Microsoft

PowerApps キャンバス アプリのコーディング規約とガイドライン

ホワイト ペーパー

要旨: このホワイト ペーパーは、大規模な組織内で働く Microsoft PowerApps アプリの作成者を対象としています。オブジェクト、コレクション、変数の命名方法の標準や、保守が容易でパフォーマンスに優れたアプリを一貫して開発し続けるためのガイドラインをご紹介します。

執筆者: Todd Baginski、 Pat Dunn

技術協力: Mehdi Slaoui Andaloussi、 Alex Belikov、 Casey Burke、 Ian Davis、 Brian Dang、 Rémi Delarboulas、 Aniket J. Gaud、 Nick Gill、 Audrie Gordon、 Erik Orum Hansen、 Eric Mckinney、 Santhosh Sudhakaran、 Hubert Sui、 Vanessa Welgemoed、 Keith Whatling

日本語訳監修: 吉田 大貴

目次

はじめに

Microsoft PowerApps は優れた生産性アプリを作成できるマイクロソフトの開発プラットフォームです。このプラットフォームはさまざまなマイクロソフト製アプリ（Microsoft Dynamics 365 for Sales、Microsoft Dynamics 365 for Service、Microsoft Dynamics 365 for Field Service、Microsoft Dynamics 365 for Marketing、Microsoft Dynamics 365 for Talent 等）の構築にも使用されています。大規模組織のお客様はマイクロソフトと同じこのプラットフォームを使用して、自社独自の基幹業務アプリを構築できます。また、組織内の個人ユーザーやチームの皆様も、このプラットフォームを使用することでコードをほとんど (あるいはまったく) 記述することなく、個人用やチーム用の生産性アプリを作成することができます。

このホワイト ペーパーの目的

このホワイト ペーパーはエンタープライズ アプリの作成者 (開発者) 向けに構成されており、PowerApps アプリの設計や構築、テスト、デプロイ、保守を受け持つ一般企業または政府で働く担当者をターゲットとしています。このホワイト ペーパーは、Microsoft PowerApps チーム、マイクロソフトの IT 部門、業界のプロフェッショナルの協力で作成されました。なお、コーディング規約や運用基準は、大規模組織のお客様が独自に策定していただいてもかまいません。ここでご紹介するガイドラインは、アプリの開発を次のような点でサポートできるように作成したものです。

- 簡潔さ
- 読みやすさ
- サポート性
- デプロイと管理の容易さ
- パフォーマンス
- アクセシビリティ

このホワイト ペーパーで扱う範囲

特に明記されていない限り、このホワイト ペーパーで取り上げるすべての機能は、2018 年 12 月以降利用可能なものです。このホワイト ペーパーでは以下については扱いません。

- アプリを構築するための PowerApps の基礎知識。このホワイト ペーパーの対象読者は実務経験のある方を想定していますが、PowerApps アプリの構築方法についての高度な知識がなくても問題ありません。PowerApps に関するブログやチュートリアル、トレーニング リソース、コミュニティ サポートについては、https://docs.microsoft.com/ja-jp/powerapps/index を参照してください。
- Microsoft Power BI を始めとする幅広い Microsoft Power Platform の構成要素
- PowerApps 以外のコード (Microsoft Azure App Service や Function App などのコード)
- ガバナンス全般およびアプリケーション ライフサイクル管理 (ALM)
- 環境の管理。この詳細については、ホワイト ペーパー『PowerApps のエンタープライズ展開を管理する』をご確認ください。

ホワイトペーパーの内容の更新

このホワイト ペーパーの内容は逐次更新される予定です。Microsoft Power Platform の機能や業界標準が変更された場合には、このホワイト ペーパーも更新されます。

マイクロソフトはお客様からのフィードバックへ常に耳を傾け、適宜 Power Platform を改良することで、皆様がより優れたアプリを構築できるよう支援しています。最も効率的なアプローチは新たな機能の登場によって変化していくため、現在のベスト プラクティスが時代に合わなくなる場合もあります。最新の標準とガイドラインを定期的にチェックされてください。

また、このホワイト ペーパーを執筆するにあたり、アドバイスや実体験をご提供いただいた協力者の皆様に**深くお礼を申し上げます**。それでは、ガイダンスを始めましょう。

一般的な命名規則

このセクションでは、命名規則の "キャメル ケース" と "パスカル ケース" について説明します。これらの用語をご存知の場合は、次のセクションに進んでいただいてかまいません。

キャメル ケース

コントロールと変数には、キャメル ケースの使用をお勧めします。キャメル ケースは複合語を小文字の接頭語で始め、以降の単語の頭文字を大文字で表記する命名方法です。オブジェクト名や変数名にスペースは含めません。たとえば、テキスト入力コントロールであれば、`txtUserEmailAddress` のように命名します。

パスカル ケース

データ ソースには、パスカル ケースの使用をお勧めします。パスカル ケースは "アッパー キャメル ケース" と呼ばれることもあります。キャメル ケースと同様に、すべてのスペースを取り除き、単語の頭文字は大文字にします。ただし、キャメル ケースと異なるのは、パスカル ケースでは 1 語目の単語の頭文字も大文字にする点です。たとえば、PowerApps 内の共通データ ソースである Microsoft Office 365 Users コネクタは、コード内では `Office365Users` という名前になります。

オブジェクトの命名規則

PowerApps アプリ内のオブジェクトを作成する際は、画面やコントロール、データ ソースに対して一定のルールで名前を付けることが大切です。そうすることでアプリの保守が容易になり、アクセシビリティが向上し、そうしたオブジェクトを参照する場合にコードが読みやすくなります。

メモ: このホワイト ペーパーの準備中に、命名規則には組織によってさまざまな違いがあることがわかりました。たとえば、あるベテランの作成者は、数式内で参照されている場合にのみコントロールの名前を変更しています。また、自身が作成するコントロールに異なる接頭語を付けている作成者もいます。

もちろん、それでも大丈夫です。このホワイト ペーパーで紹介しているオブジェクトの命名規則はガイドラインにすぎないため、各組織で独自の標準を策定していただいて問題ありません。重要なのは、一貫したルールを設けて、それに準拠することです。

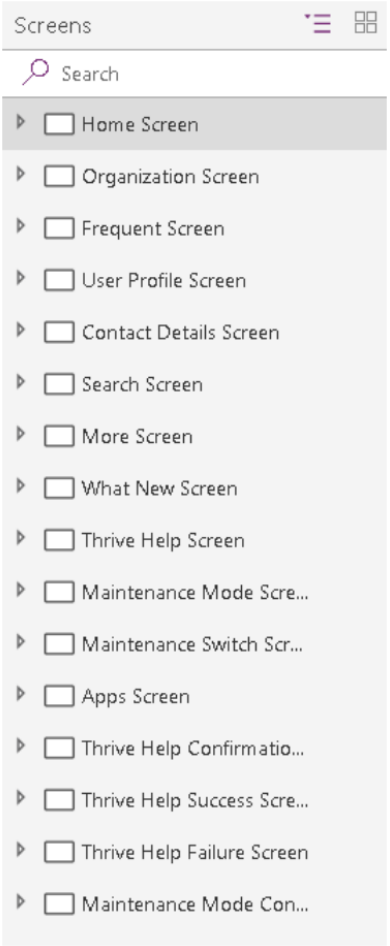
画面名

画面はその画面の目的がわかるような名前にしておくと、PowerApps Studio 内で複雑なアプリを扱いやすくなります。

見落としがちなのが、こうした画面の名前は、視覚障碍のあるユーザーが使用するスクリーン リーダーで読み上げられるという点です。そのため、**画面名には必ず平易な言葉を使用し、単語間にはスペースを入れ、省略形は使わない**ようにします。さらに、名前の末尾に "Screen (画面)" という単語を使い、名前が読み上げられたときにコンテキストがわかるようにすることをお勧めします。

次に、模範例を挙げます。

- `Home Screen`
- `Thrive Help Screen`



次のような名前は避けましょう。

- `Home`
- `LoaderScreen`
- `EmpProfDetails`
- `Thrive Help`

コントロール名

キャンバス上のコントロール名には、すべてキャメル ケースを使用することをお勧めします。最初に 3 文字の型記述子を付けた後、そのコントロールの目的を付け加えます。こうすることでコントロールの種類を見分けやすくなり、数式の作成や検索が容易になります。

良い例: `lblUserName`

以下は、一般的なコントロールの省略形をまとめた表です。

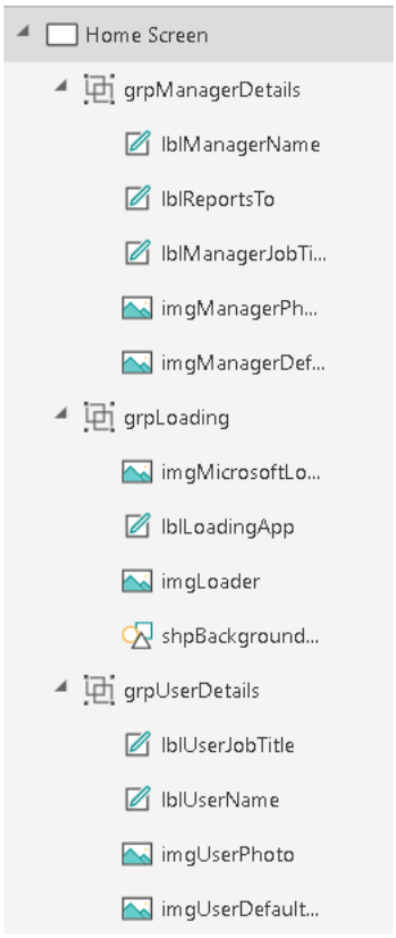
コントロール名	**省略形**
ボタン (button)	btn
カメラ コントロール (camera control)	cam
キャンバス (canvas)	can
カード (card)	crd
コレクション (collection)	col
コンボ ボックス (combo box)	cmb
日付 (dates)	dte
ドロップ ダウン (drop down)	drp
フォーム (form)	frm
ギャラリー (gallery)	gal
グループ (group)	grp
ヘッダー ページの図形 (header page shape)	hdr
HTML テキスト (html text)	htm
アイコン (icon)	ico
画像 (image)	img
ラベル (label)	lbl
ページ セクションの図形 (page section shape)	sec
(四角形、円などの) 図形 (shape)	shp
テーブル データ (table data)	tbl
テキスト入力 (text input)	txt
タイマー (timer)	tim

コントロール名は、同一のアプリ内で一意でなくてはなりません。1 つのコントロールを複数の画面で再利用する場合には、接尾語として末尾に画面の略称を付けます。たとえば `galBottomNavMenuHS` とした場合、"HS" は "ホーム画面 (Home Screen)" を表します。こうしておくと、複数の画面にわたって数式内のコントロールを参照しやすくなります。

次のような名前は避けましょう。

- `postcode`
- `Next`

次の画像からわかるように、常に一定のルールでコントロール名を付けることでアプリのナビゲーション ビューがすっきりします。また、コードもすっきりと読みやすくなります。



データ ソース名

PowerApps アプリにデータ ソースを追加する場合、そのアプリでデータ ソース名を変更することはできません。データ ソース名はソース コネクタのほか、接続を通じて取得したデータ エンティティから引き継がれます。

これには、次のような例があります。

- **ソース コネクタから名前を継承:** Office 365 Users コネクタは、コード内では `Office365Users` という名前になります。
- **接続から取得したデータ エンティティ:** SharePoint コネクタを通じて、**Employees** という名前の Microsoft SharePoint リストが返されるとします。この場合のデータ ソース名は、コード内で `Employees` と表記されます。同じく、この PowerApps アプリでは**先ほどと同じ SharePoint コネクタ**を使用し、Contractors という名前の SharePoint リストにアクセスします。この場合のデータ ソース名は、コード内で `Contractors` と表記されます。

コネクタと接続の詳細については、「PowerApps 用のキャンバス アプリ コネクタの概要」を参照してください。

標準のアクション コネクタ

LinkedIn などの関数にアクセスする標準のアクション コネクタでは、データ ソース名とその操作がパスカル ケースで表記されます (例: UpperUpperUpper)。たとえば、LinkedIn のデータ ソース名は `LinkedIn` に、操作名は `ListCompanies` になります。

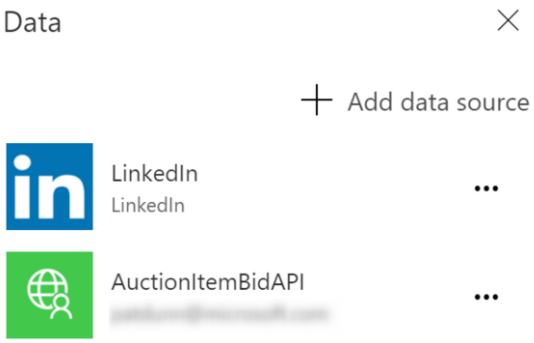
```
ClearCollect(
    colCompanies,
    LinkedIn.ListCompanies()
)
```

カスタム コネクタ

カスタム コネクタは環境内のすべての作成者が作成できます。カスタム コネクタはカスタム API (アプリケーション プログラミング インターフェイス) に接続するためのコネクタで、外部サービスとの連携に使用されます。また、IT 部門が自作した基幹業務アプリ用の API と接続する場合にも使用します。ここでも、データ ソース名とその操作にはパスカル ケースの使用をお勧めします。注意していただきたいのは、PowerApps 内で表示されるカスタム コネクタ名が、実際の名前とは異なる場合がある点です。

たとえば、**MS Auction Item Bid API** という名前のカスタム コネクタがあるとします。

このコネクタから接続を作成して PowerApps アプリに追加すると、**AuctionItemBidAPI** という名前で表示されます。



この理由を調べるには、OpenAPI ファイルの中を見てみましょう。`title` という属性があり、そこに `Auction Item Bid API` と書かれています。

```
"info": {
  "version": "v1",
  "title": "Auction Item Bid API"
},
```

PowerApps はこの属性値からスペースをすべて取り除き、データ ソースの名前として使用します。この属性値の表記をパスカル ケースに (`AuctionItemBidAPI` のように) 変更し、カスタム接続の名前として使用するとよいでしょう。こうしておけば混乱することがありません。この値を変更してから OpenAPI ファイルをインポートし、カスタム コネクタを作成します。

メモ: 既存の OpenAPI ファイルをインポートする代わりに **一から作成** オプションを使用すると、PowerApps からカスタム コネクタ名の入力が求められます。ここで入力する名前は、カスタム コネクタ名と、OpenAPI ファイル内の `title` 属性の値の両方に使用されます。この場合も、`AuctionItemBidAPI` のようなパスカル ケースの名前を入力しておけば問題ありません。

Excel のデータ テーブル

PowerApps は、Microsoft Excel のデータ テーブルを使用して Excel ワークシート内のデータに接続します。データ ソースとなる Excel ドキュメントを作成するときは、次の点に気を付けます。

- データ テーブルは、内容がわかりやすい名前にします。この名前は、データ テーブルに接続するためのコードを記述する際に PowerApps アプリ内に表示されます。
- ワークシートごとに 1 つのデータ テーブルを使用します。
- データ テーブルとワークシートは同じ名前にします。
- データ テーブル内の列の内容がわかりやすい名前にします。
- パスカル ケースを使用します。データ テーブル名の各単語の頭文字は大文字にしましょう (例: `EmployeeLeaveRequests`)。

コードの命名規則

PowerApps アプリにコードを追加する場合は、一貫した命名規則に従って変数やコレクションを命名することがより重要になります。変数が正しく命名されていれば、各変数の型や目的、スコープをすばやく見分けられるようになるはずです。

コードやオブジェクトの命名規則には、組織によってさまざまな違いがあることがわかっています。たとえば、変数の接頭語としてデータ型を使用しているチームもあれば (`strUserName` という名前で文字列を表すなど)、すべての変数名の最初にアンダースコア (_) を付け、IntelliSense でグループ化されるようにしているチームもあります。グローバル変数とコンテキスト変数の表し方も、チームによって異なります。

いずれにせよ、"共通のルールを作成し、常にそのパターンに従うこと" が大切です。

変数名

- 変数の機能がわかりやすい名前にします。変数の目的や用途を考え、それを反映した名前を付けましょう。
- グローバル変数とコンテキスト変数には、異なる接頭語を使用します。

重要ポイント: PowerApps では、コンテキスト変数とグローバル変数に同じ名前を使用できます。このことは混乱を招く要因となります。既定では、曖昧性除去演算子を使用しない限り、数式内ではコンテキスト変数が使用されるためです。こうした状況を回避するには、次の規則に従います。

- コンテキスト変数には `loc` という接頭語を付けます。
- グローバル変数には `gbl` という接頭語を付けます。

- 接頭語の後に付ける名前は、その変数の目的や用途がわかるものにします。複数の単語を使用することも可能で、その場合は単語間を特別な文字 (スペースやアンダースコアなど) で区切る必要はありません。ただし、各単語の頭文字は大文字にします。
- キャメル ケースを使用します。変数名は小文字の接頭語で始め、後に続く各単語の頭文字を大文字で表記します (例: `lowerUppperUpper`)。

次に、模範例を挙げます。

- **グローバル変数:** `gblFocusedBorderColor`
- **コンテキスト変数:** `locSuccessMessage`

次のような名前は避けましょう。

- `dSub`
- `rstFlds`
- `hideNxtBtn`
- `ttlOppCt`
- `cFV`
- `cQId`

短くてわかりにくい `EID` のような変数名は使用せず、`EmployeeId` のような名前にしましょう。

メモ: アプリ内の変数の数が多い場合は、数式バーに接頭語だけを入力すれば、変数の候補が一覧表示されます。このガイドラインに沿って変数名を付ければ、アプリの開発中に目的の変数を数式バーで簡単に見つけることができます。このアプローチは最終的に、アプリを開発する時間の短縮につながります。

コレクション名

- コレクションの内容がわかりやすい名前にします。コレクションの内容や用途を考え、それを反映した名前を付けましょう。
- コレクションの接頭語には `col` をお勧めします。
- 接頭語の後に付ける名前は、そのコレクションの目的や用途がわかるものにします。複数の単語を使用することも可能で、その場合は単語間をスペースやアンダースコアなどで区切る必要はありません。ただし、各単語の頭文字は大文字にします。
- キャメル ケースを使用します。コレクション名は `col` という小文字の接頭語で始め、後に続く各単語の頭文字を大文字で表記します (例: `colUpperUpper`)。

次に、模範例を挙げます。

- `colMenuItems`
- `colThriveApps`

次のような名前は避けましょう。

- `orderscoll`
- `tempCollection`

メモ: アプリ内のコレクションの数が多い場合は、数式バーに接頭語だけを入力すれば、コレクションの候補が一覧表示されます。変数については、このガイドラインに沿ってコレクション名を付ければ、アプリの開発中、目的のコレクションを数式バーで簡単に見つけることができます。このアプローチは最終的に、アプリの開発時間の短縮につながります。

オブジェクトとコードの整理

グループによる整理

画面上のすべてのコントロールは、グループ分けしておくことをお勧めします。コントロールの目的を簡単に見分けられるようになるほか、画面内または画面間での移動が容易になります。また、簡単に折りたたんで画面を見やすくすることができます。ギャラリー、フォーム、キャンバスの各コントロールは既にグループ化されていますが、任意で他のグループに割り当て、よりきめ細かく整理することも可能です。

任意で、試験提供されている強化されたグループ コントロール (英語) を使用すると、グループのネスト化やグループ レベルの設定、キーボード ナビゲーションなどを利用できるようになります。

テキストの書式設定機能

数式の複雑さが増すにつれて、読みやすさや保守性が影響を受けるようになります。複数の関数を含む大規模なコードのブロックは読むのが難しい場合が少なくありません。**テキストの書式設定**機能を使用すると、改行やインデントを追加し、数式を読みやすくできます。コードのコメントではクライアントにダウンロードされるアプリのパッケージで余分な空白が削除されるため、アプリを発行する前に**フォーマットの解除**機能を使用する必要がなくなります。

作成するコントロール数は最小限にする

複雑さを最小限に抑えるために、アプリ内のコントロール数はできるだけ少なくします。たとえば、4 つのイメージ コントロールを用意してそれぞれの Visible プロパティの設定を変え、互いに重ね合わせて使用しているとします。代わりに、イメージ コントロールを 1 つにし、Image プロパティ内にロジックを追加すれば、さまざまなイメージを表示することができます。

コードの最適な配置場所を見極める

PowerApps アプリが複雑になるほど、アプリをデバッグする際に目的のコードを見つけるのが難しくなります。この問題は、パターンを統一することで軽減されます。すべての例を挙げることはできませんが、このセクションではコードの最適な配置場所に関していくつかのガイドラインを紹介したいと思います。

まず、一般的なガイダンスとして、後から見つけやすいようにコードはできるだけ "トップレベル" に配置するようにします。作成者によっては、OnStart プロパティにコードを追加する方法が好まれます。このアプローチに問題はありませんが、OnStart プロパティの制約を考慮して、アプリの動作が遅いと感じさせないような工夫が必要です。コードを OnVisible プロパティに配置するのを好む作成者もいます。これは、見つけるのが簡単で、画面が表示されるたびに確実にコードが実行されるためです。

コードのカプセル化

コードはできるだけ複数の画面に分散させず、すべて 1 つの画面に収めるようにします。たとえば、ある作成者がギャラリー内に組織図を表示する社内向けのブラウザー アプリを作成したとします。組織図内の名前をクリックすると新しい画面に遷移し、対象の従業員のプロフィールが表示されます。この例で、プロフィールを読み込むためのロジックが記述されているのは、ギャラリーの OnSelect プロパティではありません。代わりにこのアプリでは、後続の画面で必要となるすべての変数を Navigate 関数内のコンテキスト変数として渡しています。ユーザーのプロフィールを読み込むすべての処理は、**User Profile** 画面によって行われます。

この例で使用されているギャラリーの OnSelect プロパティには、Navigate 関数が記述されています。

```
Navigate(
    'User Profile Screen',
    Cover,
    {
        locSelectedEmployeeID: ThisItem.id,
        locSelectedEmployeeName: ThisItem.displayName,
        locSelectedEmployeeJobTitle: ThisItem.jobTitle,
        locProfileFetchComplete: false,
        locDirectReportsFetchComplete: false,
        locMgrHierarchyFetchComplete: false,
        locPeersFetchComplete:false,
        locSelectedTab:"Profile"
    }
)
```

その後、前の画面から渡されたユーザー ID を使用して、**User Profile** 画面の OnVisible プロパティによって Office365Users.UserProfileV2 が呼び出されます。続いて実行されるコードでは、渡されたその他のコンテキスト変数が使用されます。

```
ClearCollect(colUserProfile, Office365Users.UserProfileV2(locSelectedEmployeeID))
```

メモ: 前の例では、後続の画面で前の画面の Selected プロパティを参照する代わりに、ThisItem の値をコンテキスト変数として渡しています。前述のアプローチは意図的に使用されています。このアプリ内には **User Profile** 画面への複数の移動パスがあり、この画面には、他の複数の画面からギャラリー経由でアクセスできるためです。この画面はカプセル化されているので、同じアプリ内や他のアプリ内で簡単に再利用できます。

OnStart プロパティ

一般的なルールとして、OnStart プロパティに記述するコードは最小限に抑えます。これは、デバッグに手間がかかるためです。このプロパティのコードをデバッグするには、PowerApps Studio 内で PowerApps アプリを保存し、いったん閉じてから、もう一度開いて再度コードを実行する必要があります。このプロパティ内ではコンテキスト変数を作成できません。いずれかの画面が表示される前に 1 回のみ実行される Application.OnStart として考えましょう。

OnStart プロパティは次のような使い方をお勧めします。

- **画面の遷移指定**: OnVisible プロパティとは異なり、OnStart プロパティでは Navigate 関数を使用できます。そのため、画面遷移を簡単に指定することができます。たとえば、mode という名前のパラメーターを評価し、どの画面を表示するか決定するという使い方が可能です。

```
Navigate(
    Switch(
        Param("mode"),
        "new",
        'New Order Screen',
        "edit",
        'Edit Order Screen',
        "history",
        'Order History Screen',
        'Dashboard Screen'
    ),
    ScreenTransition.None
)
```

- **偽装権限またはデバッグ権限**: OnStart プロパティに、現在のユーザーが電子メール アドレスのリストに含まれているかどうかを照合するコードを追加します。ユーザーがリストに該当する場合はデバッグ モードをオンにし、隠し画面とテキスト入力コントロールを表示します。

```
Set(
    gblAllowDebug,
    If(
        User().Email in [
            "bob@contoso.com",
            "susan@contoso.com",
            "rajesh@contoso.com"
        ],
        true,
        false
    )
)
```

  メモ: Azure Active Directory (AAD) のグループ メンバーシップを確認することで、アプリにセキュリティ設定を適用することもできます。

- **静的グローバル変数**: OnStart プロパティを使用してエラー メッセージのコレクションを作成したり、コントロールの色や罫線の幅など、グローバルなスタイル変数を設定したりします。その他のアプローチについては、「隠し構成画面を作成する」セクションを参照してください。
- **"1 回限りの" コード**: 名前が示すとおり、OnStart プロパティに記述したコードは、アプリの起動時に 1 回のみ、最初の画面が表示される前に実行されます。それに対して OnVisible プロパティのコードは、対象の画面にユーザーが移動するたびに実行されます。そのため、1 回しか実行しないコードについては、OnStart プロパティへの配置を検討します。
- **短時間で実行可能なコード**: OnStart プロパティの具体的なガイダンスについては、「パフォーマンスの最適化」セクションを参照してください。

OnStart プロパティと OnVisible プロパティの詳細については、Todd Baginski の動画「PowerApps の OnStart と OnVisible に関する開発のヒント (英語)」をご覧ください。

OnVisible プロパティ

`OnVisible` プロパティには、ユーザーが画面に移動するたびに毎回実行するコードを記述します。このプロパティにコードを追加する場合は注意が必要です。PowerApps アプリの最初の画面ではできるだけ、`OnVisible` プロパティにロジックを書き込まないようにしましょう。代わりに、コントロールのプロパティのインライン式を使用します。

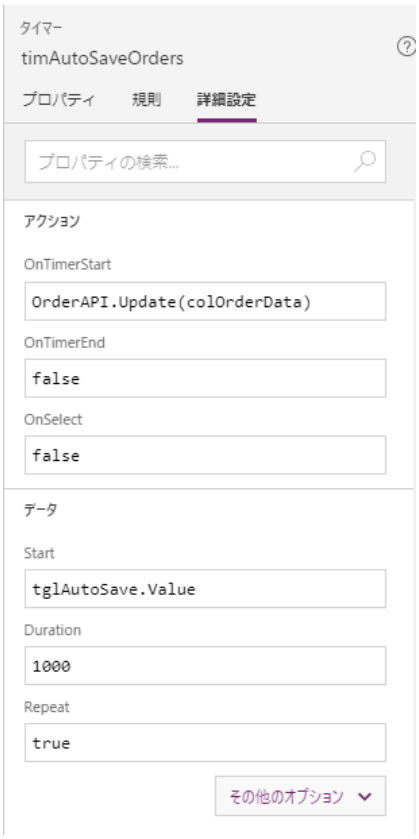
`OnVisible` プロパティは、グローバル変数やコンテキスト変数を設定するのに最適な場所です。しかし、こうした変数を設定するために処理を呼び出す場合には注意が必要になります。`Office365Users.Profile` の呼び出しや、コントロールで静的な色を再利用するための設定など、短時間で実行できる呼び出しなら問題はありません。ただし、実行に時間のかかる複雑なロジックやコードは避けましょう。

`OnVisible` プロパティに関連したパフォーマンス問題の詳細については、「負荷の高い処理の呼び出し」セクションを参照してください。

OnTimerStart プロパティ

イベント ベースでコードを実行する場合、タイマーを使用することで面白い機能を付け加えることができます。タイマー コントロールは非表示にし、`Start` プロパティでブール型の変数またはコントロール ステートを待ち受けるのが一般的な使い方です。

たとえば、自動保存機能のオン/オフをユーザーが切り替えられるフォームを作成する場合、`tglAutoSave` という名前の切り替えコントロールを作成します。その後、この画面のタイマーで `Start` プロパティを `tglAutoSave.Value` に設定すれば、`OnTimerStart` プロパティのコードによってデータを保存することができます。



OnTimerStart プロパティに ClearCollect 関数を使用するコードを記述し、指定した更新間隔でデータをリロードすることも可能です。

OnTimerStart プロパティでは Navigate 関数もサポートされます。この関数を使用すると、特定の条件が満たされた場合に、別の画面へと移動することができます。たとえば、ローディング画面で、すべてのデータが読み込まれた時点でブール型コンテキスト変数を設定すれば、タイマーを通じてデータ表示画面へと移動できます。あるいは、このプロパティを使用して、一定時間操作が行われなかった場合に "セッション タイム アウト" のメッセージ画面に移動することも可能です。

このパターンには 2 つの注意点があります。

- PowerApps Studio 内でアプリを編集している間はタイマーが起動しません。AutoStart が true に設定されている場合や、Start プロパティで式が true と評価された場合でも、OnTimerStart プロパティのコードは実行されません。ただし、プレビュー モードに切り替える (F5 キー) とコードが起動します。
- Navigate 関数を実行する場合、現在の画面でその他のコードを実行してから画面を移動するため、かなりの待ち時間が発生する場合があります。

  たとえば、ローディング画面にタイマー コントロールが配置されているとします。このコントロールの Start プロパティを locRedirect というブール型コンテキスト変数に設定し、次に示すナビゲーション コードを OnTimerStart プロパティに追加します。

```
If(
    locIsError,
    Navigate(
        'Error Screen',
        None,
        {locStatusMessage: locStatusMessage}
    ),
    Navigate(
        'Confirmation Screen',
        None,
        {locStatusMessage: locStatusMessage}
    )
)
```

  ローディング画面の OnVisible プロパティによって従業員の ID が取得され、ID が数字でなかった場合には locRedirect が false に設定されます (数字以外の従業員 ID はエラーと見なされるため)。

```
//Get the employee ID from PeopleData
UpdateContext({locUserID: Text(PeopleData.PersonnelNumber)});
If(
    !IsNumeric(PeopleData.PersonnelNumber),
    //Couldn't get personnel number from basic profile call. error out
    UpdateContext(
        {
            locIsError: true,
            locStatusMessage: "Unable to retrieve user information",
            locRedirect: true //this will cause our redirect timer to fire
        }
    )
);
```

locRedirect が true に設定されるとタイマー コントロールの OnStart コードが実行されます。ただし、OnVisible プロパティのコードがまだ実行されているため、多少の待ち時間が発生します。そのため、後続の数行分のコードについて、追加のエラー チェックを実行します。

```
//there is a small delay when a timer fires. Change the status message only if there is no error
If(
    !locIsError,
    UpdateContext({locStatusMessage: "Authorizing User..."})
);
//Get token
UpdateContext(
    {
        locSuccessFactorToken: LDService.GetSfToken(
            {
                scope: {
```

OnSelect プロパティ

コントロールの OnSelect プロパティ内のコードは、そのオブジェクトが選択されるたびに実行されます。オブジェクトの選択は、ボタンのクリックやテキスト入力コントロールの選択など、ユーザーの操作によって発生します。このプロパティで実行されるコードでは、フォームのデータを検証して確認メッセージやヒントとなるテキストを表示するほか、データソースを相手としたデータの読み書きなどを実行できます。

メモ: OnSelect プロパティには、実行に時間のかかるコードは記述しないようにします。これは、アプリの応答が停止していると思われないためです。詳細については、「パフォーマンスの最適化」および「負荷の高い処理の呼び出し」のセクションを参照してください。また、読み込み中を示すインジケーターやステータス メッセージを使用して、処理の遅さを意識させないようにするのも一案です。

OnSelect プロパティに設定されたコードは、コントロールが Select 関数を使用して選択された場合にも実行されます。お勧めの使用パターンとして、アプリの起動時に表示されるローディング画面の例を紹介しましょう。この画面にラベル コントロールを配置し、「アプリのデータを読み込んでいます」といったメッセージを表示します。このラベル コントロールの OnSelect プロパティを使用すると、データ ソースを呼び出して変数を初期化し、アプリのホーム画面に移動することができます。この場合、ラベル コントロールを選択するには、アプリの OnStart プロパティで Select 関数を呼び出します。

初期化コードは OnStart プロパティまたは OnVisible プロパティ内に配置することもできますが、前述のアプローチには次のようなメリットがあります。

- OnVisible のコードは画面の移動に対応していません。そのため、タイマーなどのコントロールにナビゲーション コードを追加する必要があります。
- OnStart コードはスプラッシュ スクリーンの表示を長引かせる原因になります。また、プレビューで提供されている **非ブロッキングの OnStart 規則を使用** 機能をオンにしている場合、想定外の結果をもたらす可能性があります。

- 先ほどのラベル コントロールがプログラムによって選択されると、データを読み込む間に、視覚障碍のあるユーザーのためにラベル コントロールのテキストがスクリ

ーン リーダーによって読み上げられます。スクリーン リーダーが使用されていると、「画面を読み込んでいます」、「アプリのデータを読み込んでいます」、「ホーム画面」といった文言が読み上げられるため、非常に効果的です。

注意: `OnVisible` プロパティ内の `Select` 関数を使用してコントロールを選択し、そのコントロールで `Navigate` 関数を使用して別の画面に移動した場合、画面を編集できない場合があります。これを避けるには、アプリ内の隠し設定画面に配置した切り替えコントロールを使用します。`OnSelect` プロパティ内でこの切り替えコントロールの状態をチェックしてから、`Navigate` 関数を呼び出すようにします。

企業向けのその他のヒント

- 最初のステートメントの後に 2 つ目の論理式が続く場合、明示的に `If` を記述して "入れ子" にする必要はありません。

```
If(
    One = 1,
        UpdateContext({Nothing: false}),
    If(One = 2,
        UpdateContext({Nothing: true}),
    If(One = 3,
        UpdateContext({All: true})
)))
```

- 2 つ目の論理式を書く場合は `If` を明示的に書かず、論理式のみを記述します。

```
If(One = 1,
      UpdateContext({Nothing: false}),
   One = 2,
      UpdateContext({Nothing: true}),
   One = 3,
      UpdateContext({All: true})
)
```

- 長い式の使用はできるだけ避けてください。
- コードの書式を手動で設定する場合は、次のガイドラインに従います。
    - 各セミコロンは改行を意味します。

```
ClearCollect(AwesomeCollection, {AwesomeStuff: "Awesome"});
UpdateContext({TemplatesGood: true});
Set(GlobalVariable, "On")
```

    - 一行が長い数式には、適当な場所に改行を入れるようにします。かっこやコンマ、コロンの前後に入れるとよいでしょう。

コーディングの一般的なガイドライン

ターゲットのクリック

コントロールのグループがクリックされたときにアクションを実行しなければならない場合、選択可能なアプローチは 3 つあります。

- 最もシンプルなアプローチは、対象のコントロールをグループ化した後、そのグループの `OnSelect` プロパティにクリック イベントを割り当てる方法です。
- グループ内のコントロールのいずれか (最も重要なコントロール) にクリック イベントを追加した後、グループ内の残りの全コントロールで `Select(controlWithLogic)` を `OnSelect` プロパティに追加し、ロジックを記述したコントロールを選択します。最初のアプローチの場合、追加のコントロールは必要なく、エディター内で簡単にコントロールを選択できます。
- グループを覆うように透明の長方形を配置し、長方形の `OnSelect` プロパティを使用します。

お勧めは 3 つ目のアプローチです。理由は、グループに含まれるコントロールが変わっても、コードには影響しないためです。また、クリックできる領域をより柔軟に設定できます。長方形の内部のコントロールを画面上で直接選択するには多少コツがいりますが、エディターの左側にある **[画面]** パネルを使えばコントロールを個別に選択できます。

このアプローチの詳細については、Todd Baginski のブログ記事「PowerApps アプリで透明の四角形を活用する方法 (英語)」を参照してください。

変数とコレクション

コンテキスト変数

コンテキスト変数の利用は最小限にします。どうしても必要な場合にのみ使用するようにしましょう。

コンテキスト変数とグローバル変数を適切に使い分けることが重要です。ある変数をすべての画面で利用できるようにするには、グローバル変数を使用します。一方、変数のスコープを単一の画面に限定する場合には、コンテキスト変数を使用します。

グローバル変数の方が適切な状況では、コンテキスト変数を画面間で渡すのは避けましょう (その方がデバッグもはるかに容易です)。

必要なコンテキスト変数はすべて、1 回の `UpdateContext` の呼び出しでまとめて更新します。こうすると、コードがより効率的になって読みやすくなります。

たとえば、次のような処理を呼び出して、複数のコンテキスト変数を更新します。

```
UpdateContext({All: true, Nothing: false, One: 1, Two: "two"})
```

各処理の呼び出しを個別に記述しないようにします。

```
UpdateContext({All: true});
UpdateContext({Nothing: false});
UpdateContext({One: 1});
UpdateContext({Two: "two"})
```

グローバル変数

変数が 1 つで済む場合は、複数の変数を使用しないようにします。以下は複数の変数の例です。

```
Set(SelectedMeetingId,First(Filter(AllFutureMeetings,isCurrent = true)).Id);
Set(SelectedMeetingName,First(Filter(AllFutureMeetings,isCurrent = true)).Subject);
Set(SelectedMeetingStartTime,First(Filter(AllFutureMeetings,isCurrent = true)).Start);
Set(SelectedMeetingEndTime,First(Filter(AllFutureMeetings,isCurrent = true)).End);
Set(SelectedMeetingHours,DateDiff(SelectedMeetingStartTime,SelectedMeetingEndTime,Hours));
```

代わりに次のように記述すれば、変数が 1 つで済みます。

```
Set(SelectedMeeting, ThisItem);

SelectedMeeting.Id;
SelectedMeeting.Subject;
SelectedMeeting.Start;
SelectedMeeting.End;
SelectedMeeting.Start;
DateDiff(SelectedMeeting.Start, SelectedMeeting.End, Hours)
```

コレクション

コレクションの利用は最小限にします。どうしても必要な場合にのみ使用するようにしましょう。

`Clear;Collect` の代わりに `ClearCollect` を使用します。

```
//Use this pattern
ClearCollect( colErrors, { Text: gblErrorText, Code: gblErrorCode } );

//Not this pattern
Clear(colErrors);
Collect( colErrors, { Text: gblErrorText, Code: gblErrorCode } )
```

ローカル コレクション内のレコード数をカウントするには、`Count(Filter())` ではなく `CountIf` を使用します。

入れ子の使用

不必要なデータ カードやキャンバスは使用しないようにします。ギャラリーが入れ子になっている場合は特に注意します (入れ子になったギャラリーは将来的に機能しなくなります)。

`ForAll` 関数のような他の演算子についても、入れ子の使用は避けましょう。

```
ClearCollect(FollowUpMeetingAttendees,ForAll(ForAll(Distinct(AttendeesList,EmailAddress.Address),LookUp(Attend
```

パフォーマンスの最適化

OnStart コード

`OnStart` プロパティは、アプリの初期化に必要な 1 回限りの処理を呼び出す際に非常に役立ちます。データの初期化処理の呼び出しにもこのプロパティを使用したくなるかもしれません。しかし、`OnStart` コードが実行されている間、ユーザーはアプリのスプラッシュ スクリーンと「データを読み込んでいます」というメッセージを見続けることになるため、読み込み時間が長く感じられます。

ユーザー エクスペリエンスを高めるためには、ダブル ダッシュ (--) などのデータのプレースホルダーをホーム画面に表示しておくことをお勧めします。その後データを取得したら、

内容を置き換えるようにします。こうすることで、ユーザーはホーム画面上のコンテンツを読み始めたり、データに依存しないコントロールを操作したりできます。たとえば、**[概要]** 画面を開くことができます。

Concurrent 関数

PowerApps ではデータ ソースを呼び出す際、モジュール内の上にあるものから順に呼び出します。複数の呼び出しを実行する場合、この上から順番に呼び出す方法がアプリのパフォーマンスに良くない影響を及ぼす場合があります。その回避策としては、タイマー コントロールを使用して、データの呼び出しを同時に実行する方法があります。ただし、このアプローチは保守とデバッグが難しく、タイマー間に依存関係がある場合は特に難しくなります。

Concurrent 関数を使用すると、タイマー コントロールを利用しなくても、複数のデータの呼び出しを同時に実行することができます。アプリ内のタイマー コントロールの OnTimerStart プロパティに複数の API の呼び出し処理が記述されている場合、次のコード スニペットで置き換えられます。保守の面ではこちらのアプローチの方がはるかに容易です。

```
    )
);
Navigate('Home Screen',Fade);

Concurrent(
    //Stores Manager details in CurrentManagerHierarchy collection
    ClearCollect(
        colCurrentManagerHierarchy,
        PeopleApi.PeopleGetMyManagerHierarchy({includePhoto: false})
    );
    //Stores peers of the Manager in CurrentPeers collection
    ClearCollect(
        colCurrentPeers,
        PeopleApi.PeopleGetUserDirectReports(
            Last(colCurrentManagerHierarchy).UserPrincipalName,
            {includePhoto: false}
        )
    );
    //Removes the manager from the CurrentPeers collection
    Remove(
        colCurrentPeers,
        Filter(
            colCurrentPeers,
            UserPrincipalName = locCurrentUser
        )
    ),
    //Collects direct reports in CurrentDirectReports collection
```

Format text Remove formatting

この呼び出しを実行するには、コードを OnVisible プロパティ内に記述します。こうしたアプローチが煩雑になった場合は、代わりに呼び出しコードをタイマー コントロール内に記述し、そのタイマーの Start プロパティ内で参照される変数を、隠しコントロールの OnVisible プロパティまたは OnSelect プロパティのどちらかで設定することもできます。また、タイマーと他のコントロールを組み合わせて、OnVisible プロパティ内のコードを実行する間、ローディング メッセージを表示することもできます。このアプローチは、アプリがきちんと動作していることをユーザーに知らせる方法として非常に有効です。詳細については、「コードの最適な配置場所を見極める」セクションを参照してください。

メモ: コードの保守をより簡単にするには、OnVisible プロパティを使用することをお勧めします。ただし、OnVisible を使用した場合、Navigate 関数が利用できなくなります。

Concurrent 関数の実例は、Todd Baginski の動画「Concurrent 関数を使用して PowerApps のパフォーマンスを高める方法 (英語)」でご覧いただけます。

委任できる呼び出しと委任できない呼び出し

データ ソースを呼び出す際は、委任できる関数と委任できない関数がある点に注意します。委任できる関数をサーバー上で評価することで、パフォーマンスを高めることができます。委任できない関数はデータをクライアントにダウンロードし、ローカルで評価する必要があります。このプロセスは委任できる呼び出しと比べて時間がかかり、扱うデータ量も多くなります。

詳細については「キャンバス アプリでの委任について」という記事を参照してください。

ローカル コレクションの使用

比較的小さなデータセットの場合、頻繁なアクセスが問題となっている場合は特に、データセットを最初にローカルのコレクションに読み込むようにすることを検討します。その後、対象のコレクション上で関数を実行したり、コレクションにコントロールをバインドしたりします。このアプローチは、委任できない呼び出し処理を頻繁に実行する場合に特に有効です。ただし、データの取得が必要となるため、最初に処理を実行する際にパフォーマンスに影響が出る点と、返せるレコード数に上限がある点に注意してください。詳細については、Mehdi Slaoui Andaloussi のブログ記事「PowerApps のパフォーマンスに関する考慮事項 (英語)」を参照してください。

SQL の最適化

データのバックエンドで Azure SQL Database を使用し、その充実した管理機能や相互接続性のメリットを活かしている組織があるかと思います。しかし、実装がうまく行えていないと同時処理が行えないだけでなく、DTU (Database Transaction Unit) のサイズを引き上げなくてはならず、コスト増につながる可能性があります。

たとえば、マイクロソフトの IT 部門は 1,700 人が参加する内部カンファレンスを開催するために、Thrive Conference アプリを構築しました。このバックエンドでは 100 DTU の SQL Database インスタンスが使用されています。マイクロソフトはパフォーマンス テストの段階で、オペレーション センターの 120 人の従業員に対し、そのアプリを同時に開くよう依頼しました。するとアプリの応答が停止しました。ネットワーク トレースを見ると、PowerApps の接続オブジェクトから HTTP 500 エラーがスローされていたことがわかりました。また、SQL のログから、サーバーはフルに活用されており、呼び出しはタイムアウトしていたことがわかりました。

カンファレンス前にアプリを書き直す時間がなかったため、マイクロソフトの IT 部門は環境を 4,000 DTU にスケールアップして同時処理の要件に対応しました。そのため、最初に予算を組んでいた 100 DTU のサーバーと比べて、コストは大幅に高くなってしまいました。その後、彼らはここで示すアプローチを使用してアプリの設計を最適化しました。今では対象の負荷を処理するのに 100 DTU のサーバーでも十分余裕があり、SQL の呼び出しが格段に速くなりました。

SQL の委任できる関数

前の委任に関するセクションを読み終えたら、委任がサポートされるデータ ソースの一覧をご覧ください。この一覧では、委任がサポートされているよく使用される関数と、Filter 関数と Lookup 関数の述語を確認できます。この情報があれば、データセット全体をダウンロードしてクライアント上で評価を実行せずに済むため、特にモバイル デバイスに関して PowerApps アプリのパフォーマンスに大きな違いが生まれます。

テーブルの代わりにビューを使用

テーブルからテーブルへとスキャンを繰り返してデータを読み込む代わりに、必要な要素を結合したビューを活用することができます。テーブルのインデックスが正しく作成されていれば、大幅な高速化を期待できます。また、サーバーで実行する委任できる関数で評価結果を制限すると、より速くなることもあります。

フローを通じたストアド プロシージャでパフォーマンスをアップ

Microsoft SQL Server を使用する PowerApps アプリでパフォーマンスの向上効果を最大化するためには、Microsoft Flow を実装し、ストアド プロシージャを呼び出します。このアプローチには、データベース設計を PowerApps アプリから切り離せるというメリットもあります。つまり、アプリに影響を与えることなく、基になるテーブルの構造を変更できます。後で説明しますが、このアプローチはセキュリティ面でも優れています。

このアプローチを既に SQL Server コネクタを使用している PowerApps アプリで使用するには、まず既存の SQL Server コネクタをアプリから完全に削除する必要があります。その後、SQL サインインを使用する SQL Server コネクタを新たに作成し、データベース内のストアド プロシージャの実行権限のみを割り当てます。最後に、フロー内でストアド プロシージャを呼び出し、PowerApps アプリのパラメーターを渡します。

前述のフローを作成し、結果を PowerApps に返す方法の詳細については、Brian Dang の記事「SQL ストアド プロシージャから配列を PowerApps に返す方法 (Split メソッド) (英語)」を参照してください。

このアプローチには、次のようなパフォーマンス上のメリットがあります。

- ストアド プロシージャはクエリの実行プランを通じて最適化されます。そのため、より短時間でデータが返されます。
- ストアド プロシージャは該当データのみを読み取りまたは書き込みするよう最適化されるので、呼び出しを委任できるかどうかはあまり重要ではなくなります。
- 最適化されたフローは、コンポーネントとして再利用することができます。そのため、環境内の他の作成者と共有して、一般的な読み取り/書き込みシナリオに利用してもらうことができます。

負荷の高い処理の呼び出し

呼び出すデータや API によっては負荷が高く、処理に時間がかかる場合があります。実行時間が長くなると、パフォーマンスが低いという認識につながります。これには、次のような緩和策があります。

- 後続のページが開く前に、負荷の高い処理を呼び出さないようにします。後続ページの読み込みはできるだけ速く終わるようにし、後続ページに遷移したら、OnVisible プロパティを使用してバックグラウンドで処理を呼び出します。
- ローディング メッセージやアニメーションを使用して、バックグラウンドで処理が進行していることをユーザーに知らせます。
- Concurrent 関数は、複数の呼び出しを並行処理できる便利な関数です。しかし、これを使用すると呼び出しの処理に時間がかかり、後続のコードの実行が妨げられることがあります。

以下は、後続ページに移動する場合の OnSelect プロパティの悪い例です。

```
Set(ShowExportConfirmDialog,false);
If(CheckPlanner,
      ForAll(Tasks,
              Planner.CreateTask(SelectedPlanId,Name,{bucketId:SelectedBucketId,dueDateTime:DueTime,assignments:AssignToId})
            )
  );


ClearCollect(Indexes,{Index:-1});

Navigate(ExportConfirm,None);
```

以下は良い例です。まずは、OnSelect プロパティ内のコードです。

```
Navigate(ExportConfirm,None)
```

次に、後続ページの OnVisible プロパティ内のコードです。

```
If(ExportConfirmed,
    Set(Loading, true);

        If(CheckPlanner.Value,
            ForAll(
                Tasks,Planner.CreateTask(
                SelectedPlan.id,Name,
                {
                    bucketId:SelectedBucket.id,
                    dueDateTime:AssnTaskDueDate,
                   assignments:AssignToUser.Id
                }
               )
            )
        );

    Set(Loading, false)
)
```

パッケージ サイズの制限

PowerApps にはアプリの読み込みを最適化するためのさまざまな機能がありますが、アプリのフットプリントを小さくするのも効果的です。フットプリントの縮小は、古いデバイスを使用する場合や、帯域幅が狭く遅い通信回線を使用する場合に特に重要です。

- アプリ内に埋め込まれているメディアを評価します。使用していないものがあれば削除してください。
- 埋め込み画像のサイズが大きすぎる場合は、PNG ファイルの代わりに、SVG 画像を使用できないか検討してください。ただし、SVG 内でテキストを使用する場合は注意が必要です。使用するフォントはクライアント上にインストールする必要があります。テキストを表示しなければならない場合は、画像にテキスト ラベルを重ねることで、問題をうまく回避できます。
- フォーム ファクターの解像度が適切かどうかを評価します。モバイル アプリの解像度は、デスクトップ アプリほど高くする必要はありません。画像の品質とサイズの適切なバランスを実際に試して見極めます。
- 使用していない画面があれば削除します。アプリの作成者や管理者のみが使用する非表示画面もあるので、誤って削除しないように注意してください。
- 1 つのアプリで多くのワークフローに対応しすぎていないか評価します。たとえば、同じアプリ内に管理者用の画面とクライアント用の画面が含まれている場合は、それぞれ個別のアプリに分けることを検討してください。このアプローチは、複数のユーザーが同じアプリで同時に作業する場合にも適しています。アプリを変更する際、テストへの完全な合格が求められる状況でも "影響の範囲" (テストの量) が抑えられます。

アプリの定期的な再発行

PowerApps の製品開発チームでは、継続的に Power Platform の最適化を図っています。こうした最適化の成果は、下位互換性の維持のために、所定のバージョン以降を使用して発行されたアプリにしか反映されない場合があります。そのため、アプリを定期的に再発行して、最適化の効果を活用できるようにすることをお勧めします。

高度な設定

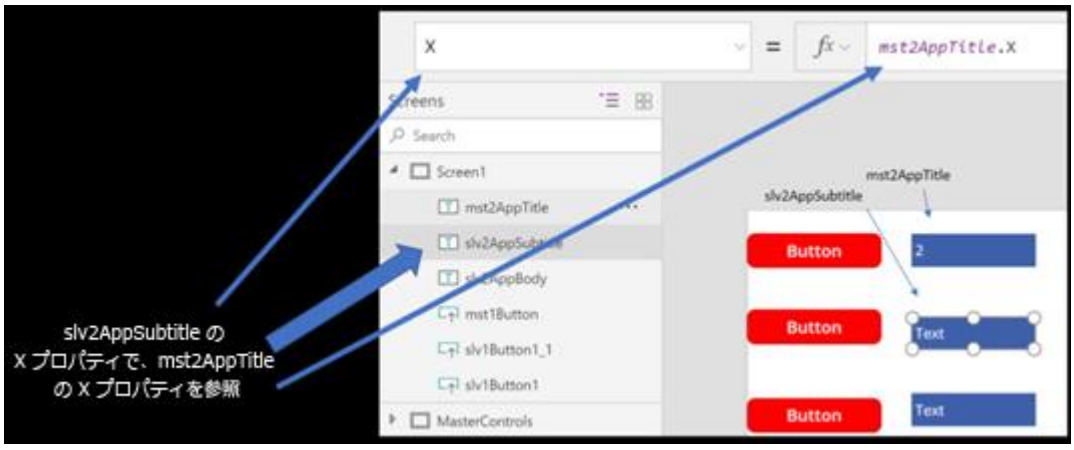
PowerApps の製品開発チームでは、アプリ作成者が任意で有効化できる各種のプレビュー機能を提供しています。こうした機能によって、大幅なパフォーマンス向上効果を期待できる場合があります。たとえば、**遅延読み込み**機能を使用して、アプリの遅延読み込みを有効化にします。すると、初期データの読み込みの際、最初の画面を表示するのに必要な画面とコードのみがランタイムによって読み込まれます。

プレビュー機能は自己責任で使用することになります。そのため、試す場合は必ずアプリのテストを十分に行うようにしてください。

アプリのデザイン

親子関係を使用した相対的スタイル指定

コントロールのスタイル指定では、1 つのコントロールのスタイルを基に、他のコントロールのスタイルを指定することをお勧めします。こうした相対的スタイル指定は通常、色、塗りつぶし、x 座標、y 座標、幅、高さといったプロパティに使用します。

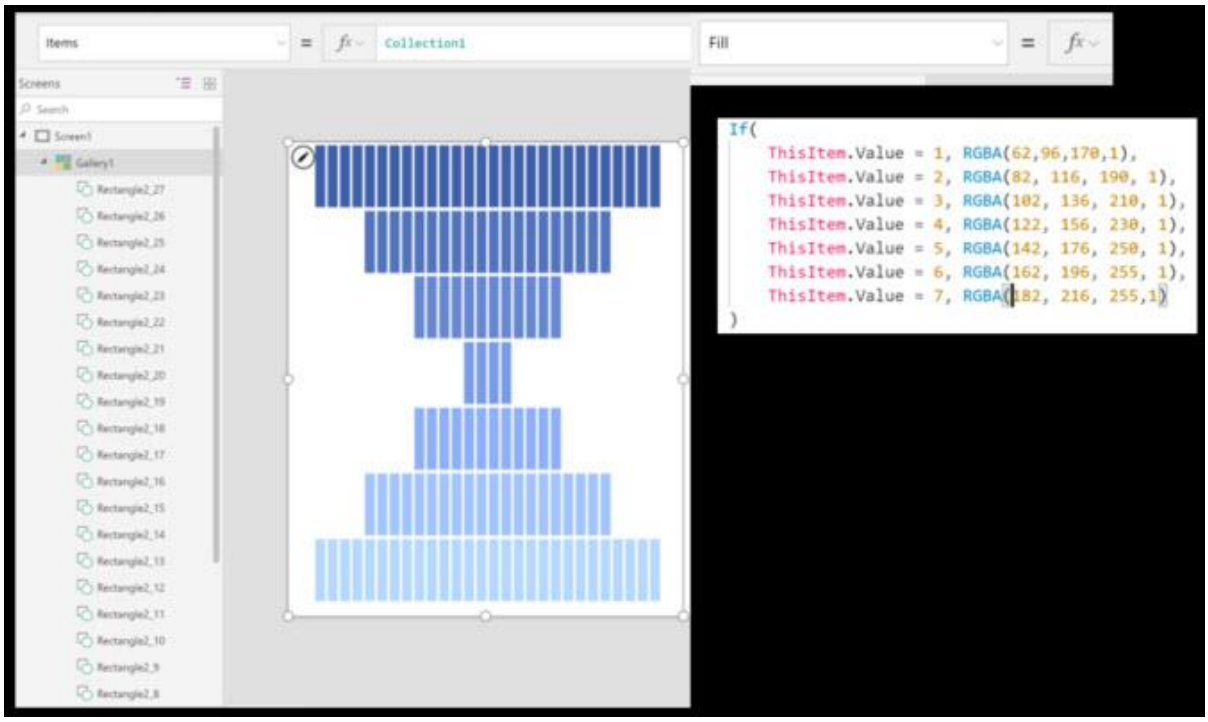


ギャラリー

繰り返しや定型的なデータを扱う場合、ほとんどのケースではギャラリーを使用します。

初期段階では "力技" (複数のコントロールを手動で配置) の方が速いかもしれませんが、後で修正するには非常に時間がかかります。

反復性のある一連の情報またはコントロールを表示する必要がある場合は、ギャラリーを使用して、内部コレクションを作成できないか必ず検討しましょう。



フォームの表示コントロールの代わりに、ギャラリー コントロールを表示フォームとして使用するのも便利です。

たとえば、3 画面から成るデータ参照を目的としたアプリがあるとします。このアプリでは、ユーザーの名前、役職、電話番号が登録されている **Users** というデータ ソースを使用します。

1 つ目の画面の **User List** 画面には `galUsers` という名前のコントロールがあります。このコントロールでは、すべてのユーザーが一覧表示されます。

2 つ目の **User Details** 画面には `galUserDetails` という名前のギャラリー コントロールのみがあります。このコントロールの `Items` プロパティは次のように設定されています。

```
Table(
    {Title: "User Name", Value: galUsers.Selected.DisplayName},
    {Title: "Job Title", Value: galUsers.Selected.JobTitle},
    {Title: "Phone Number", Value: galUsers.Selected.PhoneNumber}
)
```

フォームの表示コントロール内で 3 つの別々のデータ カードを修正するより、このメソッドの方がはるかに高速です。

フォーム

フィールドに繰り返しデータを入力する場合は、フォームが役立ちます。

テキスト ボックスをいくつも使用するのではなく、複数のフィールドをすばやくグループ化できるのもフォームの利点です。

フォームは親子関係を利用して相対的なスタイル指定が可能なため、複数の独立したテキスト ボックスの場合と比べて取り扱いがはるかに容易です。

Common Data Service

編集/挿入の操作は、単一の画面で行うことをお勧めします。

可能であれば、Patch 関数で各コントロールを参照する代わりに、カード ギャラリー コントロールを使用してデータの更新処理を行います。

コンテキスト変数に名前を付ける場合は、どのレコードと関連付けられているかがわかるようにしましょう。

複数のフォーム ファクター

同じ PowerApps アプリのスマートフォン用とタブレット用のレイアウトを作成する場合は、最初に一方のバージョンのアプリを作成し、テストを完了して確定します。その後、もう一方のバージョンに変換してから、レイアウトと画面を修正します。こうすることで、式やコントロール、変数、データ ソースなどの名前をバージョン間で統一しやすくなり、アプリのサポートと開発が**格段に**容易になります。フォーム ファクターの変換方法の詳細については、Todd Baginski のブログ記事「PowerApps アプリのレイアウトを変更する方法 (英語)」を参照してください。

構成値

ユーザー定義の設定値をモバイル アプリ内に格納するには、SaveData と LoadData を使用するとよいでしょう。これらの関数ではデータをキャッシュできるので便利です。

メモ: SaveData と LoadData は、PowerApps プレーヤー クライアント アプリ内でのみ動作します。PowerApps アプリを Web ブラウザーに読み込んだ場合はこれらの関数が機能しないため、アプリ設計の際はこの制限があることを忘れないようにしてください。

アプリを作成する際は、配色、他のアプリへの URL、デバッグ コントロールをアプリ画面に表示するかどうかなどの複数の設定を、1 つの場所から簡単に変更できるようにしておくことをお勧めします。そうすれば、作成者以外がアプリをデプロイする場合に対象の値をすばやく設定できるだけでなく、デプロイ中にコードが壊されるリスクも少なくなります。こうした設定値は、ASP.NET の web.config (英語) ファイルのようなものと考えることができます。

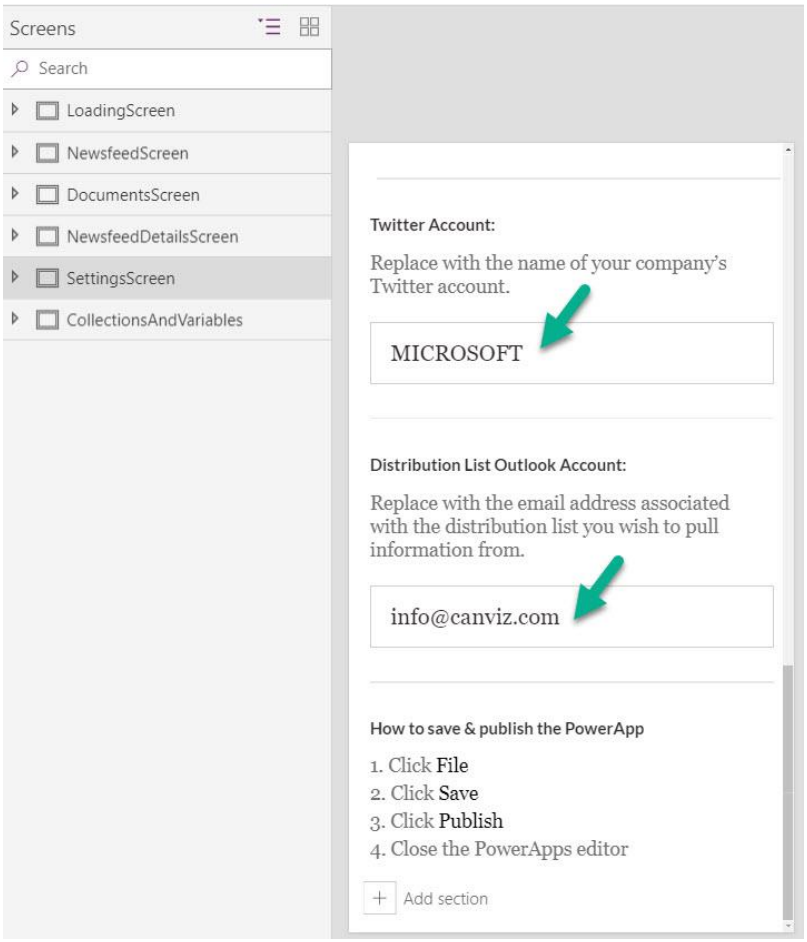
次に、構成値の格納に関するアプローチを簡単なものから順にいくつか挙げてみます。

隠し構成画面を作成する

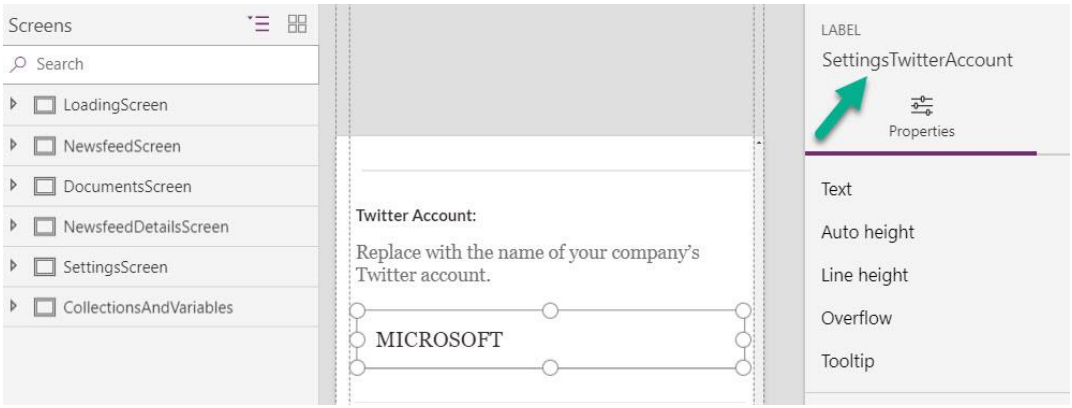
構成値の驚くほど簡単な設定方法として、隠し画面を作成し、テキスト入力コントロール内に構成値を入力しておく方法があります。この方法では、コードを編集することなくアプリの設定を変更することができます。このアプローチを使用するには、次の手順に従います。

1. 構成画面がアプリの最初の画面ではないことを確認します。最初の画面でなければどこに配置してもかまいませんが、見つけやすいよう最後の画面にすることをお勧めします。
2. ユーザーが隠し画面にアクセスできないようにします。
3. アプリの作成者や管理者が隠し画面にアクセスできるようにします。最も簡単なのは、アプリの編集中にのみ隠し画面にアクセスできるようにし、対象の画面に手動で移動する方法です。アプリのホーム画面に、アプリの作成者と管理者のみに表示される隠しボタンを用意しておき、構成画面に移動できるようにしてもよいでしょう。ユーザーがアプリの作成者または管理者であるかどうかを確かめる方法としては、電子メール アドレスのチェック (`User().Email` のチェックなど) や AAD グループのメンバーシップのチェックの他に、PowerApps for Makers コネクタを使用する方法もあります。

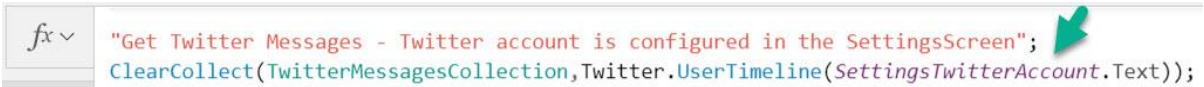
以下は、Microsoft PowerApps の Company Pulse サンプル テンプレートの例です。ここで示すテキスト入力コントロールを使用することで、PowerApps の管理者がアプリの設定値を変更できます。



次の図を見ると、Twitter のアカウント設定値を格納するコントロールの名前がわかります。



以下で、Twitter コネクタを通じて Twitter アカウントのツイートを返すために先ほどの値がどこで使用されているかが確認できます。



このアプローチは構成値を変更する最も簡単な手段ではありますが、いくつかマイナス面もあります。

- 構成値を変更して永続化するには、アプリの再発行が必要です。
- これは構成値がそのアプリ内で永続化されているからです。まずこれらの値を更新するためのプロセスを作成し、その後、他の環境に移行するためにアプリをエクスポートする必要があります。

Common Data Service に構成値を格納する

前述の方法の代わりに、Common Data Service の新しいエンティティを作成し、構成値を格納することもできます。構成値がアプリの外部で永続的に保管されているため、アプリを再デプロイすることなく、いつでも構成値を変更できます。Common Data Service のエンティティでは、環境ごとに固有の値を保管できます。たとえば、運用前環境と運用環境の URL が異なっていてもかまいません。

この方法は、構成値の保持の観点では優れているものの、マイナス面もあります。

- テキスト入力コントロールを使用する場合とは異なり、Common Data Service へのコールバックが必要になります。そのため、わずかながらパフォーマンスに影響を及ぼします。また、Common Data Service が利用できない場合 (ユーザーがモバイル デバイスを使用していて、インターネットに接続できない場合など)、アプリが正しく表示されないことがあります。
- これはキャッシュ機能がないためで、アプリを開くたびに改めて呼び出しが実行されます。
- 呼び出しに失敗しても監視機能がないため、アプリのエラーの発生は、ユーザーの報告を通じてしか知ることができません。

カスタム API を使用する

実装方法は非常に難しいですが、マイクロソフトの IT 部門は Azure App Service の構成値を名前と値のペアで Azure Table Storage 内に格納することに成功しています。

この方法では、OAuth で保護されたカスタム コネクタによって構成値を取得し、出力をキャッシュすることで (値の変更に伴うキャッシュの無効化を含む) パフォーマンスを高めています。Azure Application Insights のアラートにより、問題が起きた場合は通知を受信できるため、ユーザー セッションのトラブルシューティングが格段に容易になります。

エラー処理とデバッグ

エラー処理用の切り替えコントロール

「OnTimerStart プロパティ」セクションでは、タイマー コントロールを使用してエラー処理を行う場合の例について具体的に紹介しました。エラー処理のもう 1 つのパターンとして、切り替えコントロールを使用する方法があります。

次の図をご覧ください。

このアプローチでは、検証やエラー処理のロジックを単一のコントロール内にカプセル化できます。切り替えコントロールを使用すると、複雑な条件を評価し、`true` または `false` の値を発行することができます。他のコントロールはその値を参照してエラー メッセージの表示と非表示を切り替えたり、フォントや罫線の色を変更したり、ボタンを無効化したり、Application Insights にログを書き込んだりすることができます。このコントロールの表示と編集を可能にすると、アプリ作成者がエラー条件のオン/オフを切り替えて、ユーザー インターフェイス (UI) の反応を確認できるようになります。このアプローチは、アプリを開発またはデバッグする際の時間と手間の削減に効果的です。

キャンバス コントロールをデバッグ パネルとして使用する

アプリの開発中やテスト中は、キャンバス コントロールを使用して半透明のデバッグ用パネルを作成し、画面に重ねて表示しておくと便利です。このパネルに必要に応じて編集可能なフィールドやトグルなどの各種コントロールを配置しておけば、アプリを再生モードにしたままで変数を変更することができます。

具体的な手順については、Brian Dang の説明動画「PowerApps のベスト プラクティス: デバッグ パネル (英語)」をご覧ください。

アプリ作成者向けにデバッグ コントロールを表示する

デバッグ コントロールは、すべてのユーザーに対して表示するものではありません。そのため、デバッグ コントロールの `Visible` プロパティを (PowerApps Studio 内で) 手動で切り

替えるか、所定のユーザーに対してコントロールを表示するよう、自動で切り替える必要があります。

効果的な方法の 1 つは、PowerApps for Makers コネクタを追加することです。このコネクタは "for Makers" という名称にもかかわらず、作成者以外でも読み取り専用で処理を呼び出せます。その後、GetAppRoleAssignments 関数 (英語) を呼び出して、サインインしているユーザーが現在のアプリの作成者であるかどうかを判断します。

```
Set(gloCurrentUserEmail,User().Email);

ClearCollect(Makers,
    ForAll(PowerAppsforAppMakers.GetAppRoleAssignment("Your App ID Goes Here").value,
    {Email:properties.principal.email,Role:properties.roleName})
);

Set(gloIsMaker,
    And(gloCurrentUserEmail exactin Makers.Email,
    Not(LookUp(Makers,Email=gloCurrentUserEmail).Role="CanView"))
);
```

ここでさらに、gloIsMaker のデバッグ コントロールの Visible プロパティを設定して、対象のコントロールが作成者権限を持つユーザーのみに表示されるようにします。

このアプローチの利点は、構成テーブルを使用して特別なデバッグ権限を指定する必要がないところです。

また、電子メール アドレスをチェック (User().Email のチェックなど) したり、AAD グループのメンバーシップをチェックしたりすることで、アプリ作成者または管理者のみを対象にデバッグ コントロールの表示と非表示を切り替えることもできます。

文書化

コードのコメント

2018 年 6 月以降、コードにコメントを追加できるようになりました。アプリのコードを記述する際は、コメントを詳しく書き込むようにしましょう。コメントは何か月も経ってからアプリを見直す場合に役立つほか、そのアプリを次に担当する開発者のためにもなります。

コメントには次の 2 種類があります。

- **行コメント**: コード行の最初に二重のスラッシュ (//) が入力されている場合、PowerApps は以降の行 (// を含む) をコメントと見なします。行コメントでは、次に起こる処理の内容を説明します。また、行コメントを使用して、コード行を削除する代わりに一時的に無効化する (そうすることでテストに役立てる) ことも可能です。
- **ブロック コメント**: /* と */ で囲まれたテキストはすべて、コメントとして扱われます。行コメントが一行だけのコメントであるのに対し、ブロック コメントは複数の行にわたっていてもかまいません。ブロック コメントはコメントが複数行 (コード モジュールのヘッダーなど) になる場合に便利です。また、ブロック コメントを使用して、テスト中やデバッグ中に複数の行を一時的に無効化することもできます。

コード ブロックの前にコメントを記入する場合は特に、**テキストの書式設定**機能を使用した**後で**コメントを追加することをお勧めします。**テキストの書式設定**機能は、既存のコメントに対して次のロジックを適用します。

1. プロパティのコードがブロック コメントで始まっている場合、後続のコード行はブロック コメントに付加されます。
2. プロパティのコードが行コメントで始まっていると、後続のコード行が行コメントに付加されず、後続のコード行はコメントアウトされてしまいます。
3. プロパティ内のそれ以外の場所にある行コメントとブロック コメントは、直前のコード行に付加されます。

コメントの数の多さやテキストの長さを気にする必要はありません。PowerApps がクライアント アプリのパッケージを作成する時点で、コメントはすべて取り除かれます。そのため、コメントはパッケージ サイズには影響を及ぼさず、アプリのダウンロードや読み込みが遅くなることもありません。

文書化画面

PowerApps アプリ内で使用されているコレクションや変数について文書化するために、専用の画面を作成しておくことをお勧めします。これらの画面は、他の画面とリンクさせないようにしてください。アプリが編集モードで開かれた場合にのみに表示されるようにします。

以下は、Microsoft PowerApps の Company Pulse サンプル テンプレートの例です。