Developing IBM UrbanCode Deploy Plug-ins
Introductions

- Your organization
- Your role
- Your background and experience
  - Software development
  - IBM UrbanCode products
  - IBM Rational products
- Course expectations
Intended Audience

- This course focuses on the process and tools used in creating UrbanCode Deploy plug-ins. The course assumes a basic knowledge of UrbanCode Deploy and does not teach those concepts.
- Basic knowledge of Eclipse is assumed.
- Questions are welcomed. Questions that require technical answers may be deferred to insure all class materials are covered.
Course Goals

- At the end of this course, you should be able to:
  - Understand the IBM plug-in support model
  - Utilize the IBM UrbanCode Plug-in DevKit to build and deploy new plug-ins
  - Understand the anatomy of a plug-in
  - Build deploy plug-ins that wrap existing scripts, makes CLI calls, and make REST calls
  - Build source config plug-ins
  - Incorporate resources and properties in your plug-ins
  - Understand how to upgrade a plug-in
  - Know when and how to use post-processing scripts
Agenda

- Introduction to IBM UrbanCode Deploy Plug-ins
- The UrbanCode Deploy Plug-in DevKit
  - Lab 1 – Hello World plug-in
- A closer look at info.xml and plugin.xml
  - Lab 2 – Create a new plug-in that encapsulates a script
- Upgrading a plug-in – upgrade.xml
  - Lab 3 – Upgrade a plug-in
- Helper Groovy Scripts and calling REST API methods
  - Lab 4 – Create a new plug-in that makes a REST call
- Plug-ins and Properties
- Source Config Plug-ins
  - Lab 5 – Source Config plug-in
- Miscellaneous Topics - Post-processing scripts, logging, auto-discovery
INTRODUCTION TO IBM URBANCODE DEPLOY PLUG-INS
What is a Plug-in?

- A plug-in provides a customizable integration point between UrbanCode Deploy and 3rd party solutions involved in artifact storage, application deployment, or SDLC processes.
# UrbanCode Deploy Integration Strategies

<table>
<thead>
<tr>
<th>Type of Interaction</th>
<th>Integration Target Systems</th>
<th>Examples of Integration Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept inputs of application and component assets.</td>
<td>- Build / CI Servers</td>
<td>- Build Forge, Jenkins, Hudson</td>
</tr>
<tr>
<td></td>
<td>- SCM Systems</td>
<td>- RTC, CC, AccuRev, CVS, PVCS</td>
</tr>
<tr>
<td></td>
<td>- Repositories</td>
<td>- Maven, RAM, Subversion</td>
</tr>
<tr>
<td></td>
<td>- Build Tooling</td>
<td>- Make, MSBuild, Ant, NAnt</td>
</tr>
<tr>
<td>Prepare and provision environments to deploy applications and components to.</td>
<td>- Private Cloud</td>
<td>- PureApp, SCO, VMWare,</td>
</tr>
<tr>
<td></td>
<td>- Public Cloud</td>
<td>- Amazon EC2</td>
</tr>
<tr>
<td>Deploy applications and components to heterogeneous middleware platforms.</td>
<td>- Application Servers</td>
<td>- WAS, WebLogic, JBoss, IIS/.NET</td>
</tr>
<tr>
<td></td>
<td>- Load Balancers</td>
<td>- Citrix NetScaler, F5</td>
</tr>
<tr>
<td></td>
<td>- Portals</td>
<td>- WebSphere Portal, WebLogic Portal</td>
</tr>
<tr>
<td></td>
<td>- BPM Servers and ESBs</td>
<td>- IBM BPM, WMB, Microsoft BizTalk</td>
</tr>
<tr>
<td></td>
<td>- Messaging Infrastructure</td>
<td>- WebSphere MQ</td>
</tr>
<tr>
<td></td>
<td>- Commerce Servers</td>
<td>- WebSphere Commerce</td>
</tr>
<tr>
<td></td>
<td>- Packaged Applications</td>
<td>- SAP, Oracle Applications</td>
</tr>
<tr>
<td></td>
<td>- Appliances</td>
<td>- IBM DataPower</td>
</tr>
<tr>
<td>Integrate with a host of other systems and services for a range of SDLC tasks.</td>
<td>- Work Item / Issues Tracking</td>
<td>- Rally, JIRA, RTC</td>
</tr>
<tr>
<td></td>
<td>- Installers</td>
<td>- MSI, RPM</td>
</tr>
<tr>
<td></td>
<td>- Quality Management / Testing</td>
<td>- RQM, GH, HP QC, Selenium</td>
</tr>
<tr>
<td></td>
<td>- Scripting</td>
<td>- Groovy, Shell, PowerShell</td>
</tr>
<tr>
<td></td>
<td>- System Utilities</td>
<td>- Windows &amp; Linux System Tools, 7zip</td>
</tr>
</tbody>
</table>
Plug-in Types

- **Tier 1 plug-ins**
  - Provided with the UrbanCode Deploy product
  - Some are installed automatically and others can be added based on need
  - Fully supported

- **Tier 2 plug-ins**
  - Provided by partners. Must engage partner to obtain plug-in and receive support

- **Tier 3 plug-ins**
  - Openly developed plug-ins by anyone (IBM DevOps Services/GitHub)
  - Forum support only
Plug-ins Website

- All info and updates available here:
Why Plug-ins?

- Encapsulate best practices
  - New customers have a desire to simply call their existing scripts. While this may provide a quick win, it is seldom a scalable solution.

- Provide re-usable steps that can be included in many different processes
  - Plug-in steps are the building blocks that can then be used to create numerous process templates

- Upgrading a plug-in upgrades all process steps currently in use
Plug-in Details

- Plug-ins are packaged as simple zip files
  - Metadata files capture plug-in data
  - Scripts that provide the plug-in step implementation
  - Any supporting libraries or utility code
- UrbanCode Deploy refers to its plug-in architecture as the “Air” plug-in architecture
- Two types of plug-ins:
  - Automation – manipulate component versions typically by deploying them or integrate with other SDLC tools/processes
  - Source – integrate with external systems to import artifacts and create component versions.
Plug-ins at Runtime

- Each process step is executed by an UrbanCode Deploy agent. The plug-in that provides the step is downloaded to and cached by the agent.
  - The first time an agent executes a step from a plug-in, you may notice additional execution time. This is due to the downloading of the plug-in.
  - Plug-in versions are checked each time a step is executed by an agent. If a new version of the plug-in step is being used, the agent will pull the new version of the plug-in.
The Anatomy of an Air Plug-in

- Metadata
  - info.xml – basic information about the plug-in (author, release notes, version number)
  - plugin.xml – detailed information about each plug-in step (properties and run command), plug-in name and unique identifier, plug-in type
  - upgrade.xml – provide information about a new plug-in version to allow the upgrade process to be successful

- Payload – the code that will “do the work” of each plug-in step including any helper libraries
**Groovy recommended …**

- Steps can run any script you write. Why does UrbanCode Deploy usually use Groovy?
  - Groovy is on every agent
  - Groovy is cross platform
  - Groovy is very good at XML
  - Groovy is fun to learn, concise and effective

- Other good choices:
  - Perl, Ruby, VBScript, Python
  - Etc, etc, etc
  - Must insure runtime is available or include it in the plug-in
  - Use existing plug-ins for examples and best practices
THE URBANCODE DEPLOY PLUG-IN DEVELOPMENT KIT
The Plug-in Development Kit

- The plug-in development kit provides a set of Eclipse extensions to aid your development of UrbanCode Deploy plug-ins.
  - New Project wizard with project templates for typical plug-in strategies
  - Custom editors for each of the plug-in metadata files
  - Consistent project structure
  - Build.xml file to use Ant to produce your plug-in zip file
  - Alternative export wizard to create zip file
  - Export wizard to automatically deploy your plug-in to your UrbanCode Deploy server
  - Upgrade action to update your plug-in metadata to the next release version
  - Help Content
New Project Wizard

- Creates a new Groovy-based plug-in project
New Project Wizard

- Provides templates for common plug-in strategies
Custom Editors – info.xml

- info.xml editor
Custom Editors – plugin.xml

- plugin.xml editor
Custom Editors – upgrade.xml

- upgrade.xml editor
Project Structure

- Each project is created with the same structure
  - Src folder for developing Groovy helper scripts
  - Doc folder for any plug-in documentation
  - Imports folder for include component templates and processes in your plug-in
  - Lib folder for any needed libraries
  - License folder to capturing licensing text
  - Plugin folder for the metadata xml files
  - Releases folder holds the build output
  - Scripts folder for the Groovy implementation scripts for each plug-in step
  - Build.xml file for Ant builds
Export Wizards

- Export UCD Plug-in – standard zip extractor
- Export UCD Plug-in to UCD Server – automatically deploy your zip file
Automatically Export to UCD Server

- Right-mouse-click on zip file. File ➔ Export...
- Provide administrative user credentials

UrbanCode Deploy Server Details

Provide the UrbanCode Deploy Server URL and an administrator username and password

UrbanCode Deploy Server URL

Username

Password
Automatically Export to UCD Server

- Automatically create Application, Components, and Environments to test plug-in
Upgrade Plug-in Action

- Right-mouse-click on upgrade.xml
- Action updates all 3 metadata files to new version number
- Adds new migrate-command entry to upgrade.xml for each plug-in step
- Adds new release-notes entry to info.xml
- Manual effort to finish
Help Content

Plug-ins

IBM® UrbanCode Deploy plug-ins provide tools for creating component processes and integrations.

Plug-ins provide processing and integration functions. There are two types of plug-ins:

- Source-type plug-ins integrate with external systems to import artifacts and create component versions.
- Automation-type plug-ins are used to manipulate component versions, typically by deploying them.

Plug-ins consist of distinct processes called steps. Each step consists of a number of properties, a command that runs the step, and post-processing instructions. (Post-processing typically ensures that expected results occur.) Step properties can serve a variety of purposes, from providing input to the command, to supplying some or all the actual command itself. Automation-type plug-in properties can be set at design time in the process editor, or at run time in the user interface. Source-type plug-in properties are typically defined when a component is created.

A number of plug-ins come with the product and many others are available that can be downloaded and installed. For information about installing plug-ins, see Installing plug-ins. For information about creating plug-ins, see Creating plug-ins.

Automation-type plug-ins

You use automation plug-ins to deploy components or otherwise manage them. By combining automation plug-in steps in the process editor, you can create fully automated deployment processes. When you create a process, you drag steps onto the design area and define the properties as you go. Property values can be specified when you define the process or at run-time. The process flow is defined by drawing connections between steps. The following illustration, shows a series of automation-type plug-in steps and the connections between them.

Process: Deploy application

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 of 2</td>
<td>Deploy the web application</td>
</tr>
</tbody>
</table>

Start

Fetch the artifacts

Download Artifacts (v. 16)
Lab 1 – Hello World Plug-in

- Lab 1 in Workbook – 20 minutes
- In this lab you will create your first plug-in project and explore the UrbanCode Deploy Development Kit.
  - Create a new example plug-in project
  - Edit each of the metadata files
  - Modify the groovy script to output “hello world”
  - Build the plug-in
  - Deploy the plug-in
  - Run the plug-in
A CLOSER LOOK AT INFO.XML AND PLUGIN.XML
3 metadata files

- info.xml – basic background information about the plug-in
- plugin.xml – details of each plug-in step
- upgrade.xml – used by the UrbanCode Deploy server plug-in upgrade process

- Version number in each of these files. The version number needs to be consistent across all 3 files
Author name is required, other author info is optional

- Author name filled in from new project wizard
info.xml

- Integration type is required

```xml
<!--
**integration type IS Required**

The integration type section identifies where the plugin fits into the process excepted types are SCM, Build, Artifact, Automation, Deploy

For example:

The plugin pulls Artifacts from an artifact repository during deployment
the Type would be "Artifact"
or
The plugin deploys to a middleware server
the Type would be "Deploy"
-->

<integration type="Deploy"/>
```
Tool-description section is required

The tool-description section is used to summarize the software the plugin was created to integrate with.

For example:
Apache Ant is a software tool to automate software processes during the build or deployment of an application. Ant uses an [propitiatory XML file](https://ant.apache.org/manual/) to define build and/or deployment steps (referred to as targets by ant). The Ant executable is called to execute the targets in the build.xml.

```xml
<tool-description>
  Put your tool description here
</tool-description>
```
info.xml

- Release-version and release-note sections are required

```xml
<release-version>1</release-version>

<release-notes>
  <!--
  **release-note IS required**

  The plugin-version name must match the plugin version found in the plugin.xml file
  The release-note should document the changes, improvements, or bug fixes introduced
  with the new plugin version
  The release-note may contain text or reference a file in the plugin zip package
  
  For example your release note could be:
  Support for was added for version 10.0 for software X
  or
  releaseNotes/release1.html (where release1.html contains html formatted version information)
  -->

  <release-note plugin-version="1">
  
  </release-note>
  
  </release-notes>
</pluginInfo>
```
info.xml

- Source, licenses, related-info, and meta-html sections are not required but are there for you to add additional background data.
- Except for the release-version and release-note sections, this file stays fairly static throughout the lifetime of the plug-in.
**plugin.xml**

- Defines all of the properties and their user interface widgets that are used to capture the property values for each plug-in step.
- Defines the command string that will be used to execute the step.
- Defines the default post-processing script.
plugin.xml

- **Header**
  - The identifier is the unique identifier for the plug-in. No two loaded plug-ins can have the same identifier
  - The name of the plug-in is what is displayed in the GUI
  - The displayed version number
  - The description is what is shown in the GUI
  - The tag defines where in the step tree hierarchy the steps in the plug-in can be found.
The plugin.xml editor from the DevKit can help create new step properties.
plugin.xml

- Results in this step properties dialog
plugin.xml

- Property types
  - textBox
  - textAreaBox
  - secureBox
  - checkBox
  - selectBox
Select boxes have an additional set of values that define the choices

```xml
<property name="SelectBox" required="true">
    <property-ui type="selectBox" default-value="One" description="This is">
        <value label="One">One</value>
        <value label="Two">Two</value>
        <value label="Three">Three</value>
    </property-ui>
</property>
```
Lab 2 – Encapsulate a Script

- Lab 2 in Workbook – 20 minutes
- In this lab you will create another plug-in project. This one will encapsulate a script.
  - Create a simple script to encapsulate
  - Create a new plug-in project using a template
  - Build the plug-in
  - Deploy the plug-in
  - Run the plug-in
UPGRADING A PLUG-IN
PLUGIN.XML
The plug-in upgrade process

- The upgrade.xml file gives instructions to the server plug-in upgrade process as to how to upgrade a step
  - A `<migrate to-version="?"/>` element is required for each version beyond version 1. It encapsulates a series of `<migrate-command>` elements for each step that is to be migrated

- When a new version of a plug-in is loaded into the UrbanCode Deploy server, the following happens:
  - Every step of every existing process is examined
  - If the step is one of the plug-in steps being upgraded, the upgrade.xml file tells the upgrade process what to do.

- For large installations you may want to upgrade plug-ins during off hours
**upgrade.xml**

- The following conditions require the following elements to be created in the upgrade.xml file
  - Update a script
    - if there are no changes to the plugin.xml file (no metadata changes) but you simply want to update one of the scripts that are run that implement the logic of a step, no change is required to any of the plug-in configuration files. You can simply reload the plug-in zip file and the scripts will be replaced.
    - **NOTE:** It is recommended that you still create a new version of plug-in in this case to insure you have a record of the change
upgrade.xml

- Propagating a step to the next version
  - any time you upgrade a plug-in, you must include a
    `<migrate-command>` element for every step that you want
    to include in the new version.

```xml
<move to-version="2">
  <migrate-command name="Step Name" />
</migrate>
```

- Delete a step
  - if you want to delete a step, then simply don't include a
    `<migrate-command>` element for that step. The step will
    be marked as deleted in a process. The process must be
    fixed before it will run successfully.

- Adding a step
  - If your new plug-in version adds a step, you don't need to
    do anything in the upgrade.xml file. There are no
    processes that are using this step so there is nothing to
    upgrade.
upgrade.xml

- Rename a step
  - the `<migrate-command>` element provides a way to rename a step by including an "old" attribute.

```xml
  <migrate to-version="2">
    <migrate-command name="Step Name" old="Old Name"/>
  </migrate>
```

- Adding a step property
  - if you are adding a new property that does not have a default value, then obviously your script will have to take that into account. In this situation, you don't need to do anything special to the upgrade.xml file as long as you have a `<migrate-command>` element for the property's step. All existing process steps will get the new property but since there is no default value there is no need to migrate any existing properties. If the new property is required, you will be asked to enter a value when the process is run.
upgrade.xml

- If you are adding a new property that does have a default value, required or not, then you should add a `<migrate-properties>` element as follows. This will ensure that the properties of any existing use of the step will get migrated and the default value of the property will get assigned to the new property. If you don't include the `<migrate-properties>` element, the default value will not be applied to the property as no property migration will occur. This will be a problem if your script assumes a default value.

```xml
<migrate-command name="Step Name" old="Old Name">
  <migrate-properties>
    <migrate-property name="NewPropertyName" default="default value"/>
  </migrate-properties>
</migrate-command>
```

- Deleting a step property
  - nothing needs to be done in the upgrade.xml. The process of upgrading the step will remove the property as long as you have a `<migrate-command>` element to migrate the step.
upgrade.xml

- Renaming a step property
  - A capability similar to renaming a step exists to rename a step property using the `<migrate-properties>` element.

```xml
  <migrate-command name="Step Name" old="Old Name">
    <migrate-properties>
      <migrate-property name="NewPropertyName" old="OldPropertyName"/>
    </migrate-properties>
  </migrate-command>
```

- Changing a step property's default value
  - The `<migrate-property>` element is also used to change its default value.

```xml
  <migrate-command name="Step Name" old="Old Name">
    <migrate-properties>
      <migrate-property name="NewPropertyName" default="new default value"/>
    </migrate-properties>
  </migrate-command>
```

- You can combine the previous two concepts to both rename a property and provide a new default value.

```xml
  <migrate-command name="Step Name" old="Old Name">
    <migrate-properties>
      <migrate-property name="NewPropertyName" old="OldPropertyName" default="new default value"/>
    </migrate-properties>
  </migrate-command>
```
Upgrading a plug-in

- Upgrading a plug-in should be planned. Knowing that the plug-in upgrade process could potentially alter all occurrences of the plug-in steps in all processes, you should only upgrade plug-ins in your production instance(s) of UrbanCode Deploy after they have thoroughly been tested and proven in a non-production setting.
Lab 3 – Upgrade a Plug-in

- Lab 3 in Workbook – 20 minutes
- In this lab you will create another plug-in project. This one will upgrade the plug-in created in the previous lab.
  - Update the script to accept a parameter
  - Add a text box property to the plug-in step
  - Add a parameter to the plug-in step command to pass the property value
  - Upgrade the plug-in metadata
  - Build the plug-in
  - Run the plug-in
HELPER GROOVY SCRIPTS AND CALLING REST API METHODS
Groovy Helpers

- The example plug-in (created in Lab 1) includes a series of Groovy helper scripts that can be included in any plug-in.
- Each file has some detailed comments on how to use the class
Advanced Plug-ins

- Many plug-ins require some extensive programming that involve utilizing 3\textsuperscript{rd} part APIs or libraries. These can be done in Groovy, but you may want a higher level programming language that provide supporting libraries (i.e. Java).
- You may need to include an extensive list of jar files to provide the necessary functionality you need.
- You are responsible for adhering to the license agreements for the various libraries or APIs you may download from the internet and use.
Calling Rest APIs

- Calling REST APIs is a good example of an advanced implementation
  - Most likely have to deal with authentication and potentially SSL certificates
    - Would take advantage of the various HttpClient Apache libraries
  - Most likely have to deal with JSON payloads both outgoing and incoming
    - Would take advantage of many of the JSON serializer libraries out there
  - Might have to utilize some API functionality from another software product/application
    - Acquire any API capabilities from the vendor
Lab 4 – Call a REST API

- Lab 4 in Workbook – 20 minutes
- In this lab you will create another plug-in project. This one will call a REST API.
  - Create a new plug-in project
  - Build the plug-in
  - Run the plug-in
PLUG-INS AND PROPERTIES
Properties and Plug-ins

- As you construct your plug-in steps and determine the information that needs to be gathered from the user via step properties, think ahead to how that data should be provide
  - Properties that are unique, change frequently, and don’t follow a pattern can be entered by the user
    - Set a default value if at all possible
  - Properties that are closely tied to an environment, a component, a resource, an agent, etc. should be abstracted away to be an UrbanCode Deploy property
    - Your default value then should be a suggested property name and scope (i.e. ${p:resource/propName})
Properties and Plug-ins

Properties Available to Component Steps

- System
- Application Process
- Environment
- Component Process
- Resource

Previous Steps
- ${p:stepName/propName}

Current Step
- {p:propertyName}
- ${p:resource/propName}
- ${p:resource/role/propName}
- ${p:component/propName}
- ${p:environment/propName}
- ${p:application/propName}
- ${p:system/propName}
Properties and Plug-ins

- In your scripts, access current step properties via:
  ${p:property}

- Creating new properties for use in later steps
  - You can create a new property for use in a later step.
    For example, within a step include the following Groovy code that gets the current date in a specified format
    ```groovy
    import java.text.SimpleDateFormat
    def today = new Date()
    def formattedDate = new SimpleDateFormat("ddMMyyyy, Ka").format(today)
    outProps.put("date", formattedDate)
    ```
  - Access the date property in a subsequent step via:
    ${p:<step-name>/date}
Properties and Plug-ins

- Security sensitive properties
  - Username/passwords should be captured in a secureBox property type
  - If there is no default value, then the user must enter them in for each instance of the step
  - If you provide a default value, then you should provide a suggested property that would store the value
    ${p:resource/Dbpassword}
  - Make the property secure if you don’t want the value to be displayed
Properties and Plug-ins

- Also you can make the step property hidden

- Another option is to execute some type of script to access username/passwords
  - Take advantage of setting properties in this case if username/passwords is needed in subsequent steps
SOURCE CONFIG PLUG-INS
Source Config Plug-ins

- Source-type plug-ins are defined the same way that automation-type plug-ins are defined except for some modifications to the plugin.xml file
  - The <server:plugin-type>Source</server:plugin-type> tag identifies the type of plug-in as 'Source'
  - A source plug-in has one and only one step, which must be named “Import Version”
  - Source plug-ins have two property groups. Component type properties & Import type properties.
    - Component-type properties are filled in when the user specifies the “Create New Component” action
    - Import-type properties are filled in when the user specifies the “Import Component” action.
Source Config Plug-ins

- **Plugin.xml**
  - Plugin-type is **Source**
  - Only one step-type – **Import Version**

```xml

<header>
  <identifier id="my.sc" name="My Source Config" version="1"/>
  <description>sample source config plugin</description>
  <tag>SCM/Name</tag>

  <!-- The below tag identifies this plugin as a source plugin -->
  <server:plugin-type>Source</server:plugin-type>

</header>

<!-- A source plug-in has one step, which must be named Import Version -->

<step-type name="Import Version">
  <description>Creates a new component version and imports artifacts</description>
  <properties>
  </properties>
  <post-processing><![CDATA[
    if (properties.get("exitCode") != 0) {
      properties.put(new java.lang.String("Status"), new java.lang.String("Failure"));
    }
  ]]>}</post-processing>

</step-type>
```

IBM Corporation ©2013
Source Config Plug-ins

- Property groups

```xml
<!-- As shown in the following code fragment, source plug-ins have two property groups. -->
<!-- Component type properties are configured with the Create New Component window. -->
<!-- Import type properties are configured with the Import Component window. -->

<server:property-group name="FileSystemComponentProperties" type="Component">
   <server:description>Description</server:description>
   <server:property name="property-name" required="true">
      <server:property-ui description="property-description" label="property-label" type="textBox"/>
   </server:property>
</server:property-group>

<!-- If left empty or omitted then no dialog will appear -->

<server:property-group name="FileSystemImportProperties" type="Import">
   <server:description>Description</server:description>
   <server:property name="property-name" required="true">
      <server:property-ui description="property-description" label="property-label" type="textBox"/>
   </server:property>
</server:property-group>
```
Source Config Plug-ins

- Are run by an agent (keep this in mind)
- Based on input properties (label, stream, branch, change set, etc.) use the source config API to fetch the artifacts
- Use the UrbanCode Deploy CLI or REST API to create a component version and add files to it

```java
gitClient.checkoutRevisionOrTag(gitVersionOrTag, repoDir);
System.out.println("Uploading artifact files.");
versionClient.addVersionFiles(componentName, versionId, repoDir, ",",
    includes, excludes, executePermissions, true, charset, extensions);
```
Source Config Plug-ins

- CLI is a command-line interface that provides access to the IBM® UrbanCode Deploy server REST APIs. It can be used to find or set properties, and run numerous functions
  - You can download the CLI 'udclient.zip' from the IBM UrbanCode Deploy server by clicking Help > Tools and then clicking IBM UrbanCode Deploy Client
- UDRestClient.jar – utilized in all of the UrbanCode plug-ins to access many of the UrbanCode Deploy REST API methods.
  - Feel free to use in your source plug-ins as well
Lab 5 – Create a Source Config Plug-in

- Lab 5 in Workbook – 20 minutes
- In this lab you will create another plug-in project. This one will implement a source config integration.
  - Create a new plug-in project
  - Examine the plugin.xml file
POST-PROCESSING SCRIPTS AND LOGGING
Post-Processing Scripts

- When a plug-in step's `<command>` element finishes processing, the step's mandatory `<post-processing>` element runs.
- The plugin.xml file specifies the default post-processing script. However, this may be overridden by an individual step instance.
- Must be implemented in JavaScript!
- Post-processing scripts have access to a java.util.Properties variable called `properties`. 
Post-Processing Scripts

- The `properties` variable has several special properties: `exitCode` contains the process exit code, and `Status` contains the step's status. A `Status` value of `Success` means that the step completed successfully.
- Post-processing scripts must set the `Status` property to specify the status of the script.
  - `properties.put("Status","Success");`
Post-Processing Scripts

- Another available variable, `scanner`, can scan the step's output log (scanning occurs on the agent) and run commands based on the results. `scanner` has several public methods:
  - `register(String regex, function call)` registers a function to be called when the regular expression is matched.
  - `addLOI(Integer lineNumber)` adds a line to the lines of interest list, which are highlighted in the Log Viewer; implicitly called whenever scanner matches a line.
  - `getLinesOfInterest()` returns a java.util.List of lines of interest; can be used to remove lines.
  - `scan()` scans the log. Use after all regular expressions are registered.
Post-Processing Scripts

- Users can store post-processing scripts on the server to reuse them in component processes and in generic processes to override the default.
- Click **Settings > Post Processing Scripts** and then click **Create New Script**.
Post-Processing Scripts

- **Sample**

```javascript
properties.put("Status", "Success");

// Evaluate the built-in exitCode property, which indicates the exit code
// of the script called by the plug-in step. Typically, if the value of
// the exitCode property is non-zero, the plug-in step failed.
//
if (properties.get("exitCode") != 0) {
    properties.put("Status", "Failure");
}
```

- **Scanner**

```javascript
// Register a scanner to search for the text "error at line" in the log. The first
// argument is a regular expression. The second argument, an inline function, is
// invoked once for every line in the log output that matches the pattern. The
// "lineNumber" variable contains the line number where the match occurred, and the
// "line" variable is the full text of the line.

scanner.register("(?i)ERROR at line", function(lineNumber, line) {

    // In this case, we build up an "Error" property which
    // contains the text of all errors that are found. We find every
    // line starting with "error at line" and add it to this list.
    var errors = properties.get("Error");
    if (errors == null) {
        errors = new java.util.ArrayList();
    }
    errors.add(line);

    // If a line starting with "error at line" is found, the step has
    // failed, so we set the special "Status" property to "Failure",
    // indicating to the UrbanCode Deploy server that the step should
    // be marked as a failure.
    properties.put("Status", "Failure");
});
```
Post-Processing Scripts

- Scanner

```javascript
//
// Multiple searches can be registered with the scanner. We add a
// second search to look for some interesting text to set as an output
// property.
//
// For example, if there is a line "The value is BLUE", then we end up
// with an output property, "Value", with a value of "BLUE".
//
scanner.register("The value is", function(lineNumber, line) {
    var value = line.replace("The value is ", ");
    properties.put("Value", value);
});
scanner.scan();
//
// Convert the collected list of error strings into a single string and
// set that as an output property.
//
var errors = properties.get("Error");
if (errors == null) {
    errors = new java.util.ArrayList();
}
properties.put("Error", errors.toString());
```
Logging

- Process logs show the output log for each plug-in step
- The more information you output to `sysout`, the more information you have to debug your plug-in
New Plug-in Capabilities

- New way to reference properties (6.0.1)
  - Assume propName is a property that has no value
    - ${p:propName} will resolve to ${p:propName}
  - Now you can specify a property as ${p?:<prop>}
  - Again, assume propName has no value
    - ${p?:propName} will resolve to “” (empty string)
  - If a property has a value, ${p?:propName} and ${p:propName} will both resolve to the property’s value

- In many cases, servers may have resources already defined (i.e. app server instances, apps, cells, nodes, etc.) It is up to the user to build the necessary resource tree and create properties and values specific to the server.
Auto-Discovery and Auto-Configure

- Auto-discovery and auto-configure plug-in step types have been create to help capture this information automatically (6.0)

- Three basic capabilities:
  - An **automatic discovery** step is transparently run whenever an Agent Resource is added to the resource tree. The discovery process is generally used to discover information about the agent machine and either update its properties or automatically generate and configure sub-resources.
  - An **automatic configuration** step is an end user initiated step that runs a configuration process against a resource to configure it for deployments.
  - A **resource role** is basically a property sheet. Typically they are attached to discovered resources listing information that is relevant or useful at deployment time. The contents of the resource role may be populated by Discovery, Configuration or manually.
Auto-Discovery and Auto-Configure

- The first use of these capabilities is in the Application Deployment for WebSphere plug-in. See this plug-in for a good example.

- Auto-discovery steps are initiated when an agent resource is added to the resource tree. Every auto-discovery step from every plug-in is run by the agent at this time.

- Discovery steps are intended to make initial discoveries of technologies installed on the agent machine. Configure steps typically require additional information (usernames/passwords) and may be run numerous times based on changing environments.

- Typical discovery uses:
  - Application server discovery (Tomcat, WAS, etc.)
  - Check for the existence of a testing tool and tag agent as able to run tests
  - Determine server hostname and query CMDB for configuration info

- Discovery and configure can be done in a single discovery step, but resource intensive processes or those processing requiring admin knowledge or any processes you would want to run numerous times should be done in configuration steps.
Auto-Discovery

- Typical auto-discovery steps
  - Search for the existence of some technology (Tomcat, WAS, IIS, etc.) based on typical or default installation paths.
  - Update properties of the Agent Resource and/or create a sub-resource that identifies the discovered technology.
  - Assign a role to the sub-resource with corresponding properties that capture discovered information and others that need to be filled in for configuration.
Auto-Configure

- Typical auto-configure steps
  - Utilizing the discovered property values as well as property values supplied by the user, interrogate the server for additional technology information (i.e. WAS nodes and servers)
  - Create additional sub-resources and configure them with roles and property sets to capture the acquired information.
Auto-Discovery and Auto-Configure

- When debugging your auto-discovery and auto-configuration steps, the agent sends back log information to the server. You can find the *stdout* from the step at this location:

  `<server install dir>/logs/autoDiscovery/`
  `<server install dir>/logs/autoConfigure/`
Plugin.xml

- Additional elements are provided in the step-type definition to indicate auto-discover or auto-configure:
  - Auto-discover
    <server:type>AUTO_DISCOVERY</server:type>
  - Auto-configure (the role indicates the type of resource that you can initiate auto-configure from)
    <server:role>WebSphereCell</server:role>
    <server:type>AUTO_CONFIGURE</server:type>
Resource Groups in Plugin.xml

- Resource groups are used to define the property sets for each resource role.
  - Type attribute indicates the purpose of the property-group
  - Special-type indicates that it may be used for auto-discovery
Plug-in Imports

- You can provide UrbanCode Deploy objects as part of your plug-in (applications, components, component templates, and generic processes).
- An imports directory in your plug-in zip file will hold the json files that represent the element(s).
  - imports/applications/
  - imports/components/
  - imports/componenttemplates/
  - imports/processes/
- If a process in any of your imports uses steps in the plug-in you are developing, you must load a version of the plug-in to an UrbanCode Deploy server first, create the process using the plug-in steps, and then export the element as a json file. Then include the json in your plug-in.
- The WebSphere plug-in utilizes this capability.
Thank You

www.ibm.com/software/rational