



2024 Power Apps Coding Standards For Canvas Apps

MATTHEW DEVANEY

LAST UPDATED: 2024-06-03



Table Of Contents

Introduction	5
Naming Conventions	6
Screen Names	6
Control Names	6
Variable Names	7
Collection Names	8
Datasource Table Names	8
Variable Type Standards	9
Variable Scope.....	9
Why Use Both Local Variables And Global Variables	9
Usage Examples	10
Commenting Code	11
Why Write Code Comments?.....	11
Tips For Writing Good Comments.....	11
Line Comments vs. Block Comments	12
Commenting Style.....	12
App Settings	13
General Tab	13
Display Tab	14
Upcoming Features Tab	14
Support Tab.....	15
Reviewing Canvas Apps	16
App Checker	16
Power Apps Code Review Tool	17
Source Code Review.....	18
Functional Testing.....	19
User Acceptance Testing.....	19
App Theming Guidelines	20
Creating An App Theme	20

Theming Variables Sample Code.....	21
Branding Templates	23
Form Design Guidelines	24
Restrict Text Input Values	24
Validate Form Data	24
Implement Error Handling	25
Protect Against Loss of Unsaved Data	26
Use A Single Form To Both Edit & Display Records.....	27
Gallery Design Guidelines	28
Design Empty States.....	28
Reset The Gallery Scroll Position.....	28
Refresh To Show Current Data.....	29
Define The Sort Order	31
Do Not Show Live Updates For Search Results	31
Avoid Nested Galleries	32
Use Flexible Height Galleries.....	33
Error-Handling.....	34
Enable Formula-Level Error Management.....	34
Patch Function Error-Handling.....	35
Power Apps Forms Error-Handling.....	36
Power Automate Flow Error-Handling.....	37
If Error Function	37
Handling Unexpected Errors	38
Optimizing App Performance	39
Load Multiple Datasets Concurrently	39
Write Formulas That Use Delegation.....	40
Cache Data In Collections And Variables	41
Limit The Size Of Collections	41
“Batch Patch” Multiple Changes To A Datasource Table At Once	42
Reduce Code In The OnStart Property	43
Minimize Number Of Controls On A Single Screen.....	43
Enable DelayOutput For Text Input Controls.....	44
Do Not Reference Controls On Other Screens.....	44
Eliminate The N+1 Problem	45

Improving Code Readability	47
Apply Automatic Formatting.....	47
Use The WITH Function To Improve Readability	47
Choose Consistent Logical Operators	48
Join Text Strings & Variables	49
Remove IF Statements When The Result Is A True Or False Value.....	49
Substitute The Self Operator For The Current Control Name	50
Flatten Nested IFs	50
Alphabetical Order In Patch & UpdateContext Functions	51
Simplify Logical Comparisons When Evaluating A Boolean	52

Introduction

Welcome to the Power Apps Coding Standards For Canvas Apps.

In this guide you will find 50+ pages of coding rules, guidelines and best practices I use everyday to create Power Apps Canvas apps. I have spent the last 3 years building Power Apps every day. Now I want to share the knowledge I've gained in this set of easy-to-understand, actionable examples.

Power Apps already has an [official set of canvas coding standards](#) released back in 2018. So why did I make my own? A few reasons:

- I wanted an updated set of standards and guidelines for 2024 that includes all of the latest features
- These coding standards can be continuously improved as new Power Apps features hit “general availability” in 2024, 2025, 2026 and beyond
- Readers can leave a comments on [my website](#) describing their own best practices which I can incorporate into future versions

I hope you enjoy my Power Apps Coding Standards For Canvas Apps.



Support The Site

I don't have ads on my website because I believe in delivering the best learning experience possible. This website is paid for out of my own pocket. If you've found value in the free resources I've created, I'd love to have your support.

Click on the “Buy Me A Cat Treat” button below to support the site.

 Buy me a cat treat

Naming Conventions

Screen Names

A screen name should clearly describe its purpose in 2-3 words ending with word “Screen.” Use proper-case. A screen-reader will speak the screen name to visually-impaired users when the screen loads.

Good Examples	Bad Examples	Bad Reason
Appointments Screen	Appointments	Missing the word ‘Screen’
Order Form Screen	OrderFormScreen	Not friendly to a screen reader
Collect Signature Screen	scrCollectSignature	Not friendly to a screen reader

Control Names

A control name should show the control-type, the purpose and the screen. Use camel-case and underscores for spacing. For example, the control named `txt_OrderForm_FirstName` is a text input that captures first name on the app’s Order Form Screen.

Good Examples	Bad Examples	Bad Reason
<code>drp_NewEmployee_Department</code>	<code>drpDepartmentNewEmployee</code>	No spacing
<code>btn_OrderForm_Submit</code>	<code>btn_Submit_OrderForm</code>	Wrong order
<code>gal_Home_Appointments</code>	<code>gly_Home Appointments</code>	Non-standard control prefix

A list of standard control prefixes can be found below.

Control	Prefix	Control	Prefix	Control	Prefix
3D Object	3do	Date Picker	dte	Microsoft Stream	str
Add Picture	pic	Drop Down	drp	PDF Viewer	pdf
Address Input	add	Export	exp	Pen Input	pen
Audio	aud	Form	frm	Power BI Tile	pbi
Barcode Scanner	bar	Gallery	gal	Radio	rad
Button	btn	Group	grp	Rating	rtg
Camera Control	cam	HTML Text	htm	Rich Text Editor	rte
Canvas	cvs	Icon	ico	Shapes	shp
Card	dtc	Image	img	Slider	sld
Charts	chr	Import	imp	Table	tbl
Check Box	chk	Label	lbl	Text Input	txt
Collection	col	List Box	lst	Timer	tmr
Container	con	Map	map	Toggle	tgl
Combo Box	cmb	Measuring Camera	mcm	Video	vid
Component	cmp	Microphone	mic		

Variable Names

A variable name should show the scope of the variable and its purpose. Use camel-case with no spaces between each word. For example, the variable *gblUserEmail* is a global variable which holds the current user's email address.

Good Examples	Bad Examples	Bad Reason
<code>gblUserCurrent</code>	<code>UserCurrent</code>	No scope
<code>locPacksInBoxQuantity</code>	<code>Loc_Packs_In_Box_Quantity</code>	Improper capitalization and spacing
<code>LocsLoading</code>	<code>locBoolLoading</code>	Do not use data types in variable names
<code>varWorkdaysDuringVacation</code>	<code>varWorkdays</code>	Not descriptive enough

Collection Names

A collection name should contain the original datasource and describe its purpose. Use camel-case with no spaces between each word. For example, the collection colDvInvoices is a collection of invoices from Dataverse.

Good Examples	Bad Examples	Bad Reason
colSpEmployees	colEmployees	No datasource
colDvSalesLeads	coldv_salesleads	Improper capitalization and spacing
colNavigationMenu	NavigationMenu	Do not use data types in variable names

A standard list of datasource abbreviations can be found below:

Original Datasource	Abbreviation
Dataverse	Dv
SharePoint	Sp
SQL	Sql
Salesforce	Sf
None (created in-app)	(none)

Datasource Table Names

A datasource created by the developer should have 1-3 words to describe its purpose. Use the singular form of the word and proper-case. Be as concise and clear about the purpose of the datasource as possible.

Good Examples	Bad Examples	Bad Reason
Employee	Emp	Abbreviation instead of full word
Construction Projects	Projects	Too general, what type of projects?
Repair Orders	RepairOrders	No spacing, plural

Variable Type Standards

Variable Scope

A [variable's](#) scope determines where it can be referenced in the app. If the variable is required on multiple screens use a global variable. Otherwise, use a local or context variable instead. Choose the proper variable type by determining its scope.

Variable Type	Declaration Method	Variable Scope
Global	Set Function	Variable is available across all app screens
Local	UpdateContext Function	Variable is only available on a single app screen
One-time	With Function	Variable is only available within the with function

Why Use Both Local Variables And Global Variables

Imagine a large canvas app with many screens that only uses *global variables*. When updating a variable's value the developer must be aware of its impact across all screens. If the developer does not correctly determine how to use a variable there are unintended consequences (i.e. a software bug).

Local variables can only be used on one screen. Developers have an easier time assessing the impact to a single screen as opposed to many screens. Higher quality code can be written at a faster pace.

One-time variables are not persistently stored in memory. After the With function is executed the variable is cleared from memory and cannot be accessed outside of the function.

Usage Examples

```
// Global variable
Set(
    gblSalesTaxAmount,
    Value(txt_OrderForm_SubtotalAmount.Text) * 0.13
);

// Local variables
UpdateContext(
    {
        locLineItemsCount: 0,
        locShowConfirmationMenu: false,
        locOrderFormMode=Blank()
    }
);

// One-time variable
With(
    {varBusinessContact: LookUp('Sales Orders', ID=ThisItem.ID)},
    Concatenate(
        varBusinessContact.FirstName,
        " ",
        varBusinessContact.LastName
    )
);
```

Commenting Code

Why Write Code Comments?

Write comments to describe the [intended goal](#) of a section of Power Apps code. Knowing the intended goal helps identify mismatches between it and the actual code outcome. Code that has comments takes significantly less time for other developers to understand.

Whether or not to write comments is an [ongoing debate](#) in the software development community. Do write comments into Power Apps code. Do not use comments as an excuse to write code with poor readability.

Tips For Writing Good Comments

Use these suggestions to write high-quality comments:

- Do write comments that describe the intent of a code section. Intent means the goal.
- Do not write comments that simply restate what a line of code does. Writing [clean code](#) means other developers should be able to understand its function.
- Do keep comments updated when the goal of a code section changes.
- Do not write so many comments they are impossible to maintain. Comments create their own [technical debt](#).
- Do write comments in full sentences and use plain language
- Do not use abbreviations, acronyms or slang

Line Comments vs. Block Comments

Power Apps has two comment styles: line comments and block comments. Line comments are made on a single-line and block comments can be made across multiple lines.

Comment Style	Syntax	Example
Line	// [comment goes here]	// Validate the work order to ensure it will not be rejected upon submission.
Block	/* [comments go here] */	/* Work Order Details Screen: - Uses a single form to create, edit and view a record to minimize the number of controls in the app - Emails a signed PDF to the employee's manager after the form is submitted so it can be stored as backup */

Commenting Style

Use these commenting conventions to ensure a consistent style:

- Place comments on a separate line above the code section they are describing.
- Do not write in-line comments beside on the same line as a piece of code.
- Start comments with a capital letter
- End comment text with a period

```
// Validate the work order to ensure it will not be rejected upon submission.  
Set(  
    varValidateForm,  
    Validate(  
        'Work Orders',  
        Defaults('Work Orders'),  
        gblRecordWorkOrderCurrent  
    )  
);
```

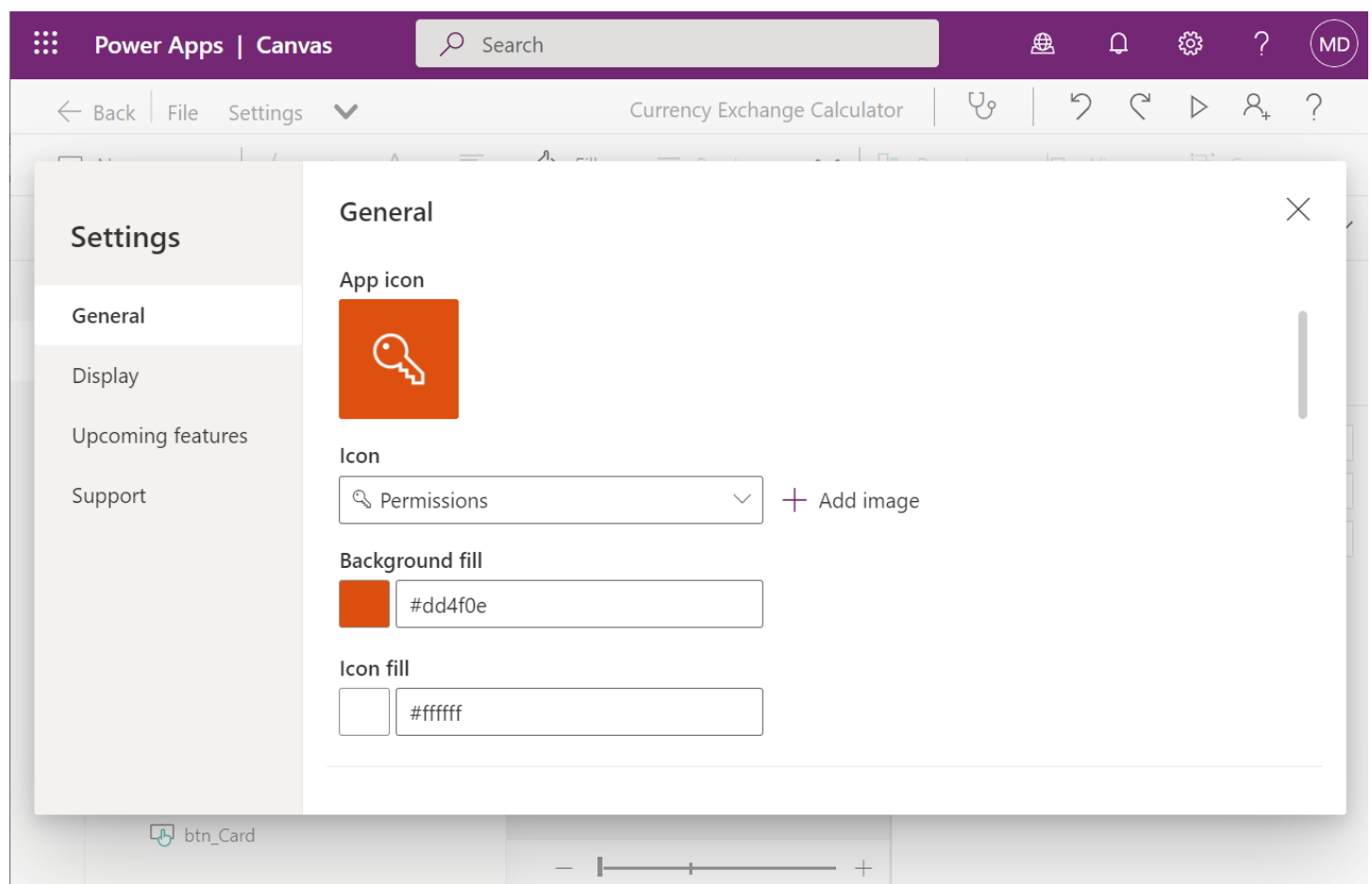
Format text Remove formatting Find and replace

App Settings

General Tab

Choose an icon before sharing an app with users. Make it consistent with company branding. Custom images must be 245px x 245px and .jpg or .png format.

Turn on the *Debug Published App* setting to enable better telemetry in Power Apps Monitor while the app is in development. Turn-off this setting when an app is pushed to production. It has a negative impact on will app performance but it is necessary for debugging during development.



Display Tab

PC & laptop users expect Power Apps to be responsive. Use [responsive design](#) unless the app is a proof-of-concept or there is not enough time in the budget.

Choose portrait orientation for mobile devices since they are held vertically in one-hand. Tablet apps can be either landscape or portrait [depending on its use case](#). Will the user be walking around performing inspections while using the app? Then use portrait mode. Will the user be seated at a table while using the app? Then use landscape mode. Can't decide [which orientation to use](#)? Design a responsive app that can change its orientation.

This table shows the recommended default display settings for each type. Start with these defaults and change them if you have a good reason.

Device	Orientation	Scale To Fit	Lock Aspect Ratio	Lock-Orientation
PC/Laptop	Landscape	No	No	No
Tablet	Landscape/Portrait	Yes	Yes	Yes
Mobile	Portrait	Yes	Yes	Yes
Tablet & Mobile	Portrait	No	No	No
All	Landscape	No	No	No

Upcoming Features Tab

Preview-features will be turned on for all Power Apps soon. It is recommended to turn all preview options on unless a feature is known to have a bug.

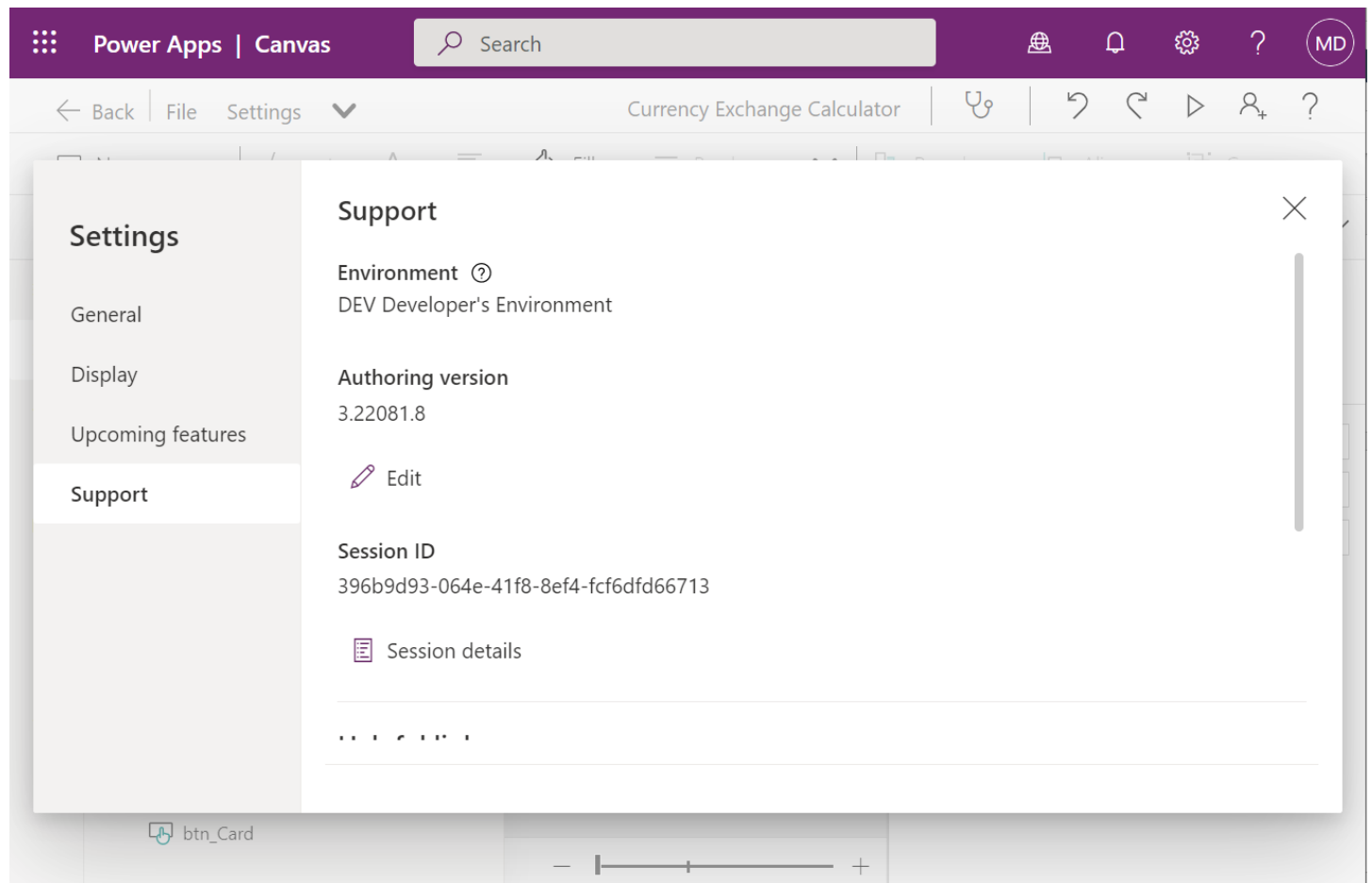
Experimental features might break, change or disappear at any-time. These features frequently have bugs or are incomplete. Do not use experimental features in production apps unless they have been thoroughly tested. Never use *Retired features*.

Do enable the known-good experimental feature *Enhanced Component Properties*.

Do not enable the preview features *Keep Recently Visited Screens In Memory* or *Expanded Media Support* or *SaveData on Power Apps Mobile Apps*

Support Tab

Power Apps authoring version determines which features and functionality are available in Power Apps Studio. While working on an app it is recommended to not change the authoring version because it can potentially introduce bugs for existing feature. If the authoring version is updated during development the app must be retested to ensure it operates as expected.

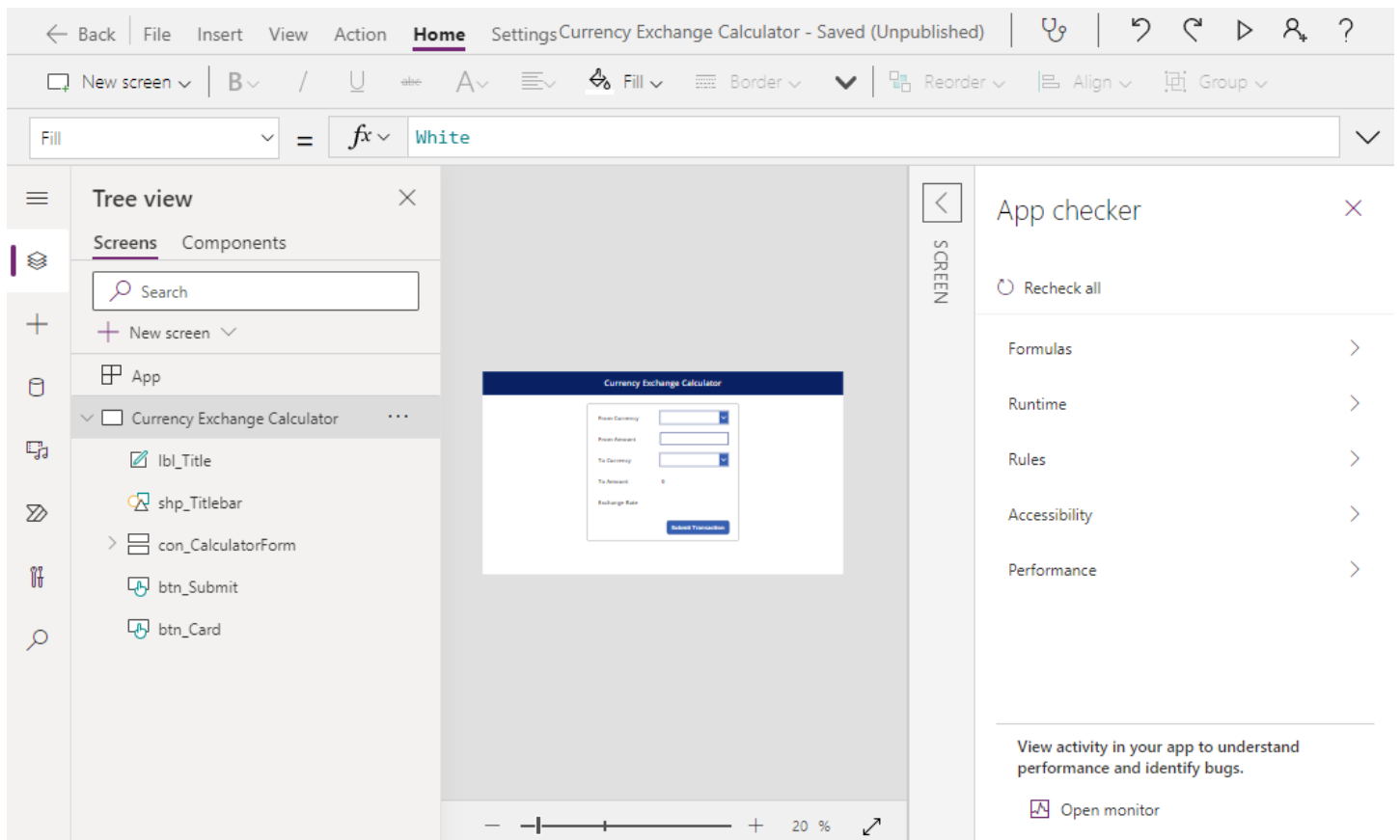


Reviewing Canvas Apps

App Checker

App checker identifies potential issues within a canvas app. A red dot will appear when there are formula errors or runtime errors to notify the developer a fix is needed. The red dot will not appear for rules, accessibility and performance errors.

Fix all issues identified in the app checker before publishing an app to production. This includes accessibility errors and performance errors. Sometimes it is not possible to clear all errors due to an error with the app checker itself. Have a strong justification for any errors that were not fixed.



Power Apps Code Review Tool

The [Power Apps Code Review Tool](#) is an automated code review tool built by Microsoft. A canvas app loaded into the review tool is analyzed against a checklist and is given a pass or fail score for each item. Failed items will tell the developer what code must be fixed.

Aim for a score of 90% with the code review tool. It is not necessary to achieve 100% because there are occasionally good reasons to avoid best practices. For example, the review tool may recommend using the concurrent function to execute parallel data requests. This technique can be problematic if the app is using too much memory on a mobile device and causes Power Apps to crash.

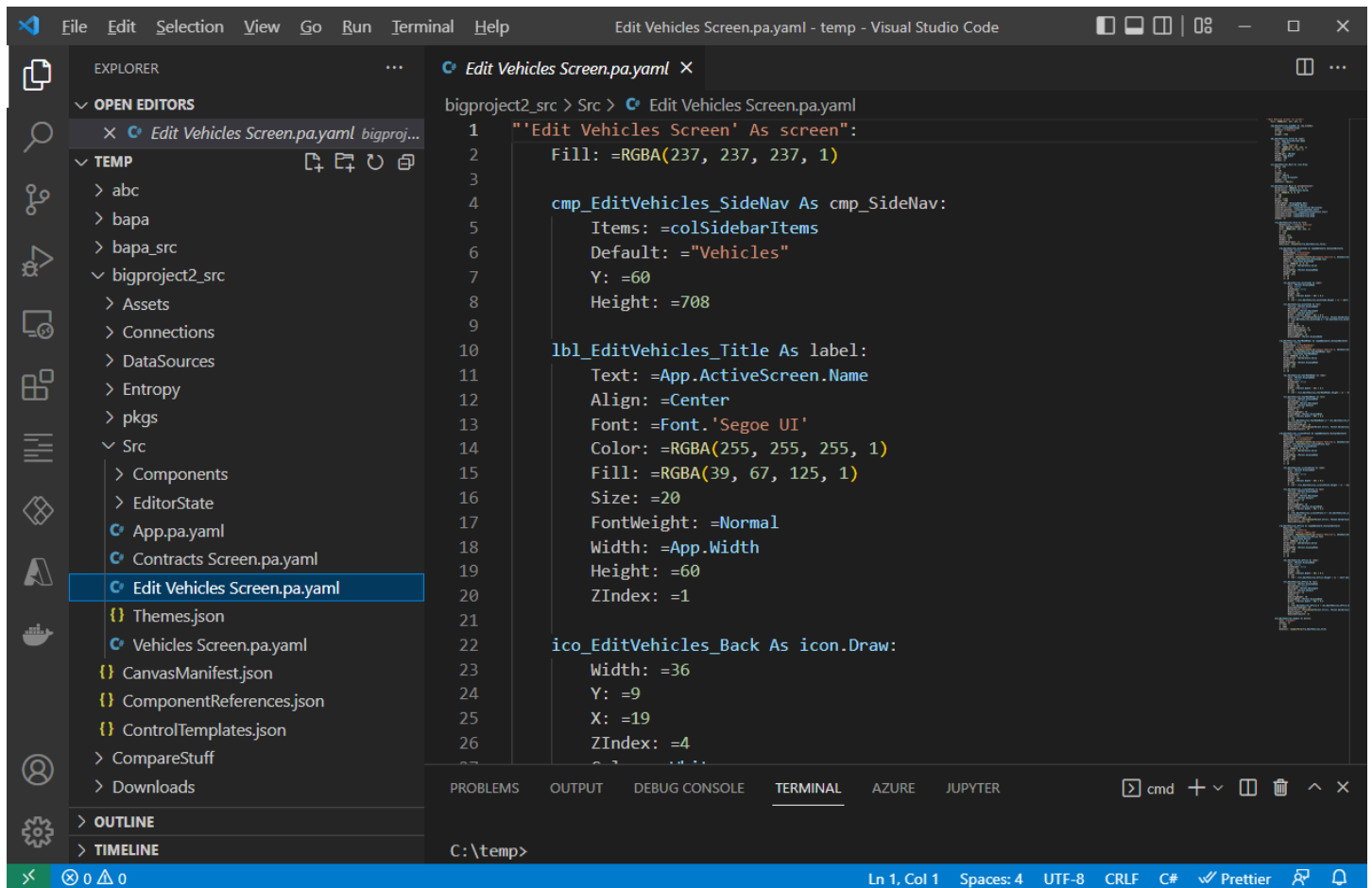
The screenshot shows the 'POWER APPS CODE REVIEW TOOL' interface. At the top, there is a blue header with a 'BACK' button and the tool's name. Below the header, the 'CODE REVIEW CHECKLIST' is displayed. On the right side, the 'SCORE' is shown as '90%' in a blue box. The checklist contains four items, each with a green status indicator and a 'PASS' dropdown menu. The items are:

- Performance: N+1 Database or API requests**
N+1 query often observed in galleries can trigger too many requests to servers. This happen when one or more controls within a gallery are bound to a LookUp/Filter operation on a data source. [Add comment](#) [VIEW DETAILS](#) PASS
- Performance: Nested Search, Filter or LookUp operations in formulas**
Consider changing nested filter such as Filter(Filter or Search(Filter .. to a single Filter. It would lead to a more compact/consice formula and perf improvment. [Add comment](#) [VIEW DETAILS](#) PASS
- Maintainability: Code Redability**
Avoid code/formulas that are long and hard to read. Examples include nested if/esle, nested operators (or, and) and other formulas. [Add comment](#) [VIEW DETAILS](#) PASS
- Performance: Use of Concurrent function**
Consider using concurrent function for parallel independent data request. [Code review tool : This app does not does not make use of Concurrent function. Please consider use it to ensure external database or API requests are performed in parallel.](#) [VIEW DETAILS](#) PASS

Source Code Review

The [Power Apps source code tool](#) is useful for manual code reviews. It unpacks an msapp file and allows the developer to review all code for a specific screen in a single file YAML file. This is useful because the developer does not have to click into each individual property of a canvas control to read the code. They can simply scan the YAML from top to bottom.

Use Visual Studio Code to read the unpacked canvas app code. The C# syntax highlighter is preferred over the YAML highlighter because Power Apps code is more similar to C#.



Functional Testing

[Functional testing](#) of an app should be performed by someone other than the original developer. Ask another developer on the team to Having a dedicated QA tester on the project team is recommended. Or, if no dedicated QA Tester is available ask another developer to engage in [peer-testing](#). Another person is more likely to uncover issues with an app's behaviour.

User Acceptance Testing

All apps must undergo [user acceptance testing](#) before being launched into production. Select a small, representative group of end users and ask them to test an app against a test script.

App Theming Guidelines

Creating An App Theme

[Define an app's theme](#) in order to achieve a consistent style throughout all screens. Create a set of theming variables in the app's [OnStart property](#) and manually apply them to each control type.

Variable Name	Purpose
gblAppColors	Color palette for the app
gblAppFonts	Heading fonts, body fonts and sizes used in the app
gblAppIcons	SVG icons used in the app
gblAppDefaults	Default values for common control properties

Keep a copy of each styled control on a hidden screen. It is more efficient to re-use controls instead of setting up a new control with a style every time.

Text

Text input

1

Item 5 Item 3

2/15/2022

1
2
3

Slider

★ ★ ☆ ☆ ☆ Button

Format | B / U | [Link] [Unlink] | [List] [List] | ...

Off Option 1 2

Theming Variables Sample Code

Use this code in the OnStart property of an app to define its theme.

```
// COLOR PALETTE
Set(
  gblAppColors,
  {
    // Primary Colors
    Primary1: ColorValue("#30475E"), // Navy Blue
    Primary2: ColorValue("#F05454"), // Light Red
    Primary3: ColorValue("#222831"), // Dark Blue
    Primary4: ColorValue("#DDDDDD"), // Light Gray

    // Accent Colors
    Black: ColorValue("#000000"),
    Cyan: ColorValue("#17A2B8"),
    Green: ColorValue("#28A745"),
    Orange: ColorValue("#FD7E14"),
    Red: ColorValue("#DC3545"),
    Teal: ColorValue("#20C997"),
    White: ColorValue("#FFFFFF"),
    Yellow: ColorValue("#FFC107"),

    // Neutral Colors
    GrayDark: ColorValue("#484644"),
    GrayMediumDark: ColorValue("#8A8886"),
    GrayMedium: ColorValue("#B3b0AD"),
    GrayMediumLight: ColorValue("#D2D0CE"),
    GrayLight: ColorValue("#F3F2F1")
  }
);
```

```

// FONTS & SIZES
Set(
  gblAppFonts,
  {
    Heading: "Roboto, Open Sans",
    Body: "Lato",
    Size: {
      Tiny: 10,
      Regular: 13,
      Subtitle: 16,
      Title: 20,
      Huge: 28
    }
  }
)

```

```

// ICONS
Set(
  gblAppIcons,
  {
    // SVG icon code is stored in an 'Import from Excel' table named AppIcons
    Checklist: LookUp(AppIcons, Name="Checklist", DataURI),
    Checkmark: LookUp(AppIcons, Name="Checkmark", DataURI)
  }
)

```

Branding Template

Form Design Guidelines

Restrict Text Input Values

When a [text input control](#) should only contain a number, change the default Format property to number. It prevents users from typing any character than number. Also, update the MaxLength property to match the field's maximum character limit.

```
// Format property of a text input named Year Project Started
Format.Number

// MaxLength property of a text input for Project Name
DataSourceInfo(Projects, DataSourceInfo.MaxLength, "Project Name")
```

Validate Form Data

[Perform data validation](#) to ensure a form is properly filled-in before submission. Check the following items:

- Required fields are not blank
- Proper formatting for phone numbers, email addresses, postal codes, URLs, dates, etc.
- Number fields are within the allowed minimum and maximum range
- Confirmation fields are matching (passwords, etc.)

Phone Number

Phone number must be in format ###-###-####

Give the user feedback when the form does not pass validation. There are [2 feedback strategies](#) to choose from:

1. Validate On Submission – check if the form passed validation when the user presses the submit button
2. Real-Time Validation – check if a field passed validation as the user types. Once a field meets all data validation criteria immediately indicate it passed.

Good feedback tells the user which fields failed and how to fix them. Use one or more of these strategies to deliver feedback:

- List the fields that failed validation and why at the top of the form
- Highlight any fields that failed validation in red
- Display an error message beside any fields that failed

Do not [disable a form's submit button until validation passes](#). If you use this pattern visually indicate why the submit button is disabled on the screen at all times.

The screenshot shows a 'Vehicle Reservation App' form with the following fields and validation status:

- Full Name:** Jane Adams (Valid, green checkmark)
- Age:** 21 (Valid, green checkmark)
- Reservation Date:** 8/19/2020 (Valid, green checkmark)
- Phone Number:** 204-998-0987 (Valid, green checkmark)
- Email Address:** jadams.companyname.com (Invalid, red X, error message: 'Not a valid email address.')

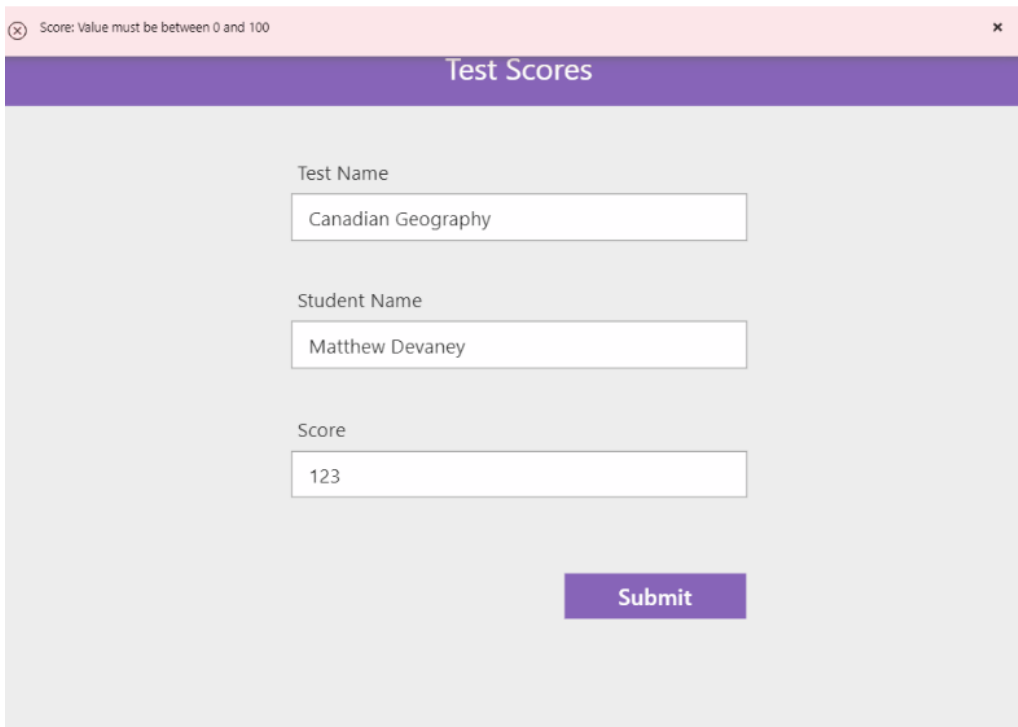
A 'Submit' button is located at the bottom right of the form.

Implement Error Handling

Never assume a form submitted successfully. Always check to make sure.

Error-handling for Power Apps [form control](#) and [patch forms](#) are performed differently. If using a Power Apps form control, catch errors using the *OnSuccess* and *OnFailure* properties. For a patch form, wrap the [Patch function](#) in an [IfError function](#) to detect an error.

When an error occurs, notify the user that form could not be submitted and why it was not successful. Do not move on to another screen until corrective action is taken.



The image shows a web form titled "Test Scores" with a purple header. The form contains three input fields: "Test Name" with the value "Canadian Geography", "Student Name" with the value "Matthew Devaney", and "Score" with the value "123". A purple "Submit" button is located at the bottom right. A pink error message banner at the top left reads "Score: Value must be between 0 and 100".

Protect Against Loss of Unsaved Data

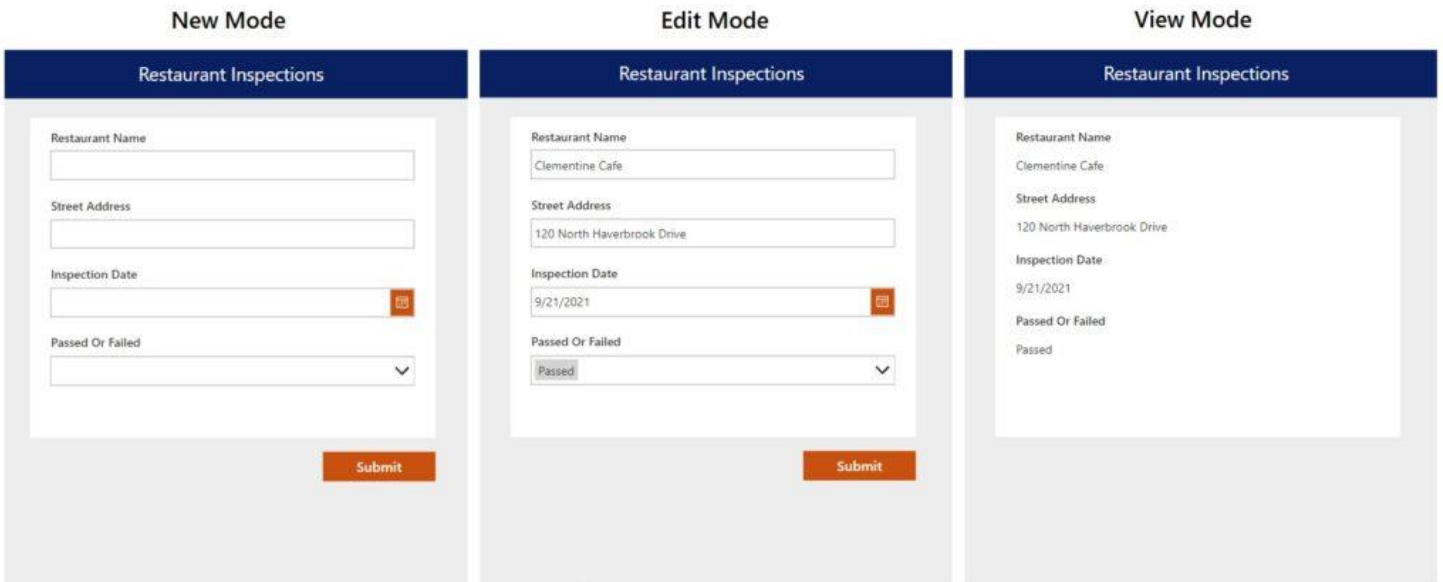
Prevent users from accidentally [exiting a form and losing unsaved data](#). Before a user leaves the screen prompt them for confirmation. Display this message on a pop-up menu: “you have unsaved data. Do you really want to close the form?” and present the choices OK or Cancel.



The image shows a confirmation dialog box with a grey border. The title is "Unsaved Data". The main text reads: "This form has unsaved data. Are you sure you want to leave the screen?". At the bottom, there are two buttons: a white "Yes" button and a dark blue "No" button.

Use A Single Form To Both Edit & Display Records

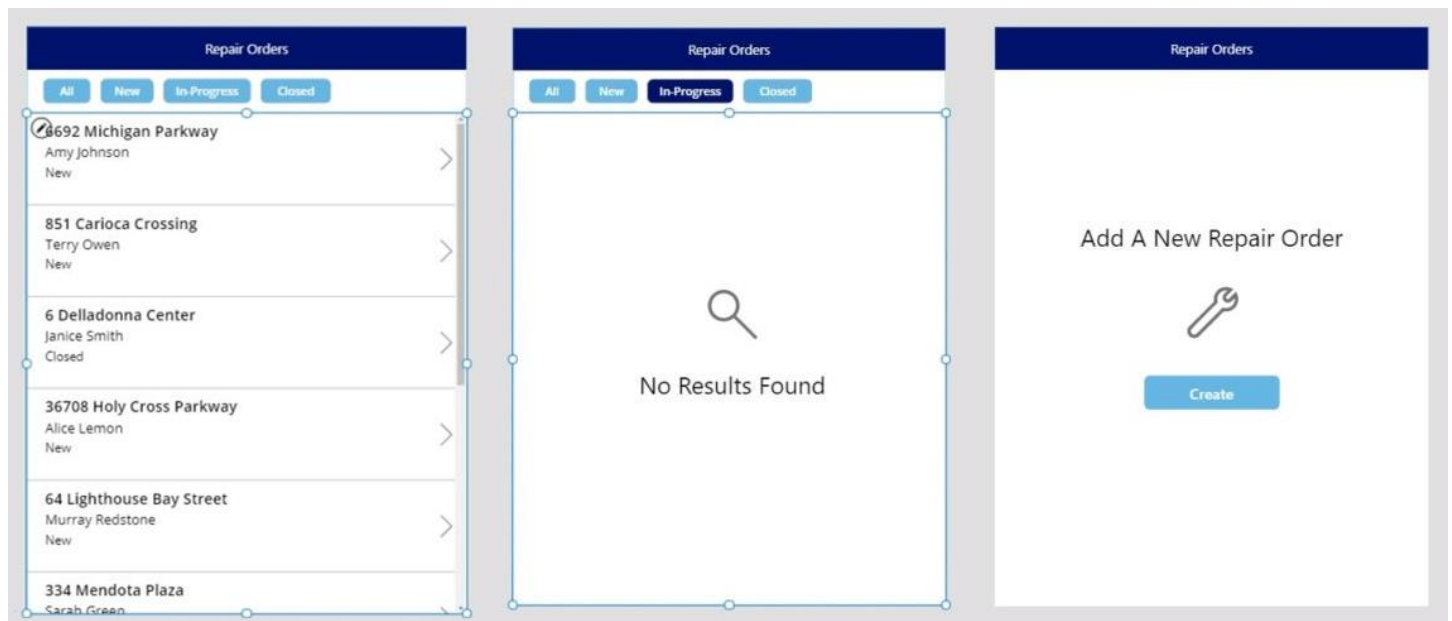
Re-use the same form to [create new records](#), [edit existing records](#) and [display records](#) in view only mode. Having only one form reduces development & maintenance time and ensures consistency. Change the mode of a Power Apps form using the NewForm, EditForm and View form functions. Patch forms require additional code to change the DisplayMode of each individual control manually.



Gallery Design Guidelines

Design Empty States

Include an [empty state](#) that appears when a gallery has no data. An empty state should tell the user why the gallery is empty and/or give directions on what actions to take next.



Reset The Gallery Scroll Position

A gallery should show its first item at the top when a screen is opened. If gallery was previously scrolled and was not reset it will remain in the same position when the screen is opened. To reset the gallery to the top position use this code in the *OnHidden* property of the gallery's screen.

```
// OnHidden property of a screen  
Reset(GalleryName);
```

Refresh To Show Current Data

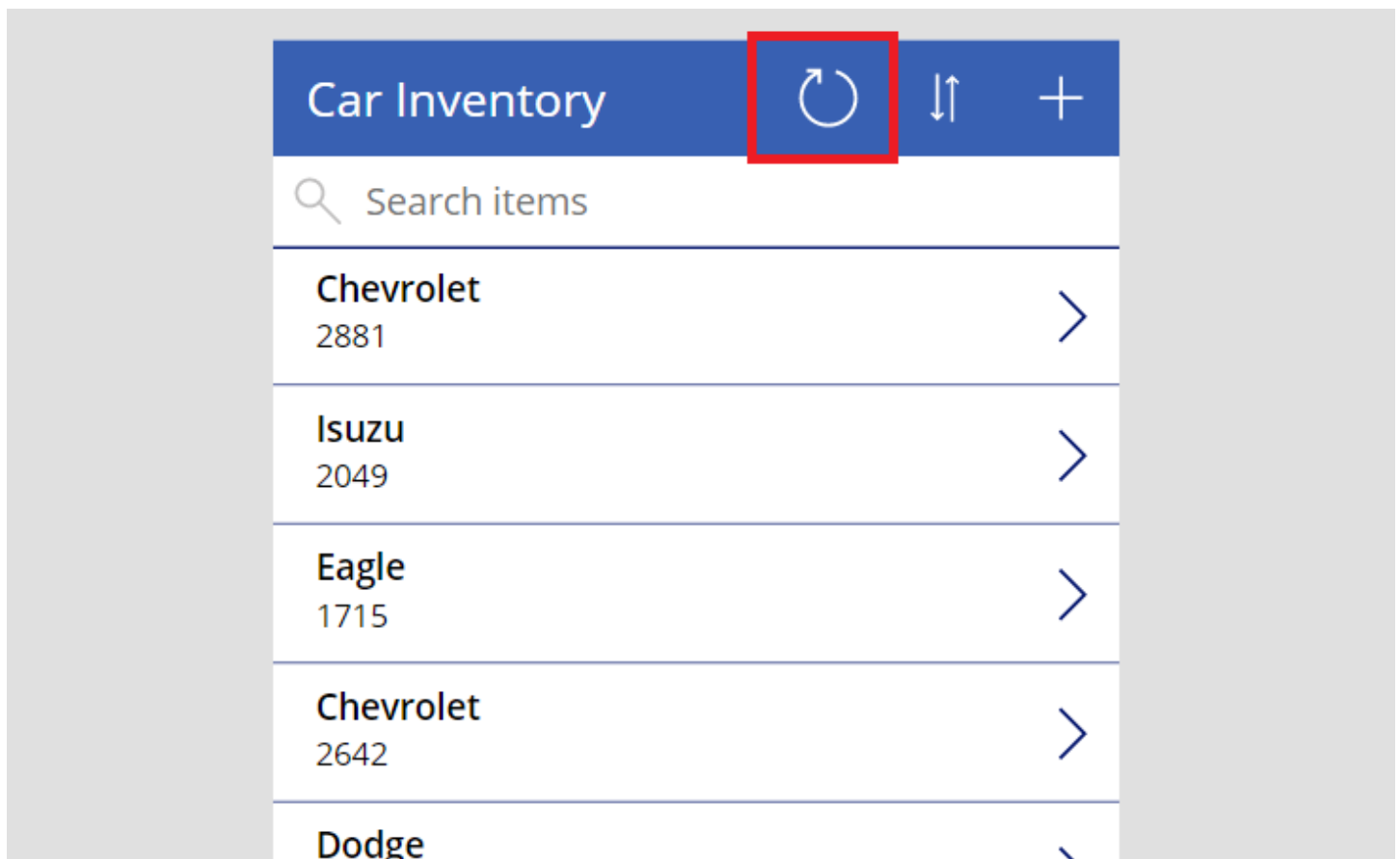
Users expect a gallery to include all recent updates to a datasource. Many times this [happens automatically with no extra effort](#) required by the developer. But sometimes a [manual refresh](#) of the datasource is needed.

Gallery data will refresh automatically:

- When using a local datasource (i.e. collections)
- During the initial load of a cloud datasource
- After using the app to perform CRUD operations on cloud datasource that the gallery is connected to

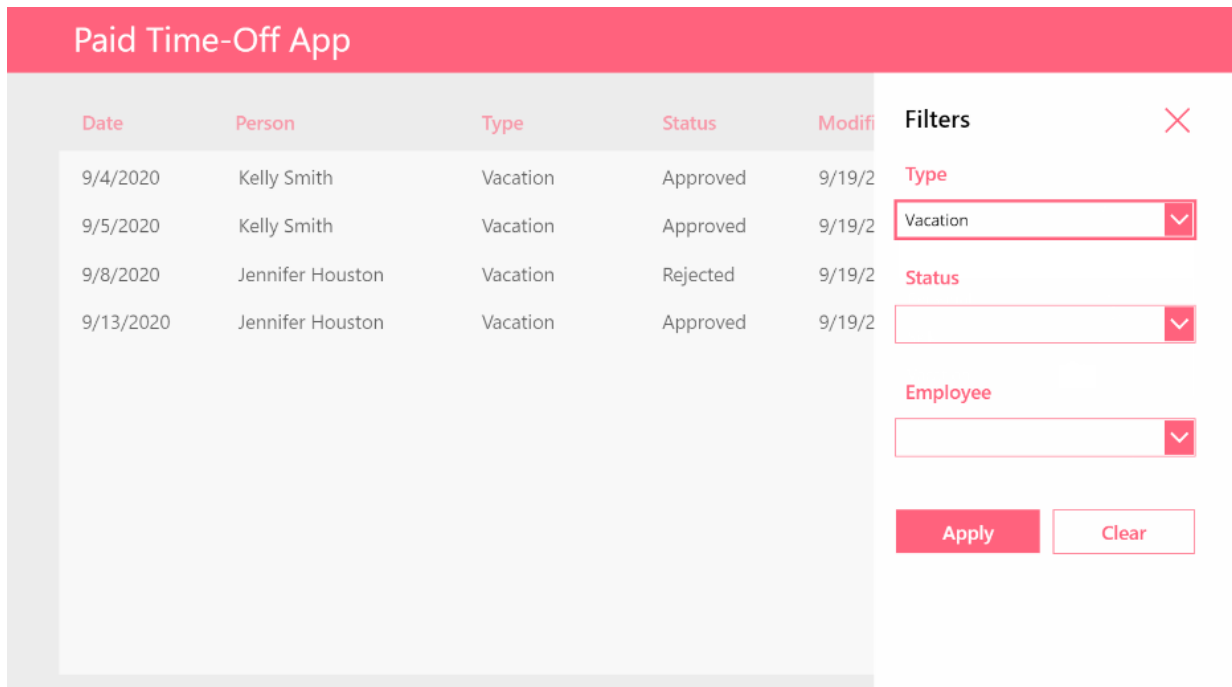
Gallery data does not automatically refresh when connected to a cloud datasource (SharePoint List, Dataverse, etc.) but no [CRUD operations](#) were performed before entering the screen. In this scenario update, the gallery before the screen is loaded using the Refresh function.

Consider giving users the ability to manually refresh a cloud datasource by pressing a refresh button/icon.



Filter Large Datasets

Allow users to filter large datasets and get the results they want. Users should be able to [filter on multiple fields at once](#).



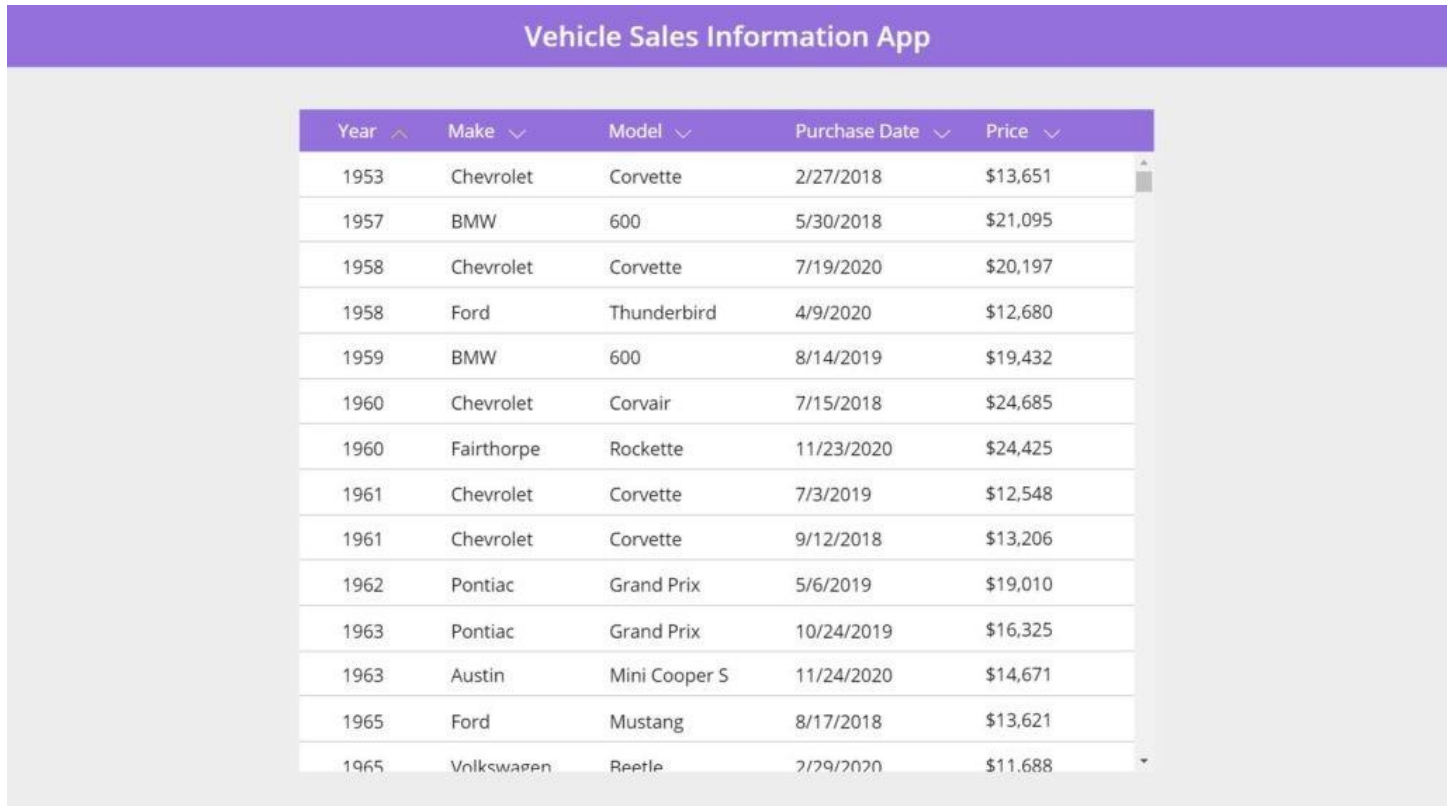
Users should be able to filter on multiple fields at once. Use this coding pattern in the items property of a gallery to support multiple dropdown filters. The code can also be adapted for other control types (combobox, datepicker, etc).

```
// ITEMS property of a gallery with three dropdowns filters
Filter(
    'Paid Time Off',
    drp_Type.Selected.Value=Blank() Or TimeOffType=drp_Type.Selected.Value,
    drp_Status.Selected.Value=Blank() Or Status=drp_Status.Selected.Value,
    drp_Employee.Selected.Value=Blank() Or Employee=drp_Employee.Selected.Value
)
```

Define The Sort Order

Data displayed in a gallery must be sorted. Galleries should be sorted by the gallery item's title (ascending) or a date (descending) displayed in the gallery. If not sorting by another column clearly indicate the sort order on the screen.

Giving a user the [ability to select the sort column and order](#) is recommended but not required.



Year ^	Make v	Model v	Purchase Date v	Price v
1953	Chevrolet	Corvette	2/27/2018	\$13,651
1957	BMW	600	5/30/2018	\$21,095
1958	Chevrolet	Corvette	7/19/2020	\$20,197
1958	Ford	Thunderbird	4/9/2020	\$12,680
1959	BMW	600	8/14/2019	\$19,432
1960	Chevrolet	Corvair	7/15/2018	\$24,685
1960	Fairthorpe	Rockette	11/23/2020	\$24,425
1961	Chevrolet	Corvette	7/3/2019	\$12,548
1961	Chevrolet	Corvette	9/12/2018	\$13,206
1962	Pontiac	Grand Prix	5/6/2019	\$19,010
1963	Pontiac	Grand Prix	10/24/2019	\$16,325
1963	Austin	Mini Cooper S	11/24/2020	\$14,671
1965	Ford	Mustang	8/17/2018	\$13,621
1965	Volkswagen	Beetle	2/29/2020	\$11,688

Do Not Show Live Updates For Search Results

Do not update the gallery with new search results as the user types into a text box. This causes poor performance. Each keypress triggers a new query to the cloud datasource and renders the output on the screen. The consequence of multiple queries being executed at once will be slowness in the app.

```
// Items property of a searchable gallery connected to a text input  
Search(Locations, txt_SearchLocation.Text, "Address", "City", "Province")
```

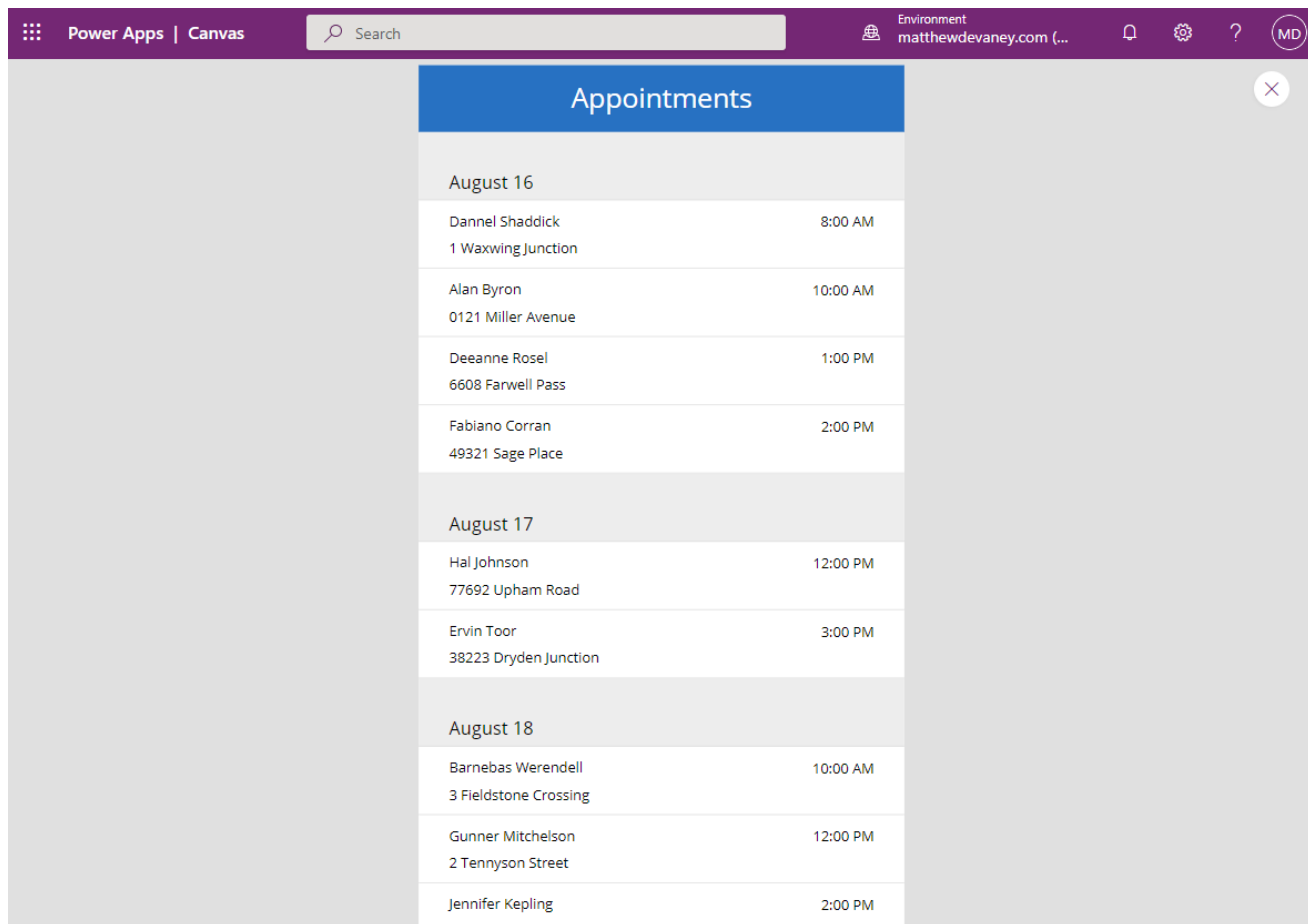
Instead, require the user to [type their search terms into a text input and press a button](#) to search once finished. Set a variable in the button's OnSelect property and use it as the search string.

```
// OnSelect property of a search button
Set(gblSearchLocation, txt_SearchLocation.Text)

// Items property of a searchable gallery using a global variable
Search(Locations, gblSearchLocation, "Address", "City", "Province")
```

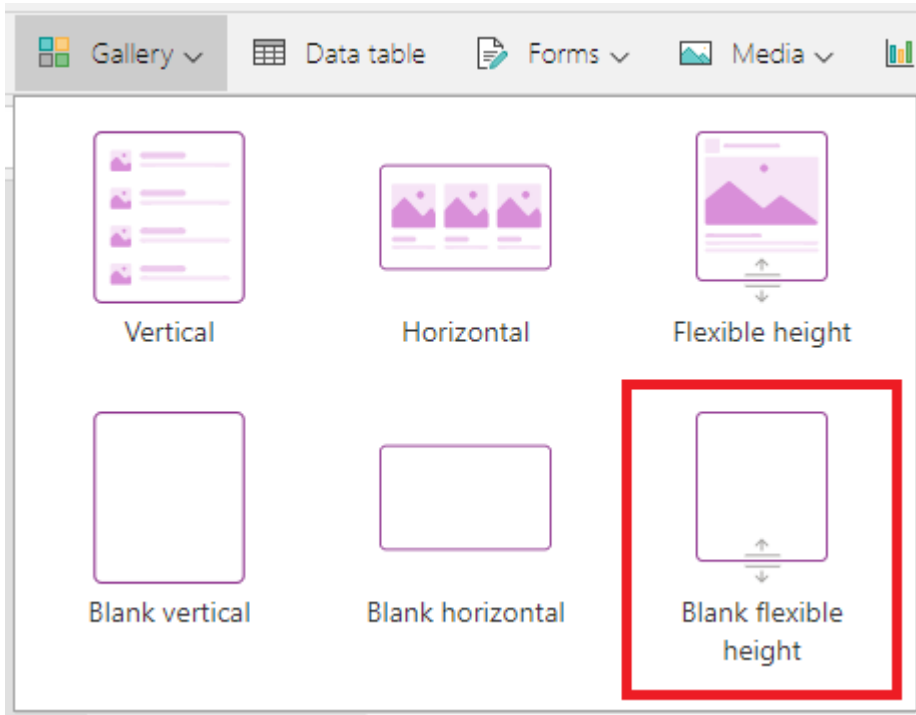
Avoid Nested Galleries

Nested galleries are a common source of poor app performance. They are slow to load and consume a lot of memory. In some cases they have been known to cause crashes in apps with large memory usage. Use collections to create a [grouped gallery](#) instead of nesting.



Use Flexible Height Galleries

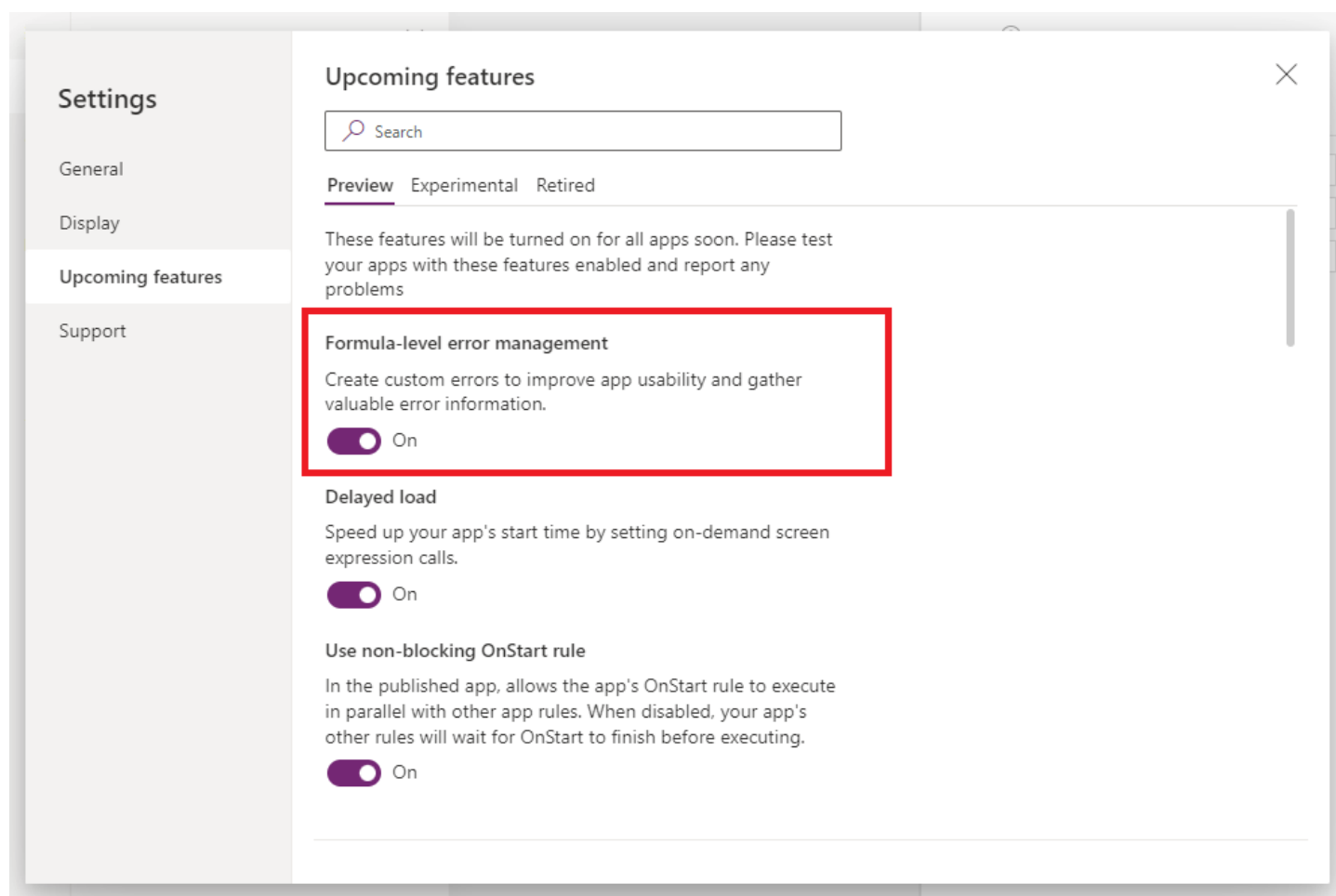
Always use a [flexible height gallery](#) instead of a blank vertical gallery. Flexible galleries have all of the same properties as a normal gallery plus they will expand to fit a row's contents. If an app's design changes to require flexibility during the development process it will be necessary to start over and completely redo the gallery.



Error-Handling

Enable Formula-Level Error Management

Open Power Apps advanced settings and turn on formula-level error management. It enables the [IfError function](#), the [IsError function](#) and the app's [OnError property](#) to be used.



Patch Function Error-Handling

Check for errors anytime data is written to a datasource with the [Patch function](#) or [Collect function](#). Even if the submitted record(s) are validated, good connectivity and the correct user permissions cannot be assumed. This is not necessary for local collections stored in memory.

```
// create a new invoice
If(
  IsError(
    Patch(
      'Invoices',
      Defaults('Invoices'),
      {
        CustomerNumber: "C0001023",
        InvoiceDate: Date(2022, 6, 13)
        PaymentTerms: "Cash On Delivery",
        TotalAmount: 13423.75
      }
    )
  ),
  // on failure
  Notify("Error: the invoice could not be created", NotificationType.Error),
  // on success
  Navigate('Success Screen')
)
```

Power Apps Forms Error-Handling

Write any code should be executed after a [Power Apps form](#) is submitted in its [OnSuccess and OnFailure properties](#). If form submission is successful, use the OnSuccess property to control what happens next. Otherwise, use the OnFailure property to display an error message that tells the user what went wrong.

```
// OnSelect property of the form's submit button
SubmitForm(frm_Invoice)

// OnSuccess property of the form
Navigate('Success Screen');

// OnFailure property of the form
Notify(
    "Error: the invoice could not be created",
    NotificationType.Error
)
```

Do not write any code after the [SubmitForm function](#) used to submit the form. If form submission fails Power Apps will still move onto the next line of code. This can result in loss of data.

```
// OnSelect property of the form's submit button
SubmitForm(frm_Invoice)
Navigate('Success Screen');
```

Power Automate Flow Error-Handling

When a Power Automate flow [triggered from Power Apps](#) its response must be checked for errors. Flows can fail due to poor connectivity. They can also return a failure response or a result with the incorrect schema. If Power Apps does not know the flow failed it will continue as normal.

```
// Get customer invoices
If(
    IsError(
        GetAllCustomerInvoices.Run("C0001023")
    ),
    // On failure
    Notify("Error: could not retrieve customer invoices", NotificationType.Error),
    // On success
    Navigate('Success Screen')
)
```

If Error Function

Use the [IfError function](#) to handle calculations that require a different value when an error occurs. In this example, if gblTasksTotal equals 0 the IfError function will return 0 instead of throwing a “divide by zero” error.

```
// calculate the percentage of tasks completed
IfError(
    gblTasksCompleted/gblTasksTotal,
    0
)
```

Handling Unexpected Errors

An app's OnError property is triggered when an unexpected error occurs. An [unexpected error](#) is any error that is not handled using the IfError or IsError function.

Use this code to quickly locate the source of unexpected errors and fix them. Do not leave this on in a production app since the error message is helpful for a developer but is confusing to a user.

If you want to log the unexpected errors in a production app use the [Trace function](#) and [Azure Application Insights](#) to silently log the errors.

```
// unexpected error notification message
Notify(
    Concatenate(
        "Error: ",
        FirstError.Message,
        "Kind: ",
        FirstError.Kind,
        ", Observed: ",
        FirstError.Observed,
        ", Source: ",
        FirstError.Source
    ),
    NotificationType.Information
);
```

Optimizing App Performance

Load Multiple Datasets Concurrently

Making connector calls sequentially is slow because the current connector call must be completed before the next one starts. The [Concurrent function](#) allows Power Apps to load data faster by simultaneously processing multiple connector calls at once. Only use the Concurrent function to retrieve data stored in cloud. There is no advantage to using concurrent when working with data already on the device (i.e. variables and collections).

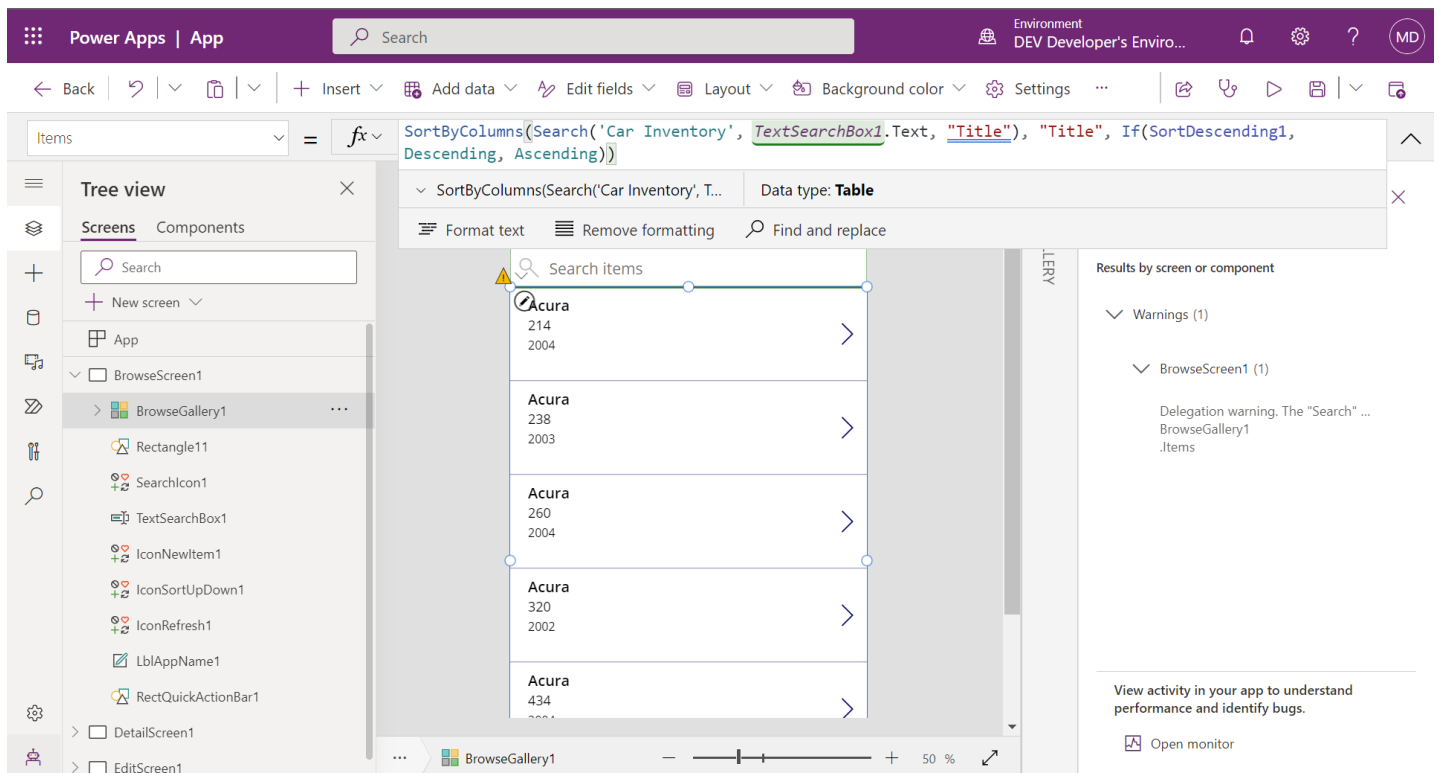
```
// Sequential code execution (slower)
Set(
    gblUserProfile,
    Office365Users.GetUserProfileV2(User().Email)
);
ClearCollect(
    colActiveProjects,
    Filter(
        Projects,
        ProjectStatus.Value="Active"
    )
)

// Simultaneous code execution (faster)
Concurrent(
    // Thread #1
    Set(
        gblUserProfile,
        Office365Users.GetUserProfileV2(User().Email)
    ),
    // Thread #2
    ClearCollect(
        colActiveProjects,
        Filter(
            Projects,
            ProjectStatus.Value="Active"
        )
    )
)
```

Write Formulas That Use Delegation

Always write formulas that can be [delegated to the cloud datasource](#). Delegation is when data operations such as filter, lookup and search are performed in the cloud (i.e. SharePoint, Dataverse) instead of on the user's device. Data operations can be performed faster in the cloud because there are more computing resources than a laptop or mobile phone. Also, less data will be transmitted to the user's device because it has already been filtered by the datasource.

Refer to the official Power Apps documentation to determine which Power Fx functions can be delegated. The supported functions are different for [SharePoint](#), [Dataverse](#) & [SQL](#). A warning will appear in the [app checker](#) when a function cannot be delegated.



Dataverse views are not subject to delegation rules. Use Dataverse views to [write filter criteria](#) that cannot be delegated using Power Apps formulas.

```
Filter(
    'Device Orders',
    'Device Orders (Views)'.Active Device Orders'
)
```


Cache Data In Collections And Variables

Store frequently used data in [collections](#) and [variables](#). Data stored in memory can be accessed very quickly. A cloud datasource must receive a connector call, perform a query and send a response back to the device before data can be displayed on-screen.

```
// Store the currency exchange rates table in memory for quicker access
ClearCollect(
    colCurrencyExchangeRates,
    'Currency Exchange Rates',
)
```

Limit The Size Of Collections

Limit the size of collections to the least number of rows and columns that required by the app. Mobile devices have tight restrictions on memory usage. Collections are stored in the device's memory. If too much memory is in use the mobile operating system will kill the Power Apps process and the app will crash.

Use the [ShowColumns function](#) to select only specific columns and drop the rest from the collection. Enable [explicit column selection](#) to fetch only table columns used in the app when connecting to Dataverse.

```
// selecting only desired columns from the accounts table
ClearCollect(
    colAccounts
    ShowColumns(
        Accounts,
        "name",
        "city",
        "state",
        "zipcode"
    )
)
```

“Batch Patch” Multiple Changes To A Datasource Table At Once

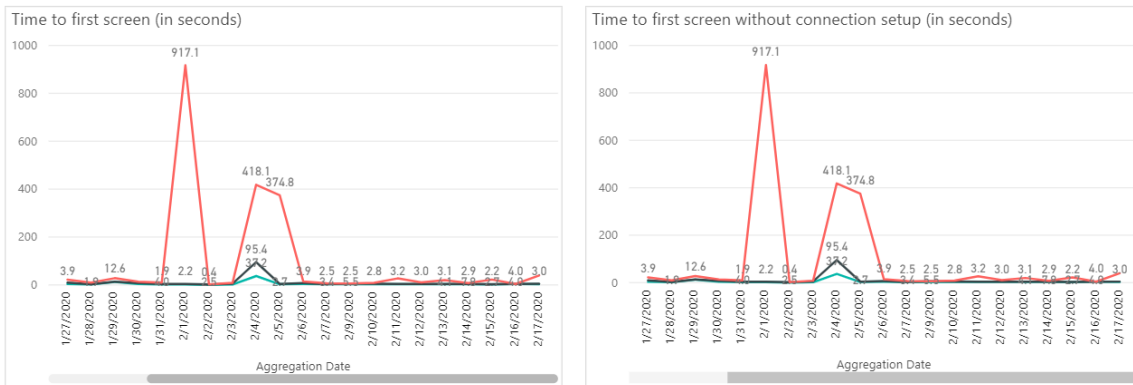
Quickly update multiple records in the same datasource table by using the “batch patch” technique. “Batch patch” enables record updates to be made simultaneously. The traditional ForAll + Patch method is slower because it makes the updates sequentially.

```
// collection of records to update
ClearCollect(
    colUpdateEmployees,
    Table(
        {ID: 2, FullName: "Alice Henderson", Active: true},
        {ID: 4, FullName: "David Wright", Active: false},
        {ID: 5, FullName: "Mary Allen", Active: false}
    )
);
// update records one-by-one (slower)
ForAll(
    colUpdateEmployees,
    Patch(
        Employees,
        LookUp(Employees, ID=colUpdateEmployees[@ID]),
        {
            FullName: colUpdateEmployees[@FullName],
            Active: colUpdateEmployees[@Active]
        }
    )
);
// bulk update multiple records at once (faster)
Patch(
    Employees,
    ShowColumns(
        colUpdateEmployees,
        "ID",
        "FullName",
        "Active"
    )
);
```

Reduce Code In The OnStart Property

The more code that is in the app's [OnStart property](#), the longer an app will take to start. Improve app startup time by initializing global variables in the [OnVisible property](#) of the app's first screen. If possible, further defer setting variables until the screen they are needed.

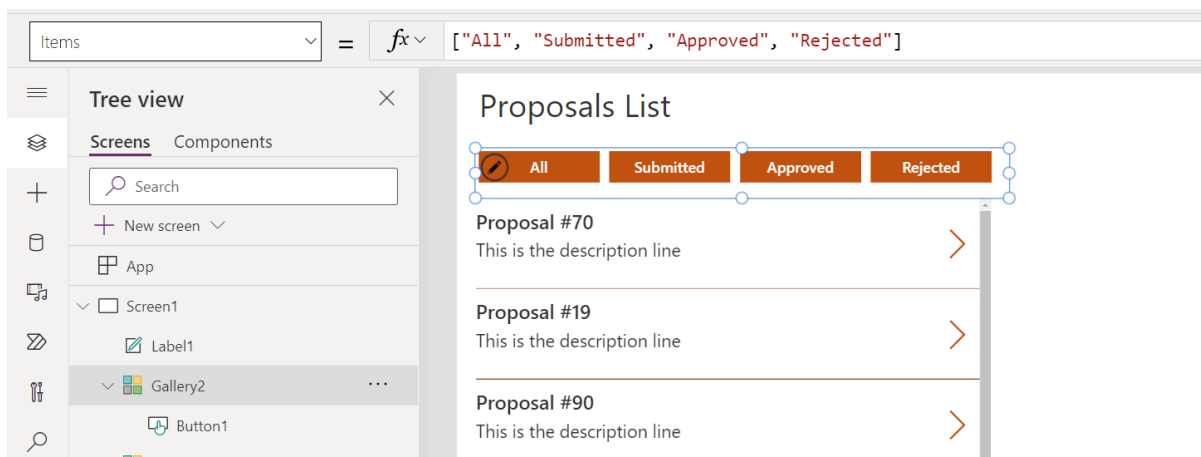
Time to first screen metrics can be found in the app's Analytics page. Go to the maker portal, click on the three dots beside the app, select Analytics (preview), then choose Performance.



Minimize Number Of Controls On A Single Screen

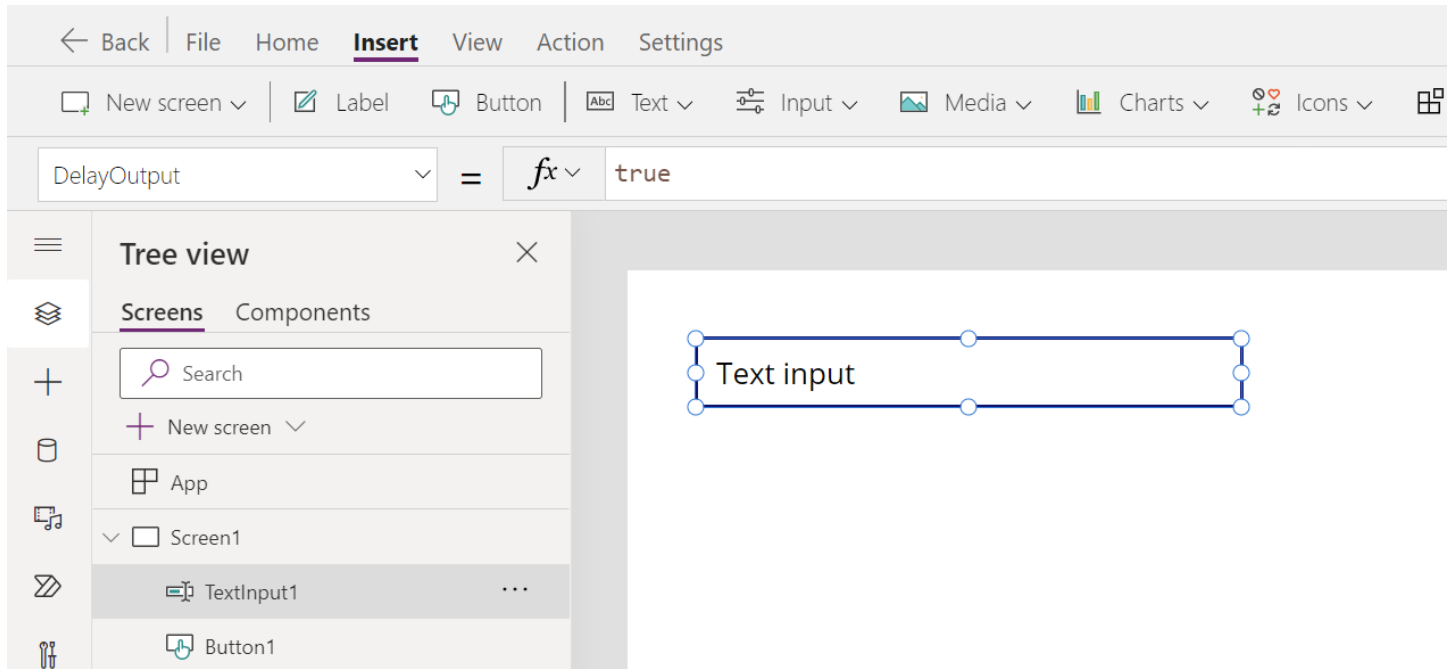
Every control added to a screen increases memory usage when the screen loads. Try to achieve a screen design with the fewest controls possible. A screen with fewer controls on it is faster to render and consumes less memory. For screens with too many controls, consider dividing their functionality across multiple screens.

Use a gallery to display repetitive controls. Each control in a gallery only counts as 1 control no matter how many times it is shown.



Enable DelayOutput For Text Input Controls

The Text property of a text input is updated after each keystroke. Set the [DelayOutput property](#) of the input to true to wait until after the user stops typing. This is useful when building a search bar connected to a gallery. With DelayOutput enabled the app will only make one request to the datasource when typing stops, as opposed to each keystroke.



Do Not Reference Controls On Other Screens

When writing formulas, only reference controls on the current screen. Do not reference controls on other screens. It will force Power Apps to keep that other screen in memory even though it is not being displayed on the device. Use a global variable to store the values found on other screens and refer to the variable instead.

Eliminate The N+1 Problem

The N+1 problem is caused when an app must make N+1 connector calls, where N is the number of items. For example, let's say we want to display a list of business Contacts in a gallery. The Items property requires 1 connector call to get Contacts from the datasource.

```
// ITEMS property of a gallery  
Contacts
```

Each Contact has a related Account (i.e. an organization). To display the Account Name we insert a label into the gallery with this code in the text property. As a result one additional connector call must be made for each row in the gallery. If there are 100 rows in the gallery, there will be 101 total connector calls total (1 gallery +100 rows).

```
// TEXT property of the account name label  
LookUp(Accounts, ID=ThisItem.AccountID, 'Account Name')
```

The solution to the N+1 problem for Dataverse is quite simple. Dataverse automatically fetches the required data in related tables during the connector call for Contacts.

```
// TEXT property of the account name label  
ThisItem.Account.'Account Name'
```

SharePoint lists are not a relational database and cannot return all related data in one connector call. We cannot eliminate N+1 but we can reduce the number of connector calls to SharePoint. Collect all data in the Accounts and Contacts prior to opening the gallery screen. Then add a new column called "Account Name" to the Contacts table by joining it with the Accounts table. Display the resulting collection in the items property of the gallery.

```
// Download all contacts and accounts prior to entering the gallery screen
ClearCollect(colAccounts, Accounts);
ClearCollect(colContacts, Contacts);

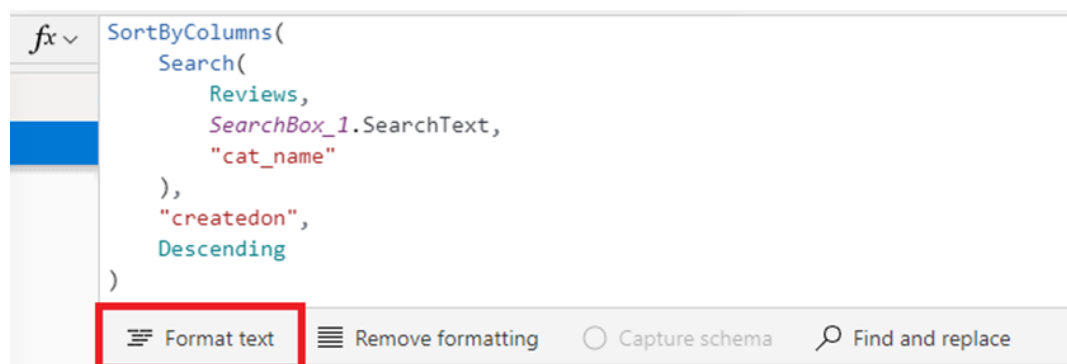
// Join tables to get account name
ClearCollect(
    colGalleryData,
    AddColumns(
        colContacts,
        "AccountName",
        LookUp(Accounts, ID=ThisItem.AccountID, 'Account Name')
    )
);

// new ITEMS property of the gallery
colGalleryData
```

Improving Code Readability

Apply Automatic Formatting

Use the formula bar's [format text command](#) to achieve a consistent coding style throughout a canvas app. Format text automatically applies indentation, spacing and line-breaks to Power Apps code. Well-formatted code has two benefits. It is easier to read and quicker to spot mistakes. A consistent coding style makes it easier for developers to work together on an app.



Use The WITH Function To Improve Readability

Power Apps [With function](#) makes long formulas more readable. For example, this formula calculates the monthly mortgage payment for a house:

```
Value(txt_InterestRate.Text)/100 * Value(txt_LoanAmount.Text) /  
(1 - (1 + Value(txt_InterestRate.Text)/100)^-Value(txt_NumberOfPayments.Text))
```

The mortgage calculation formula cannot be interpreted at-a-glance. It takes effort to parse. Compare it to the formula below using the With function. The formula is now human-readable because any complexity moved into one-time variables.

```

With(
    {
        InterestRate: Value(txt_InterestRate.Text)/100,
        LoanAmount: Value(txt_LoanAmount.Text),
        NumberOfPayments: Value(txt_NumberOfPayments.Text)
    },
    InterestRate * LoanAmount / (1 - (1 + InterestRate)^-NumberOfPayments)
);

```

Choose Consistent Logical Operators

The [logical operator And](#) can be written 3 different ways: And, And(), &&. There are often many ways to do the same thing in Power Apps code. It's OK to choose any one of these options but be consistent.

```

ClearCollect(
    colCustomers,
    {State: "NY", Status: "Active"}
);

// And operator
Filter(
    colCustomers,
    State = "NY"
    And Status="Active"
);

// And() function
Filter(
    colCustomers,
    And(
        State = "NY",
        Status="Active"
    )
);

// && operator
Filter(
    colCustomers,
    State = "NY"
    && Status="Active"
);

```


Join Text Strings & Variables

Combining text can be done multiple ways in Power Apps: the & operator, the [Concatenate function](#) or [\\$-String notation](#). Choose one way of doing it and be consistent.

```
// set variables
Set(gblUserName, User().FullName);
Set(gblStreetAddress, "123 Chestnut Street");

// join text using the & operator
"Hi, my name is"&gblUserName&" and I live at "&gblStreetAddress&".";

// join text using the Concatenate function
Concatenate("Hi, my name is", gblUserName,"and I live at ",gblStreetAddress)&".";

// join text using $-Strings
$"Hi, my name is {gblUserName} and I live at {gblStreetAddress}"
```

Remove IF Statements When The Result Is A True Or False Value

An IF statement that results in true or false is not necessary to write. Get rid of the IF statement and only write the logical comparison

```
// With IF Statement evaluates to true or false
If(gblUserRole="Manager" And gblIsDataLoaded=true, true, false)

// Without IF Statment evalulates to true or false
gblUserRole="Manager" And gblIsDataLoaded
```

Substitute The Self Operator For The Current Control Name

The [Self operator](#) is a concise way to access properties of the current control. Use Self instead of the full control name to make code quicker to understand.

```
// Reference using the current control name
ColorFade(txt_SubmitForm.Fill, -10%)

// Reference using the Self operator
ColorFade(Self.Fill, -10%)
```

Flatten Nested IFs

Nested IFs are when multiple [IF functions](#) are placed inside one other. The more levels a nested IF contains the harder it becomes to understand. Use a flat structure whenever possible to improve code readability.

```
Set(gblBankAccountBalance, 5000);
Set(gblDailyWithdrawlLimit, 1000);
Set(gblWithdrawlAmount, 100);

// Nested IFs
If(
    gblWithdrawlAmount > gblBankAccountBalance,
    Notify("Insufficient funds", NotificationType.Error),
    If(
        gblWithdrawlAmount > gblDailyWithdrawlLimit,
        Notify("Daily withdrawl limit exceeded", NotificationType.Error),
        Notify("You have Withdrawn $"&gblWithdrawlAmount, NotificationType.Success)
    )
);

// Flattened IFs
If(
    gblWithdrawlAmount > gblBankAccountBalance,
    Notify("Insufficient funds", NotificationType.Error),
    gblWithdrawlAmount > gblDailyWithdrawlLimit,
    Notify("Daily withdrawl limit exceeded", NotificationType.Error),
    Notify("You have withdrawn $"&gblWithdrawlAmount, NotificationType.Success)
);
```

Alphabetical Order In Patch & UpdateContext Functions

When [Patch functions](#) have a large number of fields it takes more time to find and update them. Use alphabetical order so the desired field can be quickly located. This technique can also be applied to the [UpdateContext function](#).

```
// create a new record
Patch(
  colContacts,
  Defaults(colContacts),
  {
    Active: true,
    Address: "67 Walnut Grove",
    Name: "Jane Smith",
    PostalCode: "R2G 3V3",
    Province: "Ontario"
  }
);

// update local variables
UpdateContext(
  {
    locAccountID: GUID(),
    locBlockerUserInput: true,
    locIsMenuVisible: false,
    locReadOnlyMode: false,
    locSelectedProperty: "Location"
  }
);
```

Simplify Logical Comparisons When Evaluating A Boolean

A boolean value itself can be used as an argument to the IF function. It is not necessary to write a logical comparison.

```
Set(gblIsBankAccountActive, true);

// logical comparison
If(
    gblIsBankAccountActive=true,
    Navigate('Withdraw Funds Screen'),
    Notify("Bank account is not active", NotificationType.Error)
);

// boolean value only
If(
    gblIsBankAccountActive,
    Navigate('Withdraw Funds Screen'),
    Notify("Bank account is not active", NotificationType.Error)
)
```