TrakSYS™ 11 Training Lab Manual Day 4

Revised August 23, 2019

Parsec Automation Corp.
180 North Riverview, No. 300
Anaheim, CA 92808 USA
Phone +1 714 996 5302
Fax +1 714 996 1845
www.parsec-corp.com

Lab 13 | Custom Forms

Overview

While TrakSYS allows you to limit access and rights to configuration entities to specific **Roles**, there are situations where a more custom editing experience is desired.

In this assignment, you will create a special editor form to enable a more limited and controlled experience for modifying **Product** entities. You will use the **Generic Edit Form** Visual Page template as a starting point and add the appropriate API and business rules to expose only the **Name**, **Code** and **Theoretical Rate** fields to be edited.

Unlike the previous page-development labs, which were built from scratch, this lab will be using a **Visual Page Template**. Although many interface needs may be specific to the company, most pages fall into a few general patterns. A template has been created for each of these patterns to provide a "head-start" when developing new content.

The **Generic Form** template, which includes standard formatting patterns, few common parts, and a pre-written script with comments. In the interest of time, this lab will utilize a modified version of this template that has already been pre-configured with necessary changes.

Estimated Time to Complete This Lab

30 Minutes

Details

The following sections describe the detailed steps to be completed for this lab.

1. Open the Product Form Parts Editor

Open the pre-configured **Product Form** located in day 4 and go the **Parts Editor**.

- Navigate to **Developer | Page Definitions | Training | Day 4 | Lab 13**
- Select **Lab 13** to reveal its list of **Hub 3s.**
- Select **Product Form** from the list of **Hubs** in slice 2.
- Select **Parts** from the **Action** menu in slice 3.

2. Add Form Input Controls

The **Product Form** layout is nearly complete; but it includes only the text input for the **Product Name**. Text inputs need to be added for the **Product Code** and **Rate** values.

- Select the top row in the 2nd column and click **+ Add Part** from the menu in the right margin.
- Select a text input part from the tree view at **TrakSYS | Forms | Text Box** and use the properties below.

Part ID	Caption	Form Key	Total Width	Variant Data Type
InputProductCode	Code	Model.ProductCode	6	None
InputRate	Rate	Model. TheoreticalRateCalculationUnitsPerMinute	6	None

3. Open the Product Form Script Editor

Open the pre-configured **Product Form** to the **Script** editor.

- From the **Parts** editor, click the **Script** option from the top menu.

4. Add Form Load Script

As a form loads, the data object (model) that will be edited (**Product** in this case) must be loaded from the database and its properties copied into the **Values Dictionary**. If the form is in New mode, then a new model must be created and populated with its parent reference (**Product Set** in this case). This is typically done in the **ContentPage_Init** method of the form.

Locate the **ContentPage_Init** method and **uncomment** the lines of code **highlighted** below. As you do so, read and attempt to follow the logic.

```
/// ***********************************************************
protected override bool ContentPage_Init()
{
  // place IsNew and IsEdit into Values to make it easier to show/hide parts in the UI
  this.Ets.Values["Model.IsNew"] = this.IsNew;
  this.Ets.Values["Model.IsEdit"] = this.IsEdit;
  this.Ets.Pages.IsClientDirtyCheckEnabled = true;

  if (this.IsNew)
  {
    // create new object from definition or parent
    var parent = this.Ets.Api.Data.DbProductSet.Load.ByID(this.ProductSetID)
        .ThrowIfLoadFailed("ProductSetID", this.ProductSetID);
    _model = this.Ets.Api.Data.DbProduct.Create.FromParentProductSet(parent)
        .ThrowIfNull("Unable to create new Product.");
    _model. TheoreticalRateCalculationUnitsPerMinute = "0"; // default numerical value
  }
  else
  {
    // load existing object
    _model = this.Ets.Api.Data.DbProduct.Load.ByID(this.ProductID)
        .ThrowIfLoadFailed("ProductID", this.ProductID);
  }

  // copy model into Values to make it accessible to input parts
  if (!this.Ets.Values.CopyFromModel(_model, "Model.")) return false;
  return true;
}
```

As the form renders, the inputs **Parts** draw the necessary values from the **Values Dictionary**.

5. Add Form Post Script

After a user clicks the **Save** button, the data must be extracted from the form inputs and re-populated to the model (**Product** in this case). This is typically done in the **Save_Click** method.

Locate the **Save_Click** method and **uncomment** the lines of code **highlighted** and **bolded** below. As you do so, read and attempt to follow the logic.

```
/// ***************************************************************
private void Save_Click(object sender, EventArgs e)
{
  // populate _model with inputs from the UI
  if (!this.Ets.Form.UpdateModelWithKeyPrefix(_model, "Model.")) return;

  // assign any calculated data prior to save
  if (this.IsNew)
  {
    _model.Attribute01 = "0";
    _model.Attribute02 = "0";
    _model.Attribute03 = "0";
  }

  // validate the entity
  if (!this.ValidateModelData()) return;

  // save the entity
  if (!this.SaveModelData()) return;

  // has successfully saved, redirect to success
  this.Ets.Pages.RedirectToSuccessUrl();
}
```

6. Add Form Validate Script

After the form inputs are populated to the model (**Product** in this case), the model must be validated. This is typically done in the **ValidateModelData** method.

Locate the **ValidateModelData** method and **uncomment** the lines of code **highlighted** and **bolded** below. As you do so, read and attempt to follow the logic.

```
/// ***************************************************************
private bool ValidateModelData()
{
  // perform built-in validation
  var coreValidate = this.Ets.Api.Data.DbProduct.ValidateForMerge(_model, this.IsNew);
  if (!this.Ets.Form.AddResultMessagesWithPrefixIfFailed(coreValidate, "Model.")) return false;

  // return validation results
  return this.Ets.Form.IsValid;
}
```

7. Add Form Save Script

If the form validation passes, the last step is to save the model (**Product** in this case) back to the TrakSYS database. This is typically done in the **SaveModelData** method.

Locate the **SaveModelData** method and **uncomment** the lines of code **highlighted** and **bolded** below. As you do so, read and attempt to follow the logic.

```
/// ***************************************************************
private bool SaveModelData()
{
  var uow = this.Ets.Api.CreateUnitOfWork();

  // save model (validation can be ignored as it has already been done)
  this.Ets.Api.Data.DbProduct.MergeIgnoreValidation(_model, this.IsNew, uow).ThrowIfFailed();
  // execute final save
  var result = uow.ExecuteReturnsResultObject();
  if(!result.Success)
  {
    this.Ets.Debug.FailFromResultMessages(result.Messages);
    return false;
  }

  return true;
}
```

8. Save the Script Changes

After making the **Script** changes described above, be sure to commit the changes using the **Save** button at the top left of the editor.

9. Access the Product List Page and test the Form

The form is now completed. Due to the changes made, it now needs a ProductSetID or a ProductID to be provided and will not longer function when navigated to directly. Test the new form by accessing the pre-configured **Product List Page** located at **Training | Day 4 | Lab 13 | Product List**. This is a simplified interface for users with limited rights to see and edit the **Products** from **Packaging Product Set B**. Clicking **New Product** or **Edit** on an existing **Product** calls the new custom **Product Form**.

Conclusion

In this exercise, you have used the development tools in TrakSYS to create a custom **Product** editing form. This custom **Page** retains the look and feel of the standard configuration **Pages** and can be integrated anywhere within the TrakSYS user interface.

It is recommended that TrakSYS developers becomes very familiar with the API and techniques to load, render, receive, validate and save information to and from the user interface. These patterns can be used to create any type of data input and capture form or user interface within a TrakSYS solution.

Common Mistakes

None

Further Exploration

Custom Product Forms

While this lab was built upon using a pre-configured template, it can also be completed using the standard **Generic Edit Form** visual page template. Try creating your own version of this **Product Form** but starting with the template.

To do this, create a new page under **Training | Day 4 | Lab 13**. For the page type, select **Visual | TrakSYS | Generic Edit Form**. Some parts have already been added and some script already exists. Use your existing **Product Form** and try to recreate it from the template, adding or removing capabilities as desired.

Lab 14 | Library Pages/Parts and Custom Types

Overview

TrakSYS has a foundation of **repeatability**, **reusability**, and **standardization**. When building screens, they are commonly one-time uses. While instancing pages and duplicating pages are useful, they do not have a form of **central governance**. The same concept applies to individual components of a page. When logic is needed in the page script to determine how a part will function, the reusability of that part decreases.

Library Parts/Pages provide a way to create standardized components in the software that act similarly to pre-compiled content. They can be complex combinations of logic and visuals that are controlled through minimal property options. While the end-user of the part may be more limited, it can typically provide for easier solutions to common, complex issues.

You will examine a few pieces of custom content, developed specifically for this lab. The parts will be simple to allow for the completion of the content in the allotted time. While the development of a new part/page is typically not a short process, it is a powerful tool to have.

Estimated Time to Complete This Lab

30 Minutes

Details

The following sections describe the detailed steps to be completed for this lab.

1. Examine the Example Visual Page

Content Parts are a combination of HTML and other scripts that create a final visual component on a screen. Since all the end-user sees is the result, it is possible to have multiple parts render in the exact same fashion, even though the underlying mechanics may not be the same.

A page has been designed to show you multiple versions of a part that all render nearly identically, even though they are designed in different ways.

- Navigate to **Training | Day 4 | Lab 14 | Content Part Example**.
- View the 3 different parts being displayed.
- Open the Parts Editor.

2. Examine the Different Parts

Each part, although similar in appearance, has a different set of properties. Below is a brief description of each version. Open the **Edit** menus for each part to view their properties.

TileTrakSYS Tile (Dynamic Width): This TrakSYS Content Part displays up to 6 lines of text in a TrakSYS-styled box. It is sized using Bootstrap CSS to dynamically resize to its container and features on-click capabilities.

TileFragment View Fragment: This TrakSYS Content Part allows for the developer to write their own HTML and ASP.Net controls. It can reference values from the values dictionary to allow for dynamic rendering. This example was created by copying the HTML from a **Tile (Dynamic Width)** TrakSYS Content Part.

TileCustom Custom Tile: This is a custom part. It was developed using the TrakSYS **Tile (Dynamic Width)** Content Part as the baseline for its appearance. The first line of text has been set and cannot be changed, but the bottom line of text is Dynamic. It features two properties that allow the user to load a System by an ID, which will then be used to determine what is shown as the bottom line of text.

3. Examine the Custom Tile's Source

Next, look at the scripting behind the **Custom Tile**.

- Navigate to **Developer | Content | Parts**.
- Select **Demo Parts | Custom Tile**.
- Select the **</>View** menu option from under **Actions** in slice 3.

Notice that the HTML used here is the same as HTML that is used for the view fragment.

- Select the **C# Script** menu option from under the **View** header.

Notice the two **ContentProperty** decorations at the top of the script. These are the two properties that were displayed when the part was viewed in the previous step.

4. Examine the Custom Part: Product Counter Tile

For the next part of the lab, you will be looking at a Custom Content Part that is designed to work with the rest of the lab.

- Navigate to **Developer | Content | Parts**.
- Select **Demo Parts | Product Counter Tile**.
- Select the **</>View** menu option from under **Actions** in slice 3. Examine the script.
- Select the **C# Script** menu option from under the **View** header. Examine the script.

This part is designed to return information about Products based upon two filters that are expected to be provided by the User. It looks for the **Product Set ID** and a **Product Type Key**.

5. Create the Custom Type

The part is designed to look for the number of Products in a Product Set that are assigned to a specific **Product Type**. For this to function, you need to configure a Product Type.

- Navigate to **Configuration | Types**.
- Find and select **Product Types**. Select the **+ New** menu option from slice 2.
- Configure a **Product Type** with the following properties…

Name	Synchronized
Key	SYNC

6. Assign the Custom Type

Once created, these custom types can be used with their matching entities. Navigate to your products and assign them to the new custom type.

- Navigate to **Configuration | Products**.
- Select the **Packaging B** Product Set.
- For each product, set the **Product Type** property to your new **Synchronized** Product Type.

7. Configure the Custom Part: Product Type Tiles

Now you can test out the Custom Part with the new configuration options. For this to work, you will need your **Product Set ID** and the **Product Type Key**. Find them in your Configuration and write them down before continuing. **Prod Type Key** has been filled out for you based upon the previous steps.

Prod Set ID	Prod Type Key
	SYNC

- Navigate to **Training | Day 4 | Lab 14 | Content Part Example**.
- Open the **Parts Editor**. Click Row 2 and select to **+ Add Part.**
- Select **Custom | Demo Parts | Product Counter Tile**.
- Set the following properties for your new part…

PartID	TileProductCounter
Product Set ID	<Your Product Set ID>
Product Type Key	<Your Product Type Key>
Width	4

- Use the **Preview** button to view your page.

Conclusion

In this exercise, you gained some familiarity with Custom Parts and Pages. This lab is intended to provide a high-level introduction to some of the patterns and capabilities of the Parts and Pages frameworks. The potential benefits and gains of Custom Parts and Pages grows as you gain more knowledge of the TrakSYS platform.

You also worked with an example of TrakSYS extensibility in the form of Custom Types. Like Capture fields on data records and Attribute fields on configuration records, Custom Types can be used to provide an additional, implementation-specific layer of grouping. This is the simplest of the extended configuration capabilities but demonstrates some of the built-in TrakSYS extensibility.

Common Mistakes

None

Further Exploration

Along with Custom Types, there are other entities to assist with handling implementation-specific needs through standard configuration.

Custom Properties

For implementations where additional fields are needed to handle custom logic processing, Custom Properties can be used. This will allow for standardized extensions to the properties of an entity without having to use custom tables.

To add a custom property, go to **Configuration | Custom Properties**. Create a Scheme, a Group, and then a Property. This will determine what new options are going to be added to the standard configuration. To determine where it will appear, the **Scheme** will need to be related to an Entity, which is also done through the same menus.

Once the Custom Properties are configured, they can be configured through the standard configuration menus for the matching entities. To access the Custom Properties for implementation-specific screens, this can be done through scripting or through SQL.

Scripting Example (Using a System) – Can be found in **C:/Scripts/CustomPropertyExample.txt**

```
// custom properties dictionary
var cp = _system.CustomProperties;
// to load the values from the model into the values dictionary
this.Ets.Values.CopyFromModel(_system.CustomProperties, "System.CustomProperty.");

// to save the custom properties in a form
foreach (var kvp in _system.CustomProperties)
{
  var cp = kvp.Value;
  string formKey = "System.CustomProperty." + kvp.Key;
  cp.Value = this.Ets.Form.UpdateModelValueByFormKey(formKey, cp.Value);
}
this.Ets.Api.Data.CustomProperty.MergeCustomProperties(_system, uow).ThrowIfFailed();
```

SQL – go to **Configuration | Custom Properties**. Under **Schemes** header, click the **Generate Views** menu option. This will create a view for every entity that contains the ID of the entity and all the custom properties associated with it named **viewCustomProperty<EntityName>**.

Custom Permissions

For implementations where specific permissions need to be granted or withheld, Custom Permissions can be used. This allows for standardized permission control without the need to cache additional user data in other formats.

Custom Permissions can only be configured through the **Root Site** of an implementation. Navigate to **Root Site | Adminsitration | Permissions**. Permissions can then be configured to have a combination of **Items** and **Actions**. The permissions are then available to be configured on all **Roles** like the standard permission sets, including the ability for use with **User Groups**.

Once configured, custom permissions will be enforced through scripting. Below are two examples of referencing Custom Permissions which can be found in
C:/Scripts/CustomPermissionsExample.txt

```
	this.Ets.HasPermission("set", "item", "optional:action");
	this.Ets.Api.User.HasPermission("login", "password", "permissionSet", "permissionItem", 
"optional:permissionAction");
```

Try creating this example: Create a new Custom Permission Set and identify a permission for accessing your **Product Form** from the previous lab.

1. Create a Grid Permission Set for “Training”
2. Create a Permission Item for “Product”
3. Create a Permission Action for “Simplified Edit”
4. Create or Open a Role – Set your new “Training – Product – Simplified Edit” to true
5. Assign that role to a User
6. On the **Product List** page from Lab 13, use the API example to check if the user has permissions. This will return a Boolean that you can pass into the Values Dictionary. Use that to Enable/Disable the “Edit” button from the Product List.
7. Add another permission check on the **Product Form** to see if the user has permission to be on the page. If the check returns a false Boolean, redirect them somewhere else, such as an error page.

Lab 15 | Logic Service and DMS Module Scripting

Overview

TrakSYS is a flexible and extensible solution building platform. Its **API** and .NET scripting integration allows solution-specific business rules to be added to the real-time monitoring and processing occurring within the **Logic Service**.

The TrakSYS **Data Management Service** allows executable **Modules** to be defined that can pre-aggregate data, connect to external business systems, and move data in and out of the TrakSYS database.

In this assignment, you will complete two tasks, first creating a **DMS Module**, then creating a **Logic Service Script Class** to both run logic and to trigger the **DMS Module**.

For the **DMS Module**, you will create a **Product Import Module** that will connect to an external **Product** list (maintained in Excel). This **Module** will synchronize the contents of the Excel list with the TrakSYS **Products** in the **Packaging B Product Set**.

For the **System Script Class**, you will create a script that will execute within the **Logic Service** engine. It will do two things. First, it will intercept execution at the time that each stoppage **Event** occurs on the System, making a minor alteration to the **Event** Record. Second, it will trigger the **DMS Module** created in the first part of the lab.

Estimated Time to Complete This Lab

30 Minutes

Details

The following sections describe the detailed steps to be completed for this lab.

1. Create a System Script Class

Define a new **System Script Class** (of the **System** type) to hold the **Event** processing script.

- Navigate to **Developer | Scripts | Logic**.
- Locate and select the **Packaging Scripts** Group.
- Select the **+ New** menu option under **Scripts** in slice 2.
- Select the **System Script Class** option, use the following properties and click **Save**.

Name	Packaging System

2. Rename the .NET Class

TrakSYS populates the **System Script** entity with a default class implementation. Change the .NET class name to **PackagingScript**.

- Navigate to **Developer | Scripts | Logic**.

- Locate and select the **Packaging System** Script Class.
- Select the **C# Script** option from the **Actions** menu in slice 3

Rename the script class by changing…

```
public class SystemScript : ETS.Core.Scripting.SystemScriptClassBase
```

… to …

```
public class PackagingScript : ETS.Core.Scripting.SystemScriptClassBase
```

This will be the name that is later used to assign the Script to the System. It will let the Logic Service know when to execute the new logic.

3. Add the Event Processing Script

In order to affect **Events** as they begin, script must be introduced into the **PostScanEventStart** method. Add .NET script to this method that loads the model object for the new **Event**, adds text to the **Notes** property, then saves the changes to the database. The **context** argument to this method is an object that contains an **EventID** property holding the ID of the **Event** that was just triggered in the **Logic Service**.

- Locate the following method signature in the script…

```
public override void PostScanEventStart(IPostScanEventStartContext context) {}
```

- Add the following script **between** the method's curly braces {}. The code for this script can be found at **C:\Scripts\SystemScriptEvent.txt**.

```
// load the event that just started
var ev = context.Api.Data.DbEvent.Load.ByID(context.EventID)
    .ThrowIfLoadFailed("ID", context.EventID);

// modify its Notes property
ev.Notes = "Note automatically added.";

// save
context.Api.Data.DbEvent.Save.UpdateExisting(ev).ThrowIfFailed();
```

- Save and close the **Script** editor.

4. Associate the Script Class with the System

Once the **System Script** is created, it must be associated with the **Packaging Line 5 System**. Scripts may be associated to one or more entities by setting the entity's **Script Class Name** property to the name of the defined .NET class. Once associated, the **System** will execute the script defined within whenever a new **Event** begins.

- Navigate to **Configuration | Systems | Packaging | Packaging B**.
- Locate and select the **Packaging Line 5 System**.
- Click the **Edit** menu option to access its properties and modify as shown below...

Script Class Name (Advanced Tab)	PackagingScript

5. Restart the Logic Service

Ensure that the **TrakSYS Logic Service** is re-started. The service will link the **Packaging Line 5 System** to the new **Script** class and execute it when new **Events** occur.

6. Generate an Event and view the Results

Use the **Line 5 Overview** to start and end **Events** on the **Packaging Line 5 System**. Start and end at least one **Event**. Afterwards, Acknowledge the Event record and view the **Notes** field.

- Navigate to **Training | Day 4 | Lab 15 | Line 5 Overview**.
- Click the **Planned** button, create a new Job, and start that Job.
- Trigger one of the **Manual Events**. **Start** and then **End** the event.
- Click the **Acknowledge Tile** to open the list of **Events** requiring attention.
- Select an **Event** from the list to reveal its properties in the right margin.

The **Notes** fields should contain the text inserted by the **Script** (“Note automatically added.”).

You have now completed the first part of the lab involving the creation of a simple Logic Service Script Class.

7. Configure a Module

The second part of the lab involves the creation of a **DMS Module**. Modules are the units of execution for the Data Management service. A Module may contain one or more **Module Steps** defining specific extensibility functions. Create a **Module** to hold the script that will import and synchronize **Product** information.

- Navigate to **Developer | Modules**.
- Select the **+ New** menu option under **Modules** and use the following properties.

Name	Product Import
Host	<name of your specific training server>
Trigger Mode	None
Trigger Key	IMPORT

Since this **Module** will be executed manually for the lab (and not on a periodic basis), set the **Trigger Mode** to **None**.

8. Configure a Module Script Step

Define a new **Module Step** (of the **Script** type) to hold the **Product** import processing script.

- Navigate to **Developer | Modules**.
- Locate and select the **Product Import** Module.
- Select the **+ New** menu option under **Steps** in slice 2.
- Select the **Script Module Step** option.
- Use the property below and click **Save.**

Name	Import

9. Add the Product Import Script

TrakSYS populates the **Module Script Step** entity with a default class implementation. Add .NET script to the **Execute(IModuleContext ctx)** method that imports **Product** data from an external Excel spreadsheet located on the file system. The **Products** in the spreadsheet should be merged with the **Packaging Product Set**.

- Navigate to **Developer | Modules**.
- Locate and select the **Product Import** Module.
- Locate the **Import** Script Module Step.
- Select the **C# Script** option from the **Actions** menu in slice 3
- Add the following script **between** the **Execute(IModuleContext ctx)** method's curly braces {}. The code for this script can be found at **C:\Scripts\ImportScript.txt**.

```
      var uow = ctx.Api.CreateUnitOfWork();

      // load the target product set
      string sql = "SELECT TOP 1 * FROM tProductSet WHERE [Key] = 'PACK.B'";
      var productSet = ctx.Api.Data.DbProductSet.Load.WithSql(sql).ThrowIfLoadFailed("Name",
"Packaging");
      var productType = ctx.Api.Data.DbProductType.Load.ByKey("SYNC").ThrowIfLoadFailed("Key",
"SYNC");

      // load the product from the external excel to a data table
      string connectionString = @"
Provider=Microsoft.ACE.OLEDB.12.0;
Data Source=C:\Scripts\ProductList.xls;
Extended Properties=""Excel 12.0; HDR=Yes; IMEX=1"";
";
      System.Data.DataTable dt = ctx.Api.Util.Db.GetDataTable("SELECT * FROM [ProductList$]",
connectionString).ThrowIfFailed();

      // loop the data table
      int productsImported = 0;
      foreach (System.Data.DataRow row in dt.Rows)
      {
        // get fields from Excel row
        string productCode = row.GetString("Code", "");
        string productName = row.GetString("Name", "");
        int rate = row.GetInteger("Rate", 0);

        // determine if Product exists
        var product = ctx.Api.Data.DbProduct.Load.ByProductCodeAndProductSetID(productCode,
productSet.ID);
        if (product == null)
        {
          // add product
          product =
ctx.Api.Data.DbProduct.Create.FromParentProductSet(productSet).ThrowIfNull("Error Creating Product
From Set");
          product.Name = productName;
          product.ProductCode = productCode;
          product.ProductTypeID = productType.ID;
          product.TheoreticalRateCalculationUnitsPerMinute = rate;
          product.Attribute01 = "0";
          product.Attribute02 = "0";
          product.Attribute03 = "0";
          product.Enabled = true;
          ctx.Api.Data.DbProduct.Save.InsertAsNew(product, uow).ThrowIfFailed();

          productsImported++;
        }
        else
        {
          // check for product type
          if(product.ProductTypeID != productType.ID) continue;

          // update product
          product.Name = productName;
          product.Attribute01 = rate.ToVariantString();
          ctx.Api.Data.DbProduct.Save.UpdateExisting(product, uow).ThrowIfFailed();

          productsImported++;
        }
      }

      // execute the changes
      uow.ExecuteReturnsResultObject().ThrowIfFailed();
      ctx.Api.Util.Log.WriteInformation("Successfully Imported {0:N0}
products.".FormatWith(productsImported), "Lab 15");
      return true;
```

- Save and close the **Script** editor.

10. Examine the Product Import Excel File

The target for Product import is a simple Excel spreadsheet called **ProductList.xls** located in the **C:\Scripts** folder. If contains columns for **Code**, **Name** and **Rate**. Any row in this sheet will be matched with **Products** in the **Packaging Product Set**. If the **Code** exists, the **Product** will be updated, if it does not exist, it will be inserted. You may add new rows or alter values as desired, however remember to close the file before running the **Import Module**.

11. Start the Data Management Service

While the individual **Modules** can be started and stopped independently, the service itself must be started. Ensure that the Data Management Service has been installed and is running.

- Open the **TrakSYS Installation Manager** and click on the **Services** tab.
- Locate the **Data Management Service**.
  - If the Status is **Not Installed**, use the **Install Data Management** option on the right-hand menu.
  - If the Status is **Stopped**, click on the **Data Management Service** row and then click the **Start** option from the bottom buttons

12. Start and execute the Module

To execute the **Import Module**, it must be first **started**, and then **executed** using the **Administration** dashboard for the **Data Management Service**.

- Navigate to **Administration | Services | <your server name> [Data Management]**.
- Locate the **Module** in the list and click the **Start** action on the row.
- After the row turns green (indicated it is loaded), click the **Execute** action on the row. This should run the Module script.
- After the Module has Executed, click the **Logs** action and view the log records.

To **re-execute** the **Import Module**, click the **Execute** icon in the row. The **Module** only needs to be restarted if the script is changed.

13. Examine the Packaging Product Set

After the **Import Module** is executed, locate and view the **Packaging Product Set** (**Configuration | Products**) and ensure that the contents of the **ProductList.xls** have been imported or updated.

14. Add the Job Processing Script

For the last part of the lab, you will modify your existing Logic Service System Script Class to trigger your Import Module on a Job end.

- Navigate to **Developer | Scripts | Logic | Packaging Scripts | Packaging System**
- Click the **C# Script** menu option from slice 3 and locate the following method signature in the script…

```
public override void PostScanJobEnd(IPostScanJobEndContext context) {}
```

- Add the following script **between** the method's curly braces {}. The code for this script can be found at **C:\Scripts\SystemScriptJob.txt**.

```
// identify parameters if needed
var moduleTriggerKey = "IMPORT";

// create unit of work
var uow = context.Api.CreateUnitOfWork();

// load the job information
var job = context.Api.Data.DbJob.Load.ByID(context.JobID).ThrowIfLoadFailed("ID", context.JobID);

// load the module information
var module =
context.Api.Data.DbModule.Load.ByTriggerKey(moduleTriggerKey).ThrowIfLoadFailed("TriggerKey",
moduleTriggerKey);

// create module trigger
var moduleTrigger = context.Api.Data.DbModuleTrigger.Create.FromParentNone();
moduleTrigger.TriggerDateTime = context.Now;
moduleTrigger.TriggerKey = moduleTriggerKey;
//moduleTrigger.Args = string.Empty;

// save module trigger
context.Api.Data.DbModuleTrigger.Save.InsertAsNew(moduleTrigger, uow);

// add additional functionality if needed

// execute the changes
var result = uow.ExecuteReturnsResultObject();
if(!result.Success)
{
  context.Api.Util.Log.WriteErrorsFromResultObject(result, "System{0}.PostJob [ LS Script Error
]".FormatWith(this.ID));
}

// write to log
context.Api.Util.Log.WriteInformation("Job '{0}' has ended. Triggered {1}
Module.".FormatWith(job.Name, moduleTriggerKey), "System{0}.PostJob [ LS Script Error
]".FormatWith(this.ID));
```

- Save and close the **Script** editor.

15. Restart the Logic Service

Ensure that the **TrakSYS Logic Service** is re-started. The service will reload the **Packaging Line 5 System** to use the changed Script class and will now execute the additional logic when a **Job End** occurs.

16. End the Job and view the Results

Use the **Line 5 Overview** to end the job you started on the **Packaging Line 5 System**. Start and end at least one **Event**. Afterwards, view the **Logs** for the **Logic Service** as well as the **DMS Module**.

- Navigate to **Training | Day 4 | Lab 15 | Line 5 Overview**.
- Click the **End** button at the top left of screen under **Job Header**.
- Navigate to **Administration | Services | OCV [Logic]**. Select the **Logs** option from the related menu and view the Logs.
- Navigate to **Administration | Services | <your server name> [Data Management]**. Select the **Logs** row option for the **Import Module** and view the Logs (The Module Log button is on the far right next to the execute button).

Both sets of logs should have a record from when the Job ended.

Conclusion

In this exercise, you have extended the standard **Logic Service** processing with solution-specific business rules using .NET scripting and the TrakSYS **API**. This is powerful technique to leverage the real-time context information available within the TrakSYS services to enable data manipulation, extended capture, and any other type of special processing required.

You have also see how **Modules** defined for the **Data Management Service** can be used to connect to external business systems, enabling the import or export of data to and from the TrakSYS database. This is very common pattern for connecting to **ERP** and other manufacturing systems to import and synchronize master data. Nearly any type of business system can be accessed using **Modules**, .NET script, the TrakSYS API and the external software's interface of choice (direct connection, APIs, file transfers, XML, web services, etc…).

You were also able to trigger that **Module** from defined logic within the **Logic Service**. While this was a simulated example, the pattern can be expanded to work with any data based upon any trigger that can be defined within TrakSYS.

Common Mistakes

None

Further Exploration

None

Lab 16 | Sites, Translations and Auditing

Overview

In a **Multi-Site implementation**, TrakSYS features the ability to provide governance across multiple sites through the **Root Site**. This includes certain features and capabilities that are intended to be uniform throughout the implementation, such as **Language Translations**.

In this assignment, you will explore the entities in the **Root Site** and see the differences in options. From the Root Site, you will then create **Custom Translations** that can be used in your solution-specific interfaces.

While **Auditing** is not covered in the main lab, Auditing can be explored as **Further Exploration**.

Estimated Time to Complete This Lab

30 Minutes

Details

The following sections describe the detailed steps to be completed for this lab.

1. Navigating to the Root Site

In any multi-site implementation, a **Root Site** is automatically added. This site is not the same as the others, as it is not intended to be used for production purposes and will have a limited set of entities. Navigate to the Root Site and examine the differences.

- Click on the **User** in the top right of the screen. Select **Sites** from the menu.
- Select **Root** from the list of sites displayed.

By default, there are no pages configured in the Root Site, so you will immediately be taken into the Configuration section. Notice that there are no Systems or Tags, and that you cannot navigate to Product configuration either. There are no p**roduction-related** configuration options available, only configuration that is used for context and reporting.

2. Root Site Entities

While it is limited, the Root Site does have specific entities and features that are unique to it. Under Administration, **Custom Permissions** and **Site** properties can be configured, and there are more **Users and Roles** combinations available. Under Configuration, **Translations** can be configured.

Navigate through each section and view the differences mentioned.

3. Configure a Translation

In some installations, **Pages** must be created to support users speaking and reading different languages. There are some **Translations** string already configured in the database for this **Page**. The following steps will show you how to create a new set of **Translations** for a **Spanish** user.

4. Create a Locale

The first step is to create the **Locale** to identify what language it is, and how it will be linked to a User.

- In the Root Site, navigate to **Configuration | Translations | Locales**.
- Use the **+ New** button under Locales to create a new **Locale**.
- Use the following properties below.

Culture Name	es

The second step would be to create a **Resource Group**. In this lab, the **Demonstration** Resource Group has been preconfigured for you. View the **Resource Group** and take note of the **Key** property. The Resource Group's Key will be used as the first half of a translation string. To find the Resource group navigate to **Root Site | Configuration | Translations.**

5. Create a Resource Item

The third step is to create the translation item, which identifies what you intend to translate.

- Navigate to **Root Site | Configuration | Translations**
- In **Translations**, select the **Demonstration** Resource Group.
- Select the **+ New** menu option under **Items** and Configure the **Resource Item** with the following properties.

Key	Edit
DefaultValue	Edit

The **Key** property will be used as the second half of the translation string.

6. Create the Translation

The last step is to create the actual translation of the text. This can be repeated for every **Locale** identified. Create a Translation of **Edit** for the **es** locale.

- Select your newly created Resource Item**, Edit**.
- In slice 3, select **Translations** from the **related** menu options.
- Select the **+ New** menu option under **Translations** in slice 2. Set the following properties.

Resource Locale	es
Value	Editar

The other properties will remain locked and are shown for reference. Once a **Translation** is made for a locale, it will no longer be displayed as an option for **Resource Locale** for that **Resource Item**. This is to prevent having multiple translations for a single **Resource Item** and **Locale**.

7. Replace a Header with a Translation

Now that the Translation has been properly configured, it can be used in a page. Use your Translation as part of a **Header** on a **Page** using **Translation** syntax.

- Navigate back to **OCV** by clicking the user name at the top right.
- Navigate to **Training | Day 4 | Lab 16 | Translations**. Open the **Parts Editor** using the developer tools at the bottom left of the screen, in the footer.
- Open the properties for the **HeaderTranslation** Part using the **Edit** button.
- Open the translation picker for the **Text** property (it will show in the right margin, blue button with diagonal arrow).
- Search for the term “Edit”. Select and assign the appropriate **Translation**.
- Click **Save** to commit the changes.

After configuring, open the end-user view of the **Page**. The **HeaderTranslation** should show with the proper **English Translation**.

8. View the Translations

To view the **Page** using the **Spanish** translation, logout of TrakSYS and log back in using the **Administrator-ES** account (who is configured with the **es Locale**).

- **Logout** using the option in the upper right menu.
- From the same menu open the **Login Page**.
- Enter **Administrator-ES** for the **Login**.
- Leave the **Password** blank.
- Navigate back to **Training | Day 4 | Lab 16 | Translations**.

Conclusion

In this exercise, you explored the Root Site and some its differences, including the **Translation** entities. You configured a full translation and referenced it in a part on a visual page.

For all standard content, translations will occur automatically if the matching language pack is installed. For this training, the **es** language pack has been installed ahead of time. While logged into the **Administrator-ES** account, you can also view translations on any existing content page, as well as the standard navigations.

In most implementations, bulk translations are not completed through the pattern shown in the lab. They are typically imported in a similar fashion to the **DMS Module** examples from previous sections. However, adjustments and minor translations can be completed through the web to prevent the need to frequently run an import module.

Common Mistakes

None

Further Exploration

Auditing

In regulated environments, Auditing is essential. To enable auditing, there are several steps to ensure that it is properly enforced.

First, determine the auditing settings. In the **Installation Manager**, select the **TrakSYS Settings** option from the **Overview** tab. Under the **Audit** group of settings, identify how you wish to audit the Configuration, Developer, Administration, and Data records.

Then, create the **Audit Triggers**. This is done through the **Installation Manager**, in the **Database** tab. It will require the proper SQL credentials to create the triggers.

Once the Audit Triggers are created, the last step is the reload the web. In the TS Web, navigate to **Administration | Settings**. Find and click the **Reload Application** button. This will reload the web to ensure that the **Audit Settings** take effect on the user interfaces.

After those steps are completed, Auditing is enabled. Navigate to any entity that you are auditing and attempt to make a change. After the changes have been made, you will be able to view the **Audit Trail** through the entity's item slice.

Root Site Configuration

In a multi-site implementation, each site has its own set of configuration. However, it is common to want some configuration to be standardized between sites. A common example of this is **Event Codes**, since they are generally used as the high-level reporting mechanism.

Create your own **Event Codes** as a **Root Site** entity. Navigate to the Root Site and go to **Configuration | Categories | Event Codes**. Create a new Group and a new Event Code. Once completed, go back to **OCV**. Navigate to **Configuration | Categories | Event** and select any Event Category. Select the **Set Event Code** menu option under **Actions** in slice 3. You should see your new **Root Site Event Code** available for selection. However, if you navigate to **Configuration | Categories | Event Codes** in OCV, you won't see it available for editing there.

This allows for the creation of certain entities that can be used in configuration but are controlled by a different set of users and permissions. It is useful for large implementation with multiple sites that require some central governance.

Translations: TODOs

During the development of a page, the process of creating translations for every string can lead to slow progress. If the developer is required to stop page development to create a new **Translation**, it leads to a disjointed development process and the quality of the page typically suffers. To alleviate that, translations also feature a **TODO:** syntax to create placeholders for future translations that can be configured after the page development is complete.

In OCV, navigate to the **Training | Day 4 | Lab 16 | Translations** page. Open the parts editor and create a new **Header** part. Set the text property to "**TODO:My Placeholder Text**".

In the Root Site navigate to **Configuration | Translations | TODOs**. This page will look through all existing pages and search for the **TODO:** syntax. In larger implementations, this may take some time to load. Once completed, it will allow for the creation of **Translations** based upon the provided results. Assign your translation a **Resource Group**, provide a **Key**, and provide a **Default Value**. A new **Translation Item** will be created and can be further configured as usual.