# OFFICE AMS: Client Side Rendering

|  |  |
| --- | --- |
| Summary: | Applies to: |
| This sample shows how to customize a field type with Client-Side Rendering (also called CSR, JS Link) technology in SharePoint 2013. | * Office 365 Multi Tenant (MT) * Office 365 Dedicated (D) * SharePoint 2013 on-premises |
| Solution: | Branding.ClientSideRendering, version 1.0 |
| Author: | Leo Qian (Canviz LLC)  Tyler Lu (Canviz LLC)  Cindy Yan (Canviz LLC)  Todd Baginski (Canviz LLC) |
| //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // THIS CODE IS PROVIDED \*AS IS\* WITHOUT WARRANTY OF  // ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING ANY  // IMPLIED WARRANTIES OF FITNESS FOR A PARTICULAR  // PURPOSE, MERCHANTABILITY, OR NON-INFRINGEMENT.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | |

# Scenario: Client side rending samples

Client Side Rendering is a new concept in SharePoint 2013. It’s provides you with a mechanism that allows you to render your own output for a set of controls that are hosted in a SharePoint page (list views, display, add and edit forms). This mechanism enables you to use well-known technologies, such as HTML and JavaScript, to define the rendering logic of custom and predefined field types.

JSLink files have the ability to quickly and easily change how a list’s views and forms are rendered. More specifically, how the fields in that list should be displayed.

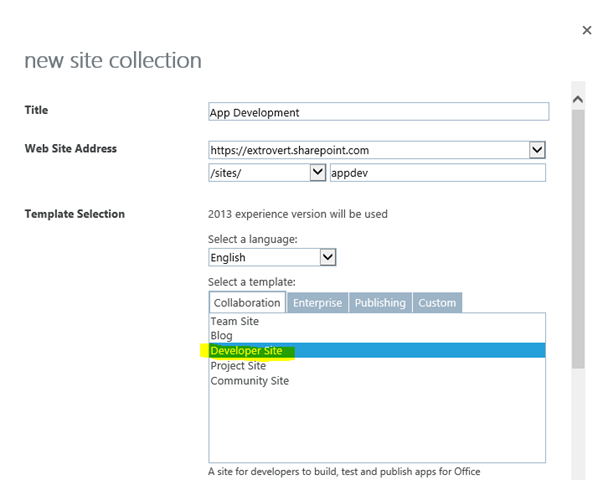
The Client Side Rendering JS files in this sample were taken from [an MSDN code sample by Muawiyah Shannak](http://code.msdn.microsoft.com/office/Client-side-rendering-JS-2ed3538a). They have been included in this code sample to demonstrate how the remote provisioning pattern may be used to deploy Client Side Rendering components and associated them with views and forms in a SharePoint list.

# Configuration & Deployment

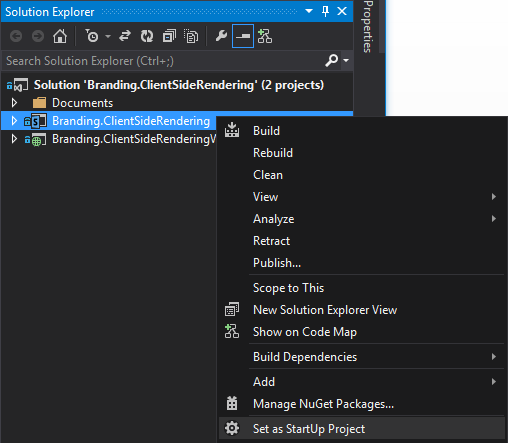
In order for the Client Side Rendering sample to render correctly, you must first create your SharePoint site collection.

## SharePoint

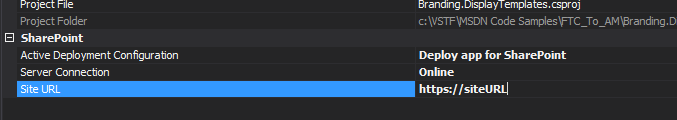
1. Navigate to your SharePoint tenancy and create a new site collection using the Developer Site template in the Collaboration tab.



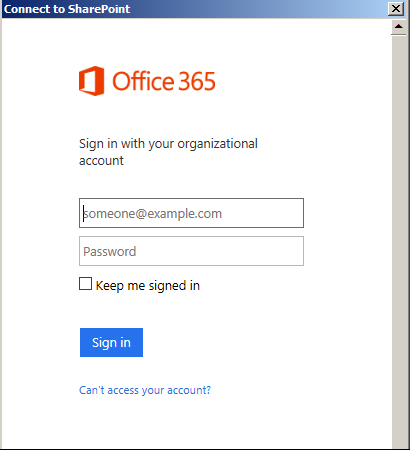
1. Once the site collection is created, open the *Branding.ClientSideRendering.sln* file with Visual Studio 2013.
2. If the *Branding.ClientSideRendering* project isn’t the StartUp project, set it as the StartUp project.



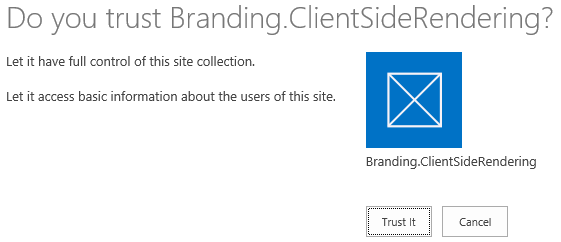
1. In the Solution Explorer, select the *Branding.ClientSideRendering* project.
2. In the Properties window, set the “Site URL” property to the site collection you previously created and configured.



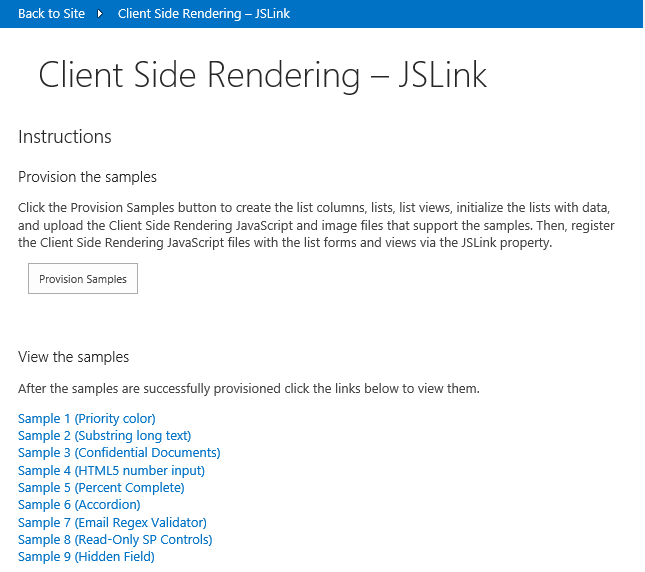
1. Press F5 or click the Start button in Visual Studio 2013.
2. Enter you user name and password to connect to your SharePoint site collection.



1. After your username and password have been verified, the trust dialog is displayed. Click the “Trust It” button.



1. After app installation, a new page will be displayed. Click the Provision Samples button to create the list columns, lists, list views, initialize the lists with data, and upload the Client Side Rendering JavaScript and image files that support the samples. The provisioning code registers the Client Side Rendering JavaScript files with the list forms and views via the JSLink property.



1. After you have successfully configured your SharePoint environment and deployed the artifacts via the app, you can view the Client Side Rendering samples by clicking on the links in the app.

# DEPLOYMENT DETAILS

The code behind in the default.aspx.cs file contains all the code used to deploy the artifacts which support this sample. This code sample uses the remote provisioning pattern to deploy the artifacts. The remote provisioning pattern uses the SharePoint Client Side Object Model to deploy the artifacts. There are many other Office AMS samples which demonstrate this same approach.

## Uploading the Client Side Rendering JavaScript files

When the Provision Sample sbutton is clicked the UploadJSFiles method is called. The UploadJSFiles method creates a folder named JSLink-Samples in the Style Library to store the Client Side Rendering JavaScript files, then it uploads the Client Side Rendering JavaScript files. Then, it creates a sub folder named imgs and uploads an image to the folder. The image is used by on of the samples.

void UploadJSFiles(Web web)

{

//Delete the folder if it exists

Microsoft.SharePoint.Client.List list =

web.Lists.GetByTitle("Style Library");

IEnumerable<Folder> results =

web.Context.LoadQuery<Folder>(list.RootFolder.Folders.Where(folder =>

folder.Name == "JSLink-Samples"));

web.Context.ExecuteQuery();

Folder samplesJSfolder = results.FirstOrDefault();

if (samplesJSfolder != null)

{

samplesJSfolder.DeleteObject();

web.Context.ExecuteQuery();

}

samplesJSfolder = list.RootFolder.Folders.Add("JSLink-Samples");

web.Context.Load(samplesJSfolder);

web.Context.ExecuteQuery();

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/Accordion.js"), samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/ConfidentialDocuments.js"), samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/DisableInput.js"),

samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/HiddenField.js"),

samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/HTML5NumberInput.js"),

samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/PercentComplete.js"),

samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/PriorityColor.js"),

samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/ReadOnlySPControls.js"),

samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/RegexValidator.js"),

samplesJSfolder);

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/SubstringLongText.js"),

samplesJSfolder);

Folder imgsFolder = samplesJSfolder.Folders.Add("imgs");

web.Context.Load(imgsFolder);

web.Context.ExecuteQuery();

UploadFileToFolder(web,

Server.MapPath("../Scripts/JSLink-Samples/imgs/Confidential.png"),

imgsFolder);

}

## creating lists, views, items, and registering client side rendering javascript files with list views and forms

After the Client Side Rendering JavaScript files and image are uploaded, the app provisions the list columns, lists, list views, initializes the lists with data, and registers the Client Side Rendering JavaScript files with the list views and forms via the JSLink property.

Not every sample demonstrates how to register the Client Side Rendering JavaScript files with list views and forms. However, Sample 5 demonstrates everything. The ProvisionSamle5 method deploys Sample 5. It illustrates all of the patterns.

void ProvisionSample5(Web web)

{

//Delete list if it already exists

ListCollection lists = web.Lists;

IEnumerable<List> results = web.Context.LoadQuery<List>(lists.Where(list

=> list.Title == "CSR-Tasks-Percent-Complete"));

web.Context.ExecuteQuery();

List existingList = results.FirstOrDefault();

if (existingList != null)

{

existingList.DeleteObject();

web.Context.ExecuteQuery();

}

//Create list

ListCreationInformation creationInfo = new ListCreationInformation();

creationInfo.Title = "CSR-Tasks-Percent-Complete";

creationInfo.TemplateType = (int)ListTemplateType.Tasks;

List newlist = web.Lists.Add(creationInfo);

newlist.Update();

web.Context.Load(newlist);

web.Context.ExecuteQuery();

//Add items

Microsoft.SharePoint.Client.ListItem item1 = newlist.AddItem(new

ListItemCreationInformation());

item1["Title"] = "Task 1";

item1["StartDate"] = "2014-1-1";

item1["DueDate"] = "2014-2-1";

item1["PercentComplete"] = "0.59";

item1.Update();

Microsoft.SharePoint.Client.ListItem item2 = newlist.AddItem(new

ListItemCreationInformation());

item2["Title"] = "Task 2";

item2["StartDate"] = "2014-1-1";

item2["DueDate"] = "2014-2-1";

item2["PercentComplete"] = "0.40";

item2.Update();

Microsoft.SharePoint.Client.ListItem item3 = newlist.AddItem(new

ListItemCreationInformation());

item3["Title"] = "Task 3";

item3["StartDate"] = "2014-1-1";

item3["DueDate"] = "2014-2-1";

item3["PercentComplete"] = "1.0";

item3.Update();

Microsoft.SharePoint.Client.ListItem item4 = newlist.AddItem(new

ListItemCreationInformation());

item4["Title"] = "Task 4";

item4["StartDate"] = "2014-1-1";

item4["DueDate"] = "2014-2-1";

item4["PercentComplete"] = "0.26";

item4.Update();

Microsoft.SharePoint.Client.ListItem item5 = newlist.AddItem(new

ListItemCreationInformation());

item5["Title"] = "Task 5";

item5["StartDate"] = "2014-1-1";

item5["DueDate"] = "2014-2-1";

item5["PercentComplete"] = "0.50";

item5.Update();

//Create sample view

ViewCreationInformation sampleViewCreateInfo = new

ViewCreationInformation();

sampleViewCreateInfo.Title = "CSR Sample View";

sampleViewCreateInfo.ViewFields = new string[] { "DocIcon", "LinkTitle",

"DueDate", "AssignedTo", "PercentComplete" };

sampleViewCreateInfo.SetAsDefaultView = true;

Microsoft.SharePoint.Client.View sampleView =

newlist.Views.Add(sampleViewCreateInfo);

sampleView.Update();

web.Context.Load(newlist, l => l.DefaultViewUrl,

l => l.DefaultDisplayFormUrl,

l => l.DefaultEditFormUrl,

l => l.DefaultNewFormUrl);

web.Context.ExecuteQuery();

//Register JS files via JSLink properties

RegisterJStoWebPart(web, newlist.DefaultViewUrl,

"~sitecollection/Style Library/JSLink-Samples/PercentComplete.js");

RegisterJStoWebPart(web, newlist.DefaultDisplayFormUrl, "~sitecollection/Style Library/JSLink-Samples/PercentComplete.js");

RegisterJStoWebPart(web, newlist.DefaultEditFormUrl,

"~sitecollection/Style Library/JSLink-Samples/PercentComplete.js");

RegisterJStoWebPart(web, newlist.DefaultNewFormUrl,

"~sitecollection/Style Library/JSLink-Samples/PercentComplete.js");

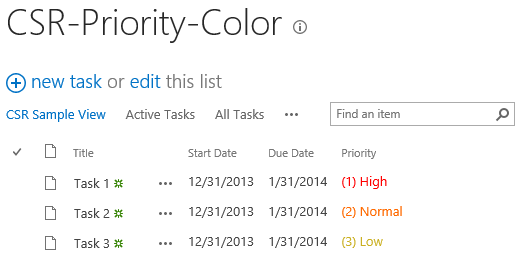
}

# Client side rendering javascript details

The following section describes the Client Side Rendering samples and the JavaScript code used to implement them.

## Sample 1 – Task priority color

This sample demonstrates how to apply formatting to a list column based on the column value. In this sample, list items which have a Priority column value of (1) High are indicated in red, items which have a Priority column value of (2) Normal are indicated in orange, and items which have a Priority column value of (3) Low are indicated in yellow.



The following code illustrates how the Priority column is formatted based on the column’s value.

(function () {

// Create object that has the context information about the field that we want to render differently

var priorityFiledContext = {};

priorityFiledContext.Templates = {};

priorityFiledContext.Templates.Fields = {

// Apply the new rendering for Priority field in List View

"Priority": { "View": priorityFiledTemplate }

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(priorityFiledContext);

})();

// This function provides the rendering logic for list view

function priorityFiledTemplate(ctx) {

var priority = ctx.CurrentItem[ctx.CurrentFieldSchema.Name];

// Return html element with appropriate color based on the Priority column’s value

switch (priority) {

case "(1) High":

return "<span style='color :#f00'>" + priority + "</span>";

break;

case "(2) Normal":

return "<span style='color :#ff6a00'>" + priority + "</span>";

break;

case "(3) Low":

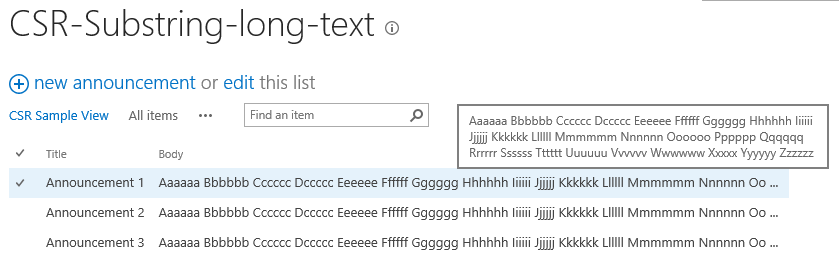
return "<span style='color :#cab023'>" + priority + "</span>";

}

}

## Sample 2 – Substring long text

This sample demonstrates how to make the Announcements list’s Body column text display shorter in the All Items view than it does out of the box. It also demonstrates how to display the entire Body column’s text as a tooltip for each list item. In the screenshot below you can see the Body column has been truncated so it only displays a subset of the text in the Body column. You can also see the tooltip which displays the entire Body column text for the first announcement in the list. This tooltip appears when you mouse over items in the list.



The following code illustrates how the Body column is truncated and how the tooltip is added via the title attribute.

(function () {

// Create object that has the context information about the field that we want to render differently

var bodyFiledContext = {};

bodyFiledContext.Templates = {};

bodyFiledContext.Templates.Fields = {

// Apply the new rendering for Body field in list view

"Body": { "View": bodyFiledTemplate }

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(bodyFiledContext);

})();

// This function provides the rendering logic

function bodyFiledTemplate(ctx) {

var bodyValue = ctx.CurrentItem[ctx.CurrentFieldSchema.Name];

//This regex expression is used to delete html tags from the Body field

var regex = /(<([^>]+)>)/ig;

bodyValue = bodyValue.replace(regex, "");

var newBodyValue = bodyValue;

if (bodyValue && bodyValue.length >= 100)

{

newBodyValue = bodyValue.substring(0, 100) + " ...";

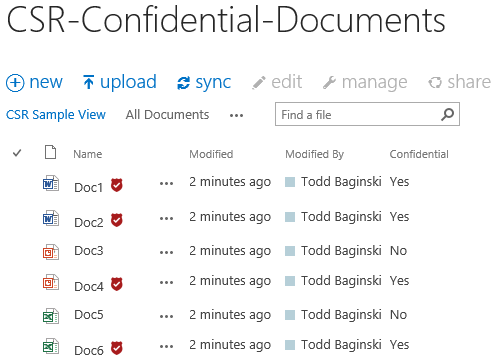
}

return "<span title='" + bodyValue + "'>" + newBodyValue + "</span>";

}

## Sample 3 – Confidential Documents

This sample demonstrates how to display an image next to a document’s Name in a document library based on a field value associated with the document. In the screenshot below you can see the red badge image displayed next to each document whose Confidential column value is Yes.



The following code illustrates how the Confidential column is evaluated to see if the document is marked as confidential. It also demonstrates how the Title column is modified to display the red badge image accordingly.

(function () {

// Create object that has the context information about the field that we want to render differently

var linkFilenameFiledContext = {};

linkFilenameFiledContext.Templates = {};

linkFilenameFiledContext.Templates.Fields = {

// Apply the new rendering for LinkFilename field in list view

"LinkFilename": { "View": linkFilenameFiledTemplate }

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(linkFilenameFiledContext);

})();

// This function provides the rendering logic

function linkFilenameFiledTemplate(ctx) {

var confidential = ctx.CurrentItem["Confidential"];

var title = ctx.CurrentItem["FileLeafRef"];

// This Regex expression use to delete extension (.docx, .pdf ...) form the file name

title = title.replace(/\.[^/.]+$/, "")

// Check confidential field value

if (confidential && confidential.toLowerCase() == 'yes') {

// Render HTML that contains the file name and the confidential icon

return title + "&nbsp;<img src= '" + \_spPageContextInfo.siteServerRelativeUrl + "/Style%20Library/JSLink-Samples/imgs/Confidential.png' alt='Confidential Document' title='Confidential Document'/>";

}

else

{

return title;

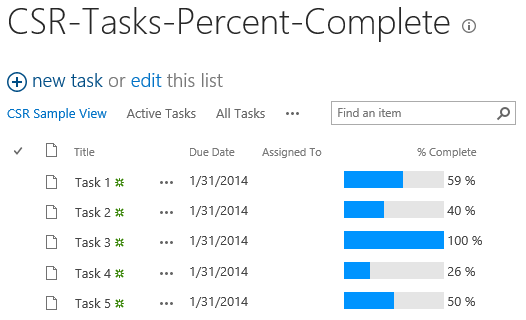
}

}

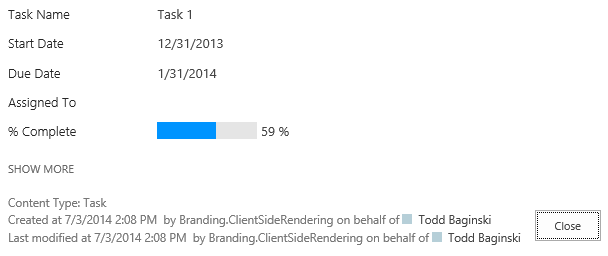
## Sample 4 – Percent complete

This sample demonstrates how to display a bar chart in the % Complete column in a Task list. In the screenshots below you can see the blue bar charts which indicate the percent each task is complete based on the value in the % Complete column.

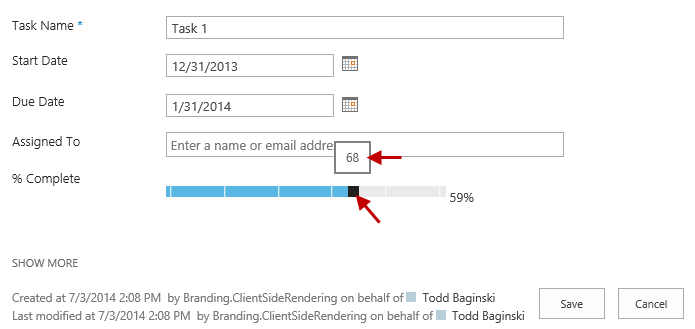
This is how the column appears in the list view. (View)



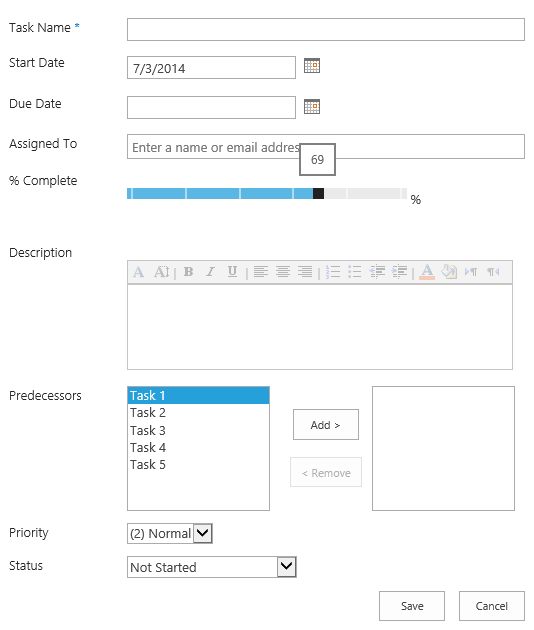
This is how the column appears when viewing a list item. (DisplayForm)



This is how the column appears when editing a list item. (EditForm) The red arrows indicate how you can slide the black box to select the value and the tool tip which appears over the currently selected value.



This is how the column appears when creating a new list item. (NewForm) The red arrows indicate how you can slide the black box to select the value and the tool tip which appears over the currently selected value.



The following code illustrates how the bar charts are created in the % Complete column and registered with the View and DisplayForms. It also demonstrates how the input controls for the % Complete column and created and registered with the New and Edit forms.

(function () {

// Create object that has the context information about the field that we want to render differently

var percentCompleteFiledContext = {};

percentCompleteFiledContext.Templates = {};

percentCompleteFiledContext.Templates.Fields = {

// Apply the new rendering for PercentComplete field in List View, Display, New and Edit forms

"PercentComplete": {

"View": percentCompleteViewFiledTemplate,

"DisplayForm": percentCompleteViewFiledTemplate,

"NewForm": percentCompleteEditFiledTemplate,

"EditForm": percentCompleteEditFiledTemplate

}

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(percentCompleteFiledContext);

})();

// This function provides the rendering logic for View and Display forms

function percentCompleteViewFiledTemplate(ctx) {

var percentComplete = ctx.CurrentItem[ctx.CurrentFieldSchema.Name];

return "<div style='background-color: #e5e5e5; width: 100px; display:inline-block;'> \

<div style='width: " + percentComplete.replace(/\s+/g, '') + "; background-color: #0094ff;'> \

&nbsp;</div></div>&nbsp;" + percentComplete;

}

// This function provides the rendering logic for New and Edit forms

function percentCompleteEditFiledTemplate(ctx) {

var formCtx = SPClientTemplates.Utility.GetFormContextForCurrentField(ctx);

// Register a callback just before submit.

formCtx.registerGetValueCallback(formCtx.fieldName, function () {

return document.getElementById('inpPercentComplete').value;

});

return "<input type='range' id='inpPercentComplete' name='inpPercentComplete' min='0' max='100' \

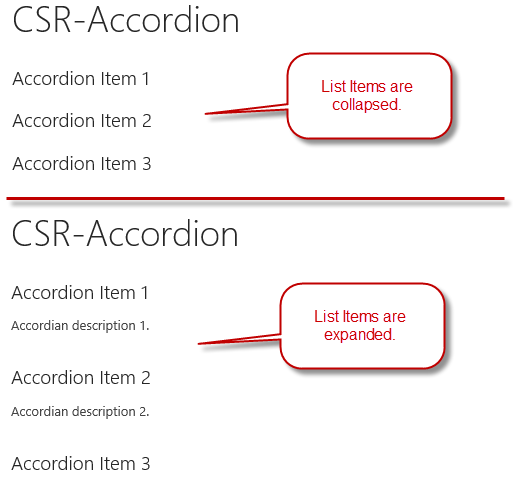
oninput='outPercentComplete.value=inpPercentComplete.value' value='" + formCtx.fieldValue + "' /> \

<output name='outPercentComplete' for='inpPercentComplete' >" + formCtx.fieldValue + "</output>%";

}

## Sample 5 – accordion

This sample demonstrates how to change the rendering template for an entire list view. In the screenshot below you can see list items when they are collapsed and how they appear once expanded. Clicking on a list item expands and collapses it.



The following code illustrates how the accordion functionality is implemented and registered with the list Template. The header, footer, and item properties are set to define the overall layout of the items in the list. The onPostRender property registers the JavaScript function to execute when the list is rendered. This function hooks up the click events and the CSS code necessary to implement the expand and collapse functionality.

(function () {

// jQuery library is required in this sample

// Fallback to loading jQuery from a CDN path if the local is unavailable

(window.jQuery || document.write('<script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-1.10.0.min.js"><\/script>'));

// Create object that has the context information about the field that we want to render differently

var accordionContext = {};

accordionContext.Templates = {};

// Be careful when adding the header for the template, because it will break the default list view render

accordionContext.Templates.Header = "<div class='accordion'>";

accordionContext.Templates.Footer = "</div>";

// Add OnPostRender event handler to add accordion click events and style

accordionContext.OnPostRender = accordionOnPostRender;

// This line of code tells the TemplateManager that we want to change all the HTML for item row rendering

accordionContext.Templates.Item = accordionTemplate;

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(accordionContext);

})();

// This function provides the rendering logic

function accordionTemplate(ctx) {

var title = ctx.CurrentItem["Title"];

var description = ctx.CurrentItem["Description"];

// Return whole item html

return "<h2>" + title + "</h2><p>" + description + "</p><br/>";

}

function accordionOnPostRender() {

// Register event to collapse and expand when clicking on accordion header

$('.accordion h2').click(function () {

$(this).next().slideToggle();

}).next().hide();

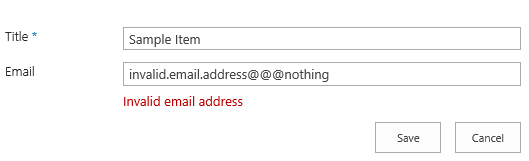
$('.accordion h2').css('cursor', 'pointer');

}

## Sample 6 – email regex validator

This sample demonstrates how to use regular expressions to validate column input values. In the screenshot below notice a red error message appears when an invalid email address is entered into the Email column input textbox.

This is how the column appears when editing a list item (EditForm) or creating a new list item (NewForm). Notice a red error message appears when an invalid email address is entered into the Email column input textbox.



The following code illustrates how the email validation for the Email column is implemented and registered with the New and Edit forms. The registerGetCallback function registers a call back function the form will fire before submittal, in this scenario the callback function returns the value in the input control for the Email column. The code also registers a validator with the form (emailValidator) as well as the callback method to handle validation errors. The validator function uses a regular expression to validate the email format. If the email format is invalid it returns the error message. The HTML in the field template contains a placeholder <span id='spnError' class='ms-formvalidation ms-csrformvalidation'></span> to display the error message.

(function () {

// Create object that has the context information about the field that we want to render differently

var emailFiledContext = {};

emailFiledContext.Templates = {};

emailFiledContext.Templates.Fields = {

// Apply the new rendering for Email field on New and Edit Forms

"Email": {

"NewForm": emailFiledTemplate,

"EditForm": emailFiledTemplate

}

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(emailFiledContext);

})();

// This function provides the rendering logic

function emailFiledTemplate(ctx) {

var formCtx = SPClientTemplates.Utility.GetFormContextForCurrentField(ctx);

// Register a callback just before submit.

formCtx.registerGetValueCallback(formCtx.fieldName, function () {

return document.getElementById('inpEmail').value;

});

//Create container for various validations

var validators = new SPClientForms.ClientValidation.ValidatorSet();

validators.RegisterValidator(new emailValidator());

// Validation failure handler.

formCtx.registerValidationErrorCallback(formCtx.fieldName, emailOnError);

formCtx.registerClientValidator(formCtx.fieldName, validators);

return "<span dir='none'><input type='text' value='" + formCtx.fieldValue + "' maxlength='255' id='inpEmail' class='ms-long'> \

<br><span id='spnError' class='ms-formvalidation ms-csrformvalidation'></span></span>";

}

// Custom validation object to validate email format

emailValidator = function () {

emailValidator.prototype.Validate = function (value) {

var isError = false;

var errorMessage = "";

//Email format Regex expression

var emailRejex = /\S+@\S+\.\S+/;

if (!emailRejex.test(value) && value.trim()) {

isError = true;

errorMessage = "Invalid email address";

}

//Send error message to error callback function (emailOnError)

return new SPClientForms.ClientValidation.ValidationResult(isError, errorMessage);

};

};

// Add error message to spnError element under the input field element

function emailOnError(error) {

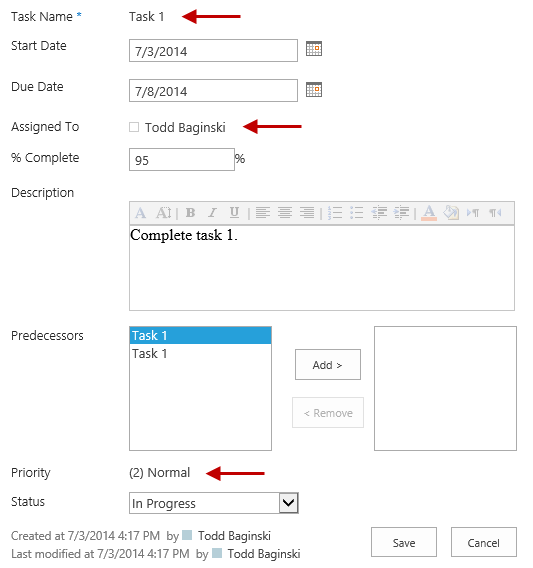
document.getElementById("spnError").innerHTML = "<span role='alert'>" + error.errorMessage + "</span>";

}

## Sample 7 – read only sp controls

This sample demonstrates how to make some fields read only in SharePoint list item Edit forms.

This is how the form appears when editing a list item (EditForm). The red arrows indicate the read only columns in the form.



The following code illustrates how the Title, AssignedTo, and Priority columns have their field templates modified to display just the field value instead of the out of the box input controls. Notice how different types of parsing must occur to extract a field’s value and display it depending on what type of field it is.

function () {

// Create object that has the context information about the field that we want to render differently

var readonlyFiledContext = {};

readonlyFiledContext.Templates = {};

readonlyFiledContext.Templates.Fields = {

// Apply the new rendering for Title, AssignedTo, and Priority fields on Edit forms

"Title": {

"EditForm": readonlyFieldTemplate

},

"AssignedTo": {

"EditForm": readonlyFieldTemplate

},

"Priority": {

"EditForm": readonlyFieldTemplate

}

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(readonlyFiledContext);

})();

// This function provides the rendering logic

function readonlyFieldTemplate(ctx) {

//Reuse SharePoint JavaScript libraries

switch (ctx.CurrentFieldSchema.FieldType) {

case "Text":

case "Number":

case "Integer":

case "Currency":

case "Choice":

case "Computed":

return SPField\_FormDisplay\_Default(ctx);

case "MultiChoice":

prepareMultiChoiceFieldValue(ctx);

return SPField\_FormDisplay\_Default(ctx);

case "Boolean":

return SPField\_FormDisplay\_DefaultNoEncode(ctx);

case "Note":

prepareNoteFieldValue(ctx);

return SPFieldNote\_Display(ctx);

case "File":

return SPFieldFile\_Display(ctx);

case "Lookup":

case "LookupMulti":

return SPFieldLookup\_Display(ctx);

case "URL":

return RenderFieldValueDefault(ctx);

case "User":

prepareUserFieldValue(ctx);

return SPFieldUser\_Display(ctx);

case "UserMulti":

prepareUserFieldValue(ctx);

return SPFieldUserMulti\_Display(ctx);

case "DateTime":

return SPFieldDateTime\_Display(ctx);

case "Attachments":

return SPFieldAttachments\_Default(ctx);

case "TaxonomyFieldType":

//Re-use JavaScript from the sp.ui.taxonomy.js SharePoint JavaScript library

return SP.UI.Taxonomy.TaxonomyFieldTemplate.renderDisplayControl(ctx);

}

}

//User control need specific formatted value to render content correctly

function prepareUserFieldValue(ctx) {

var item = ctx['CurrentItem'];

var userField = item[ctx.CurrentFieldSchema.Name];

var fieldValue = "";

for (var i = 0; i < userField.length; i++) {

fieldValue += userField[i].EntityData.SPUserID + SPClientTemplates.Utility.UserLookupDelimitString + userField[i].DisplayText;

if ((i + 1) != userField.length) {

fieldValue += SPClientTemplates.Utility.UserLookupDelimitString

}

}

ctx["CurrentFieldValue"] = fieldValue;

}

//Choice control need specific formatted value to render content correctly

function prepareMultiChoiceFieldValue(ctx) {

if (ctx["CurrentFieldValue"]) {

var fieldValue = ctx["CurrentFieldValue"];

var find = ';#';

var regExpObj = new RegExp(find, 'g');

fieldValue = fieldValue.replace(regExpObj, '; ');

fieldValue = fieldValue.replace(/^; /g, '');

fieldValue = fieldValue.replace(/; $/g, '');

ctx["CurrentFieldValue"] = fieldValue;

}

}

//Note control need specific formatted value to render content correctly

function prepareNoteFieldValue(ctx) {

if (ctx["CurrentFieldValue"]) {

var fieldValue = ctx["CurrentFieldValue"];

fieldValue = "<div>" + fieldValue.replace(/\n/g, '<br />'); + "</div>";

ctx["CurrentFieldValue"] = fieldValue;

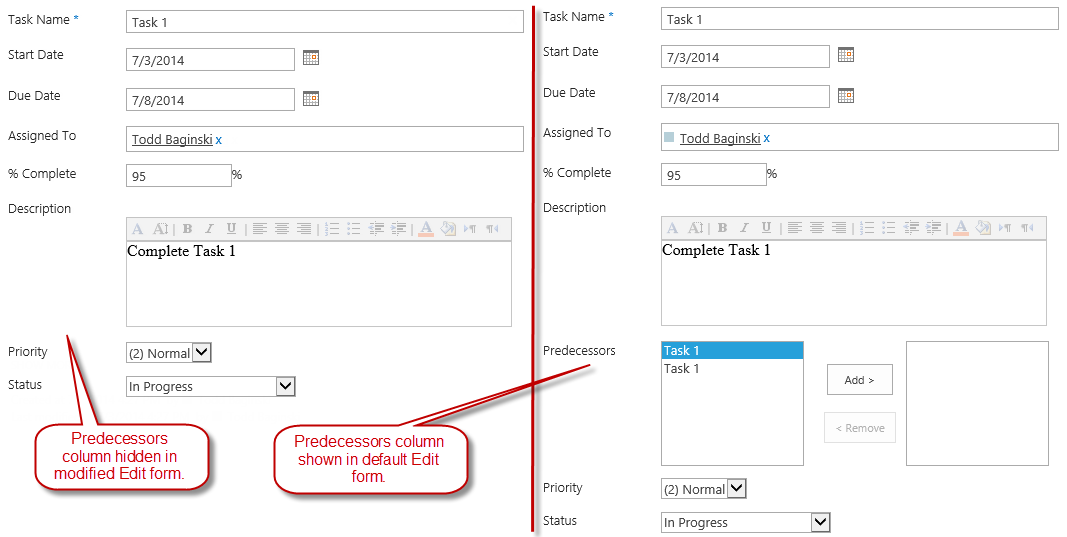
}

}

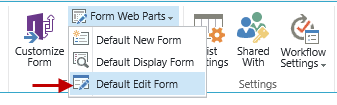
## Sample 8 – hidden field

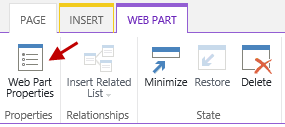
This sample demonstrates how to make some fields hidden in SharePoint list item New and Edit forms.

This is screenshots below indicate how the form appears when editing a list item (EditForm). The view on the right is the out of the box Edit form for the Tasks list, it includes the Predecessors column. The view on the left is the customized Edit form for the Tasks list, it hides the Predecessors column.

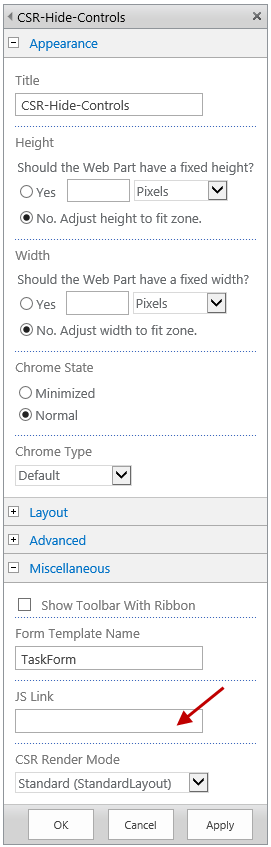


When the sample is deployed it shows the customized Edit form. To see the default Edit form follow these steps:

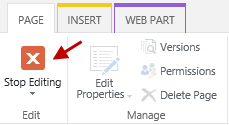
1. Navigate to the **CSR-Hide-Controls list**.
2. In the Ribbon, click the **LIST** **tab**.
3. In the Ribbon, click **Form Web Parts** and select **Default Edit Form**.
4. In the Ribbon, click **Web Part Properties**.



1. In the CSR-Hide-Controls Web Part Toolpane, expand the **Miscellaneous** section and delete the text in the **JS Link** textbox.



1. Click **OK**.
2. In the Ribbon, click **Stop Editing**.

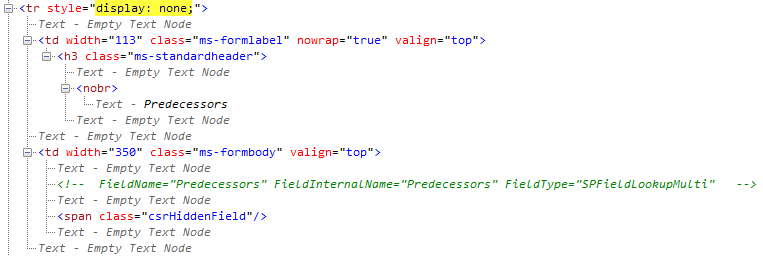


1. Edit an existing list item.

Note: These same steps may also be applied to the New form.

The following code illustrates how the Predecessors column is hidden in the New and Edit forms. The OnPostRender property registers a JavaScript function which locates and hides the Predecessor field.

Note: This client side method does not remove the column from the HTML. If you inspect the HTML you will see the Predecessors control is still part of the DOM, although it is not visible in the web browser.



(function () {

// jQuery library is required in this sample

// Fallback to loading jQuery from a CDN path if the local is unavailable

(window.jQuery || document.write('<script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-1.10.0.min.js"><\/script>'));

// Create object that has the context information about the field that we want to render differently

var hiddenFiledContext = {};

hiddenFiledContext.Templates = {};

hiddenFiledContext.Templates.OnPostRender = hiddenFiledOnPreRender;

hiddenFiledContext.Templates.Fields = {

// Apply the new rendering for Predecessors field in New and Edit forms

"Predecessors": {

"NewForm": hiddenFiledTemplate,

"EditForm": hiddenFiledTemplate

}

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(hiddenFiledContext);

})();

// This function provides the rendering logic

function hiddenFiledTemplate() {

return "<span class='csrHiddenField'></span>";

}

// This function provides the rendering logic

function hiddenFiledOnPreRender(ctx) {

jQuery(".csrHiddenField").closest("tr").hide();

}