Module 1

I. 1.1 Platform Review
    A. Point of delivery (POD) and Realms
        1. **POD** - self-contained DW hardware cluster that spawns a large number of virtual customer instances in a multi-tier service environment
            a) Contains multiple application servers, DB servers and the clustering and backup infrastructure
            b) Each POD is hosted in a third party data center
            c) PODs are partitioned into **realms**, which are associated with a given customer
            d) **Realm** can contain one ore more eCommerce sites(SF)
                (1) Typical merchant will operate in a single realm
                    (a) Within this realm, merchant can develop, stage and deploy multiple sites
                (2) Each DW customer is allocated one or more realms
    B. Instance and Instance Groups
        1. <u>Realm</u> - Collection of resources which are used for developing, testing, hosting one or more DW sites.
            a) <u>PIG</u> - Primary Instance Group
                (1) <u>Production</u> - Live instance used as actual storefront(SF)
                (2) <u>Staging</u> - Instance for Configuration, data enrichment, data import and uploading code to prepare it for testing in Develop instance
                    (a) Through data replication you can move data to production or development
                (3) <u>Development</u> - developers can test processes without impacting production SF
            b) <u>SIG</u> - Secondary Instance Group
                (1) 5 (optionally more) sandboxes (max 47)
                (2) Not as powerful performance, memory, storage BUT smaller footprint

II. 1.2 Platform Tools - Business Manager (BM)
    A. Used by merchants and developers to manage admin tasks
        1. Every DW instance has a BM
    B. BM Merchandiser Tasks Mgmt:
        1. Product and Catalog
        2. Content
        3. Marketing Campaigns
        4. Search Settings
        5. Customers
        6. Site Analytics
        7. Site Urls
    C. BM Developer Tasks Mgmt
        1. Code and Data Replication
        2. Code Versioning
        3. Site Development
        4. Data import/export

5. Global preferences for all site / organization

III. 1.3 Sitegenesis SG foundation architecture
   A. Sitegenesis - (SG) DW sample reference site
      1. Can be basis for own sites
      2. Full featured ecommerce site
      3. Use it to explore DW platform and capabilities
      4. For Devs, sample code, pipelines, scripts and isml
      5. For Merchs, provide sample configuration for catalog, categories, products, etc
   B. Can import SG into each instance in you SIG
      1. * can overwrite data and existing attributes when importing sitegenesis into a sandbox that contains a custom site - [**SO DO NOT DO IT**] *
      2. * **DO NOT IMPORT SG INTO PIG** *
      3. * Safe to import SG into an **empty** sandbox or **before** importing a custom site
   C. Sharing Data between Sites
      1. SG contains data specific to that site, BUT some data is shared among all sites in the organization
         a) Two major kinds of catalogs that can be shared
            (1) Master Catalogs - contains all products and is at organization level
            (2) Site Level (storefront catalog) - For specific site catalog category and products to control navigation for each site
               (a) Sites can have diff category and products assigned to them
         b) Therefore, sites must have only one site catalog, BUT may have many master catalogs

Module 2 : UX Studio

I. 2.1 UX studio Introduction and creating a workspace -
   A. UX studio (UXS) - eclipse specific local directory that contains eclipse projects
      1. Normally Java source directory (packages)
      2. Defines the connection to DW instances OR they point to a DW cartridge
      3. Each workspace should have only 1 DW server connection

II. 2.2 Creating a server connection
   A. To upload code to a DW server, you must create a server connection in UXS
      1. Once connected you can push/upload code to server instance, BUT you cannot pull the code to your local computer
         a) Ie, one way push connection

III. 2.3 Importing Projects (cartridges)
   A. Cartridge - project in UXS - folder with specific sub-folders and files
   B. NOTE: If SVN server, you can import projects directly from a repository (most common)

IV. 2.4 DW Views
   A. UXS provides access to specific DW programming files
      1. Sorted and given their own custom views, cartridges, pipelines,pipelets, templates, scripts
         a) Tabs for each
            (1) Tips
               (a) Can filter results by typing first letters of file to be searched
               (b) show/hide resource part to view file path of file

(c) Navigation - view all files in workspace with tree view structure,
   (i) Facilitates common tasks, like copy/paste and file compare, etc.

Module 3 Cartridges

I. Introduction
   A. Cartridges provide a directory structure that provides a flexible deployment mechanism for customized functionality
      1. Contains:
         a) Static files
         b) Images
         c) WSDL
         d) Pipelines
         e) Scripts
         f) Templates
         g) Form definitions
         h) Controllers
      2. Fully contained in one directory
      3. Specific sub folders per file type
      4. NOTE: storefront.properties is required; Generated by UXS when creating a new one

II. 3.1 Cartridge Path
   A. When a call to a file is made, DW server looks for the file starting with the first cartridge listed in Cartridge Path
      1. In order for site to use a cartridge, you must add the cartridge path in BM @ Sites -> Manage Sites ->[Site] -> settings

III. 3.2 Cartridge Types
   A. 3 types
      1. DW storefront cartridge - new SF cartridge contains a copy of default SG cartridge available in SG demo site package
      2. DW cartridge - use to build site specific reusable functionality, when there are multiple sites in a production instance
         a) Create when functionality is:
            (1) Generic / reusable in multiple sites
            (2) Integration to an external system
            (3) Specific to a localized site; CSS, images, resource files for a language specific site
      3. DW BM extension cartridge - NOT COVERED
   B. Best Practices
      1. Keep an original SG cartridge in your project for comparison
      2. Use a SF cartridge for common code that you intend to reuse in multiple sites: [client code]_code
      3. Create cartridges for site specific functionality that might override the core app_[site]
      4. Place integration code in int_[site] cartridge

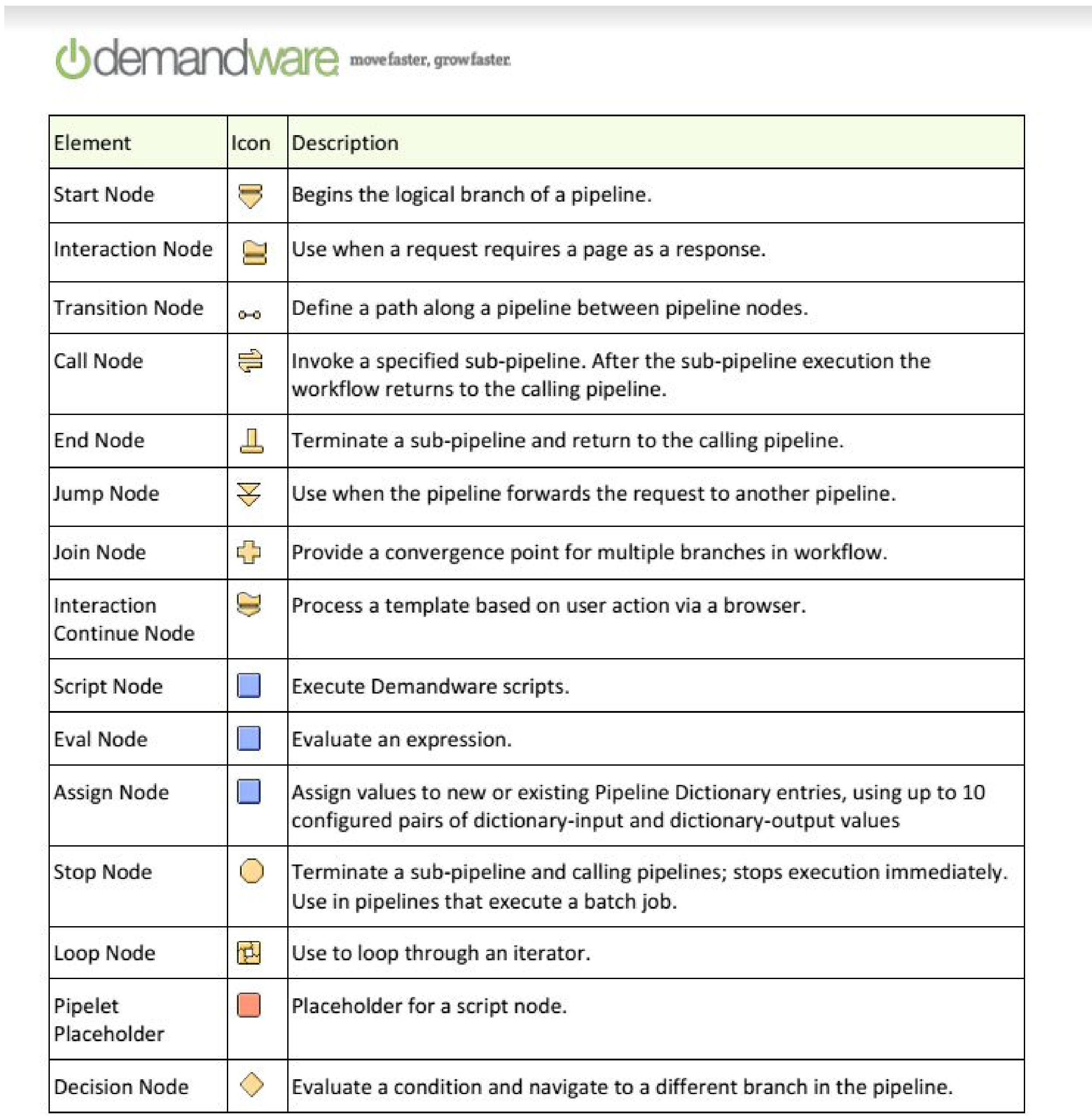
IV. 3.3 Create a new cartridge
   A. To segregate code between sites, create a new cartridge

V. 3. 4 Creating a new DW SF cartridge

A. When you create a new SF cartridge in UXS a copy of SG is downloaded to your workspace and renamed with the name you specified
    1. Reference cartridges have all the code you need for SG to work
B. SG cartridge
    1. Contains:
        a) The Business layer, server side components [pipelines, scripts] and non JS version of SF
        b) Simple presentation layer including isml templates, common css files, forms and resources
        c) Specific css and ui elements for advanced look and feel
C. After changes to new SF, they will be uploaded to DW server
    1. To view: set the cartridges to be uploaded to your workspace server
    2. Put the new cartridge in the cartridge path
    3. Disable site caching for the site

Module 4: Pipelines and JS controllers

I. Introduction
    A. <u>Pipelines</u> - logical model of a particular business process (like a flowchart)
        1. Stored in XML in file system (local and server)
        2. Defined within context of a cartridge
        3. When referenced, application looks at cartridge path and uses the first found
            a) When a pipeline with the same name exists on two cartridges, the first one found in the path is used. Therefore, use unique pipeline names
    B. Introduction to pipeline element icon
        1. <u>Start</u> - begins logical branch of pipeline
        2. <u>Interaction</u> node - when request requires a page as a response
        3. <u>Transition</u> node - defines a path along a pipeline between nodes
        4. <u>Call</u> node - invokes a specified sub pipeline
            a) When it is done executing workflow returns to calling pipeline
        5. <u>End</u> node - terminate a sub pipeline and returns to calling pipeline
        6. <u>Jump</u> node - Use to forward request to another pipeline
        7. <u>Join</u> node - convergence point for multiple branches in workflow (WF)
        8. <u>Interaction Continue Node</u> - ICN process a template based on user action via browser
        9. <u>Script</u> node - execute a DW script
        10. <u>Eval</u> node - eval an expression
        11. <u>Assign</u> node - assign values to new or existing pipeline dictionary entries,
            a) Up to 10 configured pairs of dictionary - in / dictionary - out
        12. <u>Stop</u> node - terminate a subpipeline *and* calling pipeline; stops execution immediately.
            a) Used in pipelines that execute batch jobs
        13. <u>Loop</u> node - use to loop thru an iterator
        14. <u>Pipelet</u> placeholder - placeholder for script node
        15. <u>Decision</u> node - eval a condition and navigate to a branch in the pipeline


demandware move faster, grow faster

Element	Icon	Description
Start Node	▽	Begins the logical branch of a pipeline.
Interaction Node	▭	Use when a request requires a page as a response.
Transition Node	⊶	Define a path along a pipeline between pipeline nodes.
Call Node	⇄	Invoke a specified sub-pipeline. After the sub-pipeline execution the workflow returns to the calling pipeline.
End Node	⊥	Terminate a sub-pipeline and return to the calling pipeline.
Jump Node	⩡	Use when the pipeline forwards the request to another pipeline.
Join Node	✚	Provide a convergence point for multiple branches in workflow.
Interaction Continue Node	▭	Process a template based on user action via a browser.
Script Node	🟦	Execute Demandware scripts.
Eval Node	🟦	Evaluate an expression.
Assign Node	🟦	Assign values to new or existing Pipeline Dictionary entries, using up to 10 configured pairs of dictionary-input and dictionary-output values
Stop Node	●	Terminate a sub-pipeline and calling pipelines; stops execution immediately. Use in pipelines that execute a batch job.
Loop Node	⊞	Use to loop through an iterator.
Pipelet Placeholder	🟥	Placeholder for a script node.
Decision Node	◇	Evaluate a condition and navigate to a different branch in the pipeline.

C.

II. 4.1 Create a pipeline
   A. At minimum pipeline needs one start node and one interaction node(or end/stop/jump node??) (and transition node)
   B. <u>Start</u> Node - Pipeline may have multiple start nodes but each name must be unique
      1. Each start node is a different logical branch start point
      2. Configurations
         a) Name - used in pipeline call
         b) Call Mode - specifies the accessibility from the browser
            (1) Public - can be called from browser or from another pipeline
            (2) Private can be called from another pipeline via call or jump node

- c) Secure connection required -
  - (1) False - start node can be invoked with http(s) protocols
  - (2) True - start node can only be accessed with https

C. interaction node specifies a template to display in browser

1. Configurations
   - a) Dynamic template
     - (1) True - template expression must be a variable containing a template name
       - (a) Template to be called by an interaction node is not always hardcoded in the node, it could be determined dynamically during run time from pdict
     - (2) False - template expression contains a path to the template under the template(default) folder

D. Transition node - creates a transition between two nodes

III. 4.2 Executing a pipeline

A. Pipeline can be executed from a browser via http(s) request or by call/jump nodes

1. Http - name and start node need to be at end of SF url
   - a) /CustomerService-Show
     - (1) [Pipeline Name]-[startnode]
   - b) /Product-Show?pid=908
     - (1) [Pipeline name]-[startnode] + ? + [param key] + = + [param value]

B. Steps for pipeline execution

1. Open SF at end of URL add /default/ (directory) and enter pipeline name - start node name
   - a) Eg, /Site-Sitegenesis-site/default/Hello-Start
2. Pass params, query strings after invocation

IV. JS Controllers

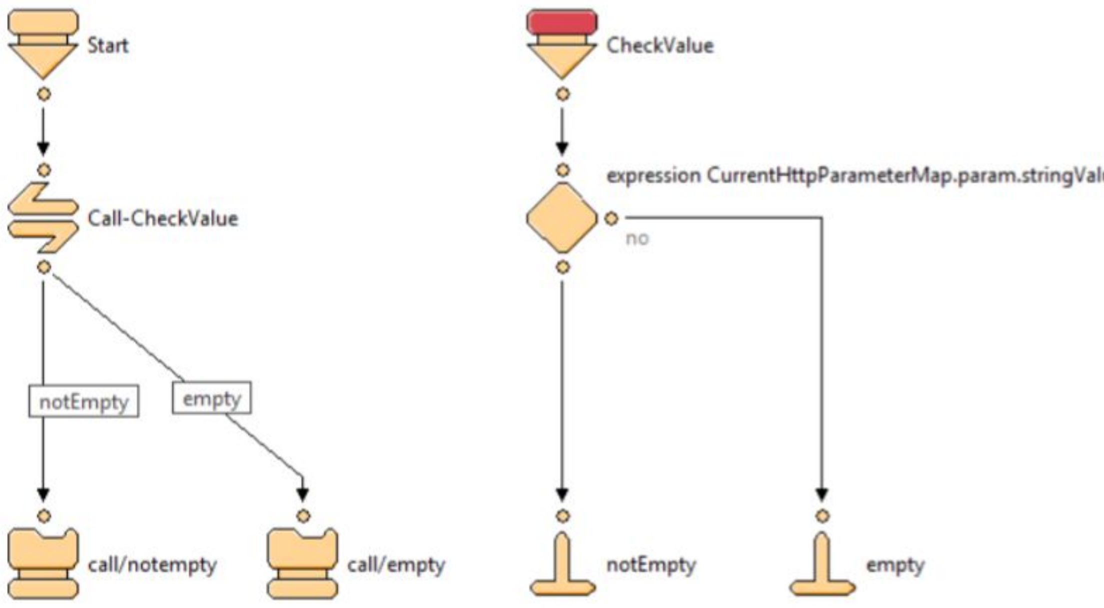
A. Benefits

1. Greater efficiency and collaboration by reducing pipeline related pain points
2. Provides choice of IDE, no need for UXS
3. Use current URL schema to leverage existing SEO features
4. Script profiler updated to mimic pipeline profiler

B. Cartridge folder structure

1. Can have pipelines and controllers OR just controllers
   - a) Controllers must be at same level as pipelines
2. If pipeline and controller share name it uses the controller, even if they are in different cartridges
3. NOTE: if sub pipeline is not named in accordance with JS method naming conventions, you MUST rename it to override it
4. <u>Require</u> - keyword used to import classes from API package to be used in code
   - a) var.guard = require('storefront_controller/cartridge/...')
5. <u>Guard</u> - exposes the function with a new name
   - a) export.Start = guard.ensure(['get'], start);
     - (1) Exposes "start" as "Start" to URL
     - (2) Can also enforce Http methods
6. ISML object gives the control to isml which can display results:
   - a) var ISML = require('dw/template/ISML')

7. Can also use response object to directly display the results [NOT RECOMMENDED]
    a) response.getWriter.println('whatever');
8. Troubleshoot by using Request Log Tool
    a) Enables you to troubleshoot error messages on your SF
    b) Part of SF toolkit (available on all instances, except production)
    c) Logs last request and any request prior to it during current session
    d) View both debug/error messages

C. Call nodes and End nodes
1. Call nodes - invokes a specific sub pipeline
    a) Sub pipeline - pipeline designed for reusability and typically is defined as private (NO URL INVOCATION)
        (1) After sub pipeline executes workflow returns to calling pipeline by an end node
        (2) Requires pipeline start node to invoke
        (3) Can define this as fixed configuration value or from pdict key
2. End node - finishes the execution of the called sub pipeline and returns a value equal to the end node name
    a) Name must be unique within sub pipeline and may be used by the calling pipe to control workflow after the call
3. After call, a transition node from the call node with same name as return value is followed
4. 

5. Jump node - invokes specified pipeline
    a) After the sub pipeline executes workflow does NOT return to calling pipeline
        (1) Subpipeline must complete task
    b) Requires
        (1) Name of pipeline to jump to
        (2) Name of pipeline start node to be used

                (3) These can be configured by config values or by pdict key

V. Pipeline dictionary (pdict) - main container for each pipeline execution
    A. created/initiated when pipeline is invoked and remains in memory as long as pipeline is executing
    B. Hashtable with key value pairs
        1. Default keys:
            a) CurrentVersion
            b) CurrentDomain
            c) CurrentOrganization
            d) CurrentSession
            e) CurrentPageMetaData
            f) CurrentRequest
            g) CurrentUser
            h) CurrentHttpParameterMap
            i) CurrentForms
            j) CurrentCustomer
    C. Passed between sub pipeline calls
    D. Pdict to global variables
        1. Controllers have access to request, response, session and customer objects IF valid import / require statements are used
        2. Pdict keys - js controller alternatives
            a) CurrentSession - TopLevel.global.session
            b) CurrentRequest - TopLevel.global.request
            c) CurrentCustomer - TopLevel.global.customer
            d) CurrentHttpParameterMap - TopLevel.global.request.httpParameterMap
            e) CurrentPageMetaData - TopLevel.global.request.pageMetaData
            f) CurrentForms - TopLevel.global.session.forms
    E. importPackage and require
        1. importPackage - (old way) must be added at beginning of script
        2. Require - (newer way) can require a module anywhere in script
            a) Means only loads functionality if needed, good for performance
    F. Passing parameters
        1. Params get added to CurrentHttpParamterMap obj in pdict (OR TopLevel.global.request.httpParameterMap in controllers)
            a) CurrentHttpParamterMap is an object of type HttpParametermap

VI. 4.4 Pipeline debugging
    A. Operates at pipeline level NOT source code level
    B. Provides step by step tracking for pipeline execution and for examining the pdict at runtime
    C. Must configure remote server connection and debugger and have a running DW system

VII. 4.5 Pipelet executes an individual business function within a DW pipeline
    A. Precoded pieces of functionality provided by DW
    B. Other types
        1. Script - invokes a custom DW script file
        2. Eval - evaluates data in pdict
        3. Assign - assigns values to keys in pdict
    C. Belongs to the bc_api cartridge
        1. Downloaded as part of DW API upon server connect

D. Documentation on function, input and output params can be seen in properties view

E. Script API in JS controllers
   1. var ProductMgr = require('dw/catalog/ProductMgr')
      a) Retrieves product Object
   2. var parameterMap = request.httpParameterMap
      a) Gets params from url
   3. var product = ProductMgr.getProduct([id])
      a) ProductMgr can get product
   4. OR it is possible to call a DW pipelet[**NOT RECOMMENDED TO CALL PIPELETS FROM JS CONTROLLERS**]
      a) var GetProductResult = new dw.system.Pipelet('GetProduct').execute({
         ProductID : pid.stringValue
         });
         var myProduct=GetProductResult.Product;
      b) Example of pipelet call

Module 5 ISML

I. <u>ISML</u> - internet store markup language
   A. Determines how data, tags, page markup are transformed into html sent to the browser
      1. Uses CSS for layout/ styling
   B. DW MVC architecture
      1. Templates = view
      2. Controller/pipelines = controller
      3. DW script API = model

II. 5.1 ISML tags and expressions
   A. ISML tags are DW proprietary extensions to html, which developers can use in templates
      1. These cannot be used in any file but ISML
      2. SGML -like extension tags that start with *is* (eg. <isprint>
      3. Describes , with html how dynamic data will be embedded and formated
      4. Divided into the following (9) groups
         a) <u>Http related</u>
            (1) <<u>iscookie</u>> - sets cookie in browser
            (2) <<u>iscontent</u>> - sets mime type
            (3) <<u>isredirect</u>> - redirects browser to specific URL
            (4) <<u>isstatus</u>> Defines status codes
         b) <u>Flow control</u>
            (1) <<u>isif</u>> - eval a condition
            (2) <u><iselse>/<iselseif></u> - specific optional logic when isif condition does not evaluate to true
            (3) <<u>isloop</u>> - creates loop statement
            (4) <<u>isnext</u>> - jumps to the next iteration in a loop statement(line does not reset loop, continues to next line as if nothing happened)
            (5) <iscontinue> - in loop will continue on to next iteration at top of loop
            (6) <<u>isbreak</u>> - terminates loop
         c) <u>Variable related</u>
            (1) <<u>isset</u>> creates variable
            (2) <<u>isremove</u>> removes variable

        d) Include
            (1) <isinclude> - includes content of one template in current template
            (2) <ismodule> - declares a custom tag
            (3) <iscomponent> - includes output of a pipeline on current page
                (a) Difference between isinclude: iscomponent is the output of a pipeline (not just a template), and can pass as many attributes as necessary(without URLUtils)
        e) Scripting
            (1) <isscript> - allows dw script execution on template
        f) Forms
            <isselect> - enhance html select tag
        g) Output
            (1) <isprint> - formats and encodes strings for script
            (2) <isslot> - creates a content slot
        h) Other
            (1) <iscomment> - adds comments
            (2) <isdecorate> - reuses template for page layout
            (3) <isreplace> - replaces content inside decorator template
        i) Active data
            (1) <isactivedatahead> - allows collection of active data from pages with a <head> tag
            (2) <isactivecontenthead> - collects category context from a page for active data collection
            (3) <isobject> - Collects specific object impressions/ views dynamically

B. ISML expressions based on DW script language
    1. DW script implements the ECMAscript standard, so access to variables, method and objects is the same as using JS
    2. They are embedded in: ${ … }
        a) Enables isml processor to interpret the expression prior to executing on isml tag or the rest of the page
        b) Uses dot notation
           ${pdict.myProduct.UUID}
    3. ISML must specify ${pdict.object.property} to access value in pdict
        a) BUT, in pipeline node property pdict access is implicit so "${..}" is not necessary
    4. Expressions can also access DW script classes and methods
        a) 2 packages are available implicitly in ISML, so these classes do not need to be fully qualified
           TopLevel package (eg, session.getCustomer())
           dw.web package (eg, URLUtils.url())
        b) TopLevel also has a class named global, which is also implied, so it is not needed in the prefix
        c) Other access to class methods need to be qualified
           ${dw.system.Site.getCurrent()}
        d) ${TopLevel.global.session.getCustomer} === ${session.getCustomer()} => true

            e) Expressions can also have complex arithmetical, boolean, string operations

    C. <isredirect>
        1. Can redirect the control to another pipeline and redirect can be permanent or temporary
            a) <isredirect location=”${URLUtils.https(‘Account…’)” permanent=”true” />
    D. <isprint>
        1. Prints formatted information output of a variable or an expression to browser
        2. Uses built in style formaters
        3. <isprint value=”${myVar}” style=”money_long” />

III. 5.2 creating and accessing variables
    A. <isset>
        1. Name and value are required attributes
        2. Must be assigned scope, default scope = session
        3. <isset name=”[name]” value=”[value]” scope=”session|request|page”
        4. Session scope example:
           <isset name=”x” value=”whatever” scope = “session” />
           ${session.custom.x}
           ${pdict.CurrentSession.custom.x}
        5. Request scope example:
           <isset name=”x” value=”whatever” scope = “request” />
           ${request.custom.x}
        6. Page scope:
           <isset name=”x” value=”whatever” scope = “page” />
           ${page.custom.x} ===> **WILL NOT WORK**
           ${x} ===> **WORKS**
    B. Value attribute = can be string or number or an isml expression accessing different value
    C. Scope attribute - accessibility level
        1. Scopes, from widest to narrowest
            a) Global preferences - available to any site within an organization, accessible via dw.system.OrganizationPreferences class
            b) Site preferences - available to any pipeline executing as part of a site; accessible via dw.system.SitePreferences class
            c) Session - available thru the whole customer session, even across multiple request
               Any variable added to session scope becomes a custom attribute of the session object
               NOTE: not a standard attributes, needs “session.custom” qualifier
            d) Pdict - available while a pipeline executes; can encompass multiple request; similar to ICN
            e) Request - available thru a single browser request - response cycle
               Does not persist in memory for subsequent request
               Typically same as pipeline scope
               NOTE: Available via request scope, like session variables must prefix request variables with “request.custom”
            f) Page - available for a specific ISML page and its locally included pages, scope is limited to the current template and any locally included template
               Accessed without prefix ${pageVar}

        g) <u>Slotcontent -</u> available only in rendering template for a content slot
        h) <u><isloop></u> only available inside loop

IV. 5.3 Reusable code in templates
  A. Use the following tags to reuse code
    1. <u><isinclude></u> - embed isml inside an invoking template, Two types:
        a) <u>Local include</u> - include code and one template inside another while generating the page
            (1) All variables from the <u>including</u> template are available in the <u>included</u> template
        b) <u>Remote include</u> - include output of another pipeline inside an isml template
            (1) Mainly used for page caching(partial)
            (2) NOTE: *pdict and page variables from invoking templates are <u>not</u> available in the included templates*
            (3) Can not include from another server

    2. <u><isdecorate></u> - decorate the enclosed content with contents of a specified decorator template - An isml template with html, css and overall design of page
    3. <u><ismodule></u> - define your own isml tags which can be used like any standard tag
    4. <u><iscomponent></u> - invokes a remote include, can pass as many attributes as you want without having to use URLUtils.methods
  B. Remote includes
    1. Syntax:
       <isinclude url="pipeline_url" />
        a) Invokes another pipeline which will return html at runtime
    2. Syntax:
       <isinclude url="${URLUtils.url('Product-IncludeLastVisited')}" />
        a) dw.web.URLUtils.url() - builds site specific url
    3. Best practice to use URLUTILS, [**NEVER HARDCODE URLS**]
    4. Page can be dynamic or cached
    5. <iscomponent> - Remote includes with passing multiple parameters
        a) Syntax:
```
<iscomponent pipeline="[string || exp]"
        locale="[string || exp]"
        [any number of additional params(key="value")]
 />
```
        b) Example:
```
<iscomponent pipeline="Product-GetLowATSThreshold"
productid="ETOTE" typeOfTV="Wide-screen"/>
```
  C. <u><isdecorate></u>
    1. <isreplace> will identify where to include decorated content
        a) Typically there is one <isreplace>
           But can have more - content will be decorated for each <isreplace>
    2. Decorate templates in SG start with *pt_*
    3. Decorator syntax:
        a) Template using a decorator
           <**isdecorate** template = "decoratorFolder/**pt_myDecorator**">
               …. Content…. To be decorated…

```
</**isdecorate**>
```

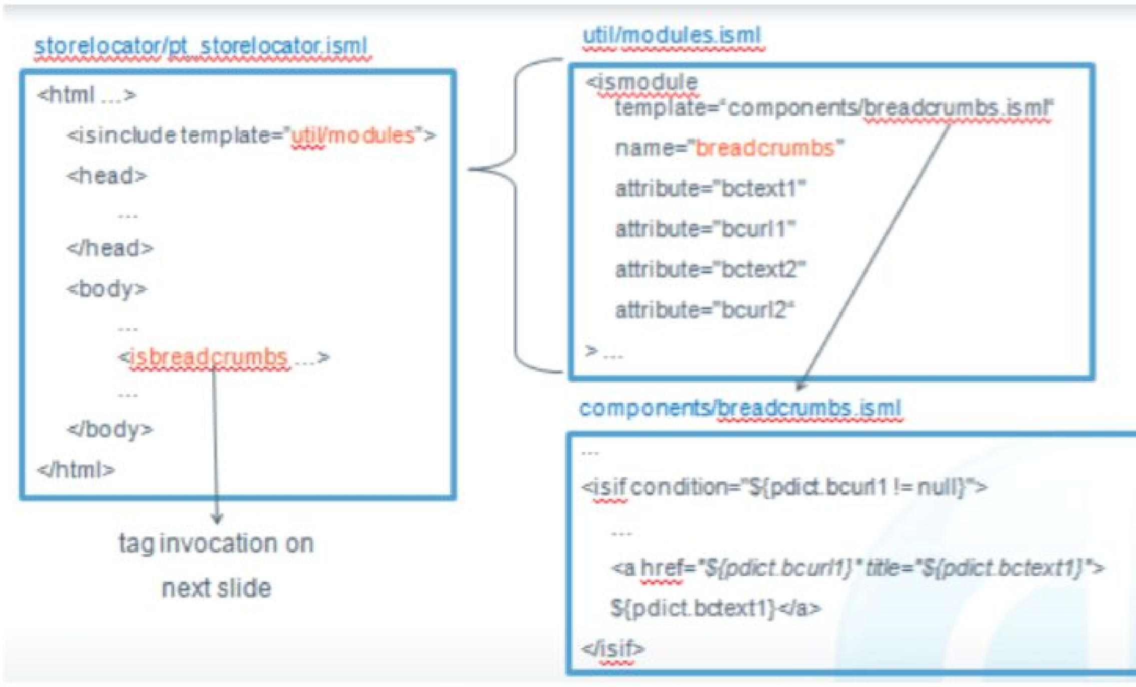
b) Decorator template: template/default/decoratorFolder/**pt_myDecorator**.isml

```
<html>
        <head>...</head>
        <body>
                This is a header.
        <isreplace/>
                This is a footer.
        </body>
</html>
```

c) Final generated page

```
<html>
        <head>...</head>
        <body>
                This is a header.
                …. Content…. To be decorated...
                This is a footer.
        </body>
/html>
```

D. Creating custom tags with <ismodule>
    1. Three files are required
        a) Isml which sets the value of any attribute of the custom tag
            (1) (ex) util/modules.isml
        b) Isml that specifies what happens when an attribute is passed
            (1) (ex) breadcrumbs.isml
        c) Isml that invokes the custom tag
            (1) (ex)pt_storelocator.isml


storelocator/pt_storelocator.isml
util/modules.isml
components/breadcrumbs.isml
tag invocation on
next slide

```
<html ...>
    <isinclude template="util/modules">
    <head>
        ...
    </head>
    <body>
        ...
            <isbreadcrumbs ...>
        ...
    </body>
</html>
```

```
<ismodule
    template="components/breadcrumbs.isml"
    name="breadcrumbs"
    attribute="bctext1"
    attribute="bcurl1"
    attribute="bctext2"
    attribute="bcurl2"
> ...
```

```
...
<isif condition="${pdict.bcurl1 != null}">
    ...
    <a href="${pdict.bcurl1}" title="${pdict.bctext1}">
    ${pdict.bctext1}</a>
</isif>
```

2.

V. 5.4 conditional statements and loops

A. Conditionals

1. Syntax:

```
<isif condition"${[isml exp]}">
        Do something if true
<iselseif condition="${[another isml exp]}">
        Do something else if this is true and previous exp is false
<iselse>
        Do this if all above are false
</isif>
```

B. Loops

1. <isloop> - Loop thru elements of a specified collection or array
    a) Support tags:
        (1) <isbreak> - terminate loop unconditionally
            (a) If loop is nested, terminates only inner loop
        (2) <isnext> jumps to next element in list of iterator(does not restart loop iteration)
            (a) Only affects inner loop
            (b) If last or empty, loop is terminated
        (3) <iscontinue> - works like typical continue keyword
    b) <isloop

```
        Iterator || items = " [exp]"
        [ Alias || var ="[exp]" ]
        [ status = "[var name]" ]
        [ begin = " [exp]" ]
        [ end = "[exp]" ]
```

[step = “[exp]”]
>
Do something in loop
</isloop

c) Attributes and usage:
   (1) <u>items</u> || iterator- expression returning an obj to iterate over
      (a) Synonymous with “iterator”
   (2) <u>var || alias</u> - name of var referencing the object in the iterative collection
      (a) Current item
   (3) <u>begin</u> - expression specifying a begin index (x) for the loop
      (a) If x > 0, then <isloop> skips the first x items
         (i) Starts looping at begin index
      (b) If x < 0, then <isloop> is skipped
   (4) <u>end</u> - expression specifying the end index (inclusive)
      (a) If end < begin index, then loop is skipped
   (5) <u>step</u> - expression specifying the step used to increase index
      (a) Ie number to increment by
         (i) If step < 1, then 1 is used
   (6) <u>status</u> - name of variable referencing loop status object, has several properties
      (a) Count - number of iterations (starts at 1)
      (b) Index - current index into the set of items while iterating
      (c) First - true if count = 1, ie first item in list
      (d) Last - true if its the last item
      (e) Odd - true if count is odd
      (f) Even - true if count is even

I. Module 6 Content Slots

A. <u>Content slots</u> - area on page where merchant defines content to display based on qualifiers and/or rules
   1. Context of slots (3 types)
      a) Global slots - appear on any page
      b) Category slots - appear on category specific pages since they depend on the category ID
      c) Folder slots - appear in content library folders dependent on folder Id
   2. Used to show different types of content
      a) 1 or many products selected by merchant
      b) Category attributes (images or other visual)
      c) Content assets from content library
      d) Static html and images from the static library
   3. Many rules that drive content slots, some of them are:
      a) Marketing Campaigns
      b) Ranks
      c) AB tests
      d) Customer groups

B. Content slots VS content assets

1. Content assets - reusable elements that DO NOT HAVE qualifiers
    a) located in content module
2. Content slots - marketing tool
    a) Controlled by campaigns
        (1) Which have *qualifiers* that affect appearance
            (a) start/end dates
            (b) Customer groups
            (c) Source codes
            (d) Coupons
            (e) Ranks
    b) 3 Steps to create (3 steps, collaborative effort bet dev and merchant)
        (1) Developer inserts <isslot> tag in a template in the location where slot should appear
        (2) Developer creates a rendering template for the slot, which defines how the slot data is presented
        (3) Merchant create a configuration for the slot in BM
    c) Creation / configuration of content slots(in depth)
        (1) Developer creates a content slot in ISML using <isslot>
            (a) Tag should be exactly where it should be displayed
                (i) Global slot
                    (a) <isslot id="header_banner" description=".." context="global" />
                (ii) Category slot
                    <isslot id="catgory_top_featured"
                        context="category" description="..."
                        context-object="${pdict.ProductSearchResult.category}
                    />
                (iii) Folder slot:
                <isslot id="fldr-landing-slotbanner"
                context="folder" description="Large Folder Landing Banner"
                context-object="${pdict.ContentSearcResult.folder}"/>
                (iv) When template is saved, new content slot will appear in list of slots in BM -> Online marketing -> Content Slots
                    (a) Platform automatically scans any template for <isslot>
        (2) Rendering template
            (a) Content slots will display 1 type of content slots, out of the 4 possible types
            (b) Developer creates a rendering template that takes into account type of content, how many objects to display and CSS styling
            (c) (ex) header_banner uses htmlslotcontainer template as a rendering template

(i)
```
<iscache type="relative" hour="24"/>
  <div class="htmlslotcontainer">
        <isif condition="${slotcontent != null}">
         <isloop
                items="${slotcontent.content}"
                  var="markupText">
        <isprint value="${markupText.markup}"
                encoding="off"/>
        </isloop>
</isif>
</div>
```
(a) <isprint encoding=”off”/> - encoding off enables html snippet to be generated without encoding

(d) Using slot content and isprint in rendering template
  (i) _SYSTEM_Slot-Render - system pipeline in the core cartridge that renders ALL slots
    (a) Uses slot configuration that the merchant creates
    (b) Provides all configuration information to rendering template by: TopLevel.global.slotcontent constant
      (i) Only slot rendering templates get data from this context

(3) Creating content slot configuration (merchant task)
  (a) BM -> Online Marketing -> content slots
  (b) Locate specific slot created by Developer
    (i) Can select existing configuration or create new
    (ii) This is where the <isslot> tag scan comes in
  (c) Select content type
    (i) Product
    (ii) Html
  (d) Product type has field for Product ids
    (i) Template field for templates that display products
  (e) Html type - html text area displays, enter html content
    (i) Template field only have templates that display html
  (f) Template menu - shows all possible rendering template available in all cartridges in cartridge path for current content type
    (i) Located in specific folder slots directory
      (a)
```
- /slot/
        - categories
        - html
        - content
        - product
```
    (ii) Html type gets content from /slot/html/

(iii) Merchant can re use an existing template or use the new one created by developer

(g) Configure a schedule, default or marketing campaign

II. 6.2 Using content link functions

A. DW uses attributes of type html in content assets, content slots(html content type),product descriptions, etc.

1. Can also add attributes of type html to any system object
2. These attributes are represented by the class: dw.content.markupText

B. NOTE: **DO NOT HARDCODE LINKS**

1. They are instance specific, would need to change them with every replication
2. Use Content Link Functions -
    a) $staticlink$ - create static link to image
    b) $url()$ - creates an absolute url that retains protocol of the outer request
    c) $httpUrl()$ - absolute url with http protocol
    d) $httpsUrl()$ absolute url with https protocol
    e) $include()$ - remote include call
        (1) Used for caching

Module 7: DW Script

A. Server side language

B. Based on JS (ECMAscript)

1. Implements ECMA - 262 and ECMA - 357 standard
    a) Aka ECMA for XML
    b) Aka E4X
2. Supports all JS extensions by Mozilla(j.s.l.7) with type specification (optional)

C. Use DW script to access data about system: products, catalog, prices, etc.

D. Written in pipelines inside decision nodes and in ISML within <isscript> tags or as expressions within ISML templates (e.g., <isif condition=”${!empty(product)}”>...)

1. Used extensively in script pipelets

I. DW Script API

A. API packages - DW script API is organized into packages(like java)

1. Inheritance is **not** possible from these classes or packages when you create a script

B. TopLevel package is default package (Java.lang)

1. No need to import
2. Standard ECMA classes and extensions
    a) Error
    b) Date
    c) Function
    d) String
    e) Math
    f) Number
    g) Xml
3. Many common variables and constants used in pipelines and Scripts
    a) Constants
        (1) PIPELINE_NEXT & PIPELINE_ERROR
            (a) Indicate the result of a script pipelet and determines which exit the pipeline takes after execution

b) Properties
  (1) Customer, Request, Session
    (a) Provides access to current customer, session, request

4. NOTE: some packages and classes end in *Mgr*
  a) These retrieve instances of business object related to the package they belong to
    (1) ex) ProductMgr.GetProduct([id])
      (a) Gets a product using a unique identifier
      (b) Returns a product instance
      (c) which can be used to find information about a product

C. Ecommerce API packages
  1. dw.campaign - for campaigns and promotions
    a) Classes
      (1) PromotionMgr
      (2) Campaign
      (3) SourceCodeGroup
      (4) Promotion
      (5) Etc
  2. dw.catalog -for catalog, product, price book, etc.
    a) Classes
      (1) CatalogMgr
      (2) Category
      (3) Product
      (4) Recommendation
      (5) PriceBook
      (6) Etc
  3. dw.content - for non product content management
    a) Classes
      (1) ContentMgr
      (2) Content
      (3) Folder
      (4) Library
      (5) Etc
  4. dw.customer - for customer profile and account
    a) Classes
      (1) CustomerMgr
      (2) Customer
      (3) Profile
      (4) ProductList
      (5) OrderHistory
      (6) Etc
  5. dw.order - for orders, including: basket, coupons, line items, payment, shipment
    a) Classes
      (1) Basket
      (2) Order
      (3) ProductLineItem
      (4) ShippingMgr
      (5) TaxMgr

            (6) Etc

D. Generic API Manager
    1. dw.crypto - encryption services using TCA, DES, Triple DES, AES,RSA, Etc
        a) Classes
            (1) Cipher
            (2) MessageDigest
    2. dw.io - input/output
        a) Classes
            (1) File
            (2) FileReader
            (3) CSVStreamReader
            (4) XMLStreamReader
            (5) Etc
    3. dw.net - networking
        a) Classes
            (1) FTPClient
            (2) HTTPClient
    4. dw.object - system base classes and custom objects
        a) Classes
            (1) PersistentObject
            (2) ExtensibleObject
            (3) CustomObjectMgr
            (4) Etc.
    5. dw.rpc - web services related API
        a) WebReference
        b) Stub
    6. dw.system- system functions
        a) Site
        b) Request
        c) Session
        d) Logger
    7. dw.util - similar to the java.util
        a) API:
            (1) Collections class
            (2) Maps class
            (3) Calendar class
    8. dw.value - immutable value objects
        a) Classes
            (1) Money
            (2) Quantity
    9. dw.web - web processing
        a) Classes
            (1) URLUtils
            (2) Forms
            (3) Cookie
            (4) HttpParameterMap
            (5) Etc

E. Using DW script in ISML

1. Embed DW script in ISML with <isscript>
   - a) Can fully qualify all classes OR import any packages at top of script

II. 7.2 Script pipelets

- A. DW script files have .ds extension
- B. Stored in /cartridge/scripts/ dir
- C. Scripts (like pipelets) can have input/output params
  1. These are configured within script text comments
     - a) (ex)

       ```
       *     @input ProductId: string comments
       *     @output Product: Object comment
       //    @[param_type] [param_name] : [Data type] [string comments]
       ```

       - (1) Use TopLevel package classes, strings, numbers objects, etc
       - (2) Can specify other data types, but they must be fully qualified
         - (a) ex) dw.catalog.Product
- D. Importing
  1. To access dw script packages and classes (other than toplevel) you must:
     - a) Import them
       - (1) Package import
         - (a) importPackage (dw.system) OR require()
       - (2) Import single class
         - (a) importClass('dw.system.logger')
       - (3) Import custom script from same cartridge:
         - (a) importScript('common/libJson.ds')
       - (4) Access script in another cartridge(must specify it)
         - (a) importScript(" [cartridge name] : [folder/] utilities.ds");
     - b) Or you can fully qualify access to a specific class and avoid importing the package or class
- E. Script and cartridge path relationship
  1. Scripts are NOT searched using cartridge path
     - a) Even though pipelines can reference scripts in the same cartridge OR from dependant cartridges
  2. Scripts are expected to be in the cartridge of the current pipeline (unless otherwise stated in the ScriptFile configuration property)
     - a) Example configuration of properties view
       - (1) Configuration -

         -OnError - PIPELET_ERROR
         -ScriptFile - solutions:product/GetProduct.ds
         -TimeOut -

         - (a) Will need to add the 'solutions' cartridge to the cartridge path so the script is found
           - (i) Path order is not important for scripts (but still is for pipelines and templates!)
- F. Script Log Output
  1. Every script pipelet used in a pipeline comes with a default dictionary output property called: ScriptLog - {type string}
     - a) Write to script log with:
       TopLevel.global.trace(msg:string, params: Object …)

            (1) Uses Java MessageFormat
            (2) NOT RECOMMENDED
        b) USE THIS INSTEAD:
            (1) Dw.system.logger API to write to log files

G. Calling a script with JS controller
    1. Script can be invoked by using <u>require</u>
        a) Var myModel = require('~/cartridge/scripts/myModel')
           Var co = myModel.doJob(takeObj)

III. 7.34 Resource API and Resource Bundles
  A. Use resource bundles / API to avoid hardcoding text strings
    1. Titles, labels, msgs, buttons, fields, names should all be externalized using bundles(aka properties files
  B. Avoid duplicate ISML templates per locale-specific templates
  C. Resource bundles have a .properties extension
    1. Contain hardcoded strings to be used in ISML templates
    2. SG bundles loosely named by the functional area where the strings are used
  D. NOTE: properties files can be suffixed: Bundlename_<<locale_id>>.properties
    1. Locale_id stands for locale term *other than the default locale*, example
        a) "De","en", (or plus country: "de_DE", "en_GB"
  E. Resource bundles contain key=value pairs
    1. (ex) account.properties
           account.header=My account
           account.sendfriend=Send to Friend
           etc
    2. Key might be compound, key.subkey
    3. Value is hardcoded string that uses Java MessageFormat syntax to implement parameter replacement
    4. Stored within /templates/resources/
    5. Strings from bundles are accessible via:
        a) dw.web.Resource.msg(key: String, bundleName: String, defaultMessage: String)
            (1) ${Resource.msg('account.header','account', null)
                (a) Account.header => localized string name,
                (b) Account => name of properties file, may be overridden by another cartridge in path
                (c) Null => default message, means key itself(account.header) will be used if no bundle is found
                    (i) Can put a default msg to show on failure
        b) Can use .msgf() to add parameters
            (1) dw.webResource.msgf(
                'singleshipping.wishlist',
                'Checkout',
                Null,
                owners.get(addressKey).profile.firstName
)}
            (2) Paired with checkout.properties
                singleshipping.wishlist = {0}\'\'s Wishlist

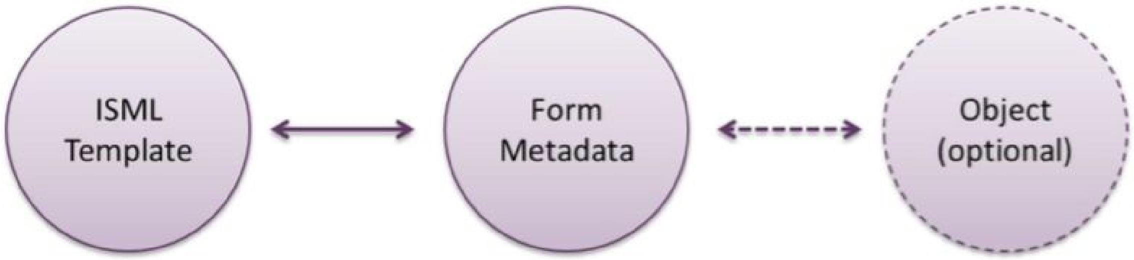
(3) Will add [owners.get(addressKey).profile.firstName] as a dynamic variable

Module 8 Forms Framework

A. Forms framework use to control how consumer entered values are validated by app, rendered in browser, and possibly stored on server

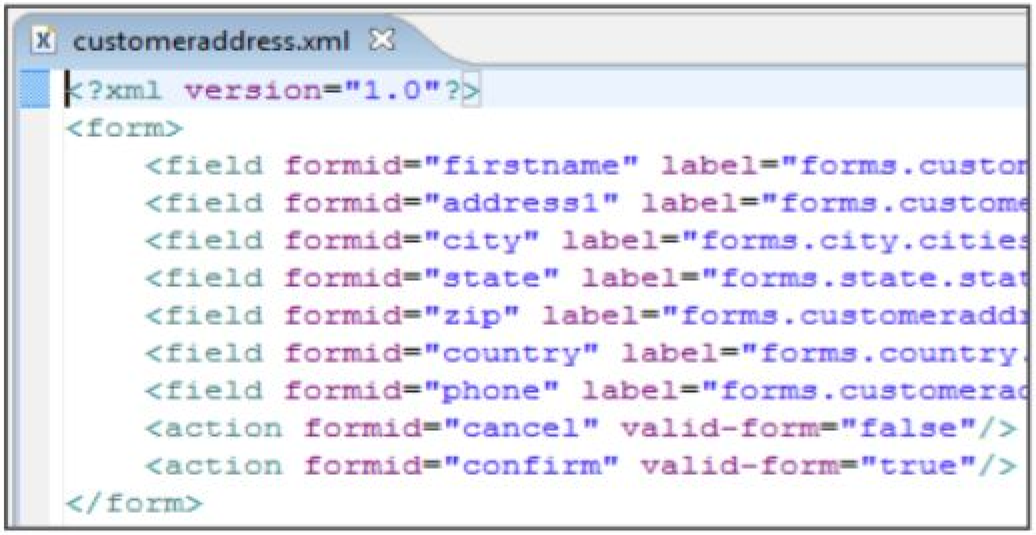
B. To use forms framework, NEED the following files
   a. Xml form to define and store metadata
   b. pipeline(controller) that will validate and process form
   c. Properties file with externalized form labels and error messages
   d. ISML which will display form to user

C. Three objects that interact with DW forms
   a. Xml metadata file in /cartridge/forms/default
      i. Describes fields, labels validation rules, and actions that apply when field is used in an isml
   b. Isml uses form metadata fields and actions to show html to user
   c. object(optional) represents single system or custom object in pdict
      i. Can be used to prefill the metadata file as well as to store submitted form data to DB



   d.

D. Examples
   a. Given this meta data file:

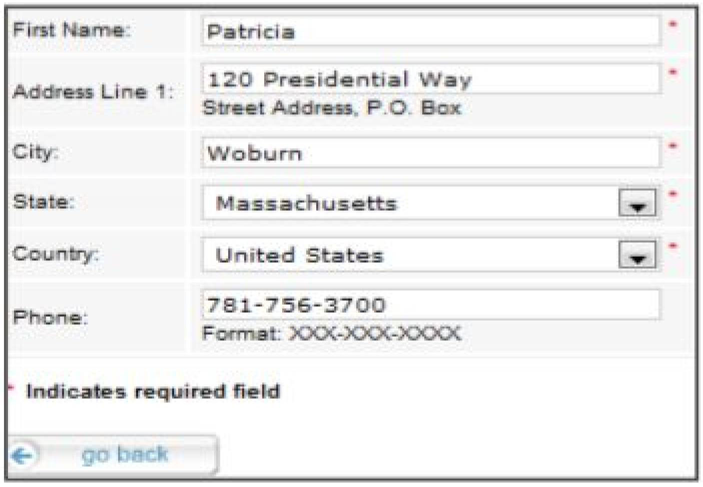


customeraddress.xml

```
<?xml version="1.0"?>
<form>
    <field formid="firstname" label="forms.custor
    <field formid="address1" label="forms.custome
    <field formid="city" label="forms.city.cities
    <field formid="state" label="forms.state.stat
    <field formid="zip" label="forms.customeraddr
    <field formid="country" label="forms.country.
    <field formid="phone" label="forms.customerad
    <action formid="cancel" valid-form="false"/>
    <action formid="confirm" valid-form="true"/>
</form>
```

      i.

b. You can create this ISML template whose fields depend on the data from the form metadata

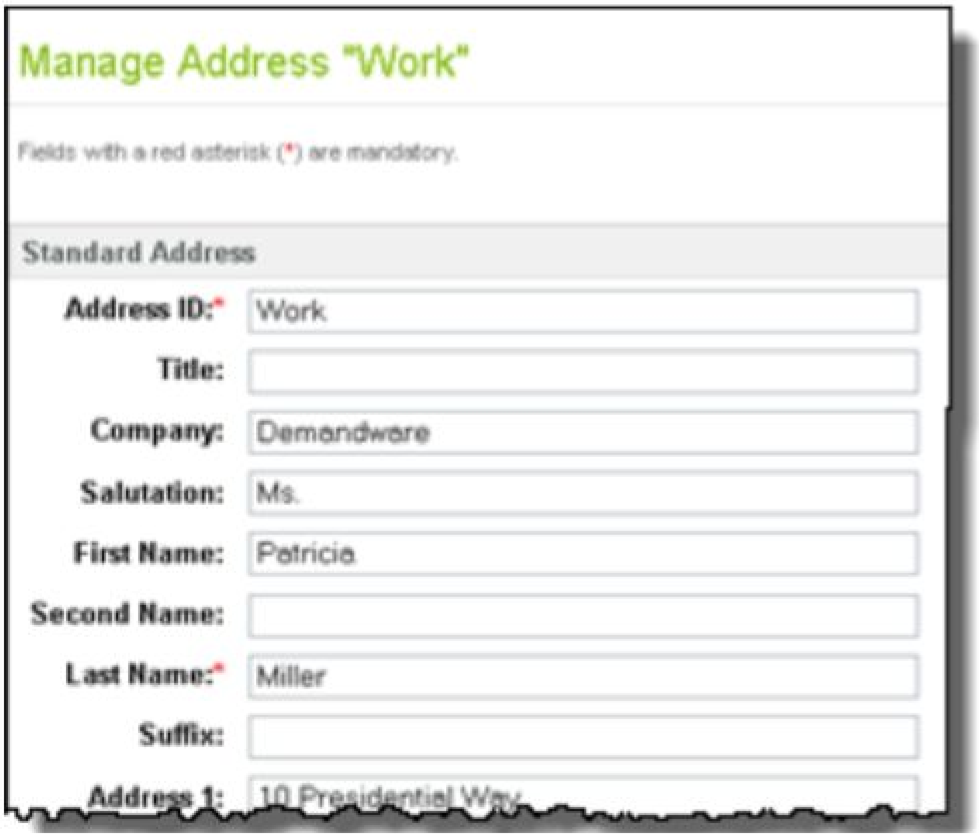


First Name:	Patricia *
Address Line 1:	120 Presidential Way * Street Address, P.O. Box
City:	Woburn *
State:	Massachusetts *
Country:	United States *
Phone:	781-756-3700 Format: XXX-XXX-XXXX

* Indicates required field

   i.

c. optionally , pdict object containing data from DB can be bound to form metadata file and allow pre filled data



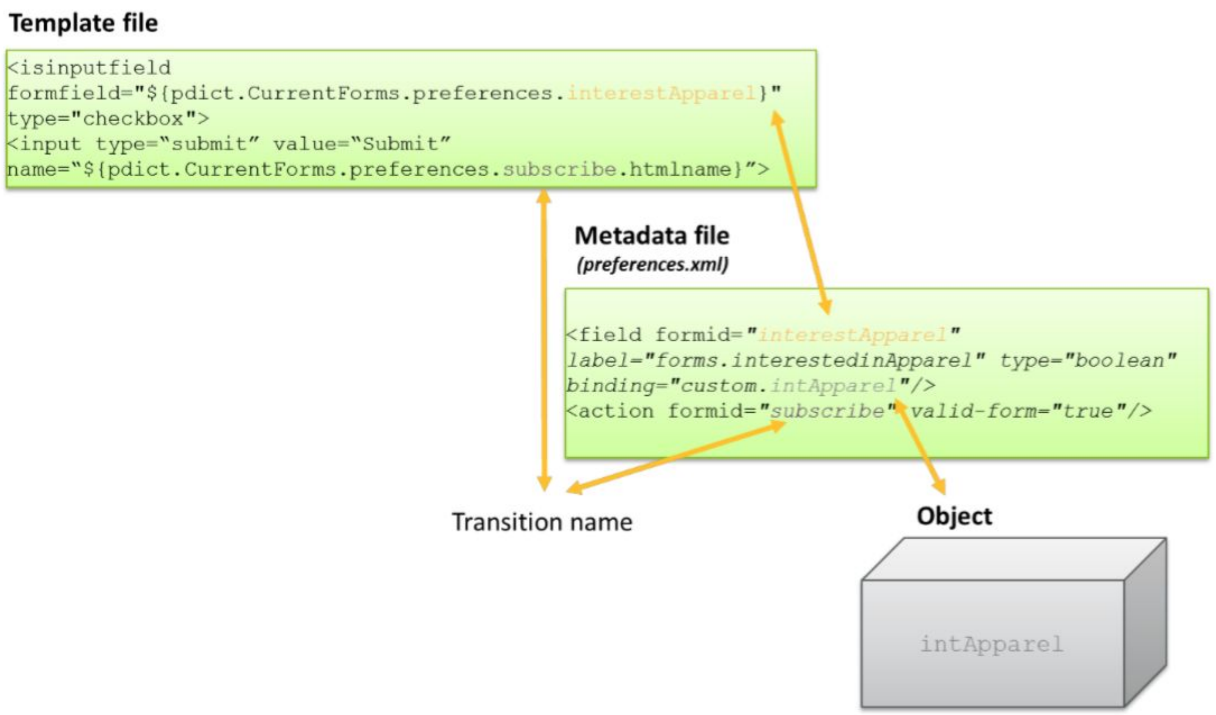
Manage Address "Work"

Fields with a red asterisk (*) are mandatory.

Standard Address

Address ID:*	Work
Title:	
Company:	Demandware
Salutation:	Ms.
First Name:	Patricia
Second Name:	
Last Name:*	Miller
Suffix:	
Address 1:	10 Presidential Way

d. Template file example:



```
<isinputfield
formfield="${pdict.CurrentForms.preferences.interestApparel}"
type="checkbox">
<input type="submit" value="Submit"
name="${pdict.CurrentForms.preferences.subscribe.htmlname}">
```

```
<field formid="interestApparel"
label="forms.interestedinApparel" type="boolean"
binding="custom.intApparel"/>
<action formid="subscribe" valid-form="true"/>
```

E.

I. **8.1 XML metadata file**

A. Contains following xml elements:

1. <u>form</u> - **required** - top level tag that contains all other elements inside it
    a) <form> … </form>
2. <u>field</u> - **required**- defines data field with many attributes, **defined below**
3. <u>Options</u> - use as a child element inside a field to pre-fill multiple options like month, days, etc.
4. <u>Option</u> - use as a child element inside an options element to specify a single option
5. <u>action</u> - **required** - defines possible action the user might take on the form
6. <u>Include</u> - Allows inclusion of one form metadata definition in another
7. <u>List</u> - allows inclusion of several items(eg collection of address) as a single field
8. <u>Group</u> - allows group of elements to be invalidated together

B. <u>Field</u> elements may have the following attributes

1. <u>formid -</u> **required** - unique id to identify the fields for ISML template and pipeline
2. <u>Type</u> - **required** - data type, **described below**
3. <u>Label</u> - usually a key to an externalized string in forms.properties resource bundle
4. <u>description</u> - description for field might be used in tooltips
5. <u>min-length,max-length</u> - restricts the field length for data entry
6. <u>min,max</u> - valid range for integer, number, and dates
7. <u>range-error</u> - message shown if value provided does not fall within specified range
8. <u>regexp</u> - reg expression for string fields (email, phone, zip, etc)
9. <u>parse-error -</u> msg shown when data entered does not match the regex, usually a key to an externalized string

10. mandatory - field is required via server-side validation when true
11. missing-error - message shown if the primary key validation error is generated in a pipeline
12. value-error - shown if an element is invalidated in a pipeline
13. Binding - used to match field element to a persistent object attribute
14. Masked - specify number of characters to mask
15. Format - format for display of dates, numbers, etc
16. whitespace - specify whitespace handling (none or remove)
17. timezoned - optional flag for date objects (true or false)
18. default-value - pre defines a value for a field
19. check-value -value when field is checked in a form
20. uncheck-value - value when field is unchecked in form

C. **Field** element **type** attribute can have the following options:

1. string - use for text data
2. integer - use for numeric data like days, months
3. number - use for quantity fields
4. boolean - use with multiple-choice fields
5. Date - use when timezoned or format are needed for dates

D. Example metadata file:

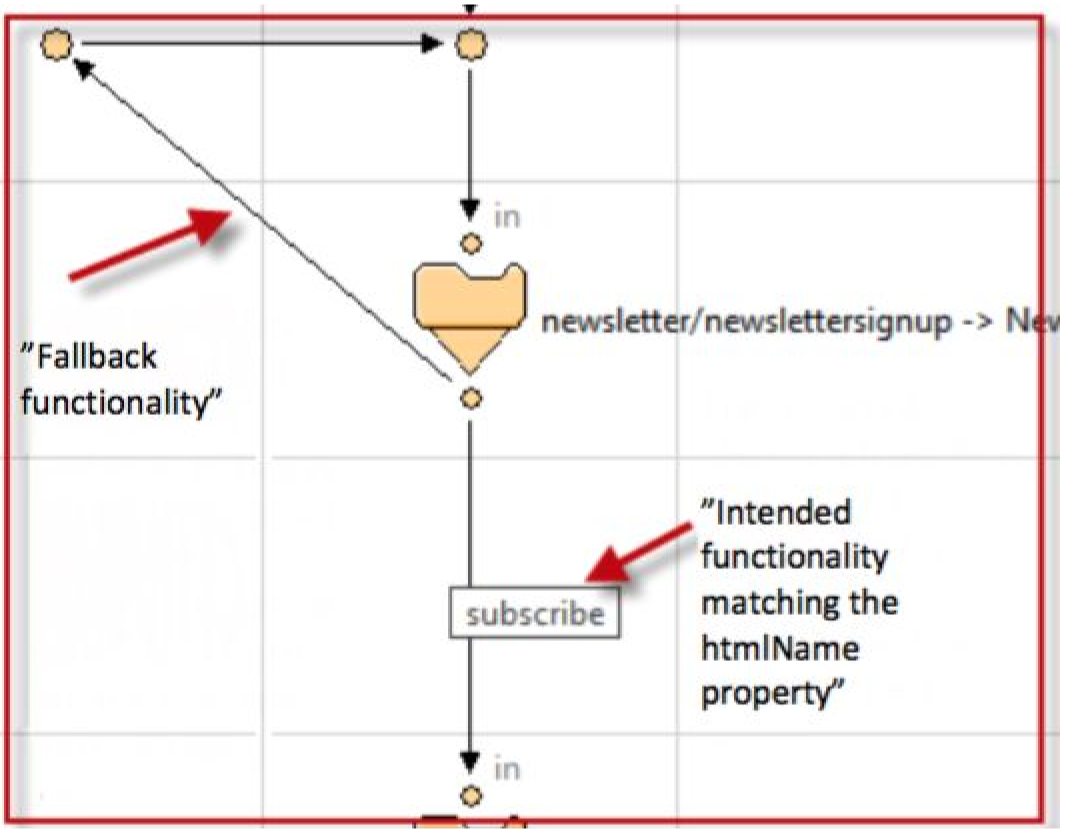
```
<?xml version="1.0"?>
<form>
        <field formid="fname" label="forms.contactus.firstname.label" type="string"
        mandatory="true" binding="custom.firstName" max-length="50"/>

        <field formid="lname" label="forms.contactus.lastname.label" type="string"
        mandatory="true" binding="custom.lastName" max-length="50"/>

        <field formid="email" label="forms.contactus.email.label" type="string" mandatory="true"
        regexp="^[\w-\.]{1,}\@([\da-zA-Z-]{1,}\.){1,}[\da-zA- Z-]{2,6}$"
        parse-error="forms.contactus.email.parse-error"
        value-error="forms.contactus.email.value-error" binding="custom.email"
        max-length="50"/>

        <action formid="subscribe" valid-form="true"/>
</form>
```

1. Fname,lname, email store the information needed to send a newsletter to a non-registered user
    - a) Fields are mandatory
    - b) Fields contain label keys that point to the cartridge/templates/resource/form.properties file
2. Email field has a regexp to define an acceptable email
    - a) Parse error key for its error message if not valid
3. Subscribe action - possible action a user can take
    - a) Valid-form = ‘true’ => means orm requires validation
        - (1) 3 required fields plus valid email format will be enforced server side

II. 8.2 ISML form template

A. <form> … </form> - defines isml template form

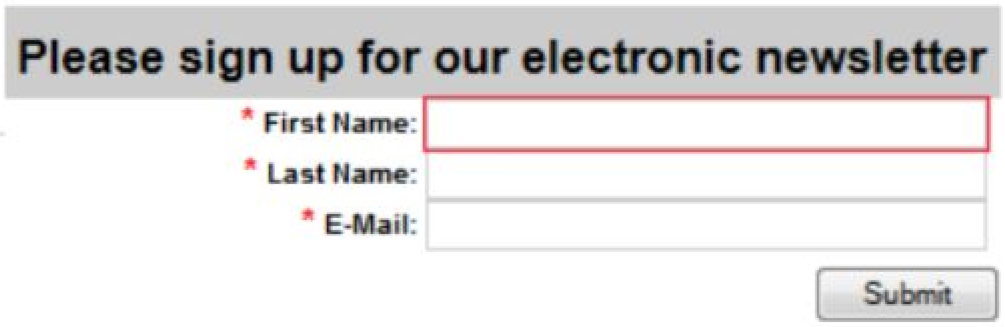
1. NOTE: possible to implement custom form action, by specifying pipeline URL. this will circumvent form framework
2. ICN - interaction continue node
    a) Specify for form action to use framework
    b) (ex)<form action = “${URLUtils.continueURL()}” method=”post”>
        (1) URLUtils.continueURL() ensures the form gets submitted to ICN, which displays the form template
        (2) This is because it works like a loop


in
newsletter/newslettersignup -> Ne
"Fallback functionality"
"Intended functionality matching the htmlName property"
subscribe
in

            (a) The ICN specifies/displays the form(newslettersignup), which has the action (<form action = “${URLUtils.continueURL()}” method=”post”>) to send user input back (continueURL) to the ICN, which validates the data and decides whether its successful (subscribe) or to fail(fallback to form)

B. To create an input field, reference the specific <u>formid</u> in the form metadata by using the object: pdict.Current forms.[form metadata file].[formid]
C. <<u>isinputfield</u>> - SG *custom* tag that facilitates the form field creation
    1. (ex)
       <isinputfield formfield=”${pdict.CurrentForms.newsletter.**fname**}” type=”input”/>
        a) Will use the **<u>fname</u>** formid from the metadata file to build html label using forms.properties(which uses form.contactus.firstname.label)
        b) /cartridge/forms/default/**newsletter**.xml
           <field formid="**fname**" label="contactus.firstname.label" type="string" mandatory="true" binding="custom.firstName" max-length="50"/>

c) Creates input field (to right of label) with necessary client side JS to enforce required field:



2. ***NOTE: This is a custom tag implemented in SG cartridge, therefore it can be modified***

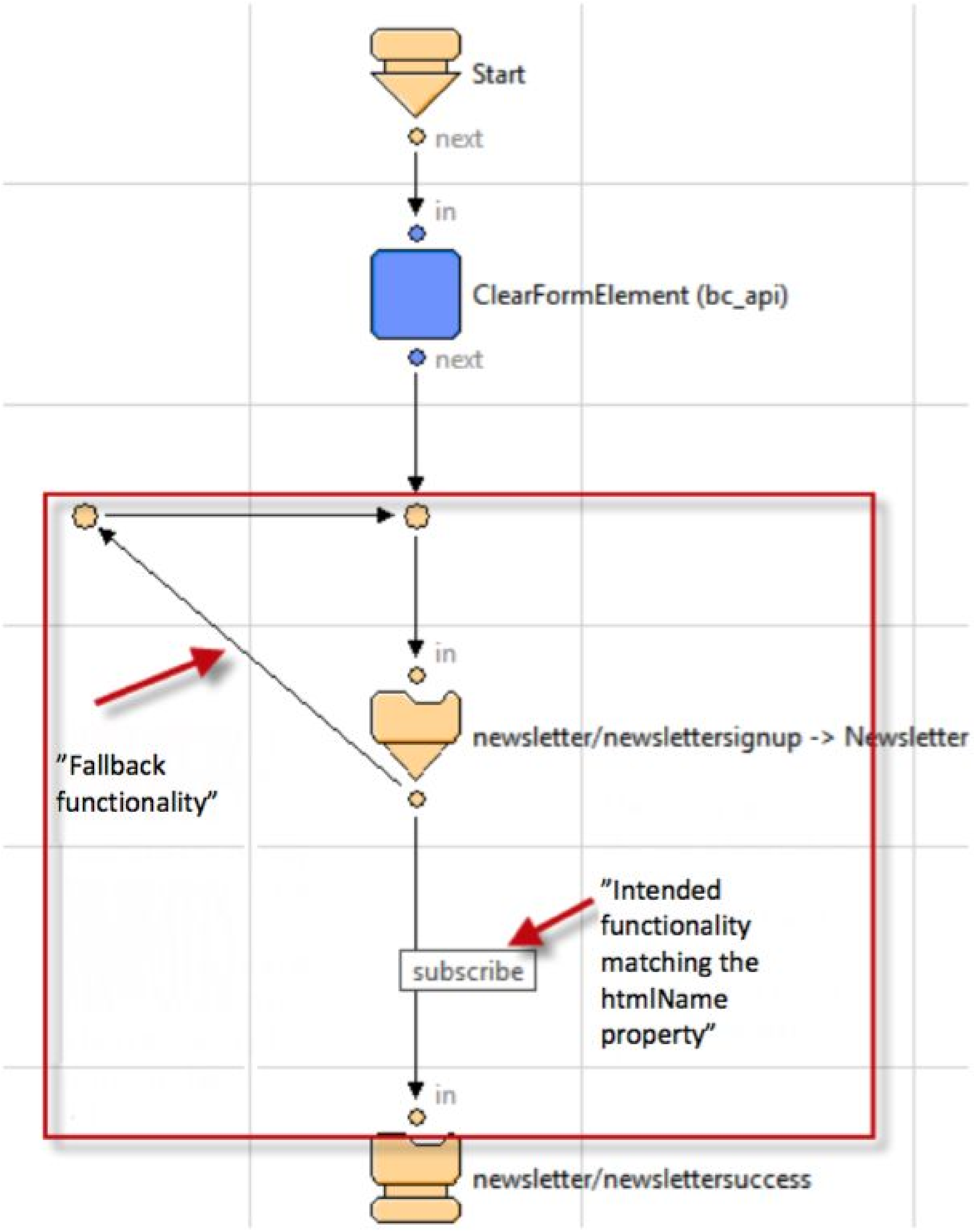
D. Final Requirement - implement button that matches the form action in form metadata
   1. <input type="submit" value="${Resource.msg('global.submit','locale',null)}" name="${pdict.CurrentForms.newsletter.subscribe.**htmlName**}"/>
      a) Creates a standard html button with name attribute that points to a specific action in the form metadata
      b) pdict.CurrentForms.newsletter.subscribe.**htmlName** refers to the **htmlName** property of the action *subscribe* in the form metadata
      c) dwfrm_newsletter_subscribe - value of property at runtime
         (1) Value specifies action for a specific form, this is necessary when the pipeline ICN determines which form action to process

III. 8.3 Form Pipeline Elements

A. DW form framework pipeline pattern steps
   1. ClearFormElement - pipelet, clears an existing form object in pdict using a specified form metadata file
   2. InvalidateFormElement - pipelet, invalidates the specified *FormElement*
   3. ICN - Shows ISML form, and then performs server side validation
   4. Transitions - which match actions from the form metadata
   5. *Next* transition - that goes back to ICN to handle validation errors

B. Steps to create form
   1. Create xml metadata file, that holds your form data
   2. Create an ISML that displays a form to the visitor
   3. Create a pipeline (with the following at minimum)
      a) Start node
      b) ClearFormElement cleans form object next time pipeline is run
      c) ICN links to the ISML file(referenced in step 2)
      d) Transition nodes to handle the following scenarios
         (1) Success - continues the pipe after the isml form, which has been successfully submitted
         (2) Fail - sends pipeline back to the form to be re-submitted

C. All together


Start
next
in
ClearFormElement (bc_api)
next
in
newsletter/newslettersignup -> Newsletter
"Fallback functionality"
subscribe
"Intended functionality matching the htmlName property"
in
newsletter/newslettersuccess

1. In the example, *next* transition node is used when validation rules are applied to allow the user to re-submit values in a form if failed the validation
2. *Subscribe* transition is for a successful submission

D. Use forms in JS controllers
    1. var myFormObject = session.forms.[metadata]
        a) Syntax for getting form object
    2. myFormObject.clearFormElement()
        a) Clears form
    3. ISML.renderTemplate('newsletter/*newslettersignup*',{
        ContinueURL: dw.web.URLUtils.https('Jnewsletter-HandleForm'),
        CurrentForms : session.forms
    });
        a) Generate form named *'newslettersignup'*
        b) Submission handled by controller named Jnewsletter
    4. var submitAction = request.triggeredFormAction.formid
        a) Finds out if submit button was pressed

Module 9 Custom Object

I. Intro
    A. <u>Custom Objects</u> (CO)- enable data to be persisted
        1. CO extend the dw data model
        2. Basically, a new table in the DB
            a) Can specify the primary key(PK) and storage attributes(like columns) that suit business needs
        3. NOTE: consider if you can use DW system object(like product, catalog, etc) before creating a new custom object
            a) * COs are best used to store static data(like configuration parameters) NOT uncontrolled amount of data(like analytics)
            b) Consider data grown and clean up of your COs
        4. DW platform governance has quotas round CO API usage and data size which will be enforced
    B. Custom Object Creator
        1. COs are created at organization level and are available for use in all SF with the organization
        2. 2 BM modules define/manage COs
            a) Custom Object Definitions - naming , PK and column specification
                (1) Admin -> Site Development
            b) Custom Object Editor - instance creation and editing
                (1) Site-<site> -> custom object -> custom object editor
        3. When defining CO specify storage scope of instances
            a) Two options
                (1) <u>Organization CO</u> - used by any site
                (2) <u>Site CO</u> - created by one site and can not be used by another
            b) The CO type can always be available to the entire organization
            c) Can specify if the CO instance is replicable - can copy from stage to production during replication process
        4. Example CO: newsletter
            a) Scenarios:
                (1) Customers can sign up, but platform does not support system table support
                (2) Subscription should be exported
                    (a) Platform should not be used for mass emails

(3) Can not use profile system object
    (a) Then only registered users would be able to sign up
(4) CO should not be replicable
    (a) Since subscription in staging should NOT be copied to production
(5) Consider clean up - after export or expiration date, cleanup batch job should run on schedule
(6) CO can also store configuration parameters to integrate with external systems
    (a) Avoids creating multiple site preferences
(7) Only replicable if settings in staging are suitable for production
(8) Can create CO with BM or programmatically
(9) First define CO data type in BM then you can create a custom object instance

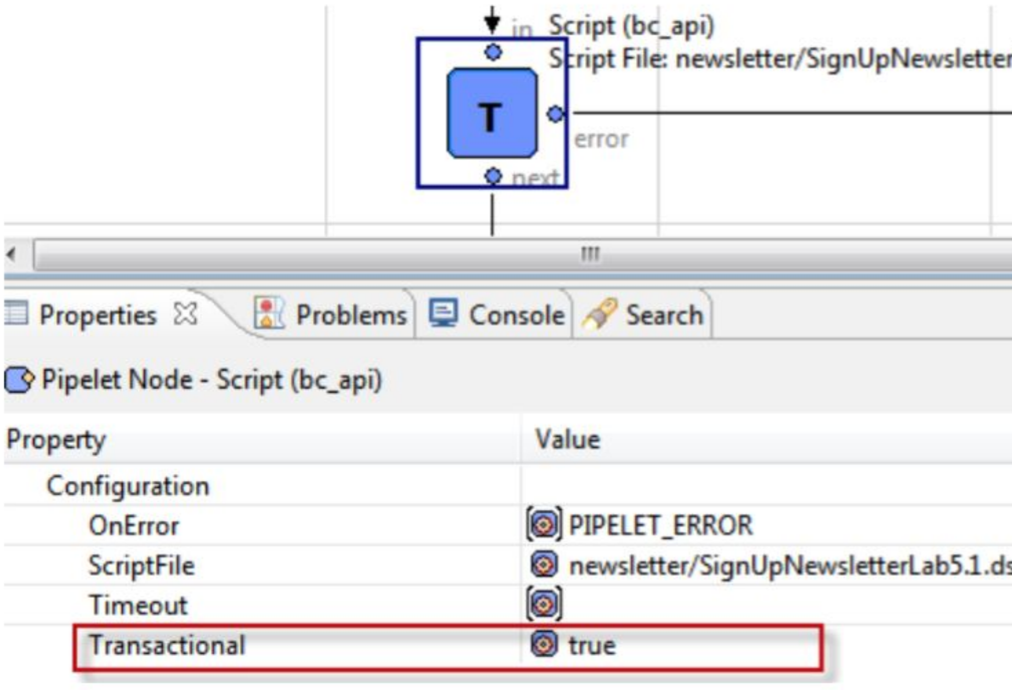
C. Creating a new custom object type in BM
   1. Admin -> Site Development -> CO definition
   2. New - new CO type
   3. Required fields:
      a) id - unique id of object type (no spaces allowed)
      b) Key attribute - unique key for CO type
      c) Data replication - CO data type will be replicable to other instances
      d) Storage scope - type is available for a site or organization
   4. Apply, creates attribute definition / attribute group tabs
   5. Attribute definition - default values created with your CO type
      a) Cannot be changed once created
      b) Create new attribute
         (1) Id field - unique name
         (2) Value type - dropdown, select data type for attribute
         (3) Apply and back
   6. Attribute group
      a) Id field - id for group
      b) Name - name or group
      c) Add attributes to group
   7. Ellipses - click to select field attributes
      a) Select attributes to add to list of checkboxes -> select
   8. Can not edit, view., add new instances of CO In editor

D. Creating a new CO instance in BM
   1. BM -> Custom Object -> CO editor -> manage CO page
   2. Select CO -> new
   3. Enter data, apply and return

II. 9.1 use DW script to create CO instances
   A. DW script API
      1. dw.object - package has the following classes
         a) CustomAttributes - attributes defined by user in BM to extends a system object or Custom Object.
            (1) Accessible via syntax: co_instance.custom.attribute
         b) CustomObject - represents an instance of a CO
         c) CustomObjectMgr - enables custom object instance creation

d) PersistentObject - enables persistentObject - enables persistent storage
e) ExtensibleObject - enables custom attributes to be added.

B. Inheritance tree
C. Object > dw.object.PesistentObject > dw.object.ExtensibleObject > dw.object.CustomObject (or dw.object.SystemObject)
    1. Means CO(dw.object.CustomObject) are persisted in DB because of PesistentObject
    2. Can have custom attributes added by an admin or developer in BM because of ExtensibleObject
    3. Classes like dw.catalog.Product and dw.system.SitePreferences (and many others) share this inheritance, Therefore they are saved in the DB and can be extended to store extra attributes
D. CustomObjectMgr.createCustomObject('NewsletterSubscription', UUIDUtils.createUUID());
    1. Enables creation of an instance of a custom object by providing the CO type and the PK
    2. UUIDUtils.createUUID() - creates a system generated unique primary key
        a) Can create own PK, assuming it will be a unique string, or else it will get an error
E. Database transaction handling
    1. Explicit - transaction is controlled via properties of transition nodes in pipeline
    2. Implicit - a transactional pipelet that automatically beings a transaction. Transition is automatically committed to DB when pipelet returns PIPELET_NEXT, OR transaction gets rolled back
        a) For implicit transaction to work pipelet that performs DB changes must have transaction property = true
            (1) Adds 'T' to pipelet



Pipelet Node - Script (bc_api)

Property	Value
Configuration	
OnError	PIPELET_ERROR
ScriptFile	newsletter/SignUpNewsletterLab5.1.ds
Timeout	
Transactional	true

        b) When executed DB transaction will occur auto. And committed implicitly at end of pipelets execution (if success)

F. Create Custom Object via script API
    1. Create CO type in BM (before creating CO programmatically)
    2. Create script that uses dw.object.CustomObjectMgr
        a) Class to create CO
    3. (ex)

```
importPackage( dw.system );
importPackage( dw.object );
function execute( args : PipelineDictionary ) : Number
{
        var co =
        CustomObjectMgr.createCustomObject("NewsletterSubscription",
        args.email);

        co.custom.firstName = args.firstName;
        co.custom.lastName = args.lastName;

        args.subscription = co;

        return PIPELET_NEXT;
  }
```

    4. NOTE: qualifiers for all custom attributes are defined in CO definition
        a) Without qualifier code will fail. Class dw.object.CustomObject does NOT have any standard attribute named: firstName, lastName, etc

G. Example implicit/explicit transaction in JS controllers/scripts
    1. Implicit -
        a)

```
Transaction.wrap(function() {
        couponStatus = cart.addCoupon(couponCode);
});
```

    2. Explicit :
        a)

```
var Transaction = require('dw/system/Transaction');
Transaction.begin();
if {
        code for the transaction
        ...
} else {
        Transaction.rollback();
        code after the rollback
        ...
}
Transaction.commit();
```

III. 9.2 Custom Logging
    A. DW logging support uses log categories and severity levels defined by apache log4j
        1. log4J supports multiple severities and categories
            a) Can capture debug msgs at different levels of granularity
    B. Levels - Debug < Info < Warn < Error < Fatal

1. If logging is enabled for certain level then it is enabled for higher levels as well (left to right)
2. * fatal / error are always on cannot be turned off

C. Can define as many levels of categories and subcategories as necessary
   1. Developer determines categorization
   2. (EX)
      a) product
      b) product.import
      c) Product.import.staging
         (1) If log enabled for a category, product, then all its subs are enabled
         (2) However, if warn is enabled on product' and debug is enabled on product import
            (a) Warn, error, fatal are logged for product and subs
            (b) Debug and info for product.import and subs

D. dw.system.Logger.getLogger() writes to custom log
   1. (ex)

```
var logger = Logger.getLogger("logFilePrefix","category" );
logger.debug(
        "Input params received in pipelet firstName: {0}\n lastName: {1}\n
        email: {2}", args.firstName, args.lastName, args.email
);
try {
    ... do something...
    return PIPELET_NEXT;
   }
catch (e)
{
logger.warn("error description: {0}", e.causeMessage );
     return PIPELET_ERROR;
   }
```

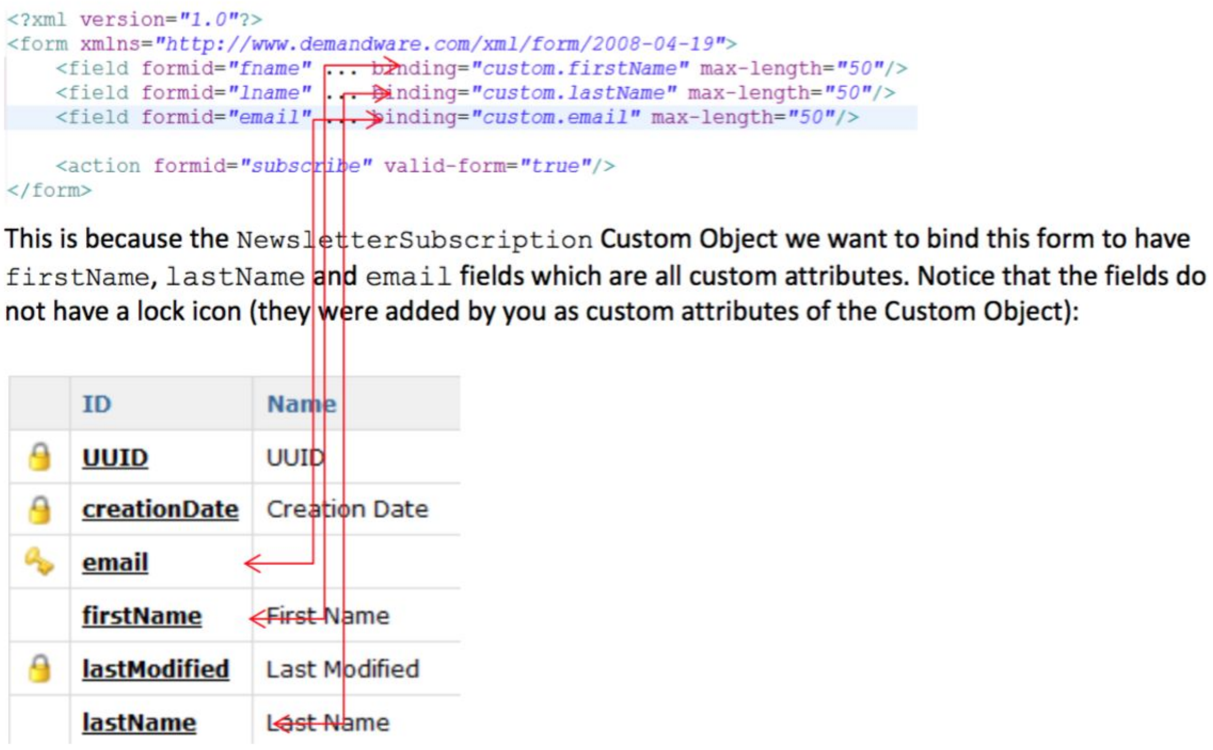
   2. Use logger object to write a message for a specific severity level: Logger.error(String msg)
      a) Use Java MessageFormat API, can specify placeholders
      b) Typically messages are NOT localized
         (1) Read internally by admins

E. Enable custom logging
   1. BM -> admin > operations > custom log settings
   2. Create log category and severity
   3. Enable
   4. Log debug to file - logs to file (up to 10mb)
      a) Usually debug/info msg or written only to memory and visible with request log tool
   5. View at admin > site development > development setup > log files

Module 10: Data binding and explicit transactions

I. Data binding with forms and objects

A. Forms Framework supports binding of persistent objects to form fields by automatically updating persistent object with form data *without* an insert statement (or DW API Reverse mechanism is also possible, ie populate a form object with data from a persistent object
B. Object bound to form must be a persistent object (system or custom) AND must be available in pdict
C. Form data must have fields with the **binding** attribute specified
formid attributes is NOT used to make the match
binding identifies what fields match the form and object



```
<?xml version="1.0"?>
<form xmlns="http://www.demandware.com/xml/form/2008-04-19">
    <field formid="fname" ... binding="custom.firstName" max-length="50"/>
    <field formid="lname" ... binding="custom.lastName" max-length="50"/>
    <field formid="email" ... binding="custom.email" max-length="50"/>

    <action formid="subscribe" valid-form="true"/>
</form>
```

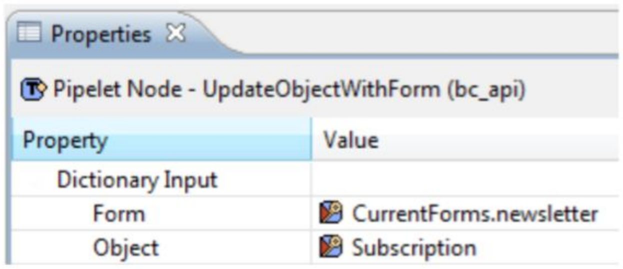
This is because the `NewsletterSubscription` Custom Object we want to bind this form to have `firstName`, `lastName` and `email` fields which are all custom attributes. Notice that the fields do not have a lock icon (they were added by you as custom attributes of the Custom Object):

	ID	Name
🔒	UUID	UUID
🔒	creationDate	Creation Date
🔑	email	
	firstName	First Name
🔒	lastModified	Last Modified
	lastName	Last Name

D. <u>Update**Object**With**Form**</u> pipelet - Updates an existing *persistent object* with data from the *form*
    1. Requires object (to update) and the form object to both be available on **pdict**
    2. Transactional by default, as the object to be updated MUST be a persistent object

3. (ex)

This example show how to define the properties of the pipelet using the `newsletter` form `NewsletterSubscription` object:



Properties

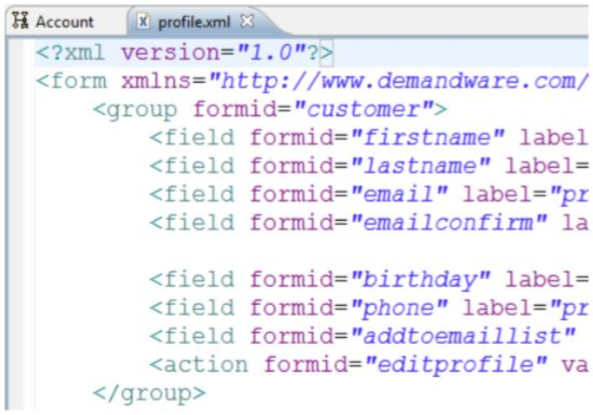
Pipelet Node - UpdateObjectWithForm (bc_api)

Property	Value
Dictionary Input	
Form	CurrentForms.newsletter
Object	Subscription

   a) Pipelet inspects CurrentForms.[form] in pdict
      (1) Tries to match every field with a binding attributes to a column in custom object
      (2) This object must be instance of CO that was placed in pdict either by:
         (a) CreateCustomObject pipelet - Creating a new instance
         (b) SearchCustomObject pipelet - retrieve an existing instance
4. If subscription is null or not instance of CO of form is not in pdict
   a) Pipelet will then fail and then transaction will rollback
5. IF successful,
   a) Then pipelet will commit transaction
6. UpdateObjectWithForm pipelet is a good way to enter new data or update existing data on object

E. Update**Form**With**Object** - updates form with data from a persistent object
1. Requires *form to update* and *object* on **pdict**
2. NOT transactional, since updated form lives on pdict scope NOT in DB
3. Form group may be updated with an object.
   a) IF bindings match just that part of the form will be updated

In the example shown, the `profile.xml` form has a `customer` group that will be updated with the existing profile data from the logged in customer:



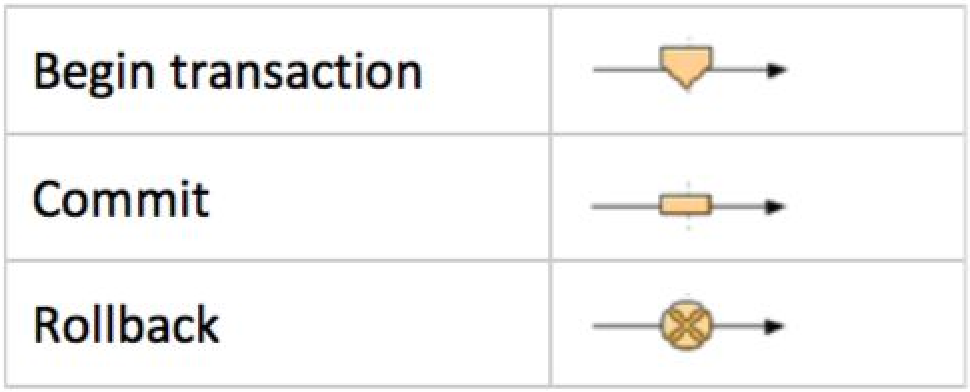
Account | profile.xml

```
<?xml version="1.0"?>
<form xmlns="http://www.demandware.com/
    <group formid="customer">
        <field formid="firstname" label
        <field formid="lastname" label=
        <field formid="email" label="pr
        <field formid="emailconfirm" la

        <field formid="birthday" label=
        <field formid="phone" label="pr
        <field formid="addtoemaillist"
        <action formid="editprofile" va
    </group>
```

II. 10.2 Explicit Database Transaction Handling

A. Use to control where/when transaction begins and ends

B. Implements at the pipeline level, by changing the transactional property of **transition** or **connectors**



Begin transaction	
Commit	
Rollback	

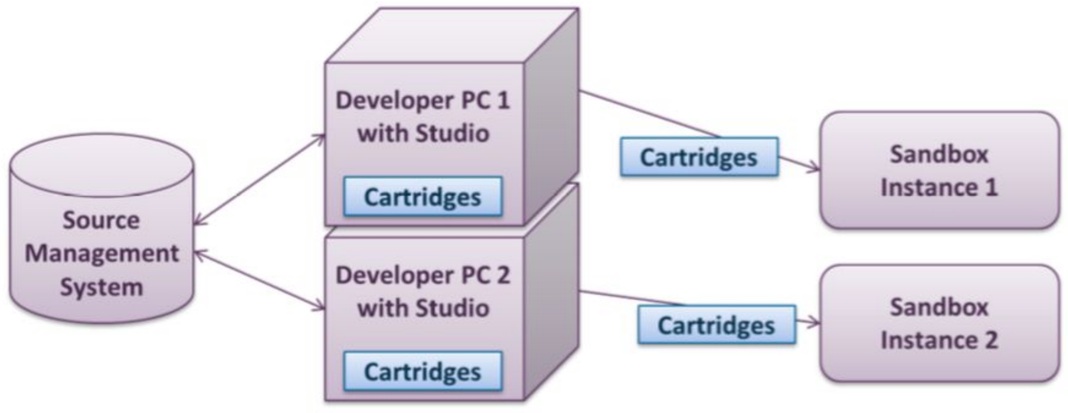
1.
2. Use this to override built-in implicit transaction to group changes that need to be part of an atomic transaction

Module 11: Site Maintenance

I. 11.1 Site and Page Caching
  A. Caching is controlled on a per page basis via ISML with <iscache> tag
    1. <iscache type='relative' hours='24' />
  B. Rules for cache tag
    1. If <iscache> occurs multiple times in a template OR its locally included template, then the shortest duration is used
    2. Caching from a **local include** affects the **including template**
    3. If NO <iscache> , then template is NOT cached
    4. Parameters
      a) type_= "relative || daily"
        (1) Relative - enables you to specify a certain period of time (Min/hours) after which the page is deleted from cache
        (2) Daily - enables you to specify an exact time when page will be deleted form cache
      b) hour = '[integer]'
        (1) indicates either caching duration or time of day
          (a) type = 'daily'
            (i) Hour value must be integer between 0-23
          (b) type='relative'
            (i) All integer values 0 or greater are valid
              (a) Default is 0, page is never cleaned from cache OR only **minute** is relevant
      c) minute = '[integer]'
        (1) Integer indicates duration or time of day
          (a) daily - must be int between 0-59
          (b) relative - all values 0 or greater are valid
            (i) Default is 0, page is never cleaned OR only **hour** matters
      d) varby ='price_promotion' (EX)

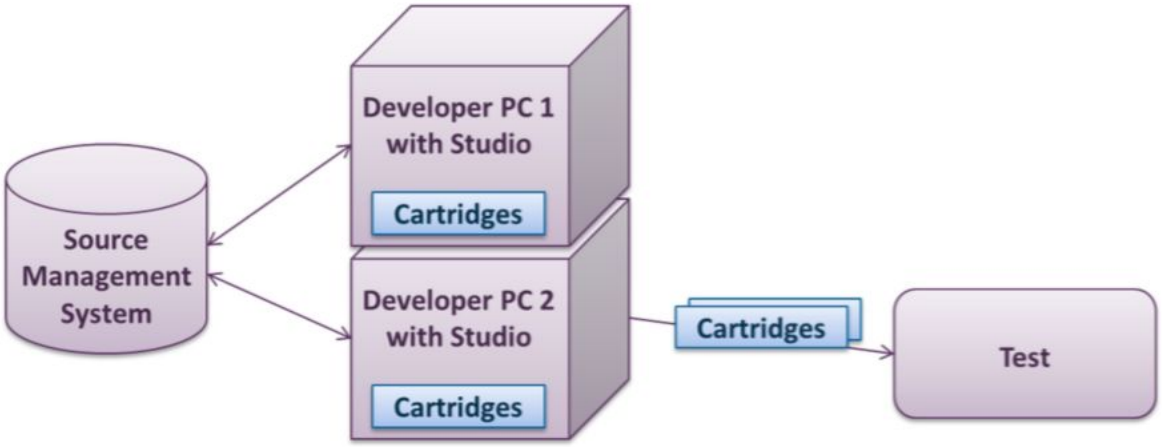
            (1) Mark page ass **personalized**; different version of the same page show different prices, promos, sorting rules, ab tests, etc. Each will be cached differently
                (a) Isml template is the same, but generated pages vary
                    (i) Caching every version of the page will benefit performance
                (b) For performance, a page should only be marked with varby IF the page is **really** personalized
    5. Frequently changing pages benefit from a **shorter** caching period.
        a) Stored pages are only invalidated and new ones pulled from server IF/WHEN:
            (1) Defined cached time is exceeded
            (2) Replication has been performed
                (a) Exception of coupons/geolocation data)
            (3) Explicit page cache invalidation triggered in BM
    6. **Best practice: disable cache in sandbox, development, staging to see changes immediately**
        a) **Production is always cached**
    7. Portion of page can be cached separately
        a) Assemble a page from snippets with different caching attributes using **remote includes.** Each part:
            (1) Must be a result of pipeline request to app server
            (2) Include with <isinclude url=”..”> or <iscomponent pipeline=”..”>
            (3) Can have different cache times or NO cache at all
            (4) **DO NOTE CACHE PAGE WITH *BUY* OR *SESSION* INFORMATION**
C. Studying page analytics to determine caching problems
    1. Site > analytics > tech reports > pipeline performance
        a) Shows caching metrics
        b) Only on production instances (not sandboxes)
        c) Caching column - if **red** then NOT cached
    2. 2 critical metrics from pipeline perspective:
        a) **Search-Show** and **Product-Show**
            (1) Used across ALL customers
            (2) Main component of most pages on DW instances
            (3) Search-Show - Avg response = 400ms
                (a) Response <= 400ms, good performance
            (4) Product-Show - Avg response = 320ms - 400ms
                (a) Response <= (320ms - 400ms), good performance
D. Page Level Caching
    1. Once the <iscache> tag is added to an ISML template the entire page will be cached for the time specified
        a) <iscache type=”relative” hour=”1” minute=”30” />
            (1) Cache is for 1 hour and 30 minutes
E. Partial Page Caching
    1. Generally, do NOT completely cache a page
        a) Only *some* parts of the page should be cached

            (1) Use **remote includes** for every part that has unique caching characteristics
            (2) Every **remote include** calls a different pipeline which generates an ISML
            (3) It's possible each template has different caching properties
        b) Syntax for **remote includes** uses *URLUtils* class to call a remote pipeline with optional parameters appending:
            (1) <isinclude url-"${URLUtils.url('Page-Include', 'cid', 'COOKIE_TEST')}">
            (2) Can also use <iscomponent>
    F. Using the SF toolkit to determine cache settings
        1. Enable cache information tool in SF toolkit
            a) Shows partial page caching implemented per page
II. 11.2 Site performance
    A. <u>Pipeline profiles</u> - BM tool provides pipeline and script performance analysis
        1. Tracks pipeline execution metrics
            a) Critical component of overall page and site load and performance
        2. BM -> Admin -> Organization -> Pipeline Profiler
            a) High level view of response times, hits, total time for a page to generate, average times, etc
        3. Look for pipelines with high average run times and high hits
            a) These are first place to improve performance
        4. While pipeline profiler runs, have access to captured script data
III. 11.3 Code replication - set and manage in BK
    A. **First** , upload a new code version to PIG staging and set code replication to occur between staging and development OR staging and production
        1. Overview:



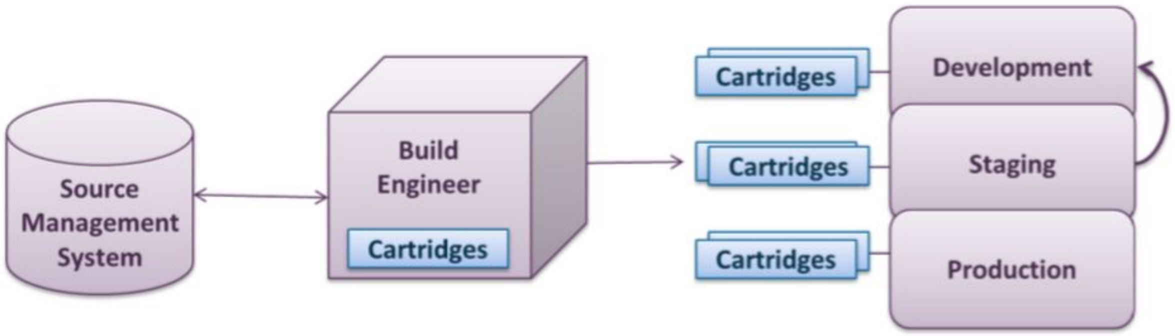
        2. Typically **source manage system** is used for code version control
            a) Each developer uses their own sandbox for developing, while checking in their code to a source management system
            b) UX studio integrates with SVN for source management
    B. **Second** , when code version is ready to upload to staging, THEN create a new code version on staging in BM -> Site -> Development -> Code Deployment

C. Third, developer uploads custom cartridge with UXS or WebDAV client using 2-factor authentication, THEN tests on SF on staging
   1. Rollback to previous version is possible
   2. For major changes use sandbox



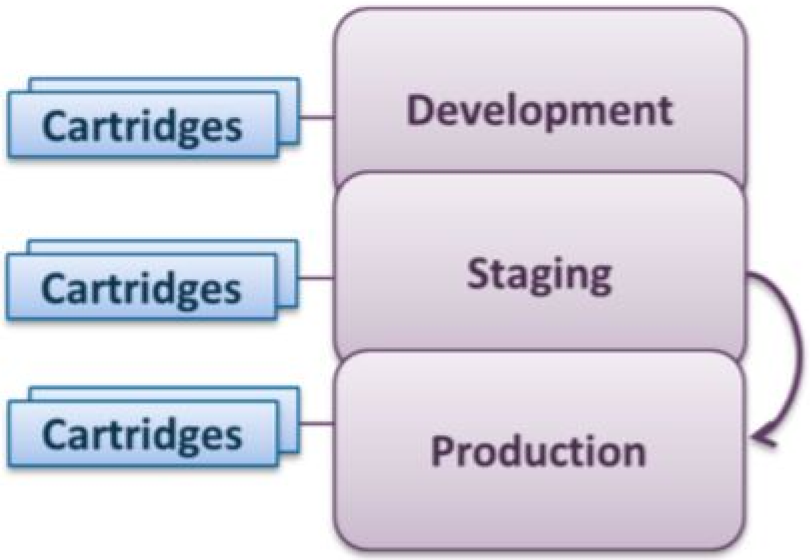
D. To test in sandbox, you will need to export site data to the global directory from staging and import it into your sandbox using *Site Import/Export* in BM

E. When testing code metadata(Site preferences, new attributes, etc) the build engineer replicates from staging -> Development:



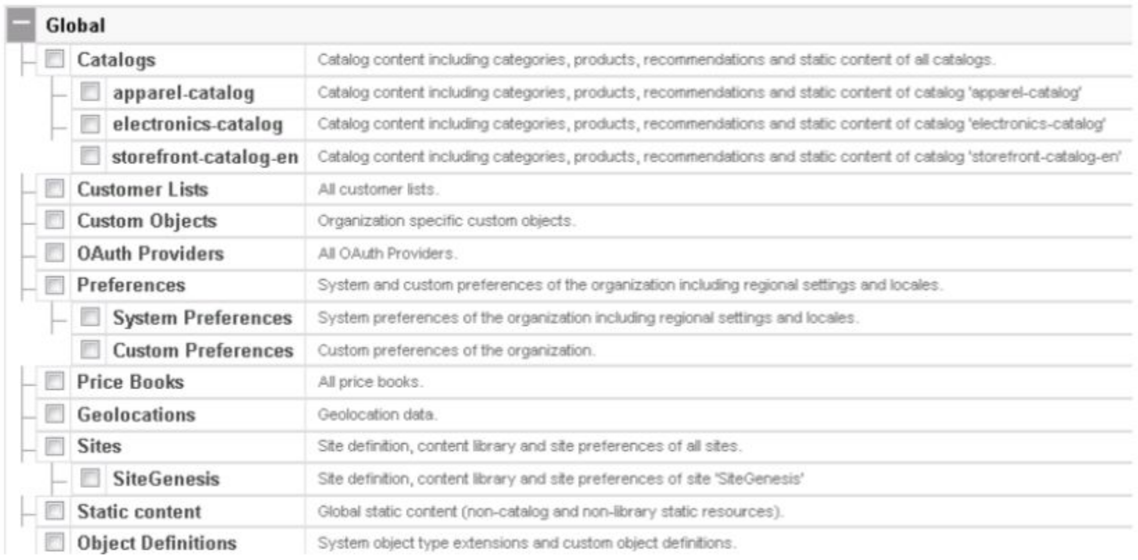
   1. Good practice for testing process without impacting production SF(eg product import feed)

F. Last step, Move code from staging to production:



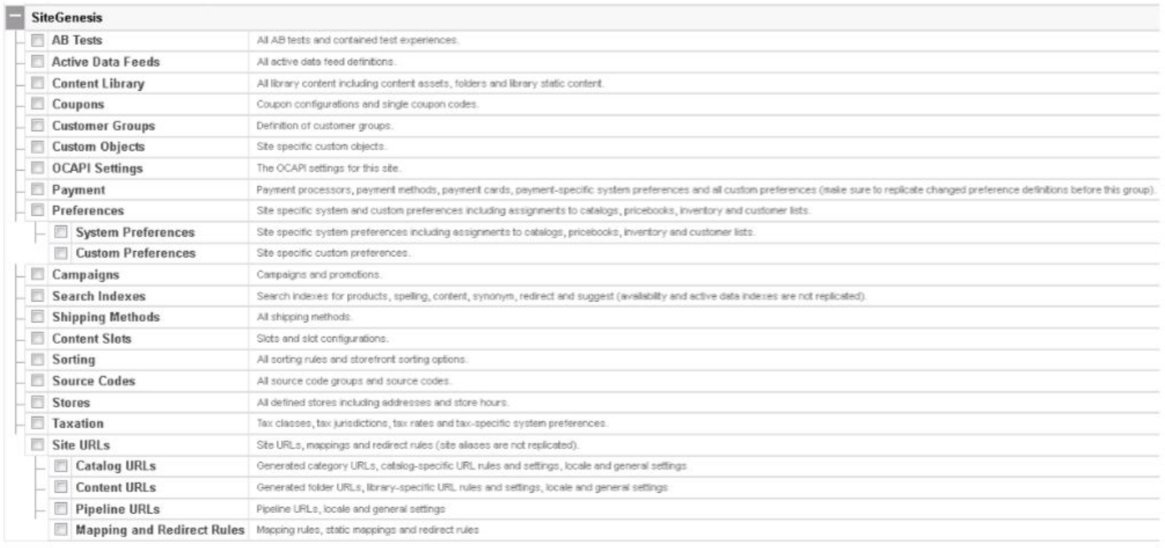
G. Full Steps to replicate code (staging -> development & staging -> production)
   1. Need BM permission for code replication
   2. Admin > Replication > *Code* replication
   3. **new** replication process
   4. Target dropdown, specify **development** or **production**
   5. **Manual** or **automatic** run=> **next**
   6. Type of replication:
      a) **Code transfer and Activation** - immediately activates new code version
      b) **Code transfer** - only transfers code(no activation)
   7. **Start** replication process
   8. **Create** to add replication process
   9. If manual, then click **start**

IV. 11.4 Data Replication
   A. Process to promote merchant edits, product and system objects from staging -> production || staging -> Development
      1. **Best practice**
         a) Stage to dev
         b) Verify data/SF
         c) Replicate stage to prod
   B. Data can be replicated granularly
      1. Organization objects (global) :



Global	
Catalogs	Catalog content including categories, products, recommendations and static content of all catalogs.
apparel-catalog	Catalog content including categories, products, recommendations and static content of catalog 'apparel-catalog'
electronics-catalog	Catalog content including categories, products, recommendations and static content of catalog 'electronics-catalog'
storefront-catalog-en	Catalog content including categories, products, recommendations and static content of catalog 'storefront-catalog-en'
Customer Lists	All customer lists.
Custom Objects	Organization specific custom objects.
OAuth Providers	All OAuth Providers.
Preferences	System and custom preferences of the organization including regional settings and locales.
System Preferences	System preferences of the organization including regional settings and locales.
Custom Preferences	Custom preferences of the organization.
Price Books	All price books.
Geolocations	Geolocation data.
Sites	Site definition, content library and site preferences of all sites.
SiteGenesis	Site definition, content library and site preferences of site 'SiteGenesis'
Static content	Global static content (non-catalog and non-library static resources).
Object Definitions	System object type extensions and custom object definitions.

2. Per site objects:



SiteGenesis	
AB Tests	All AB tests and contained test experiences.
Active Data Feeds	All active data feed definitions.
Content Library	All library content including content assets, folders and library static content.
Coupons	Coupon configurations and single coupon codes.
Customer Groups	Definition of customer groups.
Custom Objects	Site specific custom objects.
OCAPI Settings	The OCAPI settings for this site.
Payment	Payment processors, payment methods, payment cards, payment-specific system preferences and all custom preferences (make sure to replicate changed preference definitions before this group).
Preferences	Site specific system and custom preferences including assignments to catalogs, pricebooks, inventory and customer lists.
System Preferences	Site specific system preferences including assignments to catalogs, pricebooks, inventory and customer lists.
Custom Preferences	Site specific custom preferences.
Campaigns	Campaigns and promotions.
Search Indexes	Search indexes for products, spelling, content, synonym, redirect and suggest (availability and active data indexes are not replicated).
Shipping Methods	All shipping methods.
Content Slots	Slots and slot configurations.
Sorting	All sorting rules and storefront sorting options.
Source Codes	All source code groups and source codes.
Stores	All defined stores including addresses and store hours.
Taxation	Tax classes, tax jurisdictions, tax rates and tax-specific system preferences.
Site URLs	Site URLs, mappings and redirect rules (site aliases are not replicated).
Catalog URLs	Generated category URLs, catalog-specific URL rules and settings, locale and general settings
Content URLs	Generated folder URLs, library-specific URL rules and settings, locale and general settings
Pipeline URLs	Pipeline URLs, locale and general settings
Mapping and Redirect Rules	Mapping rules, static mappings and redirect rules

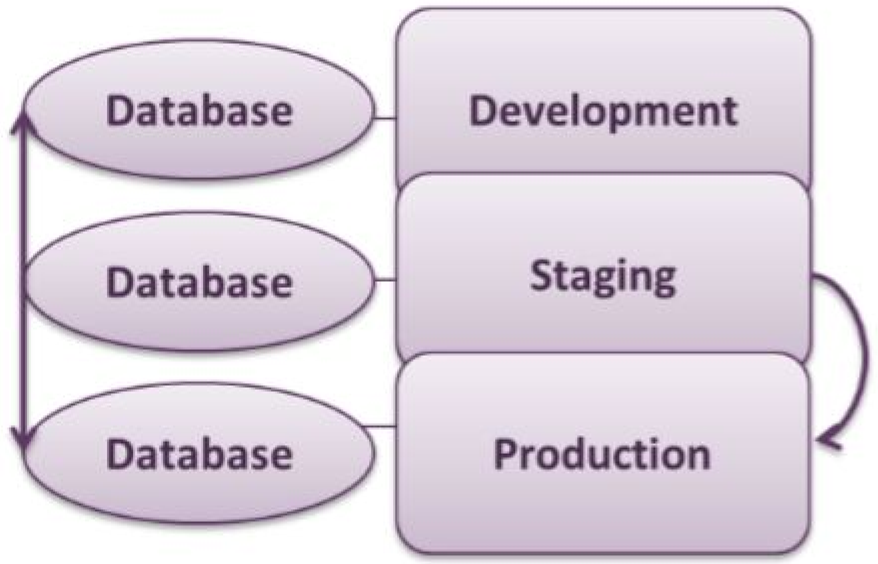
C. **Two phases of data replication**
   1. <u>Transfer</u> - long running process where data is copied from staging into shadow tables and folders on production
      a) ***NO changes in SF at this point***
   2. <u>Publishing</u> - fast process, changes in shadow tables/folders become active
      a) Page cache is purged, then new version is shown in SF

D. After Replication, a **ONE TIME** rollback (undo) is possible
   1. Reverses data to last successful replication

E. Review progress in staging logs on stage/prod instance

F. Data replication is almost identical to code replication
   1. Except you can select which data to replicate

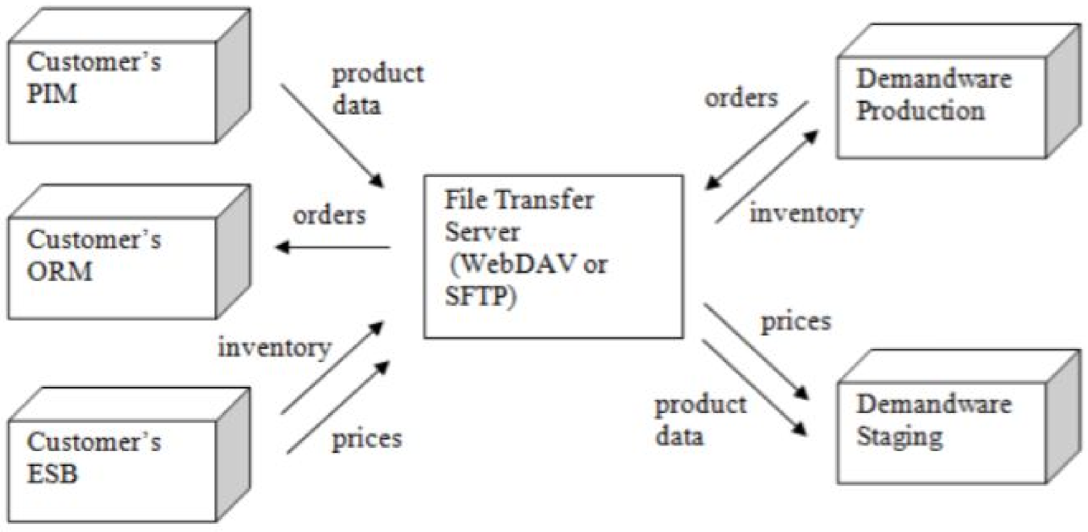
G. Like code replication, process is only allowed **one-way**, either stage -> dev || stage -> prod



H. Steps to Data replication
    1. BM admin with replication permission
    2. Admin -> replication -> *data* replication
    3. **New**
    4. Specify target - **production** or **development**
    5. Run **manual** or **auto**
        a) If auto specify time/date
    6. When to receive email, **next**
    7. Type of replication
        a) Data transfer and publishing
        b) Data transfer only
    8. **Expand** sites to **select** the site data to be replicated, **next**.
    9. **Start** to create and trigger process, **create** to add the replication product to the list , or **cancel**

Appendix A: Data Integration: Simple feeds

I. Simple feed
    A. <u>Simple feed integration(SFI)</u>- loosely couples DW and external systems by exchanging files on a file transfer server
        1. SFI supports WebDAV (HTTP || HTTPS) and SFTP for the transfer with the file transfer server
            a) WebDav HTTP - no network layer encryption - is meant for development/testing purposes only
            b) WebDAV HTTPS - SSL certificate from a DW accepted certificate authority (CA) needs to be installed at the file transfer server
        2. File Transfer Server - must be hosted by customer OR by a third party
            a) DW does NOT offer hosting file transfer servers
            b) WebDAV access to certain folders on a DW instance cannot be used for that purpose
        3. Integration flow of simple feed cartridge



B. File Format - incoming / outgoing feeds is the standard DW import/export format
    1. For XML files the schema definition (XSD files) can be downloaded from customer central
    2. **Good approach**
        a) Create a sample file to set up the data desired in BM
        b) Run an export
        c) Use exported file as template
    3. All file can be in gzip (not zip), they are automatically unzipped during processing

C. Simple feed cartridges (two of them)
    1. int_simplefeeds - implementation of generic logic
        a) Must be assigned to the SF site AND the BM site
        b) Must modify Custom_FeedJobs pipeline, it's necessary to use feed configuration custom object
            (1) DO NOT modify other code
    2. test_simplefeeds - helps troubleshoot WebDAV or SFTP connection issues
        a) Can trigger simplefeed integration jobs from SF during development
            (1) MUST NOT be assigned to site on production systems
            (2) It can be assigned to site SF on sandbox instances IF SF is password protected
    3. WebDAV - only protocol that DW supports where DW can access files external to DW. And where external systems can push files to DW
        a) ie, works in **both** directions
    4. SFTP cannot access files in DW. There is NO SFTP server in DW instances
        a) BUT DW can access an external SFTP server

D. Steps to import objects using simple feeds via DW int_simplefeed cartridge
    1. Download and import int_simplefeeds cartridge into UXS
    2. Import custom cartridges to your project code base
        a) Add int_simplefeeds to code repo (SVN)
        b) Import into studio for uploading to server
    3. Assign to sites

    a) Feeds will be processed by DW jobs that are executed in context of a site
        (1) pipelines , for jobs, are looked up in cartridges at the BM site (ie Sites-site)
        (2) Templates (used for emails) looked up in the cartridge of the respective SF sites
    b) BM > Admin > Sites > Manage Sites > Sites - site AND each SF site
        (1) under cartridges add int_simplefeeds

4. Import Custom Object type definitions
    a) Generic implementation uses a custom object - *FeedJobConfiguration* - stores configuration data
        (1) Exists in context of a site
    b) Custom Object definition is located in metadata/FeedJobConfiruationCO.xml .
        (1) Must be loaded on all instances that use the feed integration cartridge
        (2) Can store, share, distribute the definition using **Site Import/Export** -> upload the file and import
    c) NOTE: BM users with permission to modify custom objects can view and modify simple feed integration configuration data, which includes endpoints, login/pass, public key for encryption of CC data in order export feeds.
        (1) Make sure access privileges in BM are set so only authorized users can access respective modules
5. Create Job Schedules
    a) Arbitrary number of schedules and feed configurations can be set up on a DW instance, but can't tell a schedule what configuration to use
        (1) Only way to tie a schedule to a configuration is to use a unique pipeline start node
        (2) Default pipeline - *Costum_FeedJobs*
            Default start node - *StartDefault*
            (a) To set up additional configurations, need to add additional parameters and reference those when setting up schedules
    b) Most cases, jobs are triggered by predefined schedules (hourly, daily, etc)
        (1) Can also keep schedule disabled and only trigger them manually in BM
        (2) Can provide code that triggers Job with **RunJobNow** pipelet
    c) To create job schedule: BM > Admin > Operations > Job Schedules
        (1) Create a new job, name it
        (2) Set execution scope ot sites
        (3) Pipeline - Custom_FeedJobs, Start node - StartDefault
    d) **Sites** tab, select SF sites for which to execute
    e) job
        (1) NOTE: can provide site specific Feed job configurations
    f) **Resources** tab, specify all object types that job will modify

            (1) Job framework attempts to acquire locks for resources before job starts
                (a) Avoids parallel execution with other import process or data replication
    g) **Notifications** tab email to receive job execution status
        (1) Can send granular success/error emails by feed task
        (2) Can report temporary errors like communication errors with external systems as job failures to the job framework (on-temporary-error:FAIL)
            (a) Here define how to handle these errors:
                (i) Continue as Schedule
                (ii) Retry
                (iii) Stop on Error
    h) **Parameters** tab define a parameter for any resource you want to import into a site

6. Create feed job configuration
    a) Feed job configuration is stored in a site specific CO and may contain multiple tasks
    b) BM > CO > CO editor
    c) As Id provide Id used when creating new instance of the *FeedJobConfiguration* CO
        (1) The identifier used in pipeline *Custom_FeedJobs-StartDefault*, check properties of the script node
    d) The file: documentation/tasksXMLCatalogOnly.xml provides an example for TasksXML
        (1) Lists all supported tasks and contains inline documentation
        (2) Derive project specific configurations from that file by removing unneeded tasks and updating the information
        (3) May make sense to have the same task multiple times in a feed configuration, IF feeds from different sources with different contexts are processed
7. Testing
    a) Structure of XML for CO *FeedJobConfiguration* can be found in the sample file TasksXML.xml in the documentation folder
    b) sample imports in sample data folder
8. NOTE: Not advisable to expose the **RunJobNow** pipelet in a public pipeline as this could be exploited to flood your job queue. **If you must use, implement custom security mechanism so only authorized request can execute the pipelet**