HANDS-ON ACTIVITIES

DEVELOPMENT LIFECYCLE AND DEPLOYMENT

FEBRUARY, 2018

DEPLOYMENT STRATEGY

Given a scenario, compare, contrast and recommend the components and tools of a successful deployment strategy.

USE CASE

A company, Universal Containers (UIC), works with different partners and has few admin resources to take care of the day-to-day administration tasks. As a result, UC would like to find a way to automate a set of validations when migrating changes between environments in order to enforce some of their internal rules. An example of this would be the usage of particular naming conventions.

DETAILED REQUIREMENTS

1. When a package contains at least one trigger, the administrator wants to be notified and the deployment must be stopped to dedicate some time to review it/them.

2. When a package contains Apex classes or Visualforce pages, the package can be deployed only if the proper naming convention is used.

   Visualforce page names should have the following format:

   VFPXXX_<PageName> where XXX represent 3 digits.

   Apex class names should follow the following format:

   VFCXXX<ClassName> or VFCXXX<ClassName> where XXX represents 3 digits.

3. When a package contains objects with custom fields, a verification should be performed to validate if the fields contain descriptions, and the deployment should be stopped if a field description is missing.

PREREQUISITE SETUP STEPS

ALERT:

In order to complete this build material, you must have an established proficiency in XML (Extensible Markup Language) and XSLT (Extensible Stylesheet Language Transformations).

1. ANT must be installed and properly configured on your machine.

   - ☐ You can find more information using the following link: http://ant.apache.org/manual/install.html
   - ☐ To verify the installation, you can execute the "ant –version" command, and you should see something like:

     Apache Ant™ version 1.9.4 compiled on April 29, 2014

2. Extract the **ConfigurationManagement_SupportingFiles.zip** file.

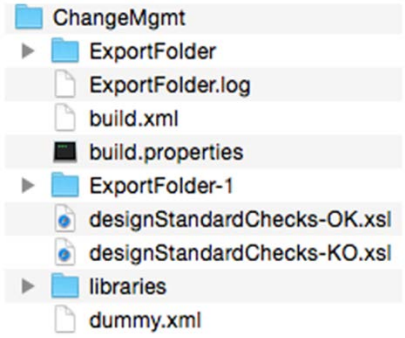
3. The different ANT commands used in this document will have to be executed from the root folder containing the build.xml file.

4. Download and copy the Force.com Migration Tool file (AKA **ant-salesforce.com.jar**) in the Libraries folder, which contains the **saxon9he.jar** file.

5. Modify the **build.properties** file to reflect your credentials to the target sandbox.

Download the supporting files to complete this build here - SUPPORTING FILES
NOTE – this link does not have a preview – you will have to download the files.

The ConfigurationManagement_SupportingFiles.zip file contains the following:



The ExportFolder.log file is generated by the build.xml script itself and may be not present when you check the content of your folder.

The 2 folders, ExportFolder and ExportFolder-1, represent 2 distinct packages. The first contains metadata not compatible with the different rules implemented, and the second contains metadata compatible with the expected rules.

Considerations

- What deployment tool can be considered?
- Do you need to consider a fully custom solution?
- Can we use existing ANT tasks to fulfill some parts of the requirement?
- Can we find an option without real development?

SOLUTION

BEST SOLUTION OVERVIEW

Different options are possible to fulfill the requirements, but in order to provide a solution that is flexible, easy to use, and easily customizable, the recommended solution is to leverage the options that do not involve code.

Requirement #1: Trigger Check

The first requirement is to be able to validate that a particular package, which is a particular folder, is either not present or does not contain files having a particular extension (here, “trigger”).

The first part of the requirement is to make sure that no trigger file is present in the usual Triggers folder available in an ANT package.

The ANT framework provides tasks that can help in this context. The first task used is called **ResourceCount** and allows us to count the number of files contained in a particular folder and matching a particular format. The second task is the **Fail** task that allows a build to be stopped (ANT process) under certain circumstances, such as if the number of files corresponding to triggers is greater than a specific threshold (here, 0). You can find more information on the ResourceCount and **Fail** tasks in the following links:

1.https://ant.apache.org/manual/Tasks/resourcecount.html
2.https://ant.apache.org/manual/Tasks/fail.html

With these 2 tasks, we can answer the original requirement. Let's start with the most important, which is to be able to make sure that there is no trigger in the usual triggers folder.

```
1  <target name="checkTriggersInPackage" depends="trigger.check" if="triggers">
2    <property name="trigger.dir" value="${export.folder}/triggers" />
3    <resourcecount property="count">
4              <fileset id="matches" dir="${trigger.dir}">
5                      <patternset id="files">
6                              <include name="**/*.trigger"/>
7                      </patternset>
8              </fileset>
9    </resourcecount>
10   <fail message="'${count}' triggers have been foumd in '${trigger.dir}'">
11             <condition>
12                     <resourcecount when="greater" count="0" refid="matches" />
13             </condition>
14   </fail>
15 </target>
```

The logic, in bold, counts–thanks to the **ResourceCount** task–the number of files having the **trigger** extension in the **trigger**s folder.

The part, handled by the **Fail** task, compares the number of files counted thanks to the preceding task and compares it with the threshold 0 and stops the build if the number of triggers is greater than 0.

The last part of the first requirement is to make sure that the logic/validation presented above is executed only if there is a **triggers** folder present in the analyzed package. In the previous code snippet, we can see that the **target** tag has 2 attributes, which are the **depends** attribute as well as the **if** attribute, which execute the target only if the **triggers** property is set up by the **trigger.check** target called by the depends attribute. The **trigger.check** target is provided below and creates the trigger's property if there is a Triggers folder present in the folder dedicated to the analyzed package.

```
1. <target name="trigger.check">
2.   <condition property="triggers">
3.             <and>
4.                       <available file="${export.folder}/triggers" type="dir" />
5.             </and>
6.   </condition>
7. </target>
```

The following text shows the output of the target calling the **checkTriggersInPackage** target on the **ExportFolder** folder, representing a package containing triggers.

```
$ ant TriggerPresenceValidation-Fail
Buildfile: /Users/xxx/Downloads/ChangeMgmt/build.xml

TriggerPresenceValidation-Fail:
    [echo] Validates if the package present in the 'ExportFolder' folder contains some
triggers

trigger.check:

checkTriggersInPackage:

BUILD FAILED
/Users/xxx/Downloads/ChangeMgmt/build.xml:182: The following error occurred while
executing this line:
/Users/xxx/Downloads/ChangeMgmt/build.xml:169: '1' triggers have been foumd in
'ExportFolder/triggers'

Total time: 0 seconds
```

The next text shows the output of the target calling the checkTriggersInPackage target on the ExportFolder-1 folder, representing a package containing no triggers.

```
$ ant TriggerPresenceValidation-Success
Buildfile: /Users/xxx/Downloads/ChangeMgmt/build.xml

TriggerPresenceValidation-Success:
    [echo] Validates if the package present in the 'ExportFolder-1' folder contains
some triggers

trigger.check:

checkTriggersInPackage:

BUILD SUCCESSFUL
Total time: 0 seconds
```

Requirement #2: Naming Convention

Fileset and Selectors can be used to fulfill the requirement around the validation of the naming convention. Indeed, Selectors are a mechanism whereby the files that make up a <fileset> can be selected based on criteria other than file name as provided by the <include> and <exclude> tags. These can be used to validate the naming convention thanks to the regex attribute.

You can find more information on the Selectors:
https://ant.apache.org/manual/Types/selectors.html

```
1.  <target name="checkClassesPackage">
2.    <property name="class.dir" value="ExportFolder/classes" />
3.    <resourcecount property="count">
4.            <fileset id="matchesGlobal" dir="${class.dir}">
5.                    <patternset id="files">
6.                            <include name="**/*.cls"/>
7.                    </patternset>
8.            </fileset>
9.    </resourcecount>
10.   <resourcecount property="countNC">
11.           <fileset id="matches" dir="${class.dir}">
12.                   <filename regex="(?:VFC|AP)[0-9][0-9][0-9]_.*.cls"/>
13.           </fileset>
```

```
14.   </resourcecount>
15.   <fail message="Found '${count}' classe(s) which do(es) not follow the naming
      convention in '${class.dir}'">
16.             <condition>
17.                       <resourcecount when="less" count="${count}"
      refid="matches" />
18.             </condition>
19.   </fail>
20.   <echo message="Naming convention validation for classes successful"/>
21.</target>
```

ALERT:

The rest of the previous snippet is very similar to the one used to fulfill Requirement #1. The only difference is in bold and counts the number of files matching the expected naming convention.

The following snippet shows the checkClassesPackage-Fail target that calls the checkClassesPackage target on the ExportFolder folder, representing a package containing an Apex class which does not follow the naming convention.

```
1. <target name="checkClassesPackage-Fail">
2.   <propertyreset name="export.folder" value="ExportFolder"/>
3.   <echo message="Validates if the package present in the '${export.folder}'
     folder contains some classes whose name follows the naming convention (in this
     case the classes do not follow the naming convention)"/>
4.   <antcall target="checkClassesPackage">
5.           <param name="class.dir" value="${export.folder}/classes"/>
6.   </antcall>
7. </target>
```

The following text shows the output of the preceding target.

```
$ ant checkClassesPackage-Fail
Buildfile: /Users/xxx/Downloads/ChangeMgmt/build.xml

checkClassesPackage-Fail:
    [echo] Validates if the package present in the 'ExportFolder' folder contains some
classes whose name follows the naming convention (in this case the classes do not
follow the naming convention)

checkClassesPackage:

BUILD FAILED
/Users/xxx/Downloads/ChangeMgmt/build.xml:240: The following error occurred while
executing this line:
/Users/xxx/Downloads/ChangeMgmt/build.xml:228: Found '1' classe(s) which do(es)
not follow the naming convention in 'ExportFolder/classes'

Total time: 0 seconds
```

The following snippet shows the checkClassesPackage-Success target that calls the checkClassesPackage target on the ExportFolder-1 folder, representing a package containing an Apex class that follows the naming convention:

```
1. <target name="checkClassesPackage-Success">
2.    <propertyreset name="export.folder" value="ExportFolder-1"/>
3.    <echo message="Validates if the package present in the '${export.folder}'
      folder contains some classes whose name follows the naming convention (in this
      case the classes follow the naming convention)"/>
4.    <antcall target="checkClassesPackage">
5.                <param name="class.dir" value="${export.folder}/classes"/>
6.    </antcall>
7. </target>
```

The following text shows the output of the preceding target.

```
$ ant checkClassesPackage-Success
Buildfile: /Users/xxx/Downloads/ChangeMgmt/build.xml

checkClassesPackage-Success:
    [echo] Validates if the package present in the 'ExportFolder-1' folder contains
some classes whose name follows the naming convention (in this case the classes
follow the naming convention)

checkClassesPackage:
    [echo] Naming convention validation for classes successful

BUILD SUCCESSFUL
Total time: 0 seconds
```

Requirement #3: Design Standard

The object's definitions, as many other metadata files accessible thanks to the metadata API, are in the XML format. ANT easily leverages XSLT to offer a way to support the requirement.

The idea here is pretty straightforward and can be described in the following steps:

1. Create a collection of XML documents corresponding to the different objects available in the Objects folder.
2. Loop through the collections of objects.

3. Loop through the fields and check if the field contains a description. If there is no description, create a row for this field which will be added to a CSV file generated through the XSLT transformation.
4. After the generation of the CSV file, the build script will check the size of the CSV file and will stop the build if the size if greater than 0.

The files designStandardChecks-KO.xsl and designStandardChecks-OK.xsl are 2 XSL stylesheets that differ only by the root folder specifying the package analyzed

The code snippet below creates a collection of XML documents corresponding to the definitions of the custom objects present in the ExportFolder containing 1 object definition with fields having no description:

```
1. <xsl:template match="/">
        a. <xsl:for-each select="collection(iri-to-
           uri('ExportFolder/objects/?select=*.object;recurse=yes'))">
              i. <xsl:apply-templates mode="inFile" select=".">
                                    a. <xsl:with-param name="folder">
2. <xsl:value-of select="tokenize(document-uri(.), '/')[last()-1]"/>
3. </xsl:with-param>
                                    a. <xsl:with-param name="filename">
4. <xsl:value-of select="tokenize(document-uri(.), '/')[last()]"/>
5. </xsl:with-param>
              i. </xsl:apply-templates>
        b. </xsl:for-each>
6. </xsl:template>ion with fields having no description:
```

The following code snippet loops through every field and checks the presence of the description attribute, and generates a row for the CSV file if there is no description.

```
1. <xsl:template match="doc:CustomObject" mode="inFile">
2.    <xsl:param name="folder" />
3.    <xsl:param name="filename" />
4.    <xsl:param name="objectlabel" select="doc:label" />
5.    <xsl:param name="curr-label" select="substring-before($filename,'.')"/>
6.
7.    <xsl:for-each select="doc:fields">
8.          <!-- Check if there is no description -->
9.          <xsl:if test="not(doc:description)">
10.                <xsl:value-of select="substring-before($filename,'.')"/>
11.<xsl:text>,</xsl:text>
```

```
12.                         <xsl:value-of select="$objectlabel"/>
13.<xsl:text>,</xsl:text>
14.                         <xsl:value-of select="doc:fullName"/>
15.<xsl:text>,</xsl:text>
16.                         <xsl:value-of select="doc:type"/>
17.                         <xsl:text>&#xa;</xsl:text>
18.                 </xsl:if>
19.   </xsl:for-each>
20.</xsl:template>
```

The code snippet below shows the designStandardCheck target which has a review property allowing you to use one of the 2 stylesheets presented above which, behind the scene, selects a particular package.

```
1.  <target name="designStandardCheck" description="Validate that the fields have
    a description attribute">
2.  <property name="review" value="designStandardChecks-KO" />
3.              <property name="myclasspath" refid="saxonpath"/>
4.
5.  <xslt in="dummy.xml" out="${review}.csv" processor
    ="org.apache.tools.ant.taskdefs.optional.TraXLiaison"
6.          style="${review}.xsl" force="true" classpathref="saxonpath">
7.              <factory name="net.sf.saxon.TransformerFactoryImpl"/>
8.              </xslt>
9.
10.  <!-- sets the count property : if the size of the file generated by the XSLT task is 0,
     we consider that there is no missing description
11.  (when a description is missing for 1 field, a row is created in the CSV file) -->
12.  <resourcecount property="count">
13.             <fileset id="matchesGlobal" dir=".">
14.                     <patternset id="files">
15.                             <include
     name="${review}.csv"/>
16.                     </patternset>
17.                     <size value="0" when="more"/>
18.             </fileset>
19.  </resourcecount>
20.
21.  <fail message="1 or more fields have no description. More information can be
     found in the ${review}.csv file ">
22.             <condition>
23.                     <!--Check if the count property of the resourcecount task is
     greater than 0-->
24.                     <resourcecount when="more" count="0"
     refid="matchesGlobal" />
25.             </condition>
26.  </fail>
27.  <echo message="All the fields have a description attribute"/>
28.</target>
```

The following output corresponds to the result of the call to the checkDesignStandard-Fail target that calls the XSLT transformation defined in the designStandardChecks-KO.xsl file.

```
$ ant checkDesignStandard-Fail
Buildfile: /Users/xxx/Downloads/ChangeMgmt/build.xml

checkDesignStandard-Fail:
    [echo] Validates if the package present in the 'ExportFolder' folder contains some
pages whose name follows the naming convention (in this case the pages follow the
naming convention)

designStandardCheck:
    [xslt] Processing /Users/xxx/Downloads/ChangeMgmt/dummy.xml to
/Users/xxx/Downloads/ChangeMgmt/designStandardChecks-KO.csv
    [xslt] Loading stylesheet
/Users/xxx/Downloads/ChangeMgmt/designStandardChecks-KO.xsl

BUILD FAILED
/Users/xxx/Downloads/ChangeMgmt/build.xml:342: The following error occurred while
executing this line:
/Users/xxx/Downloads/ChangeMgmt/build.xml:330: 1 or more fields have no
description. More information can be found in the designStandardChecks-KO.csv file

Total time: 1 second
```

The content of the CSV file generated is the following, where we can find one row per field having no description:

```
Warehouse__c,Warehouse,City__c,Text
Warehouse__c,Warehouse,Location__c,Location
Warehouse__c,Warehouse,Phone__c,Phone
Warehouse__c,Warehouse,Street_Address__c,Text
```

Each of the 2 stylesheets generates a CSV: designStandardChecks-OK.csv and designStandardChecks-KO.csv.

COMPLETE EXAMPLE

A target called "deploy-with-check" has been created, which attempts to deploy the package only if the different validations are successful.

1. <target name="deploy-with-check" depends="INIT,TO,cond2,checkPackage,deployPackage" />

The following code snippet shows that the checkPackage target that is called just before the deployPackage target in the depends attribute of the deploy-with-check target above calls the different targets covering the 3 different validations covered in the previous sections.

```
1  <target name="checkPackage" >
2  <propertyreset name="export.folder" value="ExportFolder-1"/>
3  <antcall target="checkTriggersInPackage">
4  <param name="trigger.dir" value="${export.folder}/triggers"/>
5  </antcall>
6  <antcall target="checkClassesPackage">
7  <param name="class.dir" value="${export.folder}/classes"/>
8  </antcall>
9  <antcall target="checkPagesPackage">
10 <param name="page.dir" value="${export.folder}/pages"/>
11 </antcall>
12 <antcall target="designStandardCheck">
13 <param name="review" value="designStandardChecks-OK"/>
14 </antcall>
15 </target>
```

The following output shows the result of the execution of the deployPackage target, and we can see that the different validations are done.

```
$ ant deploy-with-check
Buildfile: /Users/xxx/Downloads/ChangeMgmt/build.xml

INIT:
    [input] skipping input as property sf.checkOnly has already been set.
     [echo] sf.checkOnly ==> true
    [input] skipping input as property sf.runAllTest has already been set.

TO:
    [input] skipping input as property sf.SURL2 has already been set.
    [input] skipping input as property sf.username2 has already been set.
    [input] skipping input as property sf.pwd2 has already been set.
    [input] skipping input as property sf.token2 has already been set.

check-cond2:

cond-if2:

cond-else2:

cond2:

checkPackage:

trigger.check:

checkTriggersInPackage:
checkClassesPackage:
     [echo] Naming convention validation for classes successful

checkPagesPackage:
     [echo] Naming convention validation for pages successful

designStandardCheck:
     [xslt] Processing /Users/xxx/Downloads/ChangeMgmt/dummy.xml to
/Users/xxx/Downloads/ChangeMgmt/designStandardChecks-OK.csv
     [xslt] Loading stylesheet /Users/xxx/Downloads/ChangeMgmt/designStandardChecks-OK.xsl
     [echo] All the fields have a description attribute

deployPackage:
     [echo] DEPLOY MetaData to xxx@dev.com
     [echo] serverurl2 => https://login.salesforce.com
   [deploy] Request for a deploy submitted successfully.
   [deploy] Request ID for the current deploy task: 0AfG0000007a5kiKAA
   [deploy] Waiting for server to finish processing the request...
   [deploy] Request Status: Pending
   [deploy] Request Status: InProgress
   [deploy] Request Status: InProgress
   [deploy] Request Status: InProgress (0/3) -- Processing Type: ApexPage
   [deploy] Request Status: InProgress (1/3) -- Processing Type: CustomObject
   [deploy] Request Status: InProgress (8/3) -- Processing Type: ApexClass
   [deploy] Request Status: InProgress (8/3) -- Processing Type: ApexClass
   [deploy] Request Status: InProgress
   [deploy] Request Status: Succeeded
   [deploy] ********** DEPLOYMENT SUCCEEDED **********
   [deploy] Finished request 0AfG0000007a5kiKAA successfully.

deploy-with-check:

BUILD SUCCESSFUL
Total time: 18 seconds
```

EXTENSION OF THE EXERCISE

The build script contains the necessary targets, and especially the all target, to build a sequence of actions extracting the metadata from one org, making a set of validations, and deploying the package if a set of rules is valid.

TROUBLESHOOTING

If you receive the following error, this means that you have not added the ant-saleforce.jar file in the Libraries folder.

```
BUILD FAILED
/Users/xxx/Downloads/ChangeMgmt/build.xml:135: Problem: failed to create task or
type antlib:com.salesforce:deploy
Cause: The name is undefined.
Action: Check the spelling.
Action: Check that any custom tasks/types have been declared.
Action: Check that any <presetdef>/<macrodef> declarations have taken place.
No types or tasks have been defined in this namespace yet

This appears to be an antlib declaration.
Action: Check that the implementing library exists in one of:
      -/Users/xxx/Documents/Dev/apache-ant-1.9.4/lib
      -/Users/xxx/.ant/lib
      -a directory added on the command line with the -lib argument


Total time: 1 second
```

2. APPLICATION LIFECYCLE MANAGEMENT

Given the project risk and customer requirement, explain how to assess the benefits and risks of the different development methodologies and recommend the appropriate methodology based on the customer environment.

This document demonstrates a use case on environment management. For more details on this topic, please read Introduction to the Force.com Development Lifecycle.

USE CASE

Universal Containers is using Salesforce to run their sales processes. Their business processes have changed over the years and resulted in some of the legacy overhead being carried over. They have decided to redesign their application leveraging the latest features from the platform. As part of this initiative, they want to address the following challenges:

- The company currently has a mix of admins and developers, and everyone pushes changes to production directly.
- Environment refreshes are not planned, and this results in unavailability of the updated environment and data when required.
- Deployments are not predictable and sometimes take multiple attempts to succeed. UC wants to automate the deployment process.
- The production environment needs to be controlled and auditable for every change that occurs.

Recommend an environment strategy to address these challenges.

PREREQUISITE SETUP STEPS

1. Create a Deployment Admin in Salesforce. It's a best practice to have segregation of roles and responsibilities. It helps with auditing and access control.

CONSIDERATIONS

- What are different types of sandboxes available in each of the editions?
- What are the considerations for recommended sandbox type?
- What are the different deployment tools and techniques available with the Force.com platform?
- What are the pros and cons for each of the tools?
- What features can we leverage for auditing?

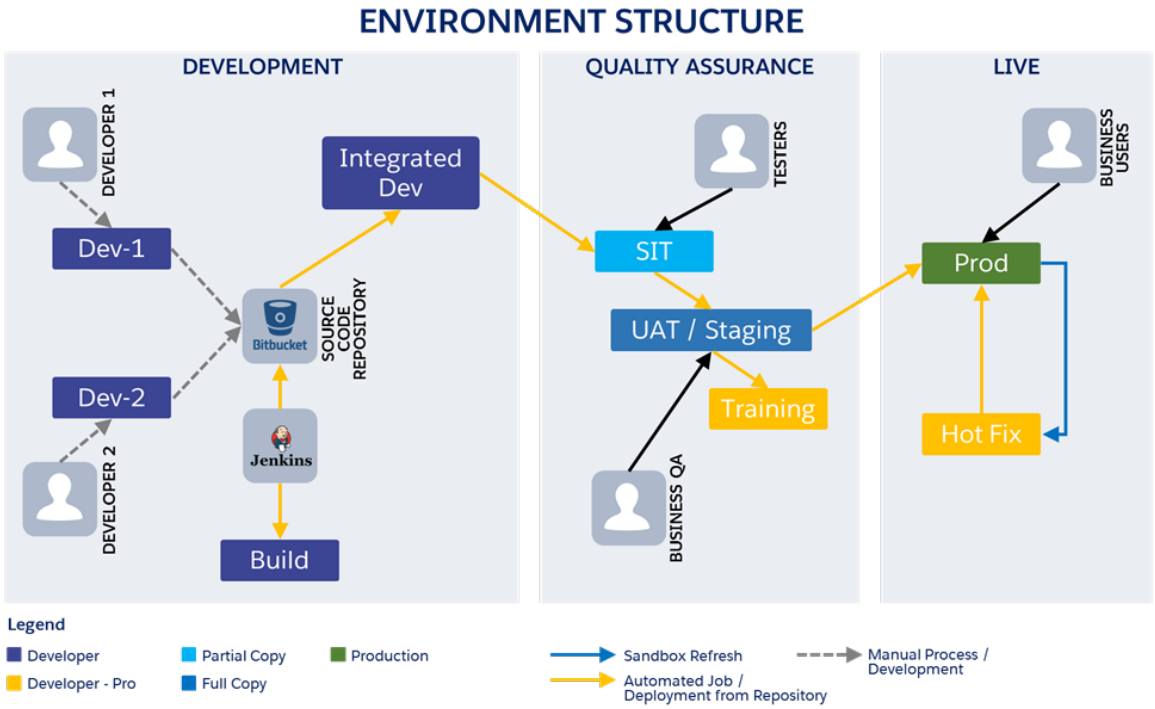
SOLUTION

BEST SOLUTION OVERVIEW

Step 1: Create a list of environments with actors for each one

- Salesforce provides different types of sandboxes, such as Developer, Developer Pro, Partial Copy, and Full Copy.
- At a bare minimum, the software development lifecycle has three stages: Build, Test, and Live.
- The Build stage is owned by the Developer. The environment used during the build stage will be used by Developers. It will not need a large amount of storage.
- The Quality Assurance team owns the Test stage. The environment used during the test stage will need data for testing. The data migration testing will need a full set of data, and a Full Copy sandbox is the right choice.
- Functional Testing or System Integration testing will need some data for testing but may not need a full set of data, so Developer Pro or Partial Copy sandboxes are the right fit here.
- Performance testing and User Acceptance Testing are the right use cases for a Full Copy sandbox, as it provides a full set of data and baseline application performance.
- The Live stage is Production and only business users have access to it.

Step 2: Map out the environment's structure



Universal Containers is running multiple projects at the same time. However, they want to standardize the process to ensure that the environments are being used optimally. They would like complete autonomy for each of the projects as well.

Universal Containers is using a Software Configuration Management (SCM) tool for version repository and collaboration. The SCM tool referred to here is Git.

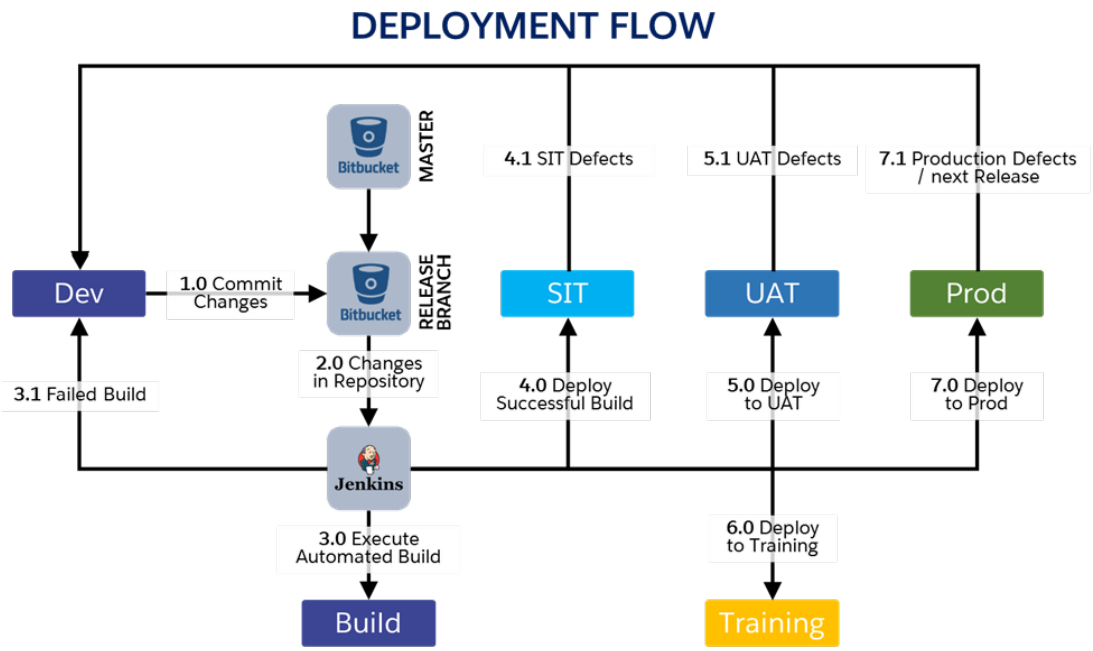
Developer 1 is working on one project in a sandbox, while Developer 2 is working on a separate project in a different sandbox. Each of the projects continues its development and commits the code to the SCM repository.

The Integrated Development sandbox is where both developers commit their project, related codebases, and test them together. This is necessary because Project 1 changes may conflict with Project 2 changes. The Integration Development environment identifies those conflicts early in the Software Development Lifecycle so that potential reworking in later phases is avoided.

If the tests in Integrated Development are successful, the project moves on to System Integration Testing (SIT), User Acceptance Testing (UAT), Training, and finally Production.

The Hotfix is an environment that is introduced to the process if there are any Production defects that need to be urgently addressed. This environment–also known as QuickFix or shortest path to Production–is mainly used by the Production Support or Level 1 Support teams. It corrects critical or serious defects at a faster rate through off-cycle deployment.

Step 3: Define the deployment path from lower environments to higher environments.



In an enterprise environment, Production and other higher environments are controlled. That means access to those environments is limited and all the changes to them need to follow a predefined Change Approval process.

We always recommend that our customers have a different set of environments to aid in the deployment flow.

We use a version control tool to commit all of our changes, since there will most likely be multiple developers working on changes. The tool promotes collaboration between the team and provides auditing on the changes. In this particular diagram, we are using BitBucket.

We use Jenkins as our continuous integration tool for a stable and predictable build and to ensure that recent changes are not resulting in broken test cases. We schedule it to run for every commit into version control.

There are two major results that we are achieving:

1. We ensure that our build is always intact by using a Developer sandbox called Build. Once the first commit has been made, Jenkins retrieves those changes from the repository, runs all the test classes, and deploys to the Build sandbox to make sure all the text classes have run successfully.
2. If there are any failures, Jenkins will notify the developer who broke the build. We don't have to wait until it deploys to the Production environment to reveal any defects.

Jenkins also automates deployments with the help of the Force.com Migration Tool. While Salesforce does provide change sets and the Force.com IDE as deployment options, the Force.com Migration Tool is the best choice for an automated build process. It is an Ant-based script that supports all metadata and allows migration of metadata from one environment to another across different production orgs. This option requires moderate technical skills for initial setup.

Once our build is successfully deployed, it will move into the SIT environment, where the Quality Assurance (QA) team can run all of our integration tests and functional scripts with certain defined data and all possible scenarios. Any defects that are found will be sent back to the Development cycle.

Next, the build will deploy to the User Acceptance Testing environment, where the same and possibly different scripts will be executed and signed off on by power users or business SMEs.

Next, we deploy to the Training environment to train the users.

The final deployment is to the Production environment, where all of our sales and service users will use the system. Any production defects or product enhancement requests will then feed back to the development cycle.

3. GOVERNANCE

Given a customer scenario, analyze and recommend the appropriate governance framework.

USE CASE

Universal Containers has been using Salesforce for sales and service for over four years. You have been asked to perform a review of Universal Containers' environment and deployment strategy. They would like to re-architect their current strategy to allow them to be more agile and responsive to their business needs. Over the last four years, they have been using the Force.com IDE to deploy metadata between their sandbox environments and also to production. They also used change sets for a limited time, but their developers found the IDE preferable.

DETAILED REQUIREMENTS

Some challenges they have faced include:

- Unpredictable results when deploying metadata. Sometimes deployments fail when they worked just fine in another sandbox.
- Lack of auditability of what has been released.
- The number of bugs reported from Production after deployments is consistently high and rising.
- The QA team is not able to keep up with the development work being done. Their testing scripts are 80% manual.
- Deployments take upwards of 3 hours, even though the actual deployment in Salesforce takes only 20 minutes. This is due to several failures before a successful deployment.
- Sandbox refreshes also take significant time. They have to manually change users, permissions, and load test data.
- Their release process doesn’t handle parallel development streams very well.
- They are unsure of whether they should have each developer in their own sandbox, multiple developers in a single sandbox, or some hybrid approach.

CURRENT INFRASTRUCTURE

The current infrastructure at Universal Containers is outlined below:

- They have a 50/50 mix of code developers and point-and-click administrators.
- Their Sales Cloud team releases software every month and their Service Cloud team releases every three months.
- They currently give each developer their own sandbox.
- UC encourages innovation and senior executives often require separate development environments for their teams' innovation work.

CONSIDERATIONS

1. Who are key individuals/roles involved in the application lifecycle management and deployment?
2. What role is each individual going to play?
3. What are the responsibilities for each role?
4. How do you align different stakeholders towards organizational goals?
5. What key performance indicators (KPIs) are critical to measure the effectiveness of release management?
6. How is the roles/responsibilities alignment tracked and measured?
7. Is the role/responsibility matrix documented? Is it a formal process?
8. Are there audit and compliance requirements for each role?

SOLUTION

BEST SOLUTION OVERVIEW

Center of Excellence (CoE) – Key Release Management Roles and Responsibilities

Complex development processes spanning multiple release schedules and systems, as well as the types of changes made in Salesforce orgs, often make the release management process in Salesforce critical and complex. Key aspects to help solve these problems are:

- Identify the key actors/roles and their responsibilities.
- Identify the tools that will be used.
- Standardize the process.
- Identify the key KPIs.
- Monitor and measure the KPIs.

The day-to-day management of the Salesforce practice will be driven by a program team, which comprises functional teams that represent Business, IT Development (scrum), Enterprise Architecture, Environment Management, Release Management, and Support teams. Each function has a variable level of influence in the way release and change Management happens with Salesforce and at the enterprise level. Early identification of the key roles and standardizing their responsibilities will go a long way in ensuring repeatable, successful releases in Salesforce.

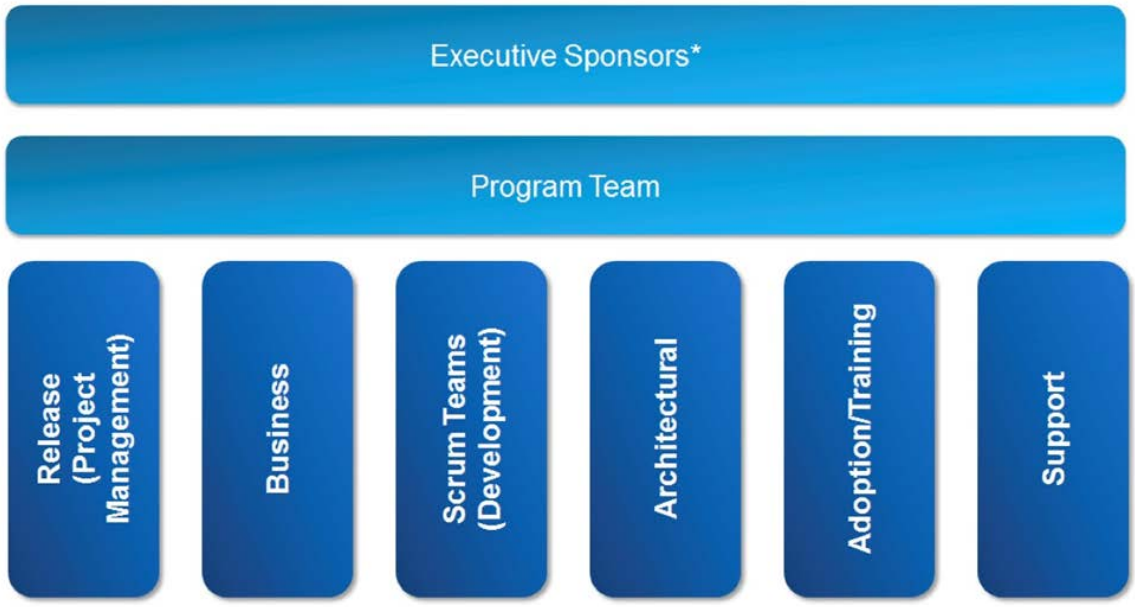


Fig 1. Typical Salesforce Release Management CoE

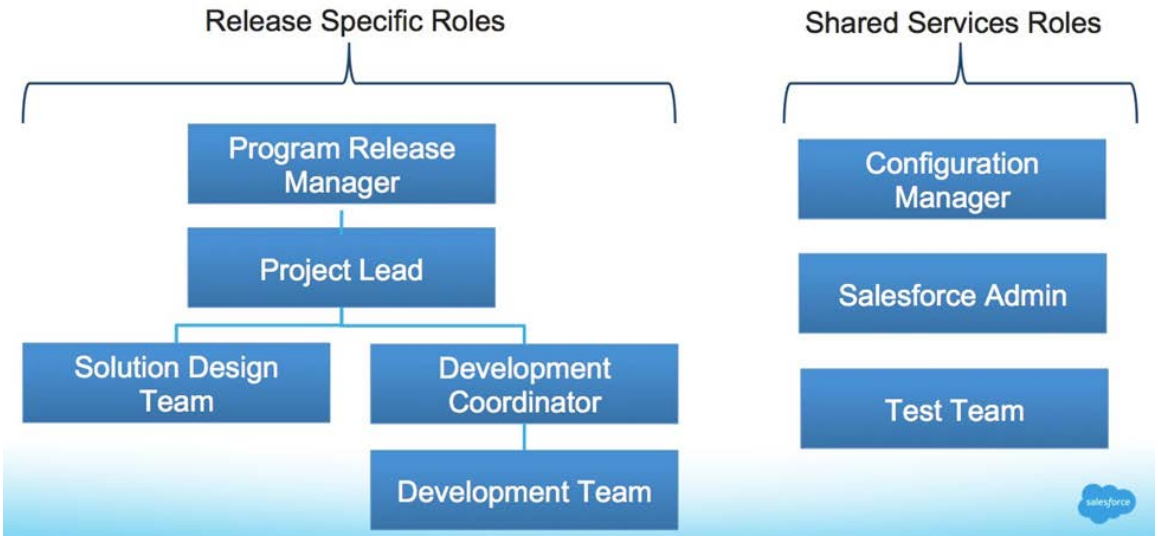


Fig 2. Release Management CoE team

Role	Key Responsibilities	Sandbox Type	Sandbox Access Level
Program Release Manager	• Coordinates and plans the development projects (work streams) for code drops and merges to System Integration Testing (SIT), Quality Assurance Testing, User Acceptance Testing (UAT), and Production Environments. • Coordinates code, metadata, and data migrations. • Smoke tests the release. • Represents the proposed release and obtain the approvals from Change Approval Board (CAB) /Change Control Board (CCB). • Creates and sets release calendar for all features and enhancements. • Coordinates the release with other releases of the enterprise. • Reviews back-out plans and authorizes the back-out plan (if required.)	All	Admin

Role	Key Responsibilities	Sandbox Type	Sandbox Access Level
Project Lead	• Ensures developers' code is integrated in the DevInt sandbox environment on daily basis. • Ensures that there is a successful build at the end of every day's development effort. • Ensures daily check-in of code and metadata elements that were changed etc. into approved version control system. • Responsible for resolving conflicts in changes made in metadata/code etc. by the developers. • Maintains the sanity of DevInt sandbox. • Ensures the unit test coverage tests are invoked after every build activity. • Ensures early resolution of conflicts in changes. For example, code or metadata is reconciled in a timely manner. • Ensures that no unauthorized developer makes changes in code beyond the DevInt sandbox environment. • Ensures that access to the DevInt sandbox is limited to authorized developers only. • Provides necessary metrics and data that show the health of changes in Dev sandboxes as they move to the DevInt Environment. For example, the dashboard of successful builds versus failed builds in DevInt sandbox, etc.	All	Admin

Role	Key Responsibilities	Sandbox Type	Sandbox Access Level
Enterprise Architecture Team	• Makes recommended list of products/tools, such as: □ Enterprise version control system(s). □ Enterprise testing tools (performance testing, test automation tools). □ Enterprise Integration tools. □ Enterprise Master Data Management tools. □ Enterprise release management tools. □ Enterprise mobile device management tools. • Makes recommended processes around: □ Data migration (large volumes). □ Metadata migration. □ Code migration. • Representation in Change Approval Board (CAB)/Change Control Board (CCB).	All	Admin
Development Coordinator	• Responsible for daily builds in the DevInt sandbox environment. • Dev Sandbox refreshes. • Data priming on Dev Sandboxes. • 1st-level merge coordinator for parallel developments. • Responsible for maintaining the Developer and DevInt sandboxes.	Dev, SIT, UAT	Admin

Role	Key Responsibilities	Sandbox Type	Sandbox Access Level
Environment Manager	• Maintains all the environments excluding PROD instance. • Authorizes Sandbox refreshes. • Publishes sandbox refresh schedules. • Plans full sandbox refreshes. • Authorizes the data loads in to sandbox environments. • Report on comparison of various sandbox environments.	Dev, SIT, UAT (all except Production)	Admin
Developers	• Ensure early resolution of conflicts in changes; code or metadata is reconciled in a timely manner.	Dev, SIT, Hotfix	Admin
Configuration Manager	• Ensures adherence to configuration management policies. • Provides Subversion access. • Responsible for trunk/branch creation for new releases and service packs. • Responsible for providing tag/label for release and code drops. • Ensures code drop tag is not modified after specified point in time.	All	Admin
Salesforce Admin	• Provides requisite access to Salesforce users. • Provides snapshot reports (for metadata comparison purposes). • Sandbox creation and refreshes. • Data loads. • Code migration to the System Integration Test (SIT), Performance Quality Testing (PQT), User Acceptance Testing (UAT), Training, and Prod environments.	All	Admin

Role	Key Responsibilities	Sandbox Type	Sandbox Access Level
Test Team	• Creates test plan and test scripts applicable for the release. • Performs PQT on developed product. • Reports defects.	All	End User Level
Support Team	• Supports ongoing production issues. • Troubleshoots issues reported. • Reproduces issue in the training/full copy sandbox. • Tests and validates hotfix.	Production, Training, UAT, Hotfix	Delegated Admin

Roles and Responsibilities in the Incident Resolution End-to-End Process

Day-to-day end user support is critical to the overall success of the platform and from the end user experience point of view. Very much similar to the overall development and deployment process, team members and their roles are critical in incident resolution. The overall program benefits from a well-defined process and clearly defined roles and responsibilities. The incident management process also helps the development team fully focus on various projects in flight while support resources manage day-to-day support with a clearly defined escalation process (tiered support model). The following table illustrates various roles in the incident resolution process:

Role	Responsibilities
End User	• Reports issue through well-defined process and tool; for example, reporting via a portal (submitting a case or sending an email). We highly recommend a case management process. • Follows up on the issue. • Tests reported issue in the sandboxes. • Confirms the closure of the issue.

Role	Responsibilities
Support Team	• Logs the issue. • Reproduces the issue in the Full Copy Sandbox. The Support Team often has a delegated admin to Production environment where a team member could try to follow steps as an end user. • Follows up with the end user for any additional details. • Escalates the case to the next level of support (admin or designated developer) for fix. • Coordinates efforts to get issue fixes in sandboxes. • Communicates with end users with the progress and estimated delivery date of the fix. • Closes the issue.
Test Team	• Tests issues in various sandboxes as defined by the incident management process.
Admin	• Fixes issues in the Hotfix environment and Production as per the deployment process (for configuration related issues.)
Developers	• Troubleshoots and fixes development related issues. • Coordinates with the Release Manager to promote changes in the sandboxes as per the defined Hotfix process.
Environment Manager	• Coordinates efforts in code/configuration migration among Full Copy/Training, Hotfix, and other sandboxes based on fixes getting deployed. This process ensures that a production issue getting fixed is not reintroduced and that the fix is patched/merged in all environments.
Release Manager	• Identifies the date and time to promote the fix in Production. • Deploys the fix in Production.

RACI Matrix

RACI is an acronym for:

R = **Responsible**: owns the project/program/initiative

A = to whom "R" is **Accountable**. Person who must sign-off or approve.

C = to be **Consulted** – the one who must be consulted for advice.

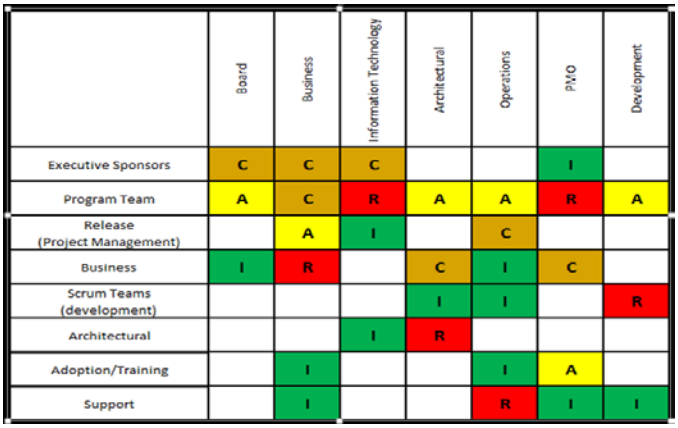
I = to be **Informed** – the one who should be informed about the success or issues but not necessarily consulted.

A complex environment and matrix-driven organization could benefit from defining a RACI Matrix. The RACI Matrix helps define the role of various individuals at the program level. It helps with transparency and alignment, improves on decision-making, and establishes accountability. Variations of the RACI matrix could be established at project, program, and/or at the Enterprise level. A sample of RACI Matrix (or also known as RACI Chart) is illustrated below.

Here are a few techniques or potential steps to develop a RACI Matrix:

- Identify major processes and sub-processes. Activities should be listed in the first column of the matrix.
- Identify all the roles and list them in the header of your chart.
- Fill out the matrix with R, A, C, I for each activity and role.
- Resolve overlaps if there are any.

It is okay to have more than one role for a few activities. For example, a business- type role could be R (Responsible) as well I (Informed) for a release.



	Board	Business	Information Technology	Architectural	Operations	PMO	Development
Executive Sponsors	C	C	C			I	
Program Team	A	C	R	A	A	R	A
Release (Project Management)		A	I		C		
Business	I	R		C	I	C	
Scrum Teams (development)				I	I		R
Architectural			I	R			
Adoption/Training		I			I	A	
Support		I			R	I	I

ADDITIONAL ACTIVITIES

To practice these activities, you can leverage one of your Playgrounds or other resources if needed;

- Ant Script Using the Force.com migration tool.

  - Create an Ant script to migrate changes from Development to Production.
  - Experiment with the different flags available and the metadata APIs.

- CRUD-based Calls
  - Create an application to perform CRUD-based calls. Click here for the Example code:
    EXAMPLE CODE