ApacheCon NA 2015

How to Avoid Common Mistakes in OFBiz Development

Presented by Adrian Crum

1Tech, Ltd.
29 Harley Street, London, W1G 9QR, UK
www.1tech.eu

Overview

- Common Getting Started Problems
- Common Design Problems
- Common Customization Problems

Common Getting Started Problems

- Use The Correct Java SDK Version
- How To Set Up Your IDE
- Learn The Architecture

Use The Correct Java SDK Version

- OFBiz Release 12 and later – use Java 1.7
- OFBiz Release 11 and earlier – use Java 1.6

If you have more than one Java SDK on your computer, then modify the ant script to use the correct one. You may also need to modify your IDE settings to use the correct Java SDK.

You can examine the macros.xml file to find out which Java SDK version OFBiz requires.

How To Set Up Your IDE

You can use any Integrated Development Environment (IDE) you prefer, but to get started as quickly as possible, it is best to use Eclipse. OFBiz includes the necessary settings for Eclipse.

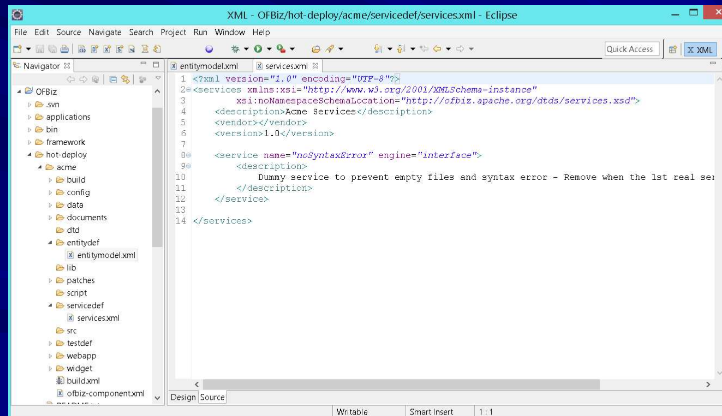
Some IDEs support importing Eclipse projects.

https://cwiki.apache.org/confluence/display/OFBIZ/Eclipse+Tips (note that some links are broken)

https://cwiki.apache.org/confluence/display/OFBIZ/Running+and+Debugging+OFBiz+in+Eclipse

How To Set Up Your IDE

If you are using Eclipse, then try using the XML perspective



Eclipse will default to the Java perspective, but most OFBiz developers don’t find that useful.

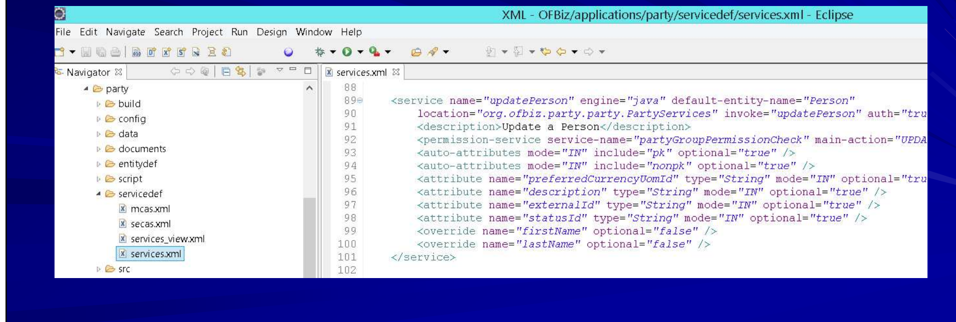
Learn The Architecture

- OFBiz uses a Service Oriented Architecture
- Start at the service level, then work your way down
- OFBiz has a persistence (data model) layer – the Entity Engine
- OFBiz has a presentation layer – the Rendering Engine (screen widgets)
- OFBiz uses a custom J2EE servlet implementation (Control Servlet)

Many new developers make the mistake of starting off by analyzing OFBiz Java code. That is a bad idea - because application development should start at a higher level on the technology stack.

Learn The Architecture

Services implement business logic and they provide an abstraction that hides implementation details from your application



The good news: service definitions can be created automatically based on an entity definition. This example is a service based on the Person entity.

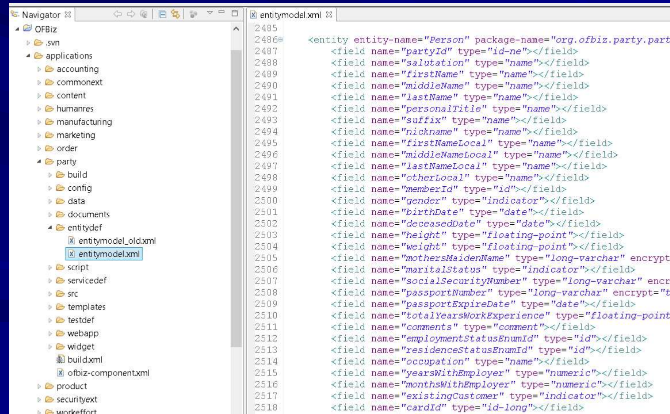
The bad news: service definitions can be created automatically based on an entity definition. If the entity model changes, then the service definition changes too. This could be good or bad. If your service is only invoked internally, then this behavior will be fine. If external systems are invoking your API, then automatic redefinitions can cause problems in those systems.

So, think carefully about how you define your services. There is no “right” or “wrong” – it’s more about what works best for your use case.

OFBiz will read these service definition files during startup, and build internal Java data structures based on them. Those Java data structures (models) are reused in other parts of the framework – like in the rendering engine. The model of this service definition can be used to automatically generate the data input fields in an HTML <form> element.

Learn The Architecture

The Entity Engine is based on data models that are defined in XML files

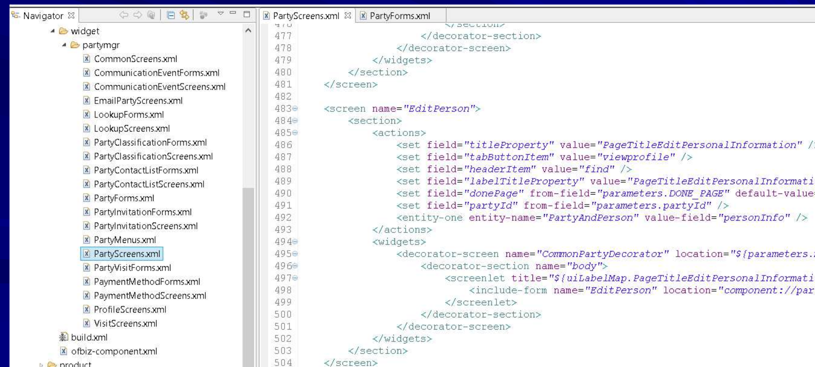


In this example, we have the entity definition for the Person entity that was referenced in the previous service definition.

OFBiz will read these entity definition files during startup, and build internal Java data structures based on them. Those Java data structures (models) are reused in other parts of the framework – like in the service definition. The model of this entity definition is used to automatically generate the IN and OUT parameters of the createPerson service.

Learn The Architecture

The Rendering Engine is based on generalized layout instructions that are defined in XML files



OFBiz will read these screen definition files when requested, and build internal Java data structures based on them. Those Java data structures (models) are passed to output-format-specific renderers that generate the desired output (HTML, PDF, CSV, etc).

A common mistake new developers make is assuming these XML screen definitions will only output HTML – so they embed markup and JavaScript in the XML. That should never be done! These XML files describe a generic layout only.

Learn The Architecture

The Rendering Engine is aware of entity definition models and service definition models – so screen and form creation are a snap

Navigator | PartyScreens.xml | PartyForms.xml

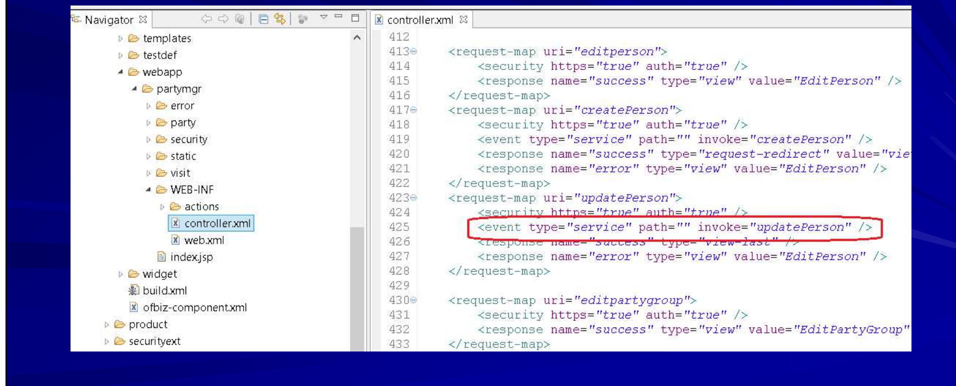
- widget
  - partymgr
    - CommonScreens.xml
    - CommunicationEventForms.xml
    - CommunicationEventScreens.xml
    - EmailPartyScreens.xml
    - LookupForms.xml
    - LookupScreens.xml
    - PartyClassificationForms.xml
    - PartyClassificationScreens.xml
    - PartyContactListForms.xml
    - PartyContactListScreens.xml
    - PartyForms.xml
    - PartyInvitationForms.xml
    - PartyInvitationScreens.xml
    - PartyMenus.xml
    - PartyScreens.xml
    - PartyVisitForms.xml
    - PaymentMethodForms.xml
    - PaymentMethodScreens.xml
    - ProfileScreens.xml
    - VisitScreens.xml

```
22
23    <form name="EditPerson" type="single" target="updatePerson" de
24        <alt-target use-when="personInfo==null" target="createPers
25        <auto-fields-service service-name="updatePerson" />
26        <field use-when="personInfo!=null" name="partyId" title="$
27            <display />
28        </field>
29        <field use-when="personInfo==null&amp;&amp;partyId==null"
30            <text />
31        </field>
32        <field use-when="personInfo==null&amp;&amp;partyId!=null"
33            <display also-hidden="false" />
34        </field>
35        <field name="firstName" title="${uiLabelMap.PartyFirstName
36            <text size="40" maxlength="60" />
37        </field>
38        <field name="lastName" title="${uiLabelMap.PartyLastName}"
39            <text size="40" maxlength="60" />
40        </field>
41        <field name="gender">
42            <drop-down allow-empty="true">
43                <option key="M" description="${uiLabelMap.CommonMa
44                <option key="F" description="${uiLabelMap.CommonFe
45            </drop-down>
46        </field>
```

In this example, the <auto-fields-service> element automatically generates an input form based on the updatePerson service definition.

Learn The Architecture

The Control Servlet provides a convenient way to map requests to services and screen definitions



The Control Servlet is aware of service definition models. In this example, the updatePerson URL is mapped to the updatePerson service. The Control Servlet will use the service definition model to map URL parameters to service IN parameters.

Common Design Problems

- Follow Best Practices
- Follow Common Design Patterns

Follow Best Practices

OFBiz Best Practices can be found on the OFBiz Wiki site

(be aware that some information might be outdated)

OFBiz Project Administration Workspace
Pages
Space tools
Best Practices Guide
HTML and CSS Best Practices
Managing Your Source Differen…
Methodology Recommendations
User Interface Layout Best Prac…

Pages / Home / Apache OFBiz Contribution and Development
Best Practices Guide
Added by Jacopo Cappellato, last edited by Jacopo Cappellato on Dec 09, 2008

Best Practices Guide
Written By: David E. Jones, jonesde@apache.org

Table of Contents
- Introduction
- General Concepts
- Data Layer
- Logic Layer
- Presentation Layer

OFBiz Project Administration Workspace
Pages
Space tools
User Interface Layout Best Practi…

Pages /… / Best Practices Guide
User Interface Layout Best Practices
Added by adrianc@hlmksw.com, last edited by Jacques Le Roux on Mar 13

- User Interface Layout Best Practices
  - Navigation
    - Main Navigation
    - Secondary Navigation
    - Application Navigation
    - Application Area Tabs
  - Guidelines for Main Screen Areas
    - Main Screen Area
    - Create or Add Screens
    - Find Screens
  - Terminology
    - New, Add

User Interface Layout Best Practices
In order to present the user with a consistent user interface (U
(manager) applications. Layout for public-facing screens (such

https://cwiki.apache.org/confluence/display/OFBADMIN/Best+Practices+Guide

https://cwiki.apache.org/confluence/display/OFBADMIN/User+Interface+Layou
t+Best+Practices

Follow Best Practices

Sometimes best practices are not followed in the project. Be careful when copying artifacts or following tutorials…

The Best Practice to follow is:

“When a screen is split up into multiple templates or screens the data preparation action should be associated only with the individual small screen that it prepares data for. This makes it easier to move templates and content pieces around and reuse them in many places. “

In other words, the data preparation logic should be contained within the screen widget that renders it – including forms, menus, and trees.

But the tutorial found on the Wiki:

https://cwiki.apache.org/confluence/display/OFBIZ/OFBiz+Tutorial+-+A+Beginners+Development+Guide

does not follow this best practice – the data preparation is done in the screen and not in the form that displays the data. Consequently, that form is not reusable – it will be empty (not contain any data) if it is used in any other screen.

Follow Common Design Patterns

- Use SOA! Implement your business logic as services – so they can be reused and exported
- Understand and emulate the data modeling patterns in The Data Model Resource book
- Understand and emulate the design patterns in the Design Patterns: Elements of Reusable Object-Oriented Software book

Even though OFBiz uses a Service Oriented Architecture (SOA), there are many examples in the project where that design pattern is not followed – resulting in duplicate code and business logic that is scattered everywhere.

One really bad example of this is the OFBiz shopping cart – which is implemented as a bunch of Java classes that are referenced by a shopping cart object stored in the HTTP session. A better approach would be to implement the shopping cart as a set of services – so they can be reused in other places.

The Data Model Resource Book does more than describe a reusable data model, it introduces modeling patterns that can be reused. Patterns like abstract types and subtypes, date ranges, etc.

Both books are essential for anyone wanting to do development work in OFBiz. Not only will they help you with your design work, they will also help you understand how/why things are done in the project itself.

Common Customization Problems

- Create A Hot-Deploy Component
- REUSE REUSE REUSE
- Extending Artifacts
- Overriding Artifacts
- Do This, Don't Do That

Create A Hot-Deploy Component

- Use the ant `create-component` target to create a new component, and then put all of your development work in there

```
C:\Develop\ofbiz>ant create-component
Buildfile: C:\Develop\ofbiz\build.xml

create-component:
    [input] Component name: (e.g. mycomponent) [Mandatory]
acme
    [input] Component resource name: (e.g. MyComponent) [Mandatory]
Acme
    [input] Webapp name: (e.g. mycomponent) [Mandatory]
acme
    [input] Base permission: (e.g. MYCOMPONENT) [Mandatory]
ACME
     [echo] The following hot-deploy component will be created:
     [echo]               Name: acme
     [echo]               Resource Name: Acme
     [echo]               Webapp Name: acme
     [echo]               Base permission: ACME
     [echo]               Folder: C:\Develop\ofbiz/hot-deploy/acme
     [echo]
    [input] Confirm:  (Y, [N], y, n)
```

Avoid putting your custom development work in existing OFBiz folders – doing so makes upgrading OFBiz difficult. By keeping all of your development work in a custom component, upgrading OFBiz is easy.

An OFBiz component can support multiple web applications, so there is no need to create a separate component for each web application.

REUSE REUSE REUSE

- 800+ entities, 2000+ services – most likely what you need is already there
- Copy UI artifacts to your custom component
- Extend or Override everything else

The biggest mistake new OFBiz developers make is reinventing the wheel. Most projects based on OFBiz should be little more than customizing the UI and making minor changes to workflows.

Your custom component can reference (point to) existing UI artifacts, or you can copy existing UI artifacts to your custom component. I recommend copying UI artifacts, because the UI is a very subjective thing that sees a lot of debate within the community – consequently its appearance changes regularly.

Conversely, business processes are fairly standard things that don't see much debate and remain mostly constant – so those processes can be reused safely as-is. A particular project might need to change those processes slightly, so OFBiz provides two mechanisms for that: extension and override.

Extending Artifacts

You can extend entities to add fields, relations, and indexes

```
<extend-entity entity-name="OrderItem">
    <field name="forOrderId" type="id" />
    <field name="forOrderItemSeqId" type="id" />
    <relation type="one-nofk" rel-entity-name="OrderItem">
        <key-map field-name="forOrderId" rel-field-name="orderId" />
        <key-map field-name="forOrderItemSeqId" rel-field-name="orderItemSeqId" />
    </relation>
</extend-entity>
```

In this example, we are adding two fields and a relationship to the OrderItem entity. These can be found in The Data Model Resource Book, but they are missing from the OFBiz data model.

The OrderItem entity is defined in the order component, and we could just add the fields and relation there – but that would make OFBiz updates/upgrades difficult. Instead, we extend the entity in our custom component and leave the original OFBiz code untouched.

View entities can be extended in the same way.

Overriding Artifacts

You can redefine entities to change fields, relations, and indexes

```
<!-- Redefine the OFBiz FacilityParty entity to change the PartyRole fk.
     This will allow us to assign parties to Facilities in various roles
     without requiring a matching PartyRole value. -->
<entity entity-name="FacilityParty" redefinition="true"
        package-name="eu.1tech.core"
        title="Facility Role Entity">
    <field name="facilityId" type="id-ne"></field>
    <field name="partyId" type="id-ne"></field>
    <field name="roleTypeId" type="id-ne"></field>
    <field name="fromDate" type="date-time"></field>
    <field name="thruDate" type="date-time"></field>
    <prim-key field="facilityId" />
    <prim-key field="partyId" />
    <prim-key field="roleTypeId" />
    <prim-key field="fromDate" />
    <relation type="one" fk-name="FACILITY_RLE_FACI" rel-entity-name="Facility">
        <key-map field-name="facilityId" />
    </relation>
    <relation type="one" fk-name="FACILITY_RLE_PRT" rel-entity-name="Party">
        <key-map field-name="partyId" />
    </relation>
    <relation type="one" fk-name="FACILITY_RLE_ROL" rel-entity-name="RoleType">
        <key-map field-name="roleTypeId" />
    </relation>
    <relation type="one-nofk" rel-entity-name="PartyRole">
        <key-map field-name="partyId"/>
        <key-map field-name="roleTypeId"/>
    </relation>
</entity>
```

In this example, we are redefining the FacilityParty entity so we can change a relationship.

One annoying thing about the OFBiz data model is the repetitious relations to PartyRole – which forces you to create a PartyRole entity value every time you want to associate a party with something. Overrides/redefinitions like this one can fix that and make the data model a lot more flexible and easy to use.

To redefine an entity, just copy-and-paste the existing entity definition to your custom component, and then make your changes. OFBiz will replace the existing entity definition with your custom one.

Be sure to set the redefinition attribute to “true” - so OFBiz does not log warnings about duplicate definitions. If you do not set the attribute to “true” your redefinition will still work.

I like to change the entity package name to make it obvious to others that I have redefined an existing OFBiz entity, but that change is not required to make the redefinition work.

Overriding Artifacts

You can override a web application

```
hot-deploy
  acme
    build
    config
    data
    documents
    dtd
    entitydef
    lib
    patches
    script
    servicedef
    src
    testdef
    webapp
    widget
    build.xml
    ofbiz-component.xml
  README.txt
runtime
```

```
8      <!-- place the config directory on the classpath to ac
9      <classpath type="dir" location="config"/>
10     <classpath type="dir" location="dtd"/>
11
12     <!-- load single or multiple external libraries -->
13     <classpath type="jar" location="build/lib/*"/>
14     <classpath type="jar" location="lib/*"/>
15
16     <!-- entity resources: model(s), eca(s), group, and da
17     <entity-resource type="model" reader-name="main" loade
30
31     <test-suite loader="main" location="testdef/AcmeTests.
32
33     <!-- web applications; will be mounted when using the
34     <webapp name="order"
35         title="Order"
36         server="default-server"
37         location="webapp/acme"
38         base-permission="OFBTOOLS,ORDERMGR"
39         mount-point="/ordermgr"/>
40 </ofbiz-component>
41
```

In this example, we will use our acme custom component to override the OFBiz Order Manager web application.

Modify the <webapp> element in ofbiz-component.xml so that it duplicates the existing Order Manager web application element, but leave the location attribute as-is.

Overriding Artifacts

You can override a web application

```
hot-deploy
  acme
    build
    config
    data
    documents
    dtd
    entitydef
    lib
    patches
    script
    servicedef
    src
    testdef
    webapp
      acme
        error
        WEB-INF
          actions
          controller.xml
          web.xml
        index.jsp
    widget
    build.xml
```

```
2  <site-conf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3          xsi:noNamespaceSchemaLocation="http://ofbiz.apache.org/dtds/site-conf.xsd">
4      <!-- The controller elements that are common to all OFBiz components
5           can be found in the following xml file. A component can override the
6           elements found in the common-controller.xml file. -->
7      <include location="component://common/webcommon/WEB-INF/common-controller.xml"/>
8      <include location="component://order/webapp/ordermgr/WEB-INF/controller.xml"/>
9
10     <description>Acme Component Site Configuration File</description>
11
12     <!-- Events to run on every request before security (chains exempt) -->
13     <!--
14     <preprocessor>
15     </preprocessor>
16     -->
17     <!-- Events to run on every request after all other processing (chains exempt) --
18     <!--
19     <postprocessor>
20         <event name="test" type="java" path="org.ofbiz.webapp.event.TestEvent" invoke
21     </postprocessor>
22     -->
23
24     <!-- Request Mappings -->
25
26     <!-- View Mappings -->
27
28 </site-conf>
```

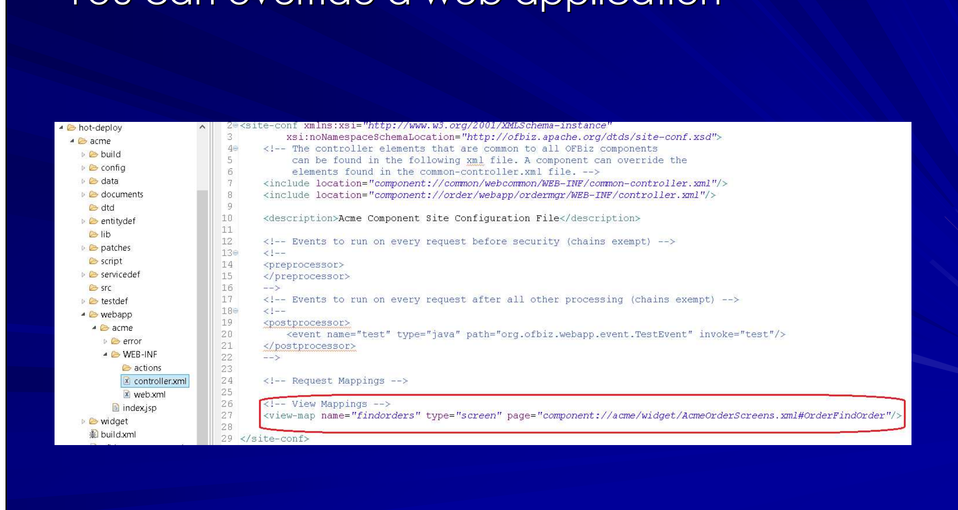
Next, we import the Order Manager request maps and view maps into our acme custom component. Simply add an <include> element in the acme controller.xml file, and then remove all existing <request-map> elements and <view-map> elements.

Overriding Artifacts

You can override a web application

One more change and our web application override is done. All of the Order manager screens reference its screen decorator, and the location of that decorator is contained in a web application context parameter. So, we need to update that parameter in our acme web application so it will use the Order Manager screen decorator. Simply change the value of the mainDecoratorLocation context parameter in the acme web.xml file.

At this point, we have completely overridden the Order Manager web application and put it under the control of our custom acme application.



Now we can replace screens in Order Manager with our own custom versions. In this example, we replace the Find Orders screen with our own version.

Extending Artifacts

You can extend form widgets and menu widgets

XML - OFBiz/applications/product/widget/catalog/ProductForms.xml - Eclipse

```
2120        <field name="contactMechIdTo" title="To Email Address">
2121            <display-entity  entity-name="ContactMech" key-field-name="contactMechId" desc
2122        </field>
2123        <field name="content"><display/></field>
2124        <field map-name="subjectMap" name="subject"><display/></field>
2125    </form>
2126
2127    <form name="EditCommEvent" extends="EditCommEvent"
2128        extends-resource="component://party/widget/partymgr/CommunicationEventForms.xml">
2129        <field name="productId" map-name="parameters"><hidden/></field>
2130    </form>
2131
2132    <form name="UpdateProductRole" type="list" target="updatePartyToProduct" title="" list
2133        odd-row-style="alternate-row" default-table-style="basic-table">
2134        <auto-fields-service service-name="updatePartyToProduct"/>
2135        <field name="productId"><hidden/></field>
2136        <field name="sequenceNum"><text size="5"/></field>
```

In this example, a form from the party component is being extended to add a product ID field. This will enable the product component to associate a communication event to a product.

Do This, Don't Do That

The following slides are some common mistakes I have encountered while working on OFBiz projects

Do This, Don't Do That

Make screen widgets reusable

```
<screen name="EditPerson">
    <section>
        <actions>
            <set field="titleProperty" value="PageTitleEditPersonalInformation" />
            <set field="tabButtonItem" value="viewprofile" />
            <set field="headerItem" value="find" />
            <set field="labelTitleProperty" value="PageTitleEditPersonalInformation" />
            <set field="donePage" from-field="parameters.DONE_PAGE" default-value="viewprofile" />
            <set field="partyId" from-field="parameters.partyId" />
         1. <entity-one entity-name="PartyAndPerson" value-field="personInfo" />
        </actions>
        <widgets>
         2. <decorator-screen name="CommonPartyDecorator" location="${parameters.mainDecoratorLocation}">
                <decorator-section name="body">
                    <screenlet title="${uiLabelMap.PageTitleEditPersonalInformation}">
                        <include-form name="EditPerson" location="component://party/widget/partymgr/PartyForms.xml" />
                    </screenlet>
                </decorator-section>
            </decorator-screen>
        </widgets>
    </section>
</screen>
```

Don't Do This:

1. Data used in the included form is gathered in the screen widget instead of in the EditPerson form widget. Therefore, the EditPerson form widget can not be reused.

2. The CommonPartyDecorator "sub-decorator" is located in a different file and its location is specified using the mainDecoratorLocation context variable. Therefore, the EditPerson screen widget can not be reused.

Do This, Don't Do That

Make screen widgets reusable

```
<form name="EditPerson" type="single" target="updatePerson" default-map-name="personInfo"
    <actions>
1.          <entity-one entity-name="PartyAndPerson" value-field="personInfo" />
    </actions>
    <alt-target use-when="personInfo==null" target="createPerson" />
    <auto-fields-service service-name="updatePerson" />
    <field use-when="personInfo!=null" name="partyId" title="${uiLabelMap.PartyPartyId}" t
        <display />
    </field>
    <field use-when="personInfo==null&amp;&amp;partyId==null" name="partyId" title="${uiLa
        <text />
    </field>
    <field use-when="personInfo==null&amp;&amp;partyId!=null" name="partyId" title="${uiLa
        <display also-hidden="false" />
    </field>
```

Do This:

1. Data used in the included form is gathered in the form widget. Now this form can be included in other screens.

Do This, Don’t Do That

Make screen widgets reusable

```
<screen name="CommonPartyDecorator">
    <section>
        <!-- Decorator code removed for conciseness -->
    </section>
</screen>

<screen name="EditPerson">
    <section>
        <actions>
            <set field="titleProperty" value="PageTitleEditPersonalInformation" />
            <set field="tabButtonItem" value="viewprofile" />
            <set field="headerItem" value="find" />
            <set field="labelTitleProperty" value="PageTitleEditPersonalInformation" />
            <set field="donePage" from-field="parameters.DONE_PAGE" default-value="viewprofile" />
            <set field="partyId" from-field="parameters.partyId" />
        </actions>
        <widgets>
        2.  <decorator-screen name="CommonPartyDecorator" location="${parameters.partyDecoratorLocation}">
                <decorator-section name="body">
                    <screenlet title="${uiLabelMap.PageTitleEditPersonalInformation}">
                        <include-form name="EditPerson" location="component://party/widget/partymgr/PartyFo
                    </screenlet>
                </decorator-section>
            </decorator-screen>
        </widgets>
    </section>
</screen>
```

Do This:

2. The CommonPartyDecorator “sub-decorator” is located in the same file and its location is specified using the partyDecoratorLocation context variable. Now this screen widget can be reused.

Do This, Don't Do That

Storing configuration settings in static final variables

UtilFormatOut.java

```
43
44     // ------------------- price format handlers -------------------
45
46     // FIXME: This is not thread-safe! DecimalFormat is not synchronized.
47     static DecimalFormat priceDecimalFormat = new DecimalFormat(
48             UtilProperties.getPropertyValue("general.properties",
49             "currency.decimal.format", "#,##0.00"));
50
51     /** Formats a Double representing a price into a string
55     public static String formatPrice(Double price) {
56         if (price == null)
57             return "";
```

Don't Do This!

Do This, Don't Do That

Storing configuration settings in static final variables

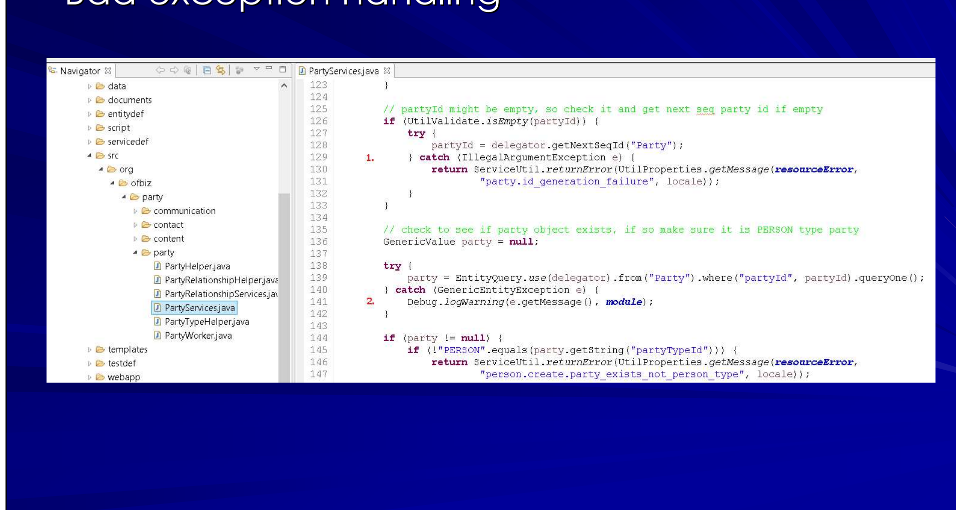
UtilFormatOut.java

```
43
44    // ------------------- price format handlers -------------------
45
46    private static DecimalFormat getDecimalFormat() {
47        return new DecimalFormat(
48            UtilProperties.getPropertyValue("general.properties",
49            "currency.decimal.format", "#,##0.00"));
50    }
51
52    /** Formats a Double representing a price into a string
56    public static String formatPrice(Double price) {
57        if (price == null)
58            return "";
```

Do This!

Do This, Don't Do That

Bad exception handling



Don't Do This:

1. Methods that throw undeclared exceptions. In this example, the service is doing the right thing – catching a thrown exception and returning an error. But the Delegator is doing the WRONG thing – it is throwing an undeclared (and unchecked) exception. There is no way a developer can know that the getNextSeqId method throws IllegalArgumentException unless you look through the Delegator implementation code.

2. Ignore thrown exceptions. The EntityQuery.use method will throw an exception if something goes wrong during the method call, but that exception is ignored in this service.

3. Multiple try-catch blocks that don't add or do anything meaningful. In this service there are other try-catch blocks that return an error, and the only difference is the error message being returned. At first glance, this appears to be meaningful, but the caught exceptions have one thing in common – they indicate something serious went wrong (like a dropped database connection). So, the returned error messages actually HIDE the real problem.

4. Only declared exceptions are caught. Every line in this service has the potential to throw an exception – OutOfMemoryException, NullPointerException, etc - yet those are not caught. The service engine

Do This, Don't Do That

Bad exception handling

```
101     * Creates a Person.
107    public static Map<String, Object> createPerson(DispatchContext ctx, Map<String, ? extends Object> context) {
108        Locale locale = (Locale) context.get("locale");
109        try {
110            Delegator delegator = ctx.getDelegator();
111
112            // Put your service implementation here
113
114            return ServiceUtil.returnSuccess();
115        } catch (Exception e) {
116            // Message text is something like "An exception was thrown in the createPerson service: "
117            String errorMsg = UtilProperties.getMessage(resource, "CreatePersonExceptionThrown", locale);
118            Debug.logWarning(e, errorMsg, module);
119            return ServiceUtil.returnError(errorMsg + e);
120        }
121    }
```

Do This!

All try-catch blocks have been removed and they are replaced with a single one. Now all exceptions are caught and a meaningful error message is returned.

If your service throws an exception that is recoverable, then add a try-catch block for that exception (inside the main try-catch block) and implement your recovery code in its catch block.

Do This, Don’t Do That

Bad event handlers

```
253    <request-map uri="addItemToShoppingList">
254        <security https="true" auth="true" />
255        <event type="service" path="" invoke="createShoppingListItem" />
256        <response name="success" type="view" value="editShoppingList" />
257        <response name="error" type="view" value="editShoppingList" />
258    </request-map>
259    <request-map uri="addBulkToShoppingList">
260        <security https="true" auth="true" />
261        <event type="java" path="org.ofbiz.order.shoppinglist.ShoppingListEvents" invoke="addBulkFromCart" />
262        <response name="success" type="view" value="editShoppingList" />
263        <response name="error" type="view" value="editShoppingList" />
264    </request-map>
265    <request-map uri="addListToCart">
266        <security https="true" auth="true" />
267        <event type="java" path="org.ofbiz.order.shoppinglist.ShoppingListEvents" invoke="addListToCart" />
268        <response name="success" type="view" value="editShoppingList" />
269        <response name="error" type="view" value="editShoppingList" />
270    </request-map>
```

Don’t Do This!

Business logic is contained in a request event of type “java” – this is bad because the specified method will be invoked WITHOUT a transaction in place. The Delegator implementation will wrap each method call in a transaction, but the event as a whole will not be wrapped in a transaction – opening the door to partial updates and data corruption.

Do This, Don't Do That

Bad event handlers

```
253  <request-map uri="addItemToShoppingList">
254      <security https="true" auth="true" />
255      <event type="service" path="" invoke="createShoppingListItem" />
256      <response name="success" type="view" value="editShoppingList" />
257      <response name="error" type="view" value="editShoppingList" />
258  </request-map>
259  <request-map uri="addBulkToShoppingList">
260      <security https="true" auth="true" />
261      <event type="service" path="" invoke="addBulkFromCart" />
262      <response name="success" type="view" value="editShoppingList" />
263      <response name="error" type="view" value="editShoppingList" />
264  </request-map>
265  <request-map uri="addListToCart">
266      <security https="true" auth="true" />
267      <event type="service" path="" invoke="addListToCart" />
268      <response name="success" type="view" value="editShoppingList" />
269      <response name="error" type="view" value="editShoppingList" />
270  </request-map>
```

Do This!

Implement business logic as a service. The service engine will wrap the service call in a transaction, so the entire process is atomic – either all of it succeeds, or all of it fails.

Use the "java" event type only for parameter validation. In other words, use it to check invariants and return an error message if the user entered invalid data, but NEVER use it to implement business logic that updates data.

How to Avoid Common Mistakes in OFBiz Development

Thank you for participating!