1.3.9 Cancel Command .....	118
1.3.10 Publishing Events .....	128
1.3.11 Troubleshooting and Debugging .....	148
1.4 Universal Extension 1.0.0 API .....	153
1.5 Universal Extension 1.1.0 API .....	157
1.5.1 UniversalExtension Class (1.1.0) .....	158
1.5.2 ExtensionResult Class (1.1.0) .....	159
1.5.3 ExtensionLogger Class .....	162
1.5.4 Universal Extension Decorators (1.1.0) .....	163
1.6 Universal Extension 1.2.0 API .....	164
1.6.1 UniversalExtension Class .....	165
1.6.2 ExtensionResult Class .....	167
1.6.3 Universal Extension Decorators .....	170
1.6.4 Logging Module .....	171
1.6.5 Event Module .....	172
1.6.6 UI Module .....	173

# Universal Extension



## Universal Controller

---

[Universal Extension for Universal Controller](#)

[Creating a Universal Extension](#)



## Getting Started

---

[How-To/Getting Started for Universal Extension Development](#)



## Universal Agent

---

[Universal Extension for Universal Agent](#)



## API

---

[Universal Extension 1.0.0 API](#)

[Universal Extension 1.1.0 API](#)

[Universal Extension 1.2.0 API](#)



The information on these pages also is located in the [Universal Controller 7.2.x Universal Extension.pdf](#).

# Universal Extension for Universal Controller

- [Summary](#)
- [Agent Registration](#)
  - [Universal Extension Deployment](#)
- [Universal Template/Extension Definition](#)
  - [Output Only Field](#)
  - [Text Field Text Type](#)
  - [Dynamic Choice Field](#)
  - [Dynamic Commands](#)
  - [Universal Output](#)
  - [Python Application Attachment](#)
- [Import/Export Template](#)
  - [Release Levels](#)
- [List Import/Export](#)
- [Log Level](#)

## Summary

The Universal Extension developer will create a new *Universal Extension-based* [Universal Template](#), declaring:

- Task fields required for defining and launching the process through the Universal Extension.
- Task Instance fields required for capturing data coming back throughout the life-cycle of the Universal Extension process.
- Task Instance commands, and their supported status(es), that can be executed through the Universal Extension.

The Universal Extension developer will then upload/attach an **Extension Archive** zip file to the Universal Template, recognized by the Universal Agent, which can either be deployed on-demand to registered agents, or automatically following a successful agent registration.

Once the Universal Template is created, and the **Extension Archive** zip file is attached, the Universal Extension developer can generate a zip package, recognized by the Universal Controller, containing both the Universal Template definition and the **Extension Archive** zip file recognized by the Universal Agent.

To create a Universal Extension, see [Creating a Universal Template](#).

## Agent Registration

All communication between the Universal Controller and the Universal Extension is managed through UAG; therefore, any registering agent that meets the release criteria can be a Universal Extension provider.

In order for the Universal Controller to know which Universal Extensions are supported by a registering agent, each Agent will report this information in the hello message handshake.

## Universal Extension Deployment

Deployment of Universal Extensions is managed by the Universal Controller.

Any Universal Agent registering into the Universal Controller can automatically accept deployment of available Universal Extensions.

**Note**

It is important to review the Universal Agent configuration option, [EXTENSION\\_ACCEPT\\_LIST](#), if you want to prohibit an agent from accepting deployment of any extension or if you want to limit deployment to specific extensions.

This allows full customization of the deployment approach, allowing agent extension configuration to be tailored based on corporate policy.

For automatic deployment of Universal Extensions to all registered agents, ensure that your agents are all installed with an **accept any extension** (default) configuration.

## On-Registration Extension Deployment

On registration deployment of accepted Universal Extensions will commence immediately upon successful agent registration, depending on the configuration of the [EXTENSION\\_DEPLOY\\_ON\\_REGISTRATION](#) UAG configuration option.

## On-Demand Extension Deployment

For Agents that are configured to accept extensions, but are not configured for on-registration deployment, their accepted extensions will be deployed only on-demand at task instance run time.

If on-demand deployment of an extension is required, the task instance will transition to the **Queued Status** with the *Deploy ExtensionStatus Attribute*.

Upon successful deployment, the *Deploy ExtensionStatus Attribute* will be cleared and the task instance will transition to the **Running Status**.

If the deployment is unsuccessful, the *Deploy ExtensionStatus Attribute* will be cleared and the task instance will transition to the **Start FailureStatus**.

If an extension cannot be deployed to the destination agent due to the extension not being an accepted extension, the task instance will transition to **Undeliverable Status**.

Once the agent configuration has been updated, and the agent restarted, on-demand deployment can commence and the task instance will transition to **Running Status** upon successful deployment, as described above.

## Universal Template/Extension Definition

The Universal Template definition will be enhanced to allow differentiating between a Universal Task that is executed by **UAG** in the form of a **Script**, and a Universal Task that is executed by a Universal **Extension**.

Under the **Universal Template Details** section, the user should still be able to refine the **Agent Type** (Any, Linux/Unix, or Windows), however, it can no longer be assumed that the **Script** related fields (**Use Common Script**, **Linux/Unix Script**, **Windows Script**, and **Windows Script File Type**) are applicable.

A new **Template Type** field, with options **Script** and **Extension**, will be introduced to distinguish between *Script-based* Universal Templates and *Universal Extension-based* Universal Templates, respectively.

The following fields, under their applicable sections, will be hidden when the **Template Type** is **Extension**.

- Universal Template Details
  - Use Common Script
  - Linux/Unix Script
  - Windows Script
  - Windows Script File Type

## Output Only Field

A Universal Extension may need to send back runtime attributes associated with a task instance.

These fields are not applicable at definition time; therefore, when specifying a Universal Template Field of type **Text**, **Integer**, **Float**, or **Boolean**, a **Restriction** option was added where you can specify **No Restriction** or **Output Only**.

**Output Only** fields will be rendered as read-only on the Universal Task Instance form, and not shown on the Universal Task form.

## Output Only Field Validation

<b>Text</b>	<ul style="list-style-type: none"> <li>The value cannot be more than 255 characters for text fields and 4000 characters for large text fields.</li> <li>The following warning will be logged in the uc.log if the output field update is rejected by the controller. <ul style="list-style-type: none"> <li>Output Fields: Field value for field with template id <i>uuid</i> and field name "<i>field-name</i>" cannot be more than [255 4000] characters. <i>{field-value}</i></li> </ul> </li> </ul>
<b>Integer</b>	<ul style="list-style-type: none"> <li>The value must be an integer between -2147483648 and 2147483647, inclusive.</li> <li>The following warning will be logged in the uc.log if the output field update is rejected by the controller. <ul style="list-style-type: none"> <li>Output Fields: Field value "<i>field-value</i>" for field with template id <i>uuid</i> and field name "<i>field-name</i>" is not a valid integer.</li> </ul> </li> </ul>
<b>Float</b>	<ul style="list-style-type: none"> <li>The value must be a valid double-precision float.</li> <li>The following warning will be logged in the uc.log if the output field update is rejected by the controller. <ul style="list-style-type: none"> <li>Output Fields: Field value "<i>field-value</i>" for field with template id <i>uuid</i> and field name "<i>field-name</i>" is not a valid float.</li> </ul> </li> </ul>
<b>Boolean</b>	<ul style="list-style-type: none"> <li>Any value other than <b>true</b> will be evaluated as <b>false</b>.</li> </ul>

## Preserve Output On Re-run

When **Restriction** is **Output Only**, an option to **Preserve Output On Re-run** can be specified.

On task instance **Re-run**, all **Output Only** field values are cleared, by default. To change this behavior, on a per field basis, enable the **Preserve Output On Re-run** option.

## Text Field Text Type

For Extension-based Universal Templates only, a **Text Type** option will be introduced for Universal Template Fields of type **Text**.

This allows for the template administrator to designate a specific content type for a **Text** field, with the following content types currently supported.

- Plain** (default)
- JSON**
- YAML**

When **JSON**, or **YAML** is selected as a **Text Type**, the following applies.

- The field value must be parsable at definition time, or at runtime, if the field contains an unresolved variable or function.
- The field value will be passed to the Universal Extension as its designated type.

## Dynamic Choice Field

**Choice** field types need to support dynamic, Universal Extension-derived options.

To accommodate this, *Universal Extension-based* Universal Templates will introduce a new **Dynamic Choice** boolean option on the **Universal Template Field** form, when the field type is **Choice**.

When the **Dynamic Choice** option is enabled, the template administrator will no longer be able to define static **Choices**.

It is feasible that the dynamic options depend on the value of one or more Universal Template Fields; therefore, a template administrator also is able to specify dependent fields through a new **Dependent Fields** option.

From the Universal Task form, dynamic choice field options will be populated through a request/response mechanism, initiated by Universal Controller to the Universal Extension.

## Dynamic Commands

A Universal Extension may support additional operations against a task instance, therefore, we need to allow for defining such operations, which in the Universal Controller we refer to as commands.

A new **Commands** tab on the Universal Template will become available if and only if the **Template Type** is **Extension**.

To define a command extension, the command definition will require the following fields.

<b>Name</b>	Unique command name, adhering to the same naming convention as a Universal Template field name.
<b>Label</b>	User friendly display name for the command, to be displayed within the client.
<b>Supported Status (es)</b>	Specifies the task instance status (or statuses) that the dynamic command should be enabled for.
<b>Dependent Fields</b>	The administrator can select zero or more Universal Template fields that are required by the command. The values of those fields will be included in the command request.
<b>Timeout</b>	<p>Specifies an optional command timeout, in seconds, if the command requires longer than the System-level default of 60 seconds.</p> <p>If the Controller (server) does not receive a command response from the <b>Extension</b> prior to the timeout being reached, a timeout message will be sent to the client (user interface), and displayed in the <a href="#">Console</a>:</p> <p>Command "<i>command-name</i>" on task instance "<i>instance-name</i>" with id <i>instance-uuid</i> timed out.</p>
<b>Execution Option</b>	Specification for whether the command runs out-of-process execution or in-process execution.
<b>Asynchronous</b>	If <b>Execution Option</b> is in-process; Specification for whether the command runs synchronously or asynchronously.

## Command Permission

Users must have **Universal** (or **ALL**) command permission and **Read** permission for the Universal Task Instance, assigned by the **Task Instance** permission type, for authorization to execute a Universal dynamic command.

## Universal Output

To allow for a *Universal Extension-based* Universal Task to contribute its own unique output upon job completion, the Controller added support for an EXTENSION output type that the Universal Extension can *optionally* return upon job completion.

## Python Application Attachment

The Python Application implementing the Universal Extension must be packaged in a zip file, containing both an extension.py and an extension.yml.

The extension.yml is the YAML metadata configuration file, as shown below.

```

---
extension:
  name: extension1
  version: "0.1.0"
  api_level: "1.0.0"
  requires_python: ">=2.9"
  python_extra_paths: "${extension_zip}/lib:${extension_zip}/test/lib"
owner:
  name: John Doe
  organization: Stonebranch, Inc.
comments: |
  These comments will appear in the Universal Template 'Extension Comments' field.
  The 'Extension Comments' field can be viewed from the Meta Data section, the List, or the Show Details.

```

The extension name and `api_level` are required, while the extension `requires_python` (default "`>=2.6`") and `python_extra_paths` are optional.

#### Note

For `requires_python`, you can use wildcards to select certain Python versions.

For example, the following configurations are supported:

- `requires_python: ">=2.7"`
- `requires_python: "!=3.9.1"`
- `requires_python: "!=3"`

The owner name and `organization` are optional.

From the Universal Template form, the zip will be uploaded by clicking the add **[+]** icon (tooltip **Upload Extension Archive**) next to the **Extension** field, and persisted in the Universal Controller database as a byte[] (BLOB).

To download an already attached Extension Archive, click the link **[ ]** icon (tooltip **Download Extension Archive**).


To remove an already attached Extension Archive, click the remove **[-]** icon (tooltip **Delete Extension Archive**).

During the upload process, the `extension.yml` metadata will be parsed and made available from the Universal Template [Metadata](#) fields.

## Import/Export Template

The Import/Export Template feature supports importing/exporting a Universal Template as a zip file.

The Universal Template zip file includes the following entries:

File Name	Description	Optional
template.json	The Universal Template definition in JSON format.	No
template_icon.png	The Universal Template <b>Icon</b> in PNG format.  Note  Icon metadata will be set as attributes in the Universal Template JSON.	Yes

extension_archive.zip	The Universal Template <b>Extension Archive</b> in ZIP format.	Yes
-----------------------	--	-----

<b>Export</b>	To export an existing Universal Template as a zip file, click the <b>Export Template</b> button in the Universal Template Details. (You can also click <b>Export Template</b> in the <a href="#">Action menu</a> that displays for that Universal Template record.)  The exported Universal Template has the following filename format: <code>unv-tmpl-<b>UniversalTemplateName</b>-<b>extensionVersion</b>.zip</code>
<b>Import</b>	To import a Universal Template zip file, click the <b>Import Template...</b> button on the Universal Templates list.

## Release Levels

Export Template sets the following release level attributes in the Universal Template JSON:

Attribute	Description
minReleaseLevel	The minimum Universal Controller release level required to import the Universal Template.  "minReleaseLevel" : "7.0.0.0"
exportReleaseLevel	The release level of the Universal Controller that the Universal Template was exported from.  "exportReleaseLevel" : "7.0.0.0"

Import Template validation prevents importing a Universal Template if the Universal Controller does not meet the minimum release level requirement:

*Cluster Node release level is 7.0.0.0, which does not meet the minimum release level of 7.0.0.1 for the Template.*

## List Import/Export

The List Import/Export feature continues to support exporting the Universal Task and Universal Template, as it has in previously releases.

Comparable to the Universal Template **Icon**, the **Extension Archive** will be encoded in the XML as base64.

List Import validation prevents an extension name from being associated with more than one Universal Template:

*The template 'template-name1' specifies an extension name 'extension-name' that is already associated with template 'template-name2'.*

## Log Level

Each registered Agent within the Universal Controller displays its configured **Log Level**, with one of the following options.

- Trace
- Debug
- Info
- Warn
- Error
- Severe

For Extension-based Universal Tasks/Templates, a **Log Level** option will be introduced at both the **Universal Task**-level, the **Universal Task Instance**-level, and the **Universal Template**-level to specify the Log Level for Universal Extension logging.

The option will have a default value of **Inherited**, meaning the following.

<b>Universal Task</b>	<ul style="list-style-type: none"> <li>Specify Inherited to inherit the template Log Level setting.</li> </ul>
<b>Universal Task Instance</b>	<ul style="list-style-type: none"> <li>Specify Inherited to inherit the template Log Level setting.</li> <li>The initial Log Level is derived from the Universal Task definition at Universal Task Instance creation time.</li> </ul>
<b>Universal Template</b>	<ul style="list-style-type: none"> <li>Specify Inherited to inherit the agent Log Level setting.</li> </ul>

At Universal Task Instance runtime, if both the instance and the template are specified as Inherited, then the instance will inherit the agent Log Level.

# Creating a Universal Extension

To create a Universal Extension:

- Define a new Template/Task in the user interface for a Universal Extension using the Universal Template/Task framework, including:
  - Task fields required for defining and launching the process through the Universal Extension.
  - Task Instance fields required for capturing data coming back throughout the life-cycle of the Universal Extension process.
  - Task Instance commands, and their supported status(es), that can be executed through the Universal Extension.
  - Universal Events, if any, that the Universal Extension may generate throughout the life-cycle of the Universal Extension process.
- Upload/attach an Extension Archive zip file to the Universal Template, recognized by the Universal Agent, which can either be deployed on-demand to registered agents, or automatically following a successful agent registration.
- Once the Universal Template is created, and the Extension Archive zip file is attached, generate a zip package recognized by Universal Controller that contains both the Universal Template definition and the Extension Archive zip file recognized by the Universal Agent.

For specific details on creating a Universal Extension, see [Creating a Universal Template](#).

# Universal Extension for Universal Agent

- [Overview](#)
- [Universal Extension Task Overview](#)
- [Implementation](#)
  - [Agent Start-up](#)
  - [Agent Registration](#)
- [Extension Manager](#)
  - [Extension Worker Process](#)
  - [Starting an Extension instance](#)
  - [Worker Process Management](#)
  - [Warm Start Processing](#)
- [UniversalExtension Base Class Package](#)
  - [UniversalExtension Interface](#)
  - [Communication Methods](#)
  - [Universal Extension Definition \(module\)](#)
- [Extension Repository](#)
  - [Extension Repository Security](#)

## Overview

A new dynamic “extension” subsystem in the Universal Automation Center (UAC) platform allows developers to implement custom solutions that can be integrated into UAC. These “solutions” could be, for example, a task, trigger, or monitor.

The initial phase of this subsystem focuses on tasks.


This feature provides an easy way for Stonebranch, customers, and/or third party developers with special domain knowledge, to develop new solutions that can be seamlessly integrated into the UAC software stack. This is especially beneficial when the Domain knowledge required for a solution does not exist within the Stonebranch development teams and/or justification does not exist to devote the necessary resources.

While the Universal Task addresses this issue to some extent, it does not provide the level of integration and customer experience that can be achieved with the Universal Extension concept.

The main differentiators between Universal Extensions and Universal Tasks are:

- **Bidirectional communication** between the Extension instance and the Universal Controller. This allows an Extension instance running on an Agent system to dynamically update the associated instance running in the Controller as the state changes. For example, a “Task” type extension could update output fields on a task instance in the Controller (for example, Job ID, Run Status, Distribution Status, and Percentage Complete).
- **Dynamic command definitions** to support a complete solution (not just a single script execution). This allows an Extension solution to do things such as:
  - Provide commands that can help populate task definition form fields (such as is currently done with the PeopleSoft task type),
  - Provide a relative set of commands to be used on a task instance form that can be used to perform actions related to the unit of work that is specific to that task instance (such as is currently done with the SAP task type: Abort SAP job, Interrupt SAP Process Chain, etc.).
- **Dynamic event definition/generation** (Universal Events) allow Extension instances to trigger actions in the Controller. Universal Events defined and monitored in the Controller could be raised by an Extension instance on an Agent.

### Note

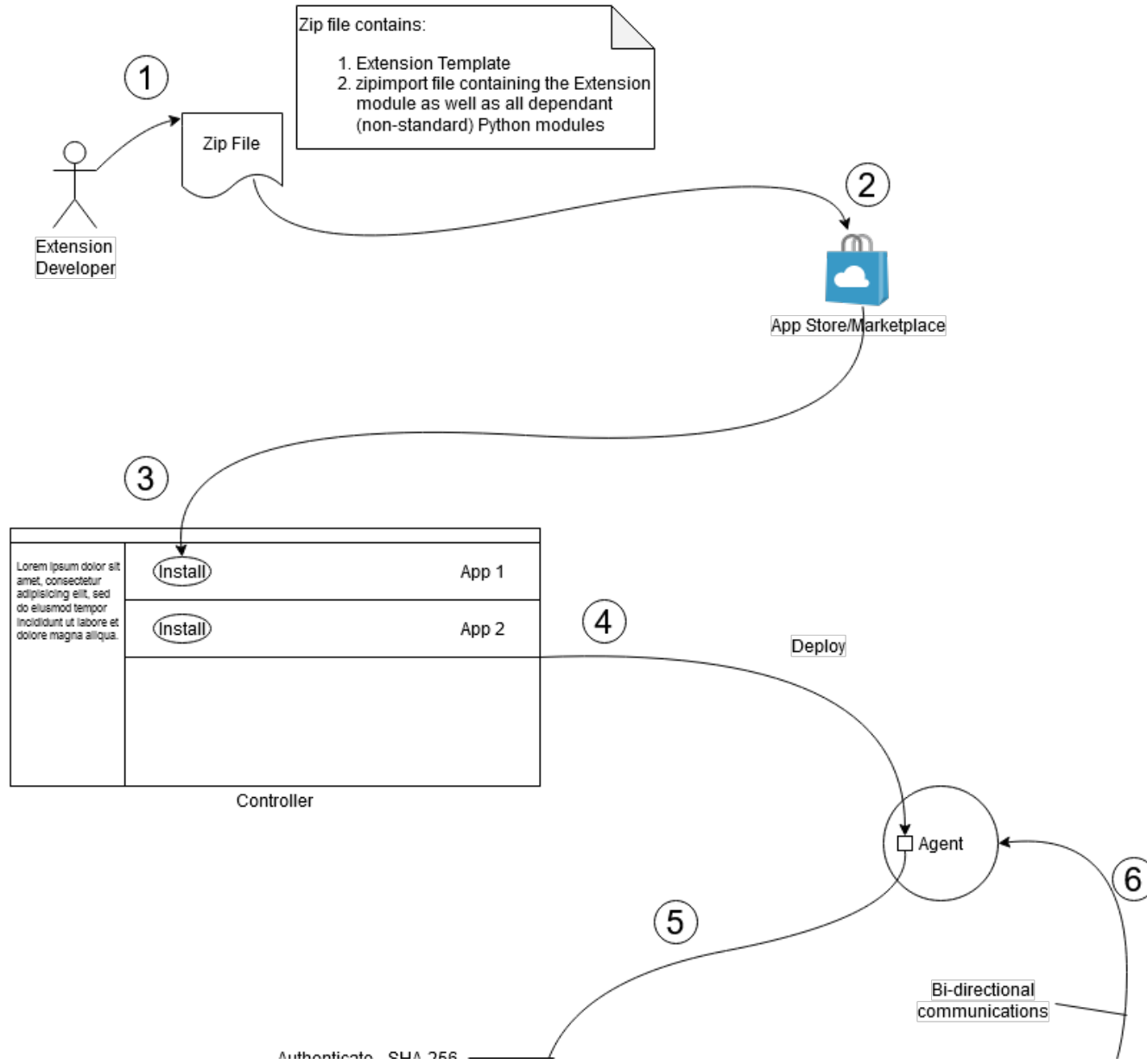
 Universal Extension is not intended to replace Universal Task. Universal Task will remain a flexible, and perhaps more accessible, solution for many automation needs.

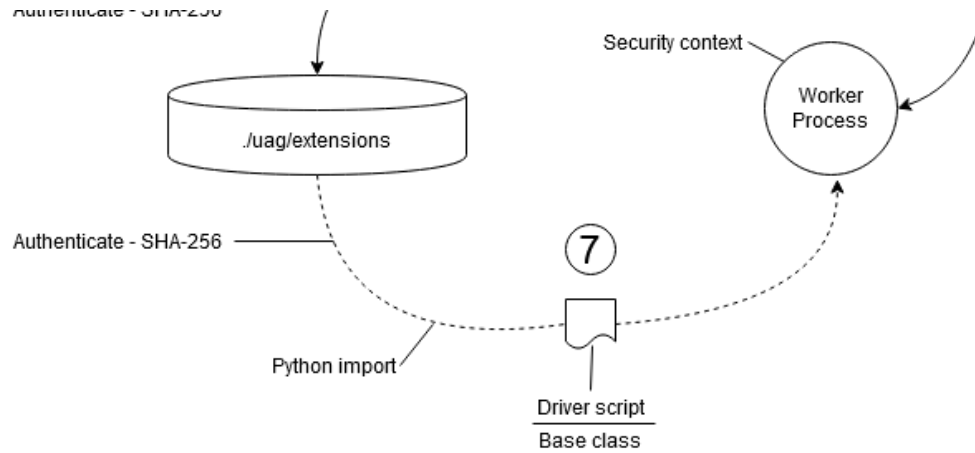
Universal Agent has been enhanced to support this new Universal Extension subsystem. The Extension subsystem allows various types of functionality to be developed and integrated into the UAC software stack (for example: tasks, triggers, and monitors).

The generic Extension concept supports the development of some primary operation (for example, task, along with an optional set of related secondary/supporting operations (Choice Commands, Dynamic Commands).

The initial phase (and this document) focuses on a "Task" type Extension. However, the architecture is designed with other extension types in mind.

## Development to Execution





As illustrated in the diagram above:

1. Developer creates Extension implementation and zips up the Extension Template, the Python Extension module and any dependent non-standard Python import modules.
2. The zipped package is uploaded to the marketplace.
3. A user downloads the extension to the Universal Controller.
4. The Universal Controller deploys the Extension to an Agent.
5. The Agent persists the Extension to the file system in the Extension Repository. (after authenticating the deployment with the Controller supplied SHA-256 checksum).
6. The Agent starts a Worker process to execute an Extension instance (supporting bi-directional communication).
7. The Agent authenticates the Extension that is to be run (comparing its SHA-256 checksum with the Controllers reported checksum) and inserts a driver script into the Worker process starts the Extension instance.

## Universal Extension Task Overview

A Universal Extension Task is comprised of the following:

- An Extension Task Template definition (defined in the Controller)
  - Input field definitions
  - Output-only field definitions
  - Custom command definitions
- An Extension module developed in Python (delivered/stored to the Agent system for execution)
  - Contains the code that provides the functionality for the Extension's primary operation (e.g. task)
  - Contains code that provides the functionality for the Extension's "Dynamic Commands"

Extension modules will be developed in an unspecified development environment.

Fully developed Universal Extensions are distributable (via the Stonebranch Marketplace, in-house Controller-to-Controller bundle promotion, etc.) to target Controller systems.

In a UAC deployment, the Controller serves as the primary repository and distributor of Universal Extensions. The Controller is responsible for pushing extension modules down to Agents as needed.

The deployment of Extension modules from Controller to Agent takes place via OMS messaging.

Note



For phase one of the Universal Extension platform, the deployment of Extension modules to Agent systems will be a manual process.

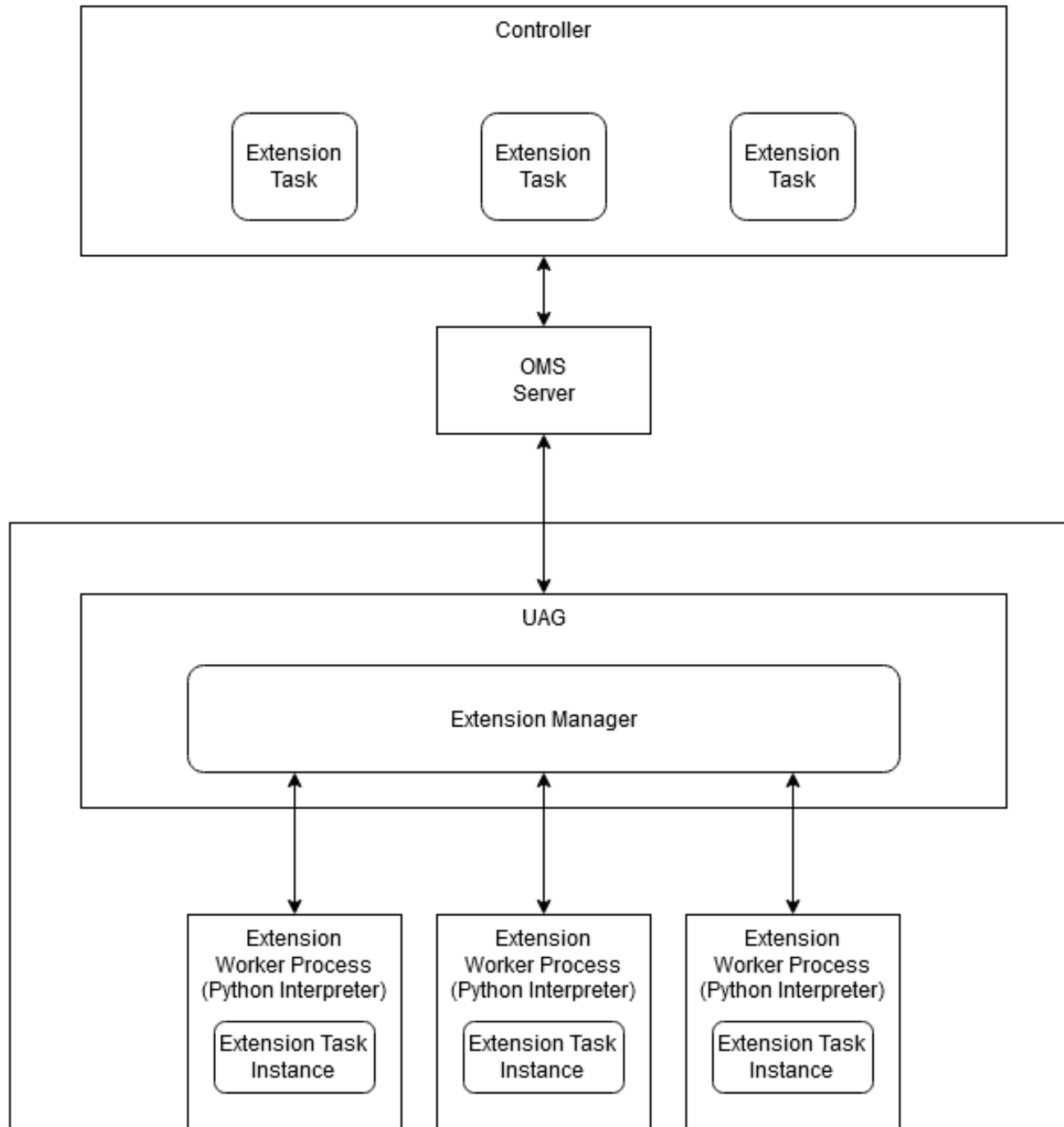
On the Agent side, Extension modules are stored in a cache repository (referred to as the **Extension Repository** throughout this document). This cache repository is simply be a directory on the Agent file system where Extensions modules passed from the Controller are stored. Extension modules are stored on the Agent file system as zipimport files (Python import modules stored as Zip archives).

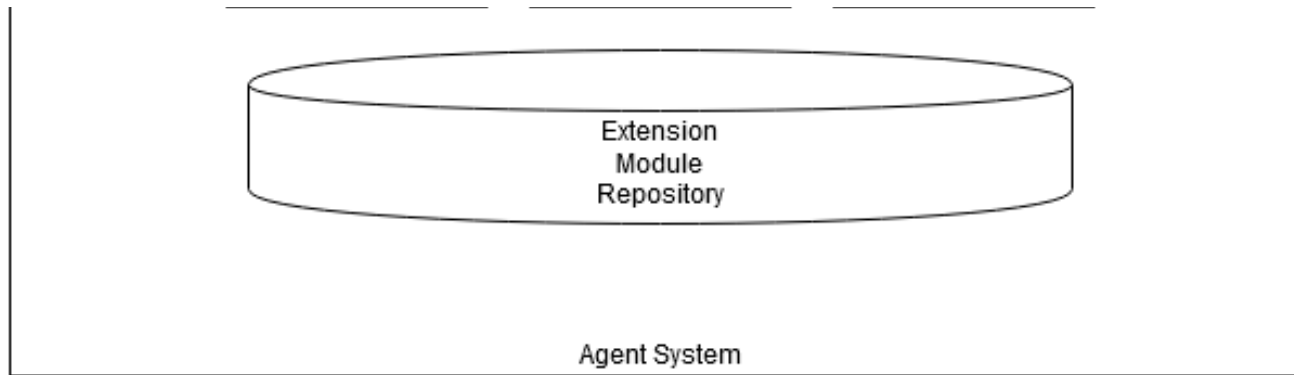
- For Windows systems (System Mode install), the Extension Repository is under: **!Program Files!Universal!universal!UAGSrv!extensions**
- For \*nix systems (System Mode install), the Extension Repository is under: **./var/opt/universal/uag/extensions**
- User mode installs use an equivalent path, for the "extension" directories above (for example, under uag in the "unvdata" install path).

The Universal Agent processes Extension tasks in much the same way as any other task type. A unit of work is executed on behalf of a Controller request. Artifacts of the unit of work are returned to the Controller (for example: standard output, standard error, return codes).

- Standard output generated from the Extension task instances is directed to \*\_stdout files in the UAG cache directory.
- Standard error generated from the Extension task instances is directed to \*\_stderr files in the UAG cache directory.

At a high level, the main difference between an Extension task and a Windows or Unix task is that the Agent is managing (or caching) a repository of modules that contain the functionality for the various "Extension" implementations.





## Implementation

UAG has been enhanced in the following ways to support the new Extension subsystem:

- Agent start-up
- Agent registration
- Extension Task-related messages
- Extension Manager component
- UniversalExtension Python Base Class
- Extension Repository

This initial implementation of Universal Extensions uses Python for the language in which Extension modules are developed.

Extension modules were developed by extending a Stonebranch provided base class (**UniversalExtension**). This base class provides the functionality required to integrate, manage, and orchestrate custom functionality running on the Agent system with resources (for example, task instances) running in the Controller.

## Agent Start-up

Upon start-up, UAG performs the following Extension specific operations:

- Python Discovery
- Extension Repository Initialization

## Python Discovery

Python discovery is the process of locating available Python executables on the system and categorizing them by version. This discovery process is performed once at start-up to prepare for efficient Python Resolution later during the execution of Extension instances.

Universal Extensions may have specific Python version requirements that must be met in order for the Extension to run correctly. These requirements are specified in the "**requires\_python**" metadata for the Extension (specified in **extension.yml** in the Extension zip module).

To satisfy the Python requirements of an Extension, UAG Extension Manager must know the available Python interpreters installed on the target system and choose one (if any) that meets all specified requirements.

Python discovery is performed by:

1. Checking for a Python distribution installed with the Universal Agent.

2. **[Windows specific]** Checking the “HKEY\_LOCAL\_MACHINE\SOFTWARE\Python\PythonCore” registry key for a Python installation.
3. Checking the value specified for the UAG Server EXTENSION\_PYTHON\_LIST configuration option, which can specify a comma-separated list of locations in which Python is installed.
  - a. This option has a default value of /usr/bin/python3, /usr/bin/python, and /usr/libexec/platform-python directories/symlinks for Unix agents.
  - b. For Windows, there is no default value. If this option is left empty, Python discovery stops at step 2.

For each Python executable found, UAG Extension Manager determines the version and record an entry (version and path) in an internal list for later lookup.

## Extension Repository Initialization

Upon Agent start-up, UAG scans its Extension Repository for installed extension modules. Extension modules are stored in the repository as zipimport files (Python import modules stored as Zip archives).

For each extension module found in the Extension Repository, UAG collects the following information:

- **Extension Name**  
The Extension module file name (not including the file extension) is used as the Extension name. This should equate to the Universal Extension Template Name of the related Universal Extension defined in the Controller system to which the Agent connects.
- **Extension Checksum**  
A SHA-256 checksum of Extension file

The collected Extension information is stored in an internal table for later use with the JSS-HELLO message during Agent registration.

## Agent Registration

### Universal Extension Deployment

An Agent registering with the Universal Controller has the ability to automatically accept deployment of available Universal Extensions.

However, the owner of an Agent deployment may wish to control which (if any) Universal Extensions are allowed to be installed by the Controller. Therefore, the following configuration options have been added to UAG to control Extension deployment:

- [EXTENSION\\_ACCEPT\\_LIST](#)
- [EXTENSION\\_DEPLOY\\_ON\\_REGISTRATION](#)

An Agent must inform the Controller of which Extensions are currently installed on its system (collected at Agent start-up). Additionally, the Agent must supply the checksum associated with each installed Extension. This is required for the Controller to know which (if any) Extensions need to be installed or replaced on the Agent system. Any Extension on the Agent system with a checksum that does not match the checksum of the associated Extension in the Controller's repository must be replaced.

## Extension Manager

UAG is enhanced with a new internal component (**Extension Manager**) that manages the execution of Extensions and facilitates the flow of messages between the Controller and the Extension instances. Extension instances run in a Python interpreter process (the **Extension Worker Process**) started in the security context of the user specified on the task definition (as do Windows and Unix/Linux tasks).

In general, the Extension Manager is responsible for handling work requests related to Extensions and managing the work resources and resulting message exchanges required to process the work.

Specifically, this involves:

- Starting and stopping Extension Worker processes.
- Authenticating Extension module with Controller provided checksum prior to import.
- Initiating Extension instances within a Worker process.
- Relaying information from Extension instances running in the Worker process to the Controller.
  - Status updates for associated output fields

- Universal Events
- Executing Extension Commands (initiated from the Controller) to Worker Process.
- Canceling Extension instance (along with the Worker process) if/when requested by the Controller.

## Extension Worker Process

Extension Worker processes are Python interpreter instances that are started by the Extension Manager. Worker processes exist to run Extension instances.

The Extension Manager controls the Worker process by formulating a small Python script designed to run a target Extension module and injecting that script into the Worker process over stdin.

The Extension instance running within the Worker process sends messages to the Extension Manager over a control channel (pipe).

Output generated by the Extension instance that is written to stdout and stderr are redirected to cache spool files that are created in the UAG cache directory.

## Security Context

Each Worker process is created on demand to process a specific Extension operation. Each Universal Extension task definition specifies the user credentials that the task should run under. If no credentials are specified, the task should run under the security context of UAG. Therefore, when a worker process is created, it is created with the security context specified by the Extension task it is intended to process.

## Starting an Extension instance

Extension instances are started when UAG receives a **JSS-LAUNCH**, **JSS-UNVCHOICEREQ**, or **JSS-UNVCMDREQ** message from the Controller.

Upon receiving the **JSS-LAUNCH**, **JSS-UNVCHOICEREQ**, or **JSS-UNVCMDREQ** message, the Extension Manager performs the following operations to start the new Extension instance:

- Authenticate the target Extension module currently residing in the Agents Extension Repository.
  - The checksum of the target extension module must match the checksum provided by the Controller in the initiating **JSS-LAUNCH**, **JSS-UNVCHOICEREQ**, or **JSS-UNVCMDREQ** message.
  - If the checksums do not match, the Extension is not started and a **JSS-STATUS(ERROR)** message will be returned to the Controller with an **ERRDESC** value of "**Extension checksum mismatch**".
  - This is considered a start failure and no further actions will be taken towards starting the extension.
- Open stdout/stderr spool files in the UAG cache directory - into which stdout and stderr of the Worker process will be redirected (<execid>\_stdout and <execid>\_stderr).
- Open a pipe to be used as a "message output channel" for the Worker process.
- Start a Worker process using CSK where:
  - The process runs under the security context of the runtime credentials specified in the **JSS** message.
    - If no credentials are specified in the JSS message, the process runs under the security context of the account the UAG runs under.
    - The stdout and stderr are redirected to the cache files mentioned above (<execid>\_stdout and <execid>\_stderr).
- Pass a handle to the write end of the "message output channel" into the Worker process.
- Generate a small Python script that will run the target Extension module with the parameters specified in the initiating **JSS** message
- Close stdin for the Worker process cause the Python interpreter to begin processing the script (and thus the Extension instance).

Generating a SHA-256 checksum for each attempted invocation of an extension instance is obviously ideal in terms of security. However, it could potentially add a significant performance hit to a system that makes heavy use of Extension tasks.

It may be sufficient to only generate a checksum (Agent side) if the cached checksum is older than some threshold. A threshold as small as 1 second could conceivably have significant impact on a burst of short lived Extension executions targeting the same Extension module.

Auto-deployment for "**Extension checksum mismatch**" start failures

The Controller could silently handle "**Extension checksum mismatch**" start failures and auto-deploy the target Extension module to the target Agent.

Following a successful deployment, the Controller could re-launch the Extension instance.

After starting the Worker process, the process resources (CSPProcess structure, “message output channel” handle, etc.) are stored in an appropriate data structure to be used for monitoring the Worker process over the lifetime of the Extension instance.

## Worker Process Management

Once the stdin of the Worker process is closed and the Extension Worker begins executing the Extension instance, no further input is required from the Extension Manager. At that point the management of the Worker process becomes a monitoring operation.

A dedicated monitoring thread will continuously monitor Running Worker processes for messages and completion. Potential messages sent from the Worker process are:

- ESS-STATUS-UPDATE
- ESS-UNVEVENT
- ESS-EXT-COMPLETE
- ESS-CMD-COMPLETE
- ESS-CHOICE-CMD-COMPLETE

## Warm Start Processing

**Warm Start Processing** is a term used to refer to a process UAG goes through upon startup by which all task instances that were active at the time of the last shutdown (intentional or otherwise) are reviewed and proper action is taken based on state and platform.

Two general scenarios exist for tasks that were active at the time of the last UAG shutdown:

1. The process associated with the task instance has completed between the time of UAG shutdown and UAG restart.
2. The process associated with the task instance is still running at the time UAG restarts.

### Scenario 1: Process has completed

For scenario 1, where the process has completed, UAG will send a **JSS-STATUS(JOB INDOUBT)** message back to the Controller to indicate that UAG is unable to determine how things turned out for the process in question. This behavior is consistent between Unix and Windows and no differences exist between Universal Extension tasks and other task types.

### Scenario 2: Process is still active

For scenario 2, where the process is still active, there is a difference in behavior between platforms. On **Unix platforms**, UAG will send a **JSS-STATUS(JOB INDOUBT)** message back to the Controller just like with scenario 1. No attempt is made to resume monitoring the process. On **Windows platforms**, UAG will resume monitoring the process. Once monitoring has resumed, the task processing continues as though there was never a termination of UAG.

## Warm Start Processing Enhancements for Universal Extension Tasks

Unlike other task types, Universal Extension Worker processes include a message channel (pipe) that allows the Extension instance to send messages back to UAG. If a UAG shutdown occurs while an Extension instance is still active, the message channel between the worker process and UAG is broken. However, all messages sent from Universal Extension instance must be processed in order to consider the execution a success.

Therefore, in order to “reconnect” to a Universal Extension instance during Warm Start Processing, UAG must regain access to the messages in addition to just monitoring the status of the process. To prevent message loss in such a case, Universal Extension instances are provided a **message cache** (in addition to the message pipe) during the initial instantiation.

The message cache is implemented as a dedicated flat file in the UAG cache directory. If the Extension Instance is unable to send messages over the pipe (due to a UAG shutdown), the Extension will revert to the message cache. This allows UAG to “reconnect” to the message stream emitted by an Extension instance that is monitored after a Warm start. The process is transparent to the Controller. This brings the Warm Start Processing behavior of Extension tasks in line with other task types (from a user/Controller perspective).

## UniversalExtension Base Class Package

The UniversalExtension base class package is a new Agent “component” that is part of the core Agent installation. It is a Python package that provides the functionality required to support the execution of custom Extensions within a Worker process.

Custom Extension implementations must provide a class that derives from the UniversalExtension base class. The UniversalExtension base class provides an interface that the Extension developer must implement. Additionally, it provides methods that allow the Extension instance to propagate state information back to the associated Extension instance running in the Controller.

The UniversalExtension base class package contains the following modules:

Module	Description
./extension_choice_result.py	Internal use
./extension_command_result.py	Internal use
./extension_result.py	Result handling class for use in Universal Extension implementation
./extension_start_result.py	Internal use
./universal_extension.py	Universal Extension base class
./deco/choice.py	Function decorator for Dynamic Choice Command in Universal Extension implementation
./deco/command.py	Function decorator for Dynamic Command in Universal Extension implementation

For detailed information on this package, see [Universal Extension 1.0.0 API](#).

## UniversalExtension Interface

### def extension\_start(self, state)

This method must be overridden by the custom Extension class that derives from the UniversalExtension class. It is called by UniversalExtension base class in response to a **JSS-LAUNCH** message sent from the Controller. This is essentially the main() function for an Extension implementation and is the starting point for work that will be performed. The **state** parameter passes in a dictionary of Extension instance fields that were defined in the Extension template.

## Communication Methods

### def update\_extension\_status(fields):

This method can be called at any time by the Extension instance. It is used to propagate state changes back to the associated extension instance in the Controller. Essentially, any/all output fields defined in the associated Extension Template can be updated using this method. This method results in an ESS-STATUS-UPDATE message being sent from the Worker process to the UAG Extension Manager, followed by a JSS-STATUS(JOB UPDATE) message being sent from UAG Extension Manager to the Controller (via OMS server).

The **fields** parameter expects a dictionary of output fields to be sent back to the controller for the associated Extension instance. Field names are implementation dependent and correlate with the Universal Template field names in the Controller's Template definition. For example:

```
{
  # Output fields to be updated
  "sysid": "BW7",
  "jobid": 81762549
}
```

## def publish\_extension\_event(event\_state):

This method can be called at any time by the Extension instance. It is used to publish a Universal Event to the associated Universal Controller.

Universal Events are optionally defined in the Universal Extension Template as part of the Universal Extension definition.

This method results in an ESS-UNVEVENT message being sent from the Worker process to the UAG Extension Manager, followed by a JSS-UNVEVENT message being sent from UAG Extension Manager to the Universal Controller (via OMS server).

The event\_state parameter expects a dictionary of event related fields - including a list of implementation defined "event attributes". For example:

```
{
# The unique Universal Event name defined within the Universal Template.
"event_name": "job_complete",

# Time to live; how long, in minutes, the Universal Event data is kept (in the Controller).
"ttl": 30,

# Dictionary of attributes associated with the Universal Event definition.
# An attribute should be added for each attribute defined in Universal Event Template
# definition associated with the specified "event_name".
# Keys in the "event_attributes" dictionary correlate with an associated
# Universal Event attribute defined in the associated template.
"event_attributes":
{
"sysid": "BW7",
"jobname": "JOB-1",
"jobid": 81762549
}
}
```

## Universal Extension Definition (module)

Universal Extension modules allow for the development of simple or complex solutions that tightly integrate with the Universal Controller.

The UniversalExtension base class supports two types of custom functionality that can be invoked by the Controller:

- A mandatory singular primary operation (e.g. task implementation)
- An optional collection of Dynamic Commands

### Primary Operation

The primary operation is the focus of the Extension and must be implemented. This is done by overriding the **extension\_start** method of the UniversalExtension base class.

Of course, **extension\_start** is just an entry point. A single primary operation could perform any number of variants or sub operations based on values passed in from the Controller.

### Secondary Operation - Dynamic Commands

Dynamic commands are intended to support the functionality of the Primary Operation of the Universal Extension, They can be used to return additional information for a task instance or to carry out some related (or unrelated) action on behalf of the task instance. Dynamic commands can pass instance specific field data values from the task instance form to the extension process that executes the command on the Agent.

To implement a Dynamic Command in an Extension module, define a function or method with the following signature:

- **def my\_command\_function\_name(self, state):**

Decorate the function with a `@dynamic_command` decorator:

- `@dynamic_command("dynamic_command_name")`

For example:

```
@dynamic_command("my_dynamic_command")
def my_command_function(self, state):
    # Do something
```


In the code snippet above, `"my_dynamic_command"` is the command name given to the Dynamic Command in the Controller's Extension definition template and `my_command_function` is the actual python function name from the Extension module that processes the command.

The `@dynamic_command` decorator provides a means for custom Extension implementations to associate a function defined in the Extension module with a Dynamic Command in the Controller's related Extension template definition. The `@dynamic_command` decorator is defined in the `./deco/command.py` module in the `universal_extension.zip` package.

When a dynamic command function is called, the `state` function variable will contain a dictionary populated with dependent fields (as defined in the Dynamic Command template definition) from the command invocation source. The keys of the dictionary will equate to the corresponding template field names and the values will be the values of the fields in the command invocation source (task instance form) at the time of invocation.

Dynamic command functions must return an object of type `ExtensionResult`. `ExtensionResult` is a class defined in module `./extension_result.py` in the `universal_extension.zip` base class module.

The `ExtensionResult` class is used to return the following attributes to the `UniversalExtension` base class:

Attribute	Description
<code>rc</code>	<p><b>Optional</b></p> <p>This attribute represents the return code of the <code>extension_start</code> operation and determines whether the Extension task instance is perceived as completing with <b>Success</b> or <b>Failed</b> by the Controller. A value of 0 indicates <b>success</b>. All other values indicate an error condition and result in a status of <b>Failed</b> in the Controller. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.</p> <p>The <code>ExtensionResult</code> class sets a <b>default value of 0</b> to the <code>rc</code> attribute.</p>
<code>message</code>	<p><b>Optional</b></p> <p>This attribute allows the extension to pass a completion message back to the Controller. If <code>rc</code> is set to 0, the message will be considered informational and will be logged by the Controller. If <code>rc</code> is non-zero, the message will be considered an error message and will be displayed to the user on the task instance form.</p> <p><b>The default value is an empty string.</b></p>
<code>output</code>	<p><b>Optional</b></p> <p>This attribute is a Boolean value that specifies if the command produced output that should be persisted and displayed under the task instance <b>Output</b> tab in the Controller task instance associated with the dynamic command invocation.</p> <p><b>The default value is False.</b></p> <p>Note   This flag allows distinguishing between a command that does not produce output and a command that produces output but the output returned was empty.</p>

<b>output_data</b>	<p><b>Optional</b></p> <p>This attribute specifies the data that will be returned to the Controller for the command execution. It is interpreted as a UTF-8 encoded text object.</p> <p>If the <b>output</b> attribute is <b>True</b>, the <b>output_data</b> will be persisted in the Controller as a record under table ops_exec_output and appearing as <b>Universal Command</b> output type from the task instance <b>Output</b> tab.</p> <p><b>The default value is None.</b></p>
<b>output_name</b>	<p>This attribute is used to provided a custom name for the associated output data. The output_name (if provided) will be used in the presentation of the output_data by the Controller.</p>

## Sample Dynamic Command Return Value

### Sample 1

```
@dynamic_command("sample_1")
def command_sample_1(self, state):
    """Dynamic command."""
    return ExtensionResult(
        message: 'sample_1 completed successfully',
        output = True,
        output_data = 'Hello from dynamic command sample_1!',
        output_name = 'DYNAMIC_OUTPUT')
```

### Sample 2

```
@dynamic_command("sample_2")
def command_sample_2(self, state):
    """Dynamic command."""
    ...
    if (something_went_wrong):
        return ExtensionResult(rc = 1, message = "Something went wrong")
    return ExtensionResult(
        message: 'sample_2 completed successfully',
        output = True,
        output_data = str(state),
        output_name = 'DYNAMIC_OUTPUT')
```

## Dynamic Command Use Cases

Extension developers can use dynamic commands for many different purposes.

Use Case	Description
Helper commands on task definition form	<p>In this scenario, commands are used to retrieve data that will be used to populate form fields (like drop-downs). The commands may use input values from dependent fields on the form to pull data that is highly relevant to the task being defined.</p> <p>This scenario matches the behavior that is seen in the existing PeopleSoft task type. In this case, the command functionality is related to the Extension type but, is unrelated to any specific work that has been executed by a task instance.</p>

<p>Action commands on a task instance form</p>	<p>In this scenario, commands are used to perform some action related to the specific unit of work associated with the task instance. An example would be a task instance that is running an SAP process chain. For this task, an action command might be to "Interrupt Process Chain".</p> <p>In order for the command to perform the action, it requires values from the task instance on which it is called. However, the required instance values are acquired from the task instance on the Controller side; no interaction with the associated running task instance on the Agent system is required. The command is run in an independent Worker process and, from there, reaches out to the SAP system to perform the requested action.</p>
<p>Action command on a task instance that interacts with the Worker process associated with the task instance</p>	<p>In this scenario, the command must run in the active Worker process that is processing the task instance. An example of this would be some type of "refresh" command that would instruct the running task instance (on the Agent side) to immediately send some state update back to the task instance in the Controller. The "refresh command" scenario would be more applicable to a "Monitor" type Extension but, could be envisioned for a "Task" type as well.</p> <p>This type of Dynamic Command that must be executed within the running Worker process that is executing the task instance adds many complications over the "out of process" scenarios described in Case 1 and Case 2. It involves synchronizing resources in a multi-threaded environment. If this "in process" command scenario is required, the UniversalExtension class would have to provide a framework that makes it easy for the Extension developer to implement.</p> <p>This use case is not supported in Phase 1.</p>

## Dynamic Choice Field Commands

Dynamic Choice commands are helper functions for dynamically populating choice fields on the Universal Extension task definition form in the Controller (like with the PeopleSoft task type).

To implement a Dynamic Choice Command in an Extension module, define a function or method with the following signature:

- `def my_choice_command_function_name(self, state):`

Decorate the function with a `@dynamic_choice_command` decorator:

- `@dynamic_choice_command("dynamic_choice_command_name")`

For example:

```
@dynamic_choice_command("choice_field")
def my_choice_command_function(self, state):
    # Do something
```

In the code snippet above, "`my_dynamic_choice_command`" is the command name given to the `Dynamic Choice_Command` in the Controller's Extension definition template and `my_command_function` is the actual python function name from the Extension module that processes the command.

The `@dynamic_choice_command` decorator provide a means for custom Extension implementations to associate a function defined in the Extension module with a Dynamic Choice Field in the Controller's related Extension template definition. The `@dynamic_choice_command` decorator is defined in the `universal_extension` module.

When a dynamic choice command is called, the `state` function variable will contain a dictionary populated with dependent fields (as defined in the Dynamic Choice Field template definition) from the command invocation source (i.e. the task definition form in the Controller). The keys of the dictionary will equate to the corresponding template field names and the values will be the values of the fields in the command invocation source (task definition form, task instance form, etc.) at the time of invocation.

Dynamic Choice command functions must return an object of type `ExtensionResult`. `ExtensionResult` is a class defined in the `extension_result` module in the `universal_extension` package. The `ExtensionResult` class is used to return the following attributes to the `UniversalExtension` base class:

Attribute	Description
-----------	-------------

<b>rc</b>	<p><b>Optional</b></p> <p>This attribute represents the return code of the <b>extension_start</b> operation and determines whether the Extension task instance is perceived as completing with <b>Success</b> or <b>Failed</b> by the Controller. A value of 0 indicates <b>success</b>. All other values indicate an error condition and result in a status of <b>Failed</b> in the Controller. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.</p> <p>The ExtensionResult class sets a <b>default value of 0</b> to the rc attribute.</p>
<b>message</b>	<p><b>Optional</b></p> <p>This attribute allows the extension to pass a completion message back to the Controller. If <b>rc</b> is set to 0, the message will be considered informational and will be logged by the Controller. If <b>rc</b> is non-zero, the message will be considered an error message and will be displayed to the user on the task instance form.</p> <p>The default value is an empty string.</p>
<b>values</b>	<p><b>Optional</b></p> <p>This attribute provides a list which can be populated with a set of string values to be returned to the Controller and used to populate the associated dynamic choice field on the Extension task form.</p> <p>The default value is an empty list.</p>

## Sample Dynamic Choice Commands

### Sample 1

```
@dynamic_choice_command("choice_1")
def choice_2(self, state):
    """Dynamic choice command."""
    self.log.info("Entering choice_2")
    message = 'Message: Hello from dynamic choice command choice_2!'
    values = ['value 1', 'value 2', 'value 3', 'value 4']
    self.log.info("Exiting choice_2")
    return ExtensionResult(message = message, values = values)
```

### Sample 2

```
@dynamic_command("sample_2")
def command_sample_2(self, state):
    """Dynamic command."""
    ...
    if (something_went_wrong):
        return ExtensionResult(rc = 1, message = "Something went wrong.")
    return ExtensionResult(
        message = 'Message: Hello from dynamic command sample_2!',
        values = ['value 1', 'value 2', 'value 3', 'value 4'])
```

## Dynamic Choice Field Use cases

### Case 1: Helper commands on task definition form

In this scenario, commands are used to retrieve data that will be used to populate form fields (like drop-downs). The commands may use input values from dependent fields on the form to pull data that is highly relevant to the task being defined.

This scenario matches the behavior that is seen in the existing PeopleSoft task type. In this case, the command functionality is related to the Extension type but, is unrelated to any specific work that has been executed by a task instance.

## Extension Repository

UAG will manage (in cooperation with the Controller) a repository of **Extension modules**. This management will include synchronization with the Controller in terms of which extension modules are available or allowed. The end goal is for the Controller to push Extension solutions down to the Agents as needed.

The extension repository is a path on the Agent file system where "Extension" modules are stored. The modules themselves are stored as zipimport files (Python import modules stored as Zip archives).

### Windows

\\Program Files\\Universal\\universal\\UAGSrv\\extensions

### \*nix

./var/opt/universal/uag/extensions

It may be desirable to have this be a configurable value. Therefore, the following configuration value is proposed for the UAG configuration file:

**extension\_repository**

## Extension Repository Security

Extension modules are written to the file system by UAG (after being pushed down from the Controller via OMS messages). Therefore, they will be owned by the account used to execute the Broker.

Extension modules must be world readable to allow Worker processes running under the security context of an unprivileged user to load the modules into the Python interpreter.

Write permission should be limited to the owner (Broker account) to prevent tampering. However, that is not enough. A malicious user with sufficient authority could potentially manipulate extension module code to perform undesirable actions. Therefore, another means is required to guaranty the integrity of the extension modules.

### Checksums

The Controller is the central repository/authority with regards to Extension modules. Therefore, the Controller can pass down a known checksum for the Extension module associated with an Extension work request. The Extension manager can then use the provided checksum to verify the integrity of the Extension module residing on the Agent's file system. If the checksums do not match, the Extension Manager will consider it a pre-start failure and return a **JSS-STATUS(ERROR)** message to the Controller with an **ERRDESC** value of "**Extension checksum mismatch**". and a **EXT\_CHECKSUM\_MISMATCH** value of **true**.

# How-To/Getting Started for Universal Extension Development

This document tells you how to develop a Universal Extension.

It contains the following pages:

Title	Description
<a href="#">Introduction</a>	Introduction to Universal Extension and its capabilities.
<a href="#">Development Environment Set-Up</a>	Requirements for setting up a Universal Extension development environment.
<a href="#">Universal Extension API</a>	Detailed Information on the Universal Extension API.
<a href="#">Task Entry Point</a>	Creating a Sample Universal Extension.
<a href="#">Dynamic Choice Field</a>	Adding Dynamic Choice fields to the Sample Universal Extension.
<a href="#">Dynamic Update and Output Only Fields</a>	Adding Output Only fields to the Sample Universal Extension.
<a href="#">Dynamic Command</a>	Adding a Dynamic Command to the Sample Universal Extension.
<a href="#">In-Process Dynamic Commands</a>	Adding two In-Process Dynamic Commands to the Sample Universal Extension.
<a href="#">Cancel Command</a>	Adding Cancel Command to the Sample Universal Extension.
<a href="#">Publishing Events</a>	Demonstration of the new event publishing API
<a href="#">Troubleshooting and Debugging</a>	Checking log levels, retrieving output, and debugging commands.

# Introduction

- [Universal Extension](#)
- [Unique Capabilities](#)
- [UIP-CLI](#)
- [UIP VS Code Extension](#)

## Universal Extension

A Universal Extension is a task type provided by Universal Controller. It allows third-party developers (or anyone with access) to implement custom solutions (Extensions) that can be tightly and seamlessly integrated into the Controller's native functionality.

Universal Extensions were designed to facilitate integrations between UAC and external sources of information or functionality. The extension implementation is stored and managed in the Controller and executed on agent systems.

Universal Extensions consist of two primary parts:

1. A Universal Template definition that is created and stored in the Controller.
2. A Python zip archive that is developed outside of the Controller and contains Python source code that implements the Extension functionality.

Although the Python zip archives are developed outside of the Controller, they are eventually uploaded to an associated Universal Template in the Controller. From there, the Controller will automatically manage the deployment of the Extension zip module down to agent systems as needed.

## Unique Capabilities

The Universal Extensions provides unique capabilities that are not available with other general purpose task types.

<b>Custom task form definition</b>	As with Universal Tasks, Universal Extensions use the Controller's Universal Template form building system. This allows a new task type to be built up field by field using types and verbiage that is natural and specific to functionality requirements of the task type being created.
<b>Dynamic Choice Fields</b>	Dynamic Choice Fields are Universal Template Choice fields that can be added to a Universal Extension Task form.  Dynamic Choice fields are backed by a custom command implementation (Dynamic Choice Command) that executes on a target agent system and send back data used to populate the drop-down. This can be used, for example, to pull data from third party system that may be needed to define a task (job names, process IDs, modes of operation, etc.).
<b>Dynamic Commands</b>	In addition to the standard instance commands (i.e. Cancel, Re-run, Retrieve Output, etc.), custom commands can be defined to extend the command capabilities of a Universal Extension Task instance.
<b>Output Only fields</b>	Fields can be defined with an Output Only restriction. These fields appear on the task instance as read-only and can be updated in real-time during task execution by the Extension instance running on the target agent system.
<b>Job Completion Output</b>	Universal Extensions support a new output type: EXTENSION. The EXTENSION output type is separated from STDOUT / STDERR and provides more advanced Success/Failure Output Contains processing.
<b>Encapsulation of 3rd party dependencies</b>	Universal Extensions support encapsulation of 3rd party modules/packages that are required for execution. This minimizes or eliminates manual deployment efforts.
<b>[NEW] Universal Event Support</b>	With 7.2.0.0, Universal Extensions can now be used to extend the Controller's monitoring capabilities through Universal Events and Universal Monitors/Universal Monitor Triggers.

## UIP-CLI

With the 7.2.0.0 release, the `uip-cli` tool has been enhanced with new features that make the process of creating, building, and uploading Extensions easier and convenient.

Specifically, with `uip-cli 1.2.0`, the ability to launch tasks and get the status/output of task instances is added. Additionally, a new Extension starter template called `ue-publisher` is added to showcase the Universal Event and Universal Monitor functionality released in 7.2.0.0.

See the [Development Environment Set-Up](#) and [Task Entry Point](#) documents for information regarding installing and using the CLI.

## UIP VS Code Extension

Alongside the `uip-cli` tool, the UIP Visual Studio Code Extension has also been enhanced to support launching tasks and retrieving status/output of task instances. The new `ue-publisher` starter template is also ready to be used with the VS Code Extension.

See the [Development Environment Set-Up](#) and [Task Entry Point](#) documents for information regarding installing and using the UIP VS Code Extension.

[Next >](#)

# Development Environment Set-Up

- [Requirements](#)
  - [Python Distribution](#)
  - [Universal Agent](#)
  - [Universal Controller](#)
  - [Platform](#)
  - [Code Editor](#)
  - [Environment](#)
  - [UIP-CLI Tool and UIP VS Code Extension](#)
  - [Environment Set-up](#)

## Requirements

### Python Distribution

Universal Extensions are implemented in Python. Therefore, an appropriate Python distribution must be available on the agent system where Universal Extensions run.

The portion of the Universal Extension implementation that is provided by Stonebranch (the **Universal Extension Base Package**) is compatible with Python versions 2.6 and higher. However, user developed Extensions are free to restrict Python version requirements to a smaller subset of Python versions in order to meet the needs of the extension.

### Universal Agent

Universal Extensions require a compatible Universal Agent for execution. The Universal Extension functionality is available in agent distributions starting with Version 7.0.0.

### Universal Controller

Universal Extensions require a compatible Controller. The Universal Extension functionality is available in Universal Controller distributions starting with Version 7.0.0.

### Platform

The Universal Extension Base Package is platform-independent. It is supported by the Universal Agent on the following platforms:

- AIX
- Linux x64 Debian
- Linux x64 Redhat
- Linux PPC64LE
- Linux S390x
- Solaris SPARC
- Solaris x64
- Windows X64

Universal Extensions can be developed on and targeted for any/all of the supported platforms.

### Code Editor

Any text editor can be used to develop Universal Extensions. However, an IDE with support for the Python language is recommended.

For the purposes of this document, the code editor of choice will be Visual Studio Code. From this point, documentation will be described in the context of a Visual Studio Code development environment however, the code and concepts will be applicable to any editor.

## Environment

As mentioned, the code editor used for this documentation will be Visual Studio Code. Visual Studio Code runs on multiple platforms (macOS, Linux, and Windows).

For this documentation, Visual Studio will be running in Windows with the "Remote WSL extension": (WSL is Windows Subsystem for Linux). This set-up is essentially a Linux development environment running in Windows.

## UIP-CLI Tool and UIP VS Code Extension

As mentioned in the previous document, `uip-cli` can be used to make the process of creating, editing, and deploying Extensions convenient. The UIP VS Code Extension takes this a step further by integrating the functionality of `uip-cli` into the VS Code IDE. However, the UIP VS Code Extension is specific to the Visual Studio Code IDE, whereas `uip-cli` is development environment agnostic.

The tutorials in the rest of the documentation will presented in the context of a Visual Studio Code development environment. Therefore, the tutorial will demonstrate the functionality using the UIP VS Code Extension. For additional details on working with `uip-cli` directly, refer to the documentation [here](#).

See the set-up instructions below for instructions regarding installing UIP VS Code Extension and `uip-cli`.

## Environment Set-up

1. [Install](#) Universal Controller version 7.0.0 or greater. This does not have to be on the development host; however, you must be able to access the Controller from the development host.
2. [Install](#) Universal Agent version 7.0.0 or greater on the target development host. Ensure that the agent is connecting to the same OMS server that is being used by the Controller.
3. Install Visual Studio Code (optional)
  - a. Install the [Python extension for Visual Studio Code](#) (optional)
  - b. Install the [Remote Development extension pack](#) (optional)
  - c. For additional details on setting up and using Visual Studio Code with WSL, Microsoft provides the following page: "[Get started using Visual Studio Code with Windows Subsystem for Linux](#)".
  - d. [Install](#) UIP VS Code Extension from the Visual Studio Code Marketplace. This can be performed from within the Visual Studio Code IDE by selecting the ViewExtensions and typing "UIP" into the search field.
  - e. **Note: `uip-cli` can be automatically installed as needed by the UIP VS Code Extension.**
4. [Install](#) `uip-cli` from the PyPi database (optional). The README on the project homepage contains instructions for downloading the CLI and verifying the install. **Note: If using the UIP VS Code Extension, the `uip-cli` install can be performed automatically as needed.**

[< Previous](#)   [Next >](#)

# Universal Extension API

- [Introduction](#)
- [Universal Extension Base Package](#)
- [Extension Class](#)
- [ExtensionResult Class](#)
- [Work Requests](#)
  - [Task Execution \(extension\\_start\)](#)
  - [Choice Commands](#)
  - [Dynamic Commands](#)
  - [Publishing Events](#)
- [Output Only Fields \(update\\_extension\\_status/update\\_output\\_fields\)](#)

## Introduction

Universal Extensions are developed by adhering to a simple API. This API is provided by the Universal Extension base package. The concise API documentation can be found here: [Universal Extension 1.2.0 API](#).

## Universal Extension Base Package

The Universal Extension Base Package (**universal\_extension**) is a Python package provided by Stonebranch that contains a collection of classes and functionality required to develop Universal Extensions. This package is distributed with the Universal Agent. It is automatically installed with all installation types for all platforms that support Universal Extensions. The package resides within a zip archive (**universal\_extension.zip**) and is installed to the following locations:

Windows	C:\Program Files\Universal\UAGSrv\uext\universal_extension.zip
Unix	/opt/universal/uagsrv/uext/universal_extension.zip

## Extension Class

An Extension implementation must provide a Python class named **Extension** that derives from the Universal Extension base class **UniversalExtension** and resides in a file named **extension.py**.

At a minimum, the **Extension** class must:

1. Reside in a file called **extension.py**.
2. Derive from base class **UniversalExtension**.
3. Provide a constructor method that, in turn, calls the initializer for the base class.
4. Implement method **extension\_start** (which is an override of a method defined in the **UniversalExtension** base class).
5. Return an instance of class **ExtensionResult** from method **extension\_start**.

The following code snippet contains the minimum requirements stated above. It has everything needed to execute as a Universal Extension. **It does not do anything, but it could execute without error.**

**Minimum Extension implementation requirement (extension.py)**

```

"""Minimum Universal Extension implementation."""
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult

class Extension(UniversalExtension):
    """Universal Extension sample module."""

    def __init__(self):
        """Init class."""
        super(Extension, self).__init__()

    def extension_start(self, fields):
        """Universal Extension primary operation."""

        return ExtensionResult()

```

## ExtensionResult Class

The **ExtensionResult** class is the required return type from all Universal Extension work requests. It can be seen in the code snippet above returning from **extension\_start**. Each type of work request supports a different set of parameters that are used to return information from that request type back to the Controller. The parameters are all optional, and appropriate defaults are set. However, it would usually not make sense to return without setting some parameters (as in the code snippet above). The use of **ExtensionResult** will be discussed further in the context of the specific work request types documented below.

## Work Requests

Universal Extensions support three types of work requests from associated Universal Extension tasks in the Controller: **Task Execution**, **Choice Commands**, and **Dynamic Commands**. These work requests are described below.

### Task Execution (extension\_start)

Task Execution is the primary operation of a Universal Extension task. This is the work request that results from launching a task. All Extensions must implement **extension\_start**. This is accomplished by adding a method named **extension\_start** with the following signature to the **Extension** class in file **extension.py**.

**def extension\_start(self, fields):**

The **extension\_start method** is the starting point for work that will be performed for a task instance. The **fields** parameter passes in a dictionary populated with field values from the associated task instance that was launched in the Controller. **The keys of the dictionary correlate with field names of the task instance** that is invoking the extension (defined in the associated Universal Template) and **the key values are the values from the associated form fields** when the task is launched. The Extension developer can perform any work that is required here: executing binaries on the system, connecting to other systems, etc.

Additional functionality required by the extension can be defined in other classes, files, and or packages. **extension\_start** is just required as the entry point.

Once the work is done, the **extension\_start** method **must** return an instance of **ExtensionResult**. **ExtensionResult** is a class provided in the **universal\_extension** base package for the purpose of returning from work requests.

The following parameters can be passed to the constructor of **ExtensionResult** for **extension\_start**:

Parameter	Type	Default	Description
rc	int, optional	0	This parameter represents the return code of the task instance that initiated the <b>extension_start</b> operation. The value returned is implementation-defined and therefore left up to the Extension developer. The value can be used by the "return code processing" of the task instance in the Controller to determine if the Extension task instance is perceived as completing with <b>Success</b> or <b>Failed</b> .
message	str, optional	None	This parameter specifies a short status string (error message or success message) to be sent to the "Status Description" field on the task instance form.
output_fields	dict, optional	None	Dictionary containing output fields. The parameter is a dictionary of output fields to be sent back to the controller for the associated Extension instance. Field names are implementation dependent and correlate with the Universal Template field names in the Controller's Template definition.
unv_output	str, optional	None	This parameter specifies the data that will be returned to the Controller for the command execution. It is interpreted as a UTF-8 encoded text object. If the output parameter is True, the output_data will be persisted in the Controller as a record under table ops_exec_output, and appearing as Universal Command output type from the task instance Output tab.

The following is an example of a simple extension\_start implementation returning a payload of "Hello world!".

```

extension_start

def extension_start(self, fields):
    """Universal Extension primary operation."""
    return ExtensionResult(unv_output = "Hello World!")
    
```

Upon execution, the extension\_start implemented above would result in the following output in the associated task instance in the Controller:

Type	Attempt	Output	Updated By	Updated
EXTENSION	1	Hello World!	ops.system	2022-03-02 15:00:38 -0500

## Choice Commands

Choice Commands support the task definition process in the Controller. They allow a drop-down "**Choice Field**" on a task definition form to call down to a Universal Extension on an agent system and retrieve data to populate the drop-down. This can be used to satisfy any number of scenarios but, typical use cases would be to pull values from a third-party system that may be needed to define the work the task is intended to perform (for example, job names, file names, modes of operation, etc.).

- **Choice Commands are optional and are not required in an Extension implementation.**
- An Extension may define any number of Choice commands.

Choice commands are defined by adding a method to the Extension class with an appropriate signature and decorating it with the **@dynamic\_choice\_command** decorator defined in the universal\_extension base package. Below is an example of a decorated function with the required signature:

**Choice Command in extension.py**

```
@dynamic_choice_command("choice_field_1")
def choice_command_1(self, fields):
    """Dynamic choice command."""
    return ExtensionResult(
        rc = 0,
        message = "Values for choice field 'choice_field_1'",
        values = ["Value 1", "Value 2", "Value 3"]
    )
```

In the code snippet above, "**choice\_field\_1**" corresponds to the field **Name** of a field of type **choice** with "**Dynamic Choice**" checked in an associated Universal Template in the Controller. The function name **choice\_command\_1** is arbitrary and could just as well have been **my\_choice\_command\_function**. The linkage between a Universal Template choice field and the backing Choice Command handler is made by matching the choice field's "Name" with the value in the `@dynamic_command` decorator. In this case "choice\_field\_1".

The screenshot shows the 'Field Details' configuration window for a choice field. The 'Name' field is highlighted with a red box and contains the text 'choice\_field\_1'. The 'Label' field contains 'Choice Field 1'. The 'Dynamic Choice' checkbox is also highlighted with a red box and is checked. Other visible settings include 'Type' set to 'Choice', 'Mapping' set to 'Choice Field 1', 'Allow Empty Choice' checked, 'Choice Sort Option' set to 'Sequence', and 'Column Span' set to '1'.

Once the work is done, the Choice Command method **must** return an instance of **ExtensionResult**. **ExtensionResult** is a class provided in the universal\_extension base package for the purpose of returning from work requests.

The following parameters can be passed to the constructor of ExtensionResult for returning from a Choice Command:

Parameter	Type	Default	Description

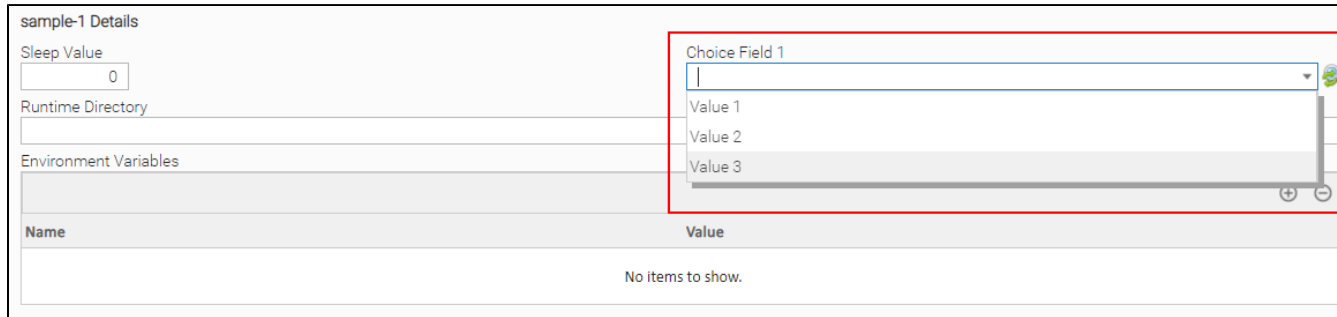
rc	<i>int, optional</i>	0	This parameter represents the return code of the 'choice' operation and determines whether the command is perceived as completing with success or failure by the Controller.  A value of 0 indicates success. All other values indicate an error condition and will be ignored by the Controller for form field population. However, the Controller will log the command response. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.
message	<i>str, optional</i>	Empty string	This parameter specifies a short status string (error message or success message) that will be logged by the Controller.
values	<i>list, optional</i>	Empty list	This parameter specifies a list which can be populated with a set of string values to be returned to the Controller and used to populate the associated dynamic choice field on an Extension task form.  The default value is an empty list.

The following is an example of a simple Choice Command implementation returning a list of hard coded values that would be used by the Controller to populate an associated Choice field drop-down:

**Dynamic Choice Command in extension.py**

```
@dynamic_choice_command("choice_field_1")
def choice_command_1(self, fields):
    """Dynamic choice command."""
    return ExtensionResult(
        rc = 0,
        message = "Values for choice field 'choice_field_1'",
        values = ["Value 1", "Value 2", "Value 3"]
    )
```

Execution, the Choice Command implemented above would result an associated Choice field drop-down in the Controller being populated:



## Dynamic Commands

Dynamic Commands are intended to support the functionality of the Primary Operation of the Universal Extension (task execution).

They can be made available to a task instance in any/all of the following task states:

- Defined
- Waiting
- Time Wait
- Held

- Exclusive Requested
- Exclusive Wait
- Resource Requested
- Resource Wait
- Execution Wait
- Undeliverable
- Queued
- Submitted
- Action Required
- Started
- Running
- Running/Problems
- Cancel Pending
- In Doubt
- Start Failure
- Confirmation Required
- Cancelled
- Failed
- Skipped
- Finished
- Success

Dynamic Commands can be used to return additional information for a task instance or to carry out some related (or unrelated) action on behalf of the task instance. The key to Dynamic commands is that they can pass instance specific field data values from the task instance form to the extension process that executes the command on the agent. This allows them to perform instance specific actions without the user needing to copy/paste values from the task instance into a new task to perform some action.

An example use case for this would be to implement a cancel command for work that was started in a third-party system. This would be independent of the task instance running (or completed) in the Controller. Instance specific values (job ID, process ID, etc. could automatically be pulled from the task instance and passed to the Command handler in the Universal Extension on the agent system).

**Dynamic Commands are optional and are not required in an Extension implementation.** An Extension may define any number of Dynamic Commands.

The following parameters can be passed to the constructor of ExtensionResult for returning from a Dynamic Command:

Parameter	Type	Default	Description
rc	<i>int, optional</i>	0	This parameter represents the return code of the Dynamic Command operation and determines whether the command is perceived as completing with success or failure by the Controller.  A value of 0 indicates success. All other values indicate an error condition and the command result will not be added to the Controller's Output tab. However, the Controller will log the command response. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.
message	<i>str, optional</i>	Empty string	The <b>message</b> parameter specifies a short status string (error message or success message) that will be logged by the Controller.
output	<i>bool, optional</i>	False	This parameter is a Boolean value that specifies if the command produced output that should be persisted and displayed under the task instance Output tab in the Controller task instance associated with the dynamic command invocation.  <b>The default value is False.</b>  This flag allows distinguishing between a command that does not produce output and a command that produces output but the output returned was empty.
output_data	<i>str, optional</i>	None	This parameter specifies the data that will be returned to the Controller for the command execution. It is interpreted as a UTF-8 encoded text object.  If the <b>output</b> attribute is <b>True</b> , the <b>output_data</b> will be persisted in the Controller as a record under table ops_exec_output, and appearing as <b>Universal Command</b> output type from the task instance <b>Output</b> tab.  <b>The default value is None.</b>
output_name	<i>str, optional</i>	None	This parameter is used to provide a custom name for the associated output data. The <b>output_name</b> (if provided) will be used in the presentation of the <b>output_data</b> by the Controller.

The following is an example of a simple Dynamic Command implementation returning a string **"Sample output"**.

#### Dynamic Command in extension.py

```
@dynamic_command("command_1")
def dynamic_command_1(self, fields):
    """Dynamic command."""
    return ExtensionResult(
        rc = 0,
        message = "command_1 output",
        output = True,
        output_data = 'Sample output',
        output_name = 'DYNAMIC_OUTPUT')
```

In the Controller, the Dynamic Command could be invoked from a task instance of the appropriate Universal Extension type.

1 sample-1 Task Instance		Last 48 hours	Refresh
Instance Name	Status	sample-1 Task Instance Commands ▶	Updated
sample-1	Success	Command 1	2022-03-02 15:21:57 -0500 2022-03-02 15:21:57
		Clear ▶	

Selecting the Dynamic Command for the task instance allows the adjustment of any dependent fields.

### Command 1 ✕

Agent \*

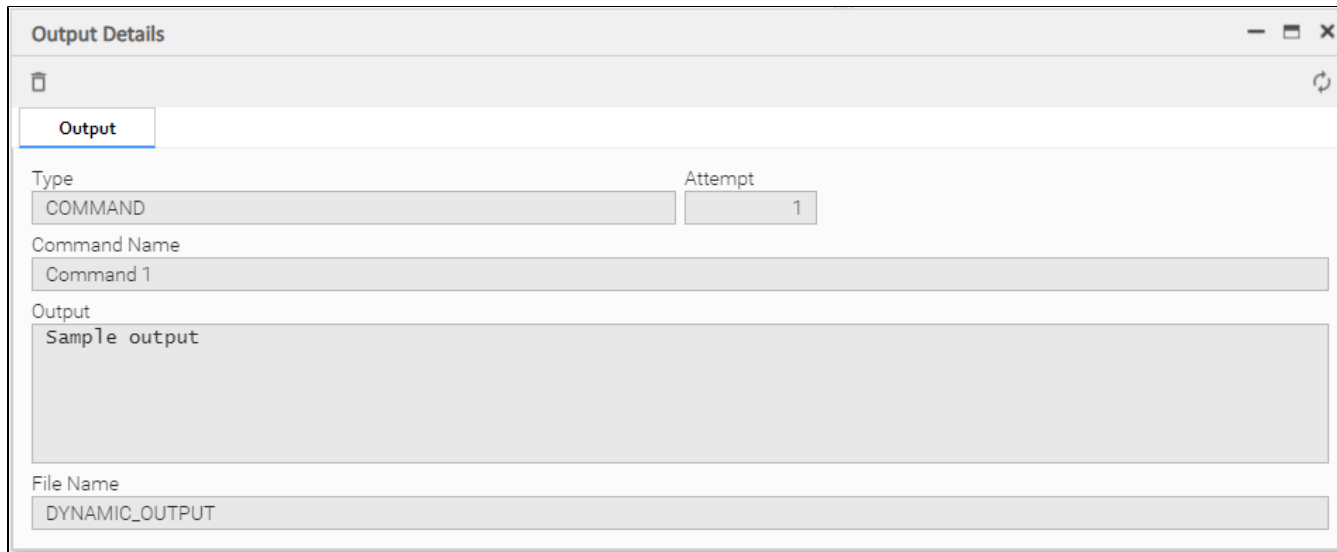
sbus44 - AGNT0001 ▼ 📄

Credentials

▼ 📄

Submit
Cancel

If the Dynamic Command produces output, a popup window appears to show the result of the Command.



Additionally, the results of Dynamic Commands are persisted with the task instance and can be viewed on the Output tab of the task instance.

Retrieve Output - sample-1			
4 Output			
Type	Attempt	Output	
COMMAND	1	Sample output	
EXTENSION	1	Hello World!	

## Publishing Events

With the 7.2.0.0 release, the ability to publish events was added to the Universal Extension API to extend the Controller's monitoring capabilities. This new addition has a multitude of use cases. For instance, Extensions can now be used to implement a custom Database Monitoring solution; Extensions publish events containing data from a database which Universal Monitors can use to detect a desired event.

To use this functionality, a Universal Event must be declared globally or locally. Shown below is the definition of a local Universal Event that is part of the `sample-1` Universal Template:

Universal Template | Fields | Commands | **Event Templates**

1 Event Template

Name	Label	Description
sample_event	Sample Event	Sample event to demonstrate Universal Event

**Event Template Details: Sample Event**

Event Template

**Details**

Name \*  Label \*

Description

Time To Live  Unmapped Attributes Policy

**Attributes**

Name	Label	Type
second	Second	Integer

The Universal Event is then tied to a Universal Monitor task as shown below:

The screenshot shows the configuration page for a Universal Monitor. The 'General' section includes fields for Name (sample-monitor-task), Version (1), Description, Member of Business Services, Resolve Name Immediately (unchecked), Hold on Start (unchecked), Virtual Resource Priority (10), Time Zone Preference (-- System Default --), and Hold Resources on Failure (unchecked). The 'Universal Monitor Details' section includes Event Type (Local), Universal Template (sample-1), Event Template (Sample Event), Universal Task Publisher (sample-1-task), and Time Scope (-- None --). The 'Universal Monitor Criteria' section has 'Match All' selected and a criterion: Second equals 28.

The `sample-monitor-task` above will end up in the "Success" status once it receives an event from the Extension attached to the `sample-1` Universal template with the "second" attribute set to 28.

On the Extension side, events are published using the **publish** method implemented in the **event** module accessible through the `universal_extension` module. The method signature is as follows:

**def publish(name, attributes, time\_to\_live=None):**

The **name** parameter specifies the target Event name, **attributes** is a dictionary specifying the Event's attributes as key-value pairs, and **time\_to\_live** specifies the maximum time the published event lives before it expires.

The code snippet below continuously publishes an event with the attributes containing the current second extracted from the time:

#### Publishing an event

```
while self.run:
    # get current seconds from the time
    second = datetime.now().second

    # Publish the event
    event.publish(
        'sample_event',
        {'second': second}
    )

    # Wait a second before continuing
    time.sleep(1)
```

Once the `sample-monitor-task` is launched, the `sample-1-task` will automatically be launched as well. The `sample-1-task` will publish an event every second. Once the condition specified in the `sample-monitor-task` is met, both tasks will end up in the "Success" state.

## Output Only Fields (`update_extension_status/update_output_fields`)

Universal Extensions provide support for an "Output Only" type field. This is a field defined in a Universal Template of type Extension that is given a restriction of Output Only.

The screenshot shows the 'Field Details' configuration window. It has a 'Field' tab selected. Under the 'General' section, the 'Name' field is highlighted with a red box and contains the text 'output\_1'. The 'Label' field contains 'Output 1'. Below this is a 'Hint' field and an 'Add To Default List View' checkbox. The 'Field Details' section includes a 'Type' dropdown set to 'Text', a 'Text Type' dropdown set to 'Plain', and a 'Mapping' dropdown set to 'Text Field 1'. At the bottom, the 'Restriction' section has two radio buttons: 'No Restriction' and 'Output Only', with 'Output Only' selected and highlighted by a red box. There is also a 'Preserve Output On Re-run' checkbox.

Output only fields can be updated at any time by a Universal Extension instance running a Task Execution request (`extension_start`) on an agent system. This provides a means for the task instance to send back relevant state information. The Output Only fields can be updated by the task instance as many times as needed.

The updating of Output Only fields is accomplished by calling the `update_extension_status` method implemented in the `UniversalExtension` base class or by calling `update_output_fields` from the `ui` module accessible through the `universal_extension` module. The `update_output_fields` method is more versatile as it can be called in another file/module that is part of the full Extension. Functionally, however, the methods are exactly the same.

Both methods have the following signature (only `update_output_fields` is shown from now on):

**`def update_output_fields(fields):`**

The `fields` parameter expects a dictionary populated with any/all Output Only fields that should be updated. The keys of the dictionary directly correlate with the Output Only field names specified in the associated Universal Template. The values of the dictionary are the values that will be used to update the corresponding Output Only field for the task instance in the Controller.

The following code snippet demonstrates two calls to update the same output only field ("output\_1"). To simulate work being done in-between the updates, a delay is created using the sleep function from the time module.

### Updating Output Only fields

```

sleep_value = fields.get('sleep_value', 5)

time.sleep(sleep_value)

# Update output fields.
out_fields = {}
out_fields["output_2"] = "Step One"
ui.update_output_fields(out_fields)

# Do some processing...
time.sleep(sleep_value)

# Update output fields.
out_fields = {}
out_fields["output_2"] = "Step Two"
ui.update_output_fields(out_fields)
    
```

The animation below shows the task instance form being updated (note the refresh icon at the top right of the task instance form was clicked continuously for the real-time updates):

sample-1 Details

Sleep Value

Choice Field 1

Output 1

Runtime Directory

Environment Variables

Name	Value
No items to show.	

[< Previous](#) [Next >](#)

# Task Entry Point

- [Introduction](#)
- [Step 1 - Create a New Extension Project using the UIP VS Code Extension](#)
  - [Select project folder](#)
  - [Selecting a starter template](#)
  - [Setting template parameter values for selected starter template](#)
  - [Step 1 supplemental - Create a New Extension Project using the CLI](#)
- [Step 2 - Deploy the Initial Extension using the UIP VS Code Extension](#)
  - [Set Project specific Environment Variables](#)
  - [Step 2 Supplemental - Deploy the Initial Extension using the CLI](#)
- [Step 3 - Create and launch a new 'ue-task-test' task](#)
  - [Step 3 Supplemental - Create and launch a new 'ue-task-test' task](#)
- [Step 4 - Modify the Extension and Deploy using the UIP VS Code Extension](#)
- [Step 5 - Reset the Extension Changes](#)

## Introduction

To demonstrate the process of Extension development, a sample Extension will be created and deployed using Visual Studio Code with the UIP Visual Studio Code Extension. The UIP Visual Studio Code Extension relies heavily on functionality provided by the `uip-cli` tool to enhance the UIP development experience. Therefore, a similar experience can be achieved with other code editors using `uip-cli` manually. This alternative use scenario will be documented as well.

The functionality of this sample Extension is contrived and serves no real purpose other than to illustrate the process of developing an extension that supports and utilizes all features available. The sample Extension, however, can be a good starting point for creating more complex Extensions.

On this page, we will cover the following:

1. Introduce the UIP Visual Studio Code Extension and `uip-cli` that will be used to create and configure the sample Extension.
2. Deploy the initial Extension without any changes and review the output.
3. Modify the sample Extension.
4. Deploy the modified Extension to the Controller and review the output.

Note that it is assumed the UIP Visual Studio Code Extension (or `uip-cli`) is already installed. See the previous document for installation instructions.

## Step 1 - Create a New Extension Project using the UIP VS Code Extension

As mentioned in [Development Environment Set-Up](#), this tutorial will be using Visual Studio Code running in Windows and connected to a WSL (Windows Subsystem for Linux) project environment.

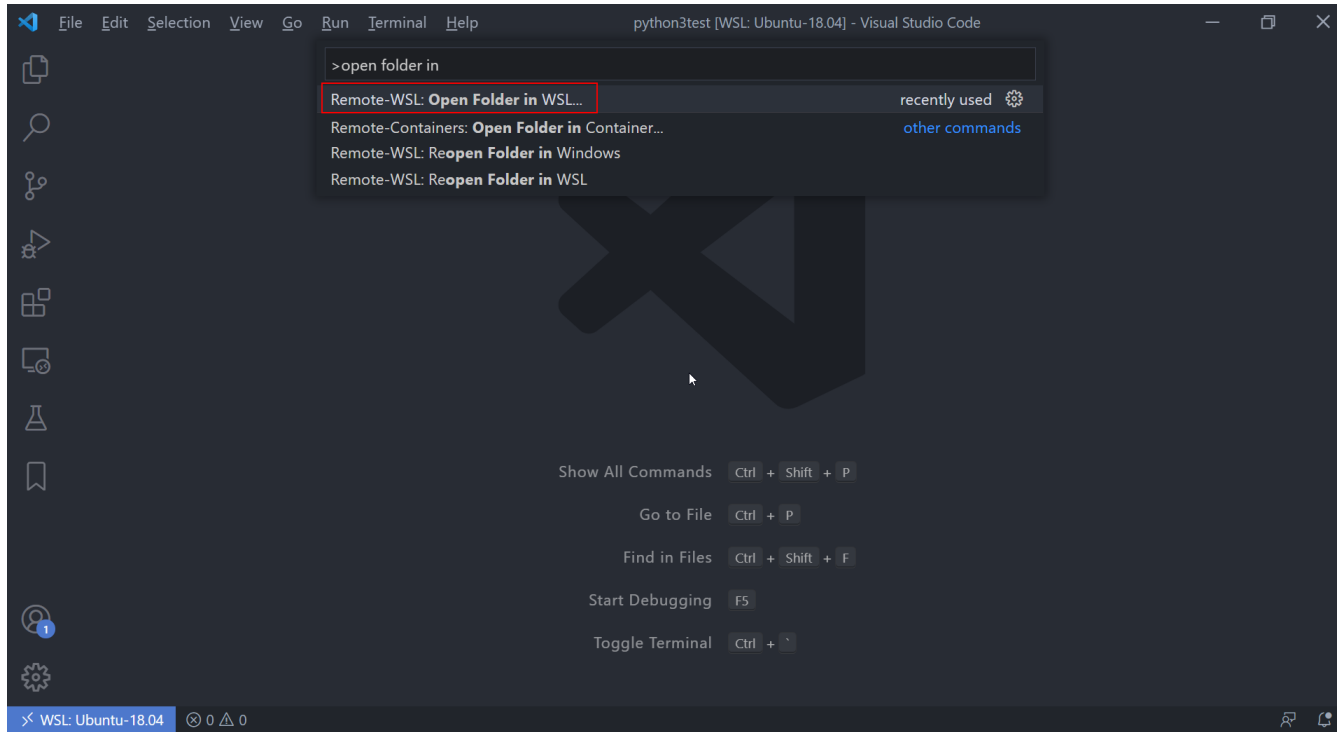
Initializing a new UIP project is a multi-step process.

1. Select a folder for VS Code to open for the new UIP project.
2. Select a starter template for the UIP project.
3. Iterate over the template parameters.

To begin, create a project directory (for example, `~/dev/extensions/sample-task`) in the WSL file system where the Universal Extension will be created.

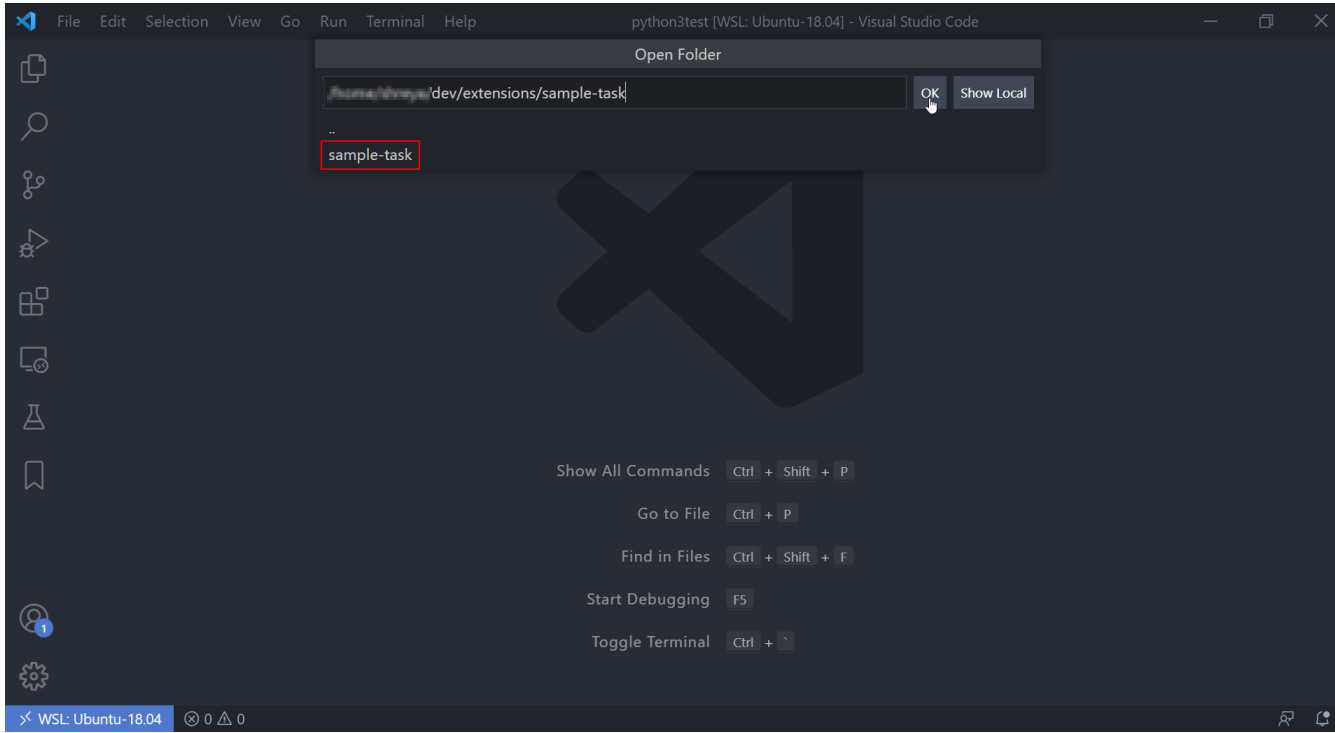
### Select project folder

Next, Use Visual Studio Code to open the **sample-task** folder in WSL:

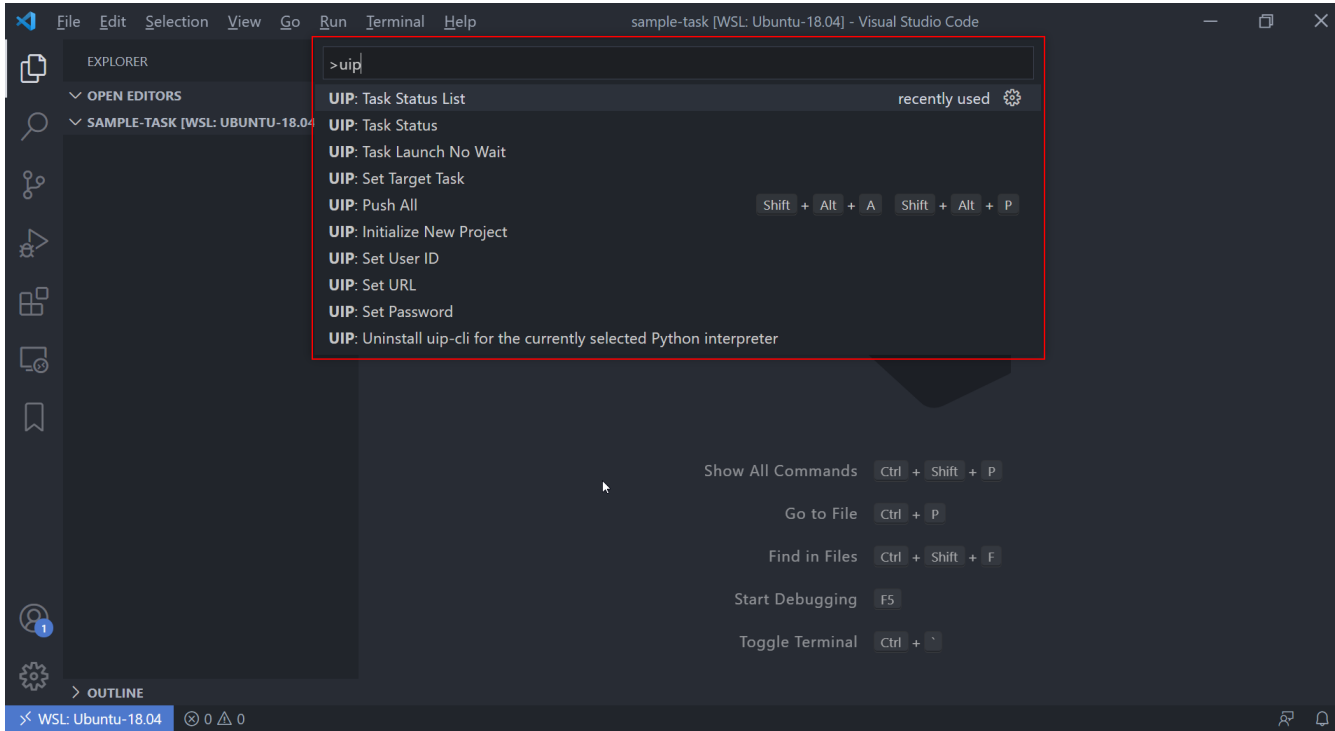


1. Open the Remote Window command pallet.
2. Select "Open Folder in WSL...".

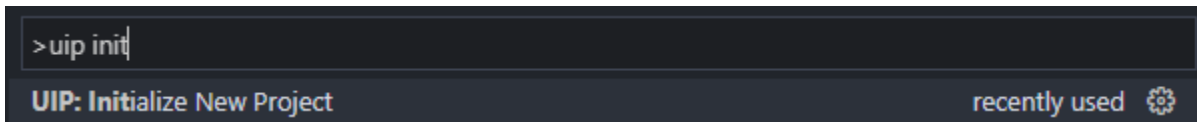
In the resulting dialog, navigate to the sample-task folder and click on Select Folder:



Now that VS Code is in the WSL environment, open the command pallet (ctrl+shift+p) and type "UIP". This will show all the UIP Extension commands:

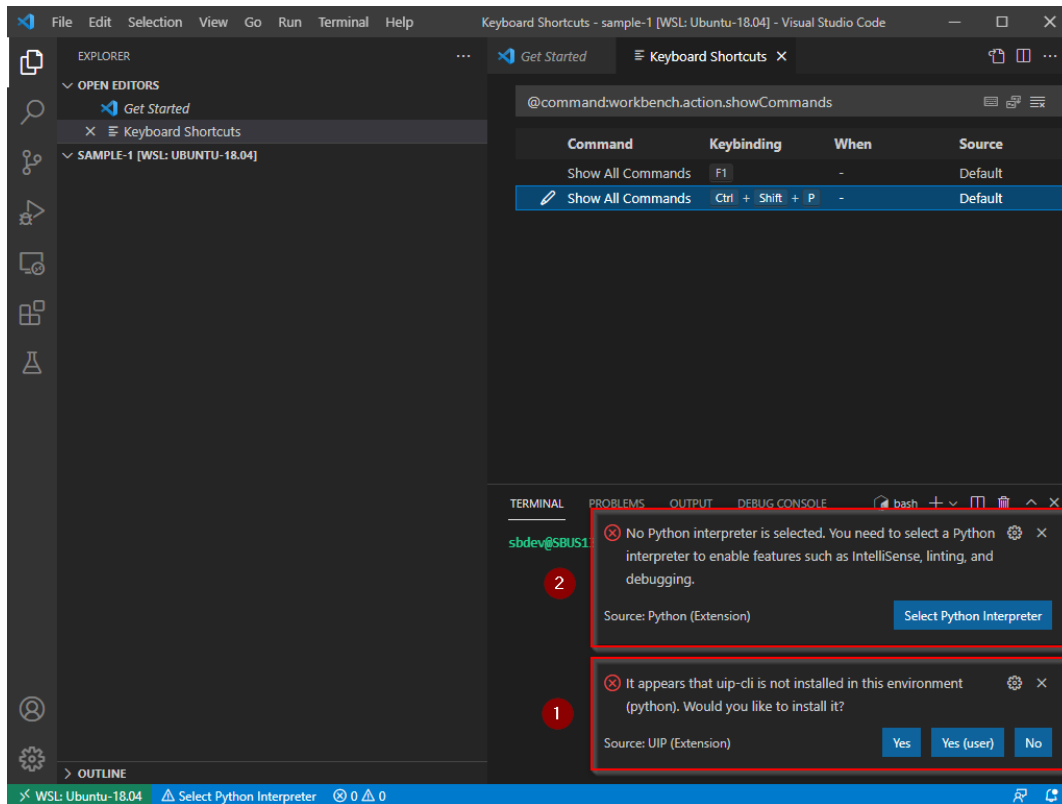


The command we are interested in now is **UIP: Initialize New Project**. Note that the commands can show up in varying orders depending on use. If the **UIP: Initialize New Project** is not visible, type "uip init" to further filter the displayed commands.



Select the **UIP: Initialize New Project** command.

If you have not yet installed uip-cli on the WSL system, you will likely be presented with two notifications after selecting **UIP: Initialize New Project**:

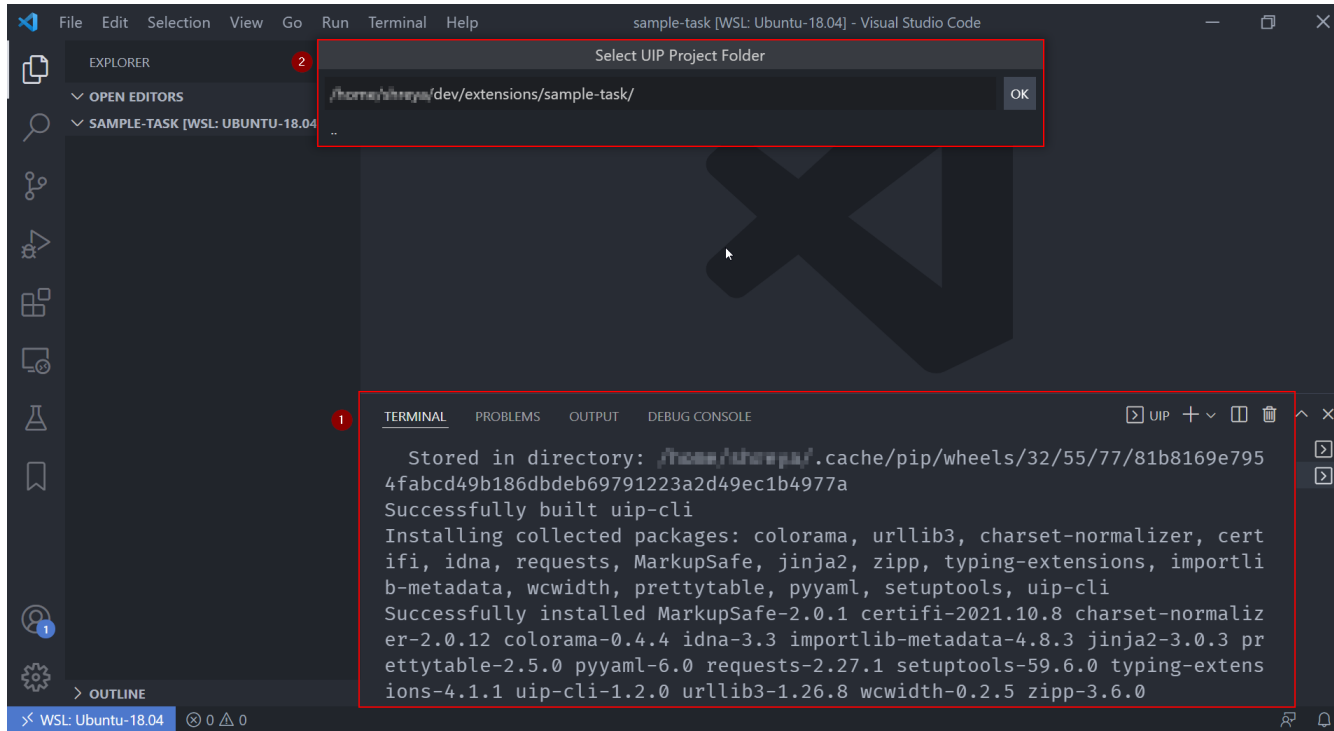


1. Indicates that uip-cli has not been installed to the selected Python environment and gives options for installing it. This notification comes from the UIP extension.
2. Indicates that a Python interpreter has not been selected. This notification comes from the Python extension

The first notification (1), indicates that uip-cli is not installed in **this** Python environment and gives some options for installation. Note that there is currently no Python interpreter for the project so, the system python ("python") is used as the default. If the indicated Python environment is agreeable, choose one of the install options.

The installation of uip-cli is performed by calling **pip install**. Selecting "**Yes (user)**" from the notification will cause pip to perform a **user scheme** install where uip-cli is installed to the Python user install directory for the platform. Typically ~/.local/, or %APPDATA%\Python on Windows. (See the Python documentation for [site.USER\\_BASE](#) for full details.). If the default Python is not acceptable or desirable for uip-cli installation, select "No". This will cancel the **UIP: Initialize New Project** operation and allow you to select a specific Python environment (from notification 2). Once an acceptable Python environment has been selected, open the command pallet again and select **UIP: Initialize New Project**. Once uip-cli has been installed to a Python environment, the notifications will no longer pop-up for UIP operations using that Python environment.

After an install option is selected, uip-cli will be installed to the target Python environment. The results of the installation procedure can be seen in a UIP terminal window that opens within VS Code. Additionally, the "Select UIP Project Folder" dialog will be displayed at the top of the VS Code window:



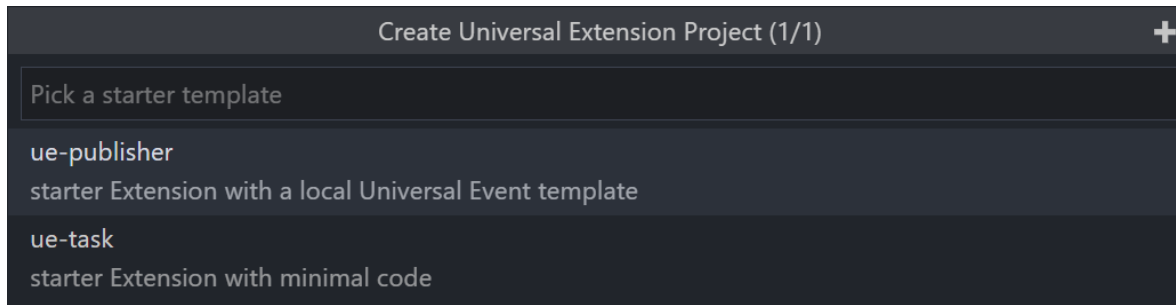
1. UIP terminal with output from uip-cli install.
2. The "Select UIP Project Folder" dialog.

At this point, the sample-task folder should be selected in the dialog so, go ahead and click "OK".

Note that, this dialog can be used to navigate to another folder. This is useful if you start the **UIP: Initialize New Project** process when VS Code is opened on a folder other than the target of initialization.

## Selecting a starter template

After selecting "OK", the UIP project initialization process starts on the selected folder and you are presented with a list of available "**starter templates**" for the UIP project. Starter templates contain boiler plate code to kick-start a project. At the time of this writing, two starter templates are available: **ue-task** and **ue-publisher**. For now, we will be working with **ue-task** (**ue-publisher** will be demoed later in the tutorial series), so go ahead and click on the **ue-task** template.

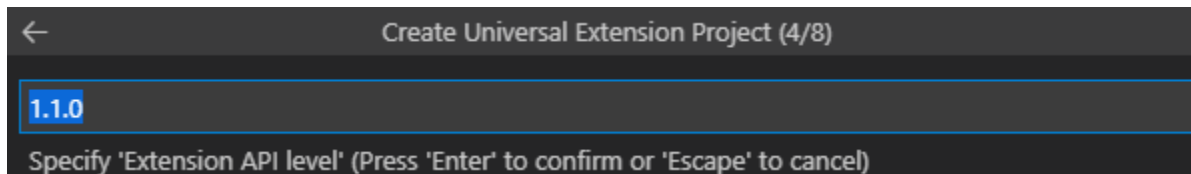
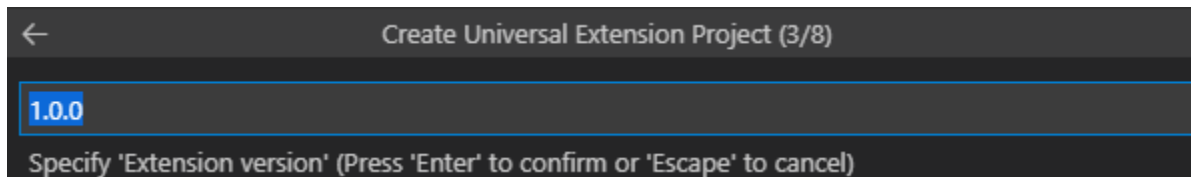
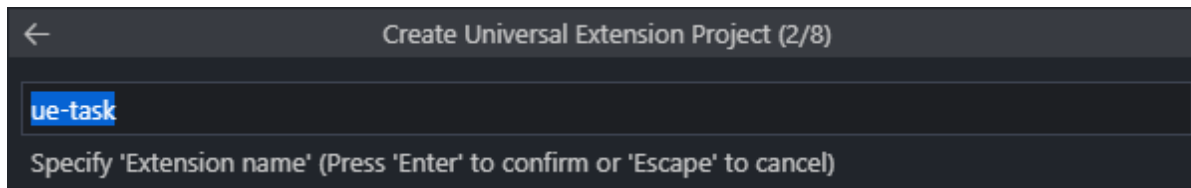


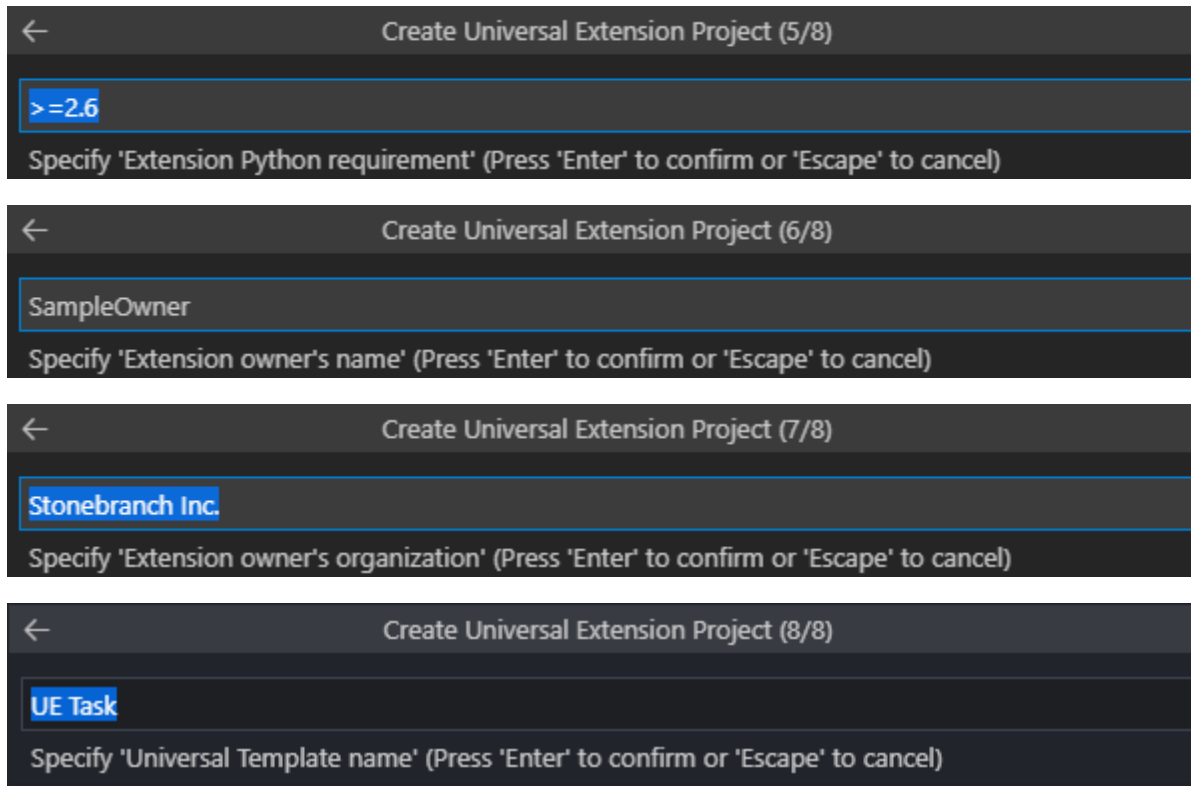
## Setting template parameter values for selected starter template

Starter templates contain parameters that allow you to supply project specific values into boilerplate template code at creation time.

Once the starter template is selected, the available parameters are presented one by one with default values pre-selected. For this tutorial, all parameters are suitable so, just pressing 'Enter' to select the default for each parameter would be sufficient. However, in the series of images below, the '**Extension Owner**' parameter was modified to use a value of "**SampleOwner**" (step 6/8). Notice that the dialog title will update to indicate which step you are on and how many steps are left to complete the dialog - (2/8), (3/8), (4/8), etc..

Note that pressing "Alt + left arrow" or clicking the icon in the top left of the image will return to the previous step (preserving the value of the current step).





After hitting enter on the last parameter:

1. A notification pops up to indicate the new project has been created.
2. The new project is created
3. The `up.yml` project configuration file is opened in an editor.
4. The generated `extension.py` source file is opened in an editor.

```

1 #
2 # (c) Copyright 2021 Stonebranch, Inc., All rights reserved.
3 #
4 # Stonebranch, Inc.
5 # Universal Integration Platform Command Line Utility Configuration File
6 # 1.2.0
7 #
8 # This configuration file specifies options used by the UIP CLI. The options
9 # and their values must be specified in proper YAML format as shown below.
10 #
11 #
12 # 'login' options:
13 #####
14 # Option: userid
15 # Default: None
16 #
17 # Used to specify the user id needed to log into the Controller
    
```

```

@sbus44:~/dev/extensions/sample-task$ uip init -t ue-task -e 'extension_name=ue-task' -e 'extension_version=1.0.0' -e 'extension_api_level=1.1.0' -e 'extension_requires_python=>=2.6' -e 'owner_name=Sample Owner' -e 'owner_organization=Stonebranch Inc.' -e 'universal_template_name=UE Task'
Successfully initialized "ue-task" template in "/home/.../dev/extensions/sample-task"
@sbus44:~/dev/extensions/sample-task$
    
```

When the “UIP: Initialize New Project” command completes, a project will be initialized in the selected folder with the following structure:

```

|-- sample-task           # Sample project directory
  |--setup.py            # Setup file for packaging extension
  |--setup.cfg          # Configuration file for configuring setup
  |
  |--__init__.py
  |
  |--.uip                # Folder used by CLI to validate directory
  |   |--config          # Configuration File Folder
  |   |--uip.yml         # Local Configuration File
  |
  |--src                 # Source directory for Extension implementation
  |   |--extension.py    # Extension implementation
  |   |--extension.yml   # Extension metadata
  |   |--__init__.py
  |   |
  |   |--templates
  |   |--template.json   # Universal Template JSON Definition File
    
```

At this point, the project is fully initialized.

## Step 1 supplemental - Create a New Extension Project using the CLI

Create a project directory (for example, `~/dev/extensions/sample-task`) where the sample-task Extension will be created, and `cd` into the directory.

As of now, the CLI offers two starter Extension templates called **ue-task** and **ue-publisher**. To see all the available starter Extension templates, type the `template-list` command:

```

@sbus44:~/dev/extensions/sample-task$ uip template-list
+-----+
| Extension Template | Description |
+-----+
| ue-publisher      | starter Extension with a local Universal Event template |
+-----+
| ue-task          | starter Extension with minimal code |
+-----+

```

Both the starter Extensions above can be configured before they are initialized.

To see the list of variables that can be used to configure one of the starter Extension templates, type the template name after the previous command.

Shown below are the list of variables for the **ue-task** starter template:

```

@sbus44:~/dev/extensions/sample-task$ uip template-list ue-task
+-----+
| Variable Name      | Default      | Description |
+-----+
| extension_name     | ue-task     | Extension name |
| extension_version  | 1.0.0       | Extension version |
| extension_api_level| 1.1.0       | Extension API level |
| extension_requires_python| >=2.6     | Extension Python requirement |
| owner_name         | Stonebranch | Extension owner's name |
| owner_organization | Stonebranch Inc. | Extension owner's organization |
| universal_template_name| UE Task    | Universal Template name |
+-----+

```

As shown in the image above, there are quite a few ways to configure the Extension. As for the sample Extension developed for this tutorial, we will work with **ue-task**, and only the `owner_name` option will be configured. Everything else will be set to the default value.

The Extension can be configured and initialized using the `init` command. There are three ways to configure the **ue-task** Extension template using the command line:

- Using the `-e` option multiple times:

```

@sbus44:~/dev/extensions/sample-task$ uip init -t ue-task -e 'owner_name=SampleOwner'
Successfully initialized "ue-task" template in "/home/shreya/dev/extensions/sample-task"

```

- Using a JSON string

```

@sbus44:~/dev/extensions/sample-task$ uip init -t ue-task -e '{"owner_name": "SampleOwner"}'
Successfully initialized "ue-task" template in "/home/shreya/dev/extensions/sample-task"

```

- Using a JSON/YAML file

- Create a YAML file called `vars.yml` and define the variables as follows:

```
1 owner_name: "SampleOwner"
```

- Use the YAML file to initialize the project:

```
@sbush44:~/dev/extensions/sample-task$ uip init -t ue-task -e '@vars.yml'
Successfully initialized "ue-task" template in "/home/shreya/dev/extensions/sample-task"
```

Pick one of the three ways shown above; it does not matter which one. Note that an optional, positional argument can be provided at the end of each of the three commands that specifies the directory in which the Extension will be initialized to. If the directory does not exist, the CLI will create it.

To verify that the Extension was configured as intended, open the `extension.yml` file (see directory structure below), and ensure the `owner_name` is `SampleOwner`.

This is the directory structure of the `sample-1` Extension:

```
|-- sample-task          # Sample project directory
  |--setup.py           # Setup file for packaging extension
  |--setup.cfg         # Configuration file for configuring setup
    |--__init__.py
    |--.uip            # Folder used by CLI to validate directory
      |--config        # Configuration File Folder
      |--uip.yml       # Local Configuration File
  |--src               # Source directory for Extension implementation
    |--extension.py    # Extension implementation
    |--extension.yml   # Extension metadata
      |--__init__.py
      |--templates
        |--template.json # Universal Template JSON Definition File
```

## Step 2 - Deploy the Initial Extension using the UIP VS Code Extension

Before deploying the Extension, let's take a look at the code to get a sense of the expected output. Open the `~/dev/extensions/sample-task/src/extension.py` in VS Code (It should already be open in the editor following the project initialization):

### extension.py

```
from __future__ import (print_function)
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import logger

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """
```

```

def __init__(self):
    """Initializes an instance of the 'Extension' class
    """
    # Call the base class initializer
    super(Extension, self).__init__()

def extension_start(self, fields):
    """Required method that serves as the starting point for work performed
    for a task instance.

    Parameters
    -----
    fields : dict
        populated with field values from the associated task instance
        launched in the Controller

    Returns
    -----
    ExtensionResult
        once the work is done, an instance of ExtensionResult must be
        returned. See the documentation for a full list of parameters that
        can be passed to the ExtensionResult class constructor
    """

    # Get the value of the 'action' field
    action = fields.get('action', [""])[0]

    if action.lower() == 'print':
        # Print to standard output...
        print("Hello STDOUT!")
    else:
        # Log to standard error...
        logger.info('Hello STDERR!')

    # Return the result with a payload containing a Hello message...
    return ExtensionResult(
        unv_output='Hello Extension!'
    )

```

The code presented above is ready to run without any modifications.

Note the following points of interest from the code above:

<b>Line 17</b>	Starts the implementation of the <code>extension_start</code> override method. This is the entry point for normal task execution and must be implemented by all Universal tasks.
<b>Line 36</b>	Extracts the value of the 'action' field passed down from the Controller (See the Info box below).
<b>Line 38-40</b>	Uses the standard Python <code>print</code> function to print the 'Hello STDOUT!' message to standard output stream if the selected action is 'print'
<b>Line 42-43</b>	Uses the <code>ExtensionLogger</code> class (exposed as part of the <code>universal_extension</code> file) to log the 'Hello STDERR!' message to the standard error stream if action is anything other than 'print'

**Line 46** Uses the `unv_output` parameter of the `ExtensionResult` class to send the 'Hello Extension!' string to the EXTENSION output payload associated with the task instance in the Controller.

What is the 'action' field?



You may have noticed the **template.json** file in the sample-task directory structure above. That is the Universal Template, in JSON format, that the Controller uses to render the template UI.

One of the things **template.json** defines is the 'action' field of type Choice with two possible values: 'print' and 'log'. Soon, the template will be pushed out to the Controller where we will be able to see it visually.

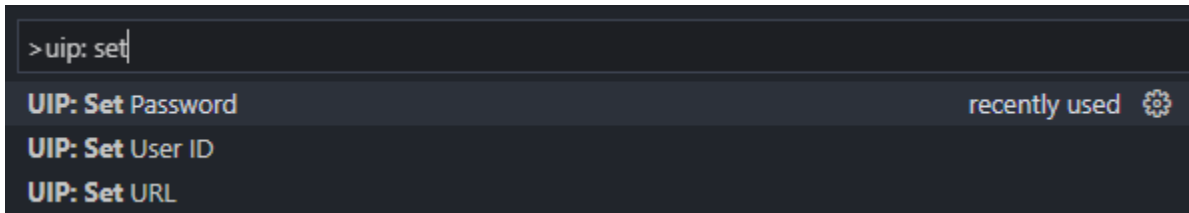
For now, keep in mind that the Extension can use fields defined in the Universal Template.

Now, let's deploy the Extension. Note that Extensions are stored in the Controller as Universal Templates of type Extension.

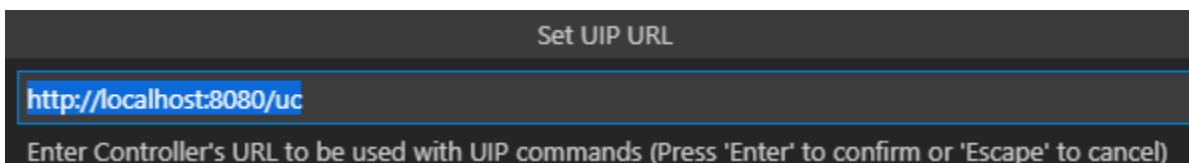
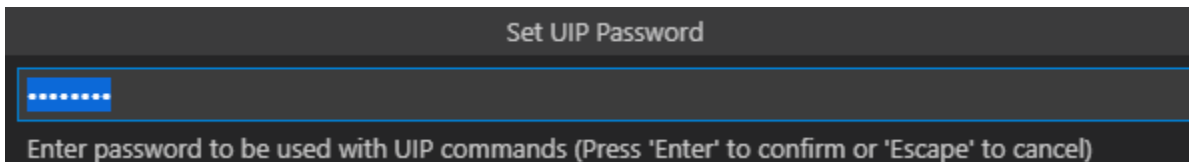
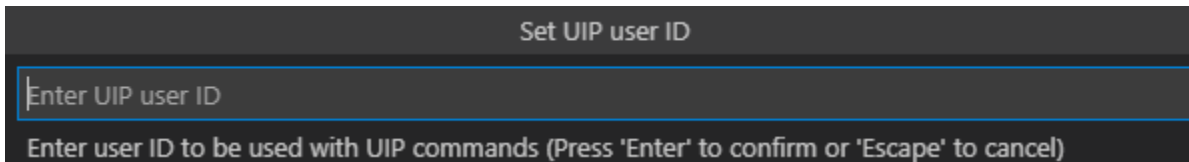
To connect to the Controller, the CLI needs the Controller's URL along with userid and password. To avoid typing the same information multiple times, the URL and userid can be stored in the local configuration file (`~/dev/extensions/sample-task/.uip/config/uip.yml`), and the password can be stored as an environment variable. Alternatively, all values can be set as environment variables and, using the UIP VS Code Extension, those values can be persisted with the project and reloaded as needed.

## Set Project specific Environment Variables

Open the VS Code command pallet (`ctrl+shift+p`) and type "**uip set**". This will show a list of UIP commands for setting environment variables:

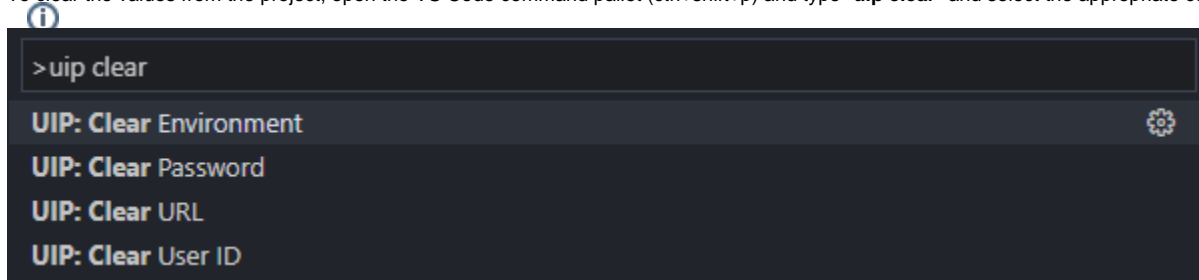


Select each command (one at a time) and enter appropriate values to connect to your Controller:



Once these values are set, they will be persisted to project specific storage and be reloaded each time the VS Code project is opened.

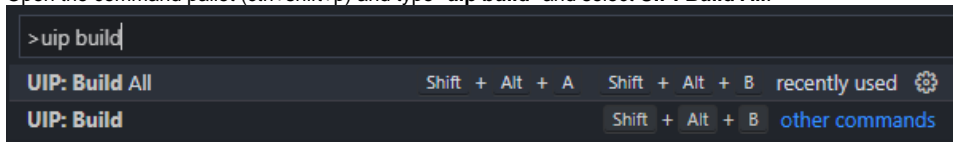
To clear the values from the project, open the VS Code command pallet (ctrl+shift+p) and type "**uip clear**" and select the appropriate command:



**UIP: Clear Environment** clears user ID, Password, and URL with a single command.

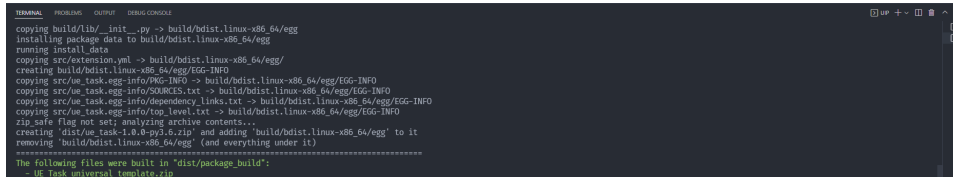
Using the UIP VS Code Extension, there are two primary ways to deploy the Extension: using the **UIP: Push** command, or the **UIP: Build All** command followed by the **UIP: Upload All** command. The latter method will be shown once below, but in subsequent deployments throughout the rest of the tutorial, the **UIP: Push** command will be used.

- Open the command pallet (ctrl+shift+p) and type "**uip build**" and select **UIP: Build All**:

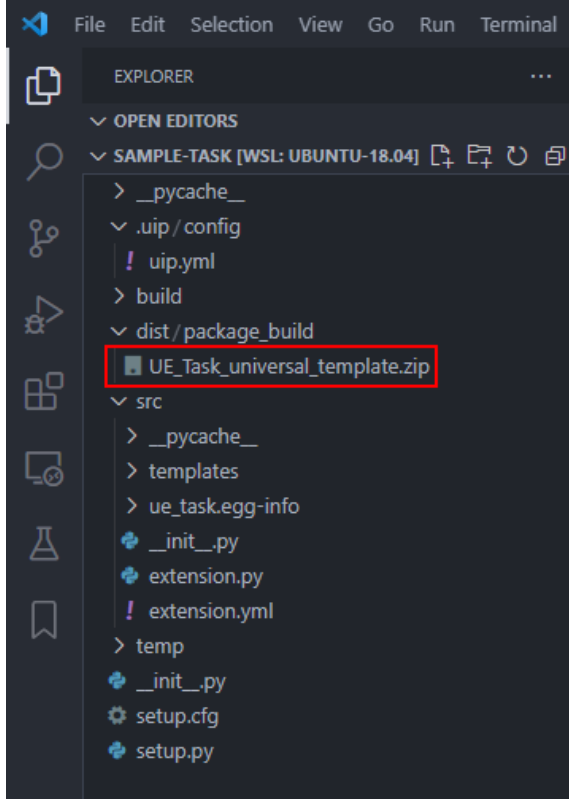


The **UIP: Build All** command will build the full package (Extension + Universal Template). Recall that the Universal Template is called **UE Task**, and it does not exist in the Controller. Therefore, the entire package must be built and uploaded.

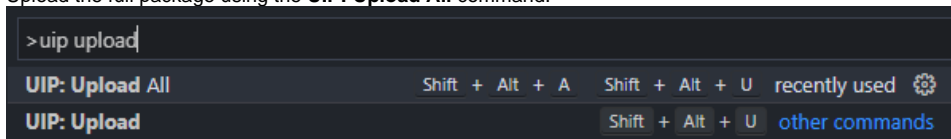
When the command is selected, a UIP terminal window will be opened in VS Code and the **uip-cli** command '**uip build -a**' will be executed. The results of the command can be reviewed in the terminal:



- Verify the full package was built by going to the `./sample-task/dist/package_build` folder, and making sure `UE_Task_universal_template.zip` file exists:

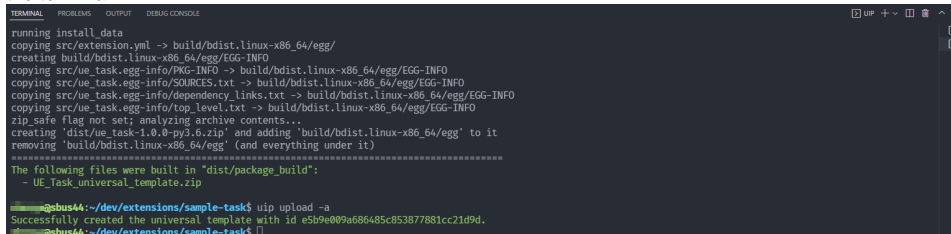


- Upload the full package using the **UIP: Upload All** command:

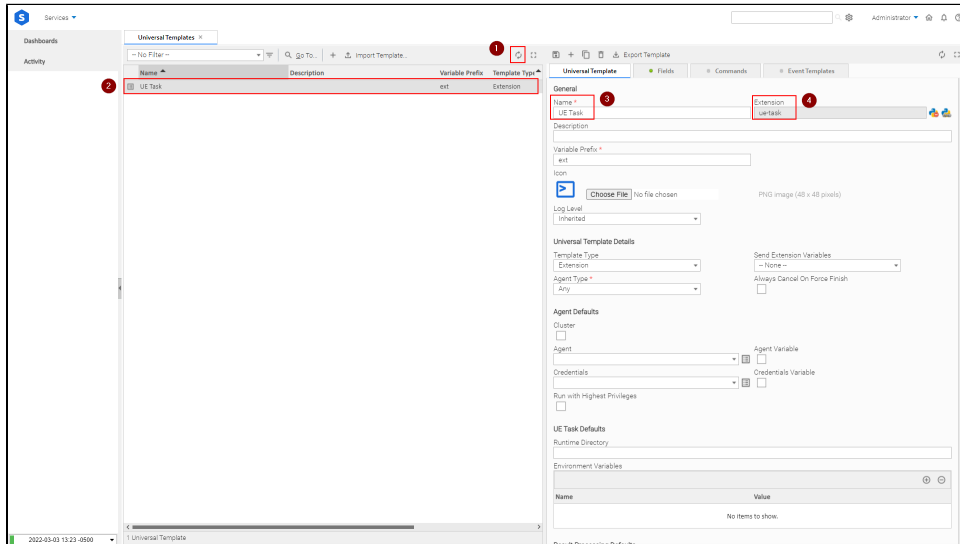
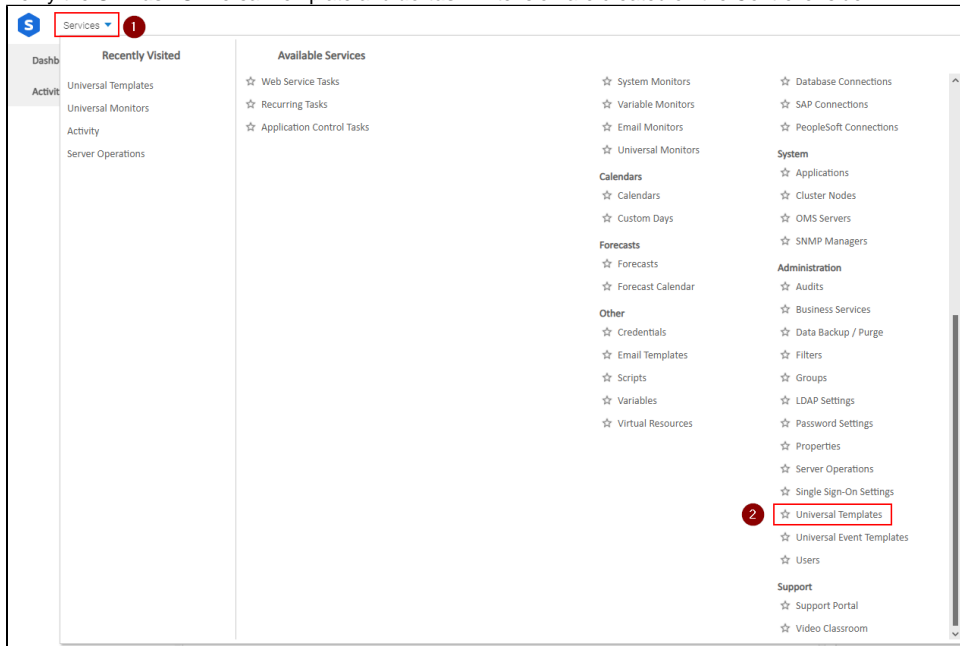


Once again, the **UIP: Upload All** option indicates the full package needs to be uploaded.

When the command is selected, a UIP terminal window will be opened (if not already open) and the `uip-cli` command `'uip upload -a'` will be executed. The results of the command can be reviewed in the terminal:



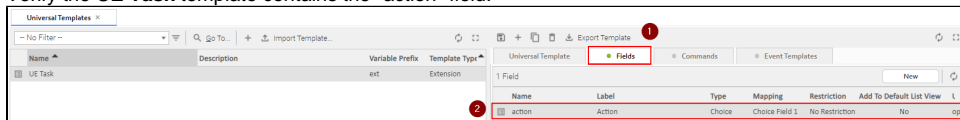
- Verify the **UE Task** Universal Template and **ue-task** Extension are created on the Controller side:



- 1) Click the "Refresh" icon, if necessary
- 2) Verify the **UE Task** template is listed
- 3) Verify the template name is **UE Task**

4) Verify the **ue-task** Extension is attached to the template

- Verify the **UE Task** template contains the "action" field:



- If you are curious, double-click the "action" field and explore the field details. In the "Choices" tab, you will see the 'print' and 'log' choices

## Step 2 Supplemental - Deploy the Initial Extension using the CLI

To connect to the Controller, the CLI needs the Controller's URL along with userid and password. To avoid typing the same information multiple times, the URL and userid will be stored in the local configuration file, and the password will be stored as an environment variable.

- Recall that the local configuration file is located in `~/dev/extensions/sample-task/.uip/config/uip.yml`. Open the file, and enter the URL and userid information near the end of the file. Shown below is a sample snippet:

```
#####
#
# Configuration Options Section
#
#####
#
userid: admin
url: http://localhost:8080/uc
# variables: <JSON string | JSON/YAML file>
# build-all: <yes/no>
# upload-all: <yes/no>
# push-all: <yes/no>
# template-name: <name>
```

- Go back to the command line, and create an environment variable called `UIP_PASSWORD` with the password as the value.
  - If using Windows, then in CMD: `set UIP_PASSWORD=<your password>`
  - If using Unix/Linux, then in the terminal: `export UIP_PASSWORD=<your password>`

Using the CLI, there are two primary ways to deploy the Extension: using the **push** command, or the **build** command followed by the **upload** command. The latter method will be shown once below, but in subsequent deployments throughout the rest of the tutorial, the **push** command will be used.

- `cd` into the **sample-task** directory, if not already in there. The CLI can only execute the deployment commands if it is called from the directory containing the **.uip** folder. In this case, it is **sample-task**.
- Issue the build command as follows:

```
$ uip build -a
```

The **-a** command will build the full package (Extension + Universal Template). Recall that the Universal Template is called **UE Task**, and it does not exist in the Controller. Therefore, the entire package must be built and uploaded.

- Verify the full package was built by going to the `~/dev/extensions/sample-task/dist/package_build` folder, and making sure **UE\_Task\_universal\_template.zip** file exists.
- Upload the full package as follows:

```
$ uip upload -a
```

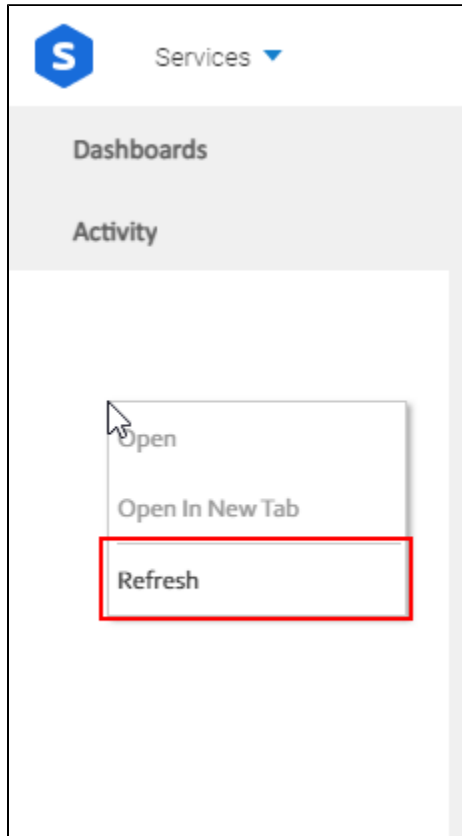
Once again, the **-a** option indicates the full package needs to be uploaded.

- Verify the **UE Task** Universal Template and Extension are created on the Controller side using the procedure shown above the supplemental

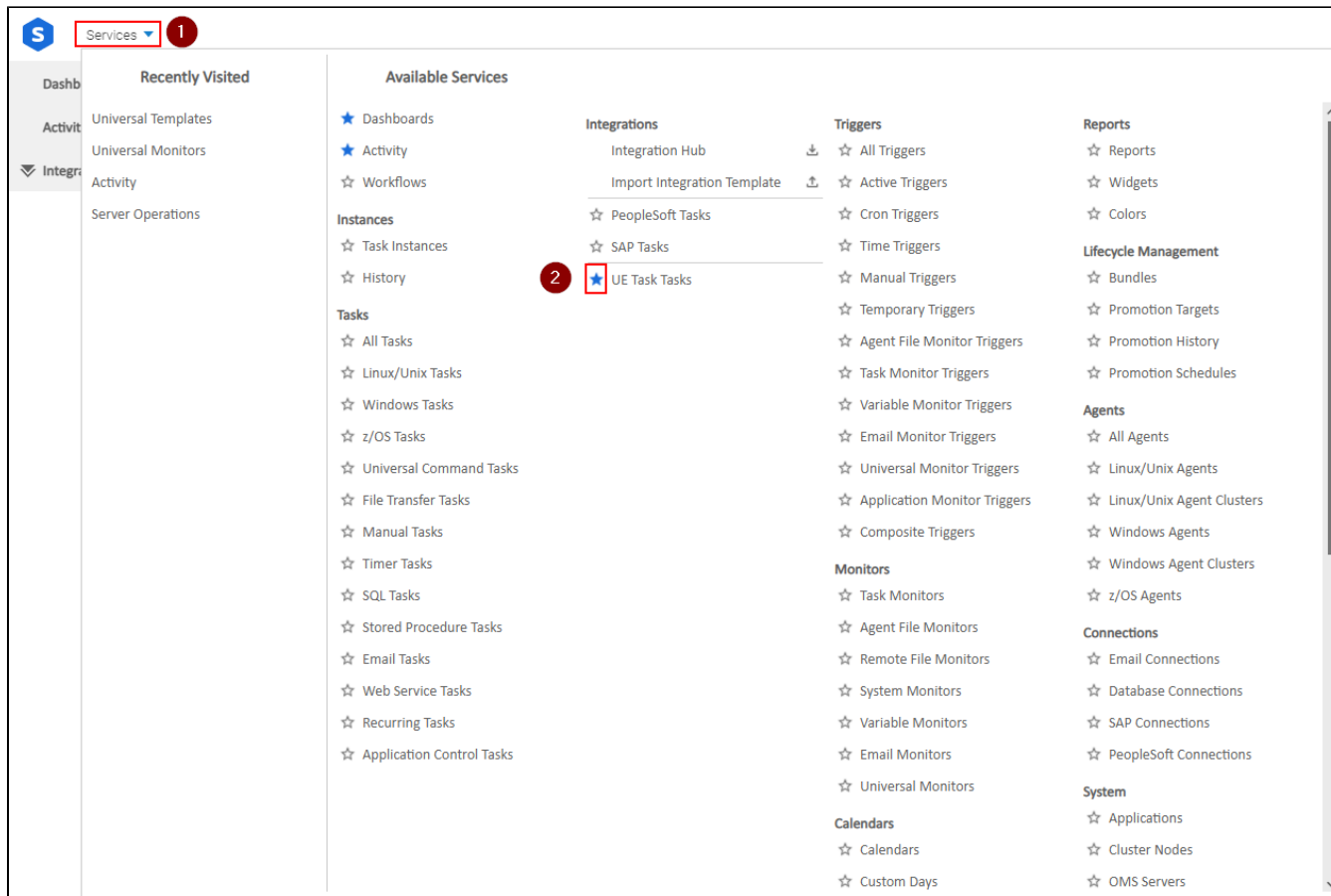
### Step 3 - Create and launch a new 'ue-task-test' task

We are now ready to create a new task to test the initial **ue-task** Extension. In order to make the new **UE Task** task type available in the Integrations section of the Controller's Services menu, we must first refresh the Navigation Tree.

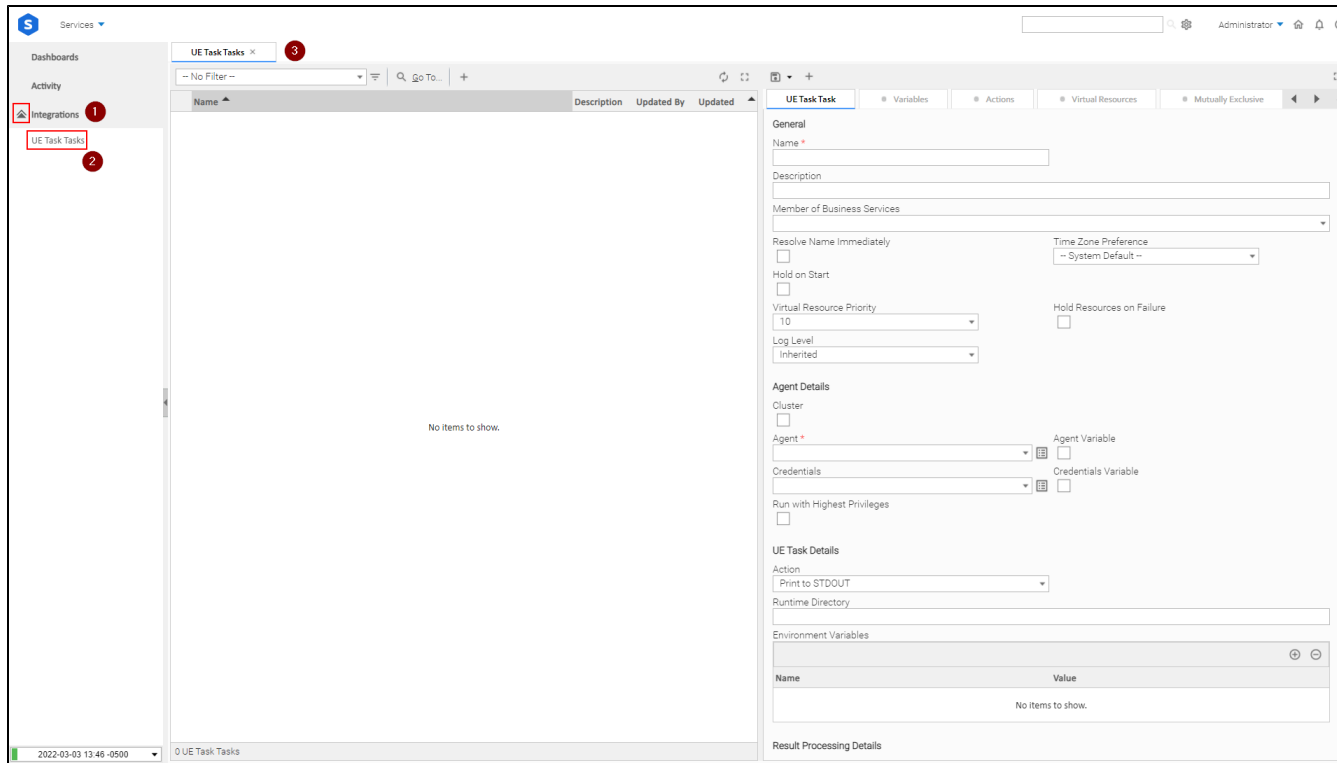
To do this, right click anywhere in the Navigation Tree and select **Refresh**



Click the "Services" menu and click the star/favorites icon next to "UE Task Tasks"

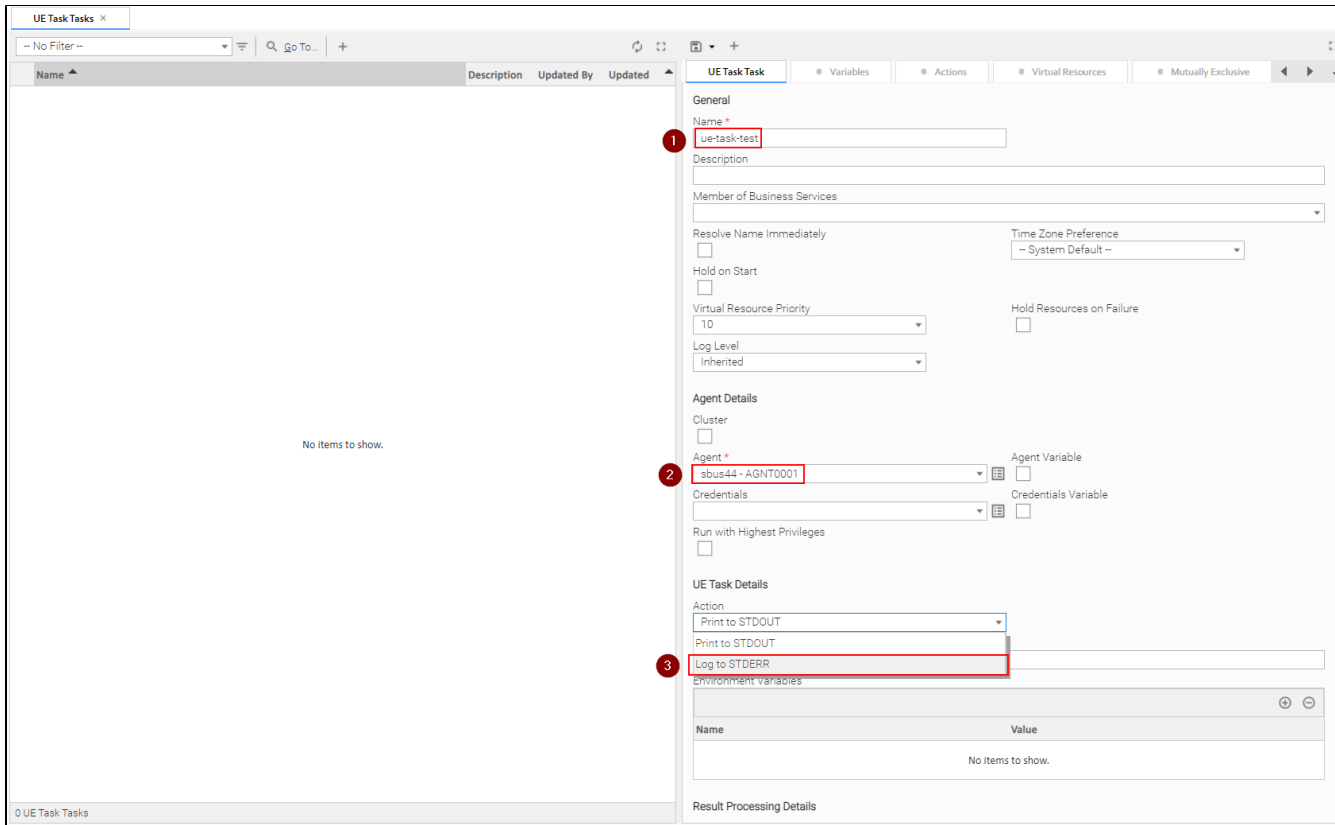


Clicking the star/favorite icon will make the "Integrations" dropdown visible in the Navigation tree. Click the "Integrations" dropdown and select "UE Test Tasks". This should open the "UE Test Tasks" tab

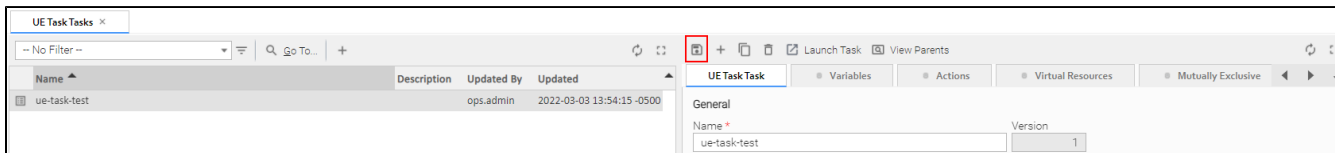


Create the task as show below

1. Give the task a name (for example, **ue-task-test** as shown)
2. Select an **active** agent of **version 7.0.0.0 or higher** for the task to run on.
3. Select the "Log to STDERR" action

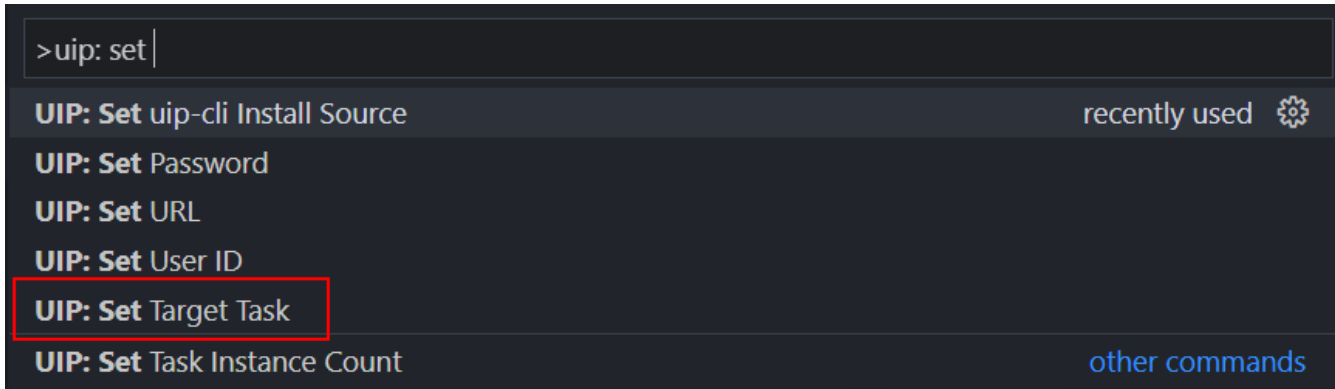


Save the task.

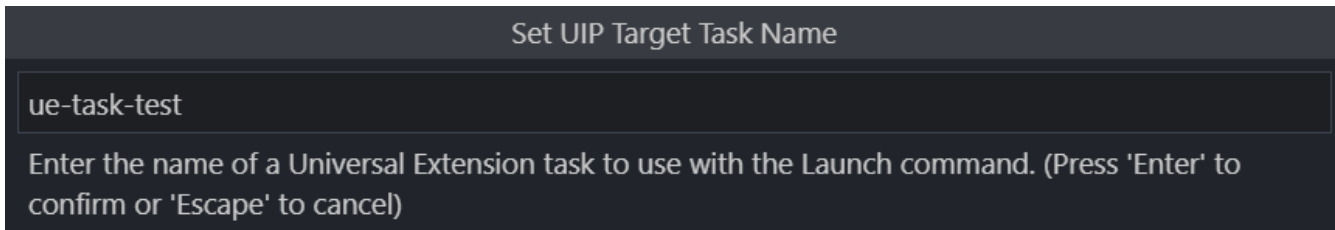


Now, we are ready to launch the task. This can either be done manually through the Controller or using the VS Code Extension as shown below.

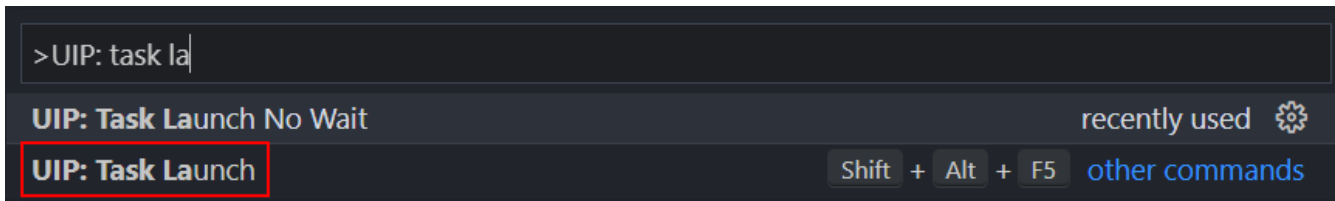
Open the command palette (ctrl + shift + p), search for "UIP: Set", and select "UIP: Set Target Task":



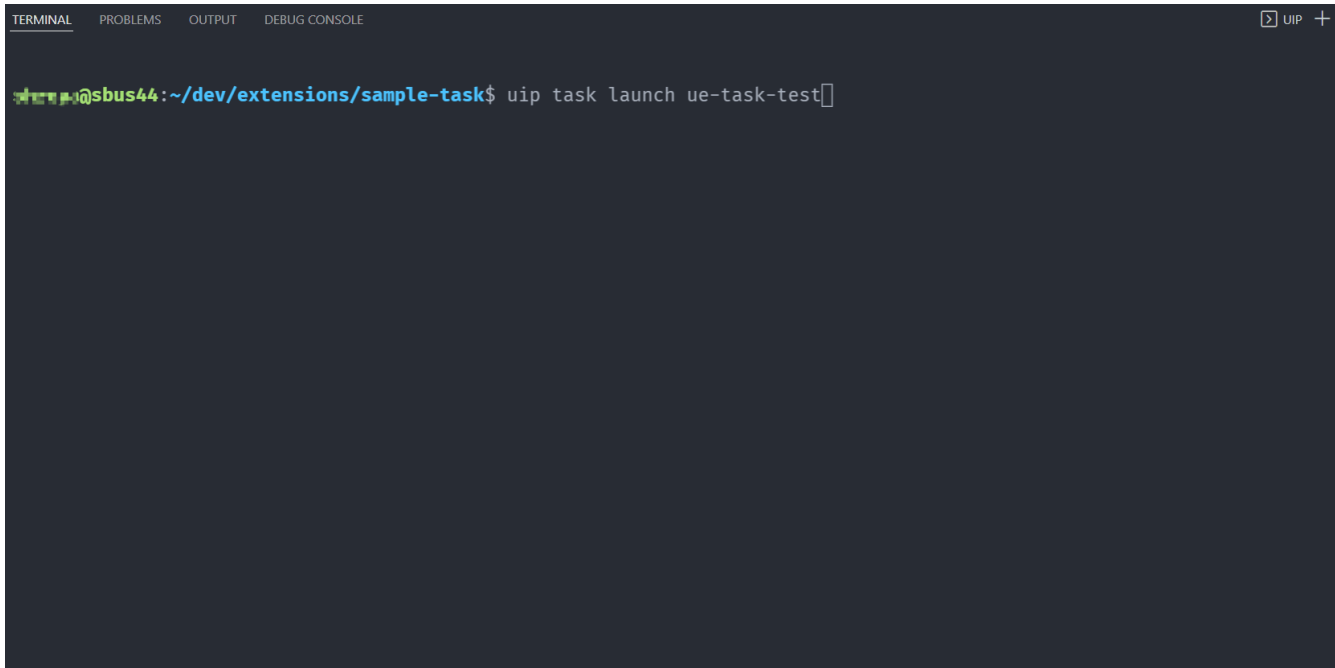
Type the task name and press enter:



Launch the task by opening the command palette and selecting "UIP: Task Launch"



The terminal should be opened, and the uip-cli will be invoked to launch the task. You should see the task status transition from Queue to Running to Success as shown below:



```
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE UIP + v
@sbus44:~/dev/extensions/sample-task$ uip task launch ue-task-test
```

Upon task completion, the task output will be retrieved and printed to the screen.

Notice that the STDOUT output section is empty since we are not printing anything using the `print()` function.

The EXTENSION output section contains "Hello Extension!" which is exactly what we coded in line 46 of **extension.py**

The STDERR output contains "Hello STDERR!" because we selected the "Log to STDERR" action in the task form. In the next step, we will change the action to "print to STDOUT" instead and observe the behavior.

### Step 3 Supplemental - Create and launch a new 'ue-task-test' task

The task creation process is exactly the same as above. As for launching the task, run the command shown in the GIF animation above.

## Step 4 - Modify the Extension and Deploy using the UIP VS Code Extension

So far, we have pushed the initial Extension, created a task for it, and tested the Extension code. Now, the **extension.py** file will be slightly modified to demonstrate the process of deploying a modified Extension using the UIP VS Code Extension.

Edit the `~/dev/extensions/sample-task/src/extension.py` file to contain the following code:

#### **extension.py**

```
from __future__ import (print_function)
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
```

```

from universal_extension import logger

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """

    def __init__(self):
        """Initializes an instance of the 'Extension' class
        """
        # Call the base class initializer
        super(Extension, self).__init__()

    def extension_start(self, fields):
        """Required method that serves as the starting point for work performed
        for a task instance.

        Parameters
        -----
        fields : dict
            populated with field values from the associated task instance
            launched in the Controller

        Returns
        -----
        ExtensionResult
            once the work is done, an instance of ExtensionResult must be
            returned. See the documentation for a full list of parameters that
            can be passed to the ExtensionResult class constructor
        """

        # Get the value of the 'action' field
        action = fields.get('action', [""])[0]

        # Print/Log the message 3 times
        for _ in range(3):
            if action.lower() == 'print':
                # Print to standard output...
                print("Hello STDOUT!")
            else:
                # Log to standard error...
                logger.info('Hello STDERR!')

        # Return the result with a payload containing a Hello message...
        return ExtensionResult(
            unv_output='Hello Extension!'
        )

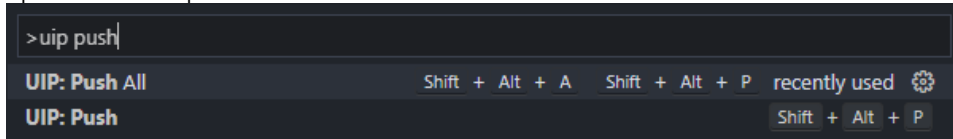
```

The changes made above are:

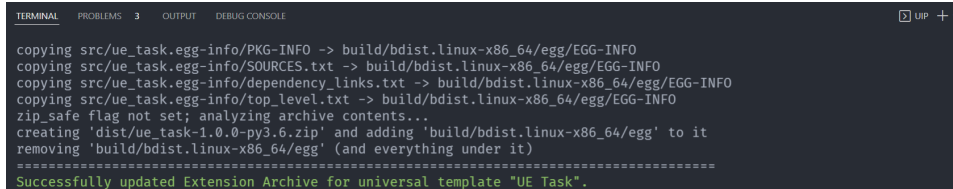
<b>Line 38</b>	For-loop that runs three times. In each iteration, either the STDOUT or STDERR message will be printed
----------------	--

Now, we need to deploy the modified Extension to the Controller. Unlike the first time where the entire package needed to be deployed, only the Extension needs to be deployed this time since that is all we changed.

- Open the command palette and run the **UIP: Push** command:



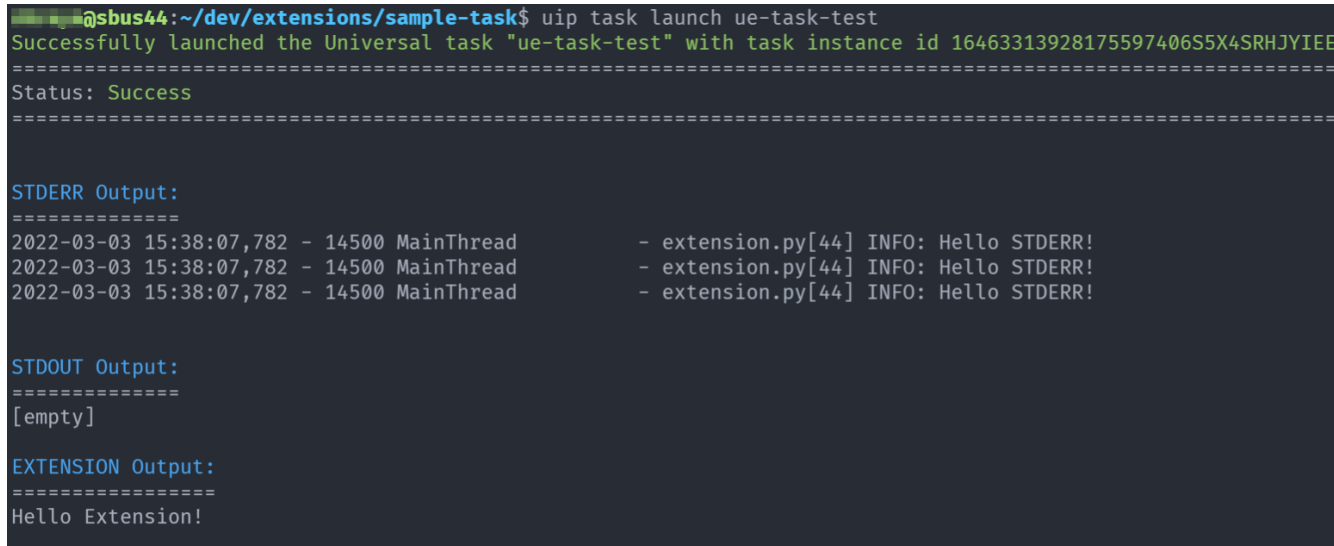
When the command is selected, a UIP terminal window will be opened (if not already open) and the **uip-cli** command 'uip push' will be executed. The results of the command can be reviewed in the terminal:



Instead of **UIP: Build All** followed by **UIP: Upload All**, this time **UIP: Push** was used as it is essentially just a combination of **UIP: Build** and **UIP: Upload**. Notice that the **UIP: Push All** command was not used given that we are only pushing the Extension and not the entire package. If the **template.json** file was modified locally, then the **UIP: Push All** command should be used to update the entire package on the Controller side.

If you are wondering how the **UIP: Push** command knows what Universal Template to push the **ue-task** extension to, open the `~/dev/extensions/sample-task/src/templates/template.json` file, and there will be a field called "name" with the value "UE Task". The "UE Task" is the name of the Universal Template. Internally, the CLI reads the **template.json** file and extracts the name from it.

To verify the **ue-task** Extension was updated successfully, run the **ue-task-test** task again as shown in the previous step. The output should look similar to the one shown below:

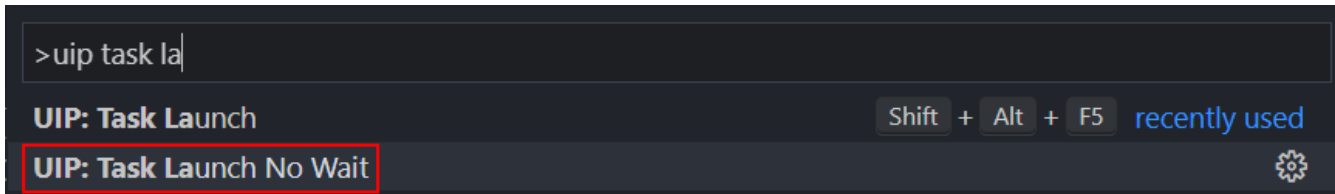


Notice that the STDOUT in the output above is still empty as expected. Now, we will change the value of the "Action" field in **ue-task-test**. Navigate to the task and change the "Action" to "Print to STDOUT":

The screenshot shows the configuration page for a task named 'ue-task-test'. The 'General' section includes fields for Name (ue-task-test), Description, Member of Business Services, Resolve Name Immediately (checkbox), Hold on Start (checkbox), Virtual Resource Priority (10), and Log Level (Inherited). The 'Agent Details' section includes Cluster (checkbox), Agent (sbus44 - AGNT0001), Credentials, and Run with Highest Privileges (checkbox). The 'UE Task Details' section shows the Action dropdown menu open, with 'Print to STDOUT' selected and highlighted by a red rectangular box. Other options in the dropdown include 'Log to STDERR' and 'Log to STDOUT'.

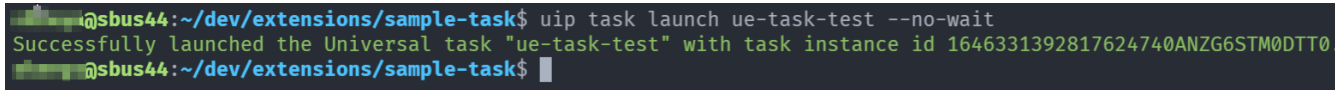
Make sure to save the task as shown previously.

Now, we will run the task using the "UIP: Task Launch **No Wait**" option shown below:



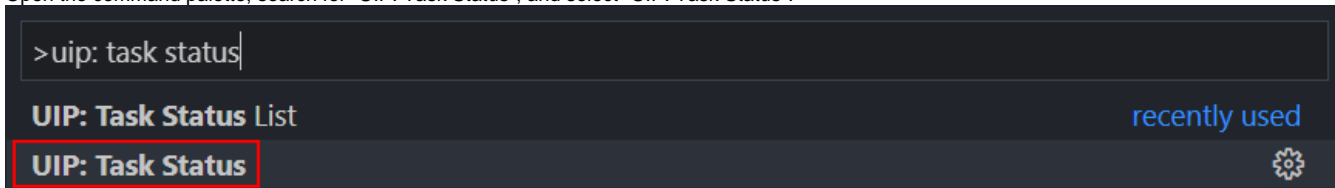
The "No Wait" signifies the task will be launched, but the CLI will not wait for the task to finish. Consequently, the final status and output of the task will NOT be retrieved.

Go ahead and launch the task using the "No Wait" option. You should see something similar as shown below:



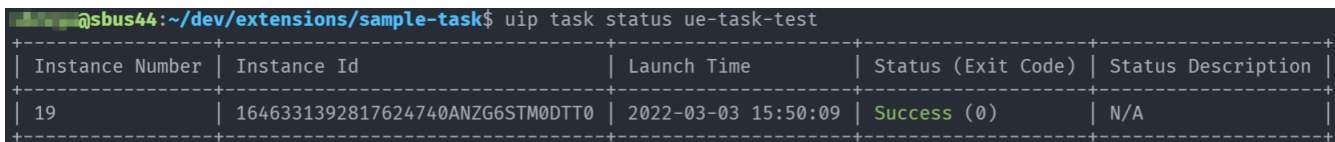
We can retrieve the task status and output manually from the Controller or use the VS Code Extension as shown below.

Open the command palette, search for "UIP: Task Status", and select "UIP: Task Status":

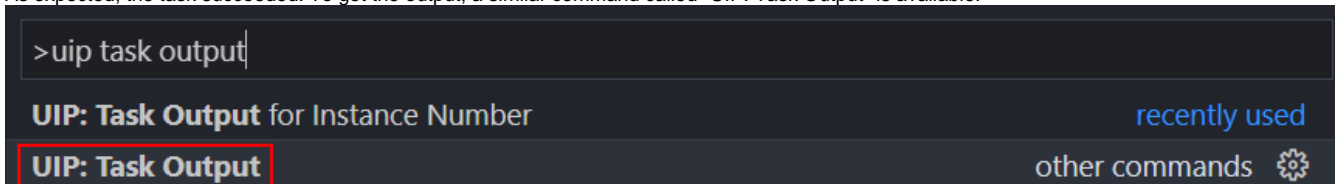


The "UIP: Task Status" command lists the status of the most recent task instance whereas the "UIP: Task Status List" gets the statuses of the 20 (this number can be configured using "UIP: Set Task Instance Count") most recent task instances.

Upon running the command, you should see something similar to the output shown below (the "Instance Number", "Instance Id", and "Launch Time" will most likely be different):

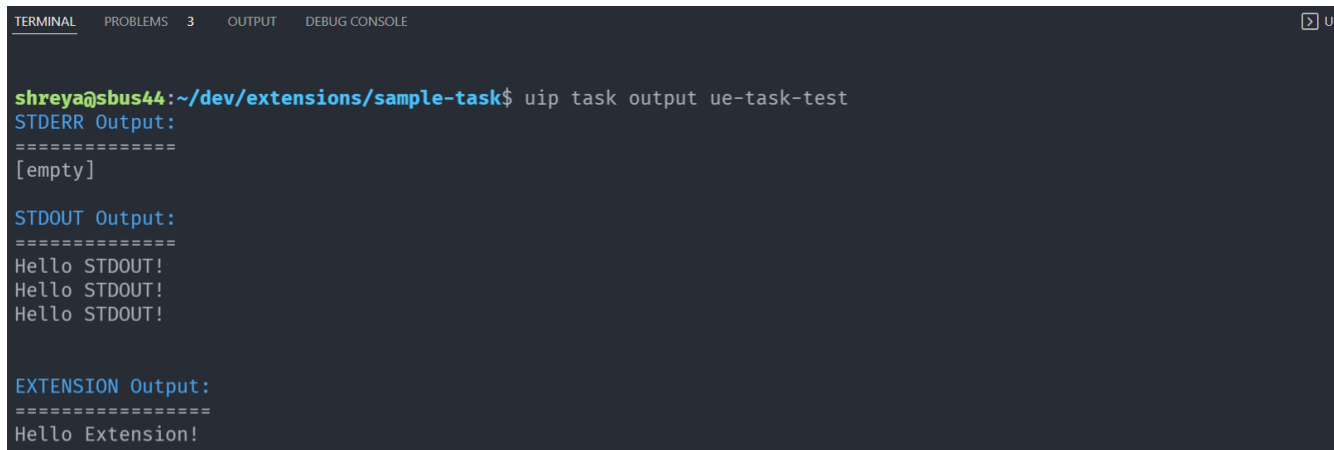


As expected, the task succeeded. To get the output, a similar command called "UIP: Task Output" is available:



The "UIP: Task Output" command retrieves the output of the latest task instance whereas the "UIP: Task Output for Instance Number" gets the output of a task instance with a specific instance number.

Since we want to get the output of the recent task instance, "UIP: Task Output" is sufficient. Run the command, and the output should be similar to one shown below:



```

TERMINAL  PROBLEMS  3  OUTPUT  DEBUG CONSOLE  UIP
shreya@sbus44: ~/dev/extensions/sample-task$ uip task output ue-task-test
STDERR Output:
=====
[empty]

STDOUT Output:
=====
Hello STDOUT!
Hello STDOUT!
Hello STDOUT!

EXTENSION Output:
=====
Hello Extension!

```

Notice that this time, the STDERR output is empty and STDOUT output is not. This is expected since we changed the value of the "Action" field in the task form.

## Step 5 - Reset the Extension Changes

The For-Loop changes that prints out the 'Hello' message multiple times were added to demonstrate the process of modifying and deploying the Extension. The next few tutorials assume that the For-Loop changes aren't in **extension.py**, so either undo the For-Loop changes or copy the code shown below to **extension.py** before moving on to the next guide:

### extension.py

```

from __future__ import (print_function)
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import logger

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """

    def __init__(self):
        """Initializes an instance of the 'Extension' class
        """
        # Call the base class initializer
        super(Extension, self).__init__()

    def extension_start(self, fields):
        """Required method that serves as the starting point for work performed
        for a task instance.

        Parameters

```

```
-----  
fields : dict  
    populated with field values from the associated task instance  
    launched in the Controller  
  
Returns  
-----  
ExtensionResult  
    once the work is done, an instance of ExtensionResult must be  
    returned. See the documentation for a full list of parameters that  
    can be passed to the ExtensionResult class constructor  
"""  
  
# Get the value of the 'action' field  
action = fields.get('action', [""])[0]  
  
if action.lower() == 'print':  
    # Print to standard output...  
    print("Hello STDOUT!")  
else:  
    # Log to standard error...  
    logger.info('Hello STDERR!')  
  
# Return the result with a payload containing a Hello message...  
return ExtensionResult(  
    unv_output='Hello Extension!'  
)
```

[< Previous](#) [Next >](#)

# Dynamic Choice Field

- [Introduction](#)
- [Step 1 - Add Dynamic Choice fields to the "UE Task" Universal Template](#)
- [Step 2 - Add backing implementation for Dynamic Choice fields to extension.py](#)
- [Step 3 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension](#)
  - [Step 3 Supplemental - Build and Upload the Extension Zip Archive Using the CLI](#)
- [Step 4 - Experience the Dynamic Choice Fields in Action](#)
- [Step 5 - Update the Local template.json](#)
  - [UIP VS Code Extension](#)
    - [Step Supplemental - CLI](#)

## Introduction

On this page, we will enhance the ue-task Extension developed in the previous chapter by adding two Dynamic Choice Fields to the "UE Task" template. A corresponding change will be made in extension.py to implement the functionality that will populate the Dynamic Choice Fields.

These choice fields will be available to "UE Task" task types at definition time and allow the user to select values that are populated by the target agent system prior to task submission.

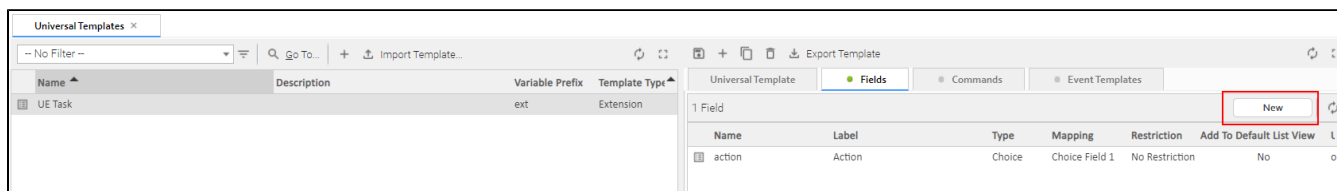
This page will cover the following:

1. Add Dynamic Choice fields to the "UE Task" Universal Template.
2. Add backing implementation for Dynamic Choice fields to the extension.py file in the sample-task Extension project.
3. Rebuild and upload the modified Extension.
4. Populate the Dynamic Choice fields on a task definition form.

## Step 1 - Add Dynamic Choice fields to the "UE Task" Universal Template

Go back to the "UE Task" Universal Template form.

Click on the "New" option in the "Fields" tab of the template



In the new field popup windows, create a new Dynamic Choice field with a name of **"primary\_choice\_field"**:

**Field Details**

Field | Choices

**General**

Name \*  Label \*

Hint

Add To Default List View

**Field Details**

Type  Mapping

Default Value

Allow Empty Choice  Allow Multiple Choices

Dynamic Choice  Dependent Fields

Choice Sort Option

**Validation**

Show If Field

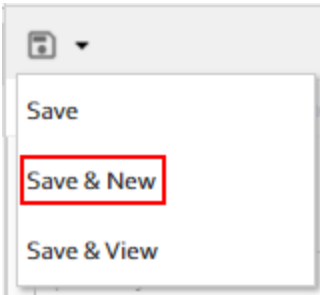
**Form Layout**

Start Row  Column Span

End Row

Note that "Start Row" is checked in the Form Layout section. This is not required. It is selected only to achieve a better layout of form fields.

Click the dropdown next to the "Save" icon and select "Save & New":



Next, create a second Dynamic Choice field with a name of "secondary\_choice\_field".

**Field Details**

Field  Choices

**General**

Name \* secondary\_choice\_field Label \* Secondary Choice

Hint

Add To Default List View

**Field Details**

Type Choice Mapping Choice Field 3

Default Value

Allow Empty Choice  Allow Multiple Choices

Dynamic Choice  Dependent Fields

Choice Sort Option Sequence

**Validation**

Show If Field -- None --

**Form Layout**

Start Row  Column Span 1

End Row

Note that "End Row" is checked in the Form Layout section. This is not required. It is selected only to achieve a better layout of form fields.

Click the "Save" icon to save the field.

## Step 2 - Add backing implementation for Dynamic Choice fields to extension.py

Open file `~/dev/extensions/sample-task/src/extension.py` in your editor of choice.

Add the implementation of the **primary\_choice\_field** and **secondary\_choice\_field** Dynamic Choice Commands:

### primary\_choice\_field Dynamic Choice Command

```

from __future__ import (print_function)
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import logger
from universal_extension.deco import dynamic_choice_command

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """

    def __init__(self):
        """Initializes an instance of the 'Extension' class
        """ # Call the base class initializer
        super(Extension, self).__init__()

    @dynamic_choice_command("primary_choice_field")
    def primary_choice_command(self, fields):
        """Dynamic choice command implementation for
        primary_choice_field.

        Parameters
        -----
        fields : dict
            populated with the values of the dependent fields
        """
        return ExtensionResult(
            rc=0,
            message="Values for choice field: 'primary_choice_field'",
            values=["Start", "Pause", "Stop", "Build", "Destroy"]
        )

    @dynamic_choice_command("secondary_choice_field")
    def secondary_choice_command(self, fields):
        """Dynamic choice command implementation for
        secondary_choice_field.

        Parameters
        -----

```

```

fields : dict
    populated with the values of the dependent fields
"""
return ExtensionResult(
    rc=0,
    message="Values for choice field: 'secondary_choice_field'",
    values=["System", "Command", "Application", "Transfer", "Evidence"]
)

def extension_start(self, fields):
    """Required method that serves as the starting point for work performed
    for a task instance.

    Parameters
    -----
    fields : dict
        populated with field values from the associated task instance
        launched in the Controller

    Returns
    -----
    ExtensionResult
        once the work is done, an instance of ExtensionResult must be
        returned. See the documentation for a full list of parameters that
        can be passed to the ExtensionResult class constructor
    """

    # Get the value of the 'action' field
    action = fields.get('action', [""])[0]


    if action.lower() == 'print':
        # Print to standard output...
        print("Hello STDOUT!")
    else:
        # Log to standard error...
        logger.info('Hello STDERR!')

    # Return the result with a payload containing a Hello message...
    return ExtensionResult(
        unv_output='Hello Extension!'
    )

```

<b>Line 5</b>	We added an import for the <code>dynamic_choice_command</code> decorator which comes from the <code>UniversalExtension</code> base package.
<b>Lines 18 to 32</b>	The complete implementation of the Dynamic Choice command that supports the Dynamic Choice field <b>primary_choice_field</b> defined in the Universal Template.
<b>Lines 34 to 48</b>	The complete implementation of the Dynamic Choice command that supports the Dynamic Choice field <b>secondary_choice_field</b> defined in the Universal Template.

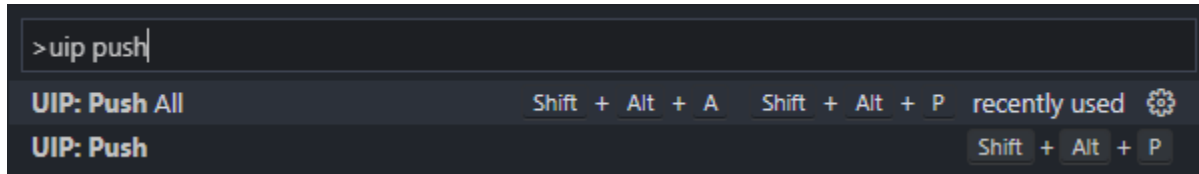
**Note**

 The value supplied to the `dynamic_choice_command` decorator must match the field name of the associated Dynamic Choice field in the Controller's Universal Template (for example, `@dynamic_choice_command("primary_choice_field")`).

## Step 3 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension

Save all changes to extension.py.

From the VS Code command pallet, execute the **UIP: Push** command as shown below:



Recall that the **UIP: Push** command builds and uploads the Extension zip archive

## Step 3 Supplemental - Build and Upload the Extension Zip Archive Using the CLI

Save all changes to extension.py.

From the command line, cd to the sample-task directory and execute the **push** command as shown below:

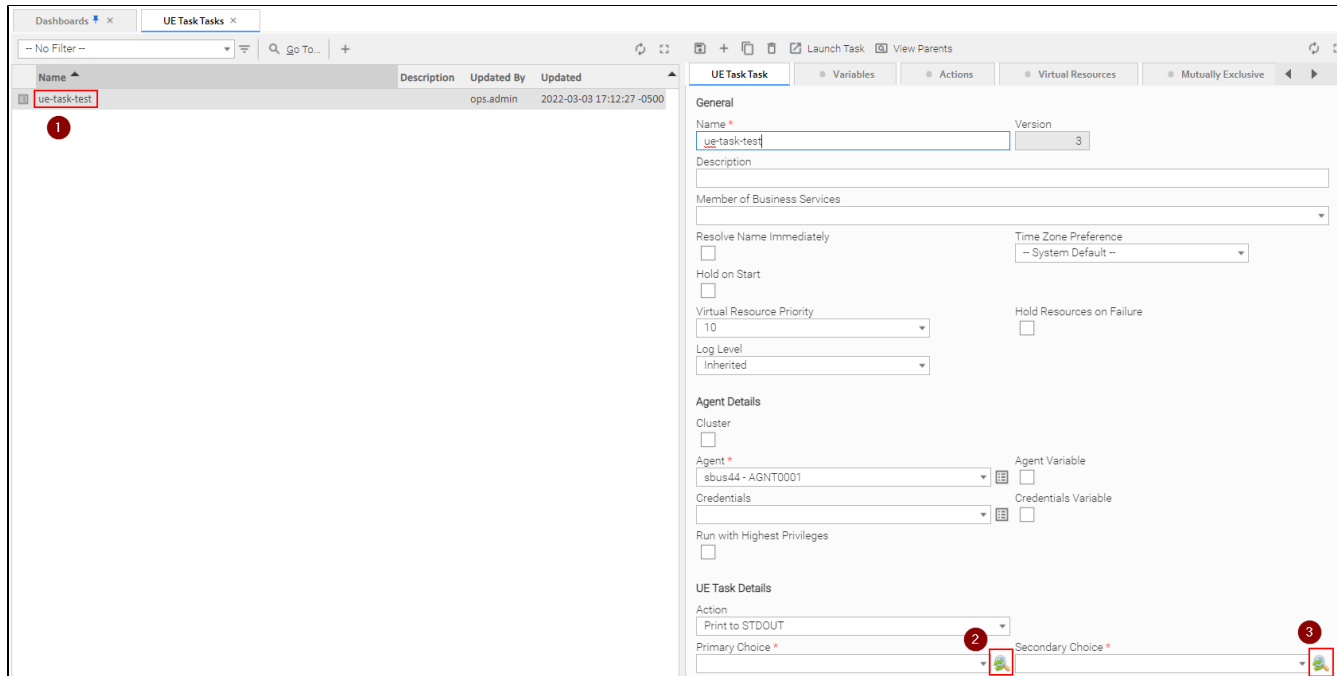
```
$ uip push
```

Recall that the **push** command builds and uploads the Extension zip archive

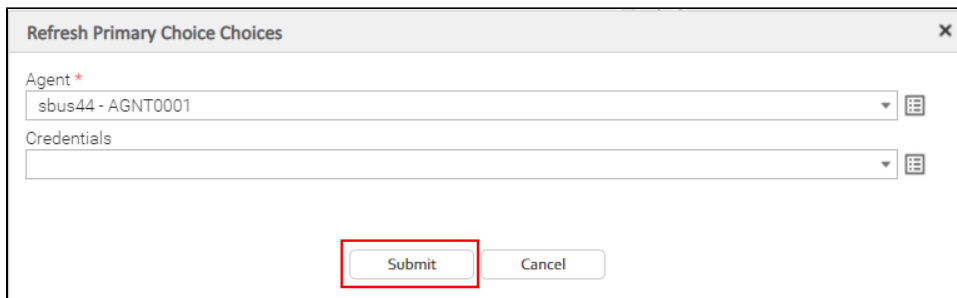
## Step 4 - Experience the Dynamic Choice Fields in Action

In the Controller, if the **UE Task** Tasks tab is open, close it now (required to pick up the addition of the Dynamic Choice fields).

Open **UE Task** Tasks tab:



1. Select the **ue-task-test** (or whatever name you gave it) task
2. Select an **active** agent of **version 7.0.0.0 or higher** for the task to run on.
3. Click on the "Primary Choice Field" search icon to initiate a Dynamic Command call. Clicking on the "Primary Choice" search icon will open the following dialog. Click the "Submit" button.



Upon clicking the Submit button, the Primary Choice Field search icon will turn grey for a moment while the Dynamic Command request is sent down to the Agent and results are returned to populate the drop-down.

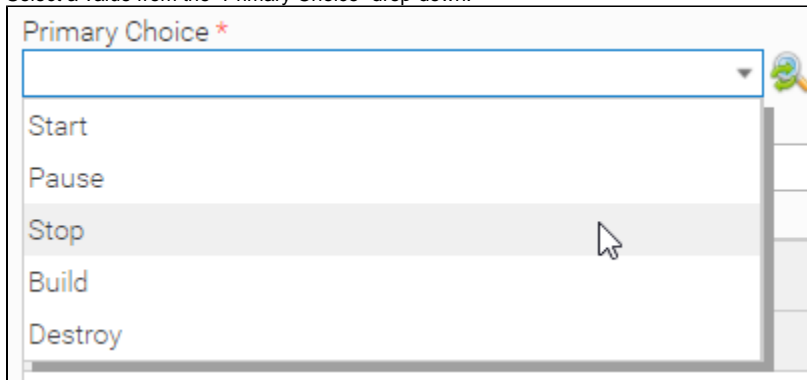
When the operation is complete, the icon will turn green again and the Primary Choice drop-down will be populated with the values supplied by `@dynamic_choice_command ("primary_choice_field")` code in the extension on the Agent system.

**You have just executed a Dynamic Choice Command from the "Primary Choice" Dynamic Choice field!**

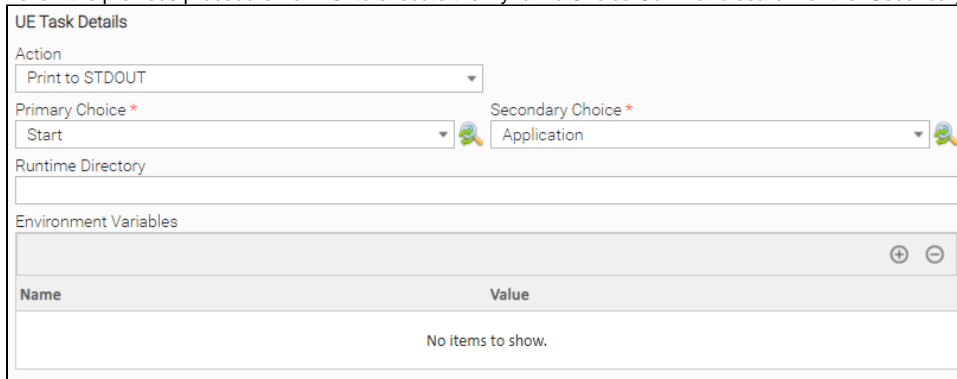
**Note**

By clicking the search icon for the first time, the Controller automatically re-deployed the modified Extension archive to the target agent.

Select a value from the "Primary Choice" drop-down.



4. Follow the previous procedure from "3" to execute the Dynamic Choice Command search for the "Secondary Choice field" and select a value from the populated dropdown.



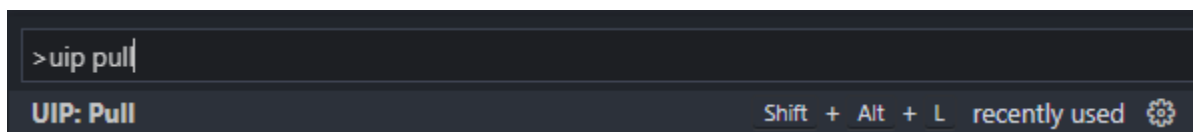
Save the task

## Step 5 - Update the Local template.json

In step 1, we modified the Universal Template by adding new Dynamic Choice Fields. Recall that inside `~/dev/extensions/sample-task/src/templates`, there is a **template.json** file. This file should correspond to the Universal Template in the Controller.

Right now, they are not both the same. The Controller's version of **template.json** has the Dynamic Choice Fields changes. To grab those changes, use the **pull** command as shown below:

### UIP VS Code Extension



```
████████@sbus44:~/dev/extensions/sample-task$ uip pull
The following files were updated:
- template.json
```

### Step Supplemental - CLI

```
████████@sbus44:~/dev/extensions/sample-task$ uip pull
The following files were updated:
- template.json
```

Now, both the local and Controller's version of the Universal Template are the same.

It is a good practice to keep both of them in-sync. Otherwise, one or the other could be overwritten. For example, if the local **template.json** does not contain any new changes that the Controller's version of Universal Template does, then issuing the **UIP: Push All** or **push -a** command will overwrite the Controller's version of the Universal Template.

[< Previous](#) [Next >](#)

# Dynamic Update and Output Only Fields

- [Introduction](#)
- [Step 1 - Output Only fields to the "UE Task" Universal Template](#)
- [Step 2 - Add support for updating Output Only fields in extension.py](#)
- [Step 3 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension](#)
  - [Step 3 Supplemental - Build and Upload the Extension Zip Archive Using the CLI](#)
- [Step 4 - Launch task and observe the output only fields](#)
- [Step 5 - Update the Local template.json](#)
  - [UIP VS Code Extension](#)
    - [Step Supplemental - CLI](#)

## Introduction

On this page, we will enhance the ue-task Extension to support Output Only fields. These are fields that are designed to be updated by an Extension instance running on an agent system. They can be used for any purpose but, a typical use case is to reflect state changes that occur in the Extension instance. That is what we will simulate here.

This page will cover the following:

1. Add Output Only fields to the "UE Task" Universal Template.
2. Modify extension.py to populate Output Only fields.
3. Execute ue-task-test task and review the Output Only field updates

## Step 1 - Output Only fields to the "UE Task" Universal Template

Go back to the "UE Task" Universal Template form.

On the Fields tab of the Universal Template, create three Output Only fields of type Text:

1. task\_action
2. step\_1
3. step\_2

**Field Details**

Field | Choices

**General**

Name \*  Label \*

Hint

Add To Default List View

**Field Details**

Type  Mapping

Text Type

Default Value

Restriction  No Restriction  Output Only

Preserve Output On Re-run

**Validation**

Show If Field

**Form Layout**

Start Row

End Row

Column Span

Click the "Save & New" option in the dropdown next to the "Save" icon and create the step\_1 and step\_2 fields. (For a better layout, don't check the "Start Row" and "End Row" boxes for step\_1 and step\_2 fields).

At this point, the Template's Field tab should be populated as follows:

Name	Label	Type	Mapping	Restriction	Add To Default List View	Updated By	Updated
<input type="checkbox"/> action	Action	Choice	Choice Field 1	No Restriction	No	ops.admin	2022-03-03 13:18:39 -0500
<input type="checkbox"/> primary_choice_field	Primary Choice	Choice	Choice Field 2	No Restriction	No	ops.admin	2022-03-03 16:47:40 -0500
<input type="checkbox"/> secondary_choice_field	Secondary Choice	Choice	Choice Field 3	No Restriction	No	ops.admin	2022-03-03 17:11:06 -0500
<input checked="" type="checkbox"/> task_action	Task Action	Text	Text Field 1	Output Only	No	ops.admin	2022-03-03 22:13:22 -0500
<input type="checkbox"/> step_1	Step 1	Text	Text Field 2	Output Only	No	ops.admin	2022-03-03 22:15:24 -0500
<input type="checkbox"/> step_2	Step 2	Text	Text Field 3	Output Only	No	ops.admin	2022-03-03 22:15:47 -0500

The **task\_action** field will be used to reflect the task action taken by the **ue-task-test** task.

Fields `step_1` and `step_2` will be used to reflect the progression of work in the task instance as it works its way through them.

## Step 2 - Add support for updating Output Only fields in extension.py

Open file `~/dev/extensions/sample-task/src/extension.py` in your editor of choice.

Enhance the `extension_start` method with the following code:

### primary\_choice\_field Dynamic Choice Command

```

from __future__ import (print_function)
import time
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import logger
from universal_extension.deco import dynamic_choice_command
from universal_extension import ui

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """

    def __init__(self):
        """Initializes an instance of the 'Extension' class
        """
        # Call the base class initializer
        super(Extension, self).__init__()

    @dynamic_choice_command("primary_choice_field")
    def primary_choice_command(self, fields):
        """Dynamic choice command implementation for
        primary_choice_field.

        Parameters
        -----
        fields : dict
            populated with the values of the dependent fields
        """
        return ExtensionResult(
            rc=0,
            message="Values for choice field: 'primary_choice_field'",
            values=["Start", "Pause", "Stop", "Build", "Destroy"]
        )

    @dynamic_choice_command("secondary_choice_field")
    def secondary_choice_command(self, fields):
        """Dynamic choice command implementation for
        secondary_choice_field.

```

```

Parameters
-----
fields : dict
    populated with the values of the dependent fields
"""
return ExtensionResult(
    rc=0,
    message="Values for choice field: 'secondary_choice_field'",
    values=["System", "Command", "Application", "Transfer", "Evidence"]
)

def extension_start(self, fields):
    """Required method that serves as the starting point for work performed
    for a task instance.

    Parameters
    -----
    fields : dict
        populated with field values from the associated task instance
        launched in the Controller

    Returns
    -----
    ExtensionResult
        once the work is done, an instance of ExtensionResult must be
        returned. See the documentation for a full list of parameters that
        can be passed to the ExtensionResult class constructor
    """

    sleep_value = 5

    # Update the 'task_action' Output Only field on the task instance form
    out_fields = {}
    task_action = "{0} {1}".format(
        fields['primary_choice_field'][0],
        fields['secondary_choice_field'][0])
    out_fields["task_action"] = task_action
    ui.update_output_fields(out_fields)

    # Sleep
    time.sleep(sleep_value)

    # Update the 'step_1' Output Only field on the task instance form
    out_fields = {}
    out_fields["step_1"] = "Step 1 - Success"
    ui.update_output_fields(out_fields)

    # Sleep
    time.sleep(sleep_value)

    # Update the 'step_2' Output Only field on the task instance form
    out_fields = {}

```

```

out_fields["step_2"] = "Step 2 - Success"
ui.update_output_fields(out_fields)

# Get the value of the 'action' field
action = fields.get('action', [""])[0]

if action.lower() == 'print':
    # Print to standard output...
    print("Hello STDOUT!")
else:
    # Log to standard error...
    logger.info('Hello STDERR!')

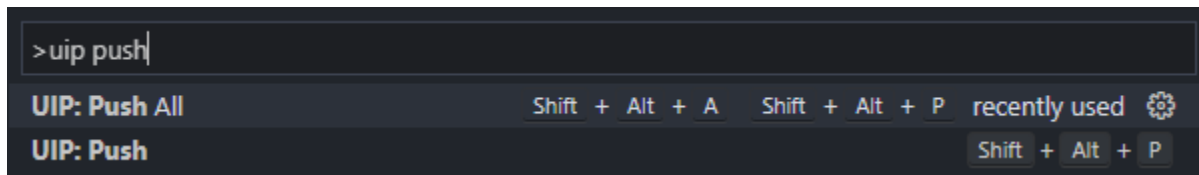
# Return the result with a payload containing a Hello message...
return ExtensionResult(
    unv_output='Hello Extension!'
)

```

<b>Line 2</b>	We added an import for the time module to gain access to the sleep function which we will use to force controlled delays between Output Only field updates.
<b>Line 7</b>	We added an import for the <b>ui</b> module to access the <b>update_output_fields()</b> method
<b>Line 70</b>	We initialize variable <b>sleep_value</b> . This variable will be used to control the sleep delays.
<b>Line 73</b>	We create a new dictionary to hold output fields to be sent back to the Controller.
<b>Lines 74 to 76</b>	We format a <b>task_action</b> string using the <b>primary_choice_field</b> and <b>secondary_choice_field</b> values passed down from the Controller.
<b>Line 77</b>	We assign the formatted <b>task_action string value</b> to the <b>out_fields dictionary</b> using key " <b>task_action</b> " to associate it with the "task_action" Output Only field in associated task instance in the Controller.
<b>Line 78</b>	This calls the <b>update_output_fields()</b> method from the <b>ui</b> module. This will send a message to the Controller and update the value of the specified fields in the associated task instance (in this case just "task_action").
<b>Lines 80 to 94</b>	This follows the same basic basic procedure to update Output Only fields <b>step_1</b> and <b>step_2</b> . A sleep delay is introduced between each output field update.

### Step 3 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension

From the VS Code command palette, execute the **UIP: Push** command as shown below:



Recall that the **UIP: Push** command builds and uploads the Extension zip archive

### Step 3 Supplemental - Build and Upload the Extension Zip Archive Using the CLI

Save all changes to extension.py.

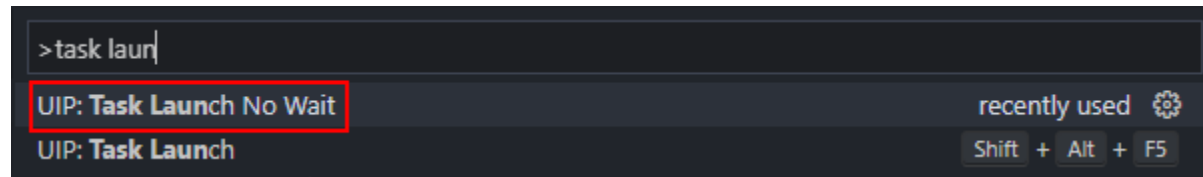
From the command line, cd to the ./sample-1 directory and execute the **push** command as shown below:

```
$ uip push
```

Recall that the **push** command builds and uploads the Extension zip archive

### Step 4 - Launch task and observe the output only fields

From the VS Code command palette, execute the **UIP: Task Launch No Wait** command as shown below:



In the Controller, open the "UE Task Tasks" tab and select the **ue-task-test** task (or whatever name you gave it).

Switch to the Instances tab for the task and you should see the recently launched tasks with a status of Running or Success (depending on timing). If the task shows a status of Running, click refresh every few seconds until it goes to Success.

Instance Name	Status	Invoked By	Start Time	End Time	Updated
ue-task-test	Success	Manually Launched	2022-03-03 22:28:03 -0500	2022-03-03 22:28:13 -0500	2022-03-03 22:28:13 -0500

Once the task goes to Success, double click on it to open the task instance form.

Scroll down to the "sample-1 Details" section and review the Output Only fields:

**UE Task Details**

Action

Primary Choice \*  Secondary Choice \*

Task Action

Step 1  Step 2

Runtime Directory

Environment Variables

Name	Value
No items to show.	

The "Task Action" field along with the "Step 1" and "Step 2" fields have been updated by the Extension instance running on the agent.

This was accomplished by the calls to `ui.update_output_fields(out_fields)` in the `extension_start` method in `extension.py`.

## Step 5 - Update the Local template.json

In step 1, we modified the Universal Template by adding new Output Only Fields. Recall that inside `~/dev/extensions/sample-task/src/templates`, there is a **template.json** file. This file should correspond to the Universal Template in the Controller.

Right now, they are not both the same. The Controller's version of **template.json** has the Dynamic Choice Fields changes. To grab those changes, use the **pull** command as shown below:

### UIP VS Code Extension

```

> uip pull
UIP: Pull Shift + Alt + L recently used
@sbus44:~/dev/extensions/sample-task$ uip pull
The following files were updated:
- template.json
    
```

### Step Supplemental - CLI

```
██████████@sbus44:~/dev/extensions/sample-task$ uip pull  
The following files were updated:  
- template.json
```

Now, both the local and Controller's version of the Universal Template are the same.

[< Previous](#) [Next >](#)

# Dynamic Command

- [Introduction](#)
- [Step 1 - Add a Dynamic Command to the "UE Task" Universal Template](#)
- [Step 2 - Add Backing Implementation for Dynamic Command to extension.py](#)
- [Step 3 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension](#)
  - [Step 3 Supplemental - Build and Upload the Extension Zip Archive Using the CLI](#)
- [Step 4 - Execute Dynamic Command](#)
- [Step 5 - Update the Local template.json](#)
  - [UIP VS Code Extension](#)
    - [Step Supplemental - CLI](#)

## Introduction

Dynamic Commands allow you to add supporting functionality to a task instance. In this chapter, we will enhance the ue-task Extension developed in the previous chapters by adding a Dynamic Command. The command we create will reset the task instance - to prepare it for a rerun scenario.

On this page, we will cover the following:

1. Add Dynamic Command to the "UE Task" Universal Template.
2. Add a backing implementation for Dynamic Command to the extension.py file in the ue-task Extension project.
3. Rebuild and upload the modified Extension.
4. Execute the Dynamic Command

## Step 1 - Add a Dynamic Command to the "UE Task" Universal Template

Go back to the "UE Task" Universal Template form.

On the Commands tab, click the "New" button and create the following "Reset Environment" command.

Note that the “Supported Status(es)” is set to: In Doubt, Cancelled, Failed, Finished, and Success. These are the task instance statuses where the command will be available for execution. For all other statuses the command will be greyed out.

Save the command.

## Step 2 - Add Backing Implementation for Dynamic Command to extension.py

Open file `~/dev/extensions/sample-task/src/extension.py` in your editor of choice.

Add the implementation of the **reset\_environment** Dynamic Command:

### reset\_environment Dynamic Command

```
from __future__ import (print_function)
import time
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import logger
from universal_extension import ui
from universal_extension.deco import dynamic_choice_command
from universal_extension.deco import dynamic_command

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """

    def __init__(self):
```

```

    """Initializes an instance of the 'Extension' class
    """
    # Call the base class initializer
    super(Extension, self).__init__()

@dynamic_choice_command("primary_choice_field")
def primary_choice_command(self, fields):
    """Dynamic choice command implementation for
    primary_choice_field.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """
    return ExtensionResult(
        rc=0,
        message="Values for choice field: 'primary_choice_field'",
        values=["Start", "Pause", "Stop", "Build", "Destroy"]
    )

@dynamic_choice_command("secondary_choice_field")
def secondary_choice_command(self, fields):
    """Dynamic choice command implementation for
    secondary_choice_field.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """
    return ExtensionResult(
        rc=0,
        message="Values for choice field: 'secondary_choice_field'",
        values=["System", "Command", "Application", "Transfer", "Evidence"]
    )

@dynamic_command("reset_environment")
def reset_environment(self, fields):
    """Dynamic command implementation for reset_environment command.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """

    # Reset the state of the Output Only 'step' fields.
    out_fields = {}
    out_fields["step_1"] = "Initial"
    out_fields["step_2"] = "Initial"
    ui.update_output_fields(out_fields)

```

```

return ExtensionResult(
    message="Message: Hello from dynamic command 'reset_environment'!",
    output=True,
    output_data='The environment has been reset.',
    output_name='DYNAMIC_OUTPUT')

def extension_start(self, fields):
    """Required method that serves as the starting point for work performed
    for a task instance.

    Parameters
    -----
    fields : dict
        populated with field values from the associated task instance
        launched in the Controller

    Returns
    -----
    ExtensionResult
        once the work is done, an instance of ExtensionResult must be
        returned. See the documentation for a full list of parameters that
        can be passed to the ExtensionResult class constructor
    """

    sleep_value = 5

    # Update the 'task_action' Output Only field on the task instance form
    out_fields = {}
    task_action = "{0} {1}".format(
        fields['primary_choice_field'][0],
        fields['secondary_choice_field'][0])
    out_fields["task_action"] = task_action
    ui.update_output_fields(out_fields)

    # Sleep
    time.sleep(sleep_value)

    # Update the 'step_1' Output Only field on the task instance form
    out_fields = {}
    out_fields["step_1"] = "Step 1 - Success"
    ui.update_output_fields(out_fields)

    # Sleep
    time.sleep(sleep_value)

    # Update the 'step_2' Output Only field on the task instance form
    out_fields = {}
    out_fields["step_2"] = "Step 2 - Success"
    ui.update_output_fields(out_fields)

    # Get the value of the 'action' field

```

```

    action = fields.get('action', [""])[0]

    if action.lower() == 'print':
        # Print to standard output...
        print("Hello STDOUT!")
    else:
        # Log to standard error...
        logger.info('Hello STDERR!')

    # Return the result with a payload containing a Hello message...
    return ExtensionResult(
        unv_output='Hello Extension!'
    )

```

<b>Line 8</b>	We added an import for the <code>dynamic_command</code> decorator which comes from the <code>UniversalExtension</code> base package. (Also moved the <code>time</code> module import to line 2)
<b>Lines 53 to 73</b>	The complete implementation of the Dynamic Command that backs the <code>reset_environment</code> command defined in the Universal Template.

Note

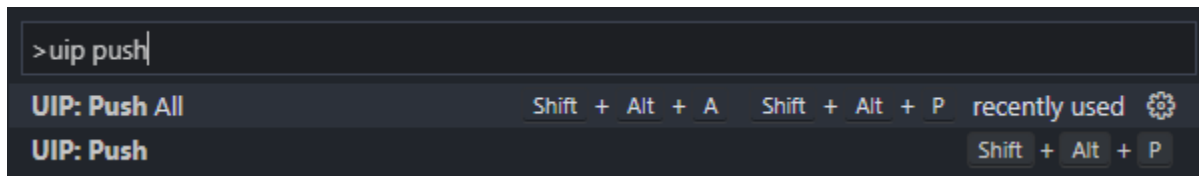


The value supplied to the `dynamic_command` decorator must match the command Name of the associated Dynamic Command defined in the Controller's Universal Template (for example, `@dynamic_command("reset_environment")`).

## Step 3 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension

Save all changes to `extension.py`.

From the VS Code command pallet, execute the **UIP: Push** command as shown below:



Recall that the **UIP: Push** command builds and uploads the Extension zip archive

## Step 3 Supplemental - Build and Upload the Extension Zip Archive Using the CLI

Save all changes to `extension.py`.

From the command line, `cd` to the `~/dev/extensions/sample-task` directory and execute the **push** command as shown below:

```
$ uip push
```

Recall that the **push** command builds and uploads the Extension zip archive

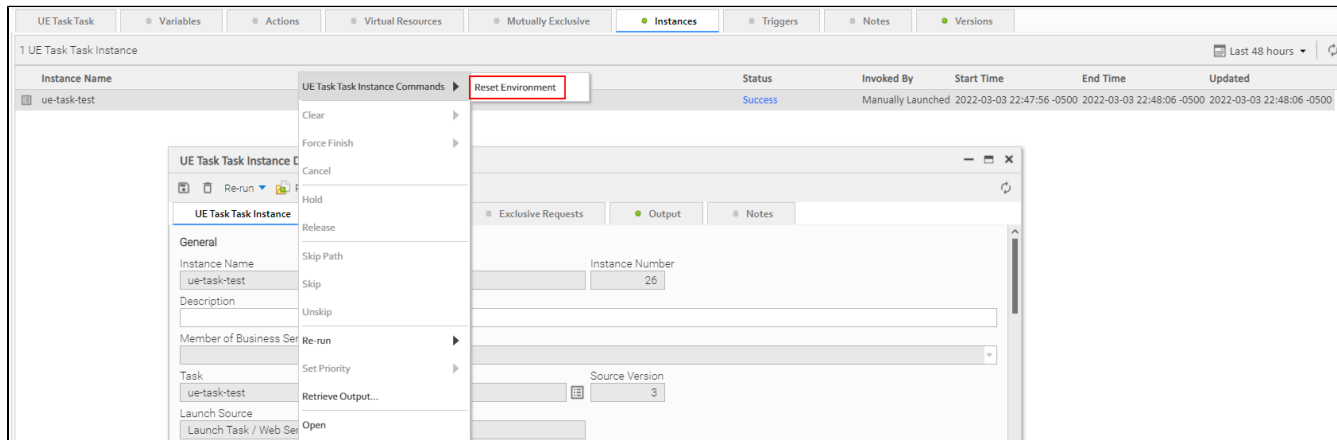
## Step 4 - Execute Dynamic Command

In the Controller, open the "UE Task tasks" tab. **If the tab is already open, it must be closed and reopened before the the new Dynamic Command will be available on the task instances.**

Switch to the Instances tab. If there are no task instances in the list, launch a new one now using the VS Code Command (or manually from the Controller) and let it run to completion.

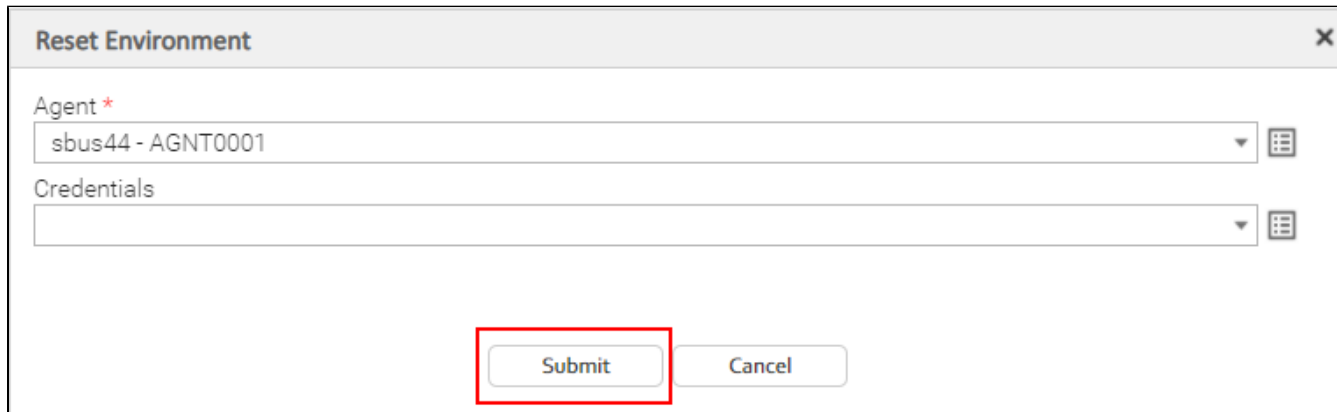
Dynamic Commands are available from the context menu generated by right-clicking on the task instance form and from the context menu generated by right-clicking on the task instance in a list of task instances (for example, in the Activity Monitor and/or the Instances tab of a task definition form).

For this example, double-click on a completed task to open the task instance form. next, right-click on the form to open a context menu. At the top of the context menu should be the "UE Task Task Instance Commands".



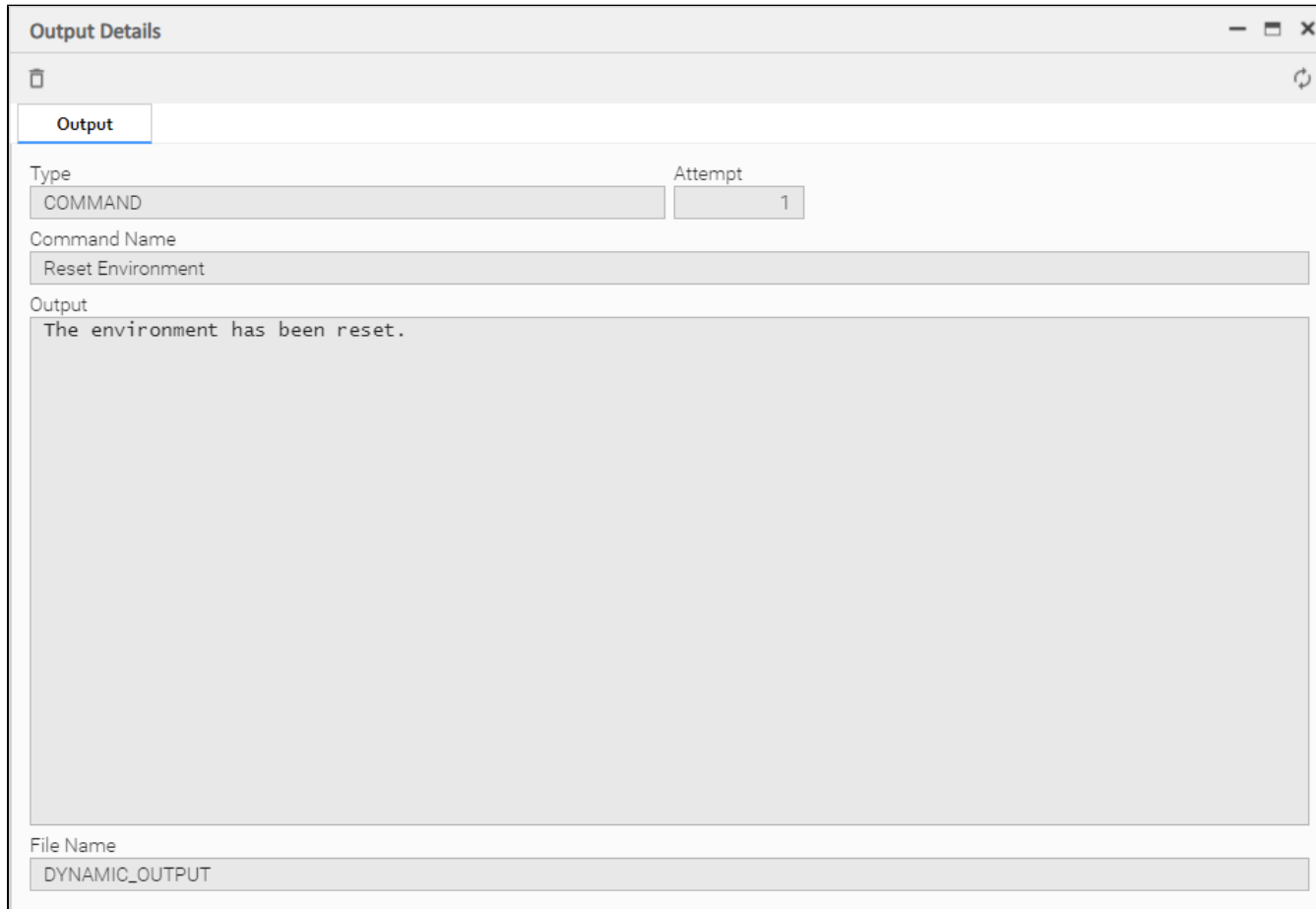
Select the **Reset Environment** command.

This displays an additional dialog, which can be used to pull in dependent fields from the task form if needed. This is optional functionality that is defined in the Universal template and made use of by the backing implementation in the extension archive.



Submit the Dynamic Command.

The immediate result of Dynamic Command is new window showing the output results of the command.



In this case, the resulting output is "The environment has been reset".

Additionally, the **Reset Environment** command was designed to "reset" the "Step" Output Only fields on the task instance form.

To see this, close the "Output Details" window and scroll the open task instance form to view the "UE Task" Details section.

**UE Task Details**

Action  
 Print to STDOUT

Primary Choice \*  
 Start

Secondary Choice \*  
 Application

Task Action  
 Start Application

Step 1  
 Step 1 - Success

Step 2  
 Step 2 - Success

To reveal the change, refresh the task instance form.

**UE Task Task Instance Details: ue-task-test**

Re-run Retrieve Output...

UE Task Task Instance Virtual Resources Exclusive Requests Output Notes

Run with Highest Privileges

**UE Task Details**

Action  
 Print to STDOUT

Primary Choice \*  
 Start

Secondary Choice \*  
 Application

Task Action  
 Start Application

Step 1  
 Initial

Step 2  
 Initial

Runtime Directory

Environment Variables

Name	Value
No items to show.	

The Step 1 and Step 2 Output Only fields have been set to **Initial**. This is a contrived example, but it demonstrates how a Dynamic Command could be used as a helper command for a task instance by performing some action on the target agent system (or beyond) and then updating the task instance in a meaningful way that may be required for a rerun scenario.

In this case, information flowed from the Dynamic Command execution back to the task instance fields. However, task specific information could also have been passed down to the Dynamic command.


## Step 5 - Update the Local template.json

In step 1, we modified the Universal Template by adding a new Dynamic command. Recall that inside `~/dev/extensions/sample-task/src/templates`, there is a **template.json** file. This file should correspond to the Universal Template in the Controller.

Right now, they are not both the same. The Controller's version of **template.json** has the Dynamic Choice Fields changes. To grab those changes, use the **pull** command as shown below:

## UIP VS Code Extension

```
>uip pul|
```

UIP: Pull Shift + Alt + L recently used 

```
@sbus44:~/dev/extensions/sample-task$ uip pull
The following files were updated:
- template.json
```

## Step Supplemental - CLI

```
@sbus44:~/dev/extensions/sample-task$ uip pull
The following files were updated:
- template.json
```

Now, both the local and Controller's version of the Universal Template are the same.

[< Previous](#) [Next >](#)

# In-Process Dynamic Commands

- [Introduction](#)
- [Step 1 - Add Asynchronous In-Process Dynamic Command to the "UE Task" Universal Template](#)
- [Step 2 - Add Synchronous In-Process Dynamic Command to the "UE Task" Universal Template](#)
- [Step 3 - Add a new text field to the "UE Task" Universal Template](#)
- [Step 4 - Add a backing implementation for both Asynchronous and Synchronous In-Process Dynamic Commands to the extension.py file](#)
- [Step 5 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension](#)
  - [Step 5 Supplemental - Build and Upload the Extension Zip Archive Using the CLI](#)
- [Step 6 - Execute the Asynchronous Dynamic Command](#)
- [Step 7 - Execute the Synchronous Dynamic Command](#)
- [Step 8 - Update the Local template.json](#)
  - [UIP VS Code Extension](#)
    - [Step Supplemental - CLI](#)

## Introduction

The Dynamic Command created in the previous chapter always runs in a separate process from the Extension instance. In this chapter, In-process Dynamic Commands will be introduced, which run within the process of an associated Extension instance. As a result of running "in-process", the dynamic commands can impact the execution of the Extension instance.

On this page, we will cover the following:

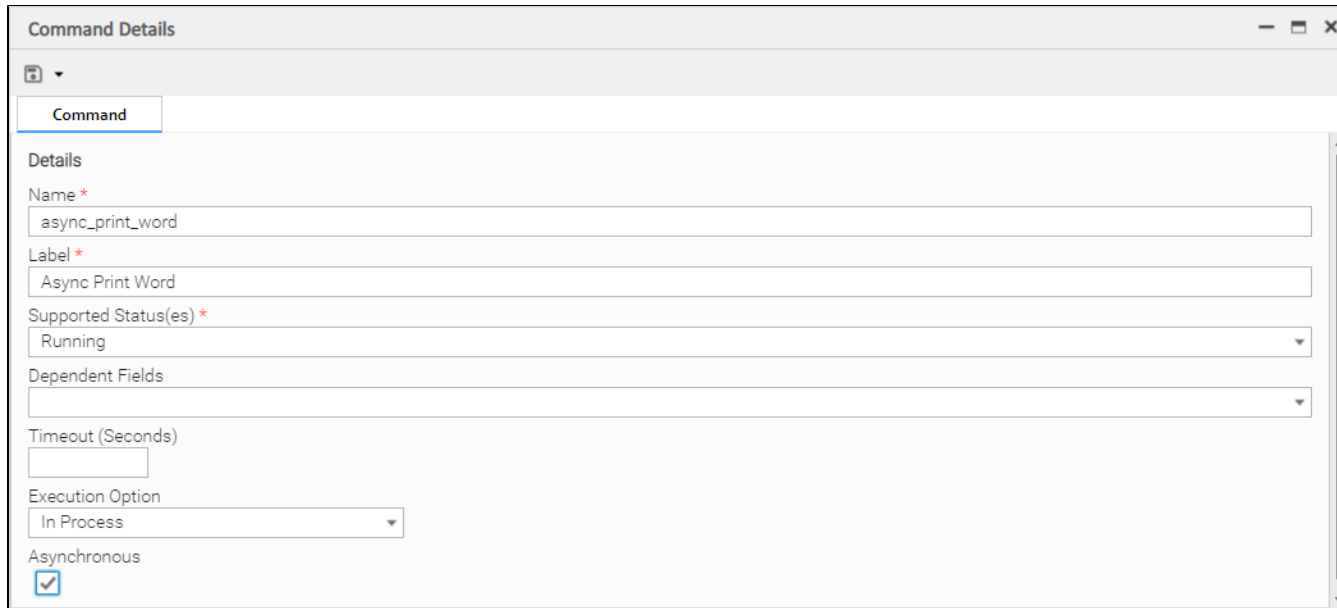
1. Add Asynchronous In-Process Dynamic Command to the "UE Task" Universal Template.
2. Add Synchronous In-Process Dynamic Command to the "UE Task" Universal Template.
3. Add a new text field to the "UE Task" Universal Template.
4. Add a backing implementation for both Asynchronous and Synchronous In-Process Dynamic Commands to the extension.py file in the ue-task Extension project.
5. Build and Upload the modified Extension.
6. Execute the Asynchronous Dynamic Command.
7. Execute the Synchronous Dynamic Command.

The steps below assume Controller version is at least 7.1.0.0

## Step 1 - Add Asynchronous In-Process Dynamic Command to the "UE Task" Universal Template

Go back to the "UE Task" Universal Template form.

On the Commands tab, click the "New" button and create the following "Async Print Word" command.



The screenshot shows a window titled "Command Details" with a "Command" tab selected. The "Details" section contains the following fields:

- Name \*:
- Label \*:
- Supported Status(es) \*:
- Dependent Fields:
- Timeout (Seconds):
- Execution Option:
- Asynchronous:

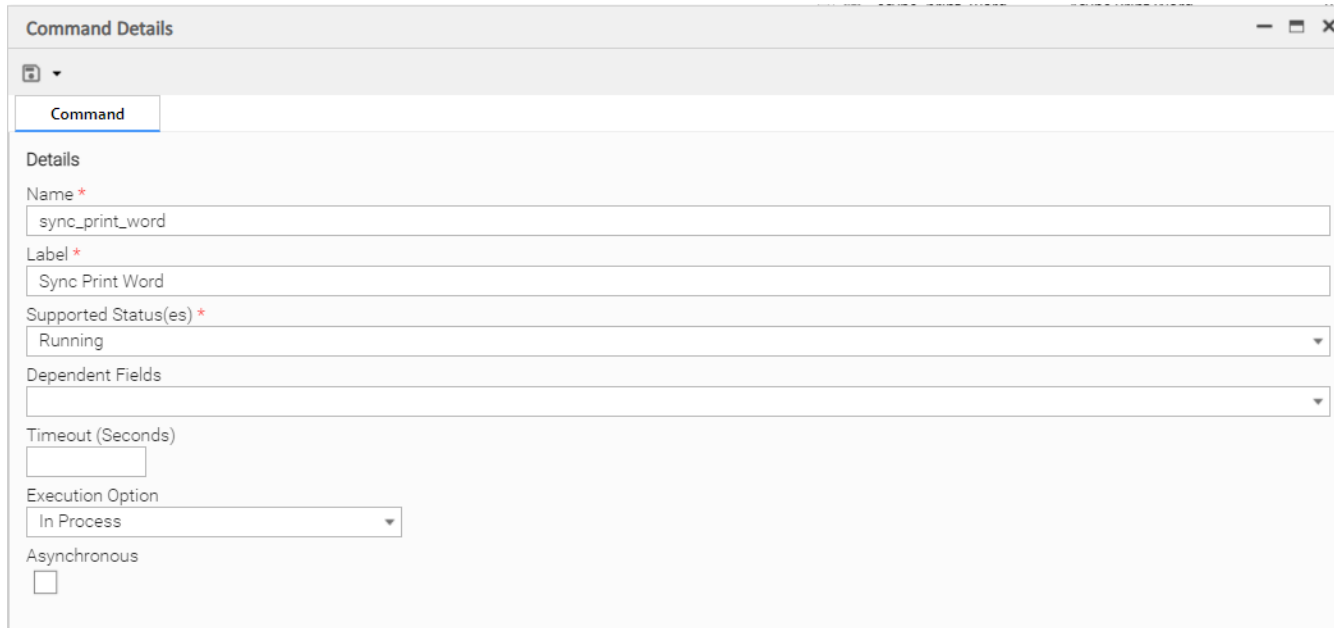
Note that the "Supported Status(es)" is set to **Running** only. This is because In-Process Dynamic commands can only run while the Extension instance is running. For all other task statuses, the command will be greyed out.

Also note that the "Execution Option" is explicitly set to **In Process**, and the **Asynchronous** option is checked.

## Step 2 - Add Synchronous In-Process Dynamic Command to the "UE Task" Universal Template

Go back to the "UE Task" Universal Template form.

On the Commands tab, click the "New" button and create the following "Sync Print Word" command.



Command Details

Command

Details

Name \*  
sync\_print\_word

Label \*  
Sync Print Word

Supported Status(es) \*  
Running

Dependent Fields

Timeout (Seconds)

Execution Option  
In Process

Asynchronous

Note that the "Supported Status(es)" is set to **Running** only. This is because In-Process Dynamic commands can only run while the Extension instance is running. For all other task statuses, the command will be greyed out.

Also note that the "Execution Option" is explicitly set to **In Process**, and the **Asynchronous** option is **NOT** checked.

## Step 3 - Add a new text field to the "UE Task" Universal Template

Go back to the "UE Task" Universal Template form.

On the Fields tab, click the "New" button and create the following "Word" field.

The screenshot shows the 'Field Details' configuration window. The 'General' tab is selected, and the 'Field' sub-tab is active. The 'Name' field is set to 'word' and the 'Label' field is set to 'Word'. The 'Hint' field is empty. The 'Add To Default List View' checkbox is unchecked. In the 'Field Details' section, the 'Type' is set to 'Text', 'Text Type' is 'Plain', and 'Mapping' is 'Text Field 4'. The 'Default Value' field is set to 'ABCD'. The 'Restriction' is set to 'No Restriction'.

Ensure that the default value is set to **ABCD**

## Step 4 - Add a backing implementation for both Asynchronous and Synchronous In-Process Dynamic Commands to the extension.py file

Open file `~/dev/extensions/sample-task/src/extension.py` in your editor of choice.

Add the implementation of the `async_print_word()` and `sync_print_word()` Dynamic Commands:

### reset\_environment Dynamic Command

```
from __future__ import (print_function)
import time
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import logger
from universal_extension import ui
from universal_extension.deco import dynamic_choice_command
from universal_extension.deco import dynamic_command

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """
```

```

def __init__(self):
    """Initializes an instance of the 'Extension' class
    """
    # Call the base class initializer
    super(Extension, self).__init__()

@dynamic_choice_command("primary_choice_field")
def primary_choice_command(self, fields):
    """Dynamic choice command implementation for
    primary_choice_field.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """
    return ExtensionResult(
        rc=0,
        message="Values for choice field: 'primary_choice_field'",
        values=["Start", "Pause", "Stop", "Build", "Destroy"]
    )

@dynamic_choice_command("secondary_choice_field")
def secondary_choice_command(self, fields):
    """Dynamic choice command implementation for
    secondary_choice_field.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """
    return ExtensionResult(
        rc=0,
        message="Values for choice field: 'secondary_choice_field'",
        values=["System", "Command", "Application", "Transfer", "Evidence"]
    )

@dynamic_command("reset_environment")
def reset_environment(self, fields):
    """Dynamic command implementation for reset_environment command.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """

    # Reset the state of the Output Only 'step' fields.
    out_fields = {}
    out_fields["step_1"] = "Initial"

```

```

out_fields["step_2"] = "Initial"
ui.update_output_fields(out_fields)

return ExtensionResult(
    message="Message: Hello from dynamic command 'reset_environment'!",
    output=True,
    output_data='The environment has been reset.',
    output_name='DYNAMIC_OUTPUT')

@dynamic_command("async_print_word")
def async_print_word(self, fields):
    """
    Adds each letter of self.WORD to self.async_queue

    If curr_index is odd, then the function will sleep
    for 5 seconds before adding self.WORD to self.async_queue
    """

    curr_index = self.async_index

    if curr_index % 2 != 0:
        self.async_index += 1
        time.sleep(5)
    else:
        self.async_index += 1

    self.async_queue.append(self.WORD[curr_index])

    return ExtensionResult(
        message="",
        output=True,
        output_data="async_print_word()",
        output_name='DYNAMIC_ASYNC_OUTPUT')

@dynamic_command("sync_print_word")
def sync_print_word(self, fields):
    """
    Adds each letter of self.WORD to self.sync_queue

    If curr_index is odd, then the function will sleep
    for 5 seconds before adding self.WORD to self.sync_queue
    """

    curr_index = self.sync_index

    if curr_index % 2 != 0:
        self.sync_index += 1
        time.sleep(5)
    else:
        self.sync_index += 1

    self.sync_queue.append(self.WORD[curr_index])

```

```

return ExtensionResult(
    message="",
    output=True,
    output_data="sync_print_word()",
    output_name='DYNAMIC_SYNC_OUTPUT')

def extension_start(self, fields):
    """Required method that serves as the starting point for work performed
    for a task instance.

    Parameters
    -----
    fields : dict
        populated with field values from the associated task instance
        launched in the Controller

    Returns
    -----
    ExtensionResult
        once the work is done, an instance of ExtensionResult must be
        returned. See the documentation for a full list of parameters that
        can be passed to the ExtensionResult class constructor
    """

    self.WORD = fields['word']
    self.async_index = 0
    self.sync_index = 0

    self.async_queue = []
    self.sync_queue = []

    # Sleep for 40 seconds to keep the Extension instance running
    # while the In-Process Dynamic Commands run
    time.sleep(40)

    # Log the Async and Sync Queues
    logger.info('Async Queue: %s' % ' '.join(self.async_queue))
    logger.info('Sync Queue: %s' % ' '.join(self.sync_queue))

    # Get the value of the 'action' field
    action = fields.get('action', [""])[0]

    if action.lower() == 'print':
        # Print to standard output...
        print("Hello STDOUT!")
    else:
        # Log to standard error...
        logger.info('Hello STDERR!')

    # Return the result with a payload containing a Hello message...
    return ExtensionResult(

```

```

        unv_output='Hello Extension!'
    )

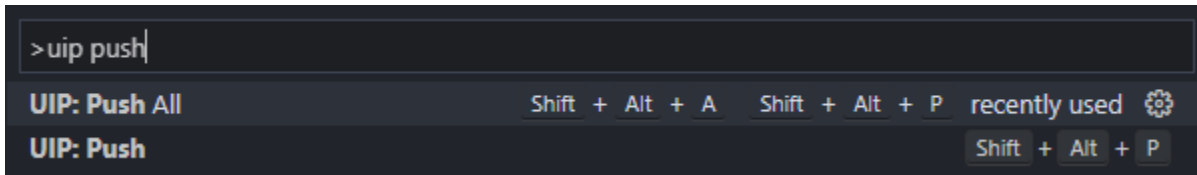
```

<b>Line 75-98</b>	The complete implementation of <code>async_print_word()</code> is defined. In short, the method adds each letter of <code>self.WORD</code> to <code>self.async_queue</code> . If <code>self.async_index</code> is odd, the function sleeps for 5 seconds before adding it to <code>self.async_queue</code> .
<b>Line 100-123</b>	The complete implementation of <code>sync_print_word()</code> is defined. The method does the exact same thing as <code>async_print_word()</code> except it works with <code>self.sync_queue</code> and <code>self.sync_index</code> .
<b>Lines 143-156</b>	The <code>extension_start()</code> method grabs the value of the <code>word</code> field and initializes the queues and index variables. It sleeps for 40 seconds to keep the process running as required by the In-Process Dynamic commands. At the end, it prints out the queues.

## Step 5 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension

Save all changes to `extension.py`.

From the VS Code command palette, execute the **UIP: Push** command as shown below:



Recall that the **UIP: Push** command builds and uploads the Extension zip archive

## Step 5 Supplemental - Build and Upload the Extension Zip Archive Using the CLI

Save all changes to `extension.py`.

From the command line, `cd` to the `~/dev/extensions/sample-task` directory and execute the `push` command as shown below:

```
$ uip push
```

Recall that the `push` command builds and uploads the Extension zip archive

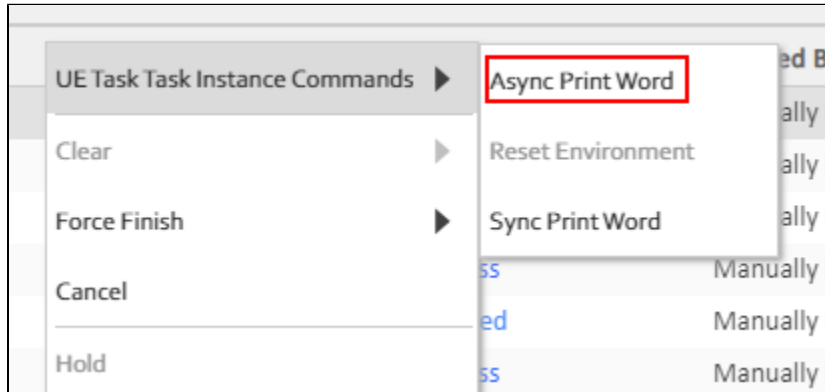
## Step 6 - Execute the Asynchronous Dynamic Command

In the Controller, open a "UE Task Tasks" tab. **If the tab is already open, it must be closed and reopened before the new In-Process Dynamic Commands are available on the task instances.**

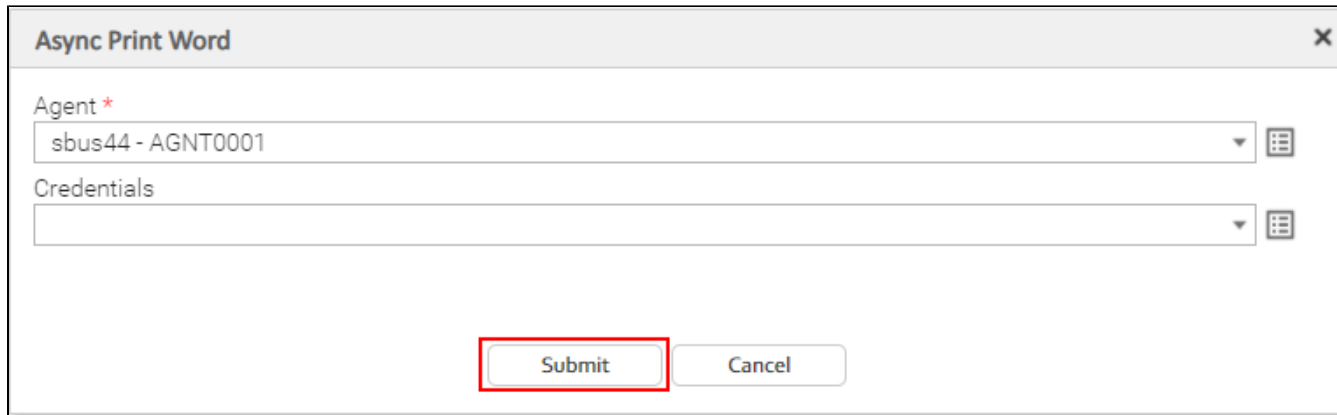
As mentioned in the previous chapter, Dynamic Commands are available from the context menu generated by right-clicking on the task instance form and from the context menu generated by right-clicking on the task instance in a list of task instances (for example, in the Activity Monitor and/or the Instances tab of a task definition form).

It is recommended to read the entire step before executing the dynamic command as it needs to be done in a timely manner to get the proper results

Launch a new task, switch to the Instances tab, right-click on the task-instance (should be in "Running" status), and in the "UE Task Task Instance Commands" dropdown, click "Async Print Word".



This displays an additional dialog, which can be used to pull in dependent fields from the task form if needed. This is optional functionality that is defined in the Universal template and made use of by the backing implementation in the extension archive.



Submit the Dynamic Command.

The immediate result of Dynamic Command is a new window showing the output results of the command.



Close the "Output" dialog, and execute "**Async Print Word**" 4 more times one after the other. Because of the 5 second sleep period when `self.async_queue` is an odd number, the "Output" dialog will not appear immediately for the 2nd (`self.async_queue == 1`) and 4th (`self.async_queue == 3`) execution. Don't wait for the dialog to appear. **Immediately** execute the next Dynamic command after the previous one. In total, the Asynchronous Dynamic Command should be executed 5 times.

After all commands are executed, wait until the task goes to completion. It should end up in "Success" state.

Once finished, right-click on the task instance and click "Retrieve Output" or use the VS Code Extension's "UIP: Task Output" command to retrieve the output as shown below:

```

@sbus44:~/dev/extensions/sample-task$ uip task output ue-task-test
STDERR Output:
=====
2022-03-04 09:20:19,970 - 10612 AsyCommand          - universal_extension.py[572] ERROR: Unhandled exception in Dynamic Command 'async_print_word'.
Traceback (most recent call last):
  File "C:\Program Files\Universal\UAGSrv\uext\universal_extension.zip\universal_extension\universal_extension.py", line 553, in _execute_dynamic_command
    result = _dynamic_commands[command](self, state)
  File "C:\Program Files\Universal\UAGSrv\extensions\ue-task.zip\extension.py", line 91, in async_print_word
    self.async_queue.append(self.WORD[curr_index])
IndexError: string index out of range
2022-03-04 09:20:38,993 - 63188 MainThread        - extension.py[154] INFO: Async Queue: A -> C -> B -> D
2022-03-04 09:20:38,996 - 63188 MainThread        - extension.py[155] INFO: Sync Queue:

STDOUT Output:
=====
Hello STDOUT!

EXTENSION Output:
=====
Hello Extension!

COMMAND Output:
=====
async_print_word()

COMMAND Output:
=====
async_print_word()

COMMAND Output:
=====
async_print_word()

COMMAND Output:
=====
async_print_word()

```

Now, let's examine the output. It will show us what exactly "Asynchronous" means in the context of Dynamic commands.

Recall that the default value of the word field was **ABCD**, but the async queue that's logged in STDERR is: **A C B D**. The **C** and **B** are flipped because of how the Asynchronous Dynamic Command executes:

- When "**Async Print Word**" is first clicked, **self.async\_index** is 0 (even), so the method does not sleep for 5 seconds and immediately adds **A** to **self.async\_queue**
- When "**Async Print Word**" is clicked for the second time, **self.async\_index** is 1 (odd), so the method sleeps for 5 seconds before adding **B** to **self.async\_queue**
- While the second "**Async Print Word**" is sleeping, "**Async Print Word**" is clicked for the third time where **self.async\_index** is 2 (even), so the method does not sleep for 5 seconds and immediately adds **C** to **self.async\_queue**
- After the third instance of "**Async Print Word**" is finished, the fourth instance of "**Async Print Word**" is launched where **self.async\_index** is 3 (odd), so it will sleep for 5 seconds before adding **D** to **self.async\_queue**
- While the fourth instance is sleeping, the second instance will have finished which will result in adding **B** to **self.async\_index**. At the same time, the fifth and last instance of "**Async Print Word**" is launched where **self.async\_index** is 4 (even).
- The fifth instance will end up with an internal failure since index 4 is out of bounds. However, this will **NOT** impact the main Extension instance or any of the other running Dynamic Command instances.
- After a couple seconds, the fourth instance will finish and add **D** to **self.async\_queue**

As noted above, any exceptions in a Dynamic Command instance will not affect the Extension instance or any other Dynamic Command instances. We can see this clearly by examining the timestamps of the Exception (**09:20:19**) vs. the log statements (**09:20:38**).

The exception occurred first, but that did not prevent the log statements from being printed in **extension\_start()**

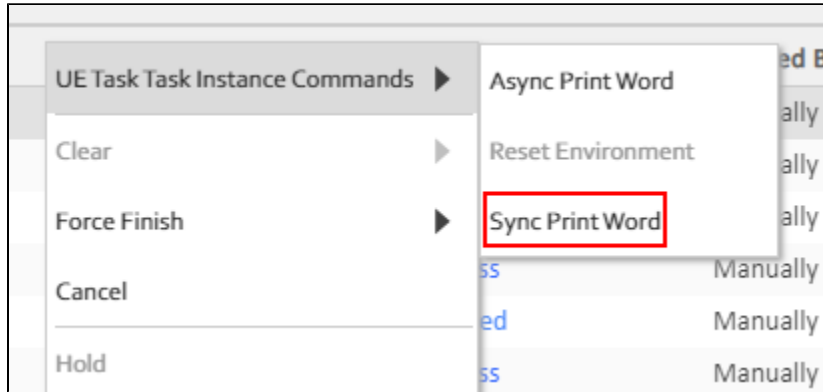
## Step 7 - Execute the Synchronous Dynamic Command

In the Controller, open a "UE Task Tasks" tab. **If the tab is already open, it must be closed and reopened before the new In-Process Dynamic Commands are available on the task instances.**

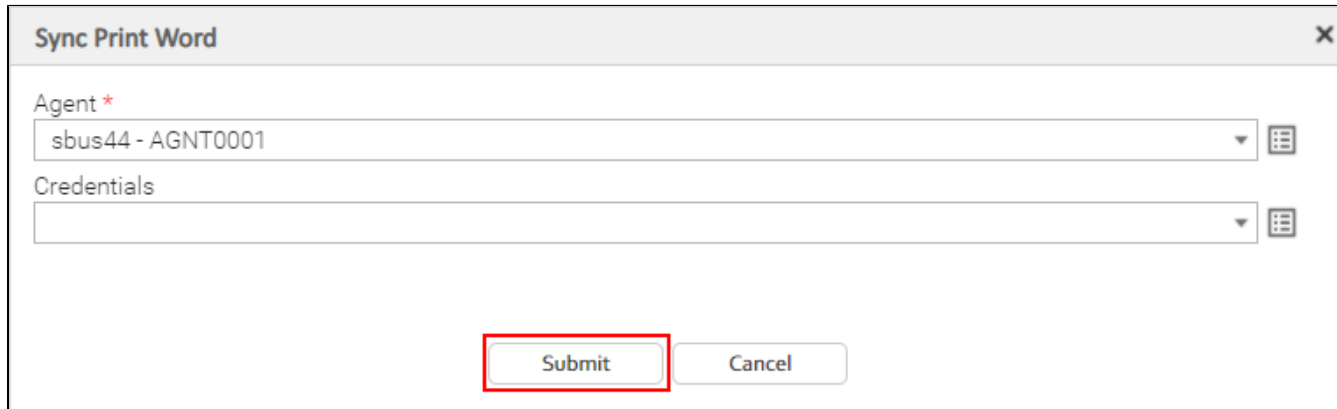
As mentioned in the previous chapter, Dynamic Commands are available from the context menu generated by right-clicking on the task instance form and from the context menu generated by right-clicking on the task instance in a list of task instances (for example, in the Activity Monitor and/or the Instances tab of a task definition form).

It is recommended to read the entire step before executing the dynamic command as it needs to be done in a timely manner to get the proper results

Launch a new task, switch to the Instances tab, right-click on the task-instance (should be in "Running" status), and in the "UE Task Task Instance Commands" dropdown, click "**Sync Print Word**".



This displays an additional dialog, which can be used to pull in dependent fields from the task form if needed. This is optional functionality that is defined in the Universal template and made use of by the backing implementation in the extension archive.



Submit the Dynamic Command.

The immediate result of Dynamic Command is a new window showing the output results of the command.



Close the "Output" dialog, and execute "**Sync Print Word**" 4 more times one after the other. In total, the Synchronous Dynamic Command should be executed 5 times. Note that the commands must be executed immediately after the previous one.

After all commands are executed, wait until the task goes to completion. It should end up in "Success" state.

Once finished, right-click on the task instance and click "Retrieve Output" or use the VS Code Extension's "UIP: Task Output" command to retrieve the output as shown below:

```

@sbus44:~/dev/extensions/sample-task$ uip task output ue-task-test
STDERR Output:
=====
2022-03-04 09:31:47,207 - 17580 SynCommandProcessor - universal_extension.py[572] ERROR: Unhandled exception in Dynamic Command 'sync_print_word'.
Traceback (most recent call last):
  File "C:\Program Files\Universal\UAGSrv\uext\universal_extension.zip\universal_extension\universal_extension.py", line 553, in _execute_dynamic_command
    result = _dynamic_commands[command](self, state)
  File "C:\Program Files\Universal\UAGSrv\extensions\ue-task.zip\extension.py", line 116, in sync_print_word
    self.sync_queue.append(self.WORD[curr_index])
IndexError: string index out of range
2022-03-04 09:32:04,282 - 72120 MainThread          - extension.py[154] INFO: Async Queue:
2022-03-04 09:32:04,285 - 72120 MainThread          - extension.py[155] INFO: Sync Queue: A -> B -> C -> D

STDOUT Output:
=====
Hello STDOUT!

EXTENSION Output:
=====
Hello Extension!

COMMAND Output:
=====
sync_print_word()

COMMAND Output:
=====
sync_print_word()

COMMAND Output:
=====
sync_print_word()

COMMAND Output:
=====
sync_print_word()

```

Notice that unlike the Asynchronous Dynamic Command, the Synchronous one logs the word in order: **A B C D**. This is because a Synchronous Dynamic Command cannot start processing until all previously received Synchronous Dynamic Commands have completed.

Similar to the Asynchronous Dynamic Command, the Exception does not impact the Extension instance or any other Dynamic Command instances.

## Step 8 - Update the Local template.json

In steps 1 to 3, we modified the Universal Template by adding new Dynamic Commands.

Recall that inside `~/dev/extensions/sample-task/src/templates`, there is a **template.json** file. This file should correspond to the Universal Template in the Controller.

Right now, they are not both the same. The Controller's version of **template.json** has the In-Process Dynamic Command changes. To grab those changes, use the **pull** command as shown below:

### UIP VS Code Extension

```

> uip pull
UIP: Pull
Shift + Alt + L recently used

```

```
████████@sbus44:~/dev/extensions/sample-task$ uip pull
The following files were updated:
- template.json
```

### Step Supplemental - CLI

```
████████@sbus44:~/dev/extensions/sample-task$ uip pull
The following files were updated:
- template.json
```

Now, both the local and Controller's version of the Universal Template are the same.

[< Previous](#) [Next >](#)

# Cancel Command

- [Introduction](#)
- [Step 1 - Add a new "cancel\\_cleanup\\_time" field to the "UE Task" Universal Template](#)
- [Step 2 - Add a backing implementation for Cancel Command to the extension.py file](#)
- [Step 2 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension](#)
  - [Step 2 Supplemental - Build and Upload the Extension Zip Archive Using the CLI](#)
- [Step 3 - Demonstrate Cancel Command](#)
  - [a. Graceful Cancellation](#)
  - [b. Timeout](#)
  - [c. Double Cancel](#)
- [Step 4 - Update the Local template.json](#)
  - [UIP VS Code Extension](#)
    - [Step Supplemental - CLI](#)

## Introduction

In versions prior to 7.1.0.0, Universal Extension task instances could be cancelled via the Controller just like any other task type, but the instances do not participate in the Cancellation process. As a result, there is no chance to do any sort of cleanup before Cancellation. Starting with 7.1.0.0, the Universal Extension API was enhanced with a new method called **extension\_cancel()** which allows for any cleanup work before the process is terminated.

On this page, we will cover the following:

1. Add a new `cancel_cleanup_time` field to the "UE Task" Universal Template
2. Add a backing implementation for Cancel Command to the `extension.py` file.
3. Build the modified Extension.
4. Upload the modified Extension.
5. Demonstrate the Cancel command in three different scenarios
  - a. Graceful Cancellation
  - b. Timeout
  - c. Double Cancel

## Step 1 - Add a new "cancel\_cleanup\_time" field to the "UE Task" Universal Template

Navigate to the "UE Task" Universal Template.

In the "Fields" tab, add a new field called "cancel\_cleanup\_time" as shown below:

The screenshot shows a configuration window titled "Field Details: cancel\_sleep\_value". It has two tabs: "Field" (selected) and "Choices". Under the "General" section, there are fields for "Name \*" (containing "cancel\_cleanup\_time") and "Label \*" (containing "Cancel Cleanup Time"). Below these is a "Hint" field and a checkbox for "Add To Default List View". The "Field Details" section includes a "Type" dropdown (set to "Integer"), a "Mapping" dropdown (set to "Integer Field 1"), a "Default Value" input (containing "0"), and a "Restriction" section with radio buttons for "No Restriction" (selected) and "Output Only".

The value of this field will be used to simulate the time spent doing the cleanup work in `extension_cancel()`.

Save the field.

## Step 2 - Add a backing implementation for Cancel Command to the extension.py file

Open file `~/dev/extensions/sample-task/src/extension.py` in your editor of choice.

Add the implementation of the `extension_cancel()` method:

### reset\_environment Dynamic Command

```
from __future__ import (print_function)
import time
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import logger
from universal_extension import ui
from universal_extension.deco import dynamic_choice_command
from universal_extension.deco import dynamic_command

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """
```

```

def __init__(self):
    """Initializes an instance of the 'Extension' class
    """
    # Call the base class initializer
    super(Extension, self).__init__()

@dynamic_choice_command("primary_choice_field")
def primary_choice_command(self, fields):
    """Dynamic choice command implementation for
    primary_choice_field.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """
    return ExtensionResult(
        rc=0,
        message="Values for choice field: 'primary_choice_field'",
        values=["Start", "Pause", "Stop", "Build", "Destroy"]
    )

@dynamic_choice_command("secondary_choice_field")
def secondary_choice_command(self, fields):
    """Dynamic choice command implementation for
    secondary_choice_field.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """
    return ExtensionResult(
        rc=0,
        message="Values for choice field: 'secondary_choice_field'",
        values=["System", "Command", "Application", "Transfer", "Evidence"]
    )

@dynamic_command("reset_environment")
def reset_environment(self, fields):
    """Dynamic command implementation for reset_environment command.

    Parameters
    -----
    fields : dict
        populated with the values of the dependent fields
    """

    # Reset the state of the Output Only 'step' fields.
    out_fields = {}
    out_fields["step_1"] = "Initial"
    out_fields["step_2"] = "Initial"

```

```

    ui.update_output_fields(out_fields)

    return ExtensionResult(
        message="Message: Hello from dynamic command 'reset_environment'",
        output=True,
        output_data='The environment has been reset.',
        output_name='DYNAMIC_OUTPUT')

@dynamic_command("async_print_word")
def async_print_word(self, fields):
    """
    Adds each letter of self.WORD to self.async_queue

    If curr_index is odd, then the function will sleep
    for 5 seconds before adding self.WORD to self.async_queue
    """

    curr_index = self.async_index

    if curr_index % 2 != 0:
        self.async_index += 1
        time.sleep(5)
    else:
        self.async_index += 1

    self.async_queue.append(self.WORD[curr_index])

    return ExtensionResult(
        message="",
        output=True,
        output_data="async_print_word()",
        output_name='DYNAMIC_ASYNC_OUTPUT')

@dynamic_command("sync_print_word")
def sync_print_word(self, fields):
    """
    Adds each letter of self.WORD to self.sync_queue

    If curr_index is odd, then the function will sleep
    for 5 seconds before adding self.WORD to self.sync_queue
    """

    curr_index = self.sync_index

    if curr_index % 2 != 0:
        self.sync_index += 1
        time.sleep(5)
    else:
        self.sync_index += 1

    self.sync_queue.append(self.WORD[curr_index])

```

```

return ExtensionResult(
    message="",
    output=True,
    output_data="sync_print_word()",
    output_name='DYNAMIC_SYNC_OUTPUT')

def extension_start(self, fields):
    """Required method that serves as the starting point for work performed
    for a task instance.

    Parameters
    -----
    fields : dict
        populated with field values from the associated task instance
        launched in the Controller

    Returns
    -----
    ExtensionResult
        once the work is done, an instance of ExtensionResult must be
        returned. See the documentation for a full list of parameters that
        can be passed to the ExtensionResult class constructor
    """

    # Extract the cancel_cleanup_time
    self.cancel_cleanup_time = fields.get('cancel_cleanup_time', 0)

    # Sleep for 45 seconds
    time.sleep(45)

    # Get the value of the 'action' field
    action = fields.get('action', [""])[0]

    if action.lower() == 'print':
        # Print to standard output...
        print("Hello STDOUT!")
    else:
        # Log to standard error...
        logger.info('Hello STDERR!')

    # Return the result with a payload containing a Hello message...
    return ExtensionResult(
        unv_output='Hello Extension!'
    )

def extension_cancel(self):
    """Optional method that allows the Extension instance to perform
    any cleanup work.
    """
    logger.info('About to sleep for %d seconds' % self.cancel_cleanup_time)
    time.sleep(self.cancel_cleanup_time)

```

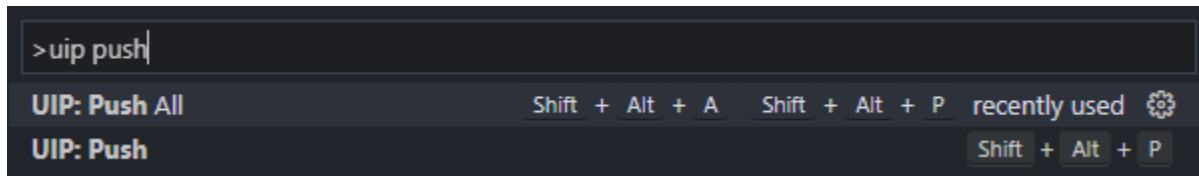
```
logger.info('Done with extension_cancel()')
```

<b>Line 144</b>	The <b>cancel_cleanup_time</b> field is extracted from <b>fields</b> and stored in <b>self.cancel_cleanup_time</b> so that it can be accessed in <b>extension_cancel()</b>
<b>Lines 147</b>	The <b>extension_start()</b> method is modified to sleep for 45 seconds.
<b>Lines 164-170</b>	The <b>extension_cancel()</b> method sleeps for <b>self.cancel_cleanup_time</b> number of seconds. It also logs two statements; one before and one after the sleep time period.

## Step 2 - Build and Upload the Extension Zip Archive Using the UIP VS Code Extension

Save all changes to `extension.py`.

From the VS Code command pallet, execute the **UIP: Push** command as shown below:



Recall that the **UIP: Push** command builds and uploads the Extension zip archive

## Step 2 Supplemental - Build and Upload the Extension Zip Archive Using the CLI

Save all changes to `extension.py`.

From the command line, `cd` to the `~/dev/extensions/sample-task` directory and execute the **push** command as shown below:

```
$ uip push
```

Recall that the **push** command builds and uploads the Extension zip archive

## Step 3 - Demonstrate Cancel Command

Before working with the Cancel command, we will need to make sure the cancel timeout value (this is NOT the `cancel_cleanup_time` field above) is set to 10 seconds. Unless you have explicitly modified the value, it will be 10 seconds by default. Open `uags.conf`, and if there is an entry for **extension\_cancel\_timeout**, make sure it is 10 seconds.

### a. Graceful Cancellation

Graceful Cancellation is when the **extension\_cancel()** method finishes before the 10 second Cancel timeout period is up.

- Navigate to the "ue-task-test" task instance form, and ensure the **Cancel Cleanup Time** field is set to 0 as shown:

The screenshot shows a form titled "UE Task Details" with the following fields:

- Action:** Print to STDOUT
- Primary Choice \*:** Start
- Secondary Choice \*:** Application
- Word:** ABCD
- Cancel Cleanup Time:** 0 (highlighted with a red box)

- Launch the task using the VS Code "UIP: Task Launch" command
- After about 2-3 seconds, right-click the task instance on the Controller and click "Cancel"
- Wait until the task status is "Cancelled"
- The entire process should look similar to:

The screenshot shows a terminal window with the following content:

```

TERMINAL  PROBLEMS  5  OUTPUT  DEBUG CONSOLE
┌─┐ UIP + ▾ 🗑
└─┘
jerry@bus44:~/dev/extensions/sample-task$ uip task launch ue-task-test

```

Notice that both the log statements were printed since the timeout period of 10 seconds did not expire before the **extension\_cancel()** cleanup finished.

## b. Timeout

Timeout is when the **extension\_cancel()** method has not finished before the 10 second Cancel timeout period is up.

- Navigate to the "ue-task-test" task instance form, and ensure the **Cancel Cleanup Time** field is set to 15 as shown:

The screenshot shows the 'UE Task Details' form with the following fields:

- Action:** Print to STDOUT
- Primary Choice \*:** Start
- Secondary Choice \*:** Application
- Word:** ABCD
- Cancel Cleanup Time:** 15 (highlighted with a red box)

- Launch the task using the VS Code "UIP: Task Launch" command
- After about 2-3 seconds, right-click the task instance on the Controller and click "Cancel"
- Wait until the task status is "Cancelled"
- The entire process should look similar to:

The terminal window shows the following command being executed:

```
uip task launch ue-task-test
```

Notice that only the first log statement was printed. Since the cancel cleanup time value was set to 15 seconds, the Cancel command timed out after 10 seconds, and the Extension process was forcefully terminated.

### c. Double Cancel

Double Cancel is when the Extension instance is "Cancelled" twice from the Controller. When the agent receives the second Cancel, it immediately terminates the Extension process regardless of whether the timeout has occurred or not.

- Navigate to the "ue-task-test" task instance form, and ensure the **Cancel Cleanup Time** field is set to 15 as shown:

UE Task Details

Action: Print to STDOUT

Primary Choice \*: Start

Secondary Choice \*: Application

Word: ABCD

Cancel Cleanup Time: 15

- Launch the task using the VS Code "UIP: Task Launch" command
- After about 2-3 seconds, right-click the task instance on the Controller and click "Cancel"
- Immediately after, right-click the task instance and click "Cancel" once again.
- The status should immediately transition to "Cancelled"
- The entire process should look similar to:

```

TERMINAL  PROBLEMS  5  OUTPUT  DEBUG CONSOLE  UIP + - [ ]
ali@bus44:~/dev/extensions/sample-task$ uip task launch ue-task-test[]

```

Notice that only the first log statement was printed. Since the Extension process was terminated by the Double Cancel before the 15 second sleep period was up, the second log statement did not get printed.

## Step 4 - Update the Local template.json

In step 1, we modified the Universal Template by adding the new `cancel_cleanup_time` field. Recall that inside `~/dev/extensions/sample-task/src/templates`, there is a **template.json** file. This file should correspond to the Universal Template in the Controller.

Right now, they are not both the same. The Controller's version of **template.json** has the new field. To grab those changes, use the **pull** command as shown below:

## UIP VS Code Extension

```

>uip pull

```

UIP: Pull Shift + Alt + L recently used

```
████████@sbus44:~/dev/extensions/sample-task$ uip pull  
The following files were updated:  
- template.json
```

### Step Supplemental - CLI

```
████████@sbus44:~/dev/extensions/sample-task$ uip pull  
The following files were updated:  
- template.json
```

Now, both the local and Controller's version of the Universal Template are the same.

[< Previous](#) [Next >](#)

# Publishing Events

- [Introduction](#)
- [Step 1 - Set up the new "UE Publisher" template](#)
- [Step 2 - Add a directory field to the "UE Publisher" template](#)
- [Step 3 - Enhance the "UE Publisher's" local event template](#)
- [Step 4 - Modify the ue-publisher extension](#)
- [Step 5 - Create a task for the "UE Publisher" template](#)
- [Step 6 - Create a Universal Monitor task and trigger](#)
- [Step 7 - Run the Universal Monitor Trigger](#)
- [Step 8 - Update the local template.json](#)
  - [UIP VS Code Extension](#)
  - [Step Supplemental - CLI](#)

## Introduction

As part of 7.2.0.0, Universal Extensions can now be used to extend the Controller's monitoring capabilities through Universal Events and Universal Monitors/Universal Monitor Triggers.

Specifically, the Universal Extension API has been enhanced to supported publishing events using the **publish** method from the **event** module.

Throughout this document, the event publishing functionality will be demonstrated using a contrived file monitor example.

On this page, we will cover the following:

1. Set up the new **UE Publisher** template.
2. Add a **directory** field to the **UE Publisher** template.
3. Enhance the **UE Publisher's** local event template.
4. Modify the **ue-publisher** Extension.
5. Create a task for the **UE Publisher** template.
6. Create a Universal Monitor task and trigger.
7. Run the Universal Monitor Trigger.
8. Update the local template.json.

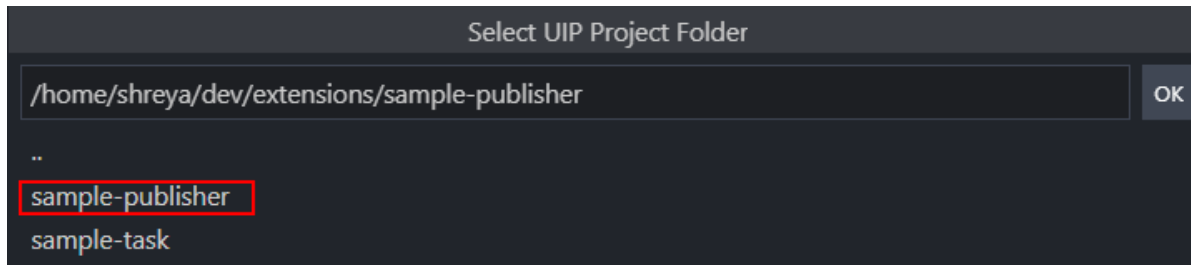
## Step 1 - Set up the new "UE Publisher" template

Up until now, we have been working with the "UE Task" Universal Template. For this tutorial, we will need to use the "UE Publisher" template that was added to the VS Code Extension and UIP-CLI.

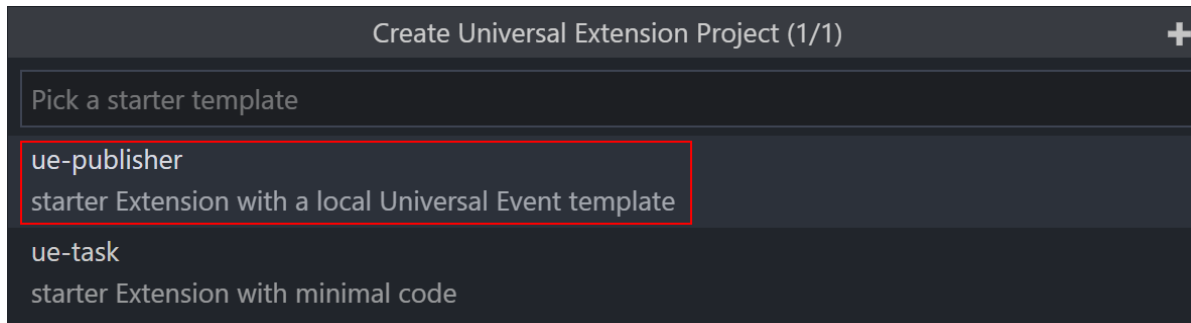
Create a new folder `~/dev/extensions/sample-publisher` where the new template will be stored. Navigate to that directory and run the **UIP: Initialize New Project** command as shown below:



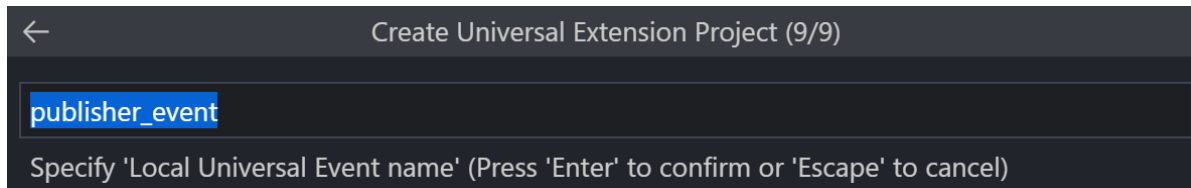
On the following prompt, select the "sample-publisher" folder:



Then, select the "ue-publisher" starter template:



In the following prompts that are used to configure the template, everything can be kept as is. Note the "UE Publisher" template has an additional option called "Local Universal Event name" shown below.



Open the `~/dev/extensions/sample-publisher/extension.py` if currently not open. Let's examine the file:

#### extension.py

```
from __future__ import (print_function)
from time import sleep
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import event

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """
```

```

def __init__(self):
    """Initializes an instance of the 'Extension' class
    """
    # Call the base class initializer
    super(Extension, self).__init__()

    # Flag to control the event loop below
    self.run = True

def extension_start(self, fields):
    """Required method that serves as the starting point for work performed
    for a task instance.

    Parameters
    -----
    fields : dict
        populated with field values from the associated task instance
        launched in the Controller

    Returns
    -----
    ExtensionResult
        once the work is done, an instance of ExtensionResult must be
        returned. See the documentation for a full list of parameters that
        can be passed to the ExtensionResult class constructor
    """

    # Get the value of the 'sleep_value' field
    sleep_value = fields.get('sleep_value', 3)

    # loop that publishes events continuously as long as self.run is True
    while self.run:
        # Publish an event
        event.publish(
            'publisher_event',
            {}
        )

        # sleep before publishing next event
        sleep(sleep_value)

    # Return the result with a payload marking the end of extension_start()
    return ExtensionResult(
        unv_output='extension_start() finished'
    )

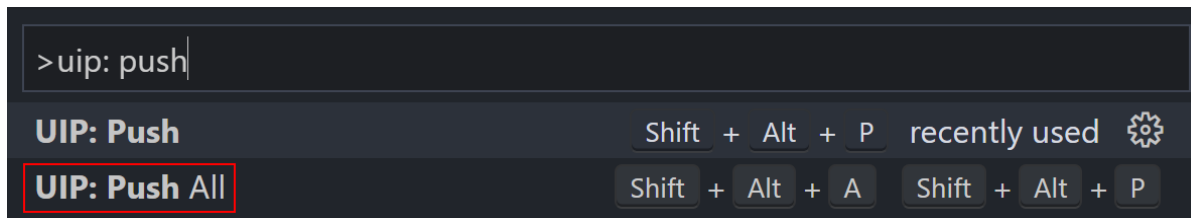
def extension_cancel(self):
    """Optional method that allows the Extension to do any cleanup work
    before finishing
    """
    # Set self.run to False which will end the event loop above

```

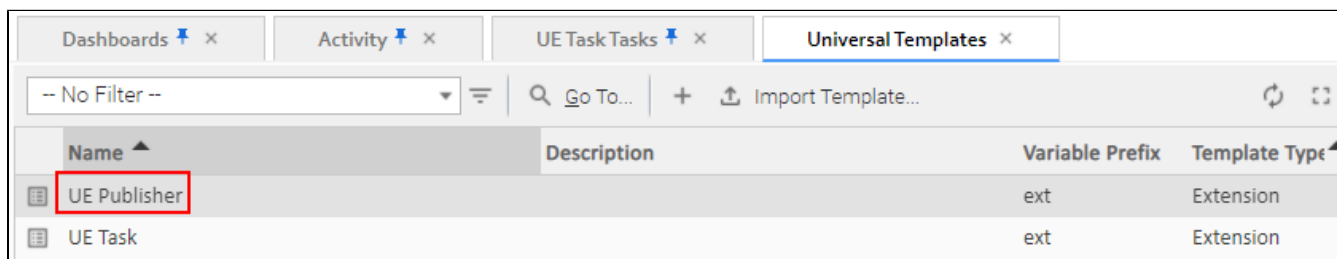
```
self.run = False
```

<b>Line 2</b>	Imports the sleep method from the time module which will be used to add a delay between successive events.
<b>Line 5</b>	Imports the event module which contains the <b>publish</b> method used later on.
<b>Line 19</b>	A flag called <b>self.run</b> with initial value of <b>True</b> is added to control the event loop.
<b>Line 40</b>	Extract the <b>sleep_value</b> field from the task instance passed down by the Controller.
<b>Line 43</b>	While loop that runs as long as the <b>self.run</b> field value is <b>True</b>
<b>Line 45-48</b>	Used to publish an empty event, as shown by the empty attributes dictionary.
<b>Line 51</b>	Used to add a tiny delay before the next event is published
<b>Line 58-63</b>	Upon receiving the Cancel command from the Controller, the <b>self.run</b> flag is set to <b>False</b>

Now, let's push the extension out to the Controller. Since this is the first time, use the **UIP: Push All** command:



If successful, you should see the "UE Publisher" template in the "Universal Templates" list:



## Step 2 - Add a directory field to the "UE Publisher" template

Navigate to the **Fields** tab of the **UE Publisher** template, and add a new field as shown below:

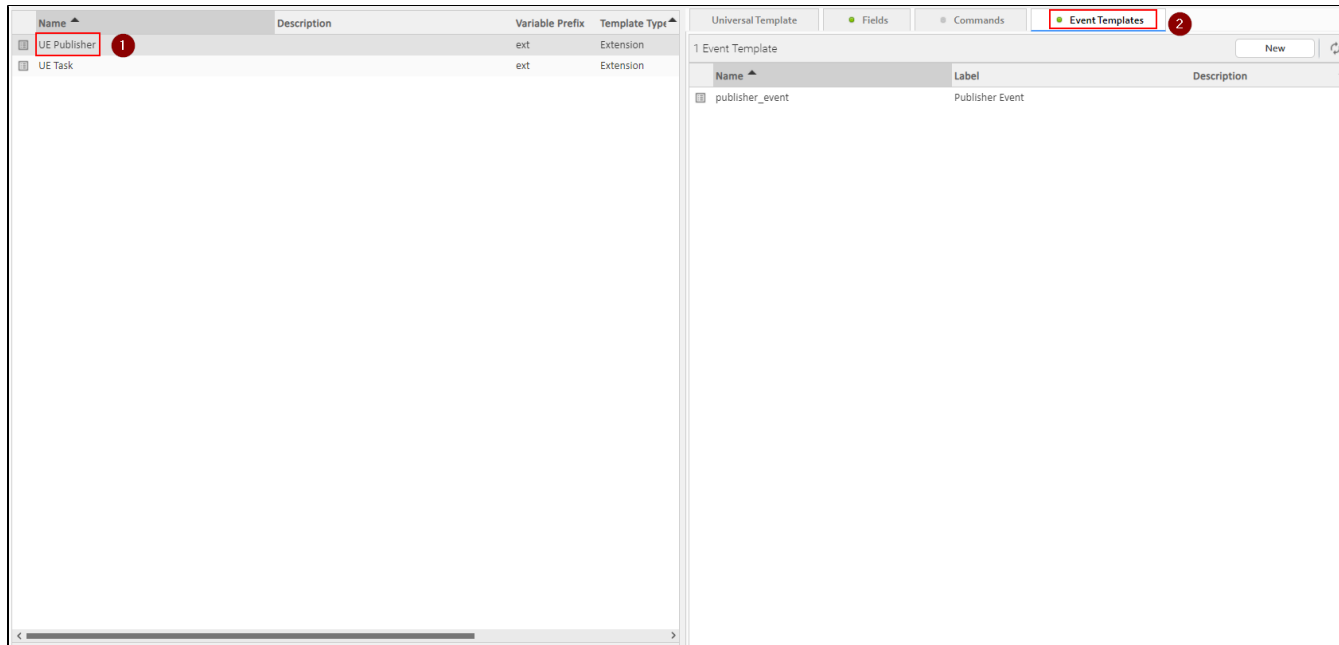
The screenshot shows the 'Field Details' configuration window for a field named 'directory'. The 'Name' field is 'directory' and the 'Label' field is 'Directory'. The 'Type' is set to 'Text' and the 'Text Type' is 'Plain'. The 'Mapping' is 'Text Field 1'. The 'Required' checkbox is checked. The 'Restriction' is set to 'No Restriction'. The 'Validation' section includes 'Required' (checked), 'Require If Field' (set to '-- None --'), and 'Show If Field' (set to '-- None --').

This required field will be used to specify the directory to get the file list of.

Save the template.

### Step 3 - Enhance the "UE Publisher's" local event template

Navigate to the **Event Templates** tab of the **UE Publisher** template:



Double click the **publisher\_event** entry, and modify it as shown below:

Event Template Details: Publisher Event

Event Template

Details

Name \* publisher\_event Label \* Publisher Event

Description

Time To Live 5 Unmapped Attributes Policy Prohibit Universal Event

Attributes

Name	Label	Type
filelist	Filelist	Text

A new attribute of type **Text** called **filelist** is added which will contain the list of files (formatted as a string) in the directory specified by the **directory** field added in the last step.

Save the Event Template.

## Step 4 - Modify the ue-publisher extension

Now, we need to enhance `~/dev/extensions/sample-publisher/extension.py` to publish the filelist event. Update the file as shown below:

### extension.py

```
from __future__ import (print_function)
from time import sleep
import os
from universal_extension import UniversalExtension
from universal_extension import ExtensionResult
from universal_extension import event

class Extension(UniversalExtension):
    """Required class that serves as the entry point for the extension
    """

    def __init__(self):
```

```

    """Initializes an instance of the 'Extension' class
    """
    # Call the base class initializer
    super(Extension, self).__init__()

    # Flag to control the event loop below
    self.run = True

def extension_start(self, fields):
    """Required method that serves as the starting point for work performed
    for a task instance.

    Parameters
    -----
    fields : dict
        populated with field values from the associated task instance
        launched in the Controller

    Returns
    -----
    ExtensionResult
        once the work is done, an instance of ExtensionResult must be
        returned. See the documentation for a full list of parameters that
        can be passed to the ExtensionResult class constructor
    """

    # Get the value of the 'sleep_value' field
    sleep_value = fields.get('sleep_value', 3)

    # Get the value of the 'directory' field
    directory = fields.get('directory', '')

    # Verify the directory exists
    if not directory:
        return ExtensionResult(
            rc=-1,
            unv_output="specified directory is empty"
        )
    elif not os.path.exists(directory):
        return ExtensionResult(
            rc=-1,
            unv_output="specified directory does not exist"
        )

    prev_filelist = set()
    # loop that publishes events continuously as long as self.run is True
    while self.run:
        curr_filelist = set(os.listdir(directory))

        # Subtracting 'prev_filelist' from 'curr_filelist' will ensure the
        # 'filelist' only contains the newly detected files.
        filelist = curr_filelist - prev_filelist

```

```

        filelist = ','.join(filelist)

        prev_filelist = curr_filelist.copy()

        if filelist:
            print(filelist)

        # Publish the event
        event.publish(
            'publisher_event',
            {"filelist": filelist}
        )

        # sleep before publishing next event
        sleep(sleep_value)

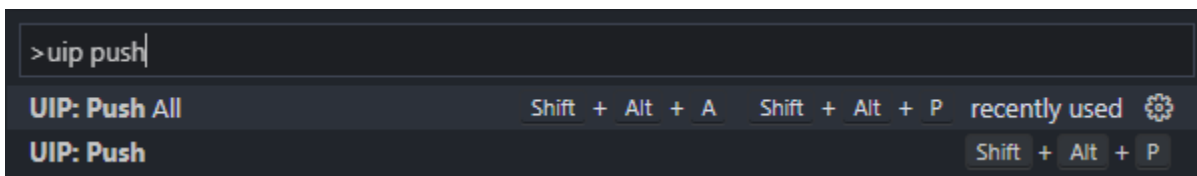
    # Return the result with a payload marking the end of extension_start()
    return ExtensionResult(
        unv_output='extension_start() finished'
    )

def extension_cancel(self):
    """Optional method that allows the Extension to do any cleanup work
    before finishing
    """
    # Set self.run to False which will end the event loop above
    self.run = False

```

<b>Line 3</b>	Imports the <code>os</code> module used to list files in an directory.
<b>Line 44</b>	Extract the <b>directory</b> field from the task instance passed down by the Controller.
<b>Line 47-56</b>	Verify the <b>directory</b> exists
<b>Line 58-80</b>	Get the current directory listing and remove the files from the previous iteration, only leaving new files discovered in the current iteration. Then, publish the filtered file listing as a comma-separated string to the <b>publisher_event</b> event.

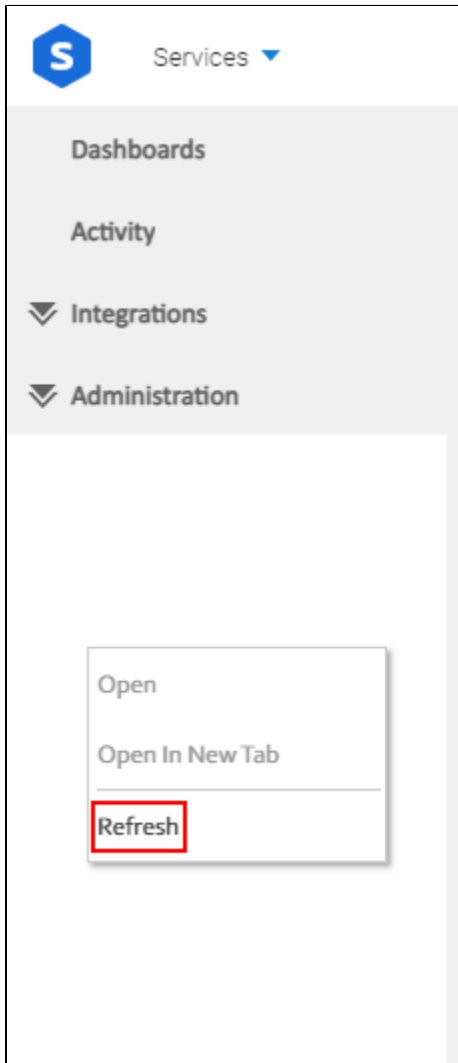
Save the changes to `extension.py` and execute the **UIP: Push** command as shown below:



Recall that the **UIP: Push** command builds and uploads the Extension zip archive

## Step 5 - Create a task for the "UE Publisher" template

Right-click on the navigation tree pane, and click **Refresh** as shown below:



Then under the **Services** menu, click **UE Publisher Tasks** under the **Integrations** section:

The screenshot shows the Universal Controller 7.2.x interface. At the top left, there is a navigation menu with a 'Services' dropdown menu. A red box highlights the 'Services' dropdown, and a red circle with the number 1 is next to it. Below the navigation menu, there are several sections: 'Recently Visited', 'Available Services', 'Integrations', 'Triggers', 'Reports', 'Lifecycle Management', 'Agents', 'Connections', and 'System'. The 'Available Services' section contains a list of services, including 'Dashboards', 'Activity', 'Workflows', 'Instances', 'Tasks', 'Monitors', and 'Calendars'. The 'UE Publisher Tasks' item is highlighted with a red box, and a red circle with the number 2 is next to it.

Create a new task called **sample-publisher-task**:

UE PublisherTask
Variables
Actions
Virtual Resources
Mutually Excludi

### General

Name \*  Version

Description

Member of Business Services

Resolve Name Immediately  Time Zone Preference

Hold on Start

Virtual Resource Priority  Hold Resources on Failure

Log Level

### Agent Details

Cluster

Agent \*  Agent Variable

Credentials  Credentials Variable

Run with Highest Privileges

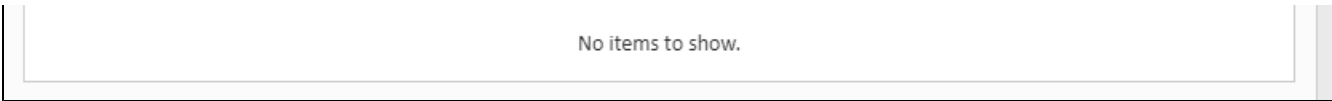
### UE Publisher Details

Sleep Value (seconds)  Directory \*

Runtime Directory

Environment Variables

Name	Value



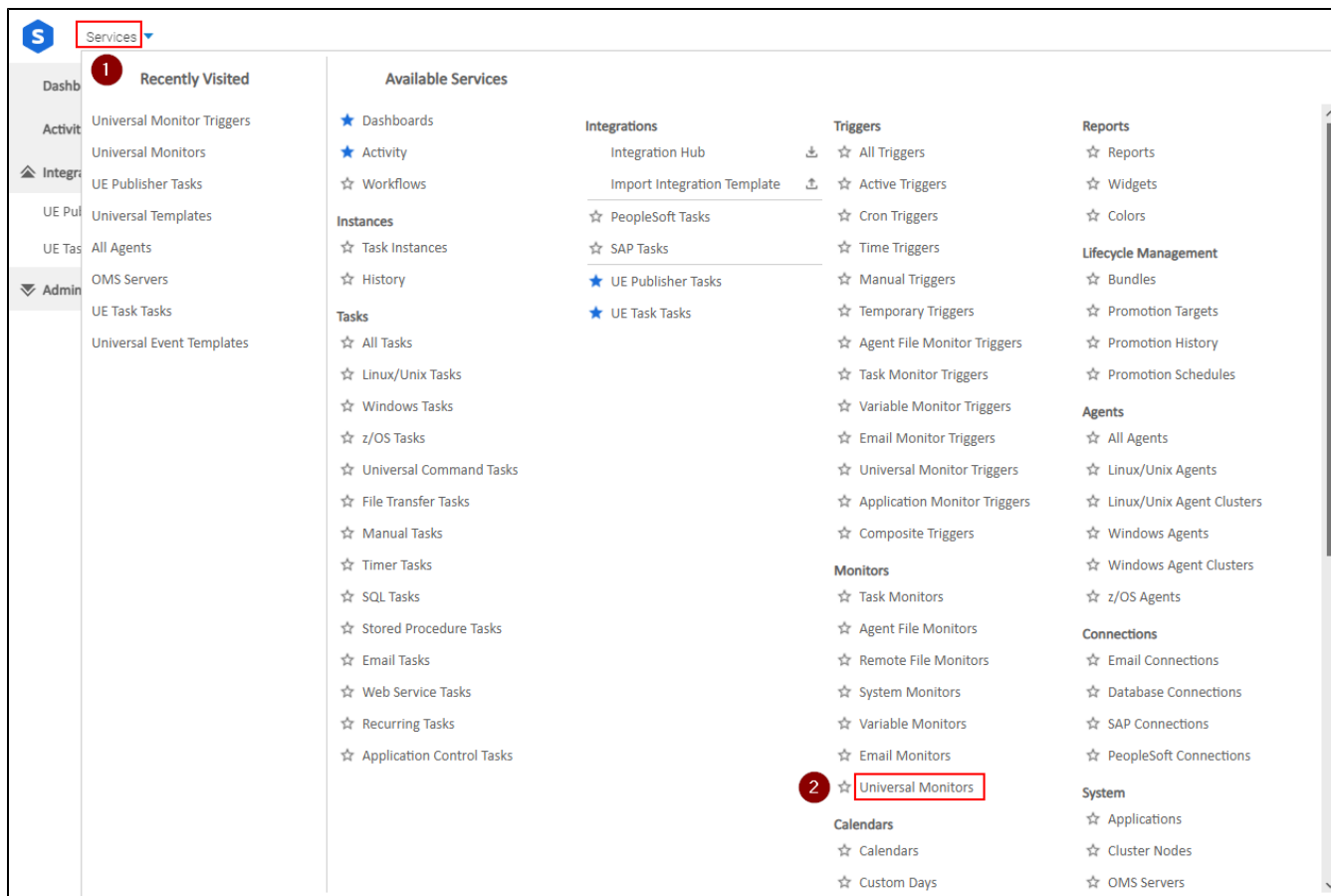
- For the directory field, type in a valid path on the system where the selected agent is running

Save the task.

## Step 6 - Create a Universal Monitor task and trigger

Now that we have the event publishing logic added, we need to attach the event itself to a Universal Monitor task.

Under the **Services** menu, click **Universal Monitors** under the **Monitors** section as shown below:



Create a new Universal Monitor task as shown below:

The screenshot shows the configuration for a Universal Monitor task. The 'Name' field is 'sample-monitor-task'. Under 'Universal Monitor Details', the 'Event Type' is 'Local', 'Universal Template' is 'UE Publisher', 'Event Template' is 'Publisher Event', and 'Universal Task Publisher' is 'sample-publisher-task'. Under 'Universal Monitor Criteria', 'Match All' is selected, and a criterion is defined as 'Filelist contains test.txt'.

- Make sure the **Event Type** is set to **Local** since the **ue\_publisher** Event Template is (locally) attached to the **UE Publisher** Universal Template
- Make sure the **Universal Template** is set to **UE Publisher**
- Make sure the **Event Template** is set to **Publisher Event** (this is the user-friendly name of **ue\_publisher** Event Template)
- Make sure the **Universal Task Publisher** is set to **sample-publisher-task**
- Make sure there is an entry for **Filelist** as shown above. As you may have guessed, this Universal Monitor task will check if **Filelist** contains **test.txt**

Save the task.

We can either run the **sample-monitor-task** as a standalone task which finishes upon detecting the target file (**test.txt**), or we can attach the task to a Universal Monitor trigger. To cover the complete functionality, we will attach it to a Universal Monitor Trigger.

Under the **Services** menu, click **Universal Monitor Triggers** under the **Triggers** section. Create a new trigger as shown below:

**Universal Monitor Trigger**    ● Variables    ● Instances    ● Notes    ● Versions

**General**

Name \* **1**  
sample-monitor-trigger

Description

Member of Business Services

Calendar \*    Time Zone  
System Default    Server (America/New\_York)

Task(s) \*  
Sleep 0 **2**

Purge By Retention Duration

**Skip Details**

Task Launch Skip Condition  
-- None --

Skip Restriction    Skip Count  
-- None --    0

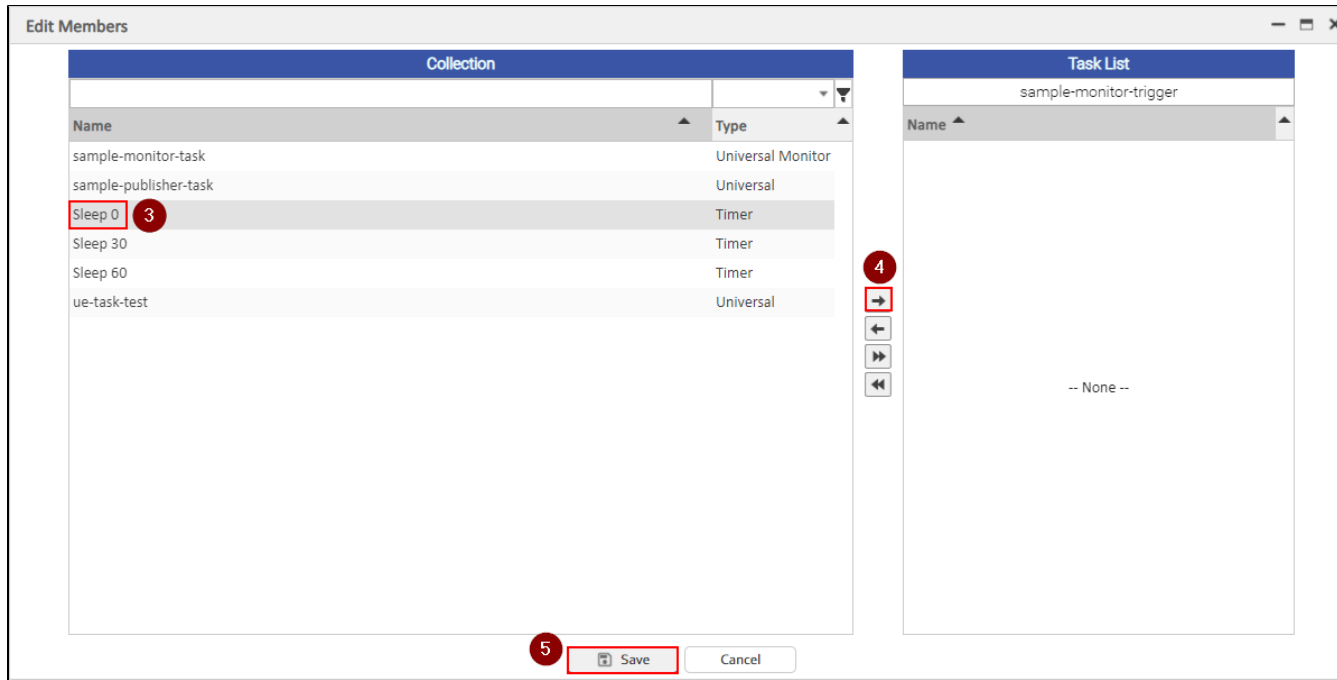
**Universal Monitor Details**

Universal Monitor \* **6**  
sample-monitor-task

**Restrictions**

Restrict Times

Special Restriction

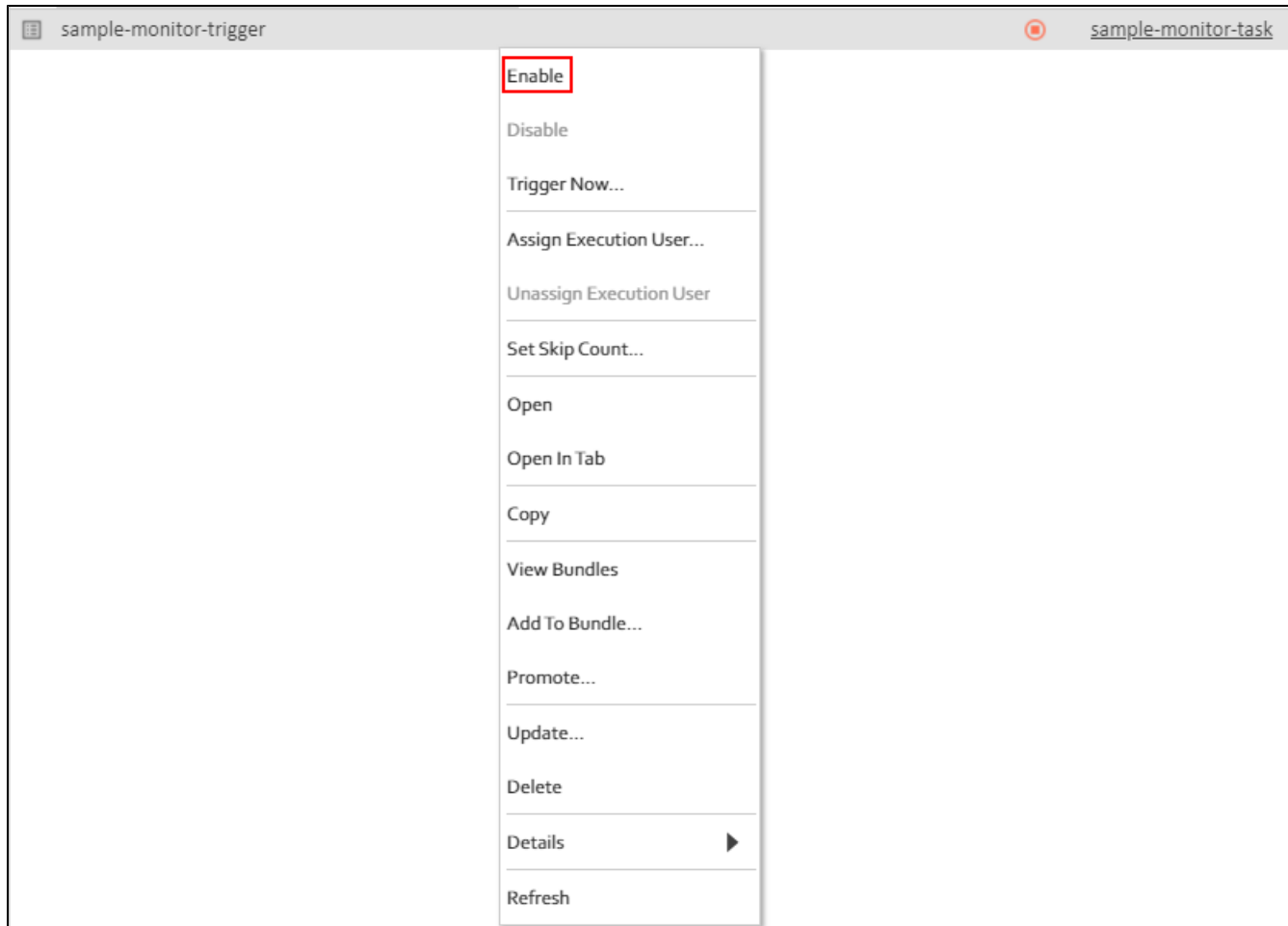


The trigger above will launch the **Sleep 0** task when the **sample-monitor-task's** specified criteria is matched.

Save the trigger.

## Step 7 - Run the Universal Monitor Trigger

We are finally ready to see the event functionality in action. Navigate to the **sample-monitor-trigger**, right-click on it, and select **Enable**:



Clicking **Enable** will launch the **sample-monitor-task** and **sample-publisher-task**. To see this, go to the **Instances** tab of the **sample-monitor-trigger**. You should see the following:

Name	Description	Universal MonitorTrigger	Variables	Instances	Notes	Versions
sample-monitor-trigger						
2 Task Instances						
Instance Name	Type	Status	Invoked By	Start Time	End Time	Updated
sample-publisher-task	Universal	Running	Trigger: sample-monitor-trigger	2022-03-07 13:02:35 -0500		2022-03-07 13:02:35 -0500
sample-monitor-task	Universal Monitor	Running	Trigger: sample-monitor-trigger	2022-03-07 13:02:35 -0500		2022-03-07 13:02:35 -0500

The **sample-publisher-task** is sending an event every 5 seconds with the filelisting. Since we haven't created the **test.txt** file in the specified directory, the **Sleep 0** task isn't triggered yet.

In your specified directory, create the **test.txt** file. After about 5 seconds, click the **Refresh** icon in the **Instances** tab of the **sample-monitor-trigger**, and you should see the following:

Universal Monitor Trigger						
Variables		Instances		Notes		Versions
3 Task Instances <span style="float: right;">Last 48 hours</span>						
Instance Name	Type	Status	Invoked By	Start Time	End Time	Updated
Sleep 0	Timer	Success	Trigger: sample-monitor-trigger	2022-03-07 13:07:55 -0500	2022-03-07 13:07:55 -0500	2022-03-07 13:07:55 -05
sample-publisher-task	Universal	Running	Trigger: sample-monitor-trigger	2022-03-07 13:02:35 -0500		2022-03-07 13:02:35 -05
sample-monitor-task	Universal Monitor	Running	Trigger: sample-monitor-trigger	2022-03-07 13:02:35 -0500		2022-03-07 13:02:35 -05

If you delete the **test.txt** file and create it again, the **Sleep 0** Timer task should be launched again by the trigger.

## Step 8 - Update the local template.json

Throughout the tutorial, we modified the Universal Template several times. Recall that inside `~/dev/extensions/sample-publisher/src/templates`, there is a **template.json** file. This file should correspond to the Universal Template in the Controller.

Right now, they are not both the same. To grab those changes, use the **pull** command as shown below:

### UIP VS Code Extension

```
> uip pull
```

UIP: Pull Shift + Alt + L recently used

```
@sbus44:~/dev/extensions/sample-publisher$ uip pull
The following files were updated:
- template.json
```

### Step Supplemental - CLI

```
@sbus44:~/dev/extensions/sample-publisher$ uip pull
The following files were updated:
- template.json
```

Now, both the local and Controller's version of the Universal Template are the same.

[< Previous](#) [Next >](#)


# Troubleshooting and Debugging

- [Log level](#)
- [Retrieve Output](#)
- [Agent Cache Directory](#)
- [Debugging Dynamic Choice Field](#)
  - [Console](#)
  - [Choice Commands under the hood](#)
  - [Logging](#)
- [Debugging Dynamic Commands](#)
  - [Console](#)
  - [Dynamic Commands under the hood](#)
  - [Logging](#)

## Log level


Universal Extensions support a configurable log level. The log level can be specified at template level:

**General**

Name \*  Extension  

Description

Variable Prefix \*

Icon   No file chosen PNG image (48 x 48 pixels)

Log Level

And overridden at task level:

**General**

Name \*  Version

Description

Member of Business Services

Resolve Name Immediately  Time Zone Preference

Hold on Start

Virtual Resource Priority  Hold Resources on Failure

**Log Level**

This allows Extension code to be written with logging statements only print if the log level is set to an appropriate level (e.g. Debug).

For example, adding the following code to an extension under **extension\_start()**:

**Logging messages**

```
# Display current log level.
level = logger.level
logger.critical("The log level is %s", level)
logger.critical("This is a critical log message.")
logger.error("This is an error log message.")
logger.warn("This is a warning log message.")
logger.info("This is an info log message.")
logger.debug("This is a debug log message.")

logger.info("state: ")
logger.info(state)
```

and setting the Log Level to Debug would produce the following output:

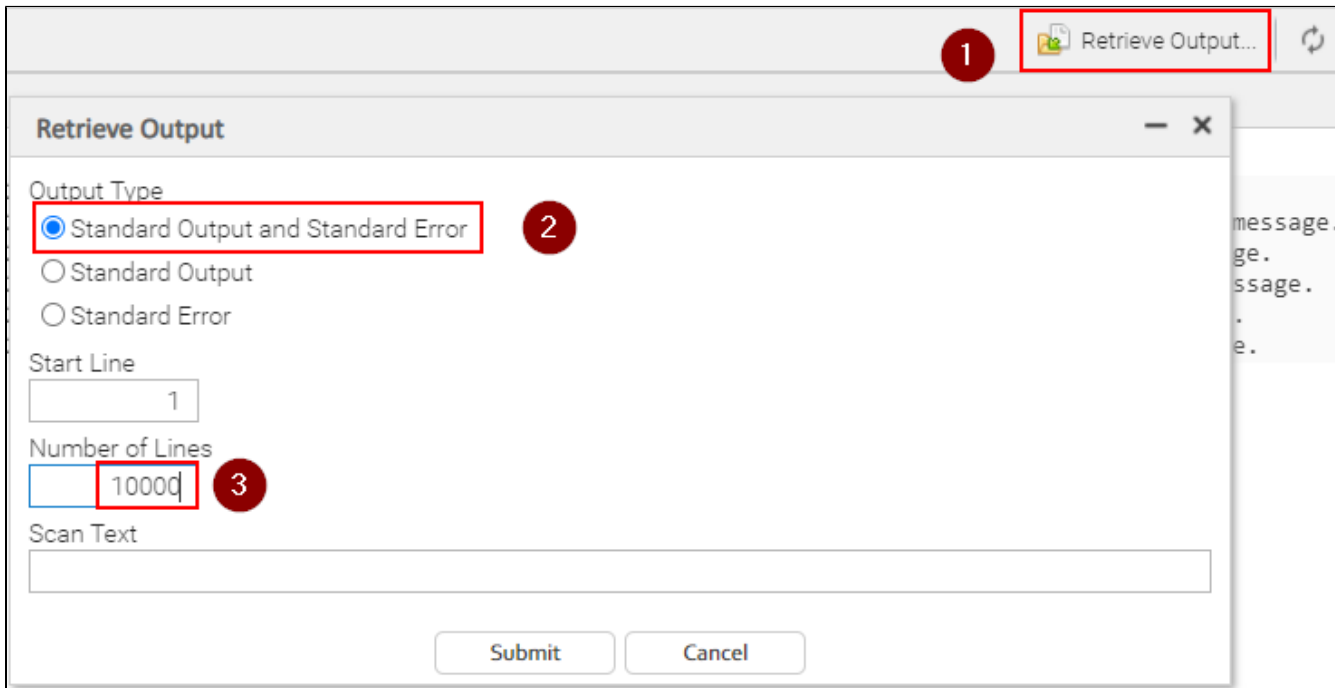
Type	Attempt	Output	Updated By	Updated
STDERR	1	2022-03-04 13:58:22,987 - 72040 MainThread - extension.py[161] CRITICAL: The log level is 10 2022-03-04 13:58:22,989 - 72040 MainThread - extension.py[162] CRITICAL: This is a critical log message. 2022-03-04 13:58:22,990 - 72040 MainThread - extension.py[163] ERROR: This is an error log message. 2022-03-04 13:58:22,990 - 72040 MainThread - extension.py[164] WARNING: This is a warning log message. 2022-03-04 13:58:22,990 - 72040 MainThread - extension.py[165] INFO: This is an info log message. 2022-03-04 13:58:22,990 - 72040 MainThread - extension.py[166] DEBUG: This is a debug log message.	ops.system	
STDOUT	1	[empty]	ops.system	
EXTENSION	1	Hello Extension!	ops.system	

As you can see, in addition to printing the message passed to the logging function, valuable context information is added automatically like timestamp, file name, and line number.

Adding Debug level statements throughout your code that can be easily turned on and off as needed will go a long way towards finding problems.

## Retrieve Output

If an Extension task completes with an unexpected result (and perhaps no output), perform a "Retrieve Output" and select "Standard Output and Standard Error". Be sure to set a sufficient line count to ensure you get back everything:



This will ensure you are seeing everything that was produced by the execution of the Extension task.

## Agent Cache Directory

As an alternative to retrieving the output from the Controller, you can go right to the cache directory on the target agent system. The standard out and standard error for all tasks are written to these directories.

Windows	C:\Program Files\Universal\UAGSrv\cache\
Unix	/opt/universal/uagsrv/cache/

The files have the format:

**<UUID>\_stdout**

**<UUID>\_stderr**

Where <UUID> is the UUID of the task instance. For example:

```
1618409011646336487EAG8TLHASQSQ7_stdout
1618409011646336487EAG8TLHASQSQ7_stderr
```

## Debugging Dynamic Choice Field

Dynamic Choice commands do not return stdout or stderr to the Controller. If the expected output does not show up in the Choice field drop-down, it may be difficult to understand what went wrong. The information provided here will help you understand how to find the cause of problems.

### Console

If a Dynamic Choice Command returns an error, the Controller will log the error message in its Console. This will provide limited information but, in some cases, it will be all that is needed to determine the cause of the problem. In other cases, it may be necessary to troubleshoot by reviewing files in the [cache directory](#).

### Choice Commands under the hood

When Choice Commands execute on the target agent system, an Extension instance is started in its own Worker process to execute the command (just like with task execution). A stdout and stderr file is created for the process under the agent's cache directory - again, just like with task execution.

If a Dynamic Command encounters an unexpected error, stack trace information will be written to stderr and, therefore, be available in a <UUID>\_stderr file in the cache directory. In the case of Dynamic Choice Commands, the UUID is the request ID of the message sent by the Controller to the target agent - not the UUID of the task from which it is issued.

### Logging

The same logging facility described above for task execution is available to Dynamic Choice Commands:

#### Logging

```
self.log.debug("This is a debug log message.")
```

While, the stderr file that contains the logged messages is not sent back to the Controller, it is available in the agent cache directory on the target agent system.

## Debugging Dynamic Commands

Dynamic commands do not return stdout or stderr to the Controller so, if the expected output does not show up in the task instance, it may be difficult to understand what went wrong. The information provided here will help you understand how to find the cause of problems.

## Console

If a Dynamic Command returns an error, the Controller will log the error message in its Console. This will provide limited information but, in some cases, it will be all that is needed to determine the cause of the problem. In other cases, it may be necessary to troubleshoot by reviewing files in the [cache directory](#).

## Dynamic Commands under the hood

When Dynamic Commands execute on the target agent system, an Extension instance is started in its own Worker process to execute the command (just like with task execution). A stdout and stderr file is created for the process under the agent's cache directory - again, just like with task execution.

If a Dynamic Command encounters an unexpected error, stack trace information will be written to stderr and, therefore, be available in a <UUID>\_stderr file in the cache directory. In the case of Dynamic Commands, the UUID is the request ID of the message sent by the Controller to the target agent - not the UUID of the task instance from which it is issued.

## Logging

The same logging facility described above for task execution is available to Dynamic Commands:

### Logging

```
logger.debug("This is a debug log message.")
```

While, the stderr file that contains the logged messages is not sent back to the Controller, it is available in the agent cache directory on the target agent system.

[< Previous](#)

# Universal Extension 1.0.0 API

- [Universal Extension Package](#)
  - [UniversalExtension class](#)
    - [Methods](#)
  - [ExtensionResult class](#)
  - [ExtensionLogger class](#)
  - [Universal Extension Decorator](#)
    - [Methods](#)

## Universal Extension Package

### UniversalExtension class

Base class for Stonebranch Universal Extension module implementations

#### Methods

`extension_start(fields)`

This method must be overridden by the custom Extension class that derives from the UniversalExtension class. It is called by UniversalExtension base class in response to a `JSS-LAUNCH` message sent from the Controller.

This is essentially the `main()` function for an Extension implementation and is the starting point for work that will be performed. The `fields` parameter passes in a dictionary of Extension instance fields that were defined in the Extension template.

<b>Input</b>	Parameters: <ul style="list-style-type: none"> <li>• <code>fields</code>: dict The <code>fields</code> parameter passes in a dictionary of implementation-dependent fields, which correspond to the Universal Template field names in the Controller's Template definition for the associated task.</li> </ul>
<b>Output</b>	ExtensionResult

`update_extension_status(fields)`

This method can be called at any time by the Extension instance. It is used to propagate state changes back to the associated extension instance in the Controller. Essentially, any/all output fields defined in the associated Extension Template can be updated using this method.

This method results in an `ESS-STATUS-UPDATE` message being sent from the Worker process to the UAG Extension Manager, followed by a `JSS-STATUS(JOB UPDATE)` message being sent from UAG Extension Manager to the Controller (via OMS server).

<b>Input</b>	Parameters:
--------------	-------------

	<ul style="list-style-type: none"> <li><i>fields</i>: dict The <i>fields</i> parameter expects a dictionary of output fields to be sent back to the controller for the associated Extension instance. Field names are implementation dependent and correlate with the Universal Template field names in the Controller's Template definition for the associated task.</li> </ul>
<b>Output</b>	None

## ExtensionResult class

class `ExtensionResult(**kwargs)`

The constructor for the `ExtensionResult` class takes different forms depending on the context. See the notes below.

### Extension Start

The form of the `ExtensionResult` constructor when instantiated in the context of *Extension Start*, is as follows:

```
ExtensionResult(rc=0, message='', output_fields=None, unv_output=None, **kwargs)
```

Parameter	Type	Default	Description
rc	<i>int, optional</i>	0	This parameter represents the return code of the task instance that initiated the <b>extension_start</b> operation. The value returned is implementation-defined and therefore left up to the Extension developer. The value can be used by the "return code processing" of the task instance in the Controller to determine if the Extension task instance is perceived as completing with <b>Success</b> or <b>Failed</b> .
message	<i>str, optional</i>	Empty string	This parameter specifies a short status string (error message or success message) to be sent to the "Status Description" field on the task instance form.
output_fields	<i>dict, optional</i>	None	Dictionary containing output fields. The parameter is a dictionary of output fields to be sent back to the controller for the associated Extension instance. Field names are implementation dependent and correlate with the Universal Template field names in the Controller's Template definition.
unv_output	<i>str, optional</i>	None	This parameter specifies the data that will be returned to the Controller for the command execution. It is interpreted as a UTF-8 encoded text object.  If the output parameter is True, the output_data will be persisted in the Controller as a record under table ops_exec_output, and appearing as Universal Command output type from the task instance Output tab.

### Choice Command

The form of the `ExtensionResult` constructor when instantiated in the context of *Choice Command*, is as follows:

```
ExtensionResult(rc=0, message='', values=None, **kwargs)
```

Parameter	Type	Default	Description
rc	<i>int, optional</i>	0	This parameter represents the return code of the 'choice' operation and determines whether the command is perceived as completing with success or failure by the Controller.  A value of 0 indicates success. All other values indicate an error condition and will be ignored by the Controller for form field population. However, the Controller will log the command response. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.
message	<i>str, optional</i>	Empty string	This parameter allows the extension to pass a completion message back to the Controller. The message will be logged by the Controller.
values	<i>list, optional</i>	Empty List	This parameter specifies a list of string values to be returned to the Controller and used to populate the associated dynamic choice field on the Extension task form.

al

### Dynamic Command

The form of the ExtensionResult constructor when instantiated in the context of *Dynamic Command*, is as follows:

```
ExtensionResult(rc=0, message='', output=False, output_data=None, output_name=None, **kwargs)
```

Parameter	Type	Default	Description
rc	<i>int, optional</i>	0	This parameter represents the return code of the Dynamic Command operation and determines whether the command is perceived as completing with success or failure by the Controller.  A value of 0 indicates success. All other values indicate an error condition and the command result will not be added to the Controller's Output tab. However, the Controller will log the command response. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.
message	<i>str, optional</i>	Empty string	The <b>message</b> parameter specifies a short status string (error message or success message) that will be logged by the Controller.
output	<i>bool, optional</i>	False	This parameter is a Boolean value that specifies if the command produced output that should be persisted and displayed under the task instance Output tab in the Controller task instance associated with the dynamic command invocation.  <b>The default value is False.</b>  This flag allows distinguishing between a command that does not produce output and a command that produces output but the output returned was empty.
output_data	<i>str, optional</i>	None	This parameter specifies the data that will be returned to the Controller for the command execution. It is interpreted as a UTF-8 encoded text object.  If the <b>output</b> attribute is <b>True</b> , the <b>output_data</b> will be persisted in the Controller as a record under table ops_exec_output, and appearing as <b>Universal Command</b> output type from the task instance <b>Output</b> tab.  <b>The default value is None.</b>
output_name	<i>str, optional</i>	None	This parameter is used to provide a custom name for the associated output data. The <b>output_name</b> (if provided) will be used in the presentation of the <b>output_data</b> by the Controller.

## ExtensionLogger class

```
classExtensionLogger(name)
```

Class for providing logging functionality for Universal Extensions. Do not instantiate this class directly. UniversalExtension instantiates this class via a call to the logging.getLogger API call.

## Universal Extension Decorator

### Methods

```
dynamic_choice_command(field_name)
```

Register a dynamic choice command.

Input	Parameters:
	<ul style="list-style-type: none"> <li><i>field_name</i>: str The field name.</li> </ul>

<b>Output</b>	ExtensionResult
---------------	-----------------

`dynamic_command(command_name)`

Register a dynamic command.

<b>Input</b>	Parameters: <ul style="list-style-type: none"><li>• <i>command_name</i>: str The name of the dynamic command.</li></ul>
<b>Output</b>	ExtensionResult

# Universal Extension 1.1.0 API

The Universal Extension 1.1.0 API was delivered with Universal Agent 7.1.0.0.

See the pages below for a description of the classes and methods available in the API.

Pages
<a href="#">UniversalExtension Class (1.1.0)</a>
<a href="#">ExtensionResult Class (1.1.0)</a>
<a href="#">ExtensionLogger Class</a>
<a href="#">Universal Extension Decorators (1.1.0)</a>

# UniversalExtension Class (1.1.0)

universal\_extension.universal\_extension API documentation

## Classes

class UniversalExtension  
Base class for Stonebranch Universal Extension module implementations

### Methods

`extension_cancel(self)`  
Implement in derived class.

---

`extension_start(self, fields)`  
Implement in derived class.

---

`update_extension_status(self, fields)`  
Propagate state changes back to the associated extension instance in the Controller.

- Can be called at any time by an Extension instance.
- Any/all output fields defined in the associated Extension Template can be updated using this method.

## Parameters

**fields**:dict

The fields parameter expects a dictionary of output fields to be sent back to the controller for the associated Extension instance. Field names are implementation dependent and correlate with the Universal Template field names in the Controller's Template definition for the associated task.

## Returns

None

## Examples

```
>>> my_ext = MyExt() # my_ext is an instance of a (derived) extension class called MyExt
>>> fields = {"foo": "bar"}
>>> my_ext.update_extension_status(fields)
```

---

# ExtensionResult Class (1.1.0)

universal\_extension.extension\_result API documentation

## Classes

```
class ExtensionResult (**kwargs)
```

Depending on the context of the initialization, different parameters need to be passed to the constructor. See the notes below.

Note

The form of the ExtensionResult constructor when instantiated in the context of **Dynamic Command**, is as follows:

### Constructor Signature

```
(rc = 0, message = "", output = False, output_data = None, output_name = None, call_frame = None, **kwargs)
```

## Parameters

**rc** :int, optional

This parameter represents the return code of the Dynamic Command operation and determines whether the Extension task instance is perceived as completing as Success or Failed by the Controller.

A value of 0 indicates success. All other values indicate an error condition and result in a status of Failed in the Controller. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.

The ExtensionResult class sets a default value of 0.

**message** :str, optional

This parameter allows the extension to pass a completion message back to the Controller. If rc is set to 0, the message will be considered informational and will be logged by the Controller. If rc is non-zero, the message will be considered an error message and will be displayed to the user on the task instance form.

The default value is an empty string.

**output** :bool, optional

This parameter specifies if the command produced output that should be persisted and displayed under the task instance Output tab in the Controller task instance associated with the dynamic command invocation.

The default value is False.

**output\_data** :str, optional

This parameter specifies the data that will be returned to the Controller for the command execution. It is interpreted as a UTF-8 encoded text object.

If the output attribute is True, the output\_data will be persisted in the Controller as a record under table ops\_exec\_output, and appearing as Universal Command output type from the task instance Output tab.

The default value is None.

**output\_name** :str, optional

This attribute is used to provide a custom name for the associated output data. The output\_name (if provided) will be used in the presentation of the output\_data by the Controller.

The default value is None.

**call\_frame** :frame, optional

Frame of a function call where previous activity of interest occurred.

The default value is None.

Note

The form of the ExtensionResult constructor when instantiated in the context of **Choice Command**, is as follows:

### Constructor Signature

*(rc = 0, message = "", values = None, call\_frame = None, \*\*kwargs)*

## Parameters

**rc** :int, optional

This parameter represents the return code of the 'choice' operation and determines whether the command is perceived as completing with success or failure by the Controller. A value of 0 indicates success. All other values indicate an error condition and will be ignored by the Controller for form field population. However, the Controller will log the command response. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.

The ExtensionResult class sets a default value of 0 to the rc attribute.

**message** :str, optional

This parameter allows the extension to pass a completion message back to the Controller. The message will be logged by the Controller.

The default value is an empty string.

**values** :list, optional

This parameter specifies a list of string values to be returned to the Controller and used to populate the associated dynamic choice field on the Extension task form.

The default value is an empty list.

**call\_frame** :frame, optional

frame of the function call where previous activity of interest occurred, by default None

Note

The form of the ExtensionResult constructor when instantiated in the context of **Extension Start**, is as follows:

### Constructor Signature

*(rc = 0, message = "", output\_fields = None, unv\_output = None, call\_frame = None, \*\*kwargs)*

## Parameters

**rc** :int, optional

This parameter represents the return code of the extension\_start operation. The associated Universal Task instance in the Controller can use this value to determine the completion status of the Extension instance. The value is implementation defined and therefore left up to the extension developer.

The ExtensionResult class sets a default value of 0.

**message** :str, optional

This parameter allows the extension to pass a completion message back to the Controller. If rc is set to 0, the message will be considered informational and will be logged by the Controller. If rc is non-zero, the message will be considered an error message and will be displayed to the user on the task instance form.

The default value is an empty string.

**output\_fields** :dict, optional

Dictionary containing output fields.

The default None.

**unv\_output** :str, optional

The value for this parameter is considered the payload of the task execution. It appears on the task instance Output tab as Universal output type.

The default value is None.

**call\_frame** :frame, optional

Frame of a function call where previous activity of interest occurred.

The default value is None.

---

# ExtensionLogger Class

universal\_extension.logger API documentation

## Global variables

`logger`  
global `logger` instance that can be imported in other modules.

## Examples

```
>>> # This example demonstrates how to import the global logger in other Python modules.
>>> # Assume that logger is imported below in test.py that resides in the same folder as extension.py
>>> from universal_extension import logger
>>> logger.info('sample info message from test.py')
>>> logger.warning('sample warning message from test.py')
>>> logger.critical('sample critical message from test.py')
```

## Classes

`class ExtensionLogger (name)`

Class for providing logging functionality for Universal Extensions. Do not instantiate this class directly. UniversalExtension instantiates this class via a call to the `logging.getLogger` API call. See the example above for instructions on obtaining the `logger` instance.

The following log levels are officially supported:

- CRITICAL
- ERROR
- WARNING
- INFO
- DEBUG

Note

In addition to the levels listed above, there is another level, **TRACE**, which is used by the Universal Extension internal API. It is **NOT** available for use in developing Extensions.

---

# Universal Extension Decorators (1.1.0)

[universal\\_extension.deco.command API documentation](#)

## Command

`dynamic_command(command_name)`  
Register a dynamic command.

### Parameters

`command_name` :str  
the command name

### Returns

None

[universal\\_extension.deco.choice API documentation](#)

## Choice

`dynamic_choice_command(field_name)`  
Register a dynamic choice command.

### Parameters

`field_name` :str  
the field name

### Returns

None

# Universal Extension 1.2.0 API

The Universal Extension 1.2.0 API was delivered with Universal Agent 7.2.0.0.

See the pages below for a description of the classes and modules available in the API.

Pages
<a href="#">UniversalExtension Class</a>
<a href="#">ExtensionResult Class</a>
<a href="#">Universal Extension Decorators</a>
<a href="#">Logging Module</a>
<a href="#">Event Module</a>
<a href="#">UI Module</a>

# UniversalExtension Class

universal\_extension.universal\_extension API documentation

## Classes

class UniversalExtension  
Base class for Stonebranch Universal Extension module implementations

### Instance variables

uip  
can be used to access the following properties of an Extension task instance:

- `task_variables`: dict
- `is_triggered`: bool
- `trigger_id`: str
- `instance_id`: str
- `monitor_id`: str

### Examples

```
>>> ops_task_id = self.uip.task_variables['ops_task_id']
>>> is_triggered = self.uip.is_triggered
>>> trigger_id = self.uip.trigger_id
>>> instance_id = self.uip.instance_id
>>> monitor_id = self.uip.monitor_id
```

---

## Methods

`extension_start(self, fields)`  
Serves as the starting point for work that will be performed for a task instance. It must be implemented in the derived class.

### Parameters

`fields`: dict  
populated with field values from the associated task instance launched in the Controller

### Returns

`ExtensionResult`  
once the work is done, an instance of `ExtensionResult` must be returned. See the `ExtensionResult` documentation for a full list of parameters that can be passed to the class constructor

---

`extension_cancel(self)`  
Optional method that allows the Extension to do any cleanup work before terminating. To use it, implement in the derived class.

### Parameters

None

## Returns

None

---

`update_extension_status(self, fields)`

Propagate state changes back to the associated extension instance in the Controller.

- Can be called at any time by an Extension instance.
- Any/all output fields defined in the associated Extension Template can be updated using this method.

## Parameters

**fields**:dict

The fields parameter expects a dictionary of output fields to be sent back to the controller for the associated Extension instance. Field names are implementation dependent and correlate with the Universal Template field names in the Controller's Template definition for the associated task.

## Returns

None

## Examples

```
>>> fields = {"foo": "bar"}
>>> self.update_extension_status(fields)
```

---

# ExtensionResult Class

universal\_extension.extension\_result API documentation

## Classes

```
class ExtensionResult (**kwargs)
```

Depending on the context of the initialization, different parameters need to be passed to the constructor. See the notes below.

Note

The form of the ExtensionResult constructor when instantiated in the context of **Dynamic Command**, is as follows:

### Constructor Signature

```
(rc = 0, message = "", output = False, output_data = None, output_name = None, call_frame = None, **kwargs)
```

## Parameters

**rc** :int, optional

This parameter represents the return code of the Dynamic Command operation and determines whether the Extension task instance is perceived as completing as Success or Failed by the Controller.

A value of 0 indicates success. All other values indicate an error condition and result in a status of Failed in the Controller. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.

The ExtensionResult class sets a default value of 0.

**message** :str, optional

This parameter allows the extension to pass a completion message back to the Controller. If rc is set to 0, the message will be considered informational and will be logged by the Controller. If rc is non-zero, the message will be considered an error message and will be displayed to the user on the task instance form.

The default value is an empty string.

**output** :bool, optional

This parameter specifies if the command produced output that should be persisted and displayed under the task instance Output tab in the Controller task instance associated with the dynamic command invocation.

The default value is False.

**output\_data** :str, optional

This parameter specifies the data that will be returned to the Controller for the command execution. It is interpreted as a UTF-8 encoded text object.

If the output attribute is True, the output\_data will be persisted in the Controller as a record under table ops\_exec\_output, and appearing as Universal Command output type from the task instance Output tab.

The default value is None.

**output\_name** :str, optional

This attribute is used to provide a custom name for the associated output data. The output\_name (if provided) will be used in the presentation of the output\_data by the Controller.

The default value is None.

## Example

```
>>> @dynamic_command('reset_environment')
>>> def reset_environment(self, fields):
>>>     # Implementation omitted
```

```
>>> return ExtensionResult(
...     message = "Message: Hello from dynamic command 'reset_environment'",
...     output = True,
...     output_data = 'The environment has been reset.',
...     output_name = 'DYNAMIC_OUTPUT'
... )
```

**Note**

The form of the ExtensionResult constructor when instantiated in the context of **Choice Command**, is as follows:

**Constructor Signature**

*(rc = 0, message = "", values = None, call\_frame = None, \*\*kwargs)*

**Parameters**

**rc** :int, optional

This parameter represents the return code of the 'choice' operation and determines whether the command is perceived as completing with success or failure by the Controller. A value of 0 indicates success. All other values indicate an error condition and will be ignored by the Controller for form field population. However, the Controller will log the command response. The non-zero value used to represent the error condition is implementation defined and therefore left up to the extension developer.

The ExtensionResult class sets a default value of 0 to the rc attribute.

**message** :str, optional

This parameter allows the extension to pass a completion message back to the Controller. The message will be logged by the Controller.

The default value is an empty string.

**values** :list, optional

This parameter specifies a list of string values to be returned to the Controller and used to populate the associated dynamic choice field on the Extension task form.

The default value is an empty list.

**Example**

```
>>> @dynamic_choice_command("primary_choice_field")
>>> def primary_choice_command(self, fields):
>>>     # Implementation omitted
>>>     return ExtensionResult(
...         rc = 0,
...         message = "Values for choice field: 'primary_choice_field'",
...         values = ["Start", "Pause", "Stop", "Build", "Destroy"]
...     )
```

**Note**

The form of the ExtensionResult constructor when instantiated in the context of **Extension Start**, is as follows:

**Constructor Signature**

*(rc = 0, message = "", output\_fields = None, unv\_output = None, call\_frame = None, \*\*kwargs)*

**Parameters**

**rc** :int, optional

This parameter represents the return code of the extension\_start operation. The associated Universal Task instance in the Controller can use this value to determine the completion status of the Extension instance. The value is implementation defined and therefore left up to the extension developer.

The ExtensionResult class sets a default value of 0.

**message** :str, optional

This parameter allows the extension to pass a completion message back to the Controller. If rc is set to 0, the message will be considered informational and will be logged by the Controller. If rc is non-zero, the message will be considered an error message and will be displayed to the user on the task instance form.

The default value is an empty string.

**output\_fields** :dict, optional

Dictionary containing output fields.

The default None.

**unv\_output** :str, optional

The value for this parameter is considered the payload of the task execution. It appears on the task instance Output tab as Universal output type.

The default value is None.

## Example

```
>>> def extension_start(self, fields):
>>>     # Implementation omitted
>>>     return ExtensionResult(
...         unv_output = "extension_start() finished successfully",
...         rc = 0
...     )
```

---

# Universal Extension Decorators

[universal\\_extension.deco.choice](#) API documentation

## Choice

`dynamic_choice_command(field_name)`

Decorator that can be used to register a dynamic choice command.

### Parameters

**field\_name** :str

the value specified must match the field name of the associated Dynamic Choice field in the Controller's Universal Template

### Returns

None

### Examples

```
>>> @dynamic_choice_command("primary_choice_field")
>>> def primary_choice_command(self, fields):
>>>     # Implementation omitted
```

[universal\\_extension.deco.command](#) API documentation

## Command

`dynamic_command(command_name)`

Decorator that can be used to register a dynamic command.

### Parameters

**command\_name** :str

the value specified must match the command name of the associated Dynamic Command defined in the Controller's Universal Template

### Returns

None

### Examples

```
>>> @dynamic_command("reset_environment")
>>> def reset_environment(self, fields):
>>>     # Implementation omitted
```

# Logging Module

`universal_extension.logger` API documentation

## Global variables

`logger`

global `logger` instance that can be imported in other modules.

The following log levels are officially supported:

- **CRITICAL**
- **ERROR**
- **WARNING**
- **INFO**
- **DEBUG**

Note

In addition to the levels listed above, there is another level, **TRACE**, which is used by the internal `UniversalExtension` API. It is **NOT** available for use in developing Extensions.

## Examples

```
>>> # This example demonstrates how to import the global logger in other Python modules.
>>> # Assume that logger is imported below in test.py that resides in the same folder as extension.py
>>> from universal_extension import logger
>>> logger.info('sample info message from test.py')
```

# Event Module

universal\_extension.event API documentation

## Functions

`publish(name, attributes, time_to_live=None)`

Publish a Universal Event to the associated Universal Controller.

- Can be called at any time by an Extension instance.
- Universal Events are optionally defined in the Universal Extension Template as part of the Universal Extension definition.

## Parameters

**name** :str

The name of a target event defined in the Controller.

**attributes** :dict

the attributes of the event

**time\_to\_live** :int, optional

maximum time (in minutes) the event can live. If no value is specified, the default value from the Universal Event Template will be used.

## Returns

None

## Examples

```
>>> from universal_extension import event
>>> event_attributes = {}
>>> event_attributes["attribute_1"] = "value_1"
>>> event_attributes["attribute_2"] = "value_2"
>>> event_attributes["attribute_3"] = "value_3"
>>> event.publish("my_event", event_attributes, 20)
```

# UI Module

universal\_extension.ui API documentation

## Functions

`update_output_fields(fields)`

Propagate state changes back to the associated extension instance in the Controller.

- Can be called at any time by an Extension instance.
- Any/all output fields defined in the associated Extension Template can be updated using this method.

## Parameters

**fields**:dict

The fields parameter expects a dictionary of output fields to be sent back to the controller for the associated Extension instance. Field names are implementation dependent and correlate with the Universal Template field names in the Controller's Template definition for the associated task.

## Returns

None

Note

This method is equivalent to the `update_extension_status` method from the UniversalExtension Class with the added functionality that it can be used from external modules.

## Examples

```
>>> from universal_extension import ui
>>> fields = {"foo": "bar"}
>>> ui.update_output_fields(fields)
```