Form Rule and ISF JavaScripts UseCase Analysis

This appendix contains the following topics:

Form Rule and ISF JavaScripts Use Case Analysis

In a complex service catalog you can easily have hundreds of instances where active form rules or ISF functions are used to enhance the service form's usability and interactivity.

1. Perform a use case analysis in order to determine what rules are required and when they will fire.
2. Perform a detailed design analysis to determine what dictionaries and fields are needed to support the specified use cases; how the dictionaries should be combined into forms; and what tools you will need to implement the requirements.
3. Define the dictionaries and forms. Specify the display and access control properties of the forms.
4. Write active form rules and sequence these to meet the detail design criteria. If required, write JavaScript functions and libraries; attach the functions to the appropriate HTML event or events.
5. Define the services which use the reusable form components previously defined.
6. Test and make fixes as required.

This section includes some examples of both rules and ISF code integrated into a service form, and the sorts of implementation decisions made to meet the requirements. These examples illustrate some common design patterns, such as:

- Adjust the appearance or behavior of dictionaries based on a specific task
- Show/hide dictionaries or fields when a service form is loaded or when the user changes the value in another field
- Manager lookup on change for Person fields
- Drill-Downs
- Ensure that Select Lists are populated during the delivery moment
- Hide dictionaries and clear fields if their value is not relevant to the current context

Use Case Analysis

Review user requirements to determine when a service form's default behavior and appearance need to be supplemented by rules or ISF. While you may document these requirements discursively, a better format might be a table which explicitly defines the behavior and its triggering events. An example is shown below.

Table 1: Example

Service Name	Use Case Description	Moment
All	If a person is high profile, display his status as read-only; if not, hide the status	Ordering

Where:

Service Name is either "all" or a list of the services which are affected by the requirement.

Use Case Description specifies, in language accessible to business users, the desired behavior of the service form.

Moment is either "all" or one or more of the moments that occur during the fulfillment of a requisition.

Try to collect all the use cases for the current project, so you can estimate the scope of the work involved.

Detailed Design

In the detailed design phase the programmer must review the use cases previously defined and specify, at a high level, the rules or JavaScript components that need to be written, and the triggering events for these rules/functions. The design incorporates a detailed specification of the forms, dictionaries and fields involved. In particular, the analysis should cover:

- Will you need to use ISF to supplement the form rules? If so, ensure that your development environment is set up to support ISF (JavaScript) development, testing and debugging, and that personnel are available with the requisite skill set.
- Will you need to use data retrieval rules or SQL option lists? If so, ensure that the development environment includes a means to test and debug SQL queries; that datasources are defined to allow you to access the desired data; and that personnel are available with knowledge of the structure of the source data and the requisite skill set.

Scenario #1: Dynamically Adjusting Form Appearance and Behavior

Functional Requirements

In requesting a database to be created, the user must choose whether the database server type is Oracle, SQLServer, or some "Other" database. If some other, nonenterprise standard database is chosen, additional information must be gathered from the user; otherwise, the additional fields are hidden.

Scenario:

Table 2: Dynamically Adjusting Form Appearance and Behavior

Is the current value of the DatabaseType field in the Database dictionary equal to "Other"?	If so, display the Description field in the same dictionary, and require the user to enter additional information.

Dictionary/Form Design

Define a dictionary called, for example, NewDatabase.

- The dictionary includes a field named DatabaseType, rendered as a radio button which allows the user to designate whether the database is Oracle, SQLServer, or Other.
- If the user selects "Other", the type of database must be provided and is mandatory. Otherwise, this field is hidden.

Detailed Rules Design

This use case can be implemented entirely through the use of conditional rules.

The AIT_DATABASE.DatabaseServer field needs rules which fires when the field's value is changed.

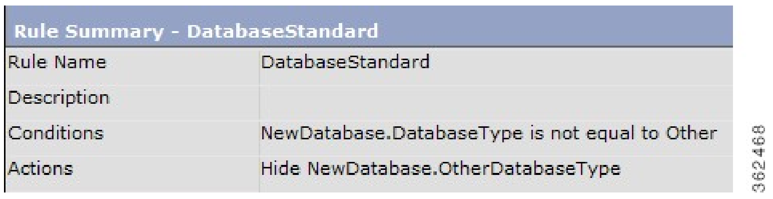
- If the value which the user selects is 'Other', a rule must display the "Other"-dependent dictionary fields and ensure that all such fields are mandatory.
- If the value which the user selects is not 'Other', a rule must ensure that the "Other"-dependent fields are not visible.

Unfortunately, this release of Service Catalog does not include if/then/else logic in the rules, so you will need two rules to implement this design.

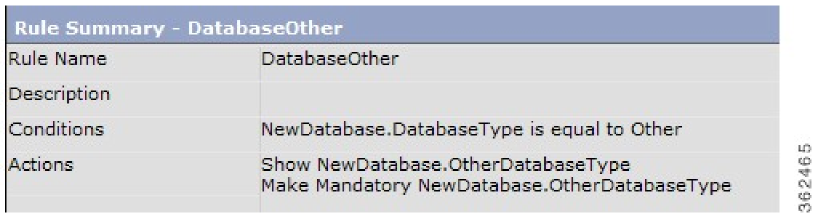
Conditional Rule Implementation

Define two conditional rules as follows:

Figure 1: Conditional Rule Implementation



Rule Summary - DatabaseStandard	
Rule Name	DatabaseStandard
Description	
Conditions	NewDatabase.DatabaseType is not equal to Other
Actions	Hide NewDatabase.OtherDatabaseType



Rule Summary - DatabaseOther	
Rule Name	DatabaseOther
Description	
Conditions	NewDatabase.DatabaseType is equal to Other
Actions	Show NewDatabase.OtherDatabaseType Make Mandatory NewDatabase.OtherDatabaseType

The Triggering Events

When should this rule be applied? In more technical terms, what is the “triggering event”, during the course of a user entering data in the service form, when this rule should “fire”?

It seems obvious that the rule needs to be executed when the user selects a value from the DatabaseServer radio button. So, use the Active Form Behavior tab for this form to associate these two rules with the “When the field changes” event. In this case, it really doesn't matter what order the rules are applied in—either one or the other will fire, but not both.

Build the Service Definition

To test this scenario, you need to complete the definition of the service which contains the active form you have just defined. You could initially create a service containing just that form, for a quick-and-dirty test, but this is clearly just a first step. Forms and their rules can interactive with other forms and their rules, so the best test is the most realistic.

Test

It's easiest to test this scenario by using several browser windows. Keep Service Designer open in one, with the Active Form Components option displayed. Then, start a new session, log in as a My Services user who has permission to order the service. See what happens.

Testing Follow up and Results

The rule as previously defined should work correctly. However, the implementation is incomplete, as the behavior is only triggered when the customer changes (or initially selects) the operating system. If the form is saved and reviewed, or submitted and displayed in a subsequent system moment when the dictionary is editable, the saved value of the DatabaseServer field must be used to adjust the appearance of the form. Therefore, an additional triggering event is required, to fire when the form is loaded.

Scenario #2: Manipulating Customer and Initiator Information

Functional Requirements

Usage needs to take into account two different scenarios that affect the one Customer dictionary and the one Customer-Initiator Form:

Scenario:

Table 3: Manipulating Customer and Initiator Information

If the service is delivered electronically ...	Display the standard set of fields required for collecting Customer information.
If the service needs to be delivered to a specific physical location ...	In addition to the standard fields, display fields pertaining to the Customer's location, as stored in the Customer profile, but allow him/her to specify an alternate service location, for example, if the location on file is out of date or the customer is temporarily at a different location.

Dictionary Design

The design can be implemented using three dictionaries:

- Design the Customer dictionary so that it includes all of the fields that must be available in both types of services—those fields that are always required and those that are required only when the requestor must specify or confirm the location where the service must be delivered.
- Design a Perform Work dictionary that will only be displayed for services to be delivered in person. The form has one field, which asks if the work is to be performed at the customer's location or a different (service) location.
- Design a Service Location dictionary, to be filled in, by default, with the location information from the Customer's profile, but which can be overridden if the requestor indicates that this is different than the location in the profile.

Form Design

The Customer and Initiator dictionaries are typically read-only in all moments. That is, data is displayed from the person's profile and cannot be changed by either the customer or any task performer.

There are at least three ways to make all fields in a dictionary read-only:

- Use the Access Control tab for the form to specify that the dictionary is viewable (not editable) for appropriate moments and participants. This control cannot be overridden by either rules or ISF—a viewable dictionary can be hidden, by it can't be made editable. And field values display as boilerplate, rather than enclosed in input fields. (Q: Would lightweight namespaces work?)
- Make the dictionary editable in the Access Control tab, but define the default fields as having an HTML input type of read-only and the location-related fields as having an input type of hidden. Both hidden and read-only fields are supplied values from an associated lightweight namespace.
- Make the dictionary editable in the Access Control tab, assign appropriate input types to the standard fields, but create a rule, applied when the form is loaded, to make All Fields in the dictionary read-only. This rule would not affect any hidden fields.

Option #2 makes the most sense in this case. Fields could potentially be made writeable, for example, if users are allowed to update out-of-date data from their person profile. And there is no extra rule to keep track off. That takes care of the Customer-Initiator form.

You also need a form in which the PerformWork and ServiceLocation dictionaries are used. Call this form ServiceLocation. (Q: Naming conventions?) The field in the PerformWork dictionary can be implemented as a check box? (Is work performed at the Customer Site?) The ServiceLocation dictionary would have fields corresponding to the person fields included in the Customer dictionary.

Detailed Rules Design

Now you need a rule for the services where a service location is required. The rule is needed to:

- Copy the default location values from the Customer dictionary to the ServiceLocation dictionary. This can be done in an onLoad event.
- If the user says that a different service location is needed, make the ServiceLocation fields writeable. This needs to be done in an onChange event for the PerformWork field. (Q: Nomenclature, again).

The rules are included in the ServiceLocation form.

Conditional Rule Implementation

Does anybody else out there remember COBOL? Most of the time, coding in COBOL was painfully verbose, but it had one really neat command: COPY CORR(esponding). The COPY CORR command copied all values, by name, from one structure to values with corresponding names in a second structure. It would be great to COPY CORR Dictionary1 TO Dictionary2, but that is not possible. So, the first rule (ServiceLocation_onLoad) should be applied in the ordering moment, and Copy Value for all location fields.

The second rule (PerformWork_onChange) is straightforward, reminiscent of the rule written in the initial scenario.

Build the Service Definitions and Test

You'll need two services to test this scenario—one that doesn't include the ServiceLocation form, and one that does.

Scenario #3: Securing Sensitive Data

Functional Requirements

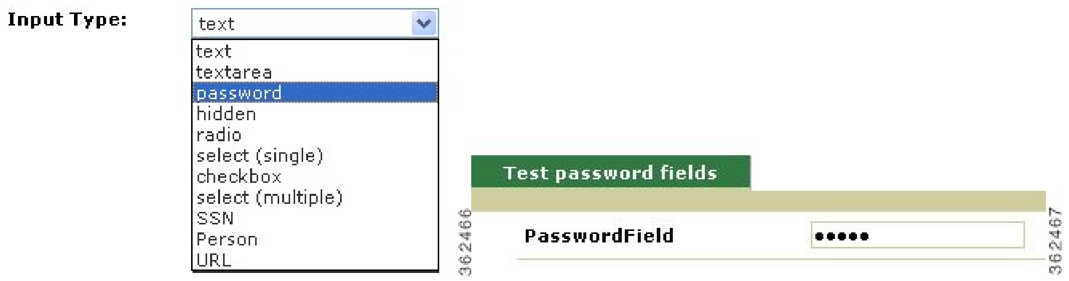
The service requires the user to specify a Social Security Number, credit card information, or other sensitive data that should be available only to people who “need to know”, not to every authorizer or task performer involved in fulfilling the service. The data needs to be protected from attempted hacks as well.

Approach 1 – Hide the Dictionary and Fields

By default, when a dictionary's display property is set to “none” in Service Designer, the dictionary and the values for any fields previously entered are not part of the generated service form seen by users. (This setting may be overridden for backward compatibility with behavior of Service Catalog versions prior to 2007. Be sure to check with your Service Catalog Administrator to verify the setting for dictionary.permission.none.show newScale property.) Therefore, the data in the field would not be visible, even to users savvy enough to view the page source from the browser.

To allow the field value to be initially provided by the user, the field must (obviously) be visible on the form, and the dictionary display setting set to Read/Write. The field's HTML representation can be set to “password”. The field value will then be displayed as a series of asterisks.

Figure 2: Hide the Dictionary and Fields



So, the value is protected from people doing shoulder-surfing. If you view the source of the page, however, the value of the field is visible. The value of the field is not accessible to ISF’s getValue() function—“undefined” is returned. The value of the field is available to the DOM, that is, via JavaScript methods such as:

```
document.getElementById('Dictionary.PasswordField').value.
```

The value of the field is also accessible to Service Link.

Password field values are hidden from user when sent back to the browser, as in the case of reviewing an existing request or performing a task with a password field value in the service form.

Approach 2 – Use Encryption

Encryption can be used in conjunction with some of the practices above to better secure sensitive data. Instead of (or in addition to) using a password field, use JavaScript functions to encrypt the sensitive data before you save it and decrypt it before displaying it on the form. An open-source algorithm on the web called “tiny encryption algorithm” could be used.

```
// Algorithm: David Wheeler & Roger Needham, Cambridge University Computer Lab
//    http://www.cl.cam.ac.uk/ftp/papers/djw-rmn/djw-rmn-tea.html (1994)
//    http://www.cl.cam.ac.uk/ftp/users/djw3/xtea.ps (1997)
//
// JavaScript implementation: Chris Veness, Movable Type Ltd
```

If you define the function to encrypt/decrypt code in a Script, it would be visible in the form if the user did a View Source. However, if you include the function in a library, it would not be visible to users who attempt to view the source, and not subject to reverse engineering.

Approach 3 – Use Secure String

Approach 4– Use Server-side Rules

Data elements that are governed by entitlement or RBAC permissions should have values re-validated on the server-side to prevent the breach of permissions by malicious attempts to manipulate form data before they are sent to the server. Always enable the implicit validation for data retrieval rules unless the rules are executed post-submission. See the Service Form Performance and Security Considerations section for more information.

Scenario #4: Computing a Value in a Form

Functional Requirements

The service requires the user to specify a "Quantity" and "Price" for an item to be ordered. The service should compute the "Extended Price", and show this value on the service form.

Dictionary/Form Design Requirements

A dictionary, call it SVC_PRICE, needs to be created, containing three Number fields. The fields may have decimal precision or not, according to detailed requirements. The dictionary is included in the service form and is writeable for the customer in the ordering moment. All fields are rendered as text fields.

ISF Detailed Design

The Total field needs to be read-only, since users are not allowed to enter a value—it must be computed. The computation needs to take place when the user changes either the Price or the Quantity.

This task requires three custom events:

- The SVC_PRICE_onLoad event sets the ExtendedPrice field to be read-only.
- The SVC_PRICE_Quantity_onChange event computes the ExtendedPrice.
- The SVC_PRICE_Price_onChange event also computes the ExtendedPrice.

JavaScript Code and Events

The first task is better accomplished when the form loads the first time. To do this, create a function in Script Manager called SVC_PRICE_onLoad:

```
SVC_PRICE_onLoad ()
{
  serviceForm.SVC_PRICE.ExtendedPrice.setReadOnly(true);
}
```

This code is associated to the "When the form is loaded (browser-side)" event in the Behavior tab.

The second task is to compute the ExtendedPrice based on the Quantity and Price. Use the "When the item is changed" event for both Quantity and Price. The code needs to verify that both fields have valid values, and then compute the ExtendedPrice. The function is called SVCPRICE_Price_onChange.

```
SVC_PRICE_Price_onChange ()
{
  serviceForm.SVC_PRICE.Total.setReadOnly(true);
  var Price = serviceForm.SVC_PRICE.Price.getValue()[0];
  var Quantity = serviceForm.SVC_PRICE.Quantity.getValue()[0];
/* Blank out current value (if any) of ExtendedPrice */
  serviceForm.SVC_PRICE.Total.setValue(['']);
/* Check is required, since check for Numeric data happens only on Submit. */
  if (isNaN (Price))
  {
    alert ('Price is not a number');
    serviceForm.SVC_PRICE.Price.setFocus(true);
    return;
  }
  if (isNaN (Quantity))
  {
    alert ('Quantity is not a number');
    serviceForm.SVC_PRICE.Quantity.setFocus(true);
    return;
  }
  var Total = Price * Quantity;
  serviceForm.SVC_PRICE.Total.setValue([Total]);
}
```

Refactored JavaScript Code

The code above, defined as a Script in Service Designer, could be attached to the onChange event for two different fields: the Price and the Quantity. In fact, this may be an efficient way to initially test the code. However, this approach, with a function name that doesn't reflect this usage and that doesn't use a library, is harder to maintain in the long run. Therefore, the following refactoring is recommended:

- Edit the function code, renaming the function something generic, like “ComputeExtendedPrice”, and extract the code from Scripts, placing it into a custom library for your application.
- Be sure the custom library is defined in **Scripts > Libraries** and that it is included in the service form when this function is required.
- Create two Scripts, which look like the following:

```
SVC_PRICE_Price_onChange ()
{
  ComputeExtendedPrice();
}
SVC_PRICE_Quantity_onChange ()
{
  ComputeExtendedPrice();
}
```

- Attach these functions to the onChange events of the Price and Quantity fields, respectively.
- Remember to upload the revised library to the application server.

Scenario #5: Formatting Two Fields in a Form

Requirements

Create a JavaScript function to format two fields in a form: A social security number and a phone number. The SSN must verify that there are 9 digits, and is formatted like “999-99-9999” Any formatting done by the user is ignored. The phone number must have 10 digits and it is formatted like (999) 999-9999.

JavaScript

Create two functions, one called **formatSSN** and the other called **formatPhoneNo**. Put both functions in a file called “isfprimerlib.js”. As documented in the ISF Coding and Best Practices, this file may reside on any directory on the application server beneath the RequestCenter.war directory; by convention, ISF libraries are placed on a directory named “isfcode”.

Create a library reference to your JavaScript file in Script Manager. Be sure to specify the library in the Libraries tab for a JavaScript function that are attached to your service.

The resulting code is:

```
function getOnlyDigits (inValue)
{
  var outValue = '';
  var aChar;
  for (i=0; i < inValue.length; i++)
  {
    aChar = inValue.charAt (i);
    if ('0' <= aChar && aChar <= '9')
    {
      outValue = outValue + aChar;
    }
  }
  return outValue;
}
function testValueLength (inValue, inLen, obField, fieldName)
{
  if ((inValue.length > inLen) || (inValue.length < inLen))
  {
    alert (fieldName + ' must have ' + inLen + ' digits and it has ' + inValue.length);
    eval('serviceForm.'+obField).setFocus(true);
    return false;
  }
  return true;
}
function formatSSN (obField)
{
  var SSNString = getOnlyDigits (eval('serviceForm.'+obField).getValue()[0]);
  if (testValueLength (SSNString, 9, obField, 'SSN'))
  {
     eval('serviceForm.'+obField).setValue([SSNString.slice (0,3) +
       '-' + SSNString.slice (3,5) + '-' + SSNString.slice (5)]);
  }
}
function formatPhoneNo (obField)
{
  var phoneString = getOnlyDigits (eval('serviceForm.'+obField).getValue()[0]);
  if (testValueLength (phoneString, 10, obField, 'Phone Number'))
  {
    eval('serviceForm.'+obField).setValue(['(' + phoneString.slice (0,3) + ') '
      + phoneString.slice (3,6) + '-' + phoneString.slice (6)]);
  }
For the field SSN create a function called Customer_SSN_onChange
 that calls formatSSN
:
Customer_SSN_onChange ()
{
  formatSSN('Customer.SSN');
}
```

For the PhoneNo field, create another function called **Customer_PhoneNo_onChange**, that calls formatPhoneNo when the value in the field changes. Just to show an alternative implementation, these functions pass the name of the field and acts “by-reference” rather than “by-value”.

```
Customer_PhoneNo_onChange ()
{
  formatPhoneNo ('Customer.PhoneNo');
}
```

Both functions are called when the value in its respective field changes.

Server-Side Associated Controls

This section explains how to transfer service form data to and from an outside server through an HTTP request. The purpose of this functionality is to allow the Service Designer to use Web widgets that reside on Web servers separate from the application server. This section only deals with the actual transfer and handling of the service form data.

The service form data is sent to the outside Web widget via an HTTP form post. The data is passed in the **wddxdataform** form in the WDDXData variable as a WDDX packet. WDDX is an XML schema for storing data structures in a serialized packet. This allows the data to be passed into or out of Service Catalog independent of the programming language used. You can use an HTTP request to a Service Catalog page to place the resulting data set back into the service form