Contents

Overview of application lifecycle management with Microsoft Power Platform

7/15/2022 • 2 minutes to read • Edit Online

The articles in this section describe how you can implement application lifecycle management (ALM) using Power Apps, Power Automate, Power Virtual Agents, and Microsoft Dataverse.

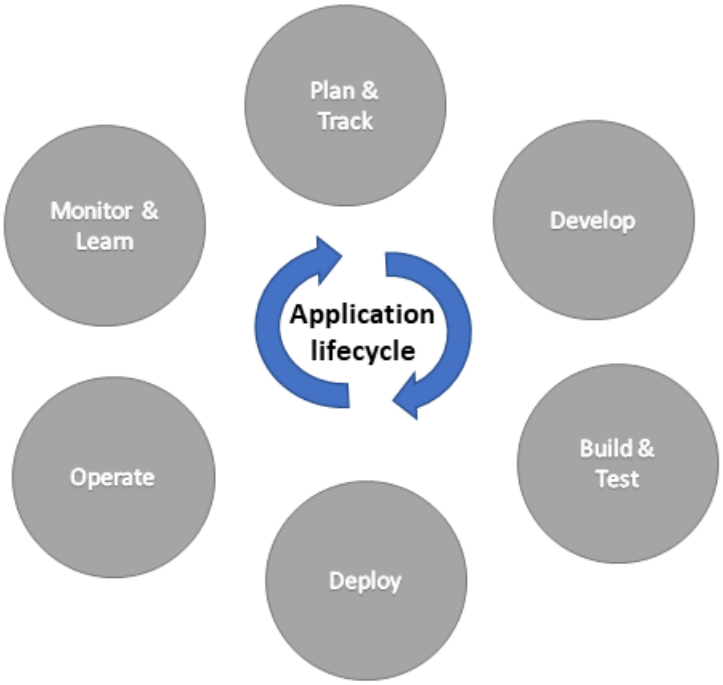
What is ALM?

ALM is the lifecycle management of applications, which includes governance, development, and maintenance. Moreover, it includes these disciplines: requirements management, software architecture, development, testing, maintenance, change management, support, continuous integration, project management, deployment, release management and governance. ALM tools provide a standardized system for communication and collaboration between software development teams and related departments, such as test and operations. These tools can also automate the process of software development and delivery. To that end, ALM combines the disciplines concerned with all aspects of the process to achieve the goal of driving efficiency through predictable and repeatable software delivery.

Key areas of ALM

1. **Governance** includes requirements management, resource management, nurturing and system administration such as data security, user access, change tracking, review, audit, deployment control, and rollback.
2. **Application development** includes identifying current problems, and planning, design, building, and testing the application and its continuous improvements. This area includes traditional developer and app maker roles.
3. **Maintenance** includes deployment of the app, and maintenance of optional and dependent technologies.

The *application lifecycle* is the cyclical software development process that involves these areas: plan and track, develop, build and test, deploy, operate, monitor, and learn from discovery.



ALM for Power Apps, Power Automate, Power Virtual Agents, and Dataverse

Dataverse in Microsoft Power Platform lets you securely store and manage data and processes that's used by business applications. To use the Power Platform features and tools available to manage ALM, all environments that participate in ALM must include a Dataverse database.

The following concepts are important for understanding ALM using the Microsoft Power Platform.

- *Solutions* are the mechanism for implementing ALM; you use them to distribute components across environments through export and import. A component represents an artifact used in your application and something that you can potentially customize. Anything that can be included in a solution is a component, such as tables, columns, canvas and model-driven apps, Power Automate flows, chatbots, charts, and plug-ins.
- *Dataverse* stores all the artifacts, including solutions.
- *Source control* should be your source of truth for storing and collaborating on your components.
- *Continuous integration and continuous delivery (CI/CD) platform* such as Azure DevOps that allows you to automate your build, test, and deployment pipeline.

For more information about how ALM and Azure DevOps technologies—combined with people and processes—enable teams to continually provide value to customers, see DevOps tools on Azure.

See also

ALM basics with Microsoft Power Platform
What is Dataverse?
Application lifecycle management for Finance and Operations apps ALM for chatbots

ALM basics with Microsoft Power Platform

7/15/2022 • 8 minutes to read • Edit Online

This article describes the components, tools, and processes needed to implement application lifecycle management (ALM).

Environments

Environments are a space to store, manage, and share your organization's business data, apps, and business processes. They also serve as containers to separate apps that might have different roles, security requirements, or target audiences. Each environment can have only one Microsoft Dataverse database.

> **IMPORTANT**
>
> When you create an environment, you can choose to install Dynamics 365 apps, such as Dynamics 365 Sales and Dynamics 365 Marketing. It is important to determine at that time if these apps are required or not because they can't be uninstalled or installed later. If you aren't building on these apps and will not require them in the future, we recommend that you not install them in your environments. This will help avoid dependency complications when you distribute solutions between environments.

Types of environments used in ALM

Using the Power Platform admin center, you can create these types of Dataverse environments:

- **Sandbox** A sandbox environment is any non-production environment of Dataverse. Isolated from production, a sandbox environment is the place to safely develop and test application changes with low risk. Sandbox environments include capabilities that would be harmful in a production environment, such as reset, delete, and copy operations. More information: Manage sandbox environments
- **Production** The environment where apps and other software are put into operation for their intended use.
- **Developer** (formally called Community). The Power Apps Developer Plan gives you access to Power Apps premium functionality, Dataverse, and Power Automate for individual use. This plan is primarily meant to build and test with Power Apps, Power Automate, and Microsoft Dataverse or for learning purposes. A developer environment is a single-user environment, and can't be used to run or share production apps.
- **Default** A single default environment is automatically created for each tenant and shared by all users in that tenant. The tenant identifies the customer, which can have one or more Microsoft subscriptions and services associated with it. Whenever a new user signs up for Power Apps, they're automatically added to the Maker role of the default environment. The default environment is created in the closest region to the default region of the Azure Active Directory (Azure AD) tenant and is named: "{Azure AD tenant name} (default)"

Create and use the correct environment for a specific purpose, such as development, test, or production.

For more information on environments, see Environments overview.

Who should have access?

Define and manage the security of your resources and data in Microsoft Dataverse. Microsoft Power Platform provides environment-level admin roles to perform tasks. Dataverse includes security roles that define the level

of access to apps, app components, and resources app makers and users have within Dataverse.

ENVIRONMENT PURPOSE	ROLES THAT HAVE ACCESS	COMMENTS
Development	App makers and developers.	App users shouldn't have access. Developers require at least the Environment Maker security role to create resources.
Test	Admins and people who are testing.	App makers, developers, and production app users shouldn't have access. Test users should have just enough privileges to perform testing.
Production	Admins and app users. Users should have just enough access to perform their tasks for the apps they use.	App makers and developers shouldn't have access, or should only have user-level privileges.
Default	By default, every user in your tenant can create and edit apps in a Dataverse default environment that has a database.	We strongly recommend that you create environments for a specific purpose, and grant the appropriate roles and privileges only to those people who need them.

More information:

- Environments overview
- Control user access to environments: security groups and licenses
- Create users and assign security roles
- Create environments

Solutions

Solutions are used to transport apps and components from one environment to another, or to apply a set of customizations to existing apps.

Solutions have these features:

- They include metadata and certain entities with configuration data. Solutions don't contain any business data.
- They can contain many different Microsoft Power Platform components, such as model-driven apps, canvas apps, site maps, flows, entities, forms, custom connectors, web resources, option sets, charts, and fields. Notice that not all entities can be included in a solution. For example, the Application User, Custom API, and Organization Setting system tables can't be added to a solution.
- They're packaged as a unit to be exported and imported to other environments, or deconstructed and checked into source control as source code for assets. Solutions are also used to apply changes to existing solutions.
- Managed solutions are used to deploy to any environment that isn't a development environment for that solution. This includes test, user acceptance testing (UAT), system integration testing (SIT), and production environments. Managed solutions can be serviced (upgrade, patch, and delete) independently from other managed solutions in an environment. As an ALM best practice, managed solutions should be generated by a build server and considered a build artifact.
- Updates to a managed solution are deployed to the previous version of the managed solution. This doesn't create an additional solution layer. You can't delete components by using an update.

- A patch contains only the changes for a parent managed solution. You should only use patches when making small updates (similar to a hotfix) and you require it to possibly be uninstalled. When patches are imported, they're layered on top of the parent solution. You can't delete components by using a patch.
- Upgrading a solution installs a new solution layer immediately above the base layer and any existing patches.
  - Applying solution upgrades involves deleting all existing patches and the base layer.
  - Solution upgrades will delete components that existed but are no longer included in the upgraded version.

More information: Solution concepts

Source control

Source control, also known as version control, is a system that maintains and securely stores software development assets and tracks changes to those assets. Change tracking is especially important when multiple app makers and developers are working on the same set of files. A source control system also gives you the ability to roll back changes or restore deleted files.

A source control system helps organizations achieve healthy ALM because the assets maintained in the source control system are the "single source of truth"—or, in other words, the single point of access and modification for your solutions.

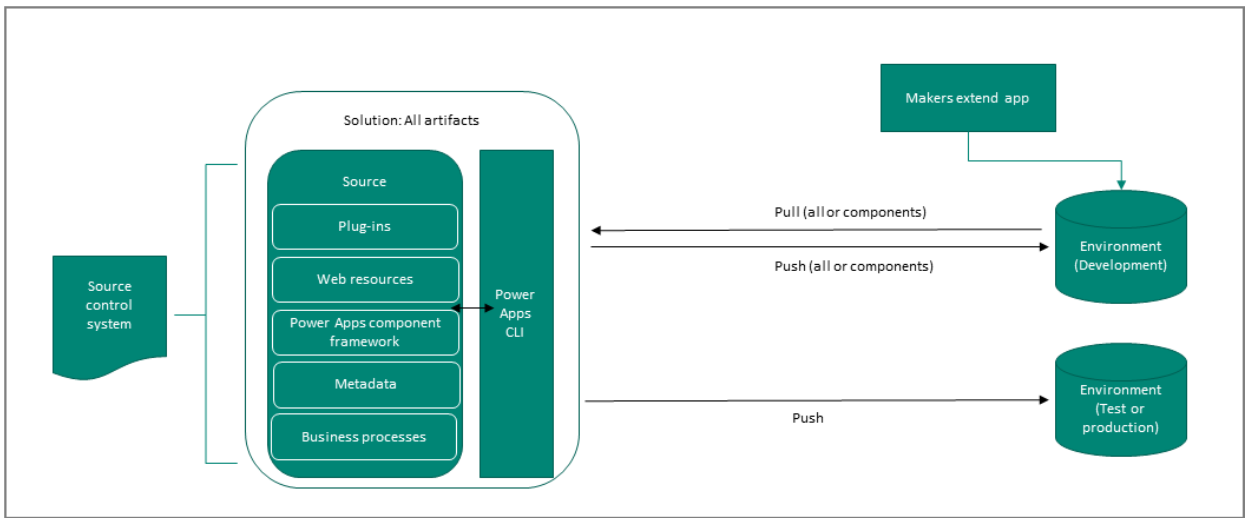
Branching and merging strategy

Nearly every source control system has some form of branching and merging support. Branching means you diverge from the main line of development and continue to do work without changing the main line. The process of merging consists of combining one branch into another, such as from a development branch into a main line branch. Some common branching strategies are trunk-based branching, release branching, and feature branching. More information: Adopt a Git branching strategy

Source control process using a solution

There are two main paths you can use when working with solutions in a source control system:

- Export the unmanaged solution and place it as unpacked in the source control system. The build process imports the packed solution as unmanaged into a temporary build environment (sandbox environment). Then, export the solution as managed and store it as a build artifact in your source control system.
- Export the solution as unmanaged and also export the solution as managed, and place both in the source control system. Although this method doesn't require a build environment, it does require maintaining two copies of all components (one copy of all unmanaged components from the unmanaged solution and one copy of all managed components from the managed solution).



More information: Build tool tasks

Automation

Automation is a key part of the application lifecycle that improves the productivity, reliability, quality, and efficiency of ALM. Automation tools and tasks are used to validate, export, pack, unpack, and export solutions in addition to creating and resetting sandbox environments.

More information: What are Microsoft Power Platform Build Tools?

Team development using shared source control

It's important to consider how you and your development team will work together to build the project. Breaking down silos and fostering views and conversations can enable your team to deliver better software. Some tools and workflows—such as those provided in Git, GitHub, and Azure DevOps—were designed for the express purpose of improving communication and software quality. Note that working with configurations in a solution system can create challenges for team development. Organizations must orchestrate changes from multiple developers to avoid merge conflicts as much as possible, because source control systems have limitations on how merges occur. We recommend that you avoid situations where multiple people make changes to complex components—such as forms, flows, and canvas apps—at the same time.

More information: Scenario 5: Supporting team development

Continuous integration and deployment

You can use any source control system and build a pipeline to start with for continuous integration and continuous deployment (CI/CD). However, this guide focuses on GitHub and Azure DevOps. GitHub is a development platform used by millions of developers. Azure DevOps provides developer services to support teams to plan work, collaborate on code development, and build and deploy applications.

To get started, you need the following:

- A GitHub account, where you can create a repository. If you don't have one, you can create one for free.
- An Azure DevOps organization. If you don't have one, you can create one for free.

More information: Create your first pipeline

Licensing

To create or edit apps and flows by using Power Apps and Power Automate, respectively, users will be required to have a per-user license for Power Apps or Power Automate or an appropriate Dynamics 365 application

license. For more information, see Licensing overview for Microsoft Power Platform. We also recommend contacting your Microsoft account representative to discuss your licensing needs.

ALM considerations

When you consider ALM as an integral part of building apps on Microsoft Power Platform, it can drastically improve speed, reliability, and user experience of the app. It also ensures that multiple developers, both traditional developers writing code and citizen developers, can jointly contribute to the application being built.

See the following articles that discuss several items to consider at the outset of any application development:

- ALM environment strategy
- Solution concepts

Environment strategy for ALM

7/15/2022 • 2 minutes to read • Edit Online

To follow application lifecycle management (ALM) principles, you'll need separate environments for app development and production. Although you can perform basic ALM with only separate development and production environments, we recommend that you also maintain at least one test environment that's separate from your development and production environments. When you have a separate test environment, you can perform end-to-end validation that includes solution deployment and application testing. Some organizations might also need more environments for user acceptance testing (UAT), systems integration testing (SIT), and training.

Separate development environments can be helpful to help isolate changes from one work effort being checked in before it's completed. Separate development environments can also be helpful to reduce situations when one person negatively affects another while making changes.

Every organization is unique, so carefully consider what your organization's environment needs are.

Development environments

You should answer questions such as:

- How many development environments do I need?
  - More information: Environments overview
- How can I automatically provision environments from source code?
  - More information: Microsoft Power Platform Build Tools for Azure DevOps
- What are the dependencies on my environments?
  - More information: Multiple solution layering and dependencies

Other environments

You should also answer the question, "Which types of non-development environments do I need?"

For example, in addition to your production environment, you might need separate test, UAT, SIT, and pre-production environments. Notice that, at a minimum, any healthy ALM practice should include using a test environment prior to deploying anything to the production environment. This ensures that you have a place to test your app, but also ensures that the deployment itself can be tested.

More information: Establishing an environment strategy for Microsoft Power Platform

Multi-geographical considerations

Power Platform environments follow a specific service update schedule as environments are updated across the world. There are six stations in total that are primarily defined by geographical location. Service updates are applied in sequence for each station. So, station 2 service updates are applied before station 3. Therefore, it's common for environments that are in different stations to have different versions at a certain point in time. For more information about the environment service update schedule, go to Released versions of Microsoft Dataverse

Solution import and environment version

When you have multiple environments in different regions, it's important to understand the following when you import a solution:

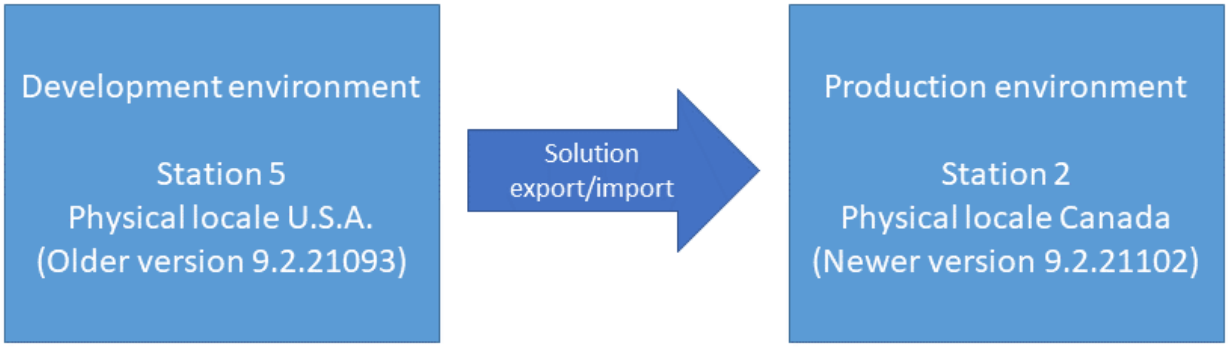
- You *can* import a solution into an environment that is a newer version than the environment where the

solution was exported.

- You *can't* reliably import a solution into an environment that's an older version than the environment where the solution was exported. This is because there might be missing components or required functionality in the older environment.

Example of successfully aligning environments with service update stations

Imagine that you have production environments in Canada and the United States. In that case, your development environments should be in North America (station 5) and not in Canada (station 2). Then, your development environments will always be the same or an earlier version than your production environments, which will curtail solution import version conflicts.



See also

Solution concepts

Solution concepts

7/15/2022 • 6 minutes to read • Edit Online

Solutions are the mechanism for implementing ALM in Power Apps and Power Automate. This article describes the following key solution concepts:

- Two types of solutions
- Solution components
- Lifecycle of a solution
- Solution publisher
- Solution and solution component dependencies

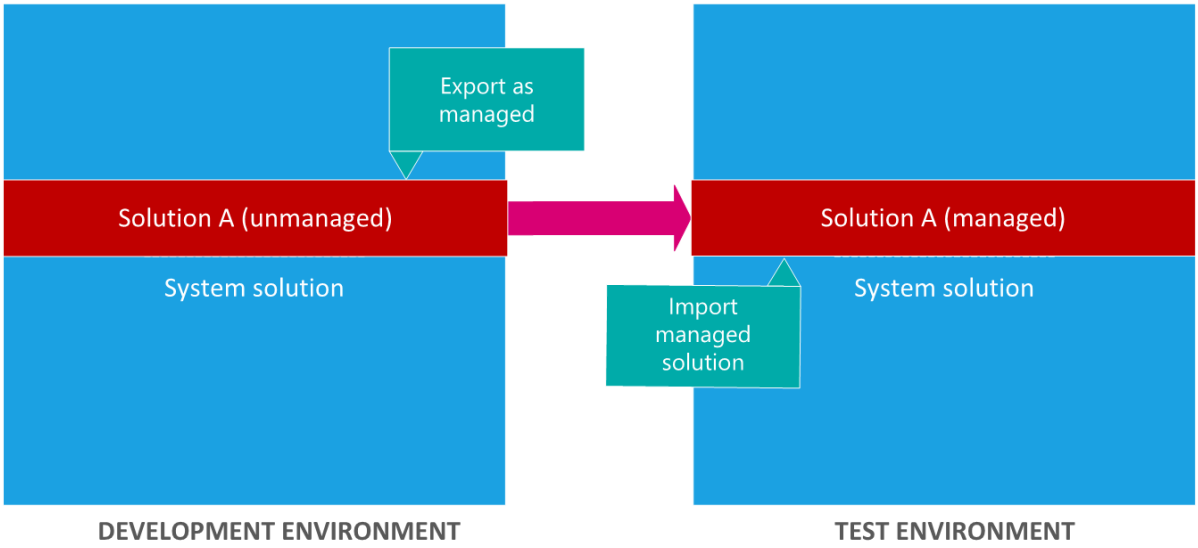
Managed and unmanaged solutions

A solution is either *managed* or *unmanaged*.

- **Unmanaged solutions** are used in development environments while you make changes to your application. Unmanaged solutions can be exported either as unmanaged or managed. Exported unmanaged versions of your solutions should be checked into your source control system. Unmanaged solutions should be considered your source for Microsoft Power Platform assets. When an unmanaged solution is deleted, only the solution container of any customizations included in it is deleted. All the unmanaged customizations remain in effect and belong to the default solution.
- **Managed solutions** are used to deploy to any environment that isn't a development environment for that solution. This includes test, UAT, SIT, and production environments. Managed solutions can be serviced independently from other managed solutions in an environment. As an ALM best practice, managed solutions should be generated by exporting an unmanaged solution as managed and considered a build artifact. Additionally:
  - You can't edit components directly within a managed solution. To edit managed components, first add them to an unmanaged solution.
    - When you do this, you create a dependency between your unmanaged customizations and the managed solution. When a dependency exists, the managed solution can't be uninstalled until you remove the dependency.
  - Some managed components can’t be edited. To verify whether a component can be edited, view the Managed properties.
  - You can't export a managed solution.
  - When a managed solution is deleted (uninstalled), all the customizations and extensions included with it are removed.

> **IMPORTANT**
>
> - You can't import a managed solution into the same environment that contains the originating unmanaged solution. To test a managed solution, you need a separate environment to import it into.
> - When you delete a managed solution, the following data is lost: data stored in custom entities that are part of the managed solution and data stored in custom attributes that are part of the managed solution on other entities that are not part of the managed solution.

Makers and developers work in development environments using unmanaged solutions, then import them to other downstream environments—such as test—as managed solutions.

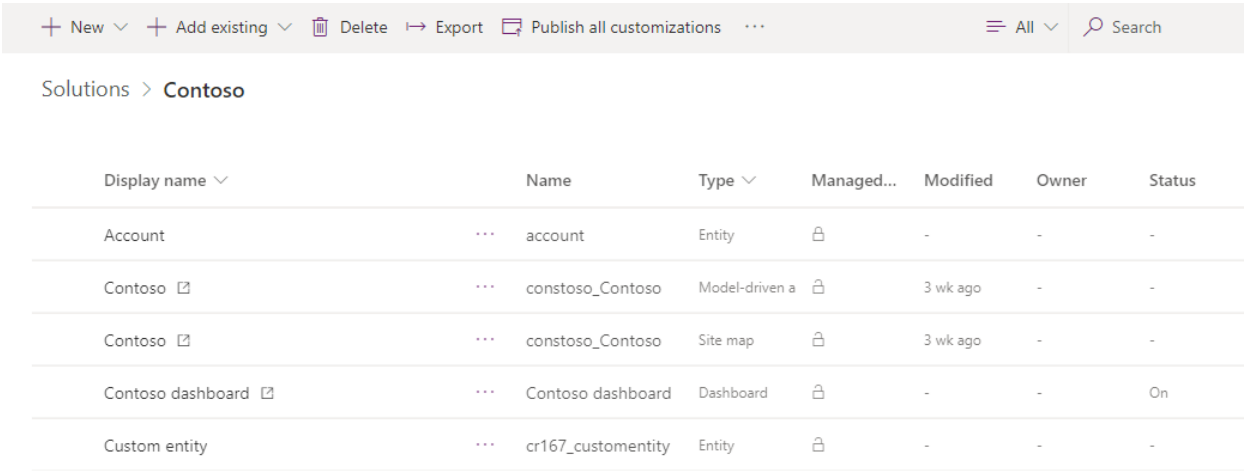


NOTE

When you customize in the development environment, you're working in the unmanaged layer. Then, when you export the unmanaged solution as a managed solution to distribute to another environment, the managed solution is imported into the environment in the managed layer. More information: Solution layers

Solution components

A component represents something that you can potentially customize. Anything that can be included in a solution is a component. To view the components included in a solution, open the solution you want. The components are listed in the **Components** list.



NOTE

- A solution can be up to 32 MB in size.
- You can't edit components directly within a managed solution.

To view a list of component types that can be added to any solution, see ComponentType Options.

Some components are nested within other components. For example, an entity contains forms, views, charts, fields, entity relationships, messages, and business rules. Each of those components requires an entity to exist. A field can't exist outside of an entity. We say that the field is dependent on the entity. There are actually twice as many types of components as shown in the preceding list, but most of them are not nested within other components and not visible in the application.

The purpose of having components is to keep track of any limitations on what can be customized using managed properties and all the dependencies so that it can be exported, imported, and (in managed solutions) deleted without leaving anything behind.

Solution lifecycle

Solutions support the following actions that help support application lifecycle processes:

- **Create** Author and export unmanaged solutions.
- **Update** Create updates to a managed solution that are deployed to the parent managed solution. You can't delete components with an update.
- **Upgrade** Import the solution as an upgrade to an existing managed solution, which removes unused components and implements upgrade logic. Upgrades involve rolling up (merging) all patches to the solution into a new version of the solution. Solution upgrades will delete components that existed but are no longer included in the upgraded version. You can choose to upgrade immediately or to stage the upgrade so that you can do some additional actions prior to completing the upgrade.
- **Patch** A patch contains only the changes for a parent managed solution, such as adding or editing components and assets. Use patches when making small updates (similar to a hotfix). When patches are imported, they're layered on top of the parent solution. You can't delete components with a patch.

Solution publisher

Every app and other solution components such as entities you create or any customization you make is part of a solution. Because every solution has a publisher, you should create your own publisher rather than use the default. You specify the publisher when you create a solution.

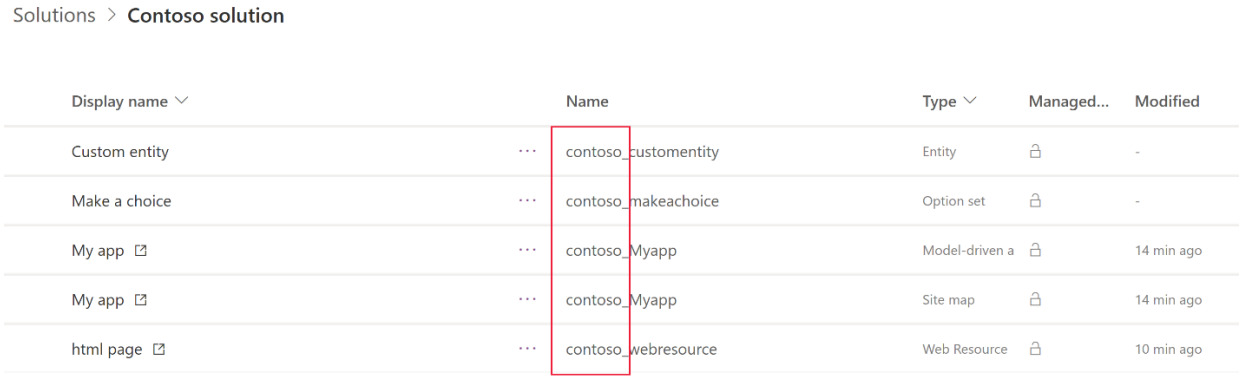
> **NOTE**
>
> Even if you don't use a custom solution, you'll be working in solutions which are known as the *Common Data Service Default Solution* and the *Default* solutions. More information: Default Solution and Common Data Service Default Solution

The publisher of a solution where a component is created is considered the owner of that component. The owner of a component controls what changes other publishers of solutions including that component are allowed to make or restricted from making. It is possible to move the ownership of a component from one solution to another within the same publisher, but not across publishers. Once you introduce a publisher for a component in a managed solution, you can't change the publisher for the component. Because of this, it's best to define a single publisher so you can change the layering model across solutions later.

The solution publisher specifies who developed the app. For this reason, you should create a solution publisher name that's meaningful.

Solution publisher prefix

A solution publisher includes a prefix. The publisher prefix is a mechanism to help avoid naming collisions. This allows for solutions from different publishers to be installed in an environment with few conflicts. For example, the Contoso solution displayed here includes a solution publisher prefix of *contoso*.



> **NOTE**
>
> When you change a solution publisher prefix, you should do it before you create any new apps or metadata items because you can't change the names of metadata items after they're created.

More information:

- Create a solution publisher prefix
- Change a solution publisher prefix

Solution dependencies

Because of the way that managed solutions are layered, some managed solutions can be dependent on solution components in other managed solutions. Some solution publishers will take advantage of this to build solutions that are modular. You may need to install a "base" managed solution first and then you can install a second managed solution that will further customize the components in the base managed solution. The second managed solution depends on solution components that are part of the first solution.

The system tracks these dependencies between solutions. If you try to install a solution that requires a base solution that isn't installed, you won't be able to install the solution. You will get a message saying that the solution requires another solution to be installed first. Similarly, because of the dependencies, you can't uninstall the base solution while a solution that depends on it is still installed. You have to uninstall the dependent solution before you can uninstall the base solution. More information: Removing dependencies

Solution component dependencies

A solution component represents something that you can potentially customize. Anything that can be included in a solution is a solution component and some components are dependant on other components. For example, the website field and account summary report are both dependant on the account entity. More information: Dependency tracking for solution components

See also

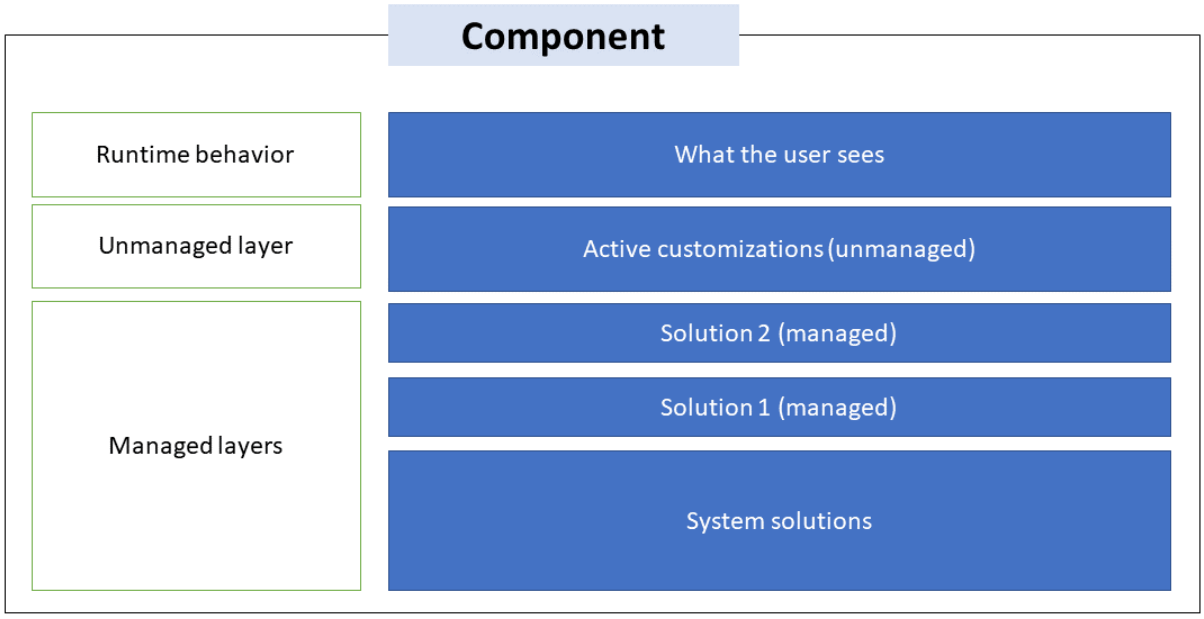
Solution layers
Create and manage environments in the Power Platform admin center

Solution layers

7/15/2022 • 3 minutes to read • Edit Online

Solution layering is implemented at a component level. Managed and unmanaged solutions exist at different layers within a Microsoft Dataverse environment. In Dataverse, there are two distinct layers:

- **Unmanaged layer** All imported unmanaged solutions and ad-hoc customizations exist at this layer. All unmanaged solutions share a single unmanaged layer.
- **Managed layers** All imported, managed solutions and the system solution exist at this level. When multiple managed solutions are installed, the last one installed is above the managed solution installed previously. This means that the second solution installed can customize the one installed before it. When two managed solutions have conflicting definitions, the runtime behavior is either "Last one wins" or a merge logic is implemented. If you uninstall a managed solution, the managed solution below it takes effect. If you uninstall all managed solutions, the default behavior defined within the system solution is applied. At the base of the managed layers level is the system layer. The system layer contains the entities and components that are required for the platform to function.

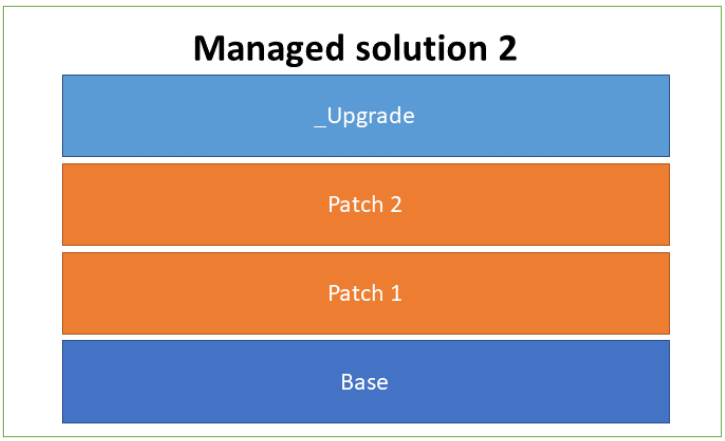


Layering within a managed solution

For each managed component, there are layers within a solution, which—depending on whether one or more patches or a pending upgrade to the solution has been imported—can include the following layers:

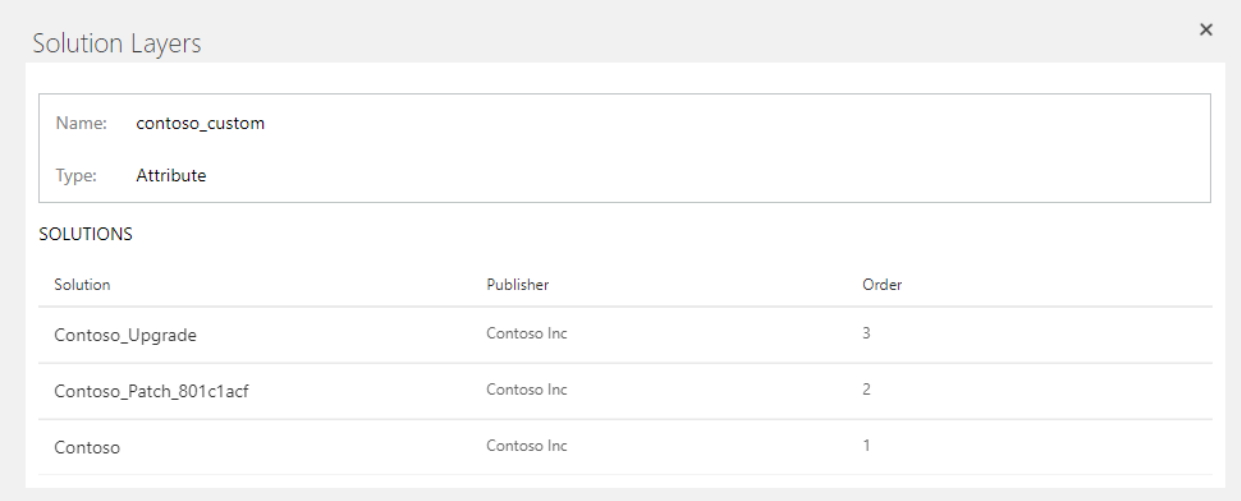
- **Base** Located at the bottom of the solution layer "stack" is the base layer. This layer includes the solution publisher, which identifies the owner of the component and the managed properties associated with it.
- **Top** The top layer is considered the current layer and defines the runtime behavior of the component. The top layer can be an upgrade or a patch, or if no patches or upgrades have been applied to the solution, the base solution determines the component runtime behavior.
- Layers added from updates:
  - **Patches** If the component has one or more solution patches imported, they're stacked on top of the base layer, with the most recent patch residing above the previous patch.

- **Pending upgrade** If a staged upgrade (named _Upgrade) is imported, it resides on top of the base and patch (if any) layers.



> **IMPORTANT**
> Using patches isn't recommended. More information: Create solution patches

The following image shows an example of solution layers for a custom column that displays the base solution, a patch, and a pending upgrade.



For information about how to view layers for a component within a solution, see Solution layers.

Merge behavior

Solution makers should understand the merge behavior when a solution is updated or when multiple solutions are installed that affect the same component. Notice that only model-driven app, form, and site map component types will be merged. All other components use "top level wins" behavior.

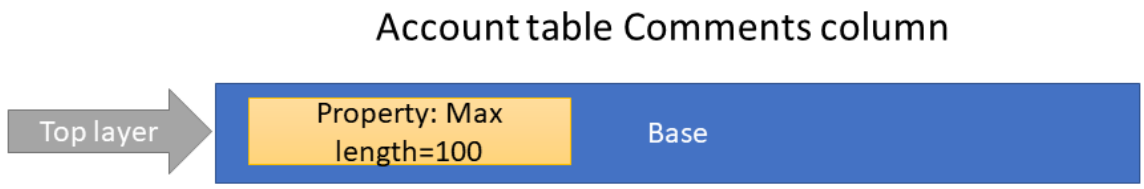
"Top wins" behavior

With the exception of the model-driven app, form, and site map components, other solution components use a "top wins" behavior where the layer that resides at the top determines how the component works at app runtime. A top layer can be introduced by a staged (pending) upgrade.

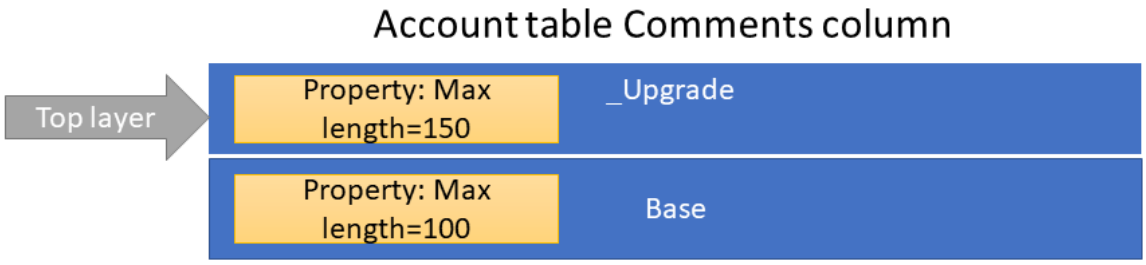
Top layer introduced by a pending upgrade

Here's an example of a top wins component behavior introduced by a stage for upgrade update to a solution. More information: Apply the upgrade or update in the target environment

1. The current top (base) layer has the **Max length** property of the **Comments** text column for the account table using the default setting of 100.



2. A solution upgrade is imported using the stage for upgrade option, which creates a new top layer. The pending upgrade includes the **Comments** text column for the account table with the **Max length** property value changed to 150.



In this situation, the **Comments** column for account records will allow up to a maximum of 150 characters during app run time.

Solution update and upgrade merge behavior

As described in the previous section, patches and a staged upgrade are stacked on top of the base solution. These can be merged by selecting **Apply upgrade** from the **Solutions** area in Power Apps, which flattens the layers and creates a new base solution.

Multiple solutions merge behavior

When you prepare your managed solution for distribution, remember that an environment might have multiple solutions installed or that other solutions might be installed in the future. Construct a solution that follows best practices so that your solution won't interfere with other solutions. More information: Use segmented solutions

The processes that Dataverse uses to merge customizations emphasize maintaining the functionality of the solution. Although every effort is made to preserve the presentation, some incompatibilities between customizations might require that the computed resolution change some presentation details in favor of maintaining the customization functionality.

See also

Understand how managed solutions are merged

Understand how managed solutions are merged

7/15/2022 • 5 minutes to read • Edit Online

When you prepare your managed solution to be installed, remember that an environment might already have multiple solutions installed or that other solutions might be installed in the future. Construct a solution that follows best practices so that your solution won't interfere with other solutions.

The processes that Microsoft Dataverse uses to merge customizations emphasize maintaining the functionality of the solution. Although every effort is made to preserve the presentation, some incompatibilities between customizations might require that the computed resolution change some presentation details in favor of maintaining the customization functionality.

Merge form customizations

The only form customizations that have to be merged are those that are performed on any entity forms that are already in the environment. Typically, this means that form customizations only have to be merged when your solution customizes the forms that were included for entities created when Dataverse was installed. One way to avoid form merging is to provide new forms for any Dataverse entities. Forms for custom entities won't require merging unless you're creating a solution that updates or modifies an existing managed solution that created the custom entities and their forms.

When a solution is packaged as a managed solution, the form definitions stored in FormXML are compared to the original FormXML and only the differences are included in the managed solution. When the managed solution is installed in a new environment, the form customization differences are then merged with the FormXML for the existing form to create a new form definition. This new form definition is what the user sees and what a system customizer can modify. When the managed solution is uninstalled, only those form elements found in the managed solution are removed.

Form merge occurs on a section-by-section basis. When you add new elements to an existing tab or section, your changes can affect or conceal the elements from the managed layers, including when the managed element is updated. This behavior occurs because the managed layers are underneath the unmanaged layer you're introducing with your customization. If you don't want to affect or conceal managed elements on the form, we recommend that you include your new elements within new container elements, such as a section or tab. This isolates your elements and reduces the possibility of affecting or concealing the elements from the managed layers. More information: Solution layers

Managed solutions that contain forms that use new security roles depend on those roles. You should include these security roles with your managed solution.

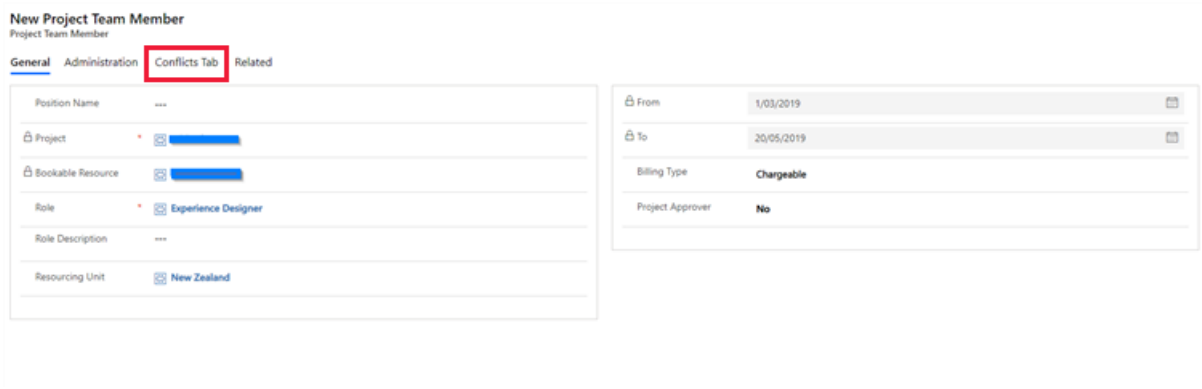
When you import a solution that includes table forms, the **Overwrite Customizations** option, even if selected, does not apply. The form being imported merges with any existing solution layers for the form.

> **NOTE**
>
> When a managed solution entity contains multiple forms and the environment entity form also contains multiple forms, the new forms aren't appended to the bottom of the list of available forms—they're interleaved with the original entity forms.

Identifying and resolving form merge conflicts

After you import a solution that includes a form, you may notice that the imported form displays a tab named

Conflicts Tab. This is an auto-generated tab, which is created when certain form components are unable to merge. To avoid any data loss, the form components that aren't able to merge are placed under the Conflicts Tab. Merge conflicts usually happen when the source and target customizations are out of sync, which leads to conflicting form customizations.



Avoid these situations that can cause form merge conflicts:

- You import two different solutions that add a component, such as a form tab, that uses the same ordinal value.
- You customize a component of the form, such as a section, in the source environment but also make the same or similar customization to the component in the target environment. Then, you export the customization from the source environment and import it into the target environment.

When the Conflicts Tab appears on an imported form, you can move the component displayed somewhere on the form. Once all the components are moved from the Conflicts Tab, you can delete or hide the Conflicts Tab.

Merge navigation (SiteMap) customizations

When a solution is packaged as managed, the SiteMap XML is compared to the original SiteMap XML and any other customizations made to the SiteMap. Only the differences are included in the managed solution. These differences include items that are changed, moved, added, or removed. When the managed solution is installed in a new environment, the SiteMap changes are merged with the SiteMap XML found for the environment where the managed solution is being installed. A new SiteMap definition is what people see.

At this point, a customizer can export the SiteMap to an unmanaged solution and that SiteMap definition will include all the elements of the active SiteMap. A customizer can then modify the SiteMap and reimport it as an unmanaged customization. Later, if the managed solution is uninstalled, the SiteMap XML that was imported with the managed solution will be referenced to remove the changes introduced with that managed solution. A new active SiteMap is then calculated.

Whenever a new visible element is added to the SiteMap, it appears at the bottom of whatever container it belongs in. For example, a new area will appear at the bottom of the navigation area. To position the elements that have been added, you must export the SiteMap, edit it to set the precise position of the elements, and then import it again as an unmanaged solution.

> **NOTE**
> Only one SiteMap customization can be applied between publishing. Any unpublished SiteMap customizations will be lost when a new SiteMap definition is imported.

Merge option set options

Each new option set option is initialized with an integer value assigned that includes an option value prefix. The option value prefix is a set of five digits prepended to the option value. An option value prefix is generated based on the solution publisher's customization prefix, but can be set to any value. The option value prefix helps differentiate new option set options created in the context of a specific solution publisher and reduces the opportunity for collisions of option values. Using the option value prefix is recommended but not required.

A managed solution usually updates or adds options for option sets that are already in the environment, for example, the Category or Industry option sets for an account. When a managed solution modifies the options available in an option set, all the options defined in the managed solution are available in the environment. When the managed solution is uninstalled, the options in the option set will be returned to their original state.

See also

Use a solution to customize

Use a solution to customize

7/15/2022 • 2 minutes to read • Edit Online

We recommend that you create a solution to manage your customizations. By using a custom solution, you can easily find just the solution components you've customized, consistently apply your solution publisher prefix, and export your solution for distribution to other environments.

If you don't use a custom solution, you'll be working in one of these default solutions in the unmanaged layer:

- **Common Data Service Default Solution**. This solution is available for makers to use by default for their customizations in an environment. The Common Data Service Default Solution is useful when you want to evaluate or learn the Power Platform. However, if you are planning on deploying your apps or flows to other environments, we recommend that makers work in their own unmanaged solutions. More information: Common Data Service Default Solution
- **Default Solution**. This is a special solution that contains all components in the system. The default solution is useful for discovering all the components and configurations in your system.

Why you shouldn't use the default solutions to manage customizations

There are a few reasons why you shouldn't create apps/flows and make customizations in either of the default solutions:

- When you use either default solution to create components, you'll also use the default publisher assigned to the solution. This often results in the wrong publisher prefix being applied to some components. More information: Solution publisher
- The default solution can't be exported; therefore, you can't distribute the default solution to another environment.
- If you aren't consistently using the same solution while you're configuring the environment, you may accidentally leave components behind that are necessary to deploy your application to another environment.

Common Data Service Default Solution

The default solution in the Power Platform is the Common Data Service Default Solution, which is associated with the Microsoft Dataverse Default Publisher. The default publisher prefix will be randomly assigned for this publisher, for example it might be *cr8a3*. This means that the name of every new item of metadata created in the default solution will have this prepended to the names used to uniquely identify the items. If you create a new entity named Animal, the unique name used by Dataverse will be cr8a3_animal. The same is true for any new fields (attributes), relationships, or option-set options. If you'll be customizing this special solution, consider changing the publisher prefix.

See also

Managed properties

Managed properties

7/15/2022 • 3 minutes to read • Edit Online

You can use managed properties to control which of your managed solution components can be customized. If you're creating solutions for other organizations, you should allow them to customize solution components where it makes sense for their unique requirements. However, you have to be able to predictably support and maintain your solution, so you should never allow any customization of critical components that provide the core functionality of your solution.

Consider using managed properties to lock down your components unless you want them customizable in the destination environment. For example, imagine a scenario where your users might want to change many of the labels to fit their own business requirements.

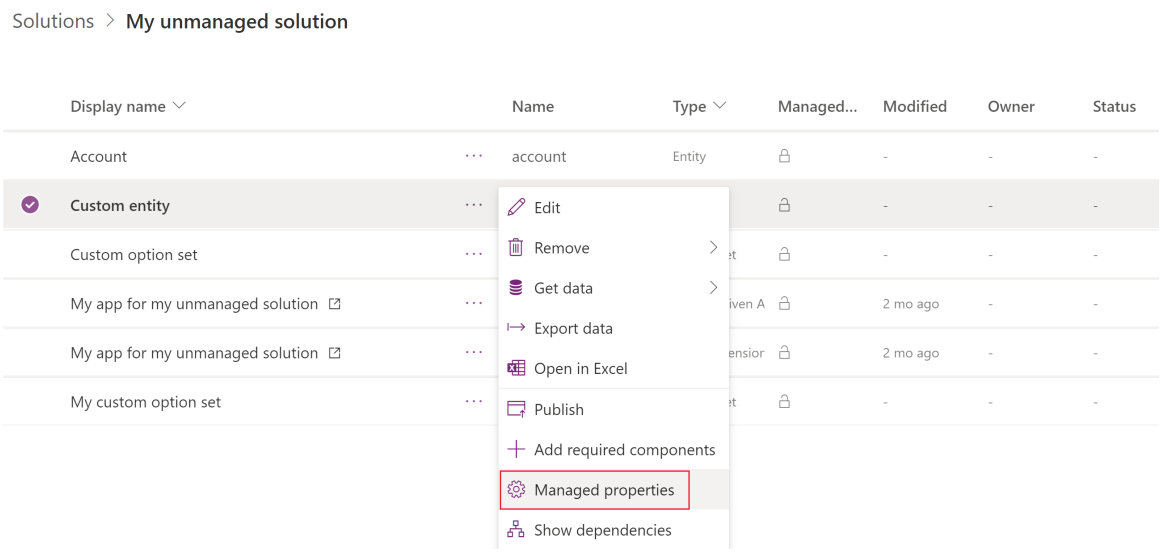
Managed properties are intended to protect your solution from modifications that might cause it to break. Managed properties don't provide digital rights management (DRM), or capabilities to license your solution or control who may import it.

You apply managed properties when the solution is unmanaged in the unmanaged layer of your development environment. The managed properties will take effect after you package the managed solution and install it in a different environment. After the managed solution is imported, the managed properties can't be updated except by an update of the solution by the original publisher.

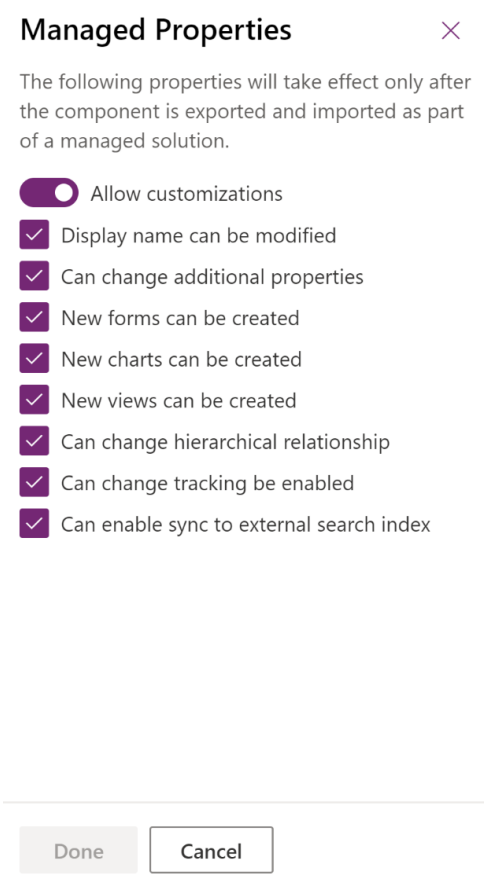
Most solution components have a **Managed properties** menu item available in the list of solution components. When you import the managed solution that contains the components, you can view—but not change—the managed properties.

View and edit entity managed properties

1. Sign in to Power Apps or Power Automate and select **Solutions** from the left pane.
2. Open the solution that you want.
3. From the list of components in the solution, select ... next to the entity that you want to view the managed properties, and then select **Managed properties**.



The managed properties page is displayed.



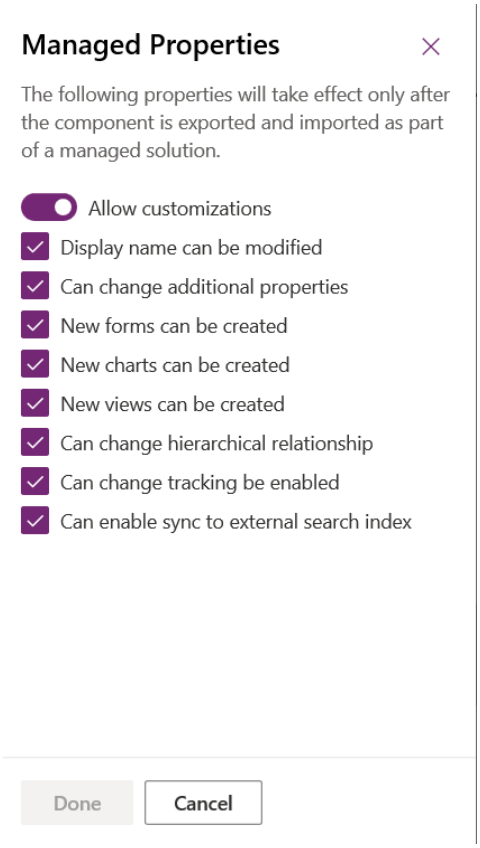
Entities have more managed properties than any other type of solution component. If the entity is customizable, you can set the following options:

OPTION	DESCRIPTION
Allow customizations	Controls all the other options. If this option is `False`, none of the other settings apply. When it is `True`, you can specify the other customization options. When `False`, it is equivalent to setting all other options to false.
Display name can be modified	Whether the entity display name can be modified.
Can Change Additional Properties	Applies to anything not covered by other options.
New forms can be created	Whether new forms can be created for the entity.
New charts can be created	Whether new charts can be created for the entity.
New views can be created	Whether new views can be created for the entity.
Can Change Hierarchical Relationship	Whether Hierarchical Relationships settings can be changed. More information: Define and query hierarchically related data
Can Change Tracking Be Enabled	Whether the entity **Change Tracking** property can be changed.
Can Enable sync to external search index	Whether the entity can be configured to enable Dataverse search. More information: Configure Dataverse search to improve search results and performance

View and edit field managed properties

Next to a custom field in a solution select ... and then select **Managed properties**.

This opens the **Managed Properties** pane.



The **Allow customizations** option controls all the other options. If this option is disabled, none of the other settings apply. When it is enabled, you can specify the other customization options.

If the field is customizable, you can enable the following options.

- **Display name can be modified**
- **Can change additional properties**: This property controls any other customizations that do not have a specific managed property.
- **New forms can be created**
- **New charts can be created**
- **New views can be created**
- **Can change hierarchical relationship**
- **Can change tracking be enabled**
- **Can enable sync to external search index**

Disabling all the individual options is equivalent to disabling **Allow customizations**.

Apply your choices and select **Done** to close the pane.

> **NOTE**
>
> If this field is a **Date and Time** field, an additional **Can change date and time behavior** property is available. More information: Behavior and format of the Date and Time field

See Update of delete a field for information about how to edit fields.

View and edit other component managed properties

You can view and edit managed properties for many other solution components, such as a web resource, process, chart, or dashboard. Next to the component in a solution select **...** and then select **Managed properties**.

View and edit relationship managed properties

While viewing entity relationships in solution explorer, select a relationship from an unmanaged solution and then choose **More Actions** > **Managed Properties** on the menu bar.

With relationships, the only managed property is **Can Be Customized**. This single setting controls all changes that can be made to the entity relationship.

See also

Use segmented solutions

Use segmented solutions

7/15/2022 • 2 minutes to read • Edit Online

Use solution segmentation so that you only include entity components that are updated when you distribute solution updates. With solution segmentation, you export solution updates with selected entity assets, such as entity fields, forms, and views, rather than entire entities with all the assets. To create a segmented solution, you use the Solutions area in Power Apps.

You can segment a solution when you select from the following options when adding an existing entity to the solution:

- **Include no components or metadata** When you don't select any components or metadata, the minimal entity information is added to the solution. Therefore, apart from the friendly name, entity attributes (metadata) or components won't be included.
- **Select components** You can segment your solution by individually selecting each component that's associated with the entity, such as fields, relationships, business rules, views, forms, and charts. Use this option to select only the components that you've added or changed with the entity, such as a new custom field or adding a form.
- **Include entity metadata** This option includes no components—such as fields, forms, views, or related entities—but does include all the metadata associated with the entity. Metadata includes the entity properties, such as auditing, duplicate detection, and change tracking.
- **Include all components** This option includes all components and metadata associated with the entity. It can include other entities or entity components such as business process flows, reports, connections, and queues. You should only use this option when you're distributing an unmanaged entity that doesn't exist in the target environment. Notice that after you select this option, you can't undo it. To segment the solution, you must remove the entity from the solution and re-add it by selecting only the changed or new components.

> **WARNING**
> Don't add components to the solution that aren't new or changed components. When your update is imported to the target environment, a solution with unintended components can cause unexpected behavior to the existing components that now lay below the layer you introduced with your solution update. For example, if you add a view for an entity that isn't updated and the view in the existing layer has customizations, the existing customizations might become inactive. More information: Solution layers

More information: Create segmented solutions

See also

Update a solution

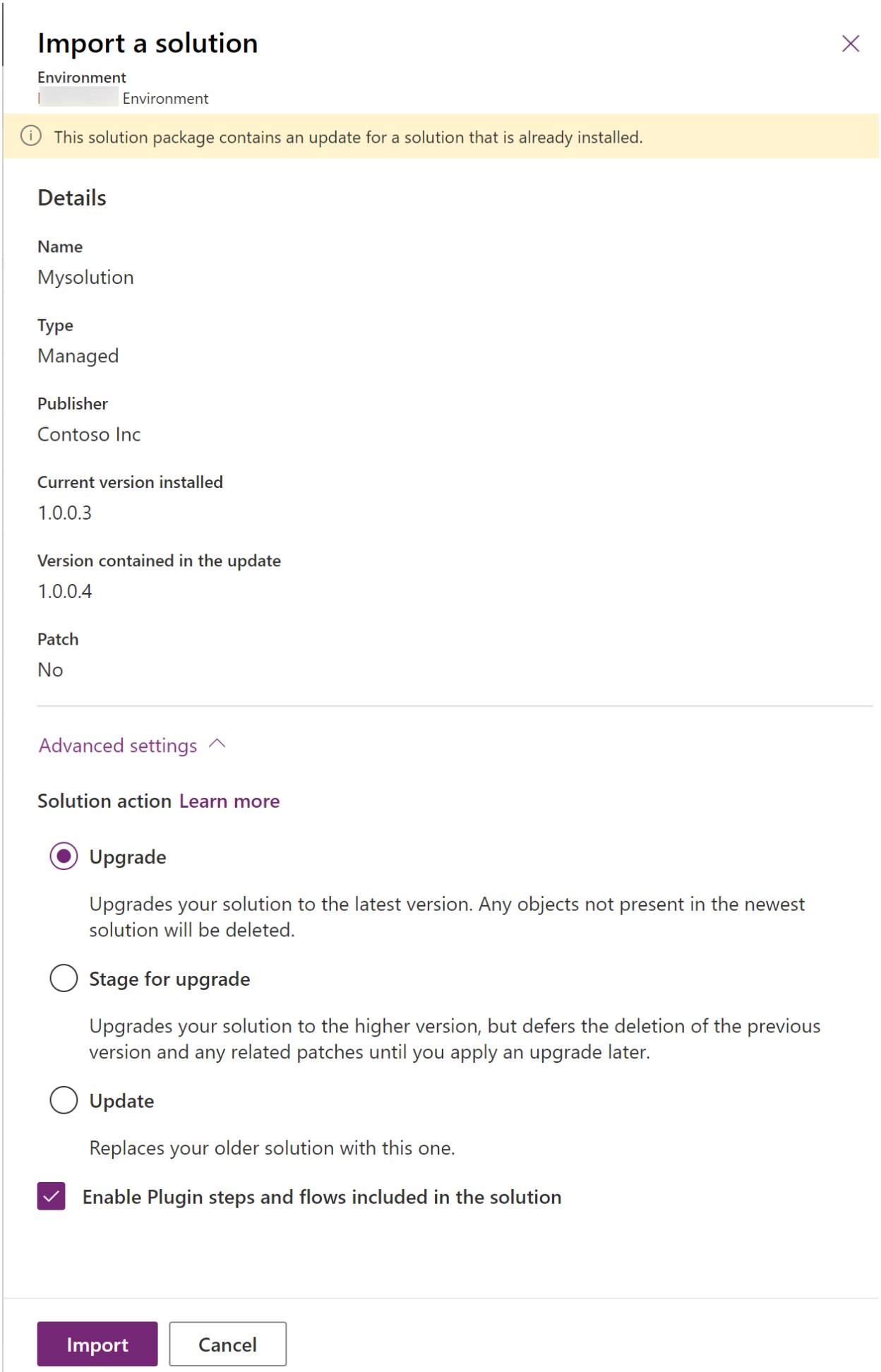
Create and update solutions

7/15/2022 • 5 minutes to read • Edit Online

To locate and work with just the components you've customized, create a solution and do all your customization there. Then, always remember to work in the context of the custom solution as you add, edit, and create components. This makes it easy to export your solution for import to another environment or as a backup. More information Create a solution

Update a solution

Make changes to your unmanaged solution, such as adding or removing components. Then, when you import a managed solution that was previously imported, the import logic detects the solution as an update and displays the following screen of options.



More information: Apply an update or upgrade for a solution

Create solution patches

You can create a patch for a parent solution and export it as a minor update to the base solution. When you clone a solution, the system rolls up all related patches into the base solution and creates a new version.

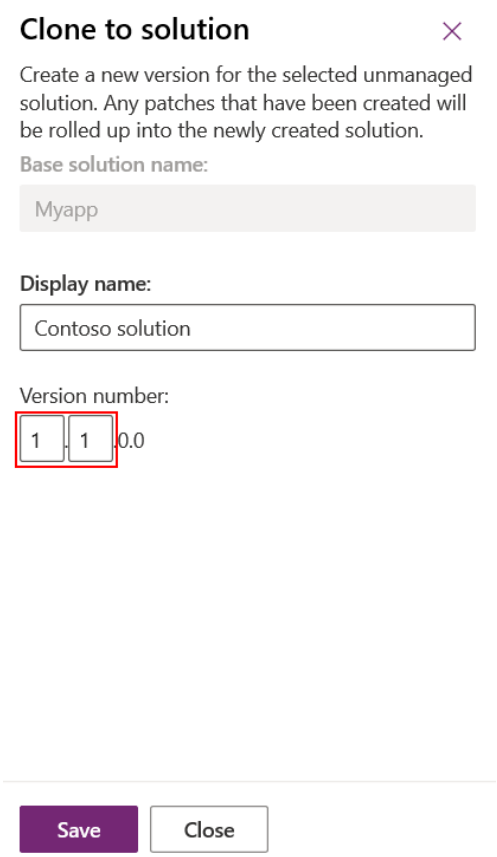
> **WARNING**
> Using clone a patch and clone solution to update a solution isn't recommended because it limits team development and increases complexity when storing your solution in a source control system. For information about how to update a solution, see Update a solution.

Creating updates using clone solution and clone to patch

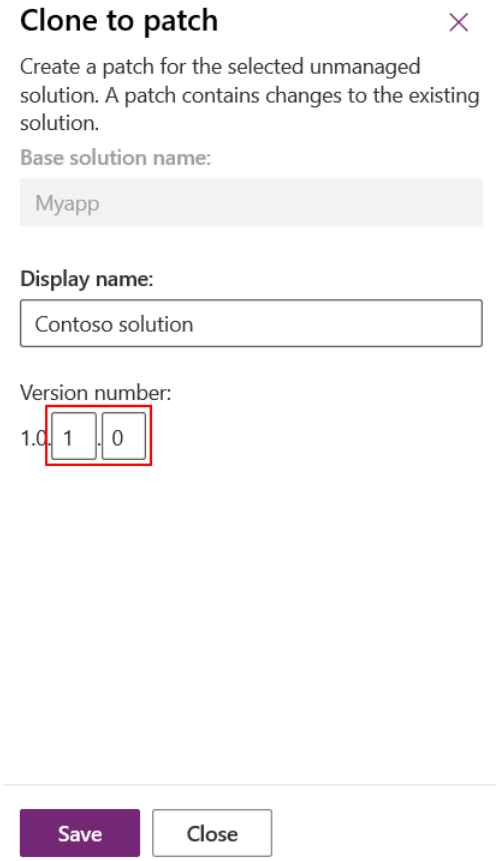
When you're working with patches and cloned solutions, keep the following information in mind:

- A patch represents an incremental minor update to the parent solution. A patch can add or update components and assets in the parent solution when installed on the target system, but it can't delete any components or assets from the parent solution.
- A patch can have only one parent solution, but a parent solution can have one or more patches.
- A patch is created from an unmanaged solution. You can't create a patch from a managed solution.
- When you import a patch into a target system, you should export it as a managed patch. Don't use unmanaged patches in production environments.
- The parent solution must be present in the target system to install a patch.
- You can delete or update a patch.
- If you delete a parent solution, all child patches are also deleted. The system gives you a warning message that you can't undo the delete operation. The deletion is performed in a single transaction. If one of the patches or the parent solution fails to delete, the entire transaction is rolled back.
- After you have created the first patch for a parent solution, the solution becomes locked, and you can't make any changes in this solution or export it. However, if you delete all of its child patches, the parent solution becomes unlocked.
- When you clone a base solution, all child patches are rolled up into the base solution and it becomes a new version. You can add, edit, or delete components and assets in the cloned solution.
- A cloned solution represents a replacement of the base solution when it's installed on the target system as a managed solution. Typically, you use a cloned solution to ship a major update to the preceding solution.

When you clone a solution, the version number you specify includes the major and minor positions.



When you clone a patch, the version number you specify includes the build and revision positions.



For more information about version numbers, see Clone solution and clone patch version numbers in this article.

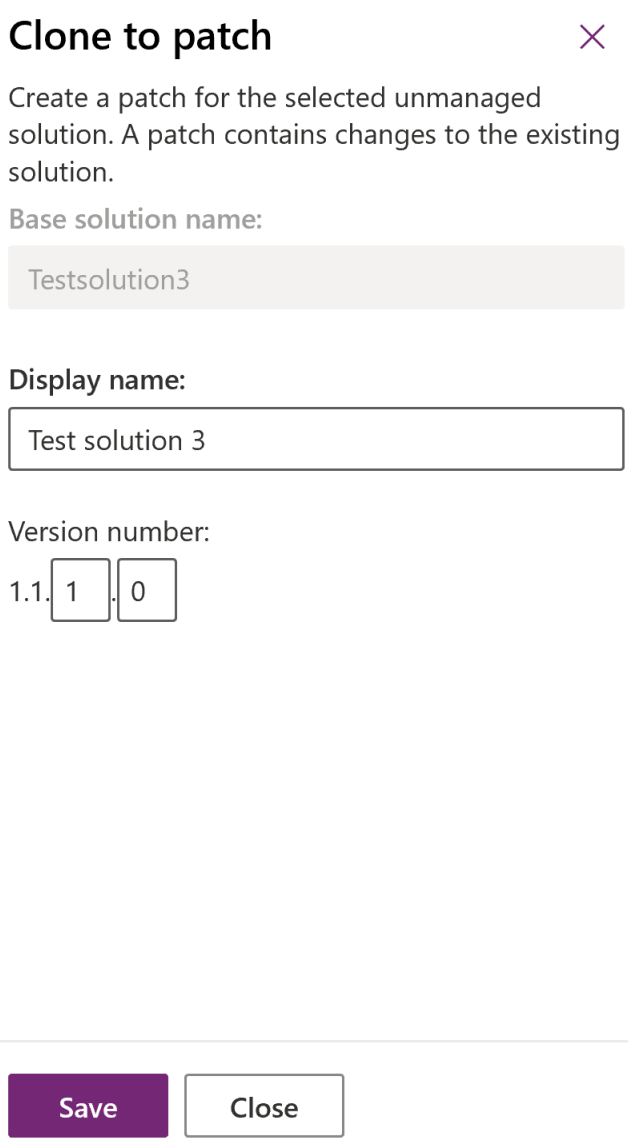
Create a solution patch

A patch contains changes to the parent solution, such as adding or editing components and assets. You don't

have to include the parent's components unless you plan to edit them.

Create a patch for an unmanaged solution

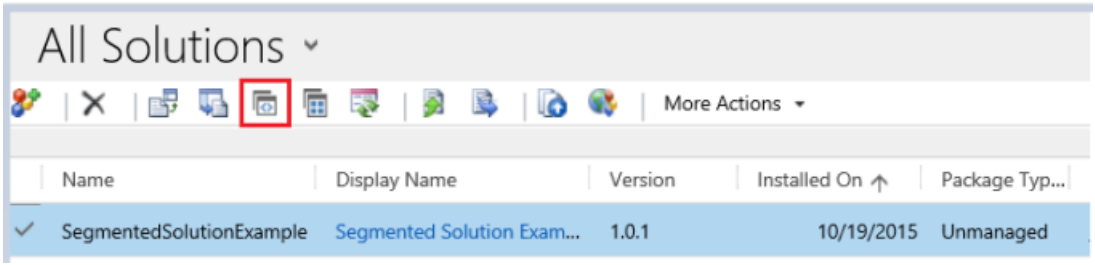
1. Go to the Power Apps portal, and then select **Solutions**.
2. In the solutions list, select an unmanaged solution to create a patch for. On the command bar, select **Clone**, and then select **Clone a Patch**. The right pane that opens contains the base solution's name and the patch version number. Select **Save**.



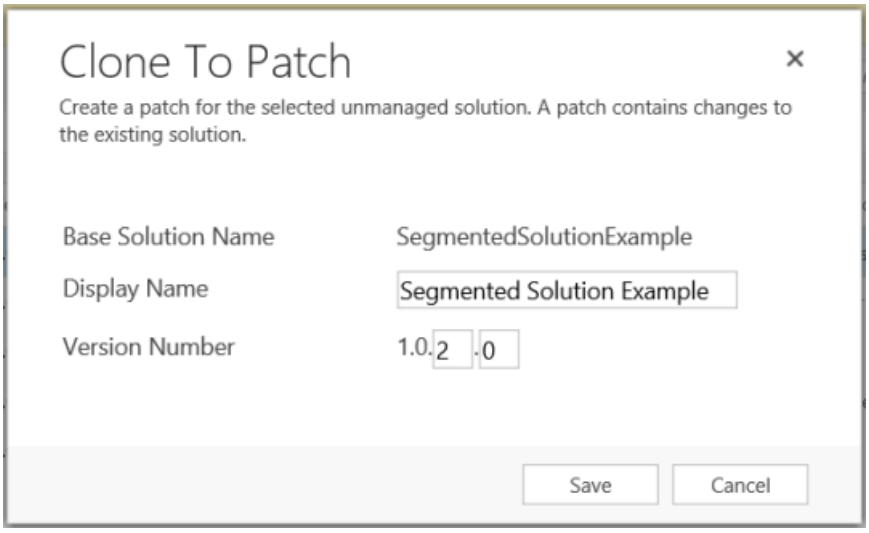
3. In the solutions list, find and open the newly created patch. Notice that the unique name of the solution has been appended with *Patchhexnumber*. Just like with the base solution, add the components and assets you want.

Create a patch using solution explorer

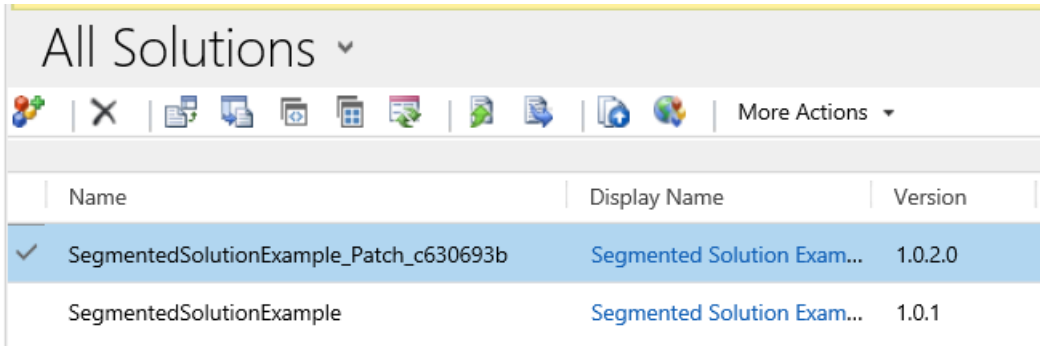
The following illustrations provide an example of creating a patch for an existing solution. Start by selecting **Clone a Patch** (in the compressed view, the **Clone a Patch** icon is depicted as two small squares, as shown below).



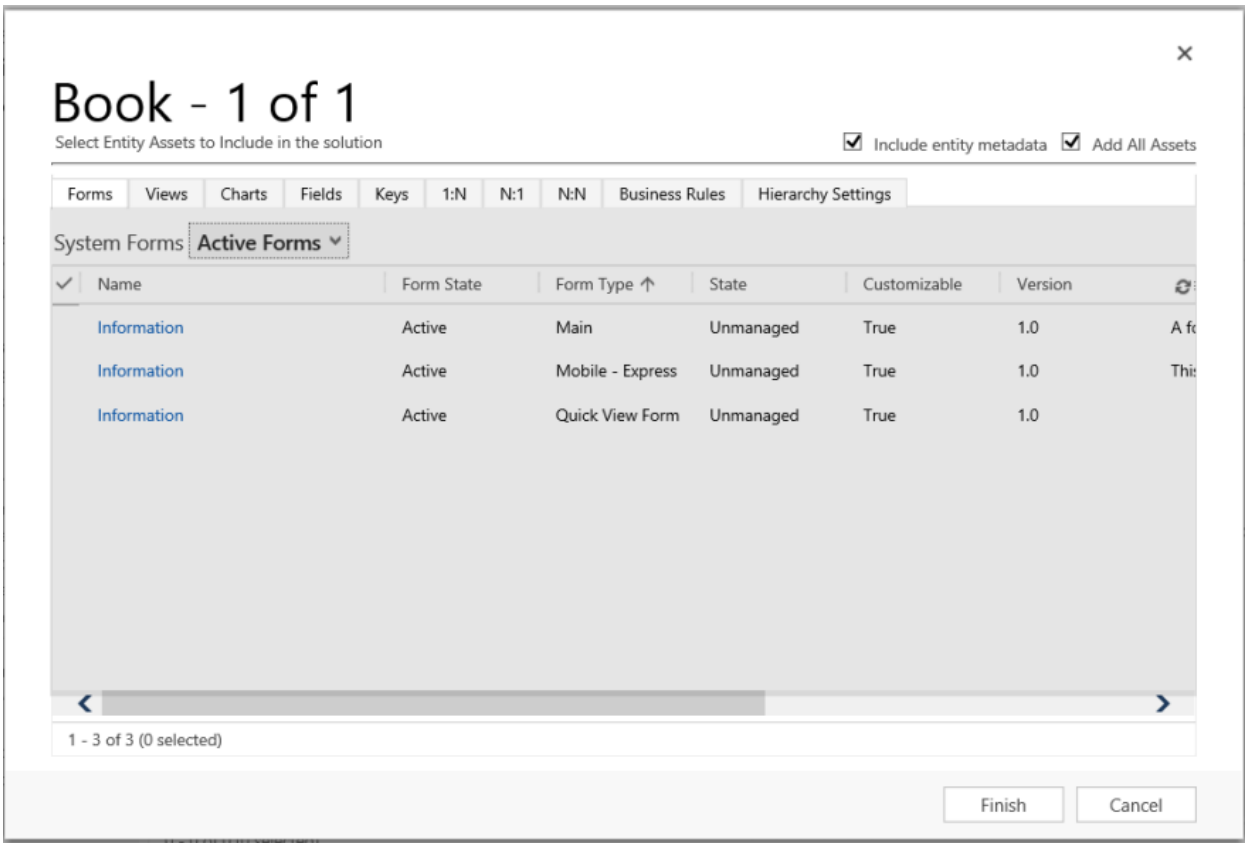
In the **Clone To Patch** dialog box you see that the version number for the patch is based on the parent solution version number, but the build number is incremented by one. Each subsequent patch has a higher build or revision number than the preceding patch.



The following screenshot shows the base solution **SegmentedSolutionExample**, version **1.0.1.0**, and the patch **SegmentedSolutionExample_Patch**, version **1.0.2.0**.



In the patch, we added a new custom entity called `Book`, and included all assets of the `Book` entity in the patch.



Clone a solution

When you clone an unmanaged solution, the original solution and all patches related to the solution are rolled up into a newly created version of the original solution. After cloning, the new solution version contains the original entities plus any components or entities that are added in a patch.

> **IMPORTANT**
> Cloning a solution merges the original solution and associated patches into a new base solution and removes the original solution and patches.

1. Go to the Power Apps portal, and then select **Solutions**.
2. In the solutions list, select an unmanaged solution to create a clone. On the command bar, select **Clone**, and then select **Clone Solution**. The right pane displays the base solution's name and the new version number. Select **Save**.

Clone solution and clone patch version numbers

A patch must have a higher build or revision number than the parent solution. It can't have a higher major or minor version. For example, for a base solution with version 3.1.5.7, a patch could be a version 3.1.5.8 or version 3.1.7.0, but not version 3.2.0.0. A cloned solution must have the version number greater than or equal to the version number of the base solution. For example, for a base solution version 3.1.5.7, a cloned solution could be a version 3.2.0.0, or version 3.1.5.7. When you clone a solution or patch, you set the major and minor version values for a cloned solution, and the build or revision values for a patch.

See also

Overview of tools and apps used with ALM

Removing dependencies

7/15/2022 • 9 minutes to read • Edit Online

Solution components often depend on other solution components. You can't delete any solution component that has dependencies from another solution component. Dependencies are records created automatically by the solutions framework to prevent required components from being deleted while one or more dependent components still include references to them. An example of a dependency is as follows: given a field is required for a form to function, if you ever try to execute an action that will result in the deletion of that field, the form will stop working.

> **NOTE**
> In this article, *delete* means that the component is completely removed from the system.

In this article, we'll discuss how to handle these dependencies and the strategies you can use to remove dependencies that you no longer need.

Dependencies of unmanaged vs. managed components

First, it's important to understand that dependencies only prevent operations that will delete a required component. The actions that can delete a component are different, depending on whether it's unmanaged or managed.

Unmanaged components

These components are represented by a single layer in the active solution. Any **Delete** operation on such a component results in the complete removal of the component.

Managed components

The deletion of managed components depends on multiple factors: the number of solution layers, the relative position of the layer that's being uninstalled, and the component publishers. For example, when a component is deleted, consider the following scenarios and what will be the expected behavior when you uninstall the various layers.

Example scenarios

The following example scenarios illustrate what happens to solution layers when solutions are uninstalled.

Scenario 1: Uninstall a single solution layer

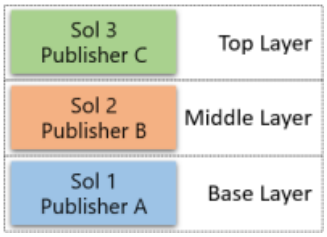
Uninstalling Solution 1 causes a component deletion because it's the only layer for the component.

Scenario 2: Uninstall solution layers from different publishers

- Uninstalling Solution 2 doesn't cause a component deletion. Only that layer will be removed.
- Uninstalling Solution 1 causes a component deletion, because the action happens in the base layer. In fact, Solution 1 can't be uninstalled in this scenario, because a solution from a different publisher extends the

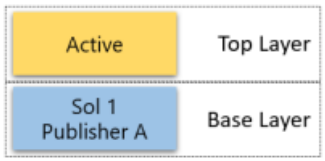
component.

Scenario 3: Uninstall multiple solution layers from different publishers



- Uninstalling Solution 3 doesn't cause a component deletion. Only that layer will be removed.
- Uninstalling Solution 2 doesn't cause a component deletion. Only that layer will be removed.
- Uninstalling Solution 1 doesn't cause a component deletion, because in this case there's another solution from the same publisher (Publisher A = Publisher C)). The platform removes the layer from Solution 1 and replaces it with the layer from Solution 3.

Scenario 4: Uninstall solution layers in an unmanaged customization

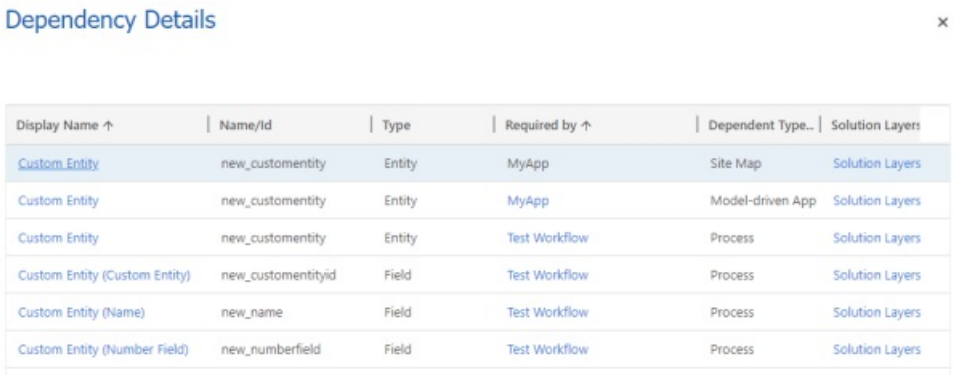


- Uninstalling the Active (unmanaged) layer doesn't cause a component deletion. Only that layer will be removed. Note that you can't uninstall the Active solution, but you can remove components by using the **Remove Active Customization** feature.
- Uninstalling Solution 1 causes a component deletion. The action happens in the base layer. Unlike scenario 2, you can uninstall Solution 1. The Active solution isn't considered an extension, and both layers will be removed.

Dependency Details page

The **Dependency Details** page lists the dependencies for the selected solution. It can be invoked by:

- Selecting **Show Dependencies** on the solution page.
- Trying to uninstall a solution, which will cause the platform to detect that dependencies exist.



The **Dependency Details** page has the following columns:

- **Display name**: The friendly name of the required component. Each component might show slightly different data to make the identification easier. In the preceding figure, you can see that the entity only shows its name, while the field displays its name and the name of its parent entity.
- **Name/Id**: The internal name of the required component.
- **Type**: The type of the required component.
- **Required by**: The friendly name of the component that requires it (the dependent component). If the

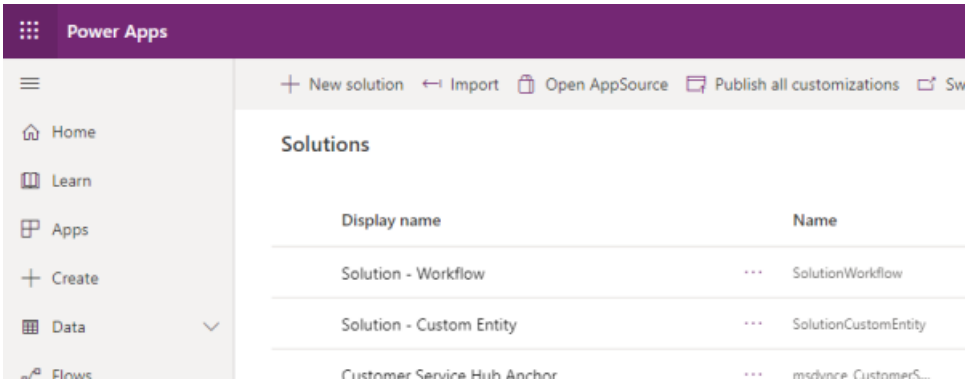
dependent component has a customization page, its name becomes a link that opens that page.

- **Dependent Type**: The type of the dependent component.
- **Solution Layers**: A link where you can see more details about the components involved in the dependency.

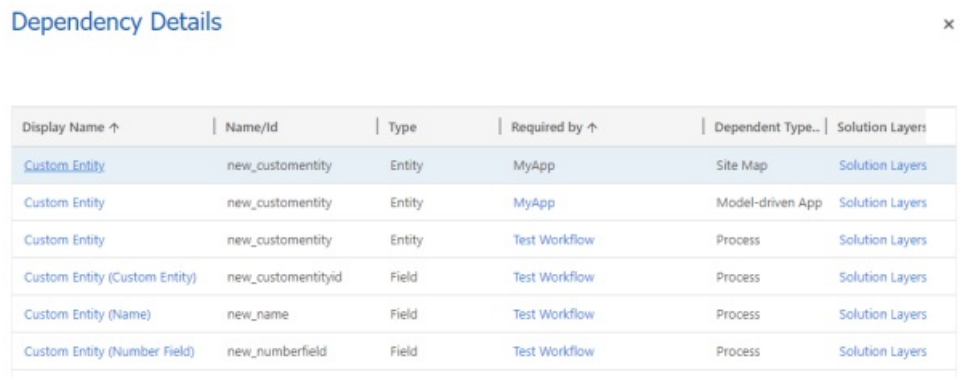
> **NOTE**
> The required component is the one that you want to delete. The dependent component is the one that has references to the required component. To remove a dependency, you must make changes that affect the dependent component, not the required component.

Diagnosing dependencies

Let's consider the following scenario. The organization below has two solutions: **Solution - Workflow** and **Solution - Custom Entity**.



The owner of the organization decided that they no longer require **Solution - Custom Entity**, tried to delete it, and was presented with the following page:

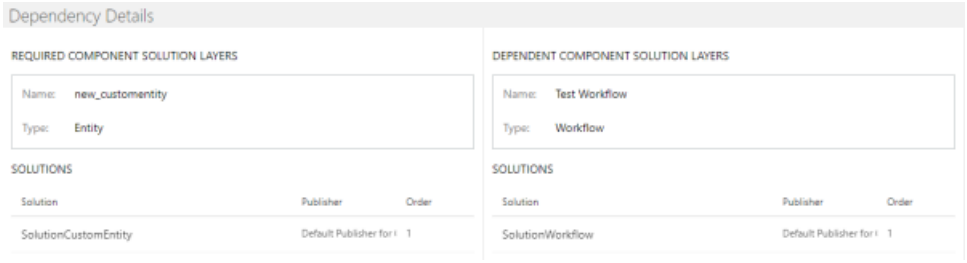


Without going into detail, we can conclude that the uninstall of the solution is trying to delete an entity named **Custom Entity** and three fields—**Custom Entity**, **Name**, and **Number Field**—and all four components have dependencies.

> **NOTE**
> Uninstalling the solution might potentially delete more components, but because they don't have dependencies, they won't appear on the list.

The next step is to check the **Solution Layers** link (rightmost column) for each dependency. That will help you decide what to do to remove the dependency.

The following figure shows dependency details between the Entity (Custom Entity) and Process (Test Workflow).

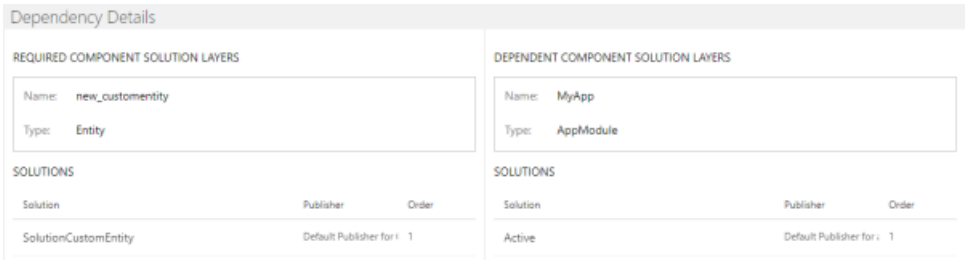


Based on the data displayed, you can see that the dependent component belongs to a solution named SolutionWorkflow. To remove this dependency, we can either:

- Update the definition of the workflow in SolutionWorkflow by removing any references to the entity or its subcomponents. Then **Update** or **Upgrade** the solution.
- Uninstall the SolutionWorkflow solution.
- Remove the workflow from a new version of the SolutionWorkflow solution, and then perform an **Upgrade**.

Because any one dependent component can prevent the removal of the solution, we recommend that you check all the dependencies and make all the required changes in a single operation.

The following figure shows dependency details between the Entity (Custom Entity) and a model-driven App (My App).



Based on the data displayed, you can see that the dependent component belongs to a solution named Active. This indicates that the dependency was created by importing an unmanaged solution, or through an unmanaged customization that was executed through the modern UI or API.

To remove this dependency, you can either:

- Edit the definition of the model-driven app to remove any reference to the entity or its subcomponents. Because model-driven apps support publishing, you must publish your changes.
- Delete the model-driven app.

> **NOTE**
> Uninstalling an unmanaged solution isn't an option for removing this dependency, because unmanaged solutions are just a means to group components.

Actions to remove a managed dependency

Managed dependencies are the ones where the dependent component is associated to a managed solution. To resolve this kind of the dependency, you must act on the solution where the component was added. That action can be different depending on what you're trying to do.

If you're trying to uninstall a solution

Follow these steps:

1. In the target organization, inspect the **Solution Layers** link to find what is the topmost solution on the list of the dependent component.

2. In the source organization, prepare a new version of that solution where the solution either doesn't contain the dependent component, or has an updated version of the dependent component that doesn't contain references to the required component. Your goal is to remove any reference to the required components in the new version of the solution.
3. Export the new version of the solution.
4. In the target organization, **Upgrade** that solution.
5. Retry the uninstall.

If you're trying to upgrade a solution

In this case, you must confirm that you wanted to delete the required component (remember that dependencies are enforced only on components that are being deleted).

If you didn't want to delete the component, you can fix the new version of the solution by adding the component back by doing the following:

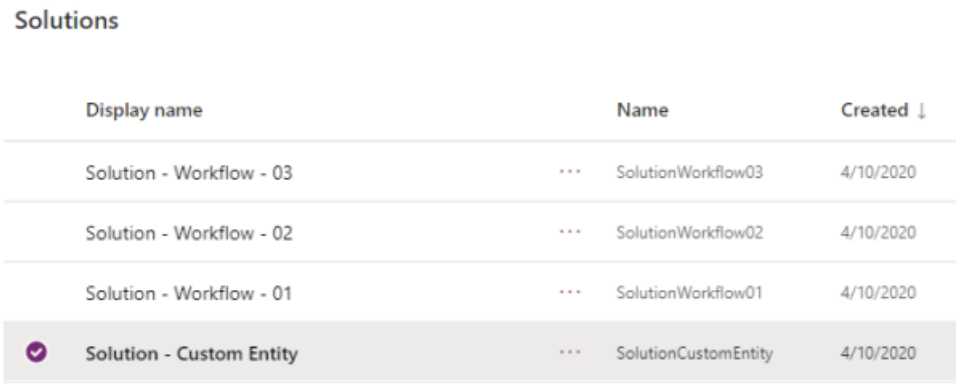
1. In the target organization, uninstall the staged solution (the solution that ends in _Upgrade).
2. In the source organization, add the required component(s) back to the solution.
3. Export the new version.
4. Retry the upgrade.

If the deletion is intentional, you must remove the dependency. Try the steps outlined in the preceding section, "If you're trying to uninstall a solution."

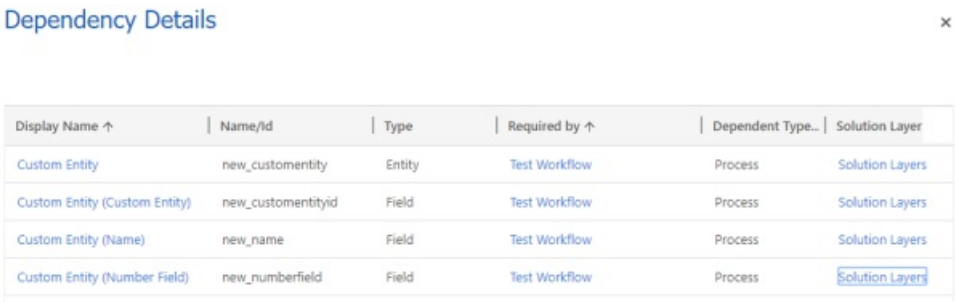
Layers and dependencies

The dependent components can be layered, so you might need to change more than one solution to completely remove a dependency. The dependency framework only calculates dependencies between the topmost layers for the required and dependent components. That means you need to work your way from the top to the bottom of the solutions of the dependent component.

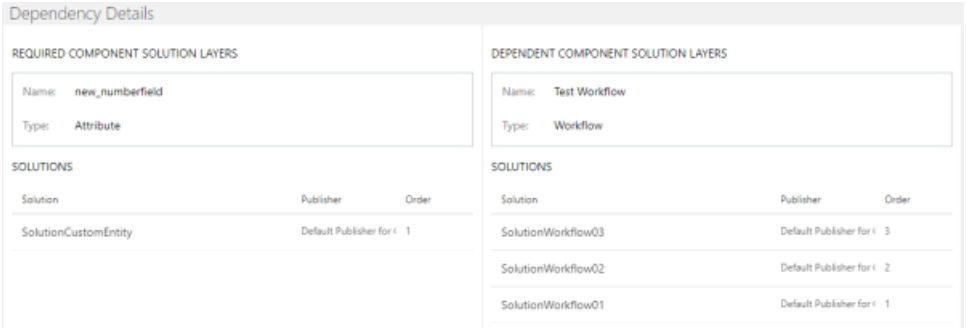
Consider the following scenario:



You try to uninstall **Solution - Custom Entity**, and the operation is blocked by dependencies.

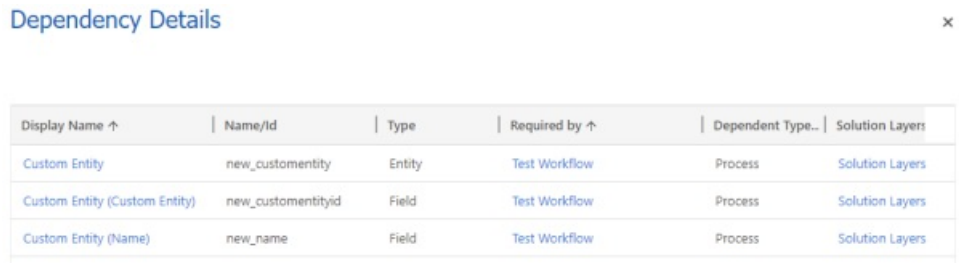


You start diagnosing the dependency by selecting **Solution Layers** on the **new_numberfield** attribute. You see the following screen:



Because dependencies are created only between the topmost layers of each component, the first step is to deal with the dependency between the **new_numberfield** attribute in SolutionCustomEntity and the **Test Workflow** workflow in SolutionWorkflow3.

To remove the dependency, you decide to uninstall SolutionWorkflow3. You do so, but when you try to uninstall the solution once more, you're presented by the same page of dependencies:



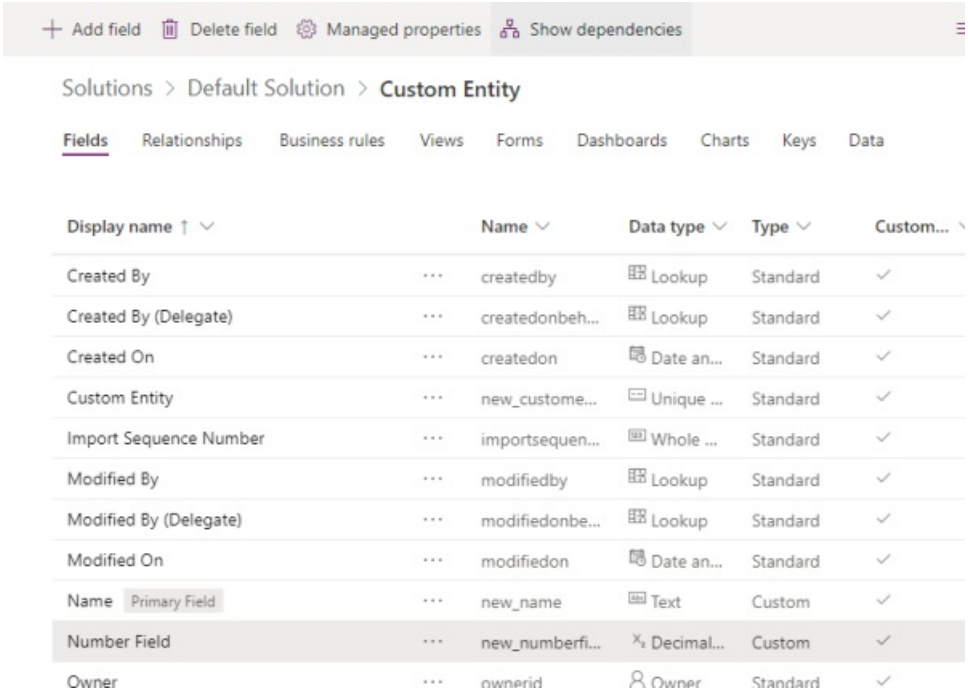
However, the **new_numberfield** attribute is no longer listed, even if it existed in more layers.

Actions to remove an unmanaged dependency

To remove unmanaged dependencies, you need to act directly on the components, not in the solutions they belong to. For example, if you want to remove the dependencies between an attribute and a form, you must edit it in the Form Editor and remove the attribute from the form. The dependency will be removed after you select **Save** and **Publish**.

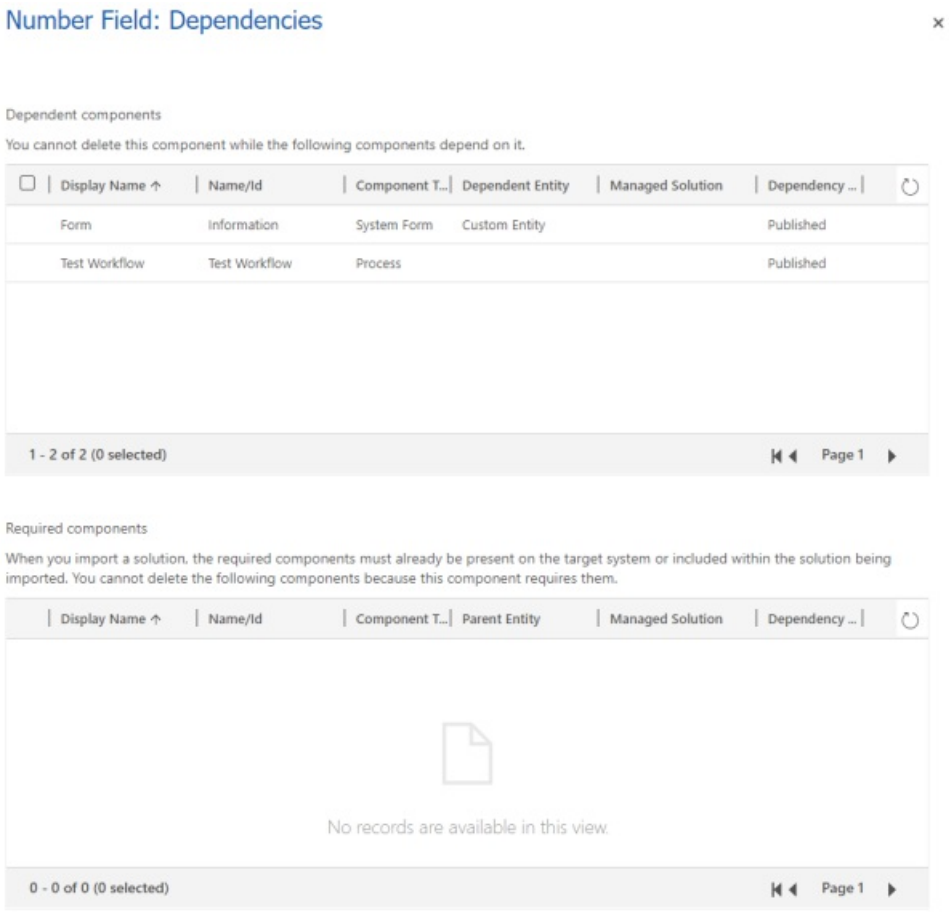
> **NOTE**
> You can also delete the dependent component. That action deletes all dependencies, along with the component.

To see the dependencies of a component, locate it in the customizations page, and then select **Show dependencies**.



The page of dependencies has two distinct parts:

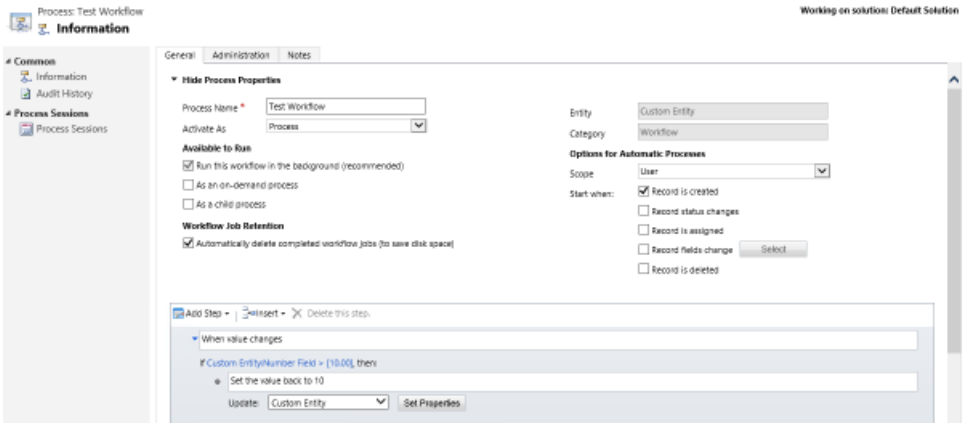
- Dependent components: A list of components that depend on the selected field. In other words, these components will have this field as their required component.
- Required components: A list of components that this field requires in order to work. In other words, these components will have this field as their dependent component.



Field and workflow

To remove dependencies between fields (attributes) and workflows (processes), locate the workflow in the customizations page.

When viewing the workflow details, find the reference to the component you no longer want the workflow to depend on. In this example, you can see the field **Number Field** being referenced in a process step.

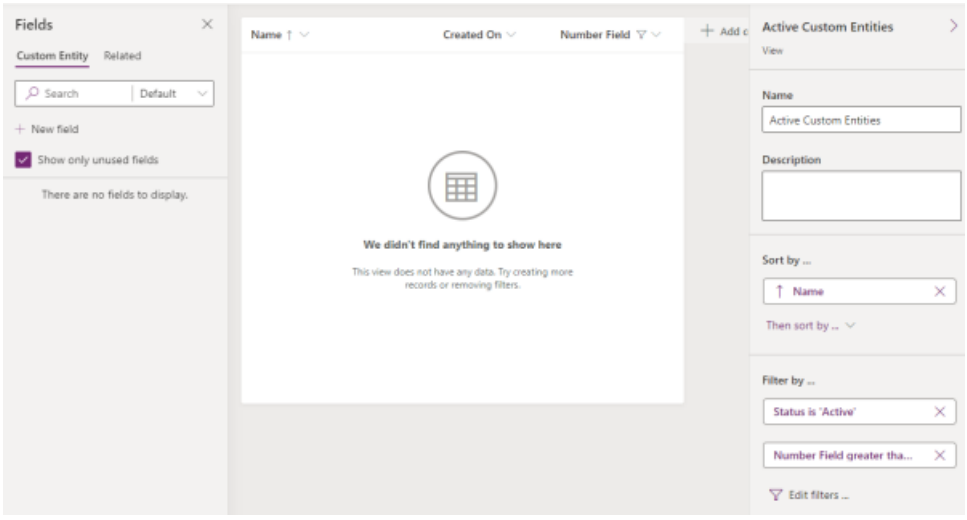


Delete (or change) the step, and then save the workflow.

Field and view

To remove dependencies between fields (attributes) and views (saved queries), locate the view in the customizations page.

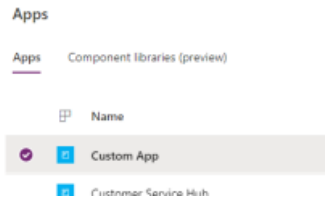
In the field editor, find the reference to the component you no longer want the view to depend on. In this example, you see the field **Number Field** being used as a select column and a filter.



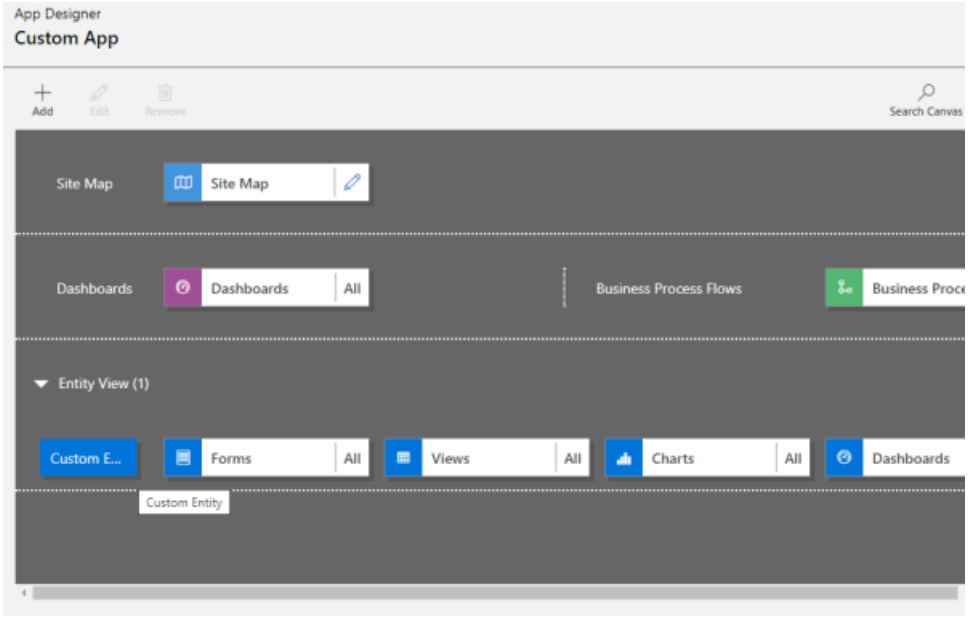
Remove both, save, and then publish the view.

Entity and model-driven apps

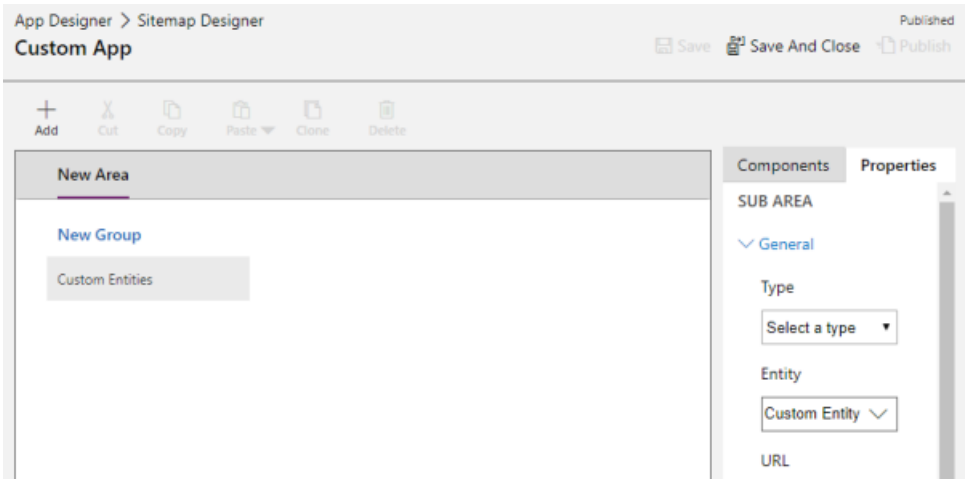
To remove dependencies between entities and model-driven apps (App Module), locate the app in the **Apps** list of the modern UI.



In the app designer, find the reference to the component you no longer want the app to depend on. In this example, you see the entity **Custom Entity** under **Entity View**.



Also, inspect the site map associated with the app, because it's likely that you'll find references there.



Remove all references, and then save and publish both the app and the site map.

> **NOTE**
>
> After being edited, components can be added to managed solutions and transported to other organizations to remove managed dependencies.

See also

Solution concepts
Solution layers
Dependency tracking for solution components]

Organize your solutions

7/15/2022 • 3 minutes to read • Edit Online

Before you create solutions, take some time to plan ahead. For example, think about how many solutions you want to release and whether the solutions will share components.

Also, determine how many Microsoft Dataverse environments you'll need to develop your line of solutions. You can use a single environment for most strategies described in this article. However, if you decide to have only one environment and later realize that you need more, it can be challenging to change the solutions if people have already installed them. Using multiple environments, although introducing more complexity, can provide better flexibility.

The following sections describe different strategies for managing solutions listed in order from simple to more complex.

Single solution

By creating a solution, you establish a working set of customizations. This makes it easier to find items that you have customized.

This approach is recommended when you only want to create a single managed solution. If you think you may have to split up the solution in the future, consider using multiple solutions.

Multiple solutions

If you have two unrelated solutions that don't share components, the most direct approach is to create two unmanaged solutions.

> **NOTE**
> It is very common in solutions to modify the application ribbons or the site map. If both of your solutions modify these solution components, they are shared components. See the following section to see how to work with shared components.

Multiple solution layering and dependencies

When you import different solutions into your target environment, you are often creating layers where the existing solution lies underneath the one being imported. When it comes to solution layering, it is important that you don't have cross-solution dependencies. Having multiple solutions in the same environment using the same unmanaged component should be avoided. This is especially true with tables.

Segment your solutions by component type when there are no cross-dependency risks. For example, have one solution that includes all of your tables, another solution that has all of your plug-ins, and a third solution that has all of your flows. These different components don't have risks of cross-solution dependencies. Therefore, it is safe to have multiple solutions formed this way in the same environment.

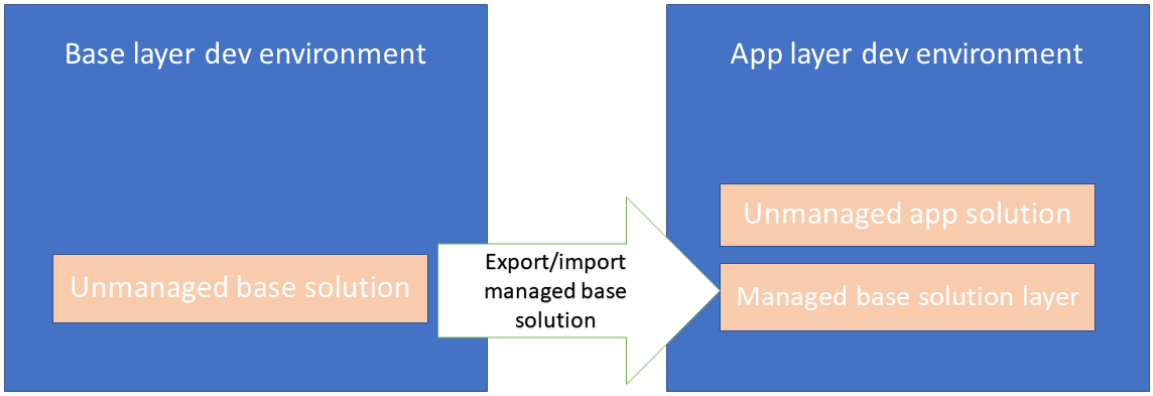
Don't have two different solutions in an environment where both contain tables. This is because there are frequently risks of a single relationship between tables, which creates a cross-solution dependency and causes solution upgrade or delete issues in the target environment at a later point in time.

When you are designing your solution layers and you want to have a structured approach for apps you should

start with a base layer. Later, you import additional solutions that will reside on top of the base layer. Subsequently, you have a base layer and extension layers on top that extend that base layer.

When you manage your projects this way, we recommend that you use a separate environment for each layer. Build your solution layering using these steps.

1. Before you create the solutions in the following steps, use a single publisher for all your solutions across your environments. More information: Solution publisher
2. In the "base" environment, you have your base solution with the unmanaged tables from that environment and no other tables. You then export this solution as managed.
3. You set up a second environment for the extension or "app" layer that will later reside on top of the base layer.
4. You import the managed base layer into the app layer environment and create an unmanaged solution for the app layer.



You can now extend the data model by adding additional tables, columns, table relationships, and so on, into the app solution. Then, export the app solution as managed. Notice that the app solution will have dependencies on the base layer solution.

In your production environment, you import the managed base layer and then import the managed app layer. This creates two managed layers in the environment with clear dependencies between the two managed solutions. Managing multiple solutions this way won't create cross-solution dependencies, which can cause solution maintenance issues, such as removing the top layer if needed.

Repeat this segmentation pattern to have as many different solutions as you need to maintain. Although we recommend that you keep the number of solutions as small as possible to keep your solution layering manageable.

See also

Use segmented solutions
Scenario 5: Supporting team development

Maintain managed solutions

7/15/2022 • 2 minutes to read • Edit Online

Before you release your managed solution you should consider how you will maintain it. Uninstalling and reinstalling a managed solution is practically never an option when the solution contains entities or attributes. This is because data is lost when entities are deleted. Fortunately, solutions provide a way to update your managed solution while maintaining the data. Exactly how you update your solutions will depend on the characteristics of the solution and the requirements of the change.

Version compatibility

Solutions can be exported from older versions of a Microsoft Dataverse environment (or Dynamics 365) and imported into newer environments, but not the reverse.

As additional service updates are applied to Dataverse, solutions exported from environments with those updates cannot be imported into environments which do not have those updates. More information: Solution concepts.

The `<ImportExportXml>` root element uses a `SolutionPackageVersion` attribute to set the value for the version that the solution is compatible with. You should not manually edit this value.

Create managed solution updates

There are two basic approaches to updating solutions:

- Release a new version of your managed solution
- Release an update for your managed solution

Release a new version of your managed solution

The preferred method is to release a new version of your managed solution. Using your original unmanaged source solution, you can make necessary changes and increase the version number of the solution before packaging it as a managed solution. When the environments that use your solution install the new version, their capabilities will be upgraded to include your changes. If you want to go back to the behavior in a previous version, simply re-install the previous version. This overwrites any solution components with the definitions from the previous version but does not remove solution components added in the newer version. Those newer solution components remain in the system but have no effect because the older solution component definitions will not use them.

During the installation of a previous version of a solution Dataverse will confirm that the person installing the previous version wants to proceed.

Release an update for your managed solution

When only a small subset of solution components urgently requires a change you can release an update to address the issue. To release an update, create a new unmanaged solution and add any components from the original unmanaged source solution that you want to update. You must associate the new unmanaged solution with the same publisher record as was used for the original solution. After you finish with your changes, package the new solution as a managed solution.

When the update solution is installed in an environment where the original solution was installed the changes included in the update will be applied to the environment. If you needs to 'roll back' to the original version you can simply uninstall the update.

Any customizations applied to the solution components in the update will be overridden. When you uninstall the update they will return.

See also

Publish your app on AppSource

Overview of tools and apps used with ALM

7/15/2022 • 3 minutes to read • Edit Online

This article gives a brief overview of the tools and apps used with application lifecycle management (ALM).

Power Platform admin center

The Power Platform admin center provides a unified portal for administrators to manage environments and settings for Power Apps, Power Automate, and model-driven apps in Dynamics 365 (such as Dynamics 365 Sales and Dynamics 365 Customer Service). From the admin center, administrators can manage environments, data integration, gateways, data policies, and get key Microsoft Power Platform metrics through Microsoft Dataverse analytics, Power Automate analytics, and Power Apps analytics.

More information:

- Power Platform admin center capabilities
- Administer Power Apps
- White paper: Administering a low-code development platform

Power Apps

Power Apps is part of a suite of apps, services, connectors, and data platform that provides a rapid application development environment to build custom apps for your business needs. Using Power Apps, you can quickly build custom business apps that connect to your business data stored either in the underlying data platform (Dataverse) or in various online and on-premises data sources, such as Microsoft 365, Dynamics 365, SQL Server, and so on. More information: What is Power Apps?

DevOps

DevOps is the combining of two historically disparate disciplines: software development and IT operations. The primary goal of DevOps is to shorten the software development lifecycle and provide continuous integration and continuous delivery (CI/CD) with high software quality. You can use Power Apps build tools to automate common build and deployment tasks related to Power Apps if your DevOps platform is Azure DevOps. This includes synchronization of solution metadata between development environments and your version control system, generating build artifacts, deploying to downstream environments, provisioning or de-provisioning of environments, and the ability to perform static analysis checks against your solution by using the Power Apps checker service. More information: Microsoft Power Platform Build Tools for Azure DevOps overview

Version control system

A version control system is a category of software tools that help record changes to files by keeping track of changes committed to software code. A version control system is a database of changes, which contains all the edits and historical versions of a software project. Version control systems allow you to maintain a single "source of truth" and recall specific versions when needed. Git is a popular example of a version control system.

IMPORTANT

Notice that *source control* applies both to Dataverse solutions and "traditional" source code. Dataverse solutions should always be part of the source code and never stored solely in Microsoft Power Platform environments. More information: Getting started: What is Git?

Configuration Migration Tool

The Configuration Migration Tool enables you to move configuration and/or reference data across environments. Configuration/reference data is different from user and transactional data and is used to define custom functionality in apps based on Dataverse. More information: Move configuration data across environments and organizations with the Configuration Migration Tool

NOTE

The Configuration Migration Tool is best suited for migrating relational configuration data. Environment variables are recommended for storing and migrating non-relational configuration parameters.

Package Deployer

Package Deployer lets administrators or developers deploy comprehensive packages of relevant assets to Dataverse instances. Packages can consist of not only solution files, but also flat files, custom code, and HTML files. Common Data Service provides you with a Visual Studio template for creating these packages that can be used with the Package Deployer tool or with PowerShell to deploy them to a Common Data Service instance. More information: Create packages for the Package Deployer

Solution Packager

Solution Packager is a tool that can unpack a compressed solution file into multiple XML files and other files, so they can be easily managed by a source control system. More information: Use the Solution Packager tool to compress and extract a solution file

Power Platform CLI

Microsoft Power Platform CLI is a simple, single-stop developer command-line interface that empowers developers and app makers to create code components. More information: What is Microsoft Power Platform CLI?

PowerShell modules

With PowerShell cmdlets for administrators, app makers, and developers, you can automate many of the monitoring, management, and quality assurance tasks that are only possible manually today in Power Apps or the Power Apps admin center.

- Online management API module. The online management API PowerShell module is used to manage Dataverse environments.
- Package deployment module. The package deployment PowerShell module is used to deploy packages to Dataverse environments and Dynamics 365 Customer Engagement(on-premises) deployments.
- Power Apps checker module. The Power Apps checker PowerShell module interacts with the Power Apps checker service so you can run static analysis jobs and download the results.

More information: Power Apps PowerShell overview

See also

Implementing healthy ALM

About healthy ALM

7/15/2022 • 2 minutes to read • Edit Online

This section provides information about the various scenarios that will help you reach the end goal of implementing and practicing healthy application lifecycle management (ALM) by using Power Apps, Power Automate, and Microsoft Dataverse in your organization.

The scenarios range from people who are new to Microsoft Power Platform to existing users practicing unhealthy ALM, and show how you can move to a recommended, successful ALM implementation.

See also

ALM basics in Power Apps

Scenario 0: ALM for a new project

7/15/2022 • 2 minutes to read • Edit Online

If you're new to Power Apps and creating your first app, follow the tasks described in this article to successfully deploy a functioning application to your production environment using a healthy application lifecycle management (ALM) strategy.

TASK	DESCRIPTION	MORE INFORMATION
1. Plan and implement your environment strategy.	Determining the environments you'll need and establishing an appropriate governance model is a critical first step. At a minimum, you should have two environments: dev and production. However, we recommend that you have three environments: dev, test, and production.	Environment strategy
2. Create a solution and publisher.	Start with a blank solution, and create a custom publisher for that solution.	Solution publisher
3. Set up your DevOps project.	Set up a DevOps project in which you'll later add several pipelines to perform required processing like export and deployment of your solution.	Setting up continuous integration and deployment Create a project
4. Create your export from development pipeline in DevOps.	Create a DevOps pipeline to export your completed unmanaged solution to a managed solution.	Create your first pipeline Build and release pipelines Build pipeline: Export a solution from development (DEV)
5. Configure and build your app.	Create your app within the solution you created.	Model-driven apps: Build your first model-driven app from scratch Canvas apps: Create an app from scratch using Microsoft Dataverse
6. Add any additional customizations to that solution.	Add additional components as needed. Choose from a vast selection of components, such as flows, AI models, export to data lake configuration, web resources, plug-ins, and even other apps.	Create components in a solution Add an existing component to a solution
7. Create a deployment pipeline in DevOps.	Create a DevOps pipeline to deploy your managed solution to one or more target production environments.	Build and release pipelines Release pipeline: Deploy to production (PROD)

TASK	DESCRIPTION	MORE INFORMATION
8. Grant access to the app.	Assign licensing and assign security roles to share applications with users.	Licensing Share a model-driven app Share a canvas app

See also

Scenario 1: Citizen development (app and flow makers)

Scenario 1: Citizen development

7/15/2022 • 2 minutes to read • Edit Online

This scenario aims at the legacy canvas app makers and flow makers in Power Apps and Power Automate, respectively, who work in a Microsoft Dataverse environment *without* a Dataverse database.

You currently build canvas apps and flows that are *not* part of a solution, and share your apps and flows by specifying each user by name or specifying a security group in Azure Active Directory who can access your app or flow.

The end goal is to move your legacy canvas app and flows to a managed application lifecycle management (ALM) model by creating apps and flows in a Dataverse solution.

> **NOTE**
> You can use the default Dataverse environment as a playground to get acquainted with the app-building experience. However, if you'll be shipping shared components (such as entities and AI models) along with your apps and flows as part of a solution, we recommend you move to a model where you have multiple environments, each dedicated for the development, testing, and release of your apps. For more information about environments, see Types of environments used in ALM.

For legacy canvas app and flow makers to participate and move to a healthy ALM model, you must do the following:

1. Work with your IT admin/organization to discuss your business processes and environment strategy, such as the number of environments, access permissions, groups, and security. More information: ALM environment strategy
2. Understand the concept of solutions in Dataverse as a mechanism to transport and share components with others. You can export a solution from one environment to another to easily and securely share your apps and flows. More information: Solution concepts
3. Follow the steps in Scenario 0: ALM for a new project to move to a healthy ALM.

See also

Scenario 2: Moving from a single production environment

Scenario 2: Moving from a single production environment

7/15/2022 • 2 minutes to read • Edit Online

This scenario aims at users or organizations that have a single Common Data Service environment with a database that they use to create, test, and deploy their apps. Having a single environment poses challenges in terms of change management and control.

To move to healthy application lifecycle management (ALM), you must:

1. Separate your development and production environments, at a minimum. The ideal solution is to have three environments, one each for development, testing, and production/deployment. More information: ALM environment strategy
2. Use solutions to ship changes across environments: More information: Use a solution to customize
3. Automate the application lifecycle process using DevOps. More information: Microsoft Power Platform Build Tools for Azure DevOps

See also

Scenario 3: Moving from unmanaged to managed solutions in your organization

Scenario 3: Moving from unmanaged to managed solutions in your organization

7/15/2022 • 3 minutes to read • Edit Online

This scenario addresses a situation where your production environment contains several unmanaged solutions or your customizations were made in the default solution. The tasks described here show you how to convert all of your unmanaged model-driven app components to managed using a single solution that will be used to create a single managed layer in your test and production environments. Later, you may want create additional solutions to develop different layering strategies and dependencies between solutions.

With the exception of your development environment, the end result is to only have managed solutions in your environments. More information: Managed and unmanaged solutions.

Prerequisites

- Separate development and production environments. Additionally, we recommend that you also maintain at least one test environment that's separate from your development and production environments.
- Use a single publisher for all your solutions across all your environments. More information: Solution publisher

Convert an unmanaged solution to managed

1. Identify and remove unnecessary tables and components.

    a. Create a back up of your production environment. You can bring back components that might be inadvertently deleted through solution export and import in the next step.

    b. Remove tables and components that are not needed from your production environment. For example, consider deleting tables with no records or very old records, or tables and components that have no dependencies. More information: View dependencies for a component

2. Create a solution to use to convert components from unmanaged to managed.

    - In your development environment, create a *single* unmanaged solution that will be used to contain *all* Microsoft Dataverse model-driven apps, tables, and dependant components, such as forms, views, fields, charts, and dashboards. Incorporating all of these components together can help reduce the chances of cross-solution layering issues that might occur later when you update or introduce new model-driven apps and customizations. More information: Create a solution
        - For *unmanaged* components, such as custom unmanaged tables, you won't use segmentation but will select **Include all components** when adding those components to the solution.
        - If there are *managed* components that you've customized, use segmentation when adding those components to the solution. For example, if you've added a custom column or changed the display name for an existing column to a Power Apps standard table, such as **Account** and **Contact**, use segmentation so you only export the customized components your project needs and not additional components that you don't intend to service. To do this, choose **Select components**, and then add only your customized components to the solution.

> **TIP**
>
> To see if a managed component has been customized, look for an unmanaged layer that will be above the base managed layer of the component. More information: View solution layers for a component

- If you have canvas apps, flows, portals apps, or plug-ins to convert, you can add them to a separate unmanaged solution now, or at a later time.
- Remember to use a single publisher for all your solutions. More information: Solution publisher

3. Deploy the managed solution.

   a. If you have an existing test environment you can go to the next step. However, we recommend that you make a copy of your production environment to use as the test environment. More information: Copy an environment
   b. Export the unmanaged solution(s) from your development environment as *managed*. More information: Export solutions
   c. If there's an unmanaged solution in your test environment that has the same name as the managed solution you want to import, delete the unmanaged solution record in the test environment. To delete the unmanaged solution record, go to Power Apps, select the test or production environment, select **Solutions**, select the unmanaged solution, and then on the command bar, select **Delete**. Notice that deleting the unmanaged solution doesn't delete the solution's unmanaged components.
   d. Import the solution into your test environment using Power Platform CLI solution import commanding with the *convert-to-managed* parameter or the DevOps tooling solution import task. Currently, you can't successfully import the solution and convert all components to managed using the Power Apps portal.
   e. There are various types of tests you can use to check whether specific functions or features of your app are working correctly. Some of these tests include unit tests, end-to-end, and user acceptance tests (UAT).
   f. After testing is completed and validated, repeat steps c-e, but instead of performing the step in your test environment perform the step in your *production* environment.
   g. If you have canvas apps, flows, or portals apps, import the managed solution(s) first into your test and then into your production environments in a similar fashion as described above.

Next steps

Now that you've moved to managed solutions, we recommend that you understand solution and component layering. Moreover, with the exception of your development environments, there shouldn't be any unmanaged customizations or unmanaged solutions in your environments. More information: Solution layers

See also

Scenario 5: Supporting team development

Scenario 4: Using DevOps to move from manual to automated ALM

7/15/2022 • 2 minutes to read • Edit Online

This scenario aims at a situation where you're shipping your solutions manually and don't have a source code control system in place.

For this scenario, you should:

1. Consider adopting a source code control system like Azure DevOps. Azure DevOps provides developer services for support teams to plan work, collaborate on code development, and build and deploy applications.
2. Export a solution from your development environment containing your apps and customizations, unpack your solution, and store the components in your source control system. Use Azure Pipelines to manage your components, and then deploy them to the target environment for testing. Finally, deploy to the production environment for user consumption.

See also

Microsoft Power Platform Build Tools for Azure DevOps

Scenario 5: Supporting team development

7/15/2022 • 2 minutes to read • Edit Online

Supporting team development consists of multiple apps and development team members in a structured release management process. Team development includes the following:

- **Conflict and merge resolution** It's important to consider which pieces will merge well and which should only be worked on by a single developer at a time, such as forms or site maps. More information: Understand how managed solutions are merged
- **Branching strategy** Your branching strategy should make sure that you can service an existing release while working on the next version.
- **Environment strategy** Developer isolation, and an environment strategy that supports it, is critical when multiple developers work on the same application. Environments should be short-lived and spun up on demand from the latest build. Each developer should have their own development environment in which only the required unmanaged solution is installed, and all dependent solutions are installed as managed.
- **Power Apps component library** Components are reusable building blocks for canvas apps so that app makers can create custom controls to use inside an app, or across apps using the component library. More information: Component library

See also

Scenario 6: Embrace your citizen developers in the ALM process

Scenario 6: Embrace your citizen developers in the ALM process

7/15/2022 • 2 minutes to read • Edit Online

Use the Center of Excellence (CoE) kit that provides customizable apps, flows, and dashboards to showcase the admin and governance capabilities in Microsoft Power Platform. Use the kit to adapt application lifecycle management (ALM) at scale in your organization.

See also

ALM for developers

Maintaining healthy model-driven app form ALM

7/15/2022 • 14 minutes to read • Edit Online

This article provides you with information about the various scenarios on how to implement and practice healthy application lifecycle management (ALM) for customizing forms in your model-driven app solutions.

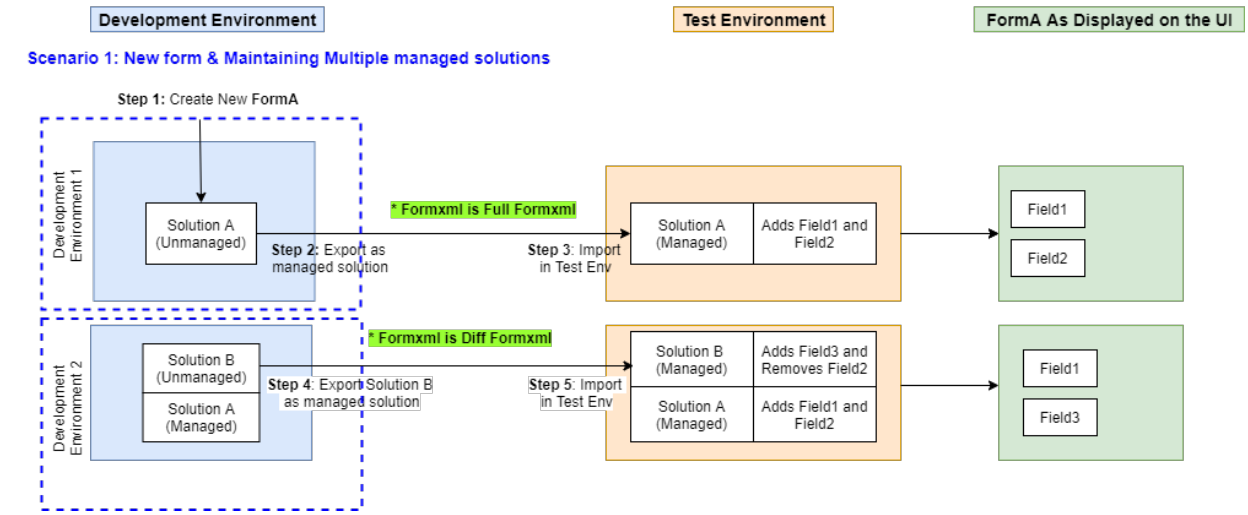
The following sections describe how form merge works and how to maintain customizations. The basic development scenarios with recommendations for maintaining successful ALM for a model-driven app form are covered in detail within each section that follows. Every scenario includes steps to follow that can help you implement a proper ALM process when updating your solution or model-driven app.

Creating a new form and maintaining it using multiple managed solutions

Follow these steps to implement healthy form ALM for this scenario.

1. Create a new form named *FormA* in your development environment and perform customizations on the form.
2. Create a new solution (named *Solution A* in the below diagram) in the development environment, which will be an unmanaged solution and add your new form. Export the solution as managed. This step exports a full FormXml for the form.
3. In your test environment, import the managed solution from step 2, which creates *FormA* in the test environment. In the below diagram, *FormA* gets created in the test environment and the UI for the form shows *Field1* and *Field2* that *Solution A* added to the form.
4. When you further customize the form you created in step 1 using a new development (source) environment, import the managed **Solution A** created in step 2, make sure the development instance you are using has **FormA** in a managed state. As shown in the diagram below, managed *Solution A* is imported in the development environment and the form is customized creating active customizations. Then, *FormA* can then be added to a new unmanaged solution (*Solution B* in the diagram) and exported as a managed solution from the development environment. This step exports a differential (diff) FormXml for the form.
5. In your test environment, import the managed solution (*Solution B*) from step 4. As shown in the below diagram *Solution B* is adding a new *Field3* to *FormA* and removing *Field2*, which was added by *Solution A*. The UI for the form in the test environment now shows *Field3* and *Field1* on the form but not *Field2* after the merge.

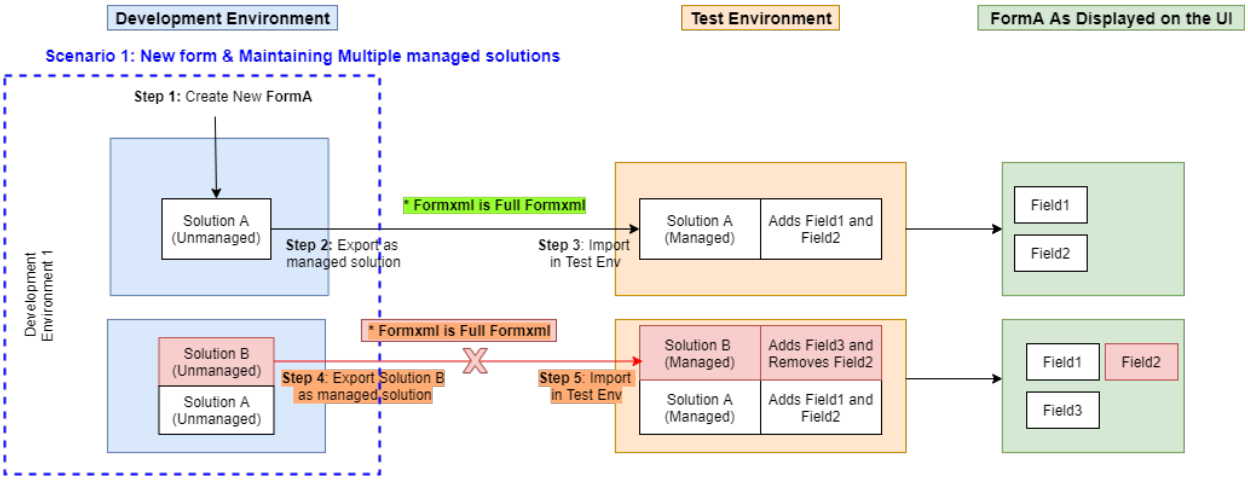
Diagram1



Unhealthy example for this scenario

As seen in the below diagram, it is not a healthy ALM practice to create multiple managed solutions from the development environment where the base solution (*Solution A*) is in an unmanaged state. This is because, when you create another unmanaged solution (*Solution B*) for the unmanaged form, the FormXml is exported as a full FormXml, instead of a diff FormXml as shown in the valid scenario above. Subsequently, changes like removing a column won't take effect.

Diagram2



Creating a new form and making customizations using patches and upgrades

Follow these steps to implement healthy form ALM for this scenario.

1. Create a new for named *FormA* in your development environment and perform customizations on the form.

2. Create a solution (*Solution A* in the below diagram), which will be an unmanaged solution and add your new form. Export the solution as managed. This step exports a full FormXml for the form.

3. In your test environment, import the managed solution from step 2, thus creating the form in the test environment. In the below diagram *FormA* gets created in the test environment and the UI for the form shows *Field1* and *Field2* that *Solution A* added to the form.

4. When you further customize the form you created in Step 1 using patches, use the same environment where *Solution A* is in an unmanaged state and create a patch for the solution and customize the form. Next, export the patch as a managed solution. This step exports a full formXml for the form.

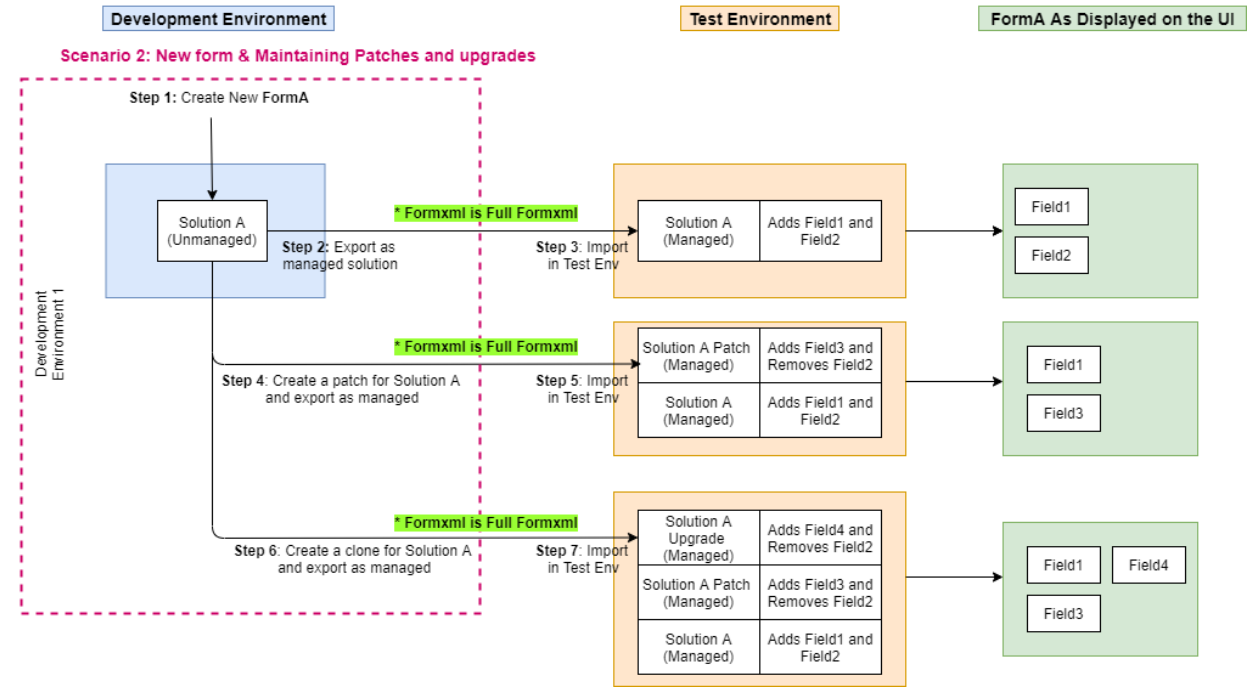
5. In your test environment, import the managed patch solution from step 4. As shown in the below diagram, the *Solution A* patch is adding a new *Field3* to FormA and removing *Field2*, which was added by *Solution A*.

> **NOTE**
>
> Patches containing full formXml are always compared to the base layer that the patch was created from and ignore any intermediate patches between the base and the current patch. As a result, *Field2* is removed since it exists in the base layer *Solution A* and the removal is detected. On the other hand, *Field3* is added by this patch solution and can't be removed by subsequent patches. Thus, fields added through patch solutions are additive in nature.

6. When you further customize the form you created in Step 1 using upgrades, use the same environment where *Solution A* is in an unmanaged state and clone *Solution A* to create the upgrade solution and customize the form. Then, export the *Solution A* upgrade as a managed solution. This step exports a full FormXml for the form.

7. In your test environment, import the managed *Solution A* upgrade from step 6. As shown in the below diagram, the *Solution A* upgrade is adding a new *Field4* to *FormA* and removing *Field2*, which was added by *Solution A*. The UI for the form in the test environment now shows *Field1*, *Field3*, and *Field4* on the form, but *Field2* will be removed after the form is merged from the import.

Diagram3

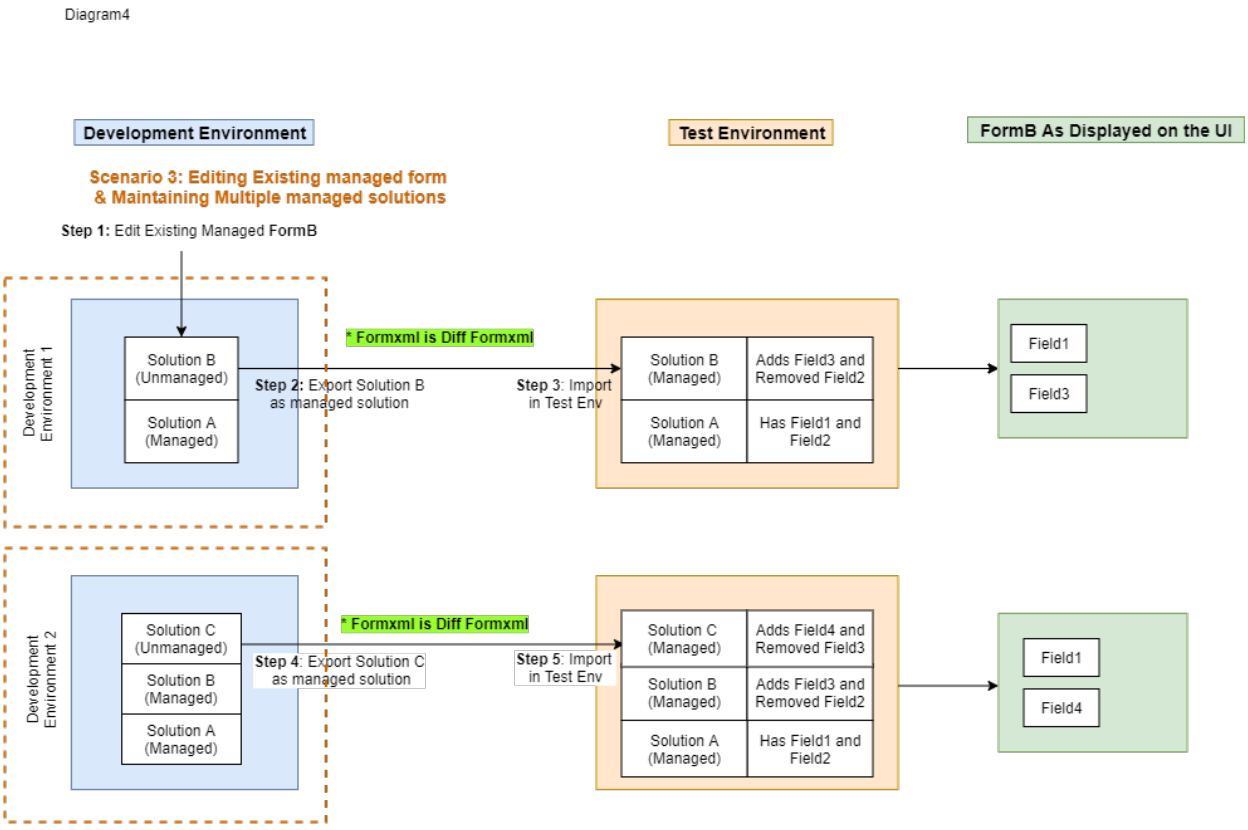


Customizing an existing managed form and maintaining it using multiple managed solutions

Follow these steps to implement healthy form ALM for this scenario.

1. Edit an existing managed form, named *FormB* in this example, in your development environment and perform customizations on the form. Note that solution A is the managed solution already installed for the form in the development environment.

2. Create a new solution (*Solution B* in the below diagram), which is an unmanaged solution, and add *FormB*. Export the solution as managed. This step exports a differential (diff) FormXml for the form.
3. In your test environment, import the managed solution from step 2, thus creating a second solution layer for the form. In the below diagram, *FormB* gets the merged changes from *Solution A* and *Solution B* in the test environment and the UI for the form shows *Field1* and *Field3* on the form but not *Field2*, which was removed by *Solution B*.
4. When you further customize the form you customized in step 1 using new managed solutions, make sure to use a new development environment which has *FormB* in a managed state. As shown in the diagram below, *Solution A* and *Solution B* managed solutions are imported in the new development environment. *FormB* is customized creating active customizations, which can then be added to a new solution (*Solution C* in the diagram) and exported as a managed solution.
5. In your test environment, import the managed *Solution C* from step 4. As shown in the below diagram, *Solution C* is adding a new *Field4* to *FormB* and removing *Field3*, which was added by *Solution B*. The UI for the form in the test environment now shows *Field1* and *Field4* on the form, but not *Field2* and *Field3*.

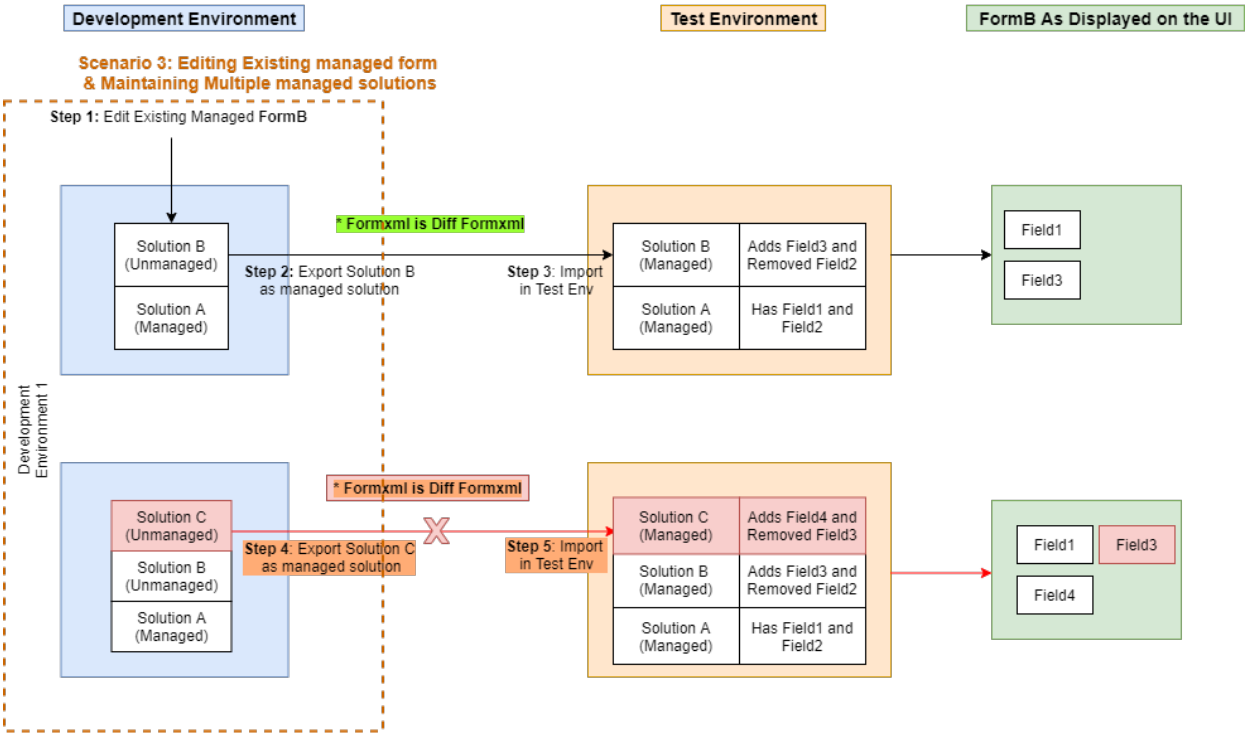


Unhealthy example for this scenario

As shown in the below diagram, it is not a healthy ALM practice to create multiple managed solutions from the development environment that contains another unmanaged solution you created for the same form. Notice that *Solution B* is in unmanaged state. When you create another unmanaged solution (*Solution C*) for *FormB*, the FormXml is exported as a diff FormXml as shown in step 4 in the above scenario. But, *FormB* also contains the changes from *Solution B*, which will get overwritten by your new changes.

For example, as seen in the diagram below, *Field3* is added to *FormB* in *Solution B*. But now when you create a new *Solution C* in this environment, with *Solution B* in unmanaged state, and remove *Field3*, *Field3* will also be removed in the development environment. *Field3* will not be tracked in the diff FormXml when the solution is exported, since the change of adding and removing this column was made in the same active layer. That means when managed *Solution C* is imported in the test environment, the form will still render the *Field3* because the diff FormXml never records it as removed (like it was removed in step 5 in the healthy form ALM scenario above). Performing your form customizations this way will lead to the development environment being inconsistent with the test environment.

Diagram5



Customizing an existing managed form and maintaining it using patches and upgrades

Follow these steps to implement healthy form ALM for this scenario.

1. Customize an existing managed form, named *FormB* in this example, in your development environment and perform customizations on the form. Note that *Solution A* is the managed solution already installed for the form in the development environment.

2. Create a solution (*Solution B*), which will be an unmanaged solution and add *FormB*. Export the solution as managed. This step exports a diff FormXml for the form.

3. In your test environment, import managed *Solution B* from step 2, thus creating a second solution layer for the form. In the below diagram, *FormB* gets the merged changes from *Solution A* and *Solution B* in the test environment. Additionally, the UI for *FormB* shows *Field1* and *Field3* on the form but not Field2, which was removed by *Solution B*.

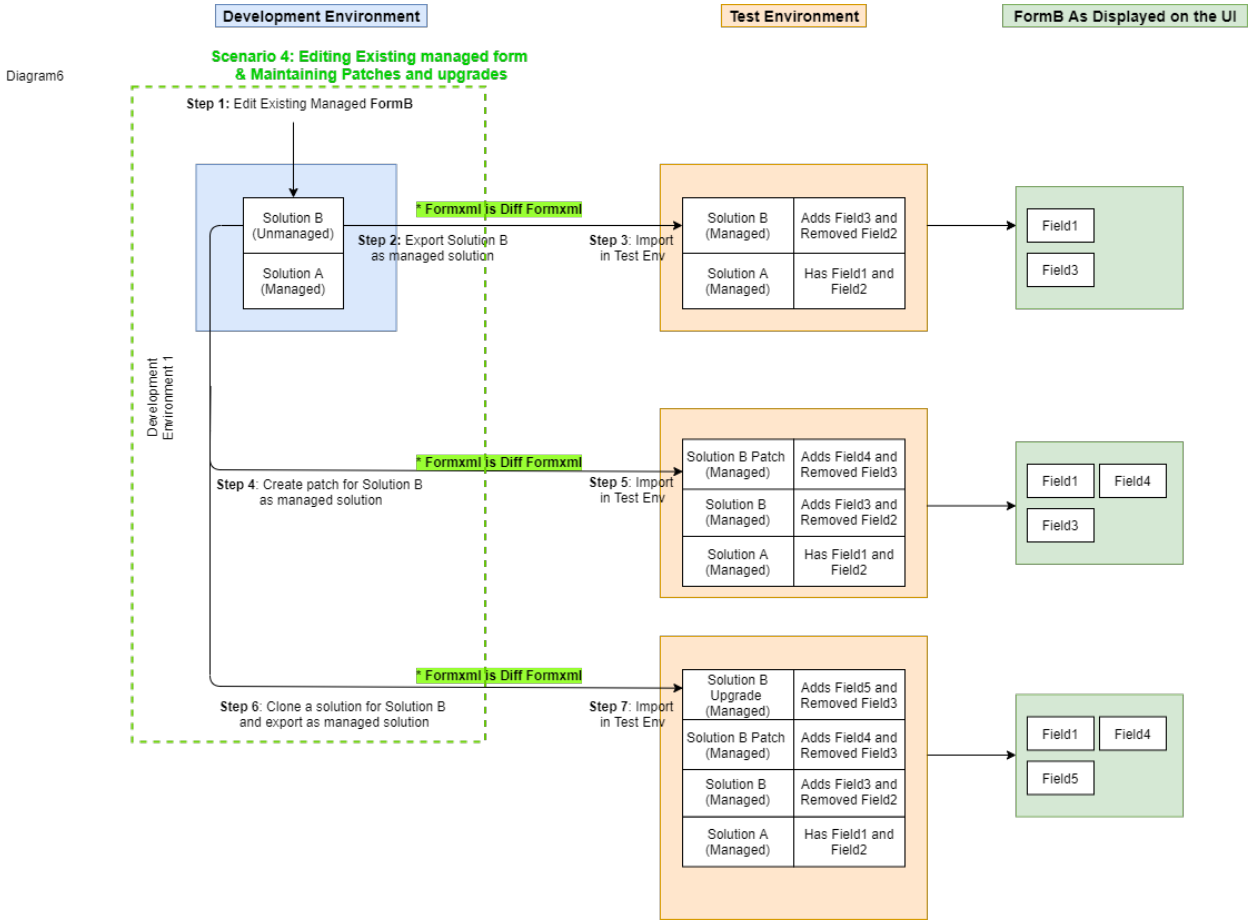
4. When you further customize the form you customized in Step 1 using a patch solution, you can use the same development environment as step 1 where *Solution B* exists in an unmanaged state. As shown in the diagram below, *Solution A* is in a managed state and *Solution B* is in an unmanaged state. The form is further customized and you create a patch for *Solution B* adding your form to this solution and exporting it as a managed patch solution. This step exports a diff FormXml.

5. In your test environment, import managed patch *Solution B* from step 4. As shown in the below diagram, *Solution B Patch* is adding a new *Field4* to *FormB* and removing *Field3*, which was added by *Solution B*.

> **NOTE**
>
> Patches are additive in nature and can't remove components, such as columns, from the form. So, *Field3* will not be removed from the form. The UI for the form in the test environment now shows *Field1*, *Field3*, and *Field4* on the form, but not *Field2*.

6. When you further customize the form you created in Step 1 using upgrades, use the same environment where *Solution B* is in an unmanaged state and clone *Solution B* to create the upgrade solution and customize *FormB*. Export the upgrade as a managed solution. This step exports a diff FormXml for the form.

7. In your test environment, import the managed *Solution B* upgrade solution from step 6. As shown in the below diagram, *Solution B Upgrade* is adding a new *Field5* to *FormB* and removing *Field3*, which was added by *Solution B*. The UI for the form in the test environment now shows *Field1*, *Field4*, and *Field5* on the form, but *Field2* and *Field3* are removed.

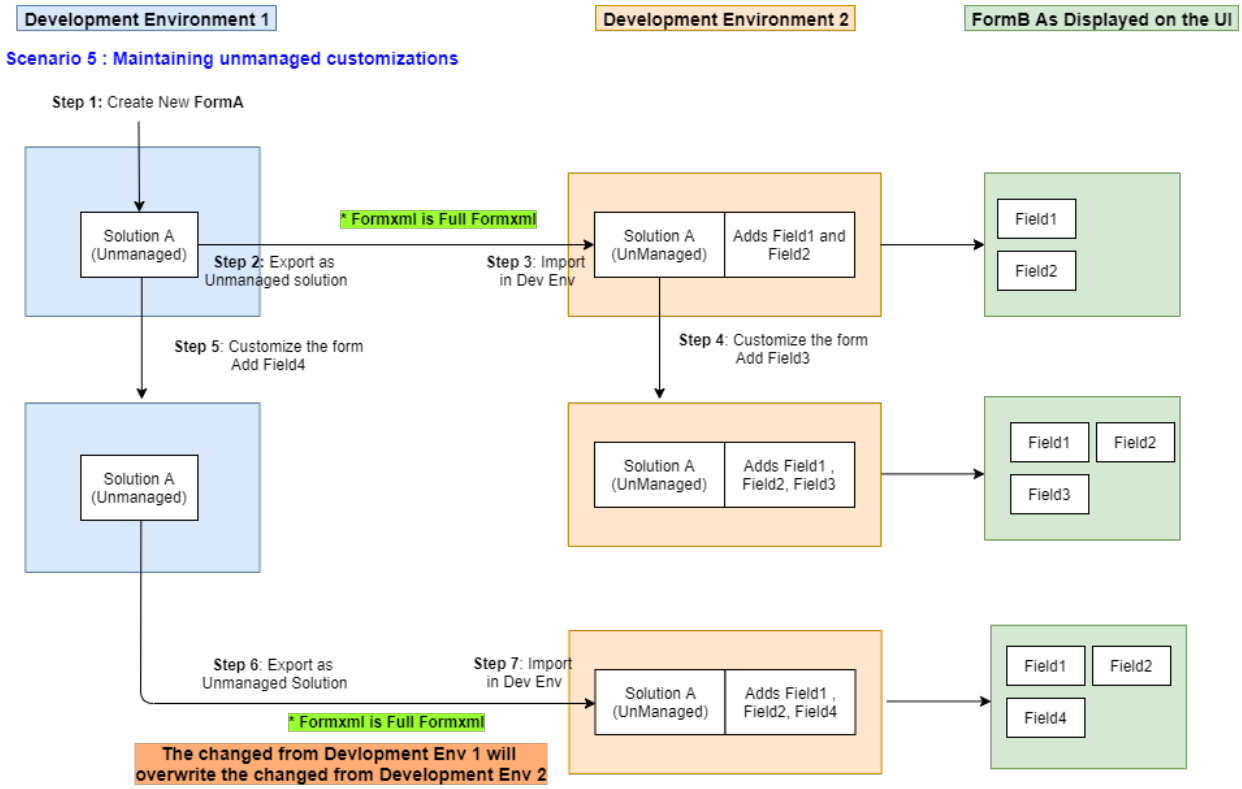


Maintaining unmanaged solutions and customizations for a new form across multiple development environments

Follow these steps to implement healthy form ALM for this scenario.

1. In *Development Environment 1*, create a new *FormA* and perform customizations on the form.
2. Create a solution (*Solution A* in the below diagram), which will be an unmanaged solution, and add your new form. Export the solution as unmanaged. This step exports a full FormXml for the form.
3. In *Development Environment 2*, import the unmanaged solution from step 2, which creates the form in *Development Environment 2*. In the below diagram, *FormA* gets created and the UI for the form shows *Field1* and *Field2* that *Solution A* added to the form.
4. You further customize the form in *Development Environment 2* making active customizations in the environment, such as adding a new column named *Field3*. *FormA* now shows *Field1*, *Field2*, and *Field3*.
5. In your *Development Environment 1*, you further customize the form also by adding *Field4*. The UI for the form in the *Development Environment 1* now shows *Field1*, *Field2*, and *Field4*.
6. Export unmanaged *Solution A* with the changes made in step 5. This step exports a full FormXml for the form.
7. In *Development Environment 2*, import unmanaged *Solution A Upgrade* from step 6. Since the solution you

are importing contains the full FormXml for *FormA*, it overwrites the active customization made in *Development Environment 1*. So, the form now shows only *Field1*, *Field2*, and *Field4*, but not *Field3*, which was the additional active customization done in *Development Environment 1*. This behavior occurs with any unmanaged solution import that has the full FormXml for the form.



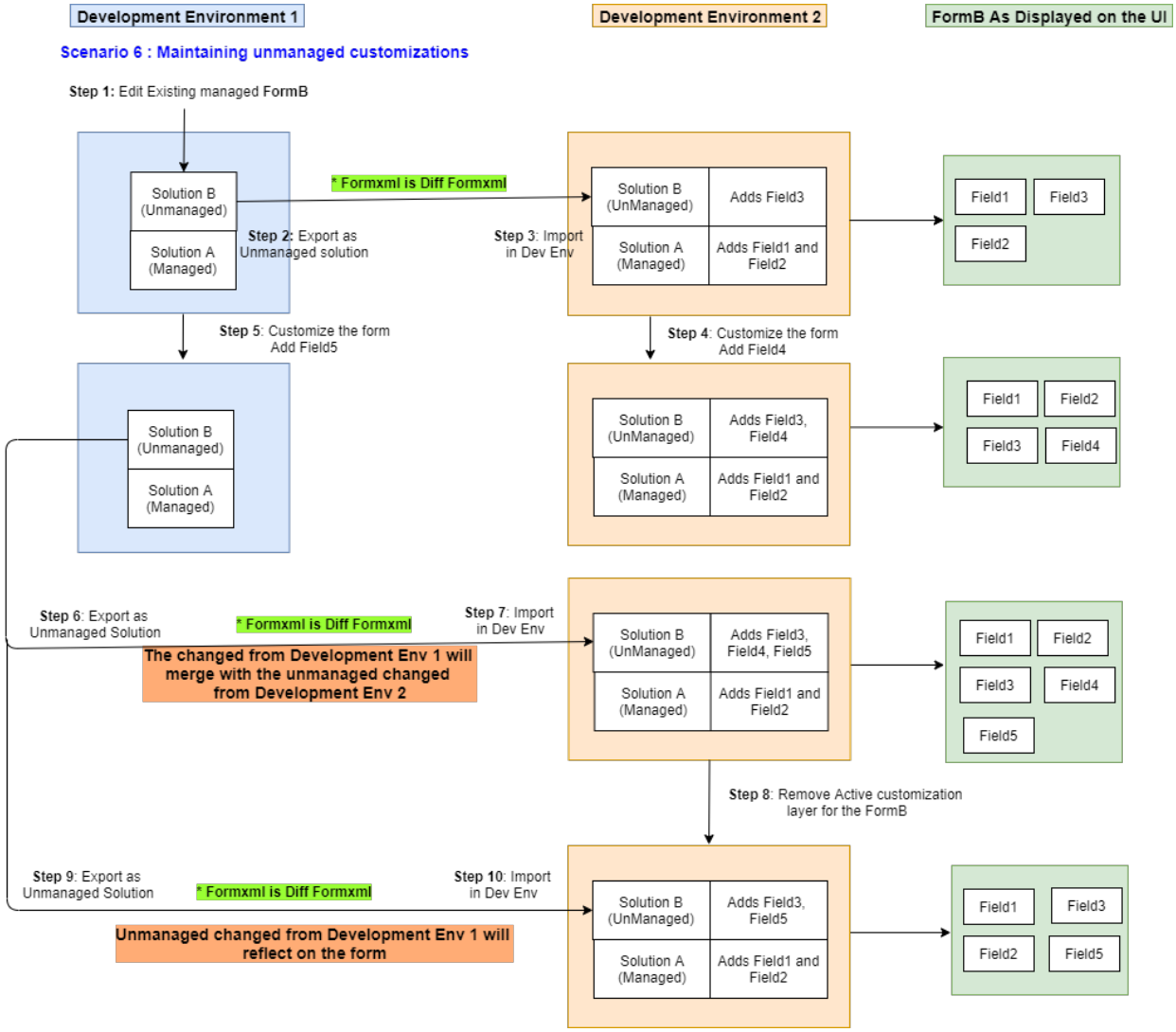
Maintaining unmanaged solutions and customizations for an existing form across multiple development environments

Follow these steps to implement healthy form ALM for this scenario.

1. In *Development Environment 1*, customize an existing form, named *FormB* in this example. Then perform customizations on the form.
2. Create a solution (*Solution B* in the below diagram), which will be an unmanaged solution, and add *FormB*. Export the solution as unmanaged. This step exports a diff FormXml for the form.
3. In *Development Environment 2*, import the unmanaged solution from step 2, thus creating a second solution layer for the form. The *FormB* UI shows *Field1*, *Field2*, and *Field3* after the form merge.
4. You further customize the form in *Development Environment 2*, making active customizations in the environment, such as adding a new column named *Field4*. *FormB* now shows *Field1*, *Field2*, *Field3*, and *Field4*.
5. In *Development Environment 1*, you further customize the form adding a new column named *Field5*. The UI for the form in *Development Environment 1* now shows *Field3* and *Field5*.
6. Export unmanaged *Solution B* with the changes made in step 5. This step exports a diff FormXml for the form.
7. In *Development Environment 2*, import unmanaged *Solution B Upgrade* from step 6. Since the solution you are importing contains the diff FormXml for *FormB,* it will merge with the active customization made in *Development Environment 1*. So, the form now shows *Field1*, *Field2*, *Field3*, *Field4*, and *Field5*. This behavior occurs for any unmanaged solution import that has the diff FormXml for the form.
8. If the form merge in step 7 is not what you want even though you are importing a diff FormXml with the unmanaged solution and you want to be able to overwrite the active customizations made in *Development Environment 2*, then remove the active layer for *FormB*. More information: Remove an unmanaged layer.
9. Export unmanaged *Solution B* with the changes made in step 5. This step exports a diff FormXml for the

form.

10. In *Development Environment 2*, import unmanaged *Solution B Upgrade* from step 9. Since there is no active layer for the form in *Development Environment 2*, (see step 8), all the changes from unmanaged *Solution B* are imported even though you are importing diff FormXml for *FormB*. So, the form now shows only *Field1*, *Field2*, *Field3*, and *Field5*. This behavior occurs for any unmanaged solution import that has the diff FormXml for the form. This is the same result as step 7 in the Maintaining unmanaged solutions and customizations for an existing form across multiple development environments scenario.



Full and differential form XML

Every exported solution package includes a customizations.xml file. Whenever a form is included in a solution, the related form definition exists within the FormXml sections of the customizations.xml file. FormXml can either be *full* or *differential (diff)*.

Full FormXml

The FormXml you get on exporting a solution for a form in an unmanaged state is called a *full* FormXml. Full means it contains the entire form definition. When you create a new form and export it, the form will always be a full FormXml because the form in the environment you are exporting from is in an unmanaged state and also is in a create state. If you export any further solutions from this same environment, those will also include a full FormXml. Because the `solutionaction` attribute indicates a diff FormXml, the full FormXml in the customization.xml file in the solution you export will not contain any `solutionaction` attributes.

Differential (diff) FormXml

The FormXml you get when exporting a solution for a form in a managed state is called a differential or *diff* FormXml. Diff means the FormXml contains only the changes done in the active customizations in that

environment and not the entire form definition. When you customize an existing managed form and export it, the form will always be a diff FormXml because it will only contain the active changes done to it. The diff FormXml in the customization.xml file in the solution you export will contain `solutionaction` attributes defining what the changes are, like **Added**, **Removed**, **Modified**.

Diff FormXml ensures that your solution will only express the changes your app needs and will be impacted less by changes from other layers. Diff FormXml also makes the solution less bulky and helps it import faster.

See also

Recommendations for healthy form ALM

Recommendations for healthy model-driven app form ALM

7/15/2022 • 2 minutes to read • Edit Online

Follow these recommendations when you customize forms:

- To create a new form, don't make manual edits to the FormXml in the customizations.xml. Instead, use the form designer to create a new form or copy an existing form by doing a **Save as**. The form designer ensures that new forms have unique ids, which avoids conflicts with existing forms. More information: Create, edit, or configure forms using the form designer.
- Don't import the same localized labels when there are existing translated labels that have translation text you want to keep. This reduces dependencies and improves solution import performance.
- If your solution publisher is the owner of a form, only the base solution in your solution package should be a full FormXml. Similarly, a solution upgrade or patch for this base solution can be full FormXml. Any other managed solution you package for this form other than the base should not be a full FormXml, but should be diff FormXml. More information: Full and differential form XML
- Forms use a merge approach when you import an update to a form. Therefore, using the **Overwrite customization** option during import doesn't have any affect on forms. We recommend that you keep this in mind when you update forms. If you want to overwrite customizations in the target environment, remove the active customizations in the unmanaged layer and then import the solution update. More information: Merge form customizations and Remove an unmanaged layer
- Don't use unmanaged solutions in your production instances. More information: Scenario 3: Moving from unmanaged to managed solutions in your organization

See also

Maintaining healthy model-driven app form ALM

ALM for developers

7/15/2022 • 2 minutes to read • Edit Online

The articles in this section describe how you as a developer can use available APIs, tools, and other resources to implement application lifecycle management (ALM) using Microsoft Power Platform.

We'll start off talking a little about team development and then dive into Azure DevOps and available build tools.

To learn more about key ALM concepts and working with solutions, see Overview of application lifecycle management.

Team development

When we refer to *team development*, we're talking about multiple developers collaborating in the same environment versus multiple developers working on the same application and sharing source code. For team development, it's a good practice to use tooling or processes to achieve developer isolation.

More information: Scenario 5: Supporting team development

Conflict management

For team development, the goal is to avoid conflict when making changes to shared code. With a source control system, branching and merging help to avoid change conflicts and keep team members from affecting each other with partially completed work in a shared repo.

Another method is to use strategies to avoid conflict. For example, you can have only one person at a time work on a complex component, to avoid a merge conflict that a source control system might not automatically be able to resolve.

Working with complex components

What are complex components? Examples include forms, canvas apps, flows, and workflows.

Coordinate efforts with other team members to avoid having more than one developer work on the same form or component at a time. If you do have multiple developers working on the same canvas app, have them work on separate components to avoid conflict.

See also

Microsoft Power Platform Build Tools for Azure DevOps
Power Apps for developers

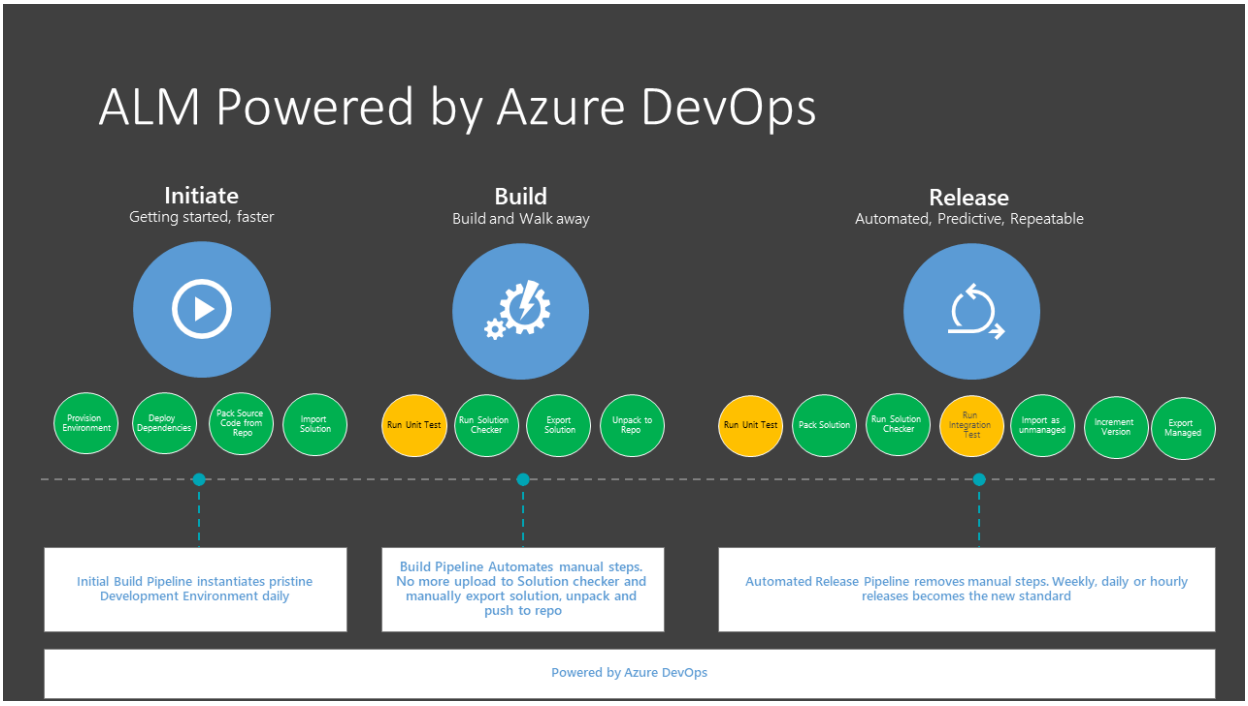
Microsoft Power Platform Build Tools for Azure DevOps

7/15/2022 • 5 minutes to read • Edit Online

Use Microsoft Power Platform Build Tools to automate common build and deployment tasks related to apps built on Microsoft Power Platform. These tasks include:

- Synchronization of solution metadata (also known as solutions) that contains the various platform components such as customer engagement apps (Dynamics 365 Sales, Customer Service, Field Service, Marketing, and Project Service Automation), canvas apps, model-driven apps, UI flows, virtual agents, AI Builder models, and connectors between development environments and source control
- Generating build artifacts
- Deploying to downstream environments
- Provisioning or de-provisioning environments
- Perform static analysis checks against solutions by using the Power Apps checker service

Microsoft Power Platform Build Tools tasks can be used along with any other available Azure DevOps tasks to compose your build and release pipelines. Pipelines that teams commonly put in place include Initiate, Export from Dev, Build, and Release.



NOTE

Microsoft Power Platform Build Tools are supported only for a Microsoft Dataverse environment with a database. More information: Create an environment with a database

Microsoft Power Platform Build Tools are now available for use in the **GCC** and **GCC High** regions.

What are Microsoft Power Platform Build Tools?

Microsoft Power Platform Build Tools are a collection of Power Platform–specific Azure DevOps build tasks that eliminate the need to manually download custom tooling and scripts to manage the application lifecycle of apps built on Microsoft Power Platform. The tasks can be used individually to perform a task, such as importing a solution into a downstream environment, or used together in a pipeline to orchestrate a scenario such as "generate a build artifact", "deploy to test", or "harvest maker changes." The build tasks can largely be categorized into four types:

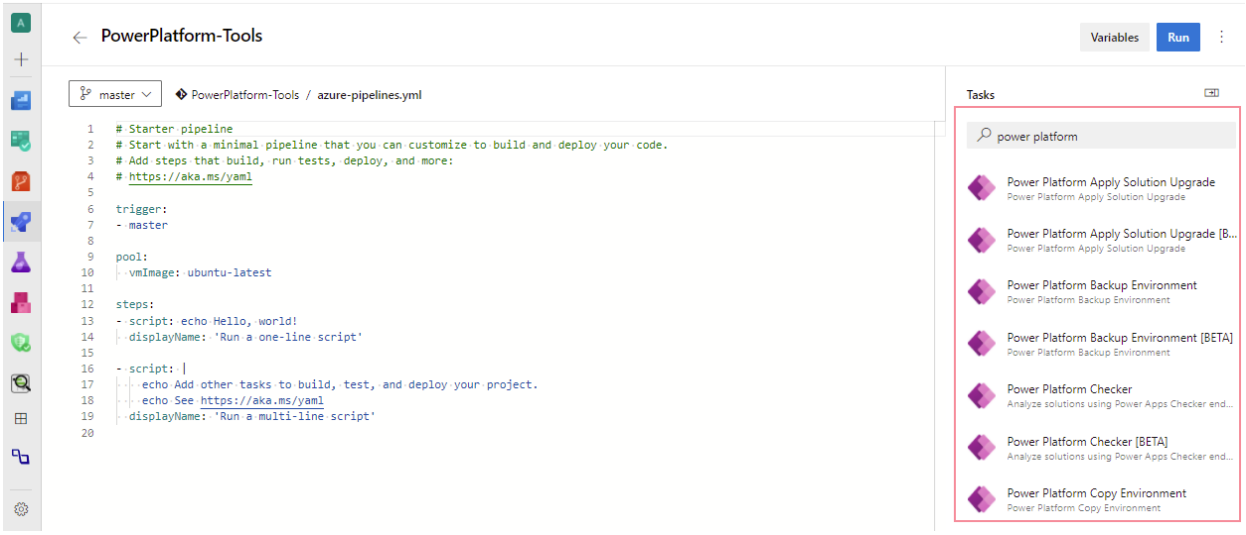
- Helper
- Quality check
- Solution
- Environment management

For more information about the available tasks, see Microsoft Power Platform Build Tools tasks.

Get Microsoft Power Platform Build Tools

Microsoft Power Platform Build Tools can be installed into your Azure DevOps organization from Azure Marketplace.

After installation, all tasks included in the Microsoft Power Platform Build Tools will be available to add into any new or existing pipeline. You can find them by searching for "Power Platform".



Connection to environments

To interact with the Microsoft Power Platform environment, a connection must be established that enables the various build tool tasks to perform the required actions. Two types of connections are available:

- Username/password: Configured as a generic service connection with username and password. Username/password doesn't support multi-factor authentication.
- Service principal and client secret: (recommended) This connection type uses service principal based authentication and supports multi-factor authentication.

Configure service connections using a service principal

To configure a connection using service principal, you must first create an application registration in Azure Active Directory (AAD) with the required permissions and then create the associated Application User in the

Microsoft Power Platform environment you want to connect to. We've offered a script to facilitate some of the steps required in the section below, while detailed information with manual step-by-step instructions are available in this article Azure application registration.

Create service principal and client secret using PowerShell

This PowerShell script helps creating and configuring the service principal to be used with the Microsoft Power Platform Build Tools tasks. It first registers an Application object and corresponding Service Principal Name (SPN) in AAD.

This application is then added as an administrator user to the Microsoft Power Platform tenant itself.

Installation

Download the following PowerShell cmdlet: https://pabuildtools.blob.core.windows.net/spn-docs-4133a3fe/New-CrmServicePrincipal.ps1

- Open a regular Windows PowerShell command prompt (standard, not PS core)
- Navigate to the folder where you saved the script, and unblock the script using the following command: `Unblock-File New-CrmServicePrincipal.ps1`
- Run the script: `.\New-CrmServicePrincipal.ps1`

The script will prompt two times with AAD login dialogs:

- First prompt: to log in as administrator to the AAD instance associated with the Microsoft Power Platform tenant
- Second prompt: to log in as tenant administrator to the Microsoft Power Platform tenant itself

Once successful, three columns are displayed:

- Power Platform TenantId
- Application ID
- Client Secret (in clear text)

Use the information displayed to configure the Power Platform service connection.

> **IMPORTANT**
> Keep the client secret safe and secure. Once the PowerShell command prompt is cleared, you cannot retrieve the same client secret again.

Configure environment with the Application ID

The Application ID must be added as an Application User in the Microsoft Power Platform environment you are connecting to. Information on how to add an application user is available in this article Application user creation.

Ensure that the added Application User has the system administrator role assigned (available from "Manage Roles" in the security settings for the application user).

Frequently asked questions (FAQs)

Do the Microsoft Power Platform Build Tools only work for Power Apps?

The build tools work for both canvas and model-driven apps, Power Virtual Agents, UI Flows and traditional flows, AI Builder, custom connectors and dataflows, all of which can now be added to a solution. This list also includes customer engagement apps (Dynamics 365 Sales, Customer Service, Field Service, Marketing, and

Project Service Automation). Separate build tasks are available for Finance and Operations applications.

I had previously installed the preview of the Build Tools - can I upgrade from the preview of Power Apps Build Tools to Power Platform Build Tools?

You cannot upgrade from the preview version as we had to introduce some breaking changes in the Generally Available release. To move from the preview version, you have to install the Microsoft Power Platform Build Tools and either rebuild your pipelines, or reconfigure your existing pipelines to use the new Build Tools tasks. You must also create new Service connections as well.

Can I include flow and canvas apps?

Yes, flows and canvas apps are solution aware so if these components are added to your solution, they can participate in the lifecycle of your app. However, some steps still require manual configurations. The need for manual configuration will be addressed later this year when we introduce environment variables and connectors. A list of current limitations are available here: Known limitations.

How much do the Microsoft Power Platform Build Tools cost?

The build tools are available at no cost. However, a valid subscription to Azure DevOps is required to utilize the Build Tools. More information is available Pricing for Azure DevOps.

I can see the extension, but why don't I have an option to install it?

*If you do not see the **install** option, then you most likely lack the necessary install privileges in your Azure DevOps organization. More info available Manage extension permissions.*

How can developers use the results of the Checker task?

The output of the Checker task is a Sarif file and both VS Code and Visual Studio extensions are available for viewing and taking action on Sarif files.

See Also

Build tool tasks
Microsoft Power Platform Build Tools labs

Microsoft Power Platform Build Tools tasks

7/15/2022 • 21 minutes to read • Edit Online

The available build tasks are described in the following sections. Afterwards, we will showcase some example Azure DevOps pipelines making use of these tasks. For information about the build tools and how to download them, see Microsoft Power Platform Build Tools for Azure DevOps.

Helper task

The available helper tasks are described below.

Power Platform Tool Installer

This task is required to be added once before any other Power Platform Build Tools tasks in build and release pipelines. This task installs a set of Power Platform–specific tools required by the agent to run the Microsoft Power Platform build tasks. This task doesn't require any more configuration when added, but contains parameters for the specific versions of each of the tools that are being installed.

To stay up to date with the tool versions over time, make sure these parameters correspond to the versions of the tools that are required for the pipeline to run properly.

YAML snippet (Installer)

```
# Installs default Power Platform Build Tools
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.tool-installer.PowerPlatformToolInstaller@0
  displayName: 'Power Platform Tool Installer'
```

```
# Installs specific versions of the Power Platform Build Tools
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.tool-installer.PowerPlatformToolInstaller@0
  displayName: 'Power Platform Tool Installer'
  inputs:
    DefaultVersion: false
    XrmToolingPackageDeploymentVersion: 3.3.0.928
```

Parameters (Installer)

PARAMETERS	DESCRIPTION
`DefaultVersion` Use default tool versions	Set to **true** to use the default version of all tools, otherwise **false**. Required (and **false**) when any tool versions are specified.

`PowerAppsAdminVersion`
`XrmToolingPackageDeploymentVersion`
`MicrosoftPowerAppsCheckerVersion`
`CrmSdkCoreToolsVersion`
Tool version
	The specific version of the tool to use.

Power Platform WhoAmI

Verifies a Power Platform environment service connection by connecting and making a WhoAmI request. This task can be useful to include early in the pipeline, to verify connectivity before processing begins.

YAML snippet (WhoAmI)

```
# Verifies an environment service connection
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.whoami.PowerPlatformWhoAmi@0
  displayName: 'Power Platform WhoAmI'

  inputs:
#   Service Principal/client secret (supports MFA)
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'My service connection'
```

```
# Verifies an environment service connection
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.whoami.PowerPlatformWhoAmi@0
  displayName: 'Power Platform WhoAmI'

  inputs:
#   Username/password (no MFA support)
    PowerPlatformEnvironment: 'My service connection'
```

Parameters (WhoAmI)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Optional) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint for the environment to connect to. Defined under **Service Connections** in **Project Settings**. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint for the environment to connect to. Defined under **Service Connections** in **Project Settings**.

Quality check

Below are the available tasks for checking the quality of a solution.

Power Platform Checker

This task runs a static analysis check on your solutions against a set of best-practice rules to identify any problematic patterns that you might have inadvertently introduced when building your solution.

YAML snippet (Checker)

```
# Static analysis check of your solution
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.checker.PowerPlatformChecker@0
  displayName: 'Power Platform Checker '
  inputs:
    PowerPlatformSPN: 'Dataverse service connection'
    RuleSet: '0ad12346-e108-40b8-a956-9a8f95ea18c9'
```

```
# Static analysis check of your solution
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.checker.PowerPlatformChecker@0
  displayName: 'Power Platform Checker '
  inputs:
    PowerPlatformSPN: 'Dataverse service connection'
    UseDefaultPACheckerEndpoint: false
    CustomPACheckerEndpoint: 'https://japan.api.advisor.powerapps.com/'
    FileLocation: sasUriFile
    FilesToAnalyzeSasUri: 'SAS URI'
    FilesToAnalyze: '**\*.zip'
    FilesToExclude: '**\*.tzip'
    RulesToOverride: 'JSON array'
    RuleSet: '0ad12346-e108-40b8-a956-9a8f95ea18c9'
```

Parameters (Checker)

PARAMETERS	DESCRIPTION
`PowerPlatformSPN` Service Connection	(Required) A connection to a licensed Microsoft Power Platform environment is required to use the Power Platform checker. Service connections are defined in **Service Connections** under **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment Note: Service Principal is the only authentication method available for the checker task so if you are using username/password for all other tasks, you will have to create a separate connection to use with the checker task. For more information on how to configure service principals to be used with this task, see Configure service principal connections for Power Platform environments.
`UseDefaultPACheckerEndpoint` Use default Power Platform Checker endpoint	By default (**true**), the geographic location of the checker service will use the same geography as the environment you connect to.
`CustomPACheckerEndpoint` Custom PAC checker endpoint	Required when `UseDefaultPACheckerEndpoint` is **false**. You have an option to specify another geo to use, for example https://japan.api.advisor.powerapps.com. For a list of available geographies, see Use the Power Platform Checker API.
`FileLocation` Location of file(s) to analyze	Required when referencing a file from a shared access signature (SAS) URL `sasUriFile`. Note: It is important to reference an exported solution file and not the unpacked source files in your repository. Both managed and unmanaged solution files can be analyzed.
`FilesToAnalyzeSasUri` SAS files to analyze	Required when `FileLocation` is set to `sasUriFile`. Enter the SAS URI. You can add more than one SAS URI through a comma (,) or semi-colon (;) separated list.
`FilesToAnalyze` Local files to analyze	Required when SAS files are not analyzed. Specify the path and file name of the zip files to analyze. Wildcards can be used. For example, enter ***.zip for all zip files in all subfolders.

PARAMETERS	DESCRIPTION
`FilesToExclude` Local files to exclude	Specify the names of files to be excluded from the analysis. If more than one, provide through a comma (,) or semi-colon (;) separated list. This list can include a full file name or a name with leading or trailing wildcards, such as *jquery or form.js
`RulesToOverride` Rules to override	A JSON array containing rules and levels to override. Accepted values for OverrideLevel are: Critical, High, Medium, Low, Informational. Example: [{"Id":"meta-remove-dup-reg","OverrideLevel":"Medium"},{"Id":"il-avoid-specialized-update-ops","OverrideLevel":"Medium"}]
`RuleSet` Rule set	(Required) Specify which rule set to apply. The following two rule sets are available: • Solution checker: This is the same rule set that is run from the Power Apps maker portal. • AppSource: This is the extended rule set that is used to certify an application before it can be published to AppSource.
`ErrorLevel` Error Level	Combined with the error threshold parameter defines the severity of errors and warnings that are allowed. Supported threshold values are <level>IssueCount where level=Critical, High, Medium, Low, and Informational.
`ErrorThreshold` Error threshold	Defines the number of errors (>=0) of a specified level that are allowed for the checker to pass the solutions being checked.
`FailOnPowerAppsCheckerAnalysisError` Fail on error	When **true**, fail if the Power Apps Checker analysis is returned as Failed or FinishedWithErrors.
`ArtifactDestinationName` DevOps artifact name	Specify the Azure DevOps artifacts name for the checker .sarif file.

Solution tasks

This set of tasks can automate solution actions. The environment tasks outlined later in this section that create, copy, or restore an environment will overwrite the service connections with the newly created environments. This makes it possible to perform solution tasks against environments that are created on demand.

Power Platform Import Solution

Imports a solution into a target environment.

YAML snippet (Import)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.import-solution.PowerPlatformImportSolution@0
  displayName: 'Power Platform Import Solution '
  inputs:
    PowerPlatformEnvironment: 'My service connection'
    SolutionInputFile: 'C:\Public\Contoso_sample_1_0_0_1_managed.zip'
    HoldingSolution: true
    OverwriteUnmanagedCustomizations: true
    SkipProductUpdateDependencies: true
    ConvertToManaged: true
```

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.import-solution.PowerPlatformImportSolution@0
  displayName: 'Power Platform Import Solution '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
    SolutionInputFile: 'C:\Public\Contoso_sample_1_0_0_1_managed.zip'
    AsyncOperation: true
    MaxAsyncWaitTime: 60
    PublishWorkflows: false
```

Parameters (Import)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to import the solution into (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to import the solution into (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`SolutionInputFile` Solution input file	(Required) The path and file name of the solution .zip file to import into the target environment (e.g., $(Build.ArtifactStagingDirectory)$(SolutionName).zip). Note: Variables give you a convenient way to get key bits of data into various parts of your pipeline. See Use predefined variables for a comprehensive list.
`HoldingSolution` Import as a holding solution	An advance parameter (true|false) used when a solution needs to be upgraded. This parameter hosts the solution in Dataverse but does not upgrade the solution until the Apply Solution Upgrade task is run.
`OverwriteUnmanagedCustomizations` Overwrite un-managed customizations	Specify whether to overwrite un-managed customizations (true|false).

PARAMETERS	DESCRIPTION
`SkipProductUpdateDependencies` Skip product update dependencies	Specify whether the enforcement of dependencies related to product updates should be skipped (true|false).
`ConvertToManaged` Convert to managed	Specify whether to import as a managed solution (true|false).
`AsyncOperation` Asynchronous import	If selected (**true**), the import operation will be performed asynchronously. This is recommended for larger solutions as this task will automatically timeout after 4 minutes otherwise. Selecting asynchronous will poll and wait until MaxAsyncWaitTime has been reached (true|false).
`MaxAsyncWaitTime` Maximum wait time	Maximum wait time in minutes for the asynchronous operation; default is 60 min (1 hr), same as Azure DevOps default for tasks.
`PublishWorkflows` Activate processes after import	Specify whether any processes (workflows) in the solution should be activated after import (true|false).
`UseDeploymentSettingsFile` Use deployment settings file	Connection references and environment variable values can be set using a deployment settings file (true|false).
`DeploymentSettingsFile` Deployment settings file	(Required when `UseDeploymentSettingsFile` =**true**) The path and file name of the deployment settings file.

Power Platform Apply Solution Upgrade

Upgrades a solution that has been imported as a holding solution.

YAML snippet (Upgrade)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.apply-solution-
upgrade.PowerPlatformApplySolutionUpgrade@0
  displayName: 'Power Platform Apply Solution Upgrade '
  inputs:
    PowerPlatformEnvironment: 'My service connection'
    SolutionName: 'Contoso_sample'
    AsyncOperation: false
```

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.apply-solution-
upgrade.PowerPlatformApplySolutionUpgrade@0
  displayName: 'Power Platform Apply Solution Upgrade '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
    SolutionName: 'Contoso_sample'
    MaxAsyncWaitTime: 45
```

Parameters (Upgrade)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to upgrade the solution into (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to upgrade the solution into (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`SolutionName` Solution name	(Required) The name of the solution to apply the upgrade. Always use the solution *Name* not its *Display Name*.
`AsyncOperation` Asynchronous upgrade	If selected (**true**), the upgrade operation will be performed as an asynchronous batch job. Selecting asynchronous will poll and wait until MaxAsyncWaitTime has been reached.
`MaxAsyncWaitTime` Maximum wait time	Maximum wait time in minutes for the asynchronous operation; default is 60 min (1 hr), same as Azure DevOps default for tasks.

> **NOTE**
>
> Variables give you a convenient way to get key bits of data into various parts of your pipeline. See Use predefined variables for a comprehensive list. You can pre-populate connection reference and environment variables information for the target environment while importing a solution using a deployment settings file.
>
> More information: Pre-populate connection references and environment variables for automated deployments

Power Platform Export Solution

Exports a solution from a source environment.

YAML snippet (Export)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.export-solution.PowerPlatformExportSolution@0
  displayName: 'Power Platform Export Solution '
  inputs:
    PowerPlatformEnvironment: 'My service connection'
    SolutionName: 'Contoso_sample'
    SolutionOutputFile: 'C:\Public\Contoso_sample_1_0_0_1_managed.zip'
    Managed: true
    MaxAsyncWaitTime: 120
```

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.export-solution.PowerPlatformExportSolution@0
  displayName: 'Power Platform Export Solution '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
    SolutionName: 'Contoso_sample'
    SolutionOutputFile: 'C:\Public\Contoso_sample_1_0_0_1_managed.zip'
    Managed: true
    MaxAsyncWaitTime: 120
    ExportAutoNumberingSettings: true
    ExportCalendarSettings: true
    ExportCustomizationSettings: true
    ExportEmailTrackingSettings: true
    ExportGeneralSettings: true
    ExportIsvConfig: true
    ExportMarketingSettings: true
    ExportOutlookSynchronizationSettings: true
    ExportRelationshipRoles: true
    ExportSales: true
```

Parameters (Export)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to upgrade the solution into (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to upgrade the solution into (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`SolutionName` Solution name	(Required) The name of the solution to export. Always use the solution *Name* not its *Display Name*.
`SolutionOutputFile` Solution output file	(Required) The path and file name of the solution.zip file to export the source environment to (e.g., $(Build.ArtifactStagingDirectory)$(SolutionName).zip). Note: Variables give you a convenient way to get key bits of data into various parts of your pipeline. See Use predefined variables for a comprehensive list.
`AsyncOperation` Asynchronous export	If selected (**true**), the export operation will be performed as an asynchronous batch job. Selecting asynchronous will poll and wait until MaxAsyncWaitTime has been reached.

PARAMETERS	DESCRIPTION
`MaxAsyncWaitTime` Maximum wait time	Maximum wait time in minutes for the asynchronous operation; default is 60 min (1 hr), same as Azure DevOps default for tasks.
`Managed` Export as managed	If selected (**true**), export the solution as a managed solution; otherwise export as an unmanaged solution.
`ExportAutoNumberingSettings` Export auto-numbering settings	Export auto-numbering settings (true|false).
`ExportCalendarSettings` Export calendar settings	Export calendar settings (true|false).
`ExportCustomizationSettings` Export customization settings	Export customization settings (true|false).
`ExportEmailTrackingSettings` Export email tracking settings	Export email tracking settings (true|false).
`ExportGeneralSettings` Export general settings	Export general settings (true|false).
`ExportIsvConfig` Export ISV configuration	Export ISV configuration (true|false).
`ExportMarketingSettings` Export marketing settings	Export marketing settings (true|false).
`ExportOutlookSynchronizationSettings` Export Outlook sync settings	Export Outlook synchronization settings (true|false).
`ExportRelationshipRoles` Export relationship roles	Export relationship roles (true|false).
`ExportSales` Exports sales	Exports sales (true|false).

Power Platform Unpack Solution

Takes a compressed solution file and decomposes it into multiple XML files so that these files can be more easily read and managed by a source control system.

YAML snippet (Unpack)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.unpack-solution.PowerPlatformUnpackSolution@0
  displayName: 'Power Platform Unpack Solution '
  inputs:
    SolutionInputFile: 'C:\Public\Contoso_sample_1_0_0_1_managed.zip'
    SolutionTargetFolder: 'C:\Public'
    SolutionType: Both
```

Parameters (Unpack)

PARAMETERS	DESCRIPTION
`SolutionInputFile` Solution input file	(Required) The path and file name of the solution.zip file to unpack.
`SolutionTargetFolder` Target folder to unpack solution	(Required) The path and target folder you want to unpack the solution into.
`SolutionType` Type of solution	(Required) The type of solution you want to unpack. Options include: **Unmanaged** (recommended), **Managed**, and **Both**.

Power Platform Pack Solution

Packs a solution represented in source control into a solution.zip file that can be imported into another environment.

YAML snippet (Pack)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.pack-solution.PowerPlatformPackSolution@0
  displayName: 'Power Platform Pack Solution '
  inputs:
    SolutionSourceFolder: 'C:\Public'
    SolutionOutputFile: 'Contoso_sample_1_0_0_1_managed.zip'
    SolutionType: Managed
```

Parameters (Pack)

PARAMETERS	DESCRIPTION
`SolutionOutputFile` Solution output file	(Required) The path and file name of the solution.zip file to pack the solution into.
`SolutionSourceFolder` Source folder of solution to pack	(Required) The path and source folder of the solution to pack.
`SolutionType` Type of solution	(Required) The type of solution you want to pack. Options include: **Managed** (recommended), **Unmanaged**, and **Both**.

Power Platform Delete Solution

Deletes a solution in the target environment.

YAML snippet (Delete)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.delete-solution.PowerPlatformDeleteSolution@0
  displayName: 'Power Platform Delete Solution '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
    SolutionName: 'Contoso_sample'
```

Parameters (Delete)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to delete the solution (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to delete the solution (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`SolutionName` Solution name	(Required) The name of the solution to delete. Always use the solution *Name* not its *Display Name*.

Power Platform Publish Customizations

Publishes all customizations in an environment.

YAML snippet (Publish)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.publish-
customizations.PowerPlatformPublishCustomizations@0
  displayName: 'Power Platform Publish Customizations '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
```

Parameters (Publish)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to publish the customizations (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment

PARAMETERS	DESCRIPTION
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to publish the customizations (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment

Power Platform Set Solution Version

Updates the version of a solution.

YAML snippet (Version)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.set-solution-
version.PowerPlatformSetSolutionVersion@0
  displayName: 'Power Platform Set Solution Version '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
    SolutionName: 'Contoso_sample'
    SolutionVersionNumber: 1.0.0.0
```

Parameters (Version)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to set the solution version (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to set the solution version (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`SolutionName` Solution name	(Required) The name of the solution to set the version for. Always use the solution *Name* not its *Display Name*.
`SolutionVersionNumber` Solution version number	(Required) Version number you want to set.

While version number can be hardcoded in the pipeline, it is recommended to use an Azure DevOps pipeline variable like BuildId. This provides options to define the exact shape of version number under the "Options" tab, for example: $(Year:yyyy)-$(Month:MM)-$(Day:dd)-$(rev:rr)-3

This definition can then be used in the Set Solution Version task by setting the Version Number property with: $(Build.BuildId) instead of hard coding 20200824.0.0.2.

Alternatively a powershell inline task script $(Get-Date -Format yyyy.MM.dd.HHmm) output set to empty variable named SolutionVersion as Write-Host ("##vso[task.setvariable variable=SolutionVersion;]$version"), Set Solution Version as $(SolutionVersion).

Power Platform Deploy Package

Deploys a package to an environment. Deploying a package as opposed to a single solution file provides an option to deploy multiple solutions, data, and code into an environment.

YAML snippet (Deploy)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.deploy-package.PowerPlatformDeployPackage@0
  displayName: 'Power Platform Deploy Package '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
    PackageFile: 'C:\Users\Public\package.dll'
    MaxAsyncWaitTime: 120
```

Parameters (Deploy)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to deploy the package into (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to deploy the package into (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type. More information: see `BuildTools.EnvironmentUrl` under Power Platform Create Environment
`PackageFile` Package file	(Required) The path and file name of the package file assembly (.dll).
`MaxAsyncWaitTime` Maximum wait time	Maximum wait time in minutes for the asynchronous operation; default is 60 min (1 hr), same as Azure DevOps default for tasks.

Environment management tasks

Automate common Environment Lifecycle Management (ELM) tasks.

Power Platform Create Environment

Creates a new environment. Creating a new environment also automatically creates `BuildTools.EnvironmentUrl`.

IMPORTANT

When set, `BuildTools.EnvironmentUrl` will be used as the **default service connection** for subsequent tasks in the pipeline. Each task described in this article only uses the endpoint from the service connection when `BuildTools.EnvironmentUrl` is not set.

A new environment can only be provisioned if your license or capacity allows for the creation of additional environments. For more information on how to view capacity see Capacity page details.

YAML snippet (Create-env)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.create-environment.PowerPlatformCreateEnvironment@0
  displayName: 'Power Platform Create Environment '
  inputs:
    PowerPlatformEnvironment: 'My service connection'
    DisplayName: 'First Coffee test'
    DomainName: firstcoffee
```

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.create-environment.PowerPlatformCreateEnvironment@0
  displayName: 'Power Platform Create Environment '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
    DisplayName: 'First Coffee prod'
    EnvironmentSku: Production
    AppsTemplate: 'D365_CustomerService,D365_FieldService'
    LocationName: canada
    LanguageName: 1036
    CurrencyName: ALL
    DomainName: firstcoffee
```

Parameters (Create-env)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to create the environment (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type.
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to create the environment (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type.
`DisplayName` Display name	(Required) The display name of the environment created.
`LocationName` Deployment region	(Required) The region that the environment should be created in.

PARAMETERS	DESCRIPTION
`EnvironmentSku` Environment type	(Required) The type of instance to deploy. Options are **Sandbox**, **Production**, **Trial**, and **SubscriptionBasedTrial**.
`AppsTemplate` Apps	For a non-trial environment type, the supported apps are D365_CustomerService, D365_FieldService, D365_ProjectServiceAutomation, and D365_Sales.
`CurrencyName` Currency	(Required) Base currency for the environment created. The currency cannot be updated after the environment is created.
`LanguageName` Language	(Required) The base language in the environment.
`DomainName` Domain Name	(Required) This is the environment-specific string that forms part of the URL. For example, for an environment with the following URL: https://powerappsbuildtasks.crm.dynamics.com, the domain name would be 'powerappsbuildtasks'. Note: If you enter a domain name that's already in use, the task appends a numeric value to the domain name, starting with 0. For the example above, the URL might become https://powerappsbuildtasks0.crm.dynamics.com.

Power Platform Delete Environment

Deletes an environment.

YAML snippet (Delete-env)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.delete-environment.PowerPlatformDeleteEnvironment@0
  displayName: 'Power Platform Delete Environment '
  inputs:
    PowerPlatformEnvironment: 'My service connection'
```

Parameters (Delete-env)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to delete the environment (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type.
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to delete the environment (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type.

Power Platform Backup Environment

Backs up an environment.

YAML snippet (Backup-env)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.backup-environment.PowerPlatformBackupEnvironment@0
  displayName: 'Power Platform Backup Environment '
  inputs:
    PowerPlatformEnvironment: 'My service connection'
    BackupLabel: 'Full backup - $(Build.BuildNumber)'
```

Parameters (Backup-env)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint that you want to back up the environment (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type.
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint that you want to back up the environment (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type.
`BackupLabel` Backup label	(Required) The label to be assigned to the backup.

Power Platform Copy Environment

Copies an environment to a target environment. Two types of copies are available: full and minimal. A *Full* copy includes both data and solution metadata (customizations), whereas a *minimal* copy only includes solution metadata and not the actual data.

YAML snippet (Copy-env)

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.copy-environment.PowerPlatformCopyEnvironment@0
  displayName: 'Power Platform Copy Environment '
  inputs:
    PowerPlatformEnvironment: 'My service connection'
    TargetEnvironmentUrl: 'https://contoso-test.crm.dynamics.com'
```

```
steps:
- task: microsoft-IsvExpTools.PowerPlatform-BuildTools.copy-environment.PowerPlatformCopyEnvironment@0
  displayName: 'Power Platform Copy Environment '
  inputs:
    authenticationType: PowerPlatformSPN
    PowerPlatformSPN: 'Dataverse service connection '
    TargetEnvironmentUrl: 'https://contoso-test.crm.dynamics.com'
    CopyType: MinimalCopy
    OverrideFriendlyName: true
    FriendlyName: 'Contoso Test'
    DisableAdminMode: false
```

Parameters (Copy-env)

PARAMETERS	DESCRIPTION
`authenticationType` Type of authentication	(Required for SPN) Specify either **PowerPlatformEnvironment** for a username/password connection or **PowerPlatformSPN** for a Service Principal/client secret connection.
`PowerPlatformEnvironment` Power Platform environment URL	The service endpoint for the source environment that you want to copy from (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type.
`PowerPlatformSPN` Power Platform Service Principal	The service endpoint for the source environment that you want to copy from (e.g., https://powerappsbuildtools.crm.dynamics.com). Defined under **Service Connections** in **Project Settings** using the **Power Platform** connection type.
`TargetEnvironmentUrl` Target environment URL	(Required) The URL for the target environment that you want to copy to.
`CopyType` Copy type	The type of copy to perform: FullCopy or MinimalCopy
`OverrideFriendlyName` Override friendly name	Change the target environment's friendly name to another name (true|false).
`FriendlyName` Friendly name	The friendly name of the target environment.
`DisableAdminMode` Disable admin mode	Whether to disable administration mode (true|false).

Build and release pipelines

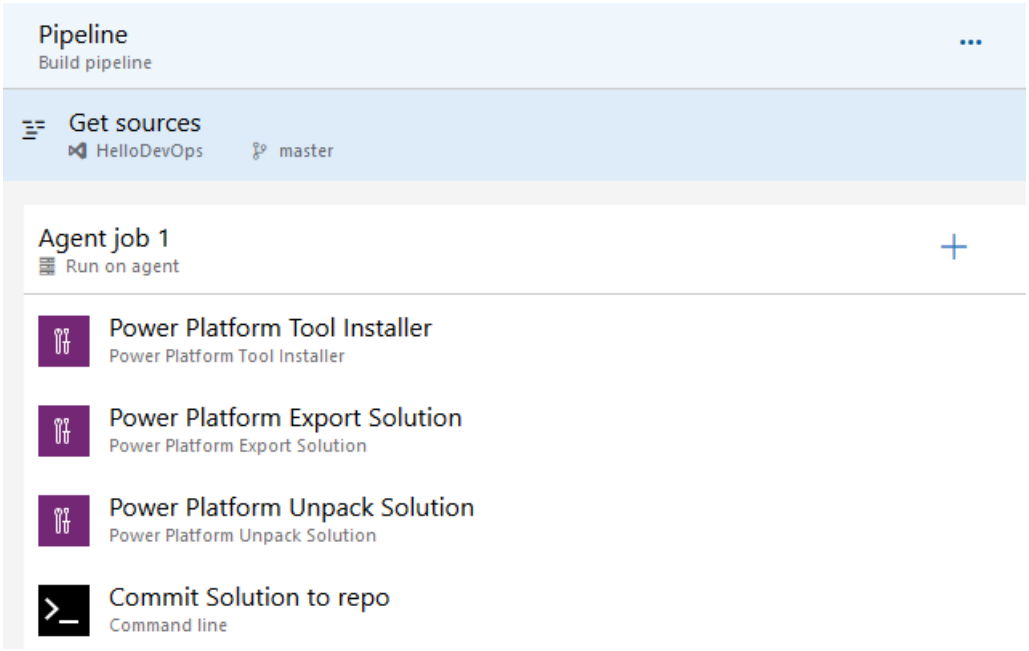
Now that we've identified what can be done using the build tools, let's see how you might apply these tools to your build and release pipelines. A conceptual overview is shown below. Let's view some details of the pipeline implementation using the build tool tasks in the sub-sections that follow.

To learn more about creating these pipelines and actually do hands-on pipeline authoring using the Microsoft Power Platform Build Tools, complete the build tools labs, which you can download from GitHub.

More information about Azure DevOps pipelines: Use Azure Pipelines

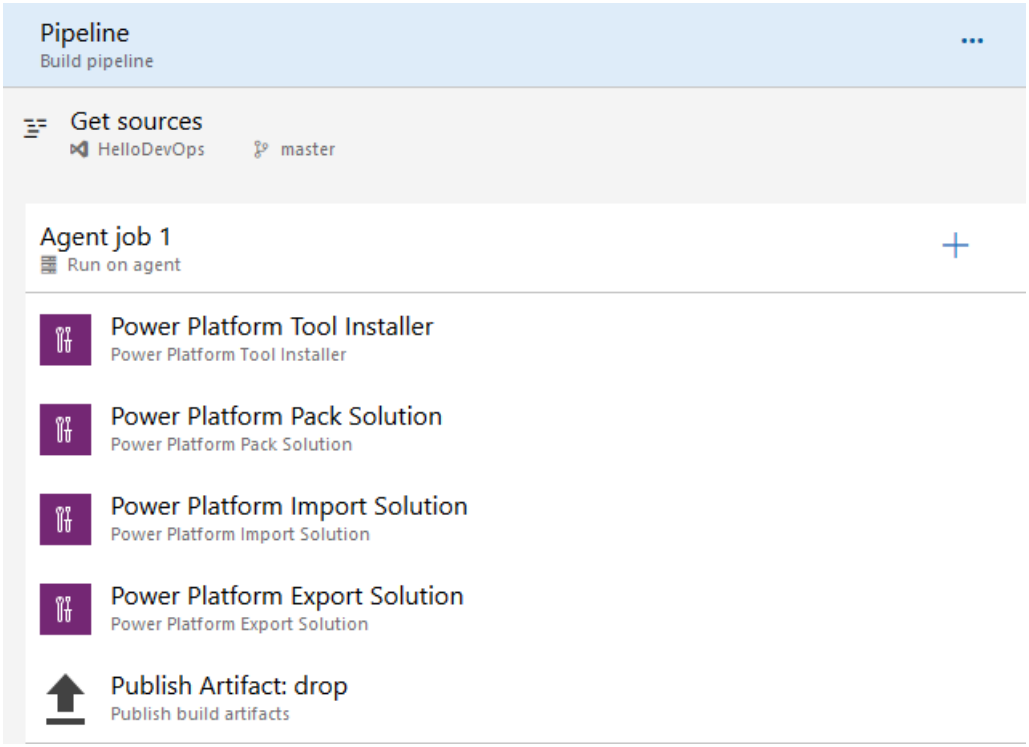
Build pipeline: Export a solution from a development environment (DEV)

The following figure shows the build tool tasks that you might add to a pipeline that exports a solution from a development environment.



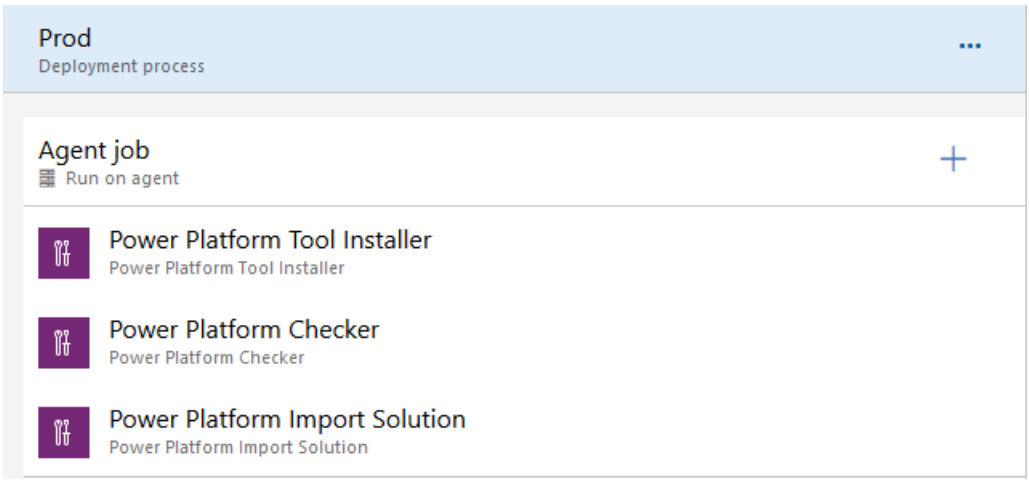
Build pipeline: Build a managed solution

The following figure shows the build tool tasks that you might add to a pipeline that builds a managed solution.



Release pipeline: Deploy to a production environment (PROD)

The following figure shows the build tool tasks that you might add to a pipeline that deploys to a production environment.



See Also

Microsoft Power Platform Build Tools for Azure DevOps

Pre-populate connection references and environment variables for automated deployments

7/15/2022 • 4 minutes to read • Edit Online

Connection references and *environment variables* enable you to interactively specify the connection details and configuration settings specific to the target environment where your app or solution is deployed.

More information:

- Connection reference overview
- Environment variables overview

After importing a solution containing connection reference and environment variable information, you are prompted to provide information specific to your environment in the UI. However, entering this information does not work well for fully automated Continuous Integration/ Continuous Delivery (CI/CD) scenarios.

To enable a fully automated deployment, you can now pre-populate the connection reference and environment variable information specific to the target environment so that you don't have to interactively provide it after importing a solution.

Deployment settings file

To pre-populate the connection reference and environment variable information for your deployment, use the deployment settings file (JSON) to store the information, and pass it as a parameter when importing the solution using Power Platform Build Tools. You can store the JSON file in your source control system to update and manage as required for your organization.

Below is an example of the deployment settings file:

```
{
  "EnvironmentVariables": [
    {
      "SchemaName": "tst_Deployment_env",
      "Value": ""
    },
    {
      "SchemaName": "tst_EnvironmentType",
      "Value": ""
    }
  ],
  "ConnectionReferences": [
    {
      "LogicalName": "tst_sharedtst5fcreateuserandjob5ffeb85c4c63870282_b4cc7",
      "ConnectionId": "",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_tst-5fcreateuserandjob-5ff805fab2693f57dc"
    },
    {
      "LogicalName": "tst_SharepointSiteURL",
      "ConnectionId": "",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_sharepointonline"
    },
    {
      "LogicalName": "tst_AzureDevopsConnRef",
      "ConnectionId": "",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_visualstudioteamservices"
    },
    {
      "LogicalName": "tst_GHConn",
      "ConnectionId": "",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_github"
    }
  ]
}
```

In the above example JSON file, the values shown as "" are missing and will need to be populated. We will address doing that later in this article.

Step 1: Generate the deployment settings file

The deployment setting file can be generated using the Power Platform CLI. You can generate the file while exporting or cloning the solution.

Option 1: Generate deployment settings file using create-settings property

Use the `create-settings` property with Power Platform CLI:

```
C:\> pac solution create-settings --solution-zip <solution_zip_file_path> --settings-file
<settings_file_name>
```

This command will generate a JSON file as shown below.

```
PS C:\Users\kartikka\Documents\GitHub\conn-reff-demo> pac solution create-settings --solution-zip .\mytestsoln.zip --setting
s-file DeploymentSettingsDev.json
Extracting Dataverse connection references and/or environment variables from: C:\Users\kartikka\Documents\GitHub\conn-reff-d
emo\mytestsoln.zip
Deployment settings file created: C:\Users\kartikka\Documents\GitHub\conn-reff-demo\DeploymentSettingsDev.json
PS C:\Users\kartikka\Documents\GitHub\conn-reff-demo>
```

In the JSON file, few values are empty in the `ConnectionReferences` section. These values need to be gathered after creating them in the target environment.

```
{
  "EnvironmentVariables": [
    {
      "SchemaName": "tyt_Deploymentenv",
      "Value": ""
    },
    {
      "SchemaName": "tyt_Deploymentsite",
      "Value": ""
    }
  ],
  "ConnectionReferences": [
    {
      "LogicalName": "tyt_AzureDevopsConnRef",
      "ConnectionId": "",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_visualstudioteamservices"
    },
    {
      "LogicalName": "tyt_GHConn",
      "ConnectionId": "",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_github"
    }
  ]
}
```

Option 2: Generate deployment settings file by cloning the solution

A project solution clone is needed for this step because it renders the original solution into a buildable format. After you have cloned a solution using Power Platform CLI, you get the following directory structure created on your computer:

```
PS C:\Users\kartikka\Documents\GitHub\conn-reff-demo> pac solution list
Connected to...mytestenv
Listing all Solutions from the current Dataverse Organization...

 Index      Unique Name                                        Friendly Name
Version

 [1]        Cr0f8cf                                            Common Data Services Default Solution
1.0.0.0
 [2]        mytestsoln                                         mytestsoln
1.0.0.0
 [3]        [redacted]VehicleQualitySolution                   Vehicle Quality Solution
1.3.0.0

PS C:\Users\kartikka\Documents\GitHub\conn-reff-demo> pac solution clone --name mytestsoln
```

Proceed to create the settings file in the context of the current folder and populate the value of the settings file as shown below.

```
PS C:\Users\kartikka\Documents\GitHub\conn-reff-demo> cd .\mytestsoln\
PS C:\Users\kartikka\Documents\GitHub\conn-reff-demo\mytestsoln> dir

    Directory: C:\Users\kartikka\Documents\GitHub\conn-reff-demo\mytestsoln

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          8/19/2021 10:37 AM                src
-a---          8/17/2021  8:51 AM            129 .gitignore
-a---          8/19/2021 10:37 AM           2475 mytestsoln.cdsproj

PS C:\Users\kartikka\Documents\GitHub\conn-reff-demo\mytestsoln> pac solution create-settings --solution-folder .\. --settin
gs-file .\DeploymentSettingsTest.json
Extracting Dataverse connection references and/or environment variables from: C:\Users\kartikka\Documents\GitHub\conn-reff-d
emo\mytestsoln\src
Deployment settings file created: C:\Users\kartikka\Documents\GitHub\conn-reff-demo\mytestsoln\DeploymentSettingsTest.json
PS C:\Users\kartikka\Documents\GitHub\conn-reff-demo\mytestsoln>
```

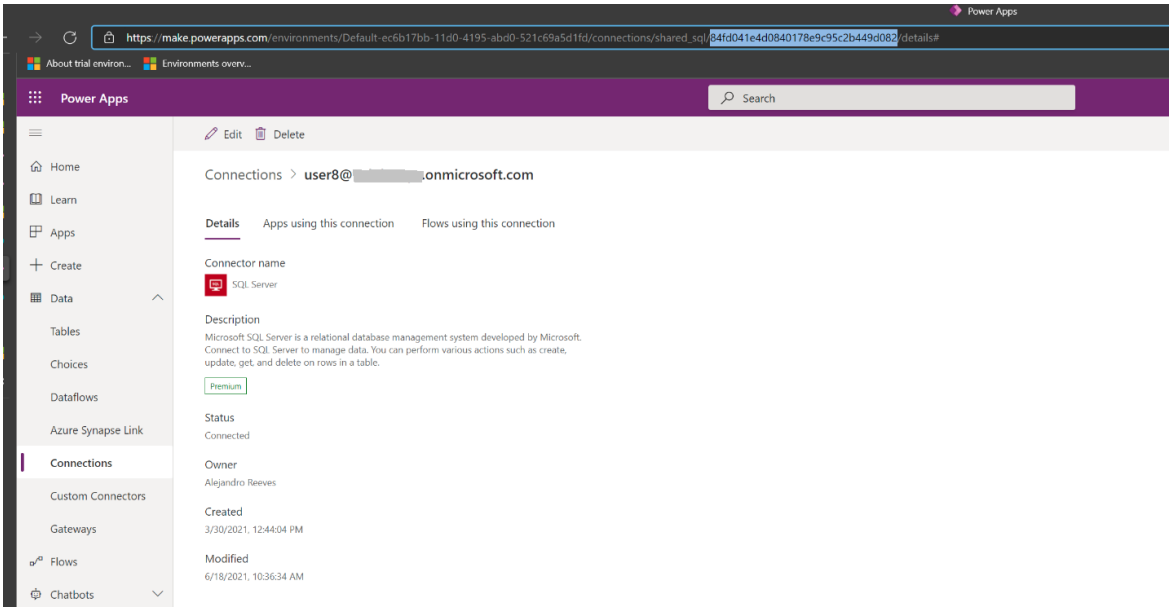
Step 2: Get the connection reference and environment variable information

To populate the deployment settings file, you will need to obtain the connection reference and environment variable information of the target solution.

Get the connection reference information

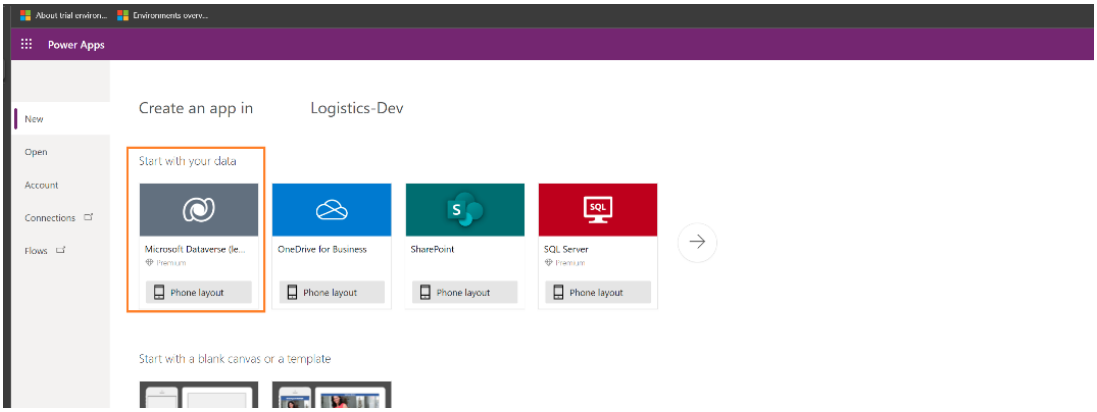
To get the connection ID of the target environment, use one of the following ways:

- Sign in to Power Apps and select your target environment. In the left navigation pane, select **Data** > **Connections**, select the connection you want to get the connection ID for and then look at the URL to get the connection ID.

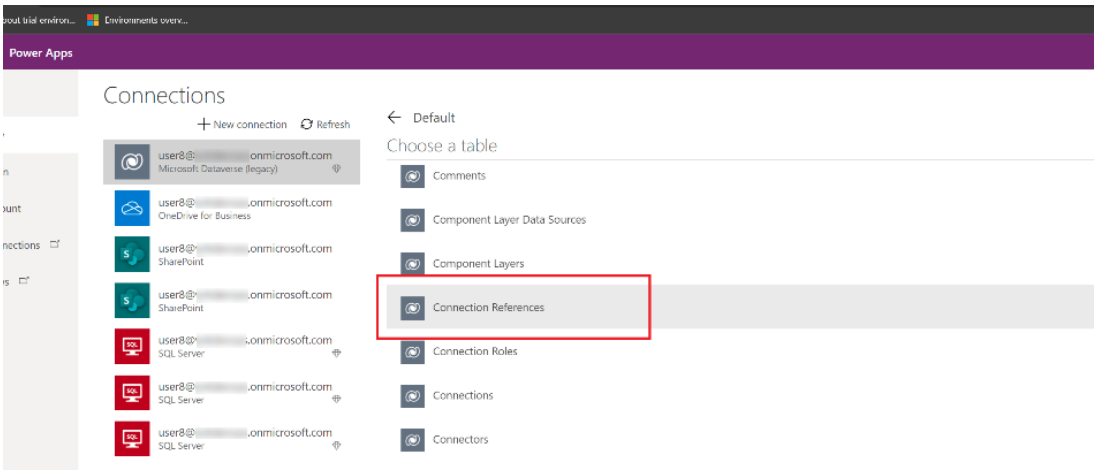


- Create a canvas app on the connection reference entity. The steps are as follows:

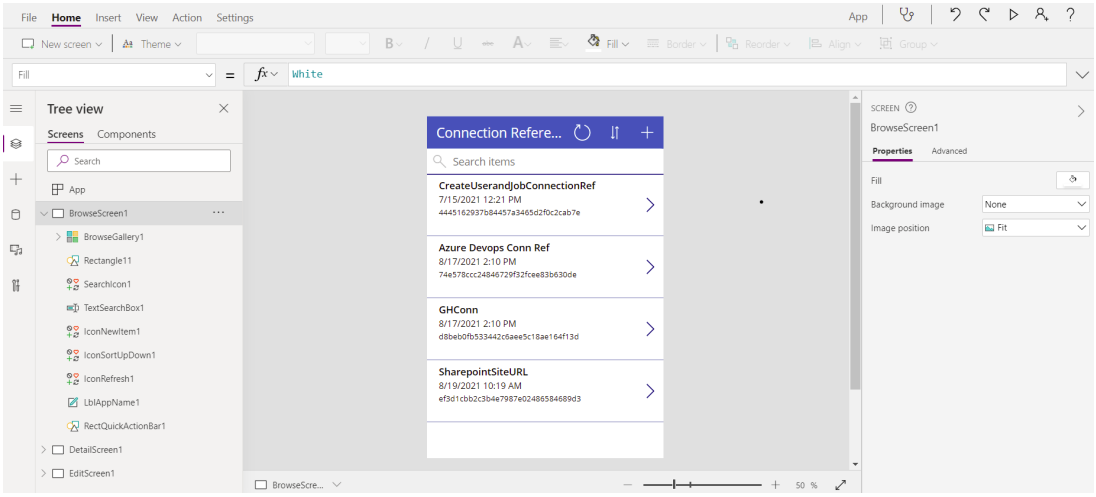
  1. Sign in to Power Apps and select your target environment.
  2. In the left navigation pane, select **Apps**, and then select **New app** > **Canvas**.
  3. Select Dataverse as your data source.



  4. Select the **Connection References** table and select **Connect**.

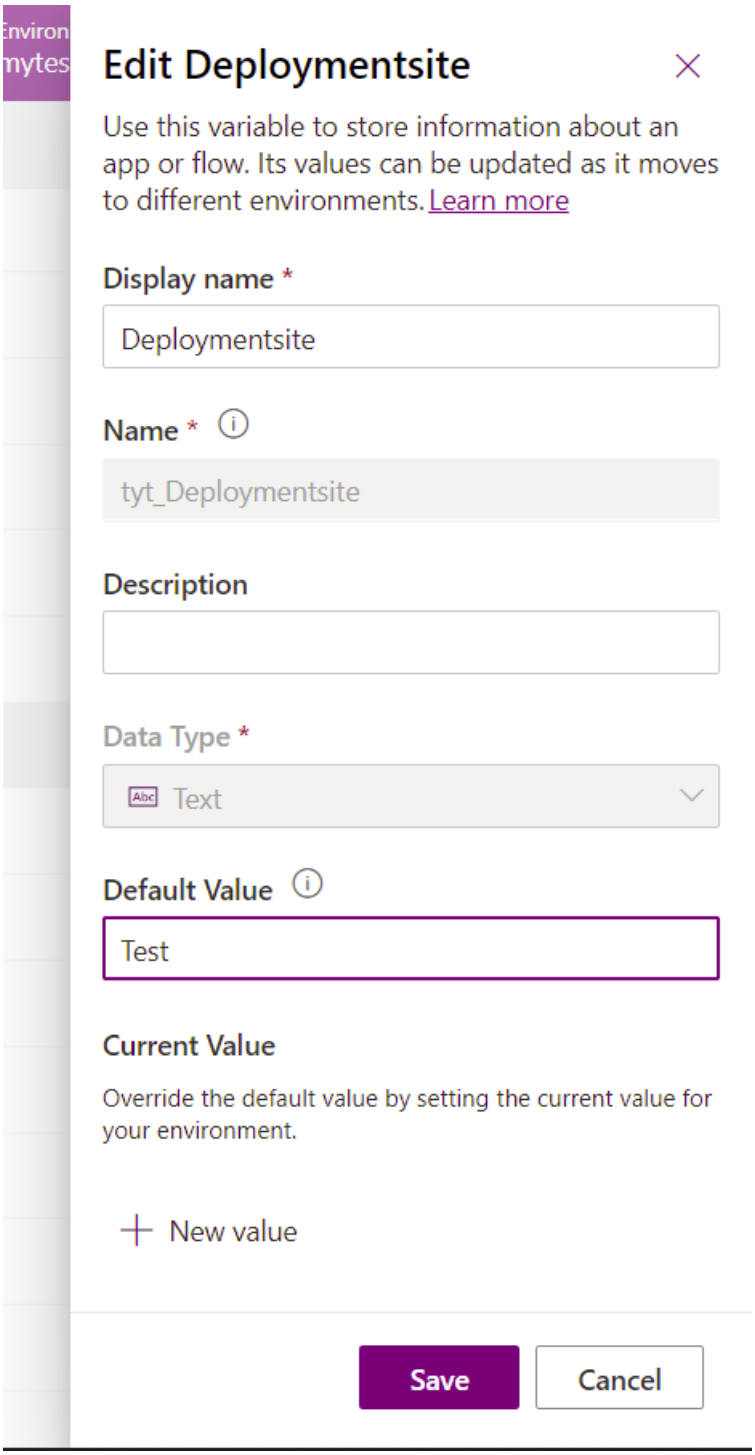


5. This will create a gallery application that will list out all the connections and their connection IDs within the environment.



Get the environment variable information

To get the values of the environment variable in the target environment, sign in to Power Apps, select the target environment, and right-click on the ellipsis and choose to edit. This will provide the information needed to populate the deployment settings file (the underlined values are the values needed for the file):



Otherwise, you can provide the appropriate value for the target environment based on your knowledge of the target environment.

Step 3: Update the values in the deployment settings file

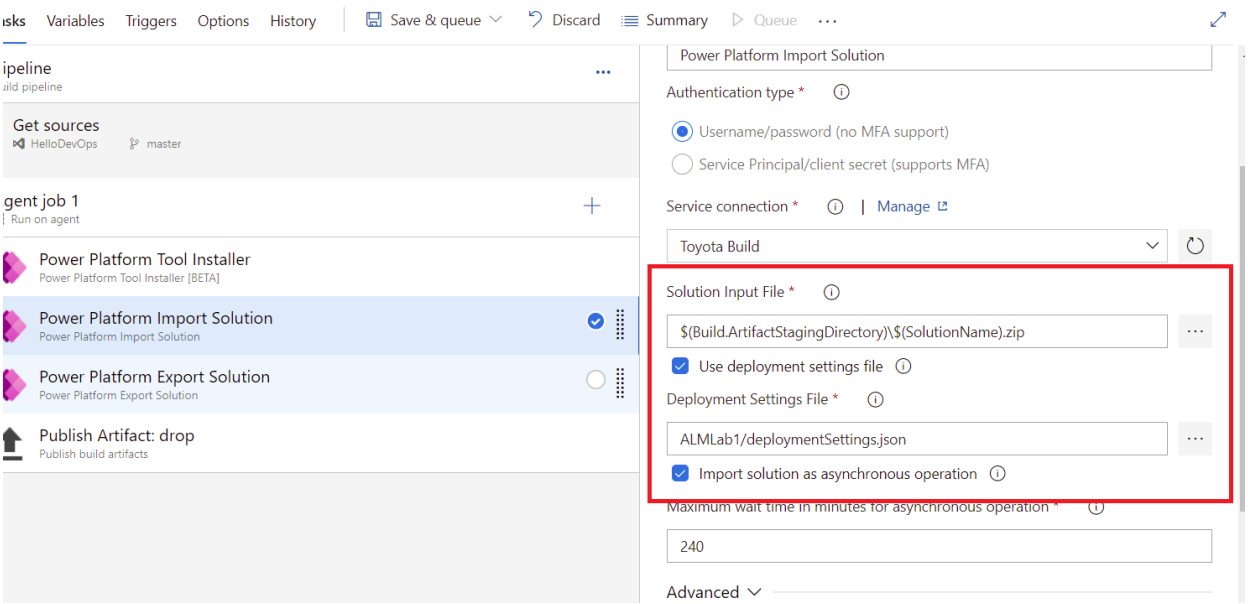
Manually edit the deployment settings file (JSON) to add the connection and environment variable information appropriately. Below is an example settings file with the (previously) missing values added.

```
{
  "EnvironmentVariables": [
    {
      "SchemaName": "tst_Deployment_env",
      "Value": "Test"
    },
    {
      "SchemaName": "tst_EnvironmentType",
      "Value": "UAT"
    }
  ],
  "ConnectionReferences": [
    {
      "LogicalName": "tst_sharedtst5fcreateuserandjob5ffeb85c4c63870282_b4cc7",
      "ConnectionId": "4445162937b84457a3465d2f0c2cab7e",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_tst-5fcreateuserandjob-5ff805fab2693f57dc"
    },
    {
      "LogicalName": "tst_SharepointSiteURL",
      "ConnectionId": "ef3d1cbb2c3b4e7987e02486584689d3",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_sharepointonline"
    },
    {
      "LogicalName": "tst_AzureDevopsConnRef",
      "ConnectionId": "74e578ccc24846729f32fcee83b630de",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_visualstudioteamservices"
    },
    {
      "LogicalName": "tst_GHConn",
      "ConnectionId": "d8beb0fb533442c6aee5c18ae164f13d",
      "ConnectorId": "/providers/Microsoft.PowerApps/apis/shared_github"
    }
  ]
}
```


Step 4: Use the deployment settings file for Build Tools tasks

Pass the deployment settings file as parameter when importing the solution into the target environment. This will create the required connection reference and environments with appropriate values upon import without having the user to interactively specify the value.

When using the Import Solution task in Power Platform Build Tools, select **Use deployment settings file** and provide the path to the deployment settings file. Next, initiate the pipeline.



See also

What is Microsoft Power Platform CLI?

Microsoft Power Platform Build Tools tasks

GitHub Actions for Microsoft Power Platform

7/15/2022 • 3 minutes to read • Edit Online

GitHub Actions enable developers to build automated software development lifecycle workflows. With GitHub Actions for Microsoft Power Platform, you can create workflows in your repository to build, test, package, release, and deploy apps; perform automation; and manage bots and other components built on Microsoft Power Platform.

GitHub Actions for Microsoft Power Platform include the following capabilities:

- Importing and exporting application metadata (also known as solutions) that contain various platform components such as canvas apps, model-driven apps, desktop flows, Power Virtual Agents chatbots, AI Builder models, customer engagement apps (Dynamics 365 Sales, Dynamics 365 Customer Service, Dynamics 365 Field Service, Dynamics 365 Marketing, and Dynamics 365 Project Service Automation), and connectors between development environments and source control.
- Deploying to downstream environments.
- Provisioning or de-provisioning environments
- Performing static analysis checks against solutions by using Power Apps solution checker.

You can use GitHub Actions for Microsoft Power Platform along with any other available GitHub Actions to compose your build and release workflows. Workflows that teams commonly put in place include provisioning development environments, exporting from a development environment to source control, generating builds, and releasing apps. GitHub Actions for Microsoft Power Platform are available at https://github.com/marketplace/actions/powerplatform-actions.

> **IMPORTANT**
> GitHub Actions for Microsoft Power Platform are supported only for a Microsoft Dataverse environment with a database. More information: Create an environment with a database

Key concepts

GitHub Actions enable you to create custom software development lifecycle workflows directly in your GitHub repository. For an overview of GitHub Actions and core concepts, review the following articles:

- About GitHub Actions
- Core concepts
- About packaging with GitHub Actions

What are GitHub Actions for Microsoft Power Platform?

GitHub Actions for Microsoft Power Platform is a collection of Microsoft Power Platform–specific GitHub Actions that eliminate the need to manually download custom tooling and scripts to manage the application lifecycle of apps built on Microsoft Power Platform. The tasks can be used individually, such as importing a solution into a downstream environment, or used together in a workflow to orchestrate a scenario such as "generate a build artifact," "deploy to test," or "harvest maker changes." The build tasks can largely be categorized into four types:

- Helper

- Quality check
- Solution
- Environment management

For more information about individual tasks, go to GitHub Actions for Microsoft Power Platform.

Get GitHub Actions for Microsoft Power Platform

You can use GitHub Actions for Microsoft Power Platform by adding the actions in your workflow definition file (.yml). Sample workflow definitions are available from the GitHub Actions lab.

Connection to environments

To interact with a Dataverse environment, a secret must be created that enables the various GitHub Actions to perform the required task. Two types of connections are available:

- Username/password: Configured as a generic service connection with username and password. Username/password authentication doesn't support multifactor authentication.
- Service principal and client secret: This connection type uses service principal–based authentication and supports multifactor authentication. Service principal authentication

Available runners

GitHub Actions for Microsoft Power Platform can run on both Microsoft Windows agents and Linux agents.

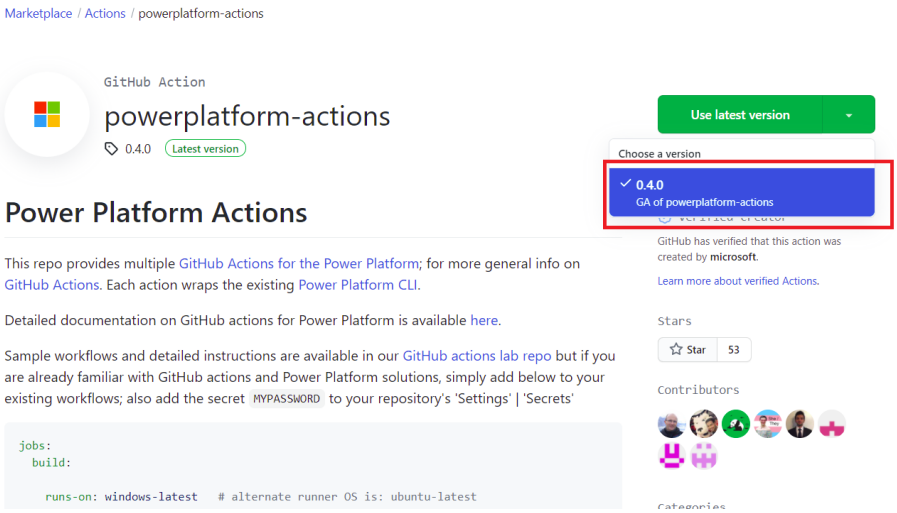
Frequently asked questions

How do I get started with GitHub Actions?

Tutorial: Get started with GitHub Actions is available right now for you try out. The tutorials show how to use service principal authentication and standard username/password authentication.

How do I get GitHub Actions for Microsoft Power Platform working in my repository?

Go to the GitHub Marketplace for Actions and search for Power Platform. When you arrive on the page, select the green button to instatiate the actions into your repository.



Do GitHub Actions only work for Power Apps?

GitHub Actions work for both canvas and model-driven apps, Power Virtual Agents, UI flows and traditional flows, AI Builder, custom connectors, and dataflows, all of which can now be added to a solution. Also included

are customer engagement apps.

Can I include flow and canvas apps?

Yes, flows and canvas apps are solution-aware, so if these components are added to your solution they can participate in the lifecycle of your app. However, some steps still require manual configuration, which will be addressed later this year when we introduce environment variables and connectors. For a list of current limitations, go to Known limitations.

How much does GitHub Actions for Microsoft Power Platform cost?

GitHub Actions are available at no cost. However, a valid GitHub subscription is required to use the actions on GitHub. To get started, 2,000 action minutes/month are available for free. More information: GitHub pricing

Can I use GitHub Actions for Microsoft Power Platform with Power Apps portals?

Yes. You can upload portal data and use the deployment profile to customize the deployment parameters.

See also

Available GitHub Actions Hands on Lab

Available GitHub Actions

Available GitHub Actions for Microsoft Power Platform development

7/15/2022 • 8 minutes to read • Edit Online

GitHub Actions for Microsoft Power Platform are described in the following sections. In addition, sample GitHub workflows shown as well. For more information about GitHub Actions and how to download them, go to GitHub Actions for Microsoft Power Platform.

Configure credentials to use with GitHub Actions with Microsoft Power Platform

Many of the actions require you to connect to a Microsoft Dataverse environment. You can add service principal or user credentials as secrets in your GitHub repository and then use them in your workflows.

- For details on how to set up secrets in GitHub, see Encrypted secrets
- For details how to set up service principal authentication for Microsoft Power Platform, see DevOps Build tools

Once configured, you can call the Service Principal from with in your Action scripts.

Parameters to define within your GitHub Action Script as Environment Variables:

- Application ID such as: `WF_APPLICATION_ID:<your application id>`
- Tenant ID such as: `WF_TENANT_ID:<your tenant id>`

The client Secret must be added and stored as a GitHub Secret, and will be referenced from within the workflow using a parameter like: `client secret: ${{secrets.CLIENT_SECRET_GITHUB_ACTIONS}}`

Helper tasks

The available helper task is described below.

whoAmI

Verifies the service connection by connecting to the service and sending a `WhoAmI` [SDK/Web API] request. This task can be useful to include early in your GitHub workflow, to verify connectivity before processing begins.

PARAMETER	DESCRIPTION
environment-url	The URL for the environment you're connecting to.
user-name	The username of the account you're using to connect with.
password-secret	The password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.

PARAMETER	DESCRIPTION
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

Solution tasks

These tasks perform actions against solutions and include the following.

import-solution

Imports a solution into a target environment.

PARAMETER	DESCRIPTION
environment-url	(Required) The URL for the target environment that you want to import the solution into (for example, https://YourOrg.crm.dynamics.com).
user-name	(Required) If you're using username/password authentication, the username of the account you're using to connect with.
password-secret	(Required) If you're using username/password authentication, the password for the account you're using to connect with.
solution-file	(Required) The path and name of the solution file you want to import.
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

export-solution

Exports a solution from a source environment.

PARAMETER	DESCRIPTION
environment-url	(Required) The URL for the environment that you want to export the solution from (for example, https://YourOrg.crm.dynamics.com).

PARAMETER	DESCRIPTION
user-name	(Required) If you're using username/password authentication, the username of the account you're using to connect with.
password-secret	(Required) If you're using username/password authentication, the password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.
solution-name	(Required) The name of the solution to export. Always use the solution's *name*, not its *display name*.
solution-output-file	(Required) The path and name of the solution.zip file to export the source environment to.
managed	(Required) Set to **true** to export as a managed solution; the default (**false**) is to export as an unmanaged solution.

unpack-solution

Takes a compressed solution file and decomposes it into multiple XML files so these files can be more easily read and managed by a source control system.

PARAMETER	DESCRIPTION
solution-file	(Required) The path and file name of the solution.zip file to unpack.
solution-folder	(Required) The path and target folder you want to unpack the solution into.
solution-type	(Required) The type of solution you want to unpack. Options include **Unmanaged** (recommended), **Managed**, and **Both**.

pack-solution

Packs a solution represented in source control into a solution.zip file that can be imported into another environment.

PARAMETER	DESCRIPTION
solution-file	(Required) The path and file name of the solution.zip file to pack the solution into (for example, out/CI/ALMLab.zip).

PARAMETER	DESCRIPTION
solution-folder	(Required) The path and source folder of the solution to pack.
solution-type	(Optional) The type of solution to pack. Options include **Unmanaged** (recommended), **Managed**, and **Both**.

publish-solution

Publishes the solution customizations.

PARAMETER	DESCRIPTION
environment-url	(Required) The URL for the environment that you want to publish the solution into (for example, https://YourOrg.crm.dynamics.com).
user-name	(Required) If you're using username/password authentication, the username of the account you're using to connect with.
password-secret	(Required) If you're using username/password authentication, the password for the account you're using to connect with.
solution-file	(Required) The path and name of the solution file you want to import.
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

clone-solution

Clones the solution for a given environment.

PARAMETER	DESCRIPTION
environment-url	(Required) The URL for the environment that you want to clone the solution from (for example, https://YourOrg.crm.dynamics.com).
user-name	(Required) If you're using username/password authentication, the username of the account you're using to connect with.
password-secret	(Required) If you're using username/password authentication, the password for the account you're using to connect with.

PARAMETER	DESCRIPTION
solution-file	(Required) The path and name of the solution file you want to import.
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.
solution-name	(Required) The path and name of the solution.zip file needed to clone (for example, out/CI/ALMLab.zip).
solution-version	Version of the solution to clone.
target-folder	Target folder to place the extracted solution into. (for example, Git repository\target-solution-folder).
Working-directory	Temporary folder for work in progress artifacts needed for cloning the solution. default: `root of the repository`

check-solution

Checks the solution file to detect inconsistencies.

PARAMETER	DESCRIPTION
environment-url	(Required) The URL for the environment that you want to clone the solution from (for example, https://YourOrg.crm.dynamics.com).
user-name	(Required) If you're using username/password authentication, the username of the account you're using to connect with.
password-secret	(Required) If you're using username/password authentication, the password for the account you're using to connect with.
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

PARAMETER	DESCRIPTION
path	(Required) The path and name of the solution file you want to check.
geo	Which geo location of the Microsoft Power Platform Checker service to use. Default value is 'united states'.
rule-level-override	Path to file a containing a JSON array of rules and their levels. Accepted values are: Critical, High, Low, and Informational. Example: [{"Id":"meta-remove-dup-reg","OverrideLevel":"Medium"},{"Id":"il-avoid-specialized-update-ops","OverrideLevel":"Medium"}]
checker-logs-artifact-name	The name of the artifact folder for which Microsoft Power Platform checker logs will be uploaded. Default value is 'CheckSolutionLogs'.

upgrade-solution

Provides the ability to upgrade the solution.

PARAMETER	DESCRIPTION
environment-url	(Required) The URL for the environment that you want to clone the solution from (for example, https://YourOrg.crm.dynamics.com).
user-name	(Required) If you're using username/password authentication, the username of the account you're using to connect with.
password-secret	(Required) If you're using username/password authentication, the password for the account you're using to connect with.
solution-file	(Required) The path and name of the solution file you want to import.
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.
solution-name	(Required) Name of the solution to upgrade.
async	Upgrades the solution asynchronously.
max-async-wait-time	Maximum asynchronous wait time in minutes. Default value is 60 minutes.

Package tasks

These tasks perform actions against packages and include the following.

deploy-package

Provides the ability to deploy a package dll or a zip file with a package.

> **NOTE**
> This action is only supported on a Windows.

PARAMETER	DESCRIPTION
environment-url	(Required) The URL for the environment that you want to clone the solution from (for example, https://YourOrg.crm.dynamics.com).
user-name	(Required) If you're using username/password authentication, the username of the account you're using to connect with.
password-secret	(Required) If you're using username/password authentication, the password for the account you're using to connect with.
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.
package	(Required) The path to the package dll or zip file with a package.

Portal tasks

These tasks perform the following actions against Power Apps portals.

upload-paportal

Uploads data to Power Apps portals.

PARAMETER	DESCRIPTION
environment-url	(Required) The URL for the environment that you want to import the solution into (for example, https://YourOrg.crm.dynamics.com).
app-id	The application ID to authenticate with. This parameter is **required** when authenticating with Service Principal credentials.

PARAMETER	DESCRIPTION
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is **required** when authenticating with Service Principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.
upload-path	Path where the website content is stored (alias: -p).
deployment-profile	Upload portal data with environment details defined through profile variables in deployment-profiles/[profile-name].deployment.yaml file.

GitHub workflow authoring

To learn more about composing GitHub workflows using GitHub actions, complete the GitHub Actions for Microsoft Power Platform labs.

More information: About GitHub Actions

See Also

GitHub Actions for Microsoft Power Platform

Available GitHub Actions for Microsoft Power Platform administration

7/15/2022 • 5 minutes to read • Edit Online

This article provides information about GitHub Actions that are available to administer Microsoft Power Platform.

Configure credentials to use within your GitHub workflows

Many of the actions require you to connect to a Microsoft Dataverse environment. You can add service principal or user credentials as secrets in your GitHub repository and then use them in the workflow.

- For details about how to set up secrets in GitHub, go to Using encrypted secrets in a workflow.
- For details about how to set up service principal authentication for Microsoft Power Platform, go to Configure service connections using a service principal. After it's configured properly, you can call the service principal from within your action scripts.

Define the following parameters within your GitHub Actions script as environment variables:

- Application ID: `WF_APPLICATION_ID:<your application id>`
- Tenant ID: `WF_TENANT_ID:<your tenant id>`

The client secret will be stored as a GitHub secret, as described in Encrypted secrets, and will be referenced from within the action script by using a parameter like `client secret: ${{secrets.CLIENT_SECRET_GITHUB_ACTIONS}}`.

Administrative tasks

The available administrative tasks are explained below.

Create an environment

PARAMETER	DESCRIPTION
user-name	The username of the account you're using to connect with.
password-secret	The password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is *required* when authenticating with service principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is *required* when authenticating with service principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.
name	The name of the environment that you're going to create.

PARAMETER	DESCRIPTION
region	The name of the region where your environment will be created. The default value is `unitedstates`.
type	The type of environment (Trial, Sandbox, Production, SubscriptionBasedTrial). More information: Trial environments
currency	The currency to use for the environment. The default value is `USD`.
language	The language to use for the environment. The default value is `English`.
templates	The templates that need to be deployed to the environment. These are passed as comma-separated values.
domain	The domain name of the environment URL. For example, `https://{contoso}0.crm.dynamics.com`

The output will be the URL of the new environment.

Copy an environment

PARAMETER	DESCRIPTION
source-url	The source URL of the environment to copy. For example, `https://source-env.crm.dynamics.com`
target-url	The target URL of the environment to copy. For example, `https://target-copy-env.crm.dynamics.com`
user-name	The username of the account you're using to connect with.
password-secret	The password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is *required* when authenticating with service principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is *required* when authenticating with service principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

Back up an environment

PARAMETER	DESCRIPTION
environment-url	The URL of the environment that needs to be backed up. For example, `https://env-to-backup.crm.dynamics.com`
backup-label	A meaningful name to use as a label for the backup of the environment.
user-name	The username of the account you're using to connect with.
password-secret	The password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is *required* when authenticating with service principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is *required* when authenticating with service principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

Restore an environment from a backup

PARAMETER	DESCRIPTION
source-url	The source URL of the environment to restore. For example, `https://env-backup-source.crm.dynamics.com`
target-url	The target URL of the environment to be restored to. For example, `https://env-target-restore.crm.dynamics.com`
user-name	The username of the account you're using to connect with.
password-secret	The password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is *required* when authenticating with service principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is *required* when authenticating with service principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

Reset an environment

PARAMETER	DESCRIPTION
environment-url	The URL of the environment that needs to be reset. For Example, `https://env-to-reset.crm.dynamics.com`
user-name	The username of the account you're using to connect with.
password-secret	The password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is *required* when authenticating with service principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is *required* when authenticating with service principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

Delete an environment

PARAMETER	DESCRIPTION
environment-url	The URL of the environment that needs to be deleted. For example, `https://env-to-delete.crm.dynamics.com`
user-name	The username of the account you're using to connect with.
password-secret	The password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is *required* when authenticating with service principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is *required* when authenticating with service principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

Helper tasks

The available helper tasks are described below.

WhoAmI

Verifies the service connection by connecting to the service and sending a `WhoAmI` [SDK/Web API] request. This task can be useful to include early in your DevOps pipeline, to verify connectivity before processing begins.

PARAMETER	DESCRIPTION
environment-url	The URL for the environment you're connecting to.
user-name	The username of the account you're using to connect with.
password-secret	The password for *user-name*. GitHub passwords are defined in **Settings** under **Secrets**. You can't retrieve a secret after it has been defined and saved.
app-id	The application ID to authenticate with. This parameter is *required* when authenticating with service principal credentials.
client-secret	The client secret used to authenticate the GitHub pipeline. This parameter is *required* when authenticating with service principal credentials.
tenant-id	The tenant ID when authenticating with `app-id` and `client-secret`.

Build and release pipeline authoring

To learn about creating multiple-action workflows and pipelines through hands-on authoring by using GitHub Actions for Microsoft Power Platform, complete the GitHub Actions for Microsoft Power Platform labs.

More information: About GitHub Actions

See also

GitHub Actions for Microsoft Power Platform

Tutorial: Get started with GitHub Actions for Microsoft Power Platform

7/15/2022 • 3 minutes to read • Edit Online

This three part tutorial will give you an opportunity to get hands on with best practices to automate building and deploying your app using GitHub Actions for Power Platform. The first two tutorials are all about setting up required environments and creating a solution to later use with GitHub Actions. If you are experienced with creating environments and solutions, you can follow the Tip below and skip to the third tutorial to begin using GitHub Actions for Power Platform.

- Create three Microsoft Dataverse environments in your tenant
- (**Highly recommended**) Create a service principal and provide the appropriate permissions
- Create a model-driven app
- Export and deploy your app using application lifecycle management (ALM) automation

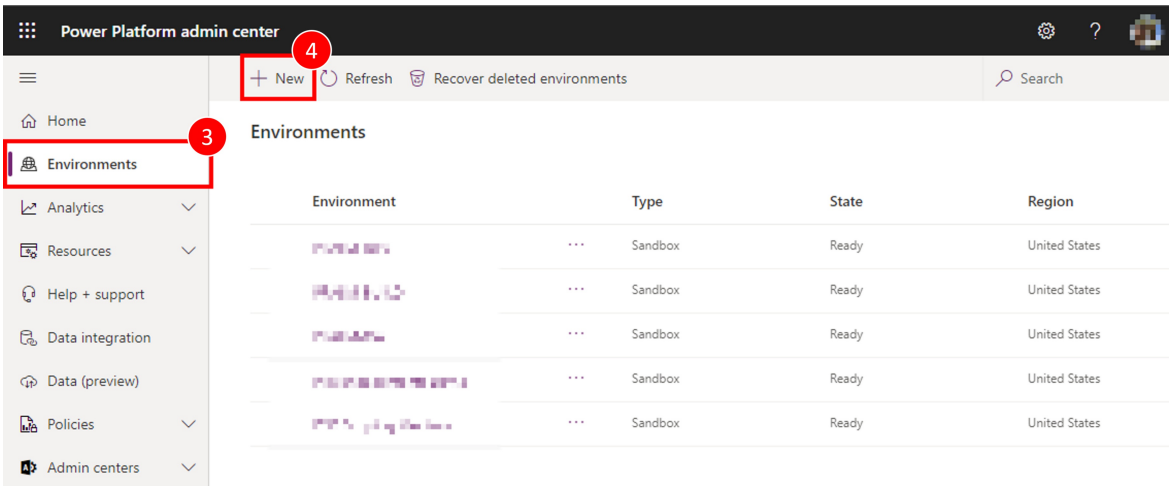
> **TIP**
> If you are already familiar with the concept of multiple Dataverse environments as well as how to use solutions to package your app, simply download and use the sample ALMLab solution and then skip to the last tutorial of this series.

Let's get started with tutorial #1 and create three Dataverse environments.

Create required environments

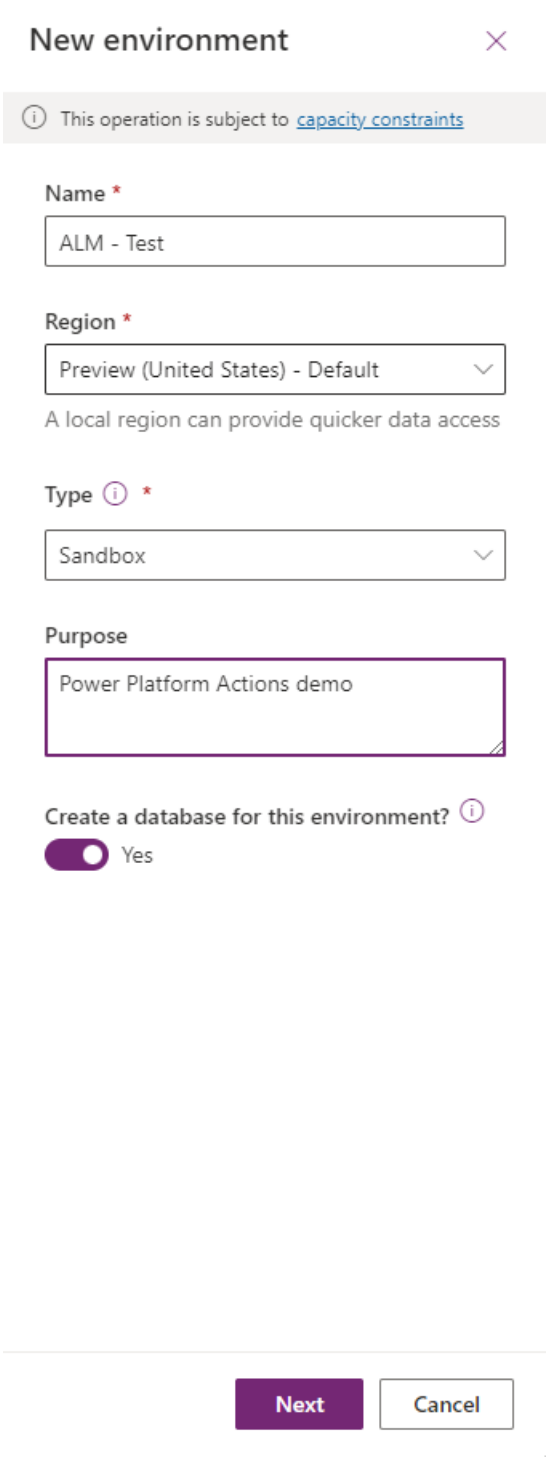
You will need to create, or have access to, three Dataverse environments in your demo or customer tenant. To create these environments, follow the instructions below. Otherwise, proceed to the end of this tutorial for the next steps.

1. Sign in to the Power Platform admin center with credentials that provide access to a tenant with a minimum 3-GB available capacity (required to create the three environments).
2. Select **Environments** in the navigation area.
3. Select **+ New** to create your first new environment.

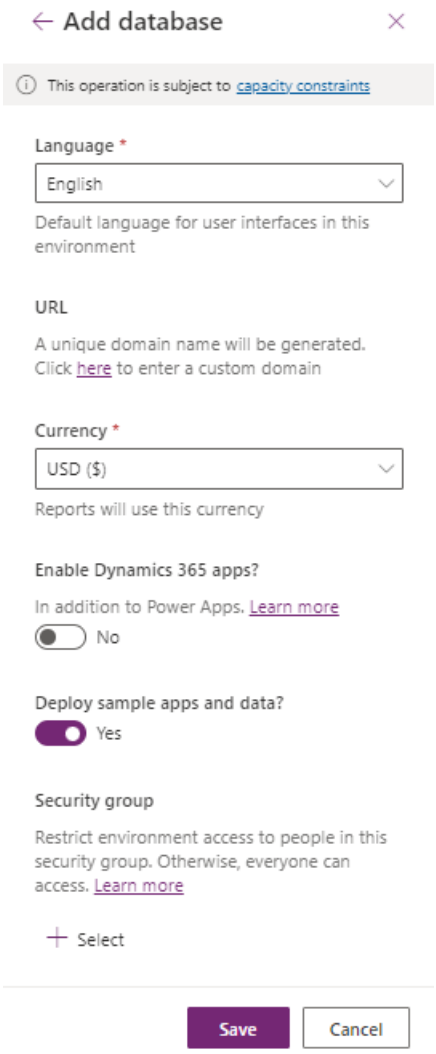


4. The first environment should be named "Your Name – dev", set the region to "United States (default)", set the Environment type to **Sandbox** (if available), if not use "Trial".

- Ensure the *Create a database for this environment* radio toggle is set to **Yes**



5. **Click Next.**
6. Set the Language and currency as preferred and set the "*Deploy sample apps and data?*" radio button to **Yes**, then **click Save**

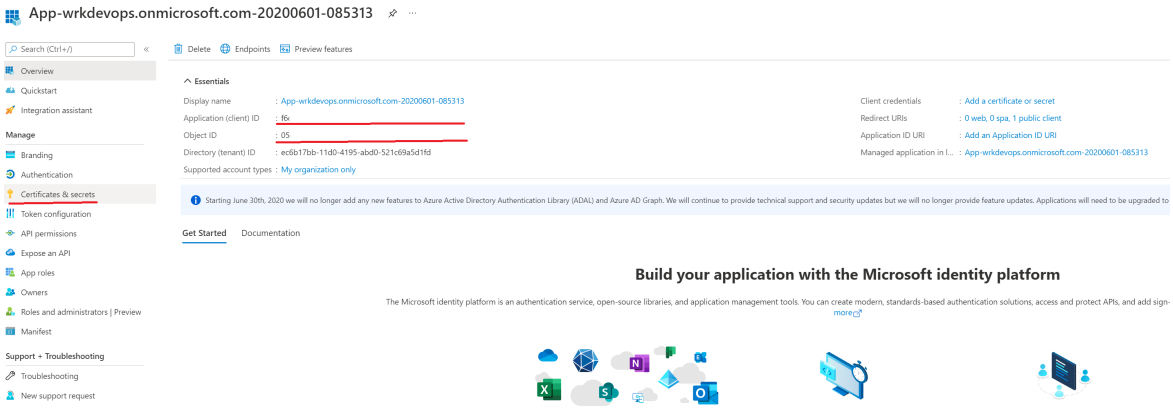


7. Your development environment has been created, follow steps 2 – 7 above to create a second environment called “Your Name – build” , and then finally, create a third environment called “Your Name – prod”

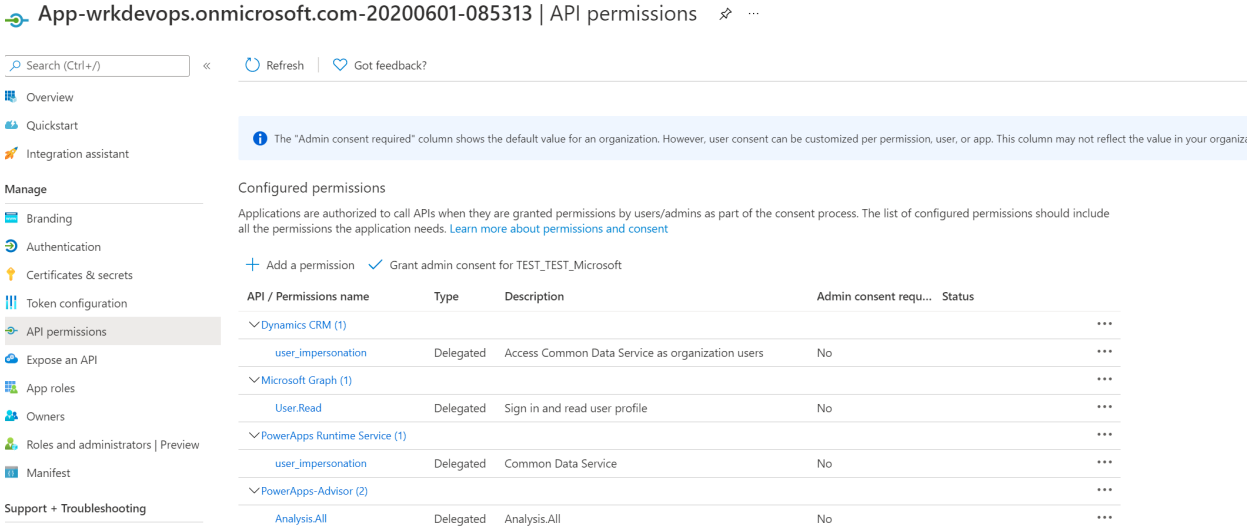
Now you have the environments that we will need for this and ready to begin the next modules of this Hands-on lab.

Create the service principal account and give it rights to the environments created

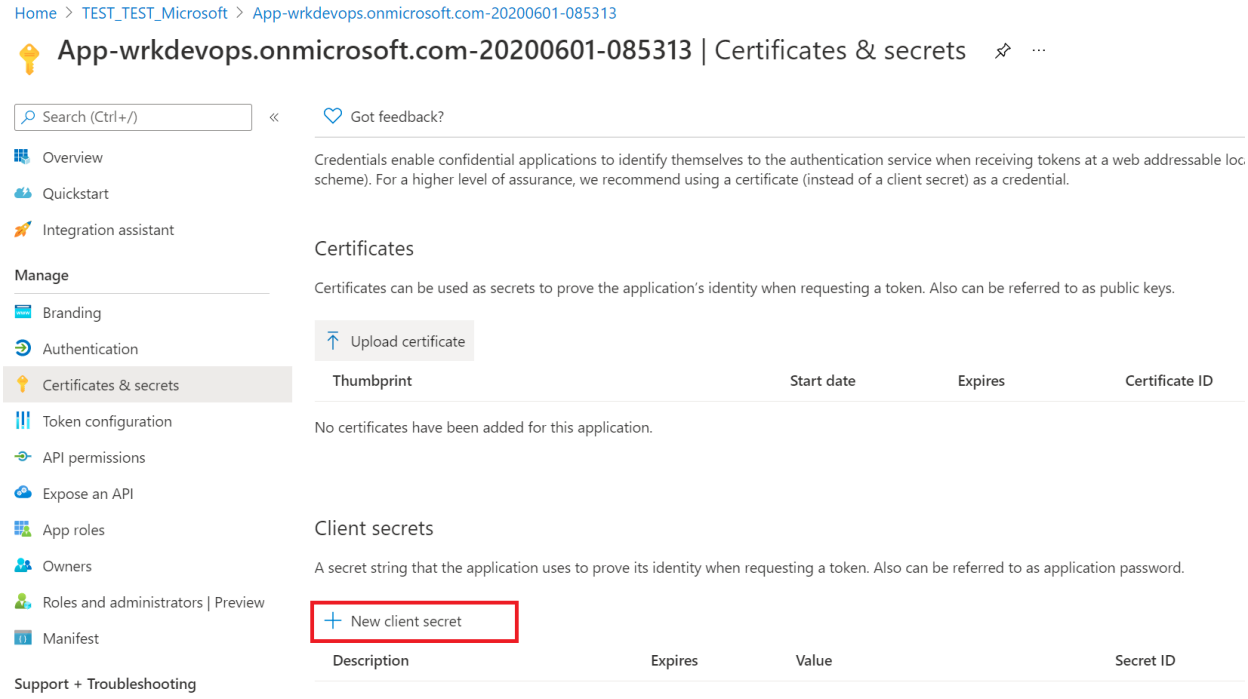
1. You will need to create an application registration within Azure Active Directory. More information: Tutorial: Register an app with Azure Active Directory
2. Upon creation of the application registration, please note and save the Directory (tenant) ID and the Application (client) ID of the application.



3. On the navigation panel of the **Overview** page, select **API permissions**.
4. Choose **+ Add a permission**, and in the **Microsoft APIs** tab, Choose **Dynamics CRM**.
5. In the **Request API permissions** form, select **Delegated permissions**, check **user_impersonation**, and then choose **Add permissions**.
6. From the **Request API permissions** form, choose **PowerApps Runtime Service**, select **Delegated permissions**, check **user_impersonation**, and then choose **Add permissions**.
7. From the **Request API permissions** form, choose **APIs my organization uses**, search for "PowerApps-Advisor" using the search field, select **PowerApps-Advisor** in the results list, select **Delegated permissions**, check **Analysis.All** rights, and then choose **Add permissions**.



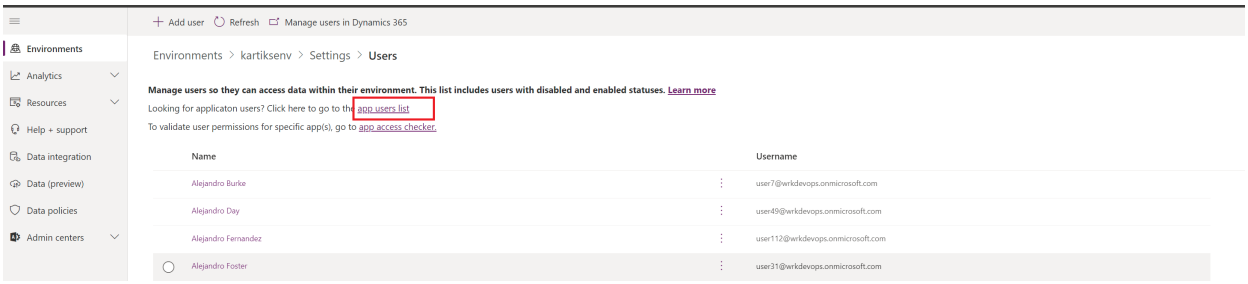
8. Next, proceed to create a client secret, in the navigation panel, select **Certificates & secrets**.
9. Below **Client secrets**, select **+ New client secret**.
10. In the form, enter a description and select **Add.** Record the secret string, you will not be able view the secret again once you leave the form.



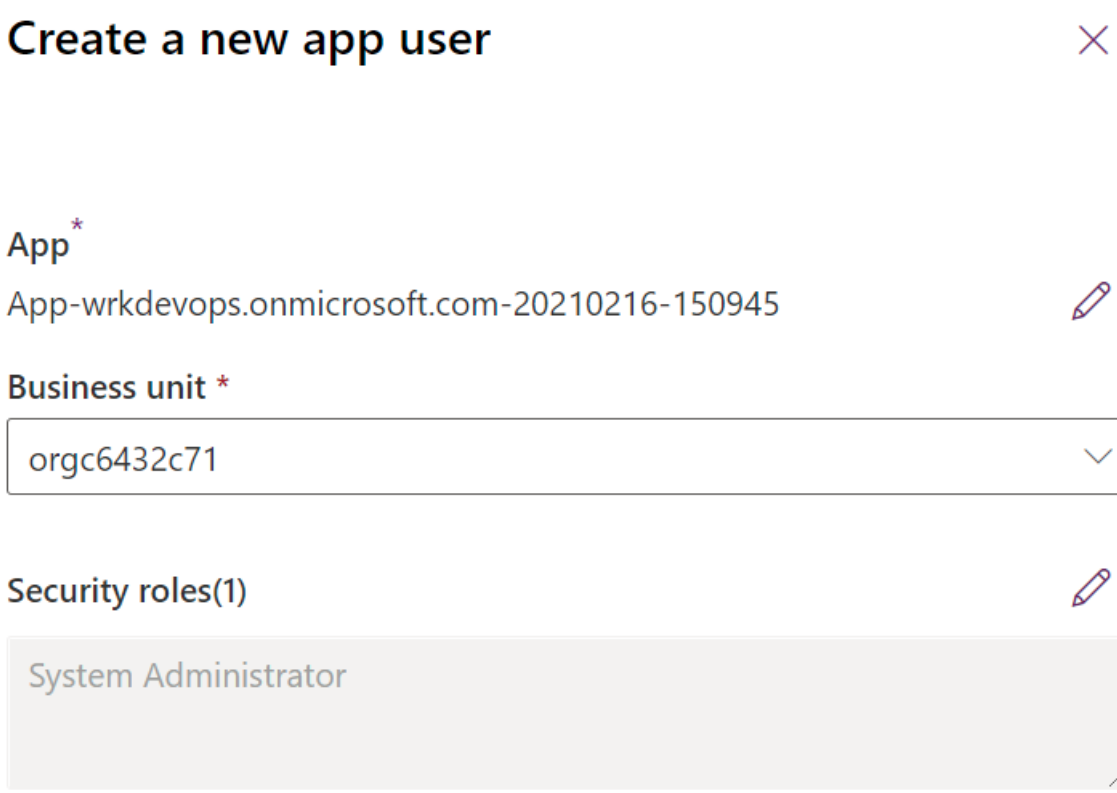
Application user creation

In order for the GitHub workflow to deploy solutions as part of a CI/CD pipeline an "Application user" needs to be given access to the environment. An "Application user" represents an unlicensed user that is authenticated using the application registration completed in the prior steps.

1. Navigate to your Dataverse environment (https://*[org]*.crm.dynamics.com).
2. Navigate to **Settings** > **Security** > **Users**.
3. Select the link **app users list**.



4. Select **+ new app user**. A panel will open on the right hand side of the screen.
5. Select **+ Add an app**. A list of all the application registrations in your Azure AD tenant is shown. Proceed to select the application name from the list of registered apps.
6. Under **Business unit**, in the drop down box, select your environment as the business unit.
7. Under **Security roles**, select **System administrator**, and then select **create**. This will allow the service principal access to the environment.



Now that you have created the service principal, you can use either the service principal or the standard username and password for your GitHub Workflow.

> **IMPORTANT**
> If you have multi-factor authentication (MFA) enabled, service principal authentication is the authentication method you want to use.

Next steps

See Also

Automate your workflow from idea to production

Tutorial: Build a model-driven app for deployment using GitHub Actions for Microsoft Power Platform

7/15/2022 • 2 minutes to read • Edit Online

In this tutorial, you will be creating a simple model-driven app to deploy using GitHub Actions for Microsoft Power Platform in the next tutorial.

- Create a model-driven app

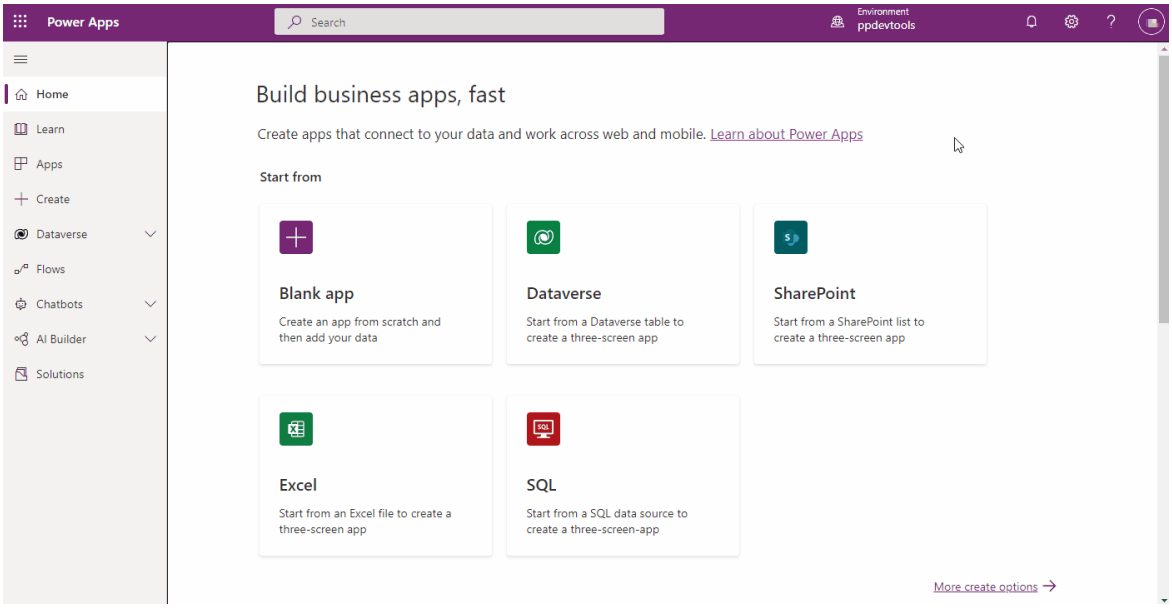
In the next tutorial, you will learn how to:

- Export and deploy your app using application lifecycle management (ALM) automation

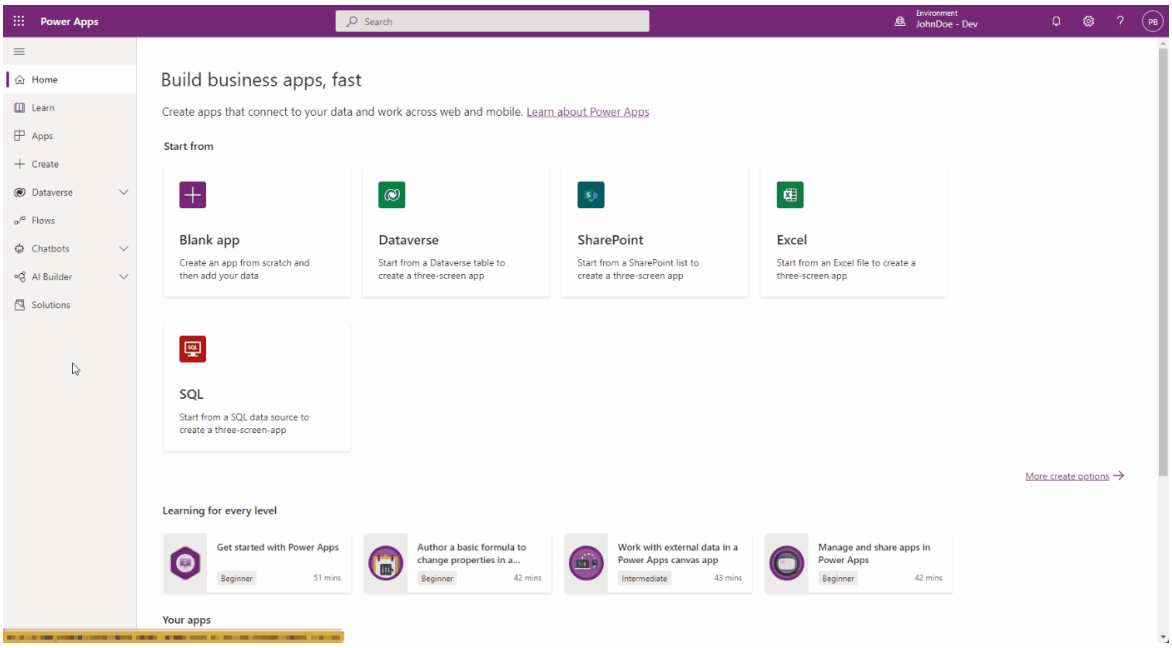
Build a model-driven app

Follow the steps below to build a model-driven app.

1. In your browser, navigate to https://make.powerapps.com and sign in with your credentials. Click the environment selector dropdown in the header and select your development environment.

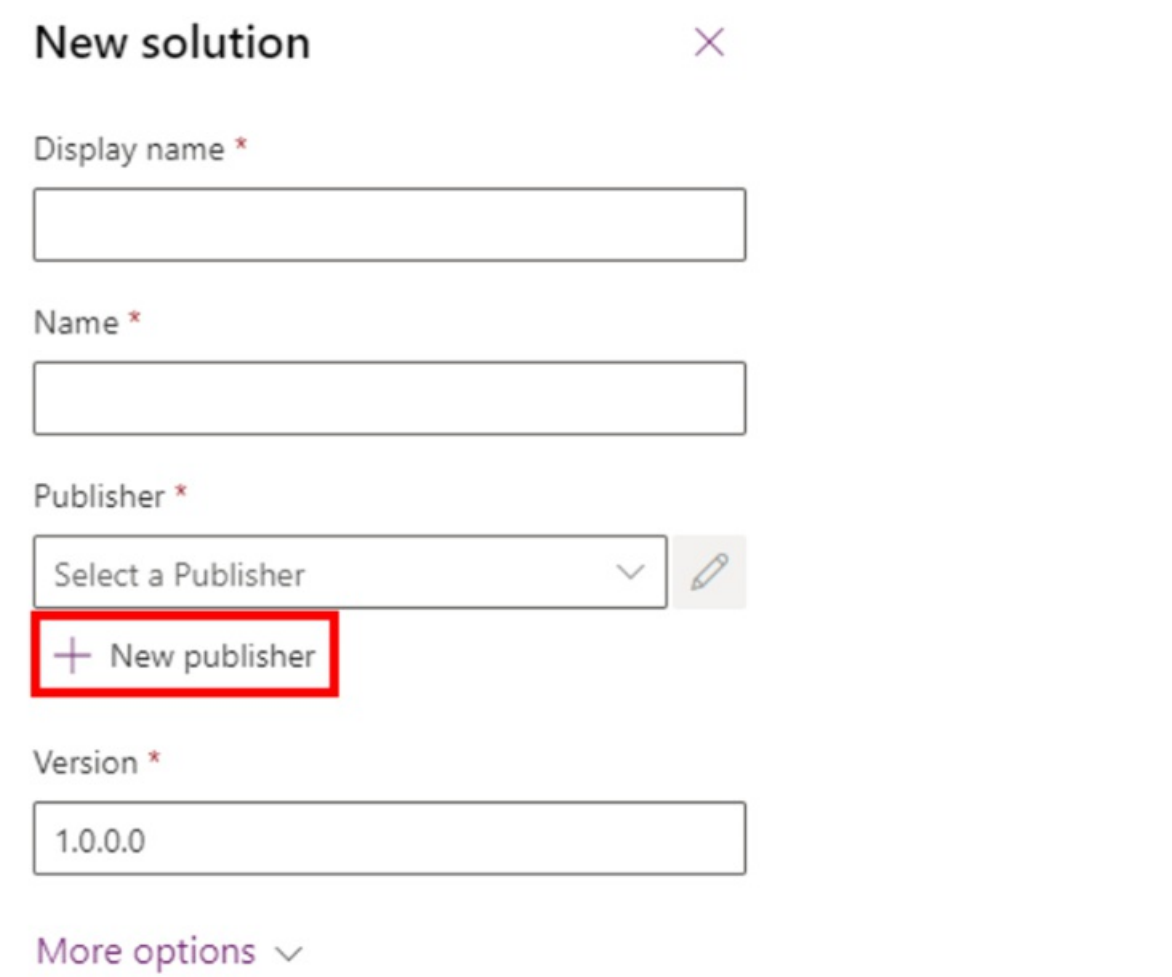


2. Click the **Solutions** area in the left navigation, and then click the **New solution** button to create a new solution.

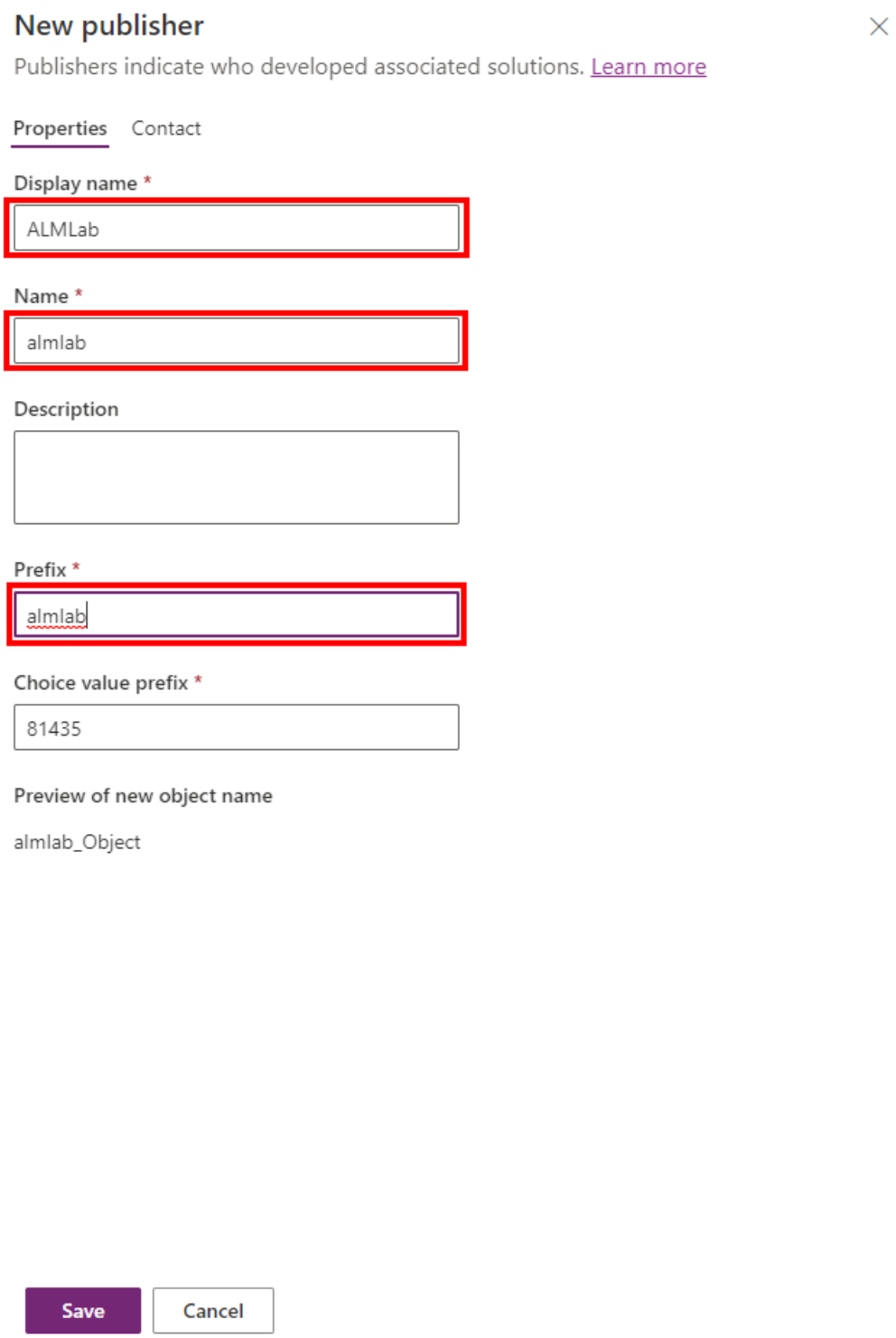


3. In the side panel that appears, enter a name for the application and then click the **Add Publisher** option.

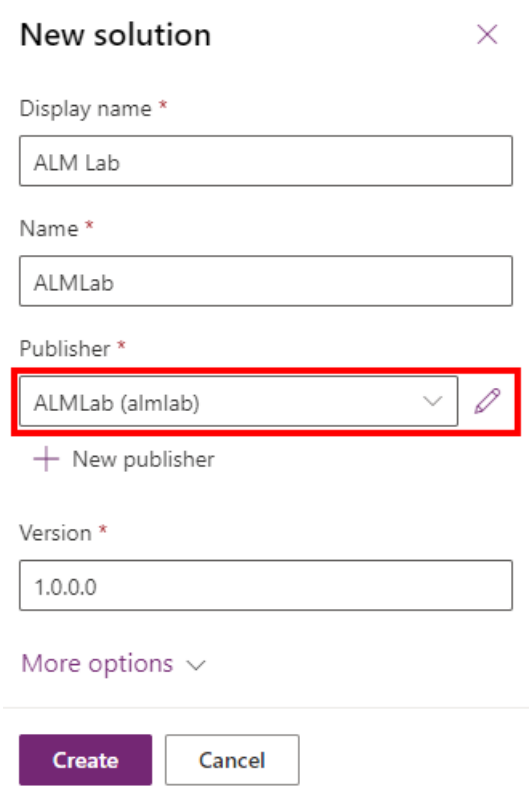
> **NOTE**
>
> The solution publisher specifies who developed the app, so you should always create a solution publisher name that is meaningful. Furthermore, the solution publisher includes a prefix, which helps you distinguish system components or components introduced by others and is also a mechanism to help avoid naming collisions. This allows for solutions from different publishers to be installed in an environment with minimal conflicts.



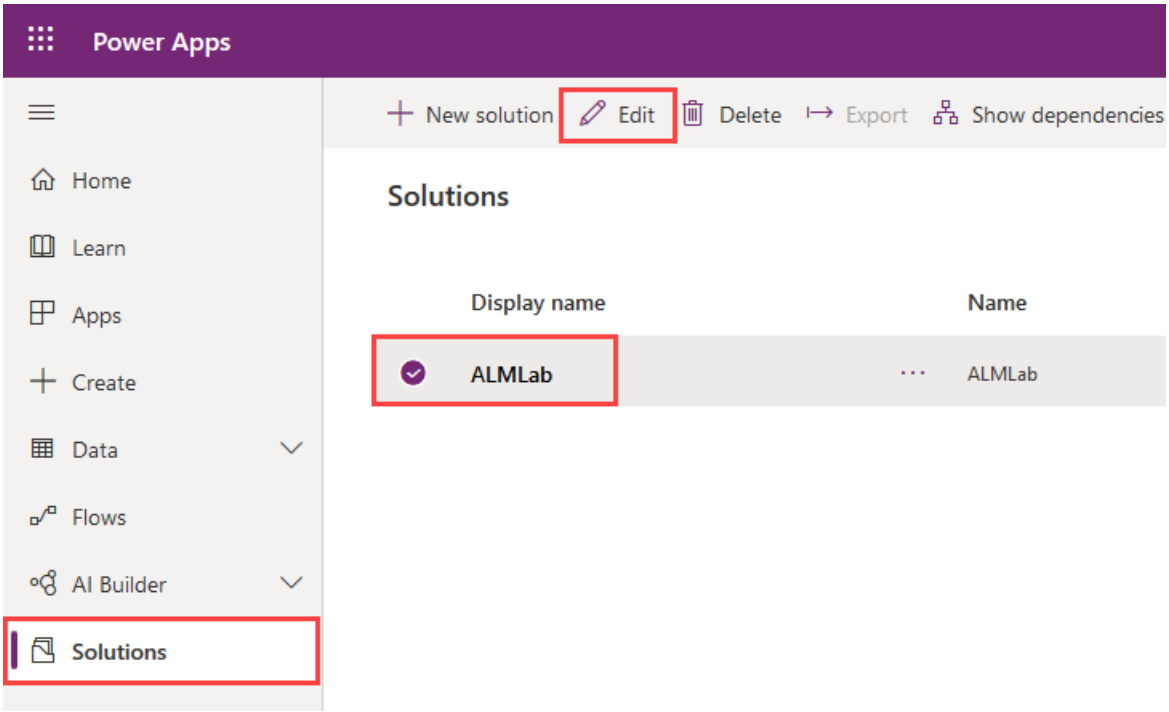
4. For the purposes of this tutorial, enter 'ALMLab' for the *display name*, *name*, and *prefix*, and then choose **Save and Close**.



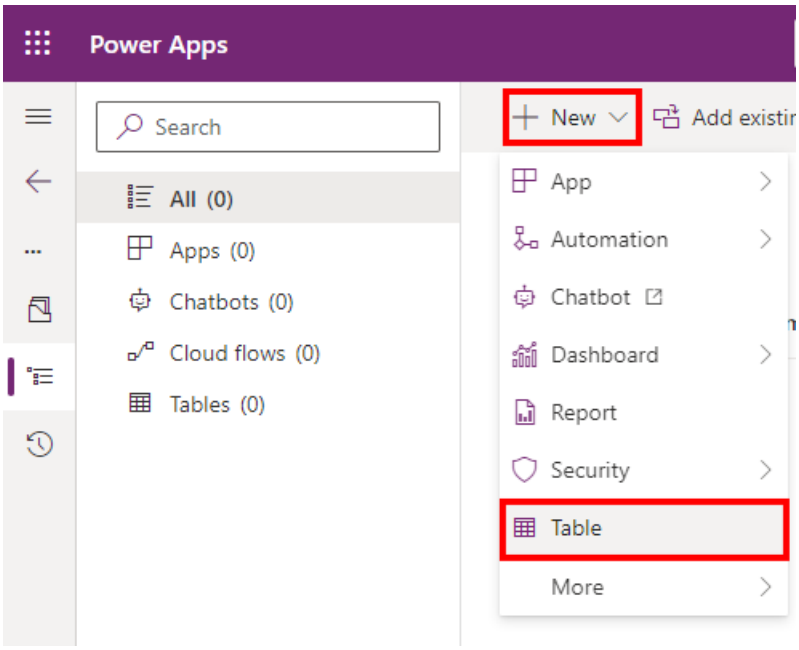
5. On the new solution panel, select the publisher that you just created and **click Create** to create a new unmanaged solution in the environment.



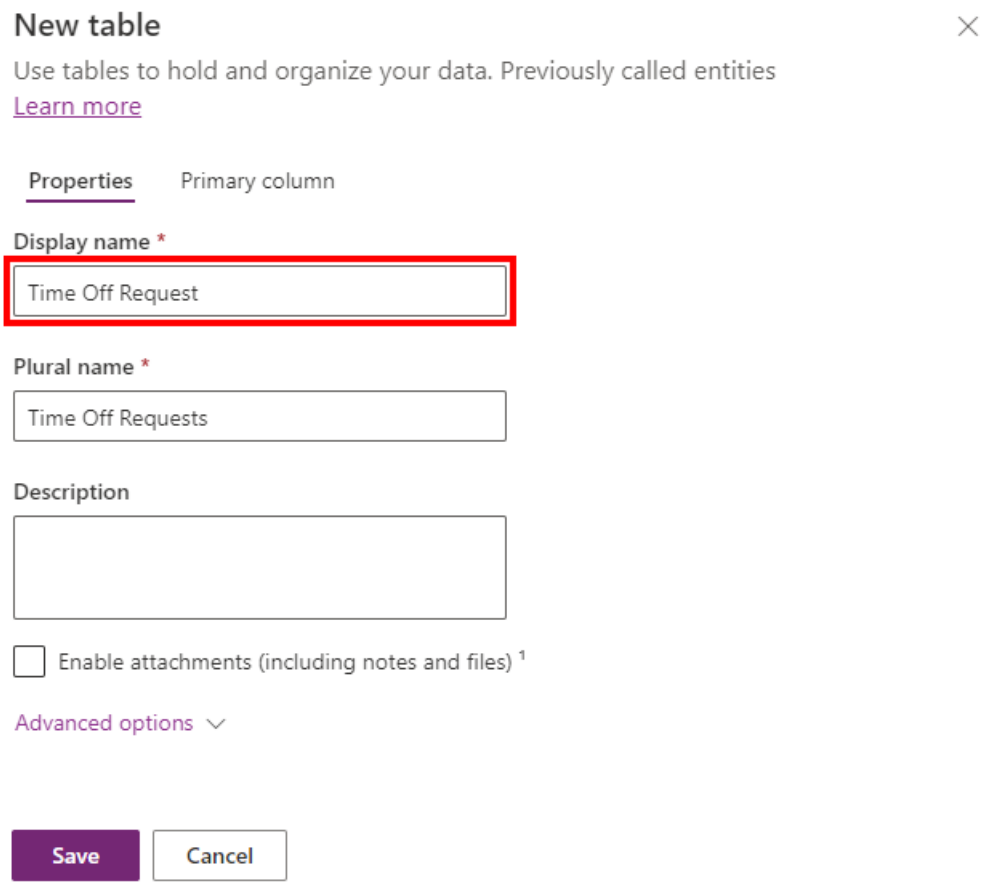
6. In the solutions list, select the solution you just created and click the **Edit** button.



7. Your new solution will be empty, and you need to add components to it. In this lab we will create a custom table. Click the **+ New** dropdown from the top navigation and select **Table**

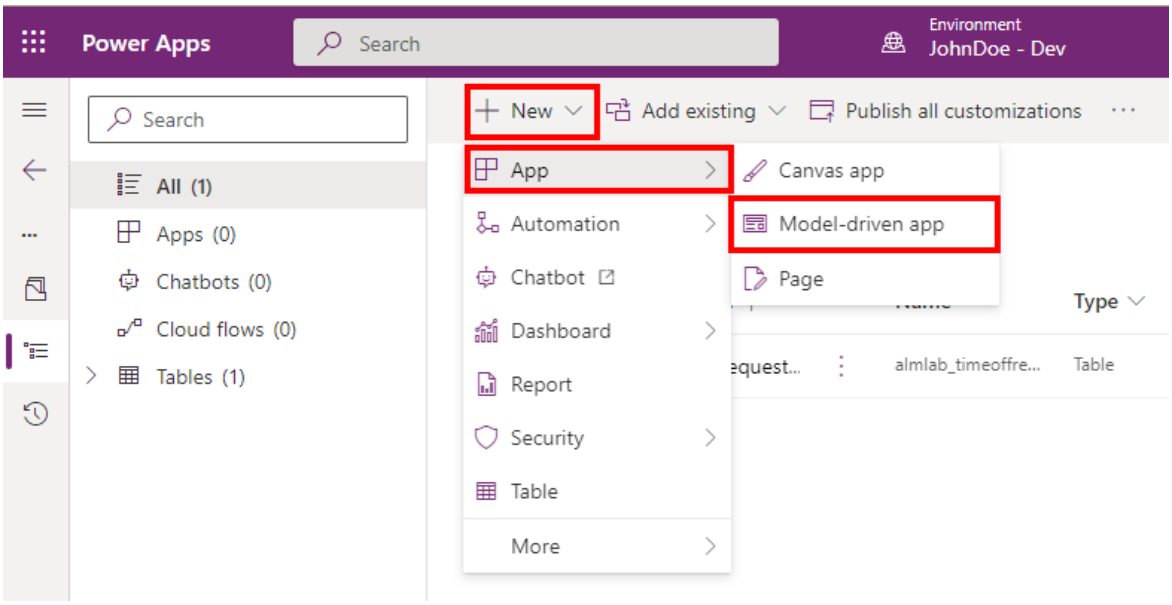


8. Enter a **display name**, plural name will be generated for you. Click **Save** to create the table.

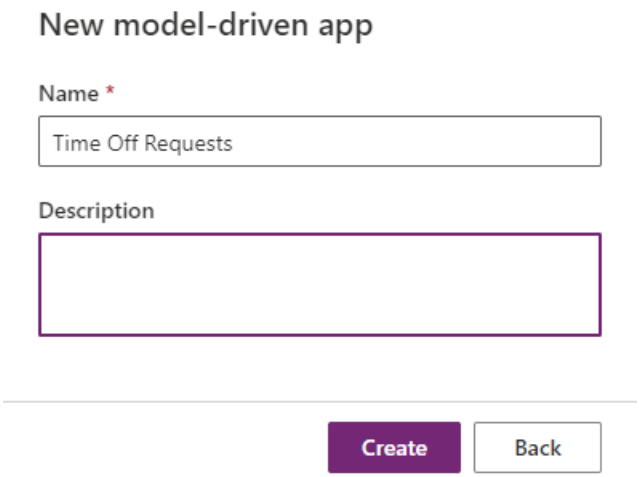


9. Once your table is created, select the solution name again to go back to the solution view to add another component.

10. Click the **+ New** dropdown, then **App**, and finally click **Model-driven app**. If you get a pop-up asking to select the creating experience choose *Modern app designer*

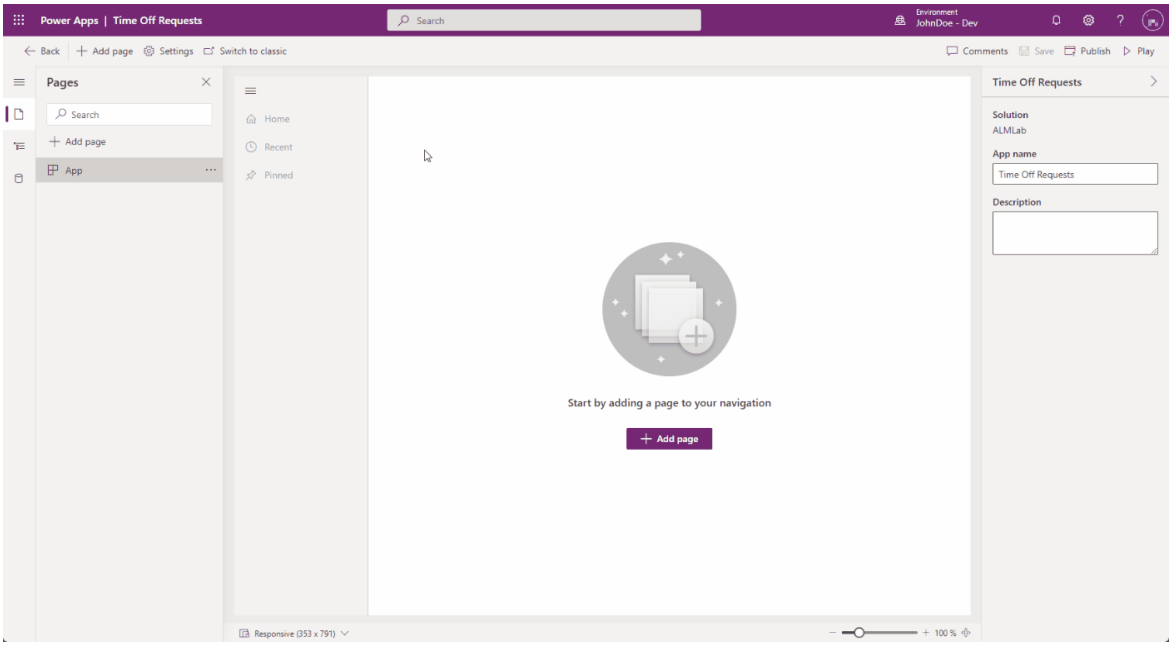


11. Enter an app name, then click the **Create** button

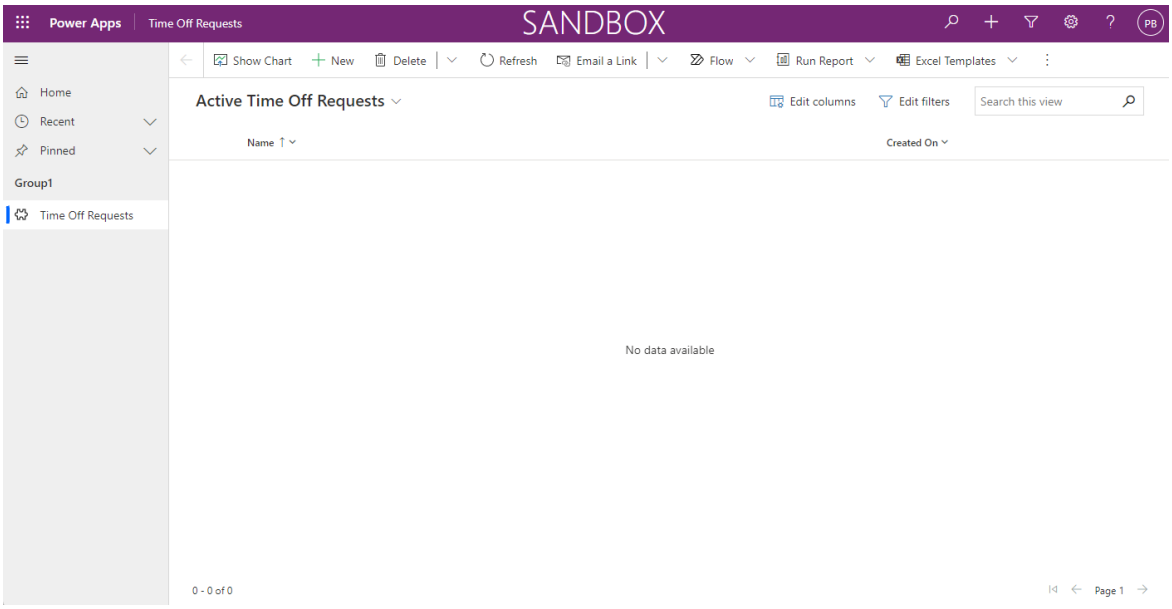


12. In the application designer, click **+ Add page**, choose **Table based view and form**, then click **Next**. On the next screen **search by name for the table you previously created**. **Check mark the selection** and click **Add**

    - Ensure that *Show in navigation* is checked



13. Click **Publish**, once the publish action is complete click **Play**.
14. This will take you to the application so that you can see how it looks. You can use the application and close the tab when you are satisfied.



Next steps

See Also

Automate your workflow from idea to production

Tutorial: Automate solution deployment using GitHub Actions for Microsoft Power Platform

7/15/2022 • 7 minutes to read • Edit Online

In this tutorial, you will learn how to:

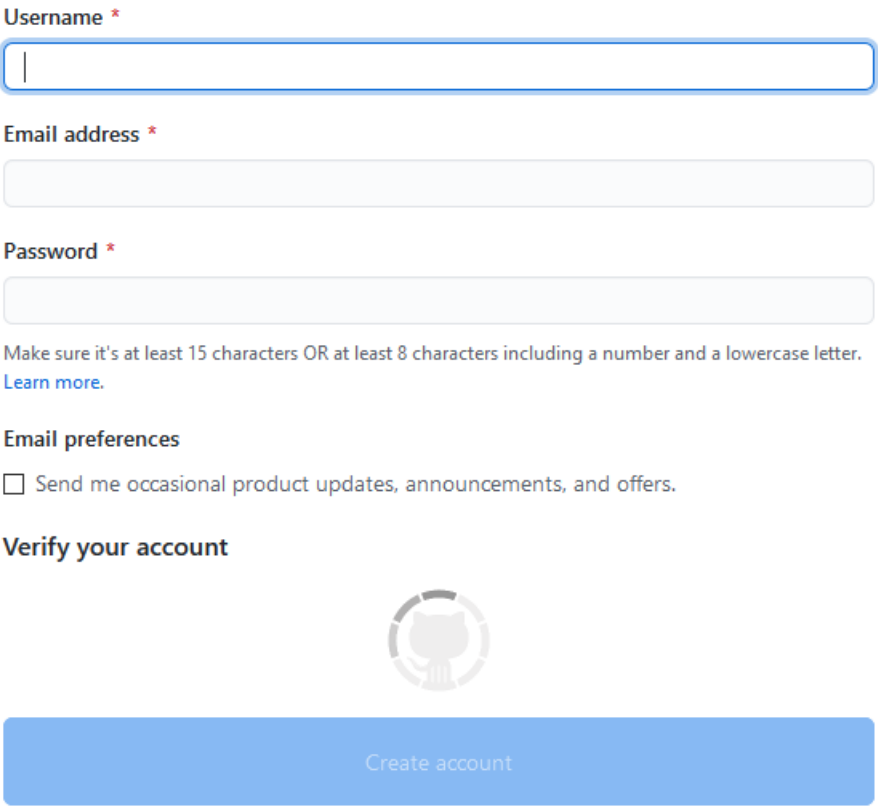
- Create a new GitHub repository
- Create two GitHub workflows using GitHub Actions for Microsoft Power Platform

The workflows can automatically export your app (as an unmanaged solution) from a development environment, generate a build artifact (managed solution), and deploy the app into your production environment. This tutorial uses the ALMLab solution you built and the environments you set up in previous tutorials.

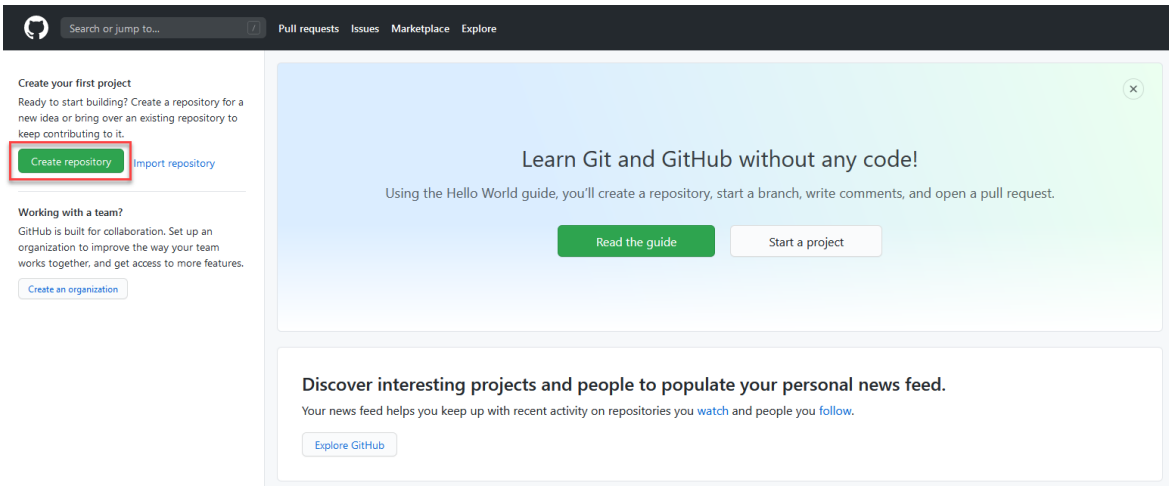
Related tutorials: Get started, and Build a model-driven app.

Create a GitHub Account

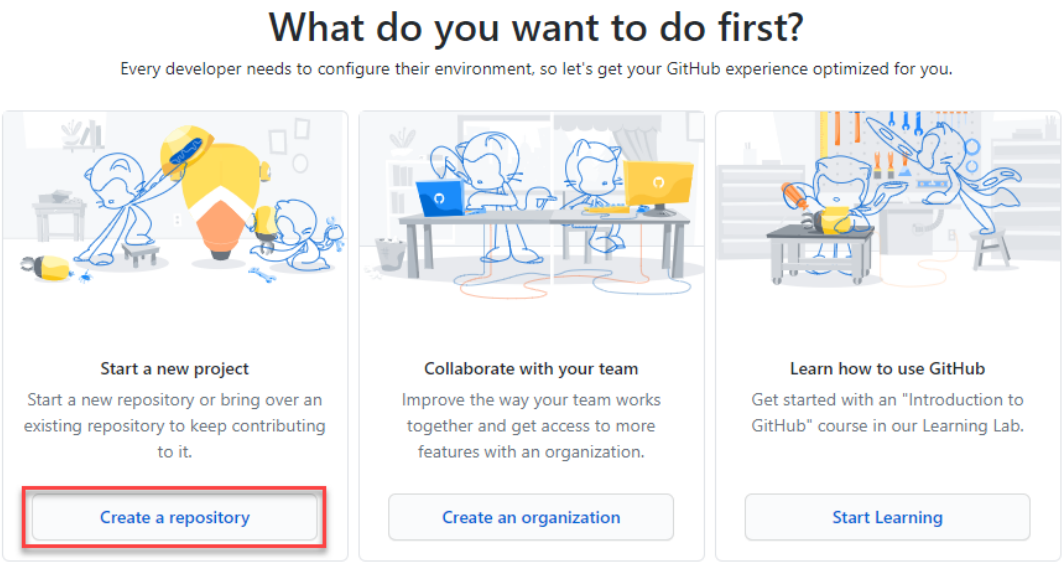
1. Go to https://github.com and click **Sign up** or **Start a free trial** (or sign in if you have an existing account).



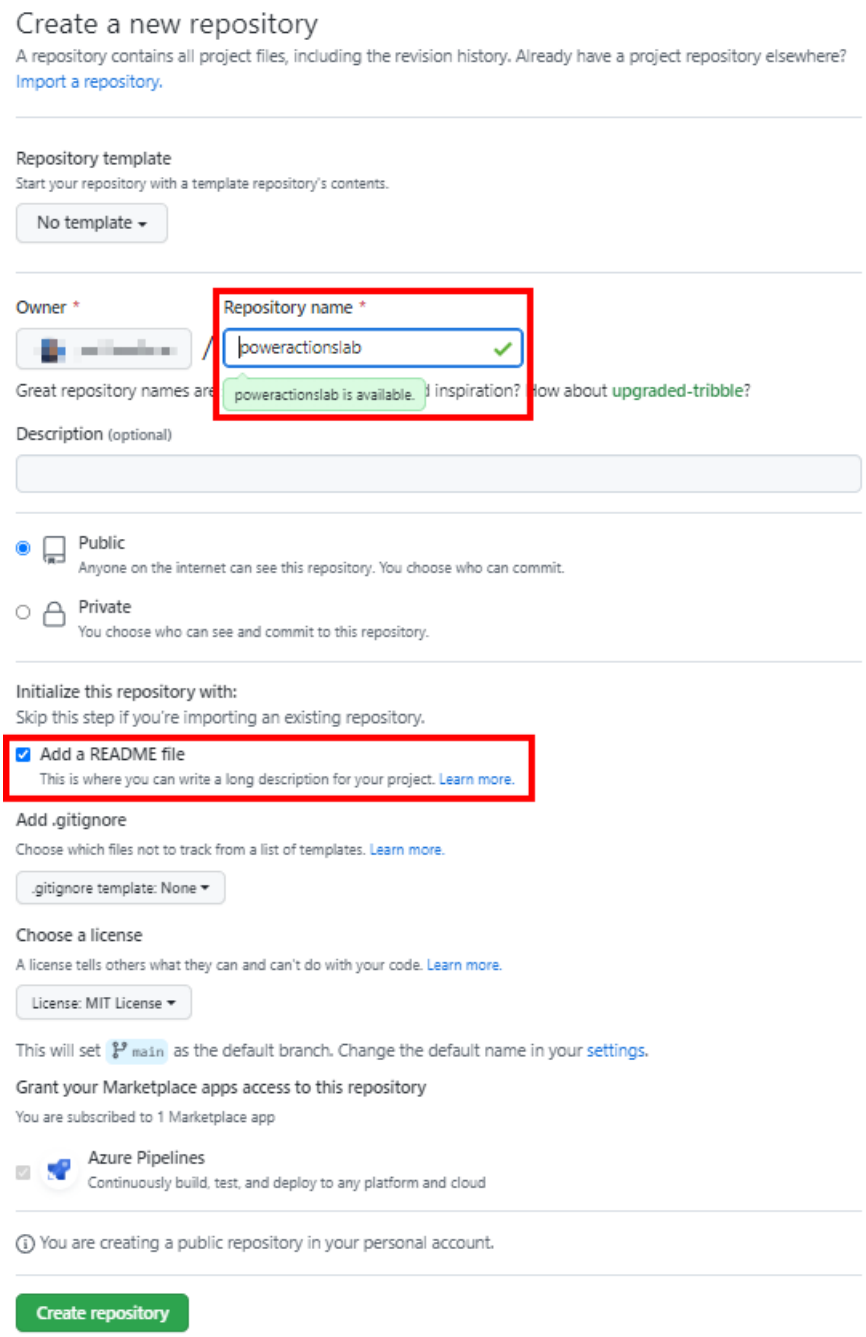
2. After you have created your account, create a repository by selecting **Create repository** or **New**.



You might see the following alternative landing screen:

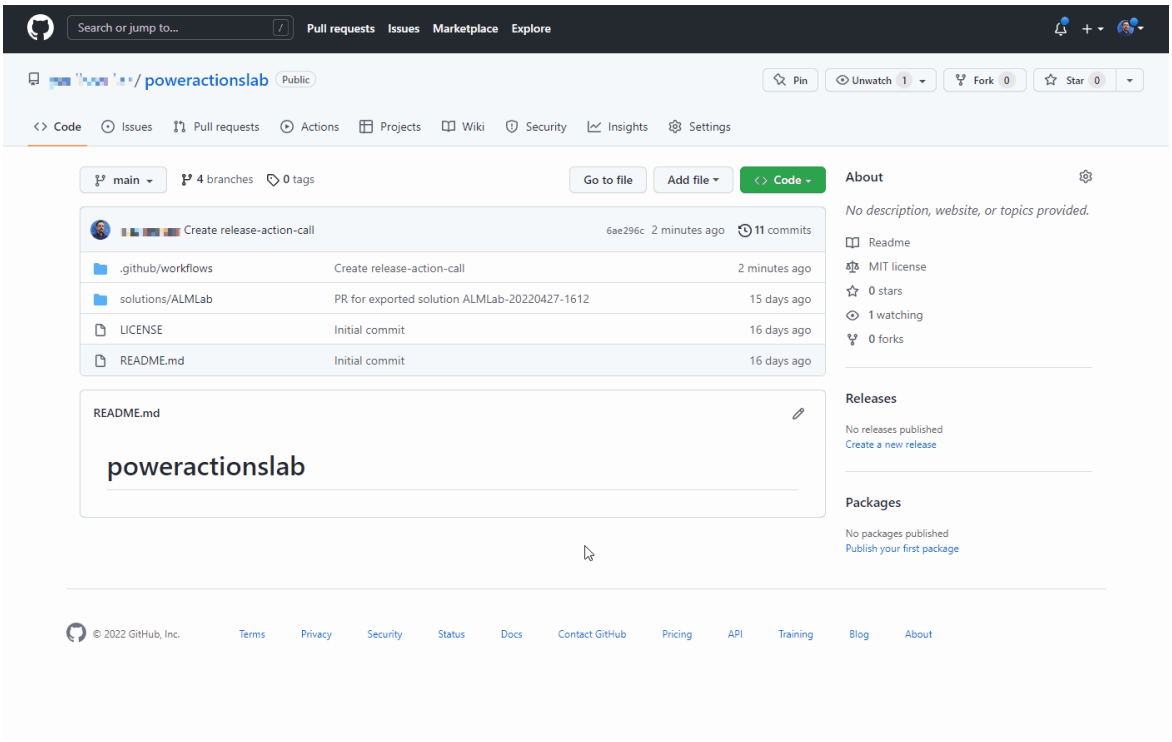


3. Create your new repository and name it 'poweractionslab'. Make sure you select **Add a README file** to initiate the repo and choose **Create repository**.



Creating a new secret for Service Principal Authentication

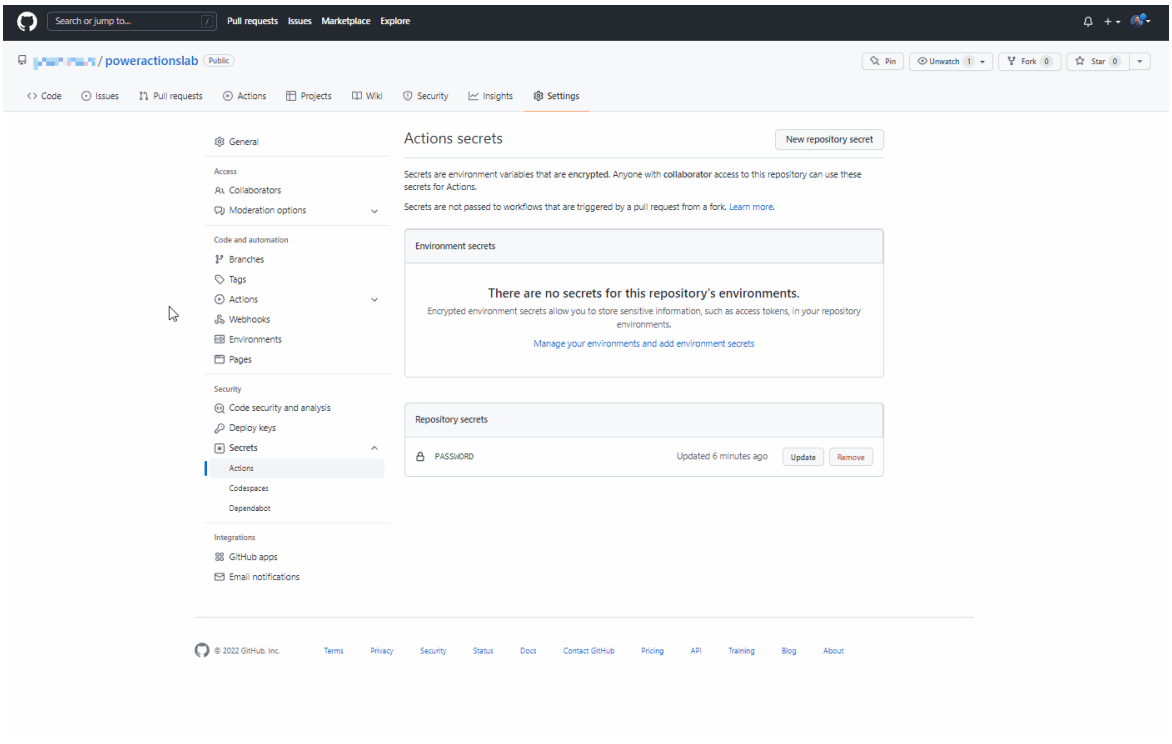
1. Navigate to to you repository and click **Settings**, then expand **Secrets**, and then and click **Actions**.
2. On the *Secrets* page, name the secret 'PowerPlatformSPN'. Use the client secret from the application registration created in Azure Active Directory and enter it into the **Value** field, and then select **Add secret**. The client secret will be referenced in the YML files used to define the GitHub workflows later in this lab.



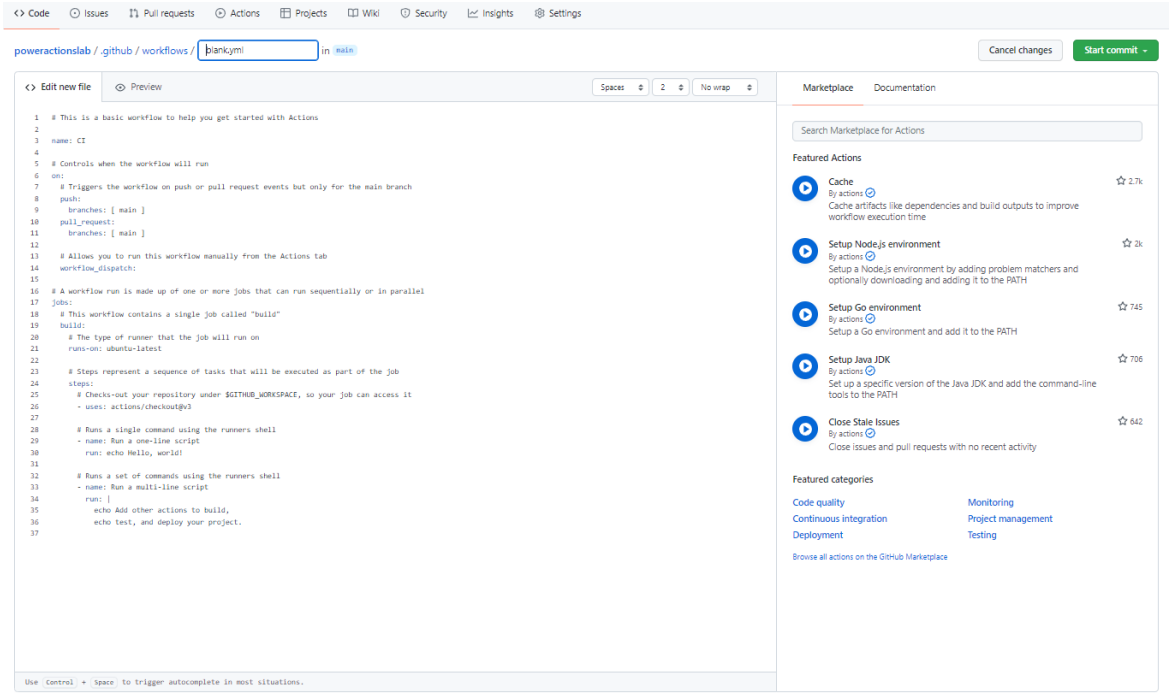
The client secret is now securely stored as a GitHub secret.

Create a workflow to export and unpack the solution file to a new branch

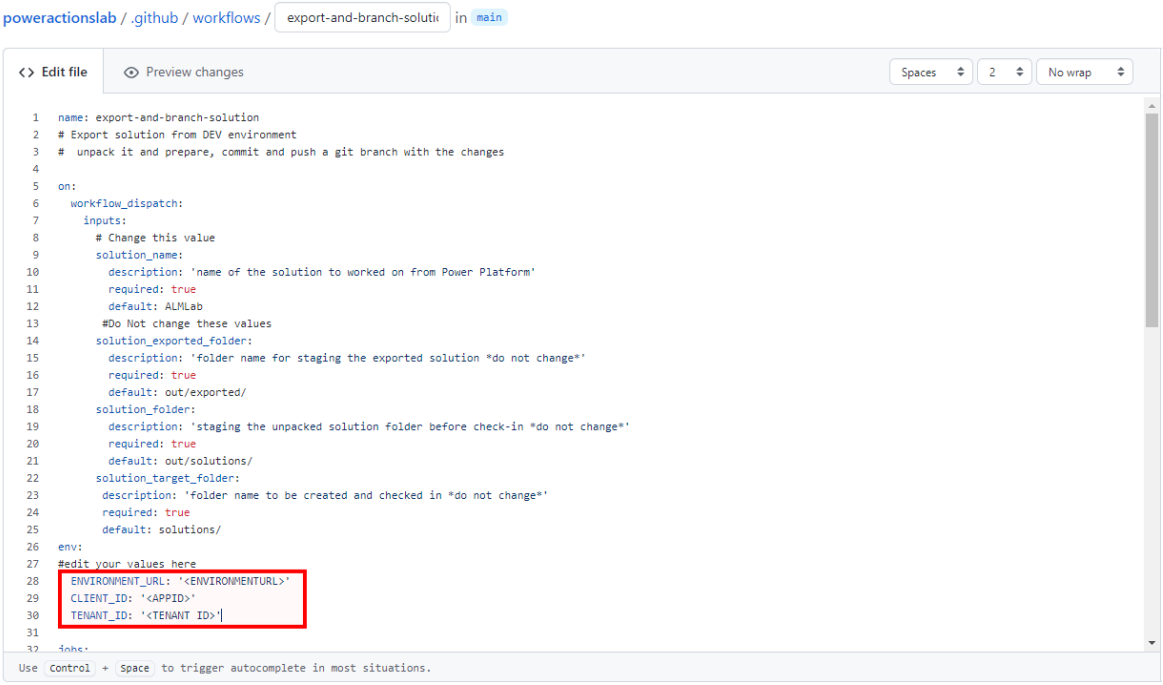
1. click on **Actions** and click **set up a workflow yourself** or click Configure in the *Simple workflow* box under the *suggested for this repository* section.



2. This will start a new YAML file with a basic workflow to help you get started with GitHub actions.



3. Delete the pre-created content, paste the content from the export-and-branch-solution-with-spn-auth.yml file, and then rename the file to 'export-and-branch-solution'.yml.




```yaml
name: export-and-branch-solution
# Export solution from DEV environment
#  unpack it and prepare, commit and push a git branch with the changes

on:
  workflow_dispatch:
    inputs:
      # Change this value
      solution_name:
        description: 'name of the solution to worked on from Power Platform'
        required: true
        default: ALMLab
       #Do Not change these values
      solution_exported_folder:
        description: 'folder name for staging the exported solution *do not change*'
        required: true
        default: out/exported/
      solution_folder:
        description: 'staging the unpacked solution folder before check-in *do not change*'
        required: true
        default: out/solutions/
      solution_target_folder:
       description: 'folder name to be created and checked in *do not change*'
       required: true
       default: solutions/
env:
#edit your values here
  ENVIRONMENT_URL: '<ENVIRONMENTURL>'
  CLIENT_ID: '<APPID>'
  TENANT_ID: '<TENANT ID>'

jobs:
```

4. Update `<ENVIRONMENTURL>` with the URL for the development environment you want to export from (for example: https://poweractionsdev.crm.dynamics.com).

5. Update `<APPID>` and `<TENANT ID>` with your values.

   - If you are using credentials, paste the export-and-branch-solution.yml file instead of the *export-and-branch-solution-with-spn-auth.yml* file contents. Update `<USERNAME>` with the username you are using to connect to the environment.

```
75 lines (65 sloc) | 2.6 KB                                                    Raw  Blame

name: export-and-branch-solution
# Export solution from DEV environment
#  unpack it and prepare, commit and push a git branch with the changes

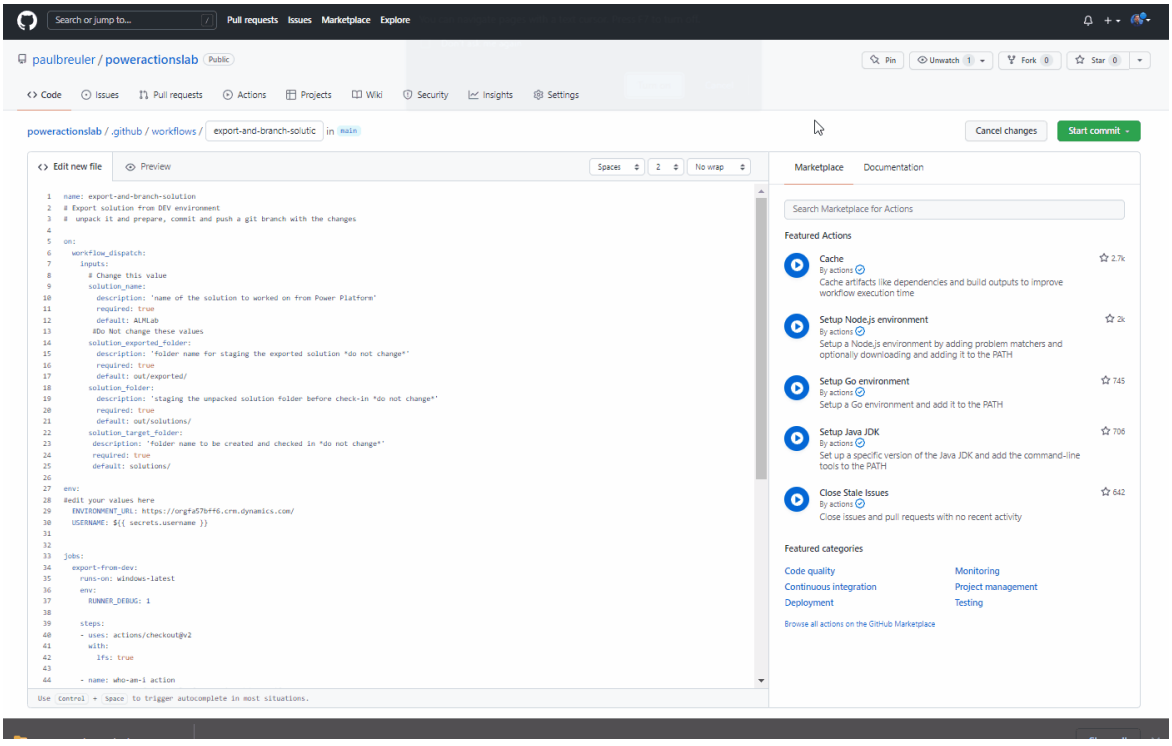
on:
  workflow_dispatch:
    inputs:
      # Change this value
      solution_name:
        description: 'name of the solution to worked on from Power Platform'
        required: true
        default: ALMLab
      #Do Not change these values
      solution_exported_folder:
        description: 'folder name for staging the exported solution *do not change*'
        required: true
        default: out/exported/
      solution_folder:
        description: 'staging the unpacked solution folder before check-in *do not change*'
        required: true
        default: out/solutions/
      solution_target_folder:
       description: 'folder name to be created and checked in *do not change*'
       required: true
       default: solutions/

env:
#edit your values here
  ENVIRONMENT_URL: <ENVIRONMENTURL>
  USERNAME: <USERNAME>
```

TIP

If you are not familiar with GitHub Actions and want to learn more check out the official documentation at https://docs.github.com/en/actions.

6. You are now ready to commit your changes. Select **Start commit**, type **Create export yml** in the title field, and then add a description (optional). Next, click **Commit new file**.



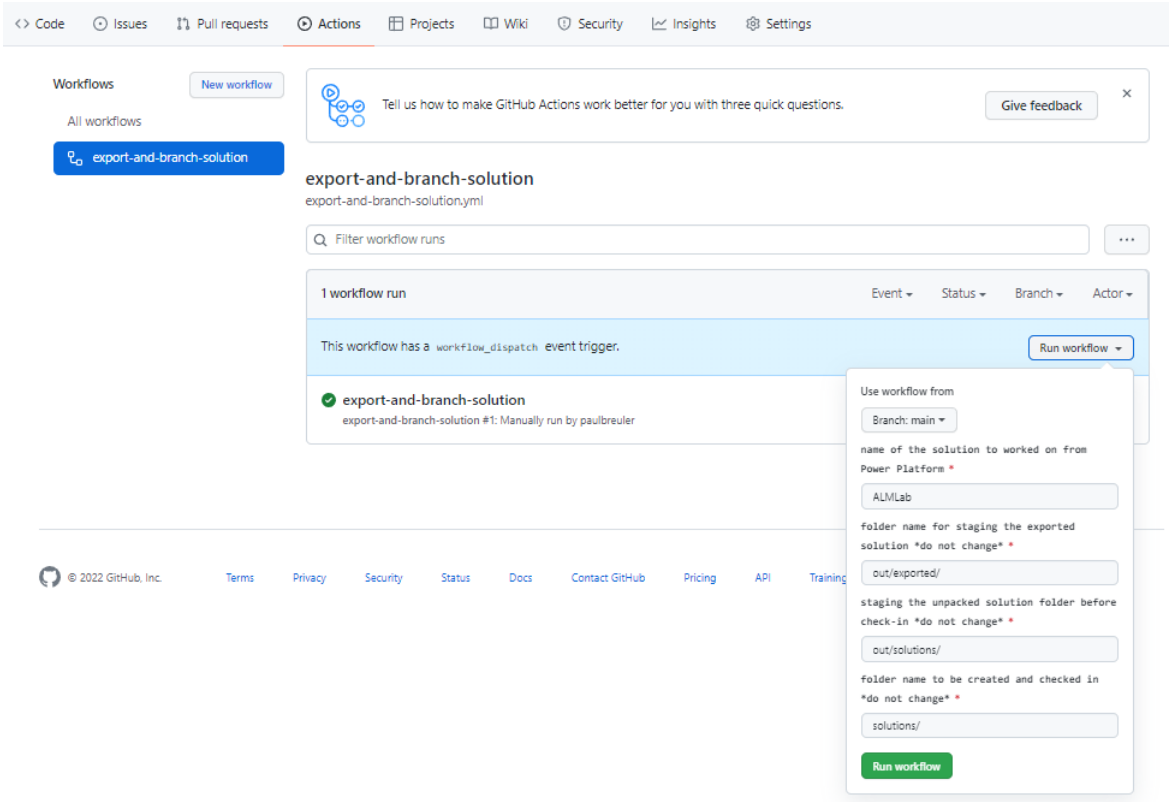
Congratulations, you have just created your first GitHub workflow using the following actions:

- **Who Am I**: Ensures that you can successfully connect to the environment you are exporting from.
- **Export Solution**: Exports the solution file from your development environment.
- **Unpack Solution**: The solution file that is exported from the server is a compressed (zip) file with consolidated configuration files. These initial files are not suitable for source code management as they are not structured to make it feasible for source code management systems to properly do differencing on the files and capture the changes you want to commit to source control. You need to 'unpack' the solution files to make them suitable for source control storage and processing.

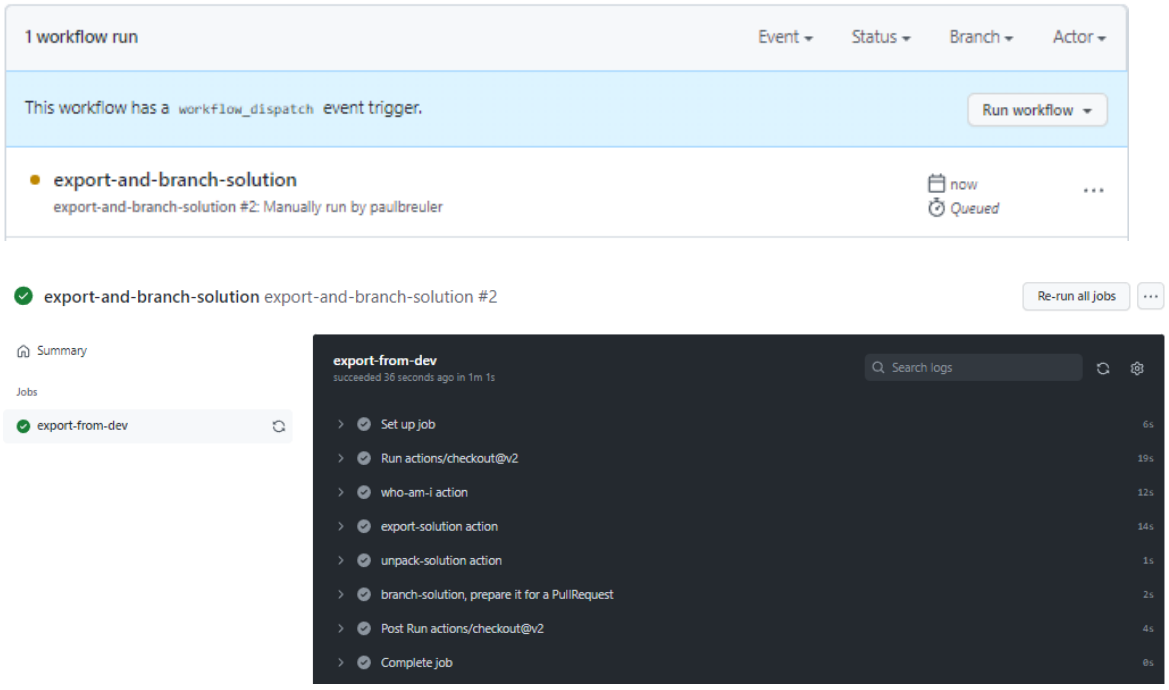
- **Branch Solution**: Creates a new branch to store the exported solution.

Test the export and unpack workflow

1. Next, test that the workflow runs. Navigate to **Actions**, **Run workflow**, and choose **Run workflow**. If you have a different solution name than 'ALMLab' then change the value here but leave the other values as is.



2. After 5–10 seconds the workflow will start, and you can select the running workflow to monitor progress.



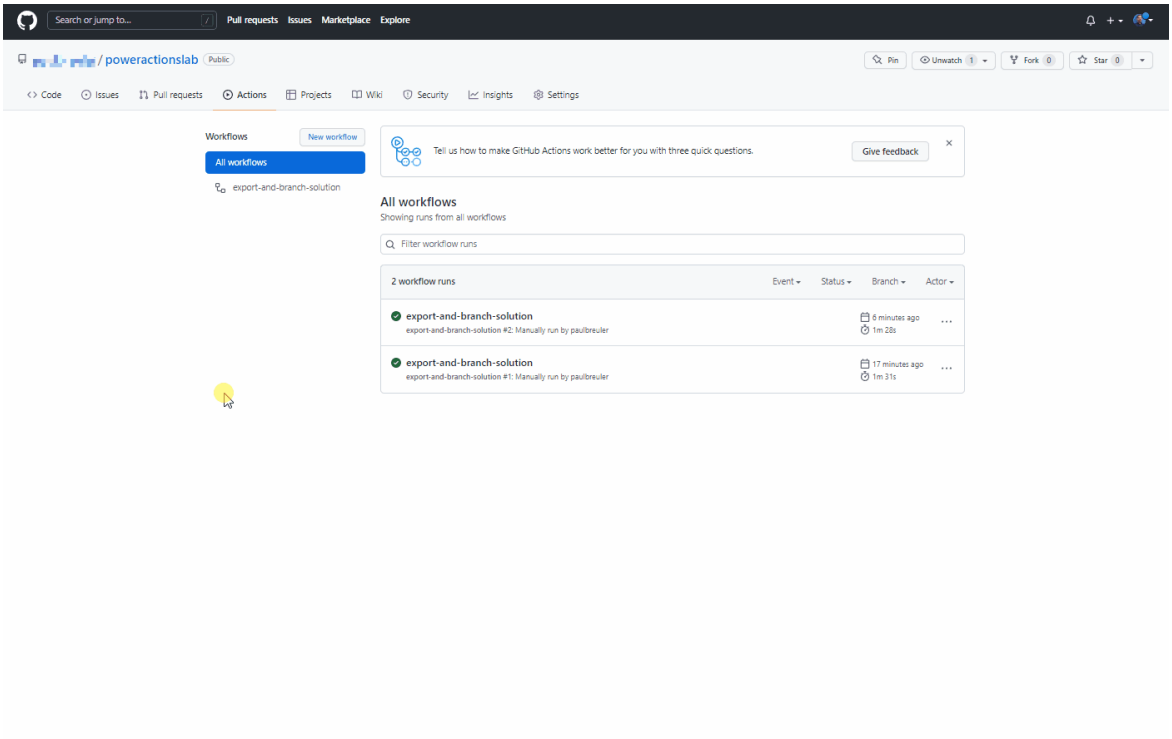
3. After the workflow has completed, validate that a new branch has been created with the solution unpacked to the solutions/ALMLab folder. **Navigate to the *Code* tab** and **expand the *branches* drop-down**.

4. Select the branch that was created by the action.

5. Validate that the solutions/ALMLab folder has been created in the new branch and then create a Pull request to merge the changes into the main branch. **Click *Contribute*** and in the flyout **click *Open Pull request***.

6. On the *Open a Pull request* screen, add a title and description, as desired, then **click *Create pull request.***

7. The screen will update showing the newly create pull request. As the pull request is created confirmation will be provided showing that our branch has no conflict with the main branch. This confirmation means that the changes can be merged into the main branch automatically. **Click *Merge pull request*** and then **click *Confirm merge***. Optionally, click delete branch to clean up the now defunct branch.



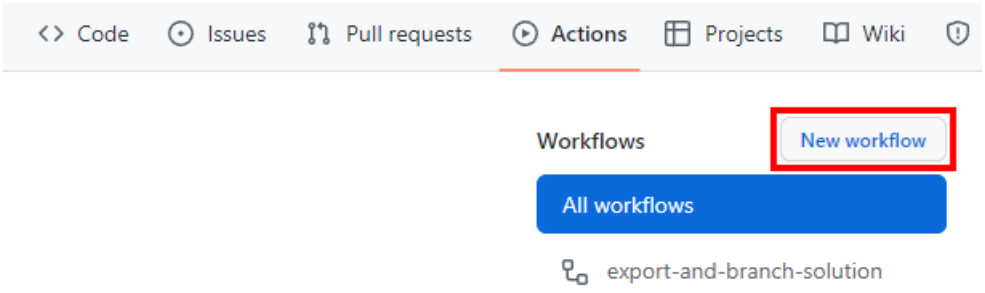
8. Navigate back to the default (main) branch and validate the solution is now available there as well.

Create a reusable workflow to generate a build artifact and import to production

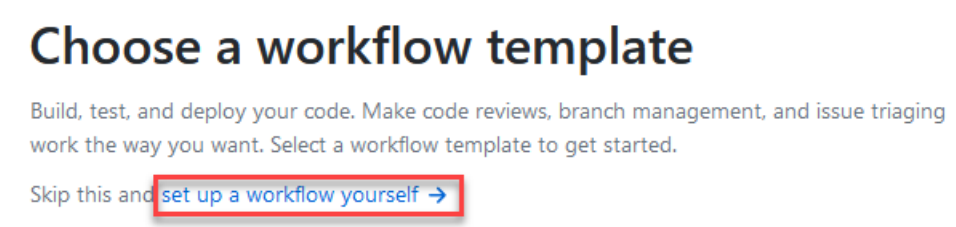
In this section, we will create an additional workflow that:

- Creates a managed solution and publishes it as a GitHub artifact
- Imports the build artifact into the production environment

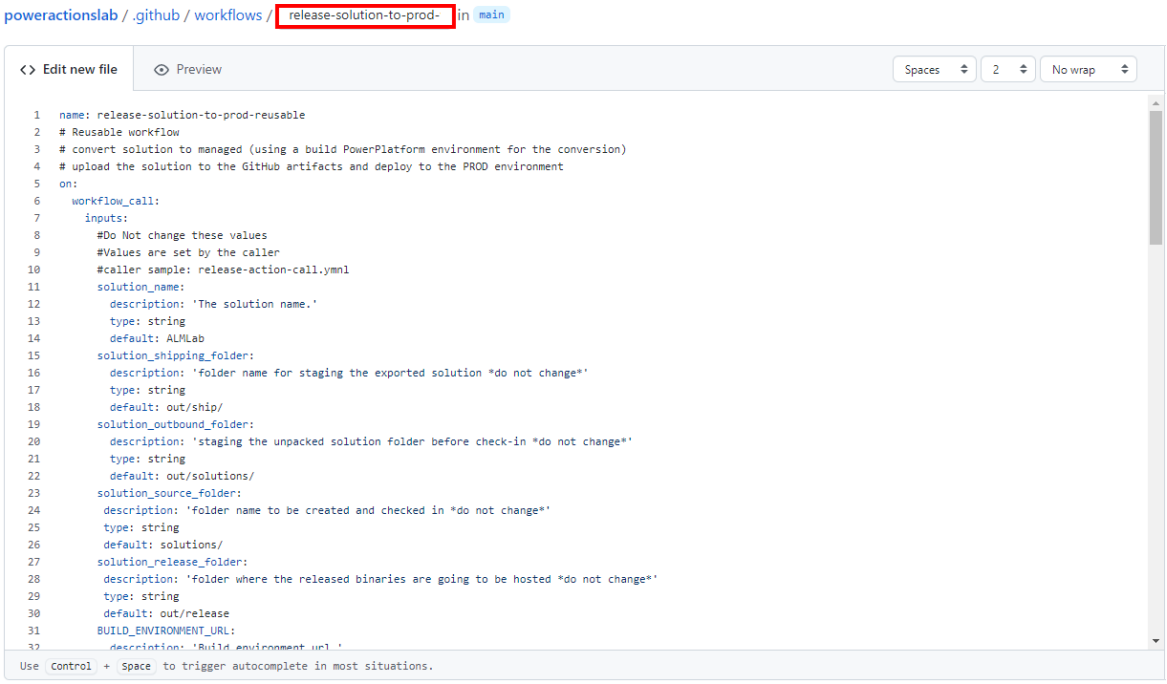
1. Navigate to **Actions** and select **New workflow**.



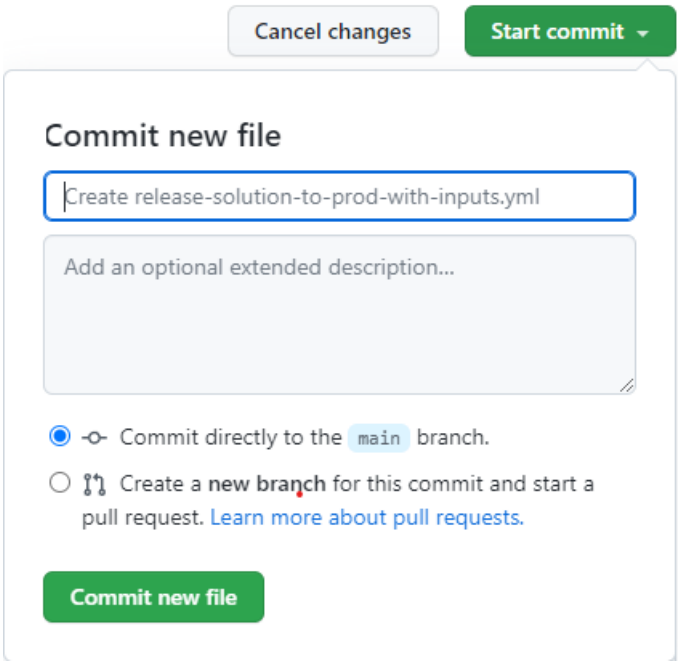
2. Chose **setup a workflow yourself**.



3. Rename the title of the workflow to 'release-solution-to-prod-with-inputs' and copy the content from the release-solution-to-prod-with-inputs.yml file and paste it into the **Edit new file** screen.



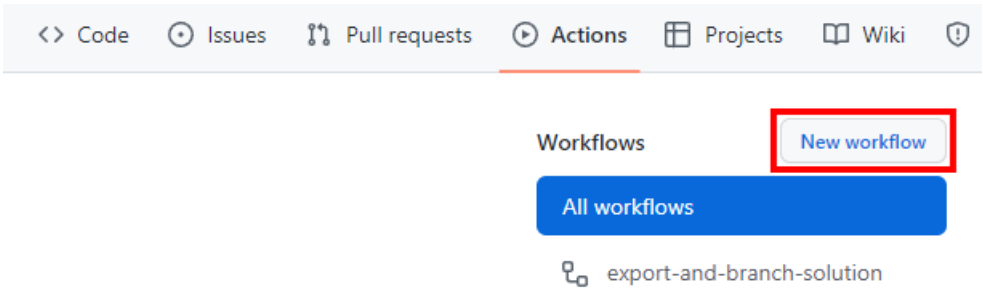
4. Commit the changes. Choose **Start commit** and then add a title and description (optional). Next, select **Commit new file**.



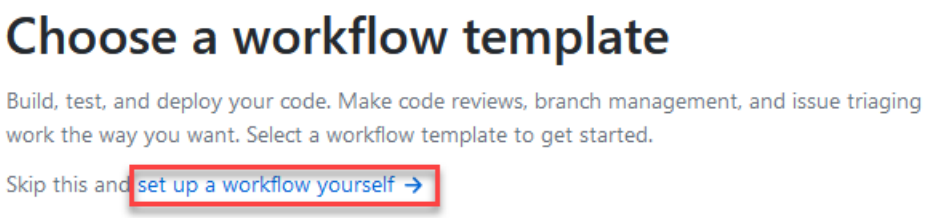
Call the reusable workflow on the release event

In this section, we will call the reusable workflow on the release event.

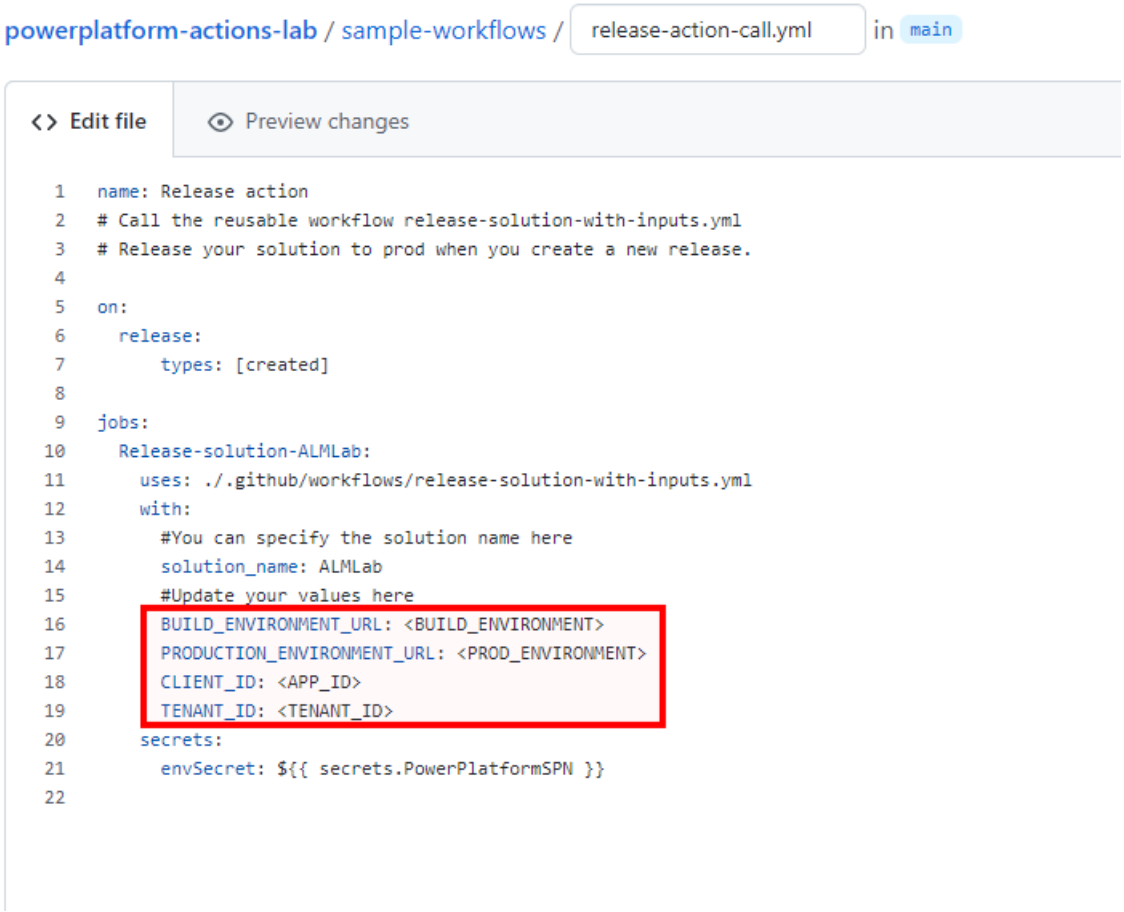
1. Navigate to **Actions** and select **New workflow**.



2. Choose **setup a workflow yourself**.



3. Rename the title of the workflow to 'release-action-call' and copy the content from the release-action-call.yml file and paste it into the **Edit new file** screen.

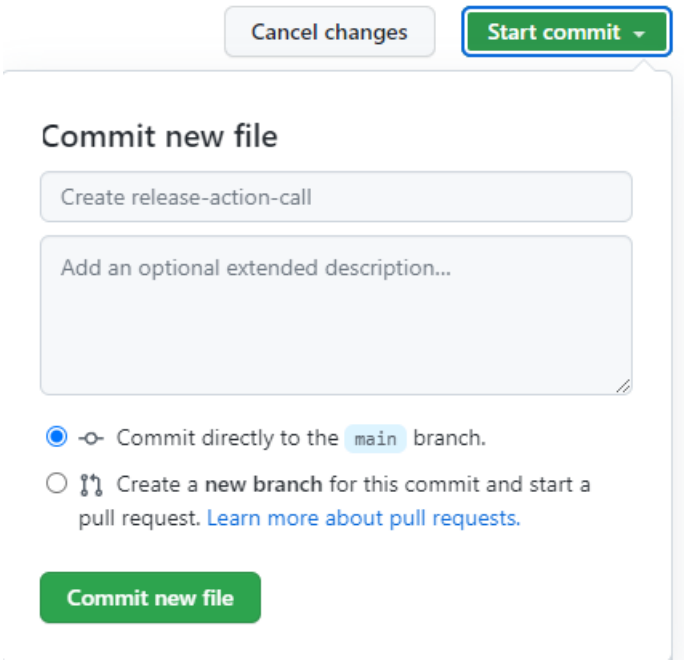


4. Update the following variables in the new workflow file:

- Update `<BUILD_ENVIRONMENT>` with the URL for the build environment you are using to generate the managed solution. For example: https://poweractionsbuild.crm.dynamics.com.
- Update `<PROD_ENVIRONMENT>` with the URL for the production environment you are deploying to. For example: https://poweractionsprod.crm.dynamics.com.

- Update `<APP_ID>` with the Application (Client) ID that can be found in the App registrations blade of the Microsoft Azure Portal by clicking into the registration created previously in this tutorial.
- Update `<TENANT_ID>` with the Directory (tenant) ID that can be found in the App registrations blade of the Microsoft Azure Portal by clicking into the registration created previously in this tutorial.

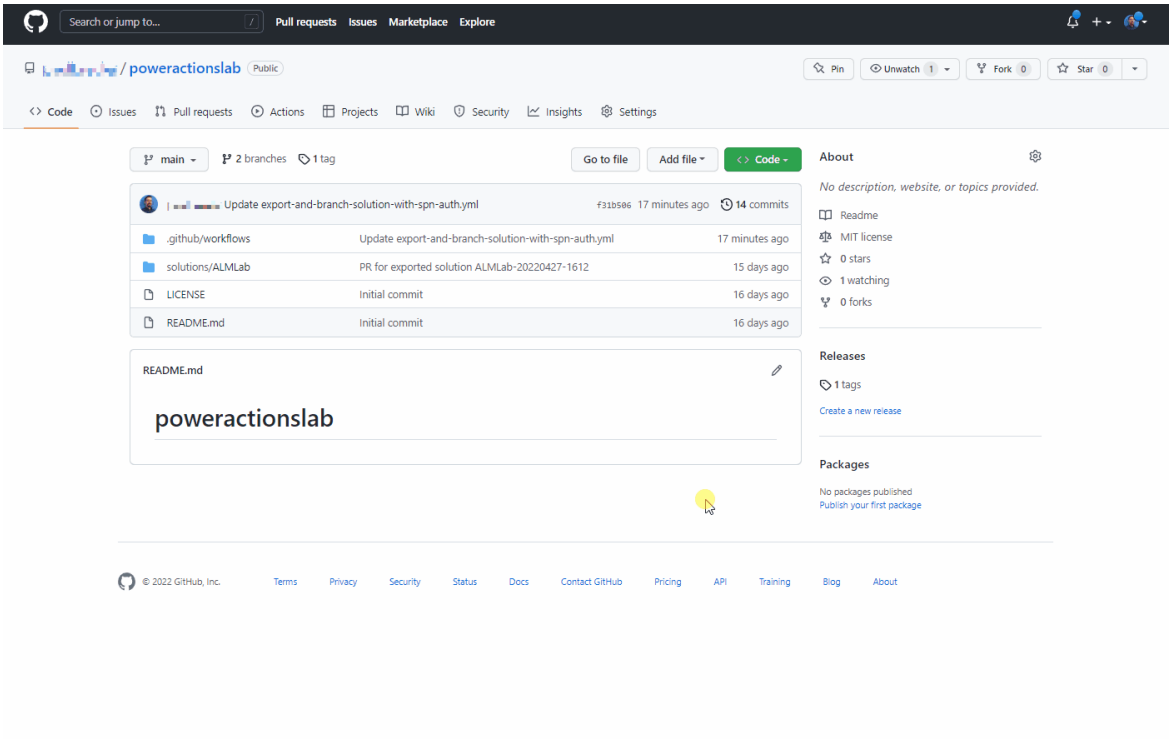
5. Commit the changes. Choose **Start commit** and then add a title (optional) and description (optional). Next, select **Commit new file**.



Test the release to production workflow

You are now ready to test the last workflow. This workflow is triggered when a new release is deployed to production.

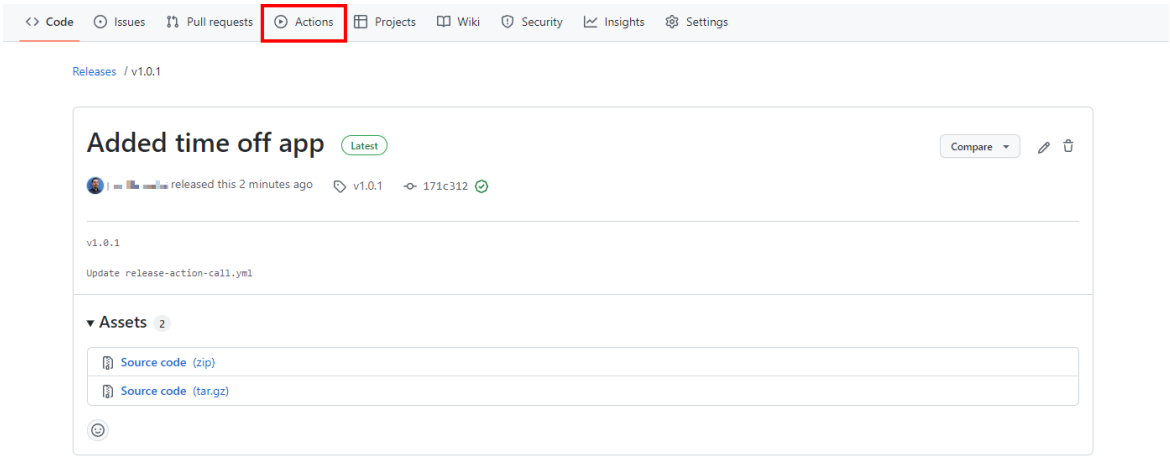
1. Navigate to **Releases**.



2. Select **Draft a new release**.

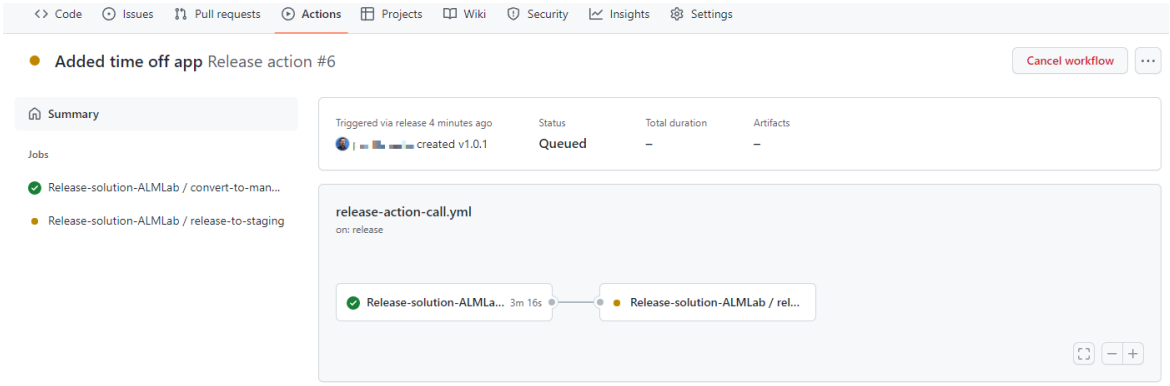
3. Add a release tag, a title, and choose **Publish release**.

4. Select **Actions** to view the running workflow.



5. Choose the running workflow to view the actions as they run.

6. Wait for each action to complete.

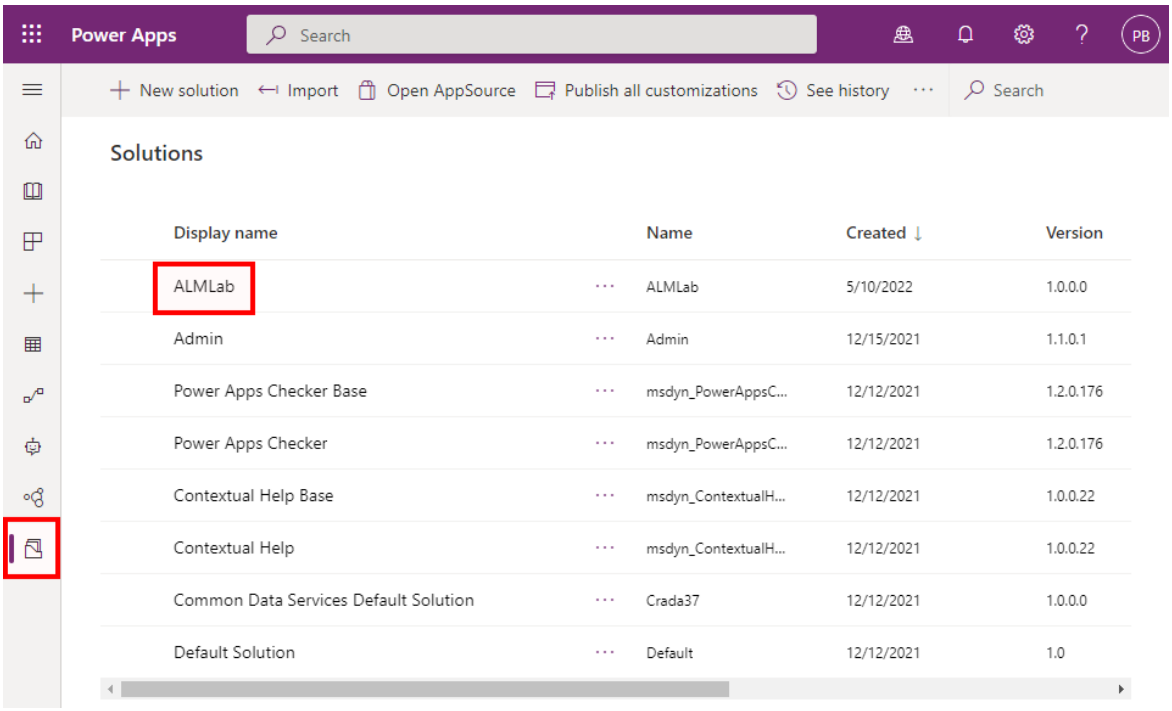


7. After the workflow has completed, log into your production environment and validate that the solution has been deployed as a managed solution.

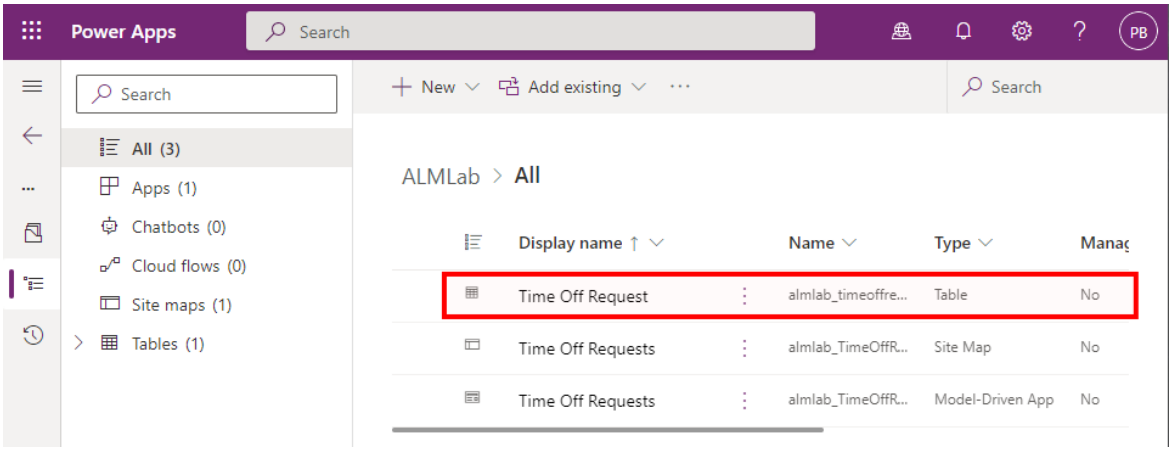
Deploy the update and review changes before production release

We will now test the end-to-end process and then see how we can view and validate changes to an app before it is deployed to production.

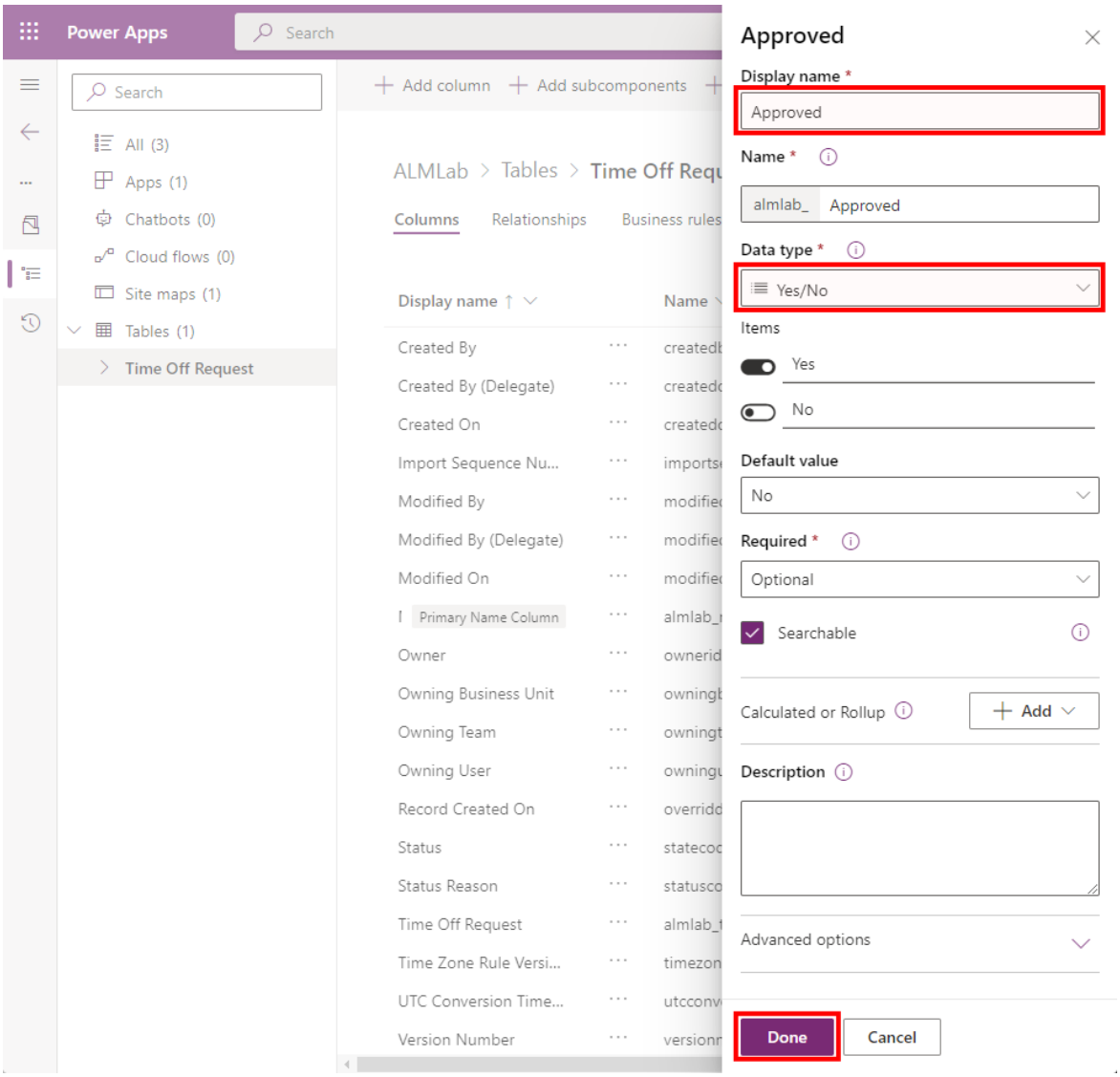
1. Navigate to the ALMLab solution in your development environment and choose **Edit** or click the solutions display name.



2. Select and view the **Time off Request** table.



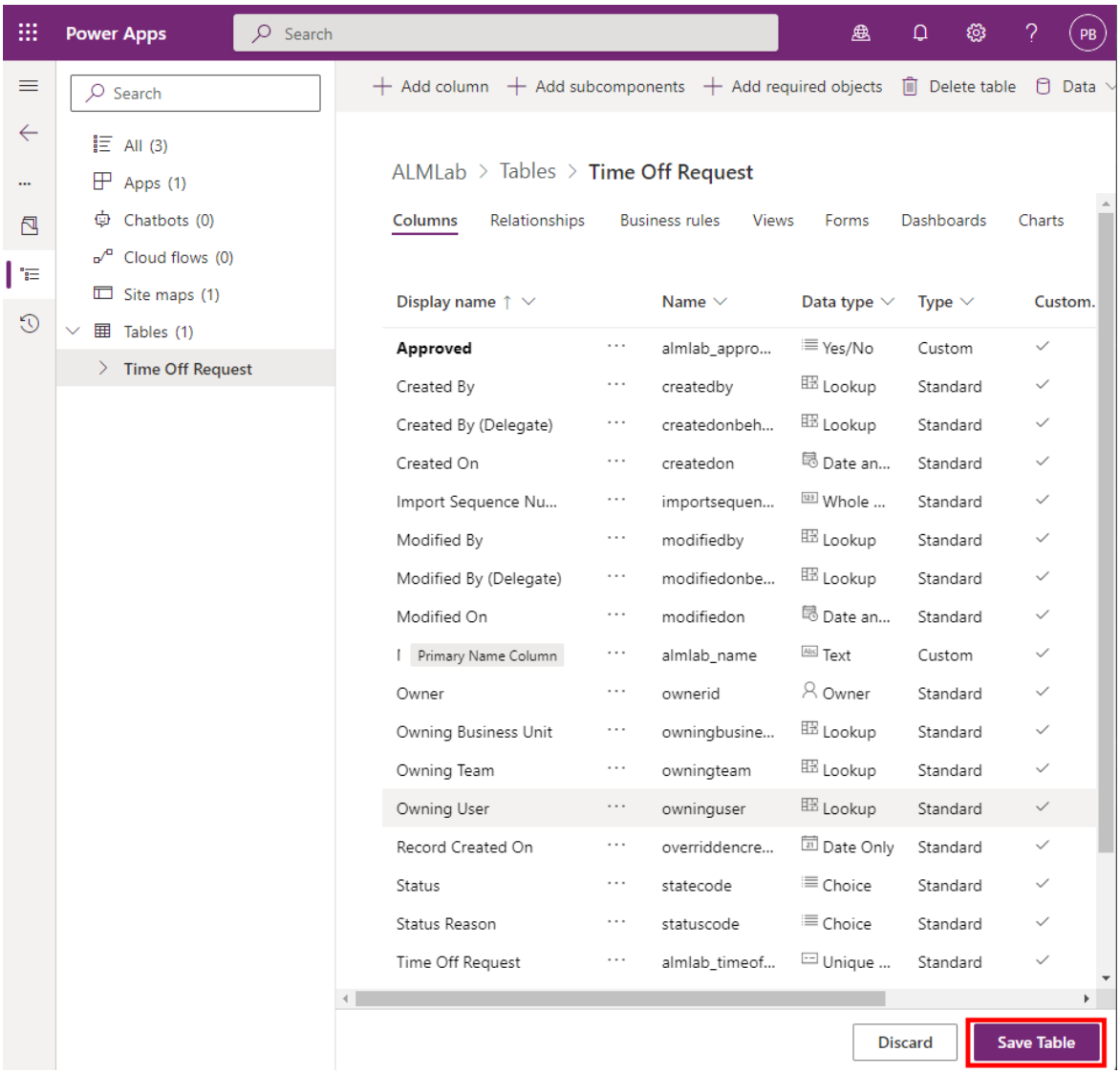
3. Click **+ Add column** and create the new column as shown in the figure below.



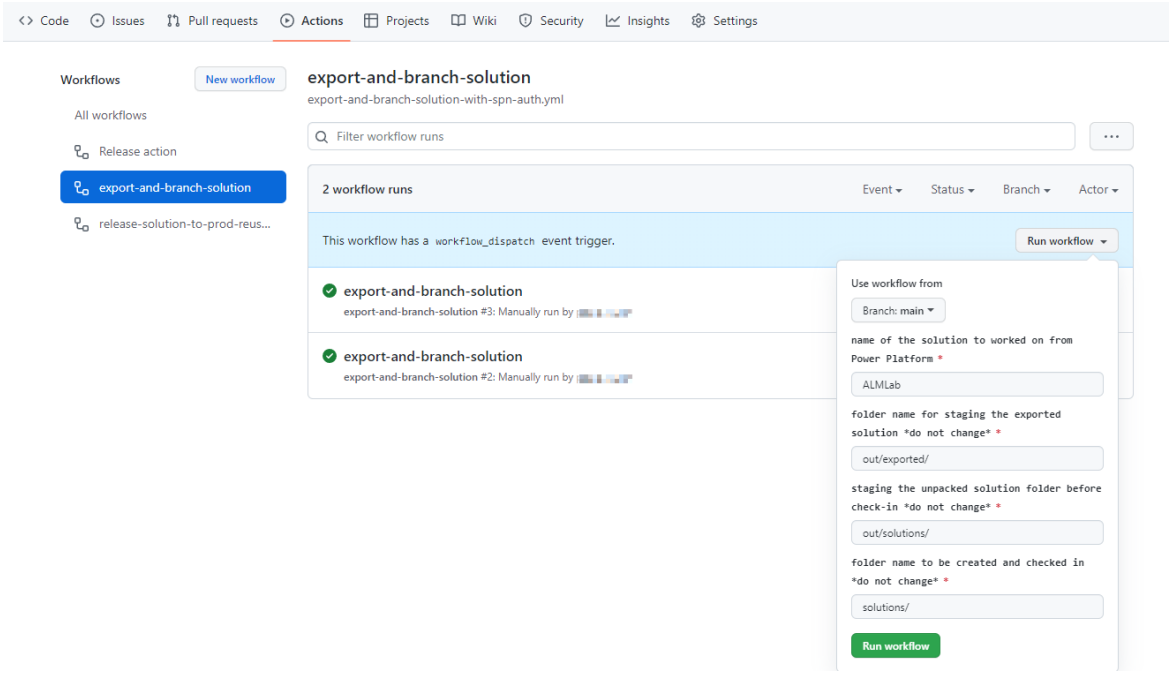
Updated Field values:

- **Display name**: Approved
- **Data type**: Yes/No

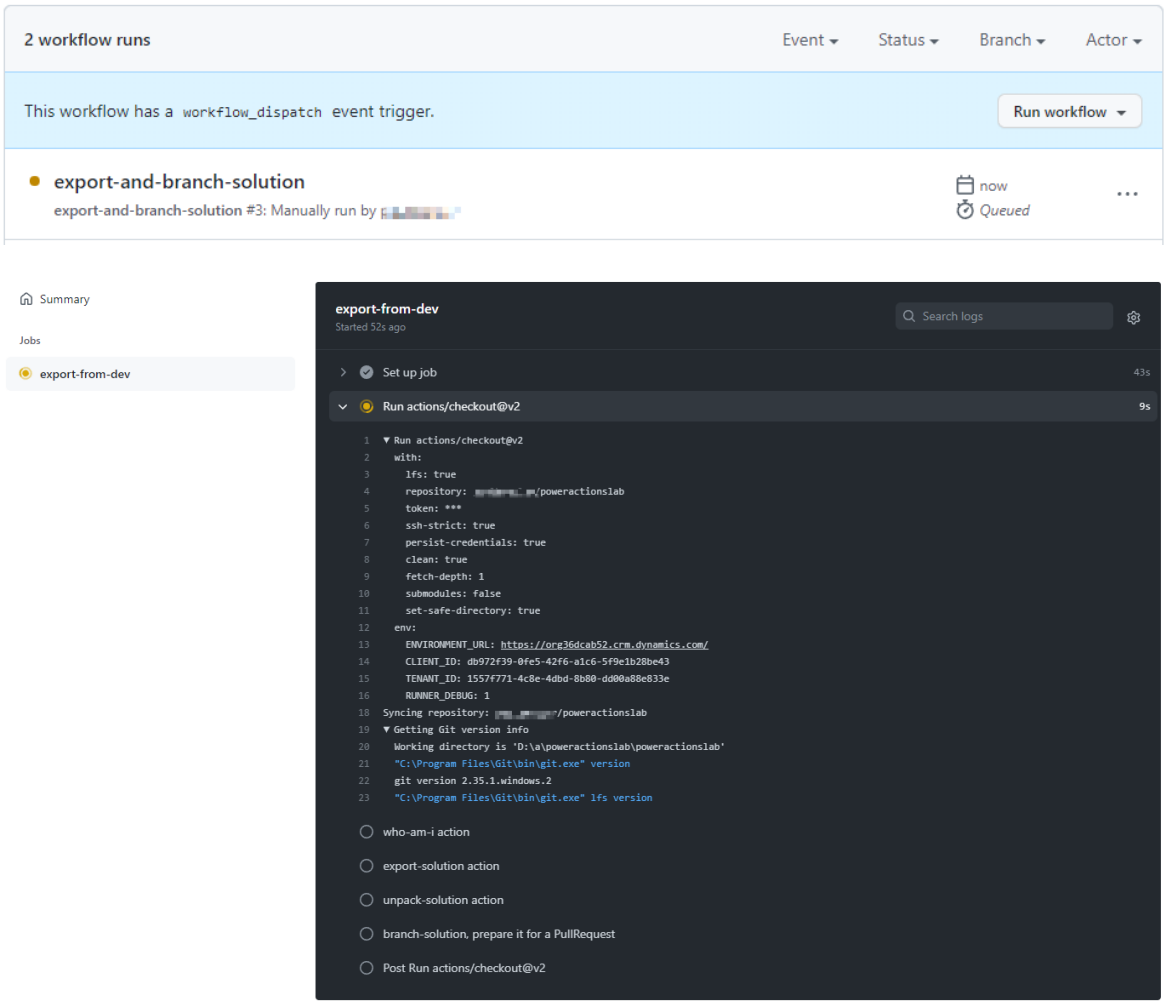
4. Click **Done**.
5. Click **Save Table**



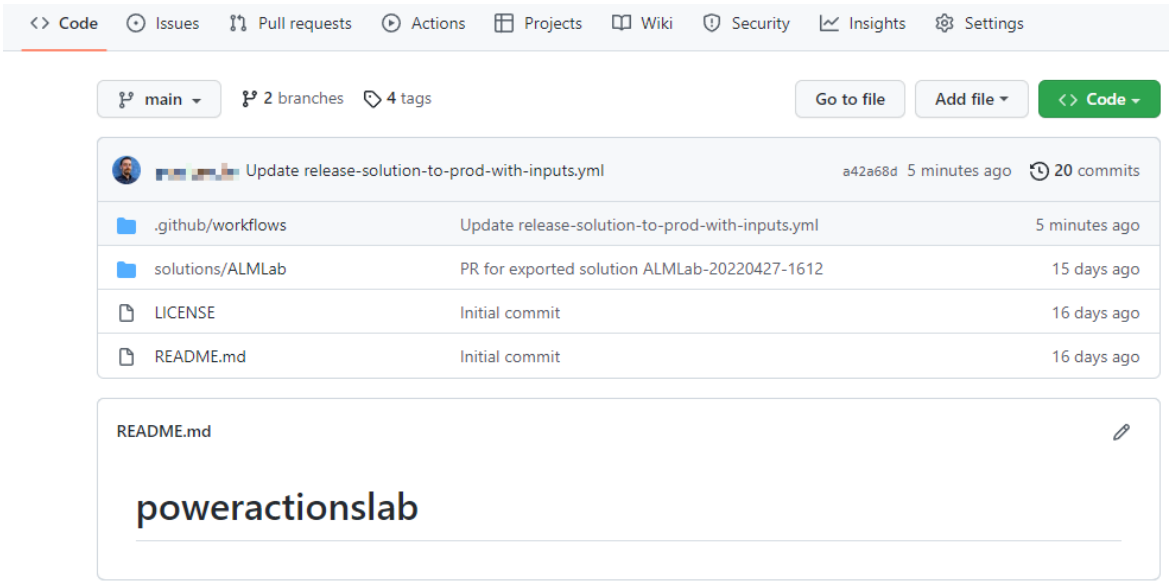
6. Navigate back to your GitHub repositories **Actions** tab, choose **Run workflow**, and select the **Run workflow** button.



7. After 5–10 seconds, the workflow will start and you can click on the running workflow to monitor its progress.

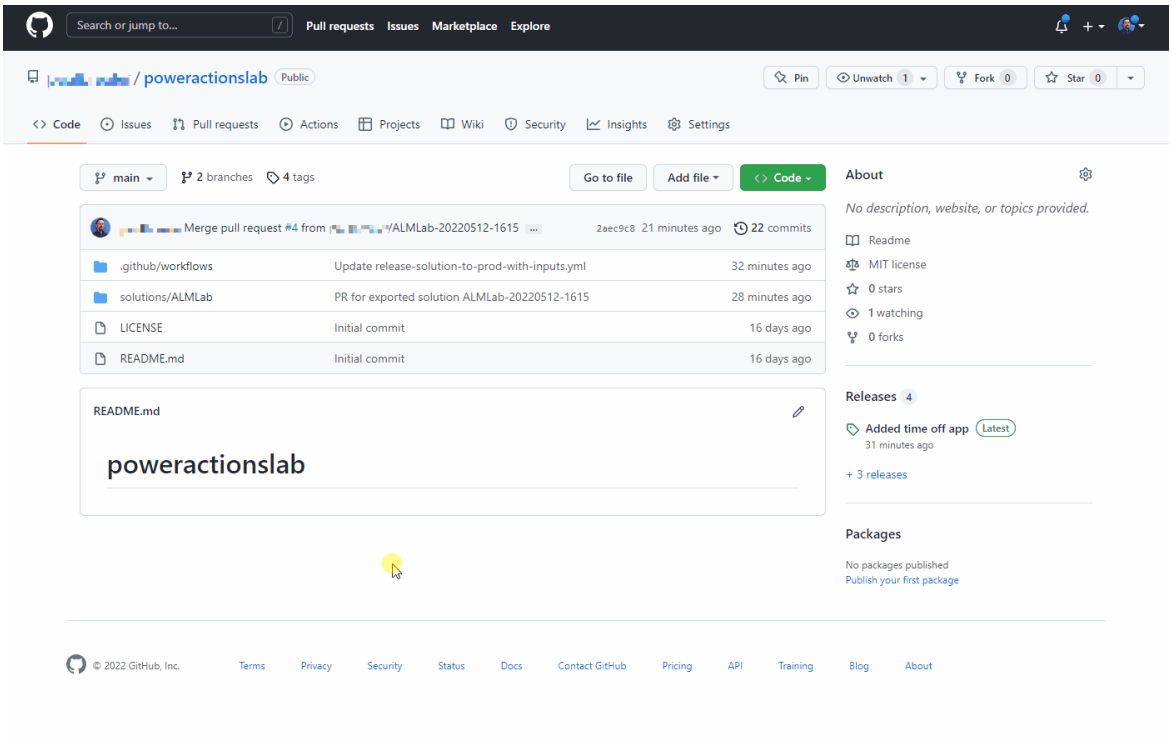


8. After the workflow completes, navigate to the new branch by selecting **Code** and then **Branches**.



9. Select the branch that was created by the expand **contribute** and click **Open pull request**.
10. Add a title (optional) and then click **Create pull request**.
11. After the content updates, click the **Files changed** tab.
12. Notice that the changes to the solution are highlighted in green to indicate that this section of the file was added when compared to the same file in the main branch.
13. Navigate back to the **Conversation** tab. Select **Pull requests** and then select the pull request previously created.

14. On the **Pull request** page, select **Squash and merge** to merge the updated solution file into your main branch, optionally delete that is now merged into main.



15. Follow the steps in the Test the release to production workflow section to create a new release and validate that the updated solution has been deployed to your production environment.

Congratulations, you have successfully setup a sample CI/CD workflow using GitHub actions!

See Also

Automate your workflow from idea to production

Power Apps component framework

7/15/2022 • 2 minutes to read • Edit Online

To be accessible by Power Apps makers, components in the Power Apps component framework must be packaged in a solution, exported, and then imported into a Power Apps environment with Dataverse. The following sections describe how to do this.

For more information about using ALM techniques with code components see Code components application lifecycle management (ALM).

Package and deploy a code component

This section describes how to import code components into Microsoft Dataverse so that the components are available to Power Apps makers.

After implementing the code components by using the Power Platform CLI, the next step is to pack all the code component elements into a solution file and import the solution file into Dataverse so that you can see the code components in the maker runtime experience.

To create and import a solution file:

1. Create a new folder in the folder that has the cdsproj file, and name it **Solutions** (or any name of your choice) by using the CLI command `mkdir Solutions`. Navigate to the directory by using the command `cd Solutions`.

2. Create a new solution project by using the following command. The solution project is used for bundling the code component into a solution zip (compressed) file that's used for importing into Dataverse.

   ```
   pac solution init --publisher-name \<enter your publisher name\>
   --publisher-prefix \<enter your publisher prefix\>
   ```

   NOTE

   The publisher-name and publisher-prefix values must be unique to your environment. More information: Solution publisher and Solution publisher prefix

3. After the new solution project is created, refer the **Solutions** folder to the location where the created sample component is located. You can add the reference by using the command shown below. This reference informs the solution project about which code components should be added during the build. You can add references to multiple components in a single solution project.

   ```
   pac solution add-reference --path \<path to your Power Apps component framework project\>
   ```dotnetcli
   ```

4. To generate a zip file from the solution project, go to your solution project directory and build the project by using the following command. This command uses the MSBuild program to build the solution project by pulling down the NuGet dependencies as part of the restore. Only use `/restore` the first time the solution project is built. For every build after that, you can run the command `msbuild`.
   ```

```
msbuild /t:build /restore
```

TIP

- If *MSBuild* 15.9.* is not in the path, open Developer Command Prompt for Visual Studio 2017 to run the `msbuild` commands.
- Building the solution in the *debug* configuration generates an unmanaged solution package. A managed solution package is generated by building the solution in *release* configuration. These settings can be overridden by specifying the SolutionPackageType property in the cdsproj file.
- You can set the `msbuild` configuration to **Release** to issue a production build. Example: `msbuild /p:configuration=Release`
- If you encounter an error that says "Ambiguous project name" when running the msbuild command on your solution, ensure that your solution name and project name aren't the same.

5. The generated solution files are located in the \bin\debug\ (or \bin\release) folder after the build is successful.

6. You can use the Microsoft Power Platform Build Tools to automate importing the solution into a Dataverse environment; otherwise, you can manually import the solution into Dataverse by using the web portal.

Additional tasks that you can do with the framework and solutions

Below are links to additional common tasks that you can do when working with the framework and solutions.

- Create a solution project based on an existing solution in Dataverse
- Create a plug-in project and add a reference to it in your solution
- Remove components from a solution

See also

Plug-ins

Plug-ins

7/15/2022 • 3 minutes to read • Edit Online

A solution is used to package and deploy plug-ins and custom workflow activities to other environments. For example, the sequence below defines a simplistic development and deployment sequence.

1. Create a custom publisher and unmanaged solution in your DEV environment.
2. Write one or more plug-ins or custom workflow activities.
3. Register the plug-ins or custom workflow activities in the unmanaged solution you created in step 1.
4. Export the unmanaged solution as a managed solution.
5. Import the managed solution into another environment (that is, TEST or PROD).

In the real world, you debug the code in the TEST environment, go back and update the unmanaged solution with the revised code, and export to a managed solution. Along the way you use revision control to manage the code updates and solution versions. For more information about revision control and versioning of solutions, see Source control.

When planning your solution design, consider whether you'll place your custom code and other customizations (customized entities, forms, views, and so on) in the same solution or you'll divide these customizations among multiple solutions, where one solution contains the custom code and another solution contains the other customizations (customized entities, forms, views, and so on).

> **TIP**
> Start with a custom publisher and unmanaged solution, and then develop and test the plug-in or custom workflow activity in that solution. We recommend against developing a plug-in or custom workflow activity in the default solution and then adding it to a custom solution.

Register a plug-in or custom workflow activity in a custom unmanaged solution

After you've created a custom publisher and unmanaged solution, and have written the custom code, you're ready to register the code in the unmanaged solution and begin testing it.

Register a custom workflow activity assembly

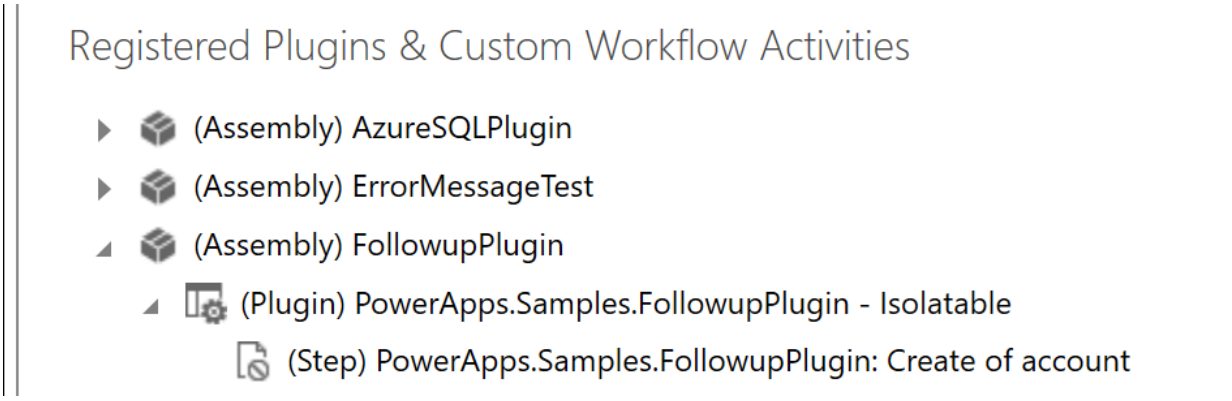
To distribute a custom workflow activity in a solution, you must add the registered assembly that contains it to an unmanaged solution. First, register the custom workflow assembly, and then add the assembly to a solution by following these steps.

1. Create a new solution in the Power Apps maker portal, or use an existing solution. To create a new solution, select **Solutions** > **New solution**, and then enter the required information for your solution.
2. With **Solutions** selected in the left navigation pane, select the solution name in the list, and then select **Add existing** > **Other** > **Plug-in assembly**.
3. Search for the compiled custom workflow assembly by name.
4. Select the custom workflow activity assembly, and then select **Add**.

Register a plug-in assembly and step

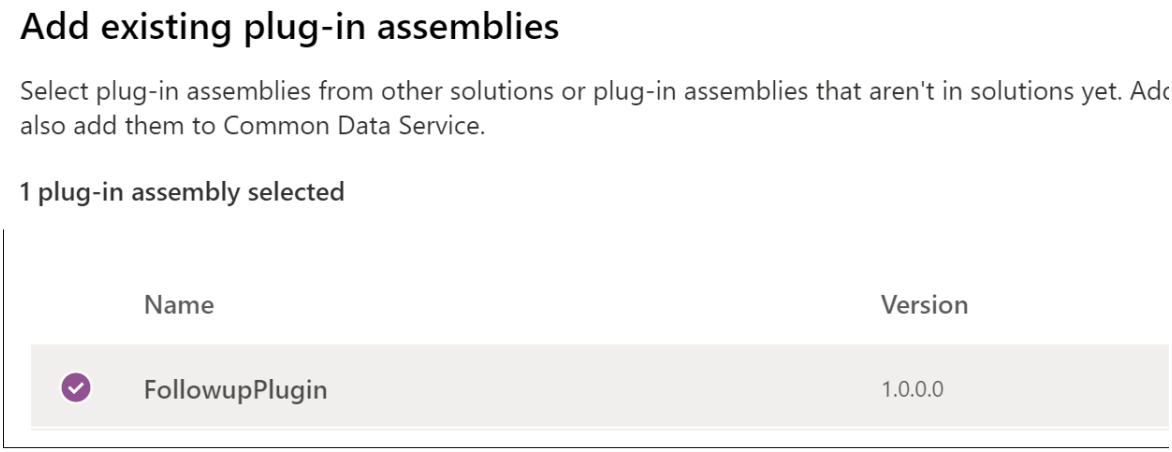
The procedure to register a plug-in is similar to registering a custom workflow activity assembly, except you must also register one or more *steps* which identify the conditions under which the plug-in should be executed by Microsoft Dataverse.

To begin, follow these instructions to register a plug-in and step by using the Plug-in Registration tool. Next, we'll create a solution and then add the plug-in assembly and step to the solution by using the modern maker interface. The example "Followup" plug-in and step registration we'll use is shown in the following illustration.



Let's get started adding those components to our solution.

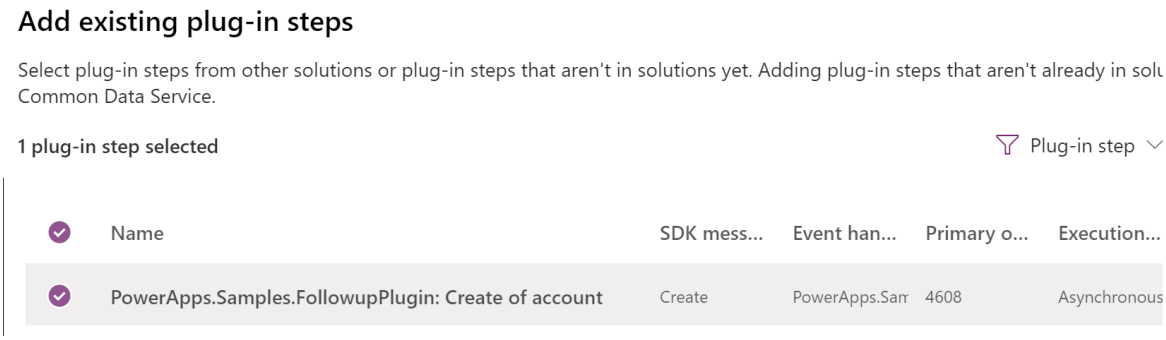
1. Create a new solution in the Power Apps maker portal, or use an existing solution. To create a new solution, select **Solutions** > **New solution**, and enter the required information for your solution.
2. With **Solutions** selected in the left navigation panel, select the solution name in the list, and then select **Add existing** > **Other** > **Plug-in assembly**.
3. Search for the compiled plug-in assembly by name.
4. Select the plug-in assembly, and then select **Add**.



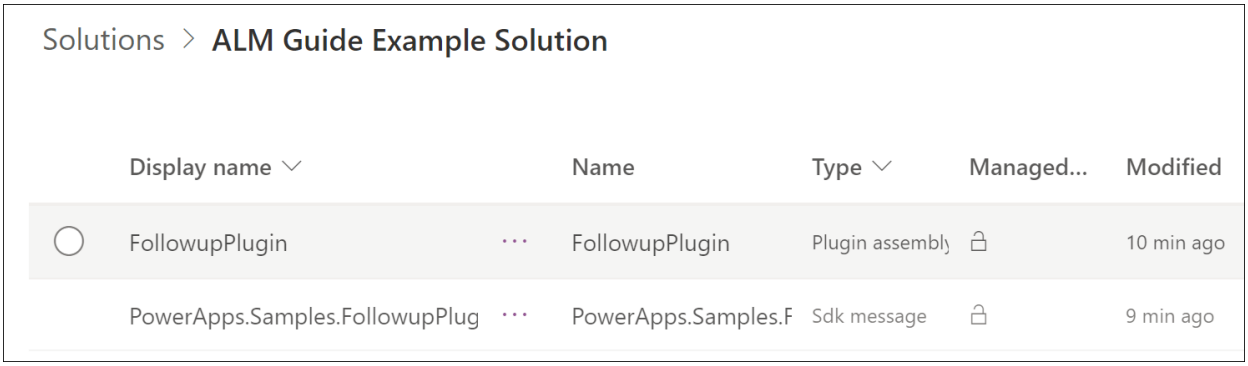
5. Add a step to the solution by selecting **Add existing** > **Other** > **Plug-in step**.

> **TIP**
> In the Plug-in Registration tool, a step is called a **step**. In the classic interface **Solution Explorer**, a step is called an **Sdk message processing step**. In the modern maker interface, a step is called an **Plug-in step**.

6. Search for the registered step, select it, and then select **Add**.



The resulting solution with the two components is shown in the following illustration.



It's possible to add the step to the solution before (or without) adding the related plug-in assembly. To add the assembly, select the option (...) menu next to the plug-in step name, select **Add required components**, and then select **OK**. This automatically adds the related plug-in assembly. Don't add the plug-in assembly to the solution if you intend to provide that assembly in another solution that the current solution will be dependent on.

Note that removing the plug-in assembly from the solution won't remove any steps that depend on it. You must remove those steps individually.

See also

Web resources

Web resources

7/15/2022 • 2 minutes to read • Edit Online

Solutions can be used to package and deploy web resources to other environments. To manually add a web resource to a solution, follow these steps.

1. Open the unmanaged solution you want to add the web resource to by using Power Apps
2. Select **Add existing** > **Other** > **Web resource**.
3. Search for the web resource by name.
4. Select the web resource, and then select **Add**

You can also add web resources to a solution by using code. Use the `SolutionUniqueName` optional parameter to associate a web resource with a specific solution when the web resource is created. This is demonstrated in this sample: Import files as web resources.

See also

Configuration Migration and Package Deployer tools

Use managed properties

7/15/2022 • 4 minutes to read • Edit Online

You can control which of your managed solution components are customizable by using managed properties. By default, all custom solution components are customizable. Each solution component has a **Can be customized** (`IsCustomizable`) property. As long as this property value is set to true, more properties specific to the type of solution component can be specified. If you set the `IsCustomizable.Value` property to false, after the solution is installed as a managed solution the solution component will not be customizable.

Managed properties ensure that only a solution from the same publisher will be able to change the component. Managed properties will only affect managed components and does not force this in the development environments where the component is still unmanaged. The use of the `IsCustomized` managed property is intended to ensure that no other solution layer from any other publisher, and no active customizations can be done on the component once it is installed as a managed solution.

The following table lists some managed properties for a subset of available solution components.

COMPONENT	DISPLAY NAME	PROPERTY
Entity	Can be customized	IsCustomizable. `Value`
Entity	Display name can be modified	IsRenameable. `Value`
Entity	Can be related entity in relationship	CanBeRelatedEntityInRelationship. `Value` (Read Only)
Entity	Can be primary entity in relationship	CanBePrimaryEntityInRelationship. `Value` (Read Only)
Entity	Can be in many-to-many relationship	CanBeInManyToMany. `Value` (Read Only)
Entity	New forms can be created	CanCreateForms. `Value`
Entity	New charts can be created	CanCreateCharts. `Value`
Entity	New views can be created	CanCreateViews. `Value`
Entity	Can change any other entity properties not represented by a managed property	CanModifyAdditionalSettings. `Value`
Entity	Can create attributes	CanCreateAttributes. `Value`
Field (Attribute)	Can be customized	IsCustomizable. `Value`
Field (Attribute)	Display name can be modified	IsRenameable. `Value`
Field (Attribute)	Can change requirement level	RequiredLevel. `CanBeChanged` Note: `RequiredLevel` is the only managed property to use the `CanBeChanged` property.
Field (Attribute)	Can change any other attribute properties not represented by a managed property	CanModifyAdditionalSettings. `Value`

COMPONENT	DISPLAY NAME	PROPERTY
Entity Relationship	Can be customized	IsCustomizable.`Value`
Form	Can be customized	`SystemForm.IsCustomizable.Value`
Chart	Can be customized	`SavedQueryVisualization.IsCustomizable.Value`
View	Can be customized	`SavedQuery.IsCustomizable.Value`
Option Set	Can be customized	IsCustomizable.`Value`
Web Resource	Can be customized	`WebResource.IsCustomizable.Value`
Workflow	Can be customized	`Workflow.IsCustomizable.Value`
Workflow	Is Custom Processing Step Allowed For Other Publishers	`Workflow.IsCustomProcessingStepAllowedForOtherPublish`
Assembly	Can be customized	`SdkMessageProcessingStep.IsCustomizable.Value`
Assembly Registration	Can be customized	`ServiceEndpoint.IsCustomizable.Value`
E-mail Template	Can be customized	`Template.IsCustomizable.Value`
KB Article Template	Can be customized	`KbArticleTemplate.IsCustomizable.Value`
Contract Template	Can be customized	`ContractTemplate.IsCustomizable.Value`
Mail Merge Template	Can be customized	`MailMergeTemplate.IsCustomizable.Value`
Dashboard	Can be customized	`SystemForm.IsCustomizable.Value`
Security Roles	Can be customized	`Role.IsCustomizable.Value`
System Form	Can be deleted	CanBeDeleted.`Value`
System Query	Can be deleted	CanBeDeleted.`Value`

Workflow Is Custom Processing Step Allowed For Other Publishers

This managed property controls whether plug-in steps registered by 3rd parties for messages created by custom process actions will run. The default value is `false`, which means plug-in steps registered which do not use the same solution publisher will not run. When this is `true`, the publisher of the custom process action allows registered plug-in registration steps to run.

Update managed properties

After you release your managed solution, you may decide that you want to change the managed properties. You can only change managed properties to make them less restrictive. For example, after your initial release you can decide to allow customization of an entity.

You update managed properties for your solution by releasing an update to your solution with the changed managed properties. Your managed solution can only be updated by another managed solution associated with the same publisher record as the original managed solution. If your update includes a change in managed properties to make them more restrictive, those managed property changes will be ignored but other changes in the update will be applied.

Because the original publisher is a requirement to update managed properties for a managed solution, any unmanaged solution cannot be associated with a publisher that has been used to install a managed solution.

> **NOTE**
>
> This means you will not be able to develop an update for your solution by using an organization where your managed solution is installed.

Check managed properties

Use IsComponentCustomizableRequest to check whether a solution component is customizable. Alternatively, you can check the solution component properties but you must consider that the ultimate determination of the meaning depends on the values of several properties. Each solution component has an `IsCustomizable` property. When a solution component is installed as part of a managed solution, the `IsManaged` property will be true. Managed properties are only enforced for managed solutions. When checking managed properties to determine if an individual solution component is customizable, you must check both the `IsCustomizable` and `IsManaged` properties. A solution component where `IsCustomizable` is false and `IsManaged` is false, is customizable.

Entity and attribute have more managed properties in addition to `IsCustomizable`. These managed properties are not updated if `IsCustomizable` is set to false. This means that in addition to checking the individual managed property, you must also check the `IsCustomizable` property to see if the managed property is being enforced.

See also

Managed properties

Dependency tracking for solution components

7/15/2022 • 9 minutes to read • Edit Online

Solutions are made of solution components. You'll use the **Solutions** area in Microsoft Dataverse to create or add solution components. You can perform these actions programmatically by using the AddSolutionComponentRequest message or any messages that create or update solution components that include a `SolutionUniqueName` parameter.

Solution components often depend on other solution components. You can't delete any solution component that has dependencies on another solution component. For example, a customized ribbon typically requires image or script web resources to display icons and perform actions using scripts. As long as the customized ribbon is in the solution, the specific web resources it uses are required. Before you can delete the web resources you must remove references to them in the customized ribbon. These solution component dependencies can be viewed in the application by clicking **Show Dependencies**.

This topic describes the types of solution components you can include in your solutions and how they depend on each other.

All solution components

The complete list of available solutions component types is located in the system `componenttype` global option set. The supported range of values for this property is available by including the file `OptionSets.cs` or `OptionSets.vb` in your project. However, many of the solution component types listed there are for internal use only and the list doesn't provide information about the relationships between solution components.

Solution component dependencies

Solution component dependencies help make sure you have a reliable experience working with solutions. They prevent actions that you normally perform from unintentionally breaking customizations defined in a solution. These dependencies are what allow a managed solution to be installed and uninstalled simply by importing or deleting a solution.

The solutions framework automatically tracks dependencies for solution components. Every operation on a solution component automatically calculates any dependencies to other components in the system. The dependency information is used to maintain the integrity of the system and prevent operations that could lead to an inconsistent state.

As a result of dependency tracking the following behaviors are enforced:

- Deletion of a component is prevented if another component in the system depends on it.
- Export of a solution warns the user if there are any missing components that could cause failure when importing that solution in another system.

  Warnings during export can be ignored if the solution developer intends that the solution is only to be installed in an organization where dependent components are expected to exist. For example, when you create a solution that is designed to be installed over a pre-installed "base" solution.
- Import of a solution fails if all required components aren't included in the solution and also don't exist in the target system.

- Additionally, when you import a managed solution all required components must match the package type of the solution. A component in a managed solution can only depend on another managed component.

There are three types of solution component dependencies:

Solution Internal
Internal dependencies are managed by Dataverse. They exist when a particular solution component can't exist without another solution component.

Published
Published dependencies are created when two solution components are related to each other and then published. To remove this type of dependency, the association must be removed and the entities published again.

Unpublished
Unpublished dependencies apply to the unpublished version of a publishable solution component that is being updated. After the solution component is published, it becomes a published dependency.

Solution internal dependencies are dependencies where actions with a solution component require an action for another solution component. For example, if you delete an entity, you should expect that all the entity attributes will be deleted with it. Any entity relationships with other entities will also be deleted.

However, an internal dependency may lead to a published dependency and still require manual intervention. For example, if you include a lookup field on an entity form and then delete the primary entity in the relationship, you can't complete that deletion until you remove the lookup field from the related entity form and then publish the form.

When you perform actions programmatically with solutions, you can use messages related to the `Dependency` entity. See Dependency Entity for messages you can use to identify dependencies that may exist before you delete a component or uninstall a solution.

Common Solution components

These are the solution components displayed in the application and the components that you'll work with directly when adding or removing solution components using the solution page. Each of the other types of solution components will depend on one or more of these solution components to exist.

Application Ribbons (RibbonCustomization)	Entity (Entity)	Report (Report)
Article Template (KBArticleTemplate)	Field Security Profile (FieldSecurityProfile)	SDK Message Processing Step (SDKMessageProcessingStep)
Connection Role (ConnectionRole)	Mail Merge Template (MailMergeTemplate)	Security Role (Role)
Contract Template (ContractTemplate)	Option Set (OptionSet)	Service Endpoint (ServiceEndpoint)
Dashboard or Entity Form (SystemForm)	Plug-in Assembly (PluginAssembly)	Site Map (SiteMap)
Email Template (EmailTemplate)	Process (Workflow)	Web Resource (WebResource)

Application ribbons (RibbonCustomization)

Ribbon customizations for the application ribbon and entity ribbon templates. Application ribbons don't include definitions of ribbons at the entity or form level.

Custom application ribbons frequently have published dependencies on web resources. Web resources are used to define ribbon button icons and JavaScript functions to control when ribbon elements are displayed or what actions are performed when a particular ribbon control is used. Dependencies are only created when the ribbon definitions use the `$webresource:` directive to associate the web resource to the ribbon. More information: $webresource directive

Article template (KBArticleTemplate)

Template that contains the standard attributes of an article. There is always an internal dependency between the article template and the KbArticle entity.

Connection role (ConnectionRole)

Role describing a relationship between a two records. Each connection role defines what types of entity records can be linked using the connection role. This creates a published dependency between the connection role and the entity.

Contract template (ContractTemplate)

Template that contains the standard attributes of a contract. There is always an internal dependency between the contract template and the contract entity.

Dashboard or entity form (SystemForm)

System form entity records are used to define dashboards and entity forms. When a SystemForm is used as an entity form there is an internal dependency on the entity. When a SystemForm is used as a dashboard there are no internal dependencies. Both entity forms and dashboards commonly have published dependencies related to their content. An entity form may have lookup fields that depend on an entity relationship. Both dashboards and entity forms may contain charts or subgrids that will create a published dependency on a view, which then has an internal dependency on an entity. A published dependency on web resources can be created because of content displayed within the dashboard or form or when a form contains JavaScript libraries. Entity forms have published dependencies on any attributes that are displayed as fields in the form.

Email template (EmailTemplate)

Template that contains the standard attributes of an email message. An email template typically includes fields that insert data from specified entity attributes. An email template can be linked to a specific entity when it is created so there can be an internal dependency on the entity. A global email template isn't associated with a specific entity, but it may have published dependencies on entity attributes used to provide data. A process (workflow) frequently is configured to send an email using an email template creating a published dependency with the workflow.

Entity (Entity)

The primary structure used to model and manage data in Dataverse. Charts, forms, entity relationships, views, and attributes associated with an entity are deleted automatically when the entity is deleted because of the internal dependencies between them. Entities frequently have published dependencies with processes, dashboards, and email templates.

Field security profile (FieldSecurityProfile)

Profile that defines the access level for secured attributes.

Mail merge template (MailMergeTemplate)

Template that contains the standard attributes of a mail merge document. A mail merge template has a published dependency on the entity it's associated with.

Option set (OptionSet)

An option set defines a set of options. A picklist attribute uses an option set to define the options provided. Several picklist attributes may use a global option set so that the options they provide are always the same and can be maintained in one place. A published dependency occurs when a picklist attribute references a global option set. You can't delete a global option set that is being used by a picklist attribute.

Plug-in assembly (PluginAssembly)

Assembly that contains one or more plug-in types. Plug-ins are registered to events that are typically associated with an entity. This creates a published dependency.

Process (Workflow)

Set of logical rules that define the steps necessary to automate a specific business process, task, or set of actions to be performed. Processes provide a wide range of actions that create published dependencies on any other solution component referenced by the process. Each process also has a published dependency on the entity it's associated with.

Report (Report)

Data summary in an easy-to-read layout. A report has published dependencies on any entity or attribute data included in the report. Each report must also be associated with a Report category creating an internal dependency on a solution component called Report Related Category (ReportCategory). Reports may be configured to be subreports creating a published dependency with the parent report.

SDK message processing step (SDKMessageProcessingStep)

Stage in the execution pipeline that a plug-in is to execute.

Security role (Role)

Grouping of security privileges. Users are assigned roles that authorize their access to the Dataverse system. Entity forms can be associated with specific security roles to control who can view the form. This creates a published dependency between the security role and the form.

> **NOTE**
> Only security roles from the organization business unit can be added to a solution. Only a user with read access to those security roles can add them to a solution.

Service endpoint (ServiceEndpoint)

Service endpoint that can be contacted.

Site map (SiteMap)

XML data used to control the application navigation pane. The site map may be linked to display an HTML web resource or an icon in the site map may use an image web resource. When the `$webresource:` directive is used to establish these associations a published dependency is created. More information: $webresource directive

Web resource (WebResource)

Data equivalent to files used in web development. Web resources provide client-side components that are used to provide custom user interface elements. Web resources may have published dependencies with entity forms, ribbons and the SiteMap. When the `$webresource:` directive is used to establish associations in a ribbon or the SiteMap, a published dependency is created. For more information, see $webresource directive.

> **NOTE**
>
> Web resources may depend on other web resources based on relative links. For example, an HTML web resource may use a CSS or script web resource. A Silverlight web resource displayed outside of an entity form or chart must have an HTML web resource to host it. These dependencies aren't tracked as solution dependencies.

See also

Solution concepts
Removing dependencies
Environment strategy
Work with solutions using the SDK APIs]
Solution Entity Reference
SolutionComponent Entity Reference

Check for solution component dependencies

7/15/2022 • 2 minutes to read • Edit Online

When you edit solutions you may find that you can't delete a solution component because it has a published dependency with another solution component. Or, you may not be able to uninstall a managed solution because one of the components in the managed solution has been used in a customization in another unmanaged solution.

The following table lists the messages that you can use to retrieve data about solution component dependencies.

MESSAGE	DESCRIPTION
RetrieveDependentComponentsRequest	Returns a list of dependencies for solution components that directly depend on a solution component. For example, when you use this message for a global option set solution component, dependency records for solution components representing any option set attributes that reference the global option set solution component are returned. When you use this message for the solution component record for the account entity, dependency records for all of the solution components representing attributes, views, and forms used for that entity are returned.
RetrieveRequiredComponentsRequest	Returns a list of the dependencies for solution components that another solution component directly depends on. This message provides the reverse of the `RetrieveDependentComponentsRequest` message.
RetrieveDependenciesForDeleteRequest	Returns a list of all the dependencies for solution components that could prevent deleting a solution component.
RetrieveDependenciesForUninstallRequest	Returns a list of all the dependencies for solution components that could prevent uninstalling a managed solution.

Create solutions that support multiple languages

7/15/2022 • 11 minutes to read • Edit Online

Microsoft Dataverse supports multiple languages. If you want your solution to be installed for organizations that include different base languages or that have multiple languages provisioned, take this into account when planning your solution. The following table lists tactics to use along with solution components to include in a solution that supports multiple languages.

TACTIC	SOLUTION COMPONENT TYPE
String (RESX) web resources	Web Resources
Embedded labels	Application Navigation (SiteMap) Ribbons
Export and import translations	Attributes Charts Dashboards Entity Entity Relationships Forms Messages Option Sets Views
Localization in base language strings	Contract Templates Connection Roles Processes (Workflow) Security Roles Field Security Profiles
Localization not required	SDK Message Processing Steps Service Endpoints
Separate component for each language	Article Templates Email Templates Mail Merge Templates Reports Dialogs
Use XML web resources as language resources	Plug-in Assemblies

The following sections provide additional details for each tactic.

String (RESX) web resources

With string (RESX) web resources added with Dataverse developers have a more robust option to create web resources that support multiple languages. More information String (RESX) web resources.

Embedded labels

Each of the solution components that use this tactic requires that any localized text be included in the solution

component.

Ribbons

When a language pack is installed, the application ribbon automatically displays localized text for all of the default text in the ribbon. System labels are defined in a `ResourceId` attribute value that is for internal use only. When you add your own text you should use the `<LocLabels>` element to provide localized text for the languages you support. More information: Use Localized Labels with Ribbons

SiteMap

When a language pack is installed, the default text in the application navigation bar automatically displays localized text. To override the default text or to provide your own text, use the `<Titles>` element. The `Titles` element should contain a `<Title>` element that contains localized text for any languages your solution supports. If a `Title` element isn't available for the user's preferred language, the title that corresponds to the organization base language is displayed.

The `<SubArea>` element allows for passing the user's language preference by using the `userlcid` parameter so that content that is the target of the `SubArea.Url` attribute can be aware of the user's language preference and adjust accordingly. More information: Passing Parameters to a URL Using SiteMap

Export and import translations

Localizable labels for the solution components in the following table can be exported for localization.

Entities	Attributes	Relationships
Global Option Sets	Entity Messages	Entity Forms
Entity Views (SavedQuery)	Charts	Dashboards

Translating labels and display strings

You can only perform customizations in the application by using the base language. Therefore, when you want to provide localized labels and display strings for these customizations, you must export the text of the labels so that they can be localized for any other languages enabled for the organization. Use the following steps:

1. Ensure that the organization that you're working on has all the MUI packs installed and languages provisioned for languages you want to provide translations for.
2. Create your solution and modify the components.
3. After you have finished developing your solution use the "Export Translations" functionality. This generates a Office Excel spreadsheet (CrmTranslations.xml) that contains all the labels that need translation.
4. In the spreadsheet, provide the corresponding translations.
5. Import translations back into the same Dataverse organization using the "Import Translations" functionality and publish your changes.
6. The next time the solution is exported it carries all the translations that you provided.

   When a solution is imported, labels for languages that aren't available in the target system are discarded and a warning is logged.

   If labels for the base language of the target system are not provided in the solution package, the labels of

the base language of the source are used instead. For example, if you import a solution that contains labels for English and French with English as the base language, but the target system has Japanese and French with Japanese as the base language, English labels are used instead of Japanese labels. The base languages labels cannot be **null** or empty.

Exporting translations

Before you export translations you must first install the language packs and provision all the languages you want to have localized. You can export the translations in the web application or by using the ExportTranslationRequest message. For more information, see Export Customized Entity and Field Text for Translation.

Translating text

When you open the CrmTranslations.xml file in Office Excel you will see the three worksheets listed in the following table.

WORKSHEET	DESCRIPTION
Information	Displays information about the organization and solution that the labels and strings were exported from.
Display Strings	Display strings that represent the text of any messages associated with a metadata component. This table includes error messages and strings used for system ribbon elements.
Localized Labels	Displays all of the text for any metadata component labels.

You can send this file to a linguistic expert, translation agency, or localization firm. They will need to provide localized strings for any of the empty cells.

> **NOTE**
> For custom entities there are some common labels that are shared with system entities, such as **Created On** or **Created By**. Because you have already installed and provisioned the languages, if you export languages for the default solution you may be able to match some labels in your custom entities with localized text for identical labels used by other entities. This can reduce the localization costs and improve consistency.

After the text in the worksheets has been localized, add both the CrmTranslations.xml and [Content_Types].xml files to a single compressed .zip file. You can now import this file.

If you prefer to work with the exported files programmatically as an XML document, see Word, Excel, and PowerPoint Standards Support for information about the schemas these files use.

Importing translated text

> **IMPORTANT**
> You can only import translated text back into the same organization it was exported from.

After you have exported the customized entity or attribute text and had it translated, you can import the translated text strings in the web application by using the ImportTranslationRequest message. The file that you import must be a compressed file that contains the CrmTranslations.xml and the [Content_Types].xml file at the root. For more information, see Import Translated Entity and Field Text.

After you import the completed translations, customized text appears for users who work in the languages that you had the text translated into.

> **NOTE**
>
> Dataverse cannot import translated text that is over 500 characters long. If any of the items in your translation file are longer than 500 characters, the import process fails. If the import process fails, review the line in the file that caused the failure, reduce the number of characters, and try to import again.

Because customization is supported only in the base language, you may be working in Dataverse with the base language set as your language preference. To verify that the translated text appears, you must change your language preference for the Dataverse user interface. To perform additional customization work, you must change back to the base language.

Localization in base language strings

Some solution components do not support multiple languages. These components include names or text that can only be meaningful in a specific language. If you create a solution for a specific language, define these solution components for the intended organization base language.

If you need to support multiple languages, one tactic is to include localization within the base language strings. For example, if you have a Connection Role named "Friend" and you need to support English, Spanish, and German, you might use the text "Friend (Amigo / Freund)" as the name of the connection role. Due to issues of the length of the text there are limitations on how many languages can be supported using this tactic.

Some solution components in this group are only visible to administrators. Because customization of the system can only be done in the organization base language it is not necessary to provide multiple language versions. **Security Roles** and **Field Security Profile** components belong to this group.

Contract Templates provide a description of a type of service contract. These require text for the **Name** and **Abbreviation** fields. You should consider using names and abbreviations that are unique and appropriate for all users of the organization.

Connection Roles rely on a person selecting descriptive Connection Role categories and names. Because these may be relatively short, it is recommended that you include localization in base language strings.

Processes (Workflows) that are initiated for events can work well as long as they do not need to update records with text that is to be localized. It is possible to use a workflow assembly so that logic that might apply to localized text could use the same strategy as plug-in assemblies (Use XML Web Resources as Language Resources).

On-Demand workflows require a name so that people can select them. In addition to including localization within the name of the on-demand workflow, another tactic is to create multiple workflows with localized names that each call the same child process. However, all users will see the complete list of on-demand workflows, not just the ones in their preferred user interface language.

Localization not required

SDK Message Processing Step and **Service Endpoint** solution components do not expose localizable text to users. If it is important that these components have names and descriptions that correspond to the organization's base language, you can create and export a managed solution with names and descriptions in that language.

Separate component for each language

The following solution components each may contain a considerable amount of text to be localized:

- Article Templates
- Email Templates
- Mail Merge Templates
- Reports
- Dialogs

  For these types of solution components, the recommended tactic is to create separate components for each language. This means that you typically create a base managed solution that contains your core solution components and then a separate managed solution that contains these solution components for each language. After customers install the base solution, they can install the managed solutions for the languages they have provisioned for the organization.

  Unlike **Processes (Workflows)**, you can create **Dialogs** that will reflect the user's current language preference settings and display the dialogs only to users of that language.

Create a localized dialog box

1. Install the appropriate language pack and provision the language.

   For more information, see Language Pack Installation Instructions.

2. Change your personal options to specify the **User Interface Language** for the language you want for the dialog.
3. Navigate to **Settings** and, in the **Process Center** group, select **Processes**.
4. Click **New** and create the dialog in the language that you specified.
5. After you have created the dialog, change your personal options to specify the organization base language.
6. While using the organization base language you can navigate to the **Solutions** area in **Settings** and add the localized dialog as part of a solution.

   The dialog created in the other language will only be displayed to users who view Dataverse using that language.

Use XML web resources as language resources

Plug-in assembly solution components can send messages to an end user by throwing an InvalidPluginExecutionException as well as creating and updating records. Unlike Silverlight web resources, plug-ins cannot use resource files.

When a plug-in requires localized text, you can use an XML web resource to store the localized strings so the plug-in can access them when needed. The structure of the XML is your option, but you may want to follow the structure used by ASP.NET Resource (.resx) files to create separate XML web resources for each language. For example, the following is an XML web resource named **localizedString.en_US** that follows the pattern used by . resx files.

```
<root>
 <data name="ErrorMessage">
  <value>There was an error completing this action. Please try again.</value>
 </data>
 <data name="Welcome">
  <value>Welcome</value>
 </data>
</root>
```

The following code shows how a localized message can be passed back in a plug-in to display a message to a user. It is for the pre-validation stage of a `Delete` event for the `Account` entity:

```
protected void ExecutePreValidateAccountDelete(LocalPluginContext localContext)
  {
   if (localContext == null)
   {
    throw new ArgumentNullException("localContext");
   }
   int OrgLanguage = RetrieveOrganizationBaseLanguageCode(localContext.OrganizationService);
   int UserLanguage = RetrieveUserUILanguageCode(localContext.OrganizationService,
 localContext.PluginExecutionContext.InitiatingUserId);
   String fallBackResourceFile = "";
   switch (OrgLanguage)
   {
    case 1033:
     fallBackResourceFile = "new_localizedStrings.en_US";
     break;
    case 1041:
     fallBackResourceFile = "new_localizedStrings.ja_JP";
     break;
    case 1031:
     fallBackResourceFile = "new_localizedStrings.de_DE";
     break;
    case 1036:
     fallBackResourceFile = "new_localizedStrings.fr_FR";
     break;
    case 1034:
     fallBackResourceFile = "new_localizedStrings.es_ES";
     break;
    case 1049:
     fallBackResourceFile = "new_localizedStrings.ru_RU";
     break;
    default:
     fallBackResourceFile = "new_localizedStrings.en_US";
     break;
   }
   String ResourceFile = "";
   switch (UserLanguage)
   {
    case 1033:
     ResourceFile = "new_localizedStrings.en_US";
     break;
    case 1041:
     ResourceFile = "new_localizedStrings.ja_JP";
     break;
    case 1031:
     ResourceFile = "new_localizedStrings.de_DE";
     break;
    case 1036:
     ResourceFile = "new_localizedStrings.fr_FR";
     break;
    case 1034:
     ResourceFile = "new_localizedStrings.es_ES";
     break;
    case 1049:
     ResourceFile = "new_localizedStrings.ru_RU";
```

```
     ResourceFile = new_localizedstringfile_xml;
      break;
     default:
      ResourceFile = fallBackResourceFile;
      break;
    }
    XmlDocument messages = RetrieveXmlWebResourceByName(localContext, ResourceFile);
    String message = RetrieveLocalizedStringFromWebResource(localContext, messages, "ErrorMessage");
    throw new InvalidPluginExecutionException(message);
   }
   protected static int RetrieveOrganizationBaseLanguageCode(IOrganizationService service)
   {
    QueryExpression organizationEntityQuery = new QueryExpression("organization");
    organizationEntityQuery.ColumnSet.AddColumn("languagecode");
    EntityCollection organizationEntities = service.RetrieveMultiple(organizationEntityQuery);
    return (int)organizationEntities[0].Attributes["languagecode"];
   }
   protected static int RetrieveUserUILanguageCode(IOrganizationService service, Guid userId)
   {
    QueryExpression userSettingsQuery = new QueryExpression("usersettings");
    userSettingsQuery.ColumnSet.AddColumns("uilanguageid", "systemuserid");
    userSettingsQuery.Criteria.AddCondition("systemuserid", ConditionOperator.Equal, userId);
    EntityCollection userSettings = service.RetrieveMultiple(userSettingsQuery);
    if (userSettings.Entities.Count > 0)
    {
     return (int)userSettings.Entities[0]["uilanguageid"];
    }
    return 0;
   }
   protected static XmlDocument RetrieveXmlWebResourceByName(LocalPluginContext context, string
webresourceSchemaName)
   {
    context.TracingService.Trace("Begin:RetrieveXmlWebResourceByName, webresourceSchemaName={0}",
webresourceSchemaName);
    QueryExpression webresourceQuery = new QueryExpression("webresource");
    webresourceQuery.ColumnSet.AddColumn("content");
    webresourceQuery.Criteria.AddCondition("name", ConditionOperator.Equal, webresourceSchemaName);
    EntityCollection webresources = context.OrganizationService.RetrieveMultiple(webresourceQuery);
    context.TracingService.Trace("Webresources Returned from server. Count={0}",
webresources.Entities.Count);
    if (webresources.Entities.Count > 0)
    {
     byte[] bytes = Convert.FromBase64String((string)webresources.Entities[0]["content"]);
     // The bytes would contain the ByteOrderMask. Encoding.UTF8.GetString() does not remove the BOM.
     // Stream Reader auto detects the BOM and removes it on the text
     XmlDocument document = new XmlDocument();
     document.XmlResolver = null;
     using (MemoryStream ms = new MemoryStream(bytes))
     {
      using (StreamReader sr = new StreamReader(ms))
      {
       document.Load(sr);
      }
     }
     context.TracingService.Trace("End:RetrieveXmlWebResourceByName , webresourceSchemaName={0}",
webresourceSchemaName);
     return document;
    }
    else
    {
     context.TracingService.Trace("{0} Webresource missing. Reinstall the solution", webresourceSchemaName);
     throw new InvalidPluginExecutionException(String.Format("Unable to locate the web resource {0}.",
webresourceSchemaName));
     return null;
  // This line never reached
    }
   }
   protected static string RetrieveLocalizedStringFromWebResource(LocalPluginContext context, XmlDocument
resource, string resourceId)
   {
```

```
  {
   XmlNode valueNode = resource.SelectSingleNode(string.Format(CultureInfo.InvariantCulture,
"./root/data[@name='{0}']/value", resourceId));
   if (valueNode != null)
   {
    return valueNode.InnerText;
   }
   else
   {
    context.TracingService.Trace("No Node Found for {0} ", resourceId);
    throw new InvalidPluginExecutionException(String.Format("ResourceID {0} was not found.", resourceId));
   }
  }
```

See also

Localize product property values

Configuration Migration tool

7/15/2022 • 2 minutes to read • Edit Online

The Configuration Migration tool is used to transport configuration and test data from one environment to another. It provides the means to capture such data, use that data in source control, and use that data to automate testing. Don't just rely on an environment for isolation: use source control instead.

Test data is data that's necessary to run your tests (that is, sample data). Configuration capture can't be automated.

> **TIP**
> You can also automate running the Configuration Migration tool with PowerShell by using the Microsoft.Xrm.Tooling.ConfigurationMigration module.

For more information about using the tool, including how to download it, see Move configuration data across environments and organizations with the Configuration Migration tool.

See also

Package Deployer tool
Solution Packager tool

Create packages for the Package Deployer tool

7/15/2022 • 16 minutes to read • Edit Online

Package Deployer lets administrators deploy packages on Microsoft Dataverse instances. A *package* can consist of any or all of the following:

- One or more Dataverse solution files.
- Flat files or exported configuration data file from the Configuration Migration tool. For more information about the tool, see Move configuration data across instances and organizations with the Configuration Migration tool.
- Custom code that can run before, while, or after the package is deployed to the Dataverse instance.
- HTML content specific to the package that can display at the beginning and end of the deployment process. This can be useful to provide a description of the solutions and files that are deployed in the package.

A Visual Studio 2015 or later template (available for download) is used to create packages. After creating a package, use the Package Deployer tool to deploy your package to a Dataverse instance.

Prerequisites

- Ensure that you have all the solutions and files ready that you want to include in the package.
- Microsoft .NET Framework 4.6.2
- Visual Studio 2015 or later version
- NuGet Package Manager for Visual Studio 2015
- Microsoft Dynamics CRM SDK Templates for Visual Studio that contains the package template. You can get it by downloading the Microsoft Dynamics CRM SDK Templates and double-click the `CRMSDKTemplates.vsix` file to install the template in Visual Studio.

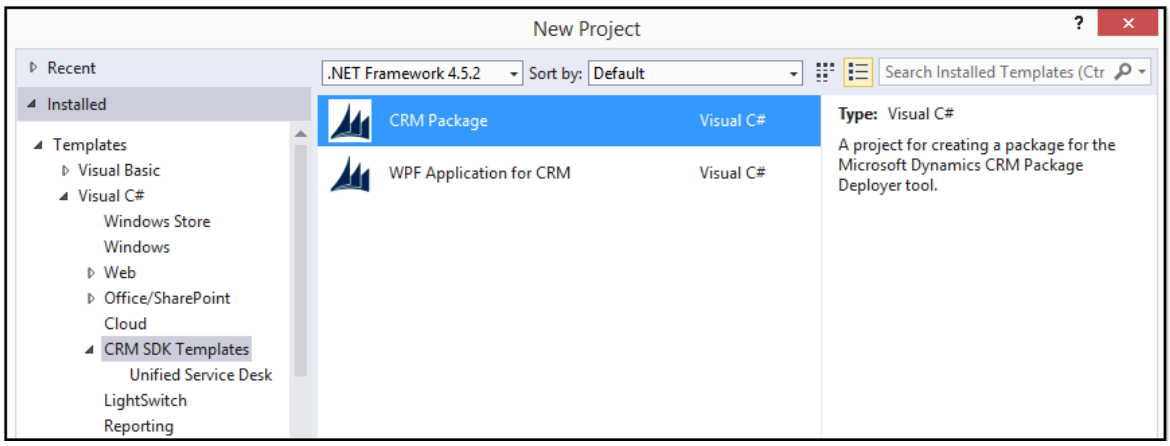
Create a package

Perform the following five steps to create a package:

- Step 1: Create a project using the template
- Step 2: Add your files to the project
- Step 3: Update the HTML files
- Step 4: Specify the configuration values for the package
- Step 5: Define custom code for your package

Step 1: Create a project using the template

1. Start Visual Studio, and create a new project.
2. In the **New Project** dialog box:
   a. From the list of installed templates, expand **Visual C#**, and select **Dynamics 365 SDK Templates**.
   b. Ensure that **.NET Framework 4.6.2** is selected.
   c. Select **Dynamics 365 Package**.
   d. Specify the name and location of the project, and click **OK**.



Step 2: Add your files to the project

1. In the **Solutions Explorer** pane, add your solutions and files under the **PkgFolder** folder.
2. For each file that you add under the **PkgFolder** folder, in the **Properties** pane, set the **Copy to Output Directory** value to **Copy Always**. This ensures that your file is available in the generated package.

Step 3: Update the HTML files: English and other languages

1. In the Solution Explorer pane, expand **PkgFolder** > **Content** > **en-us**. You'll find two folders called `EndHTML` and `WelcomeHTML`. These folders contain the HTML and associated files that enable you to display information at the end and beginning of the package deployment process. Edit the files in the HTML folder of these folders to add information for your package.

2. You can also add the HTML files in your package in other languages so that the content in the HTML appears in the language based on the locale settings of the user's computer. To do so:

   a. Create a copy of the **en-us** folder under **PkgFolder** > **Content**.

   b. Rename the copied folder to the appropriate language. For example, for the Spanish language, rename it to **es-ES**.

   c. Modify the content of the HTML files to add Spanish content.

Step 4: Specify the configuration values for the package

1. Define the package configuration by adding information about your package in the **ImportConfig.xml** file available in the **PkgFolder**. Double-click the file to open it for editing. The following list provides information about each parameter and node in the config file.

   `installsampledata`

   `True` or `false`. If `true`, installs sample data to Dataverse instance. This is the same sample data that you can install from **Settings** > **Data Management** area in Dataverse.

   `waitforsampledatatoinstall`

   True or **false**. If **true**, and if **installsampledata** is also set to **true**, waits for sample data to install before deploying the package.

   > **NOTE**
   >
   > Ensure that you set **installsampledata** to **true** if you are setting `waitforsampledatatoinstall` to **true**.

   `agentdesktopzipfile`

   File name of the zip file to unpack. If you specify a .zip file name here, it adds a screen during the package deployment process that prompts you to select a location where you want to unpack the contents of the file.

This is commonly used for creating packages for Unified Service Desk for Dynamics 365. For information about Unified Service Desk, see Administration Guide for Unified Service Desk 3.0.

`agentdesktopexename`
Name of the .exe or .msi file in the zip file or a URL to be invoked at the end of the deployment process.

This is commonly used for creating packages for Unified Service Desk.

`crmmigdataimportfile`
File name of the default configuration data file (.zip) exported using the Configuration Migration tool.

- You can also import a localized version of the configuration data file based on the locale ID (LCID) specified using new runtime settings while running the package deployer. Use the `<cmtdatafile>` node (explained later) to specify the localized versions of the configuration data file in a package and then use the `OverrideConfigurationDataFileLanguage` method (explained later) to specify the logic for importing the configuration data file based on the locale ID specified using the runtime settings. You cannot import more than one configuration data file using a package at a time.

- For Dataverse (on-premises), if your configuration data file contains user information, and both the source and target Dataverse instances are on the same Active Directory Domain, user information will be imported to the target Dataverse instance. To import user information to a Dataverse (on-premises) instance on a different domain, you must include the user map file (.xml) generated using the Configuration Migration tool in your project, and specify it along with the configuration data file using the `usermapfilename` attribute in the `<cmtdatafile>` node explained later. User information cannot be imported to Dataverse instances.
  `<solutions>` node
  Contains an array of `<configsolutionfile>` nodes that describe the solutions to import. The order of the solutions under this node indicates the order in which the solutions will be imported on the target Dataverse instance.

  `<configsolutionfile>` node
  Use this node under the `<solutions>` node to specify the individual solutions and the following information for each solution to be imported:

- `solutionpackagefilename`: Specify the .zip file name of your solution. Required.

- `overwriteunmanagedcustomizations`: Specify whether to overwrite any unmanaged customizations when importing a solution that already exists in the target Dynamics 365 instance. This is optional, and if you do not specify this attribute, by default the unmanaged customizations in the existing solution are maintained on the target Dynamics 365 instance.

- `publishworkflowsandactivateplugins`: Specify whether to publish workflows and activate plug-ins in the target Dynamics 365 instance after the solution is imported. This is optional, and if you do not specify not specify this attribute, by default the workflows are published and plug-ins are activated after the solution is imported on the target Dynamics 365 instance.

  You can add multiple solution file names in a package by adding as many `<configsolutionfile>` nodes. For example, if you want three solution files to be imported, add them like this:

```
<solutions>
<configsolutionfile solutionpackagefilename="SampleSolutionOne_1_0_managed.zip"
overwriteunmanagedcustomizations="false"
publishworkflowsandactivateplugins="true"/>
<configsolutionfile solutionpackagefilename="SampleSolutionTwo_1_0_managed.zip"
overwriteunmanagedcustomizations="false"
publishworkflowsandactivateplugins="true"/>
<configsolutionfile solutionpackagefilename="SampleSolutionThree_1_0_managed.zip" />
</solutions>
```

`<filestoimport>` node
Contains an array of `<configimportfile>` and `<zipimportdetails>` nodes that are used to describe individual files and zip files respectively to be imported.

`<configimportfile>` node
Use this node under the `<configimportfile>` node to describe a file to be imported to Dataverse. You can add multiple files in a package by adding as many `<configimportfile>` nodes.

```
<filestoimport>
<configimportfile filename="File.csv"
filetype="CSV"
associatedmap="FileMap"
importtoentity="FileEntity"
datadelimiter=""
fielddelimiter="comma"
enableduplicatedetection="true"
isfirstrowheader="true"
isrecordownerateam="false"
owneruser=""
waitforimporttocomplete="true" />
<configimportfile filename="File.zip"
filetype="ZIP"
associatedmap="FileMapName"
importtoentity="FileEntity"
datadelimiter=""
fielddelimiter="comma"
enableduplicatedetection="true"
isfirstrowheader="true"
isrecordownerateam="false"
owneruser=""
waitforimporttocomplete="true"/>

</filestoimport>
```

This has the following attributes:

ATTRIBUTE	DESCRIPTION
`filename`	Name of the file that contains the import data. If the file is a .zip file, a `<zipimportdetails>` node must be present with a `<zipimportdetail>` node for each file in the .zip file.
`filetype`	This can be csv, xml, or zip.

ATTRIBUTE	DESCRIPTION
`associatedmap`	Name of the Dataverse import data map to use with this file. If blank, attempts to use the system determined import data map name for this file.
`importtoentity`	Can be the name of the exe in the zip file, a URL, or an .msi file to provide a link to invoke at the end of the process.
`datadelimiter`	Name of the data delimiter used in the import file. Valid values are singlequote or doublequotes.
`fielddelimiter`	Name of the field delimiter used in the import file. Valid values are comma or colon, or singlequote.
`enableduplicatedetection`	Indicates whether to enable duplicate detections rules on data import. Valid values are **true** or **false**.
`isfirstrowheader`	Used to denote that the first row of the import file contains the field names. Valid values are `true` or `false`.
`isrecordownerateam`	Indicates whether the owner of the record on import should be a team. Valid values are `true` or `false`.
`owneruser`	Indicates the user ID that should own the records. The default value is the currently logged in user.
`waitforimporttocomplete`	If `true`, the system waits for the import to complete before proceeding. If `false`, it queues the jobs and moves on.

`<zipimportdetails>` node

This node contains an array of `<zipimportdetail>` nodes that describe the files included in a zip file that is used to import to Dynamics 365.

`<zipimportdetail>` node

Use this node under the `<zipimportdetails>` node to provide information about an individual file in a .zip file that is specified in the `<configimportfile>` node.

```
<filestoimport>
...
...
<zipimportdetails>
<zipimportdetail filename="subfile1.csv" filetype="csv" importtoentity="account" />
<zipimportdetail filename="subfile2.csv" filetype="csv" importtoentity="contact" />
</zipimportdetails>
</filestoimport>
```

This has the following attributes:

ATTRIBUTE	DESCRIPTION
`filename`	Name of the file that contains the import data.

ATTRIBUTE	DESCRIPTION
`filetype`	This can be csv or xml.
`importtoentity`	Can be the name of the exe in the zip file, a url, or an .msi file to provide a link to invoke at the end of the process.

`<filesmapstoimport>` node
This node contains an array of `<configmapimportfile>` nodes to import. The order of the map files in this node indicates the order in which they are imported. For information about data maps, see Create data maps for import.

`<configimportmapfile>` node
Use this node under the `<filesmapstoimport>` node to provide information about an individual map file to import in Dataverse.

```
<filesmapstoimport>
<configimportmapfile filename="FileMap.xml" />
</filesmapstoimport>
```

`<cmtdatafiles>` node
This node contains an array of `<cmtdatafile>` nodes that containslocalized version of the configuration data file to be imported.

`<cmtdatafile>` node
Use this node under the `<cmtdatafiles>` node to specify the localized configuration data files along with locale ID (required) and user information map file (optional). For example:

```
<cmtdatafiles>
<cmtdatafile filename="data_1033.zip" lcid="1033" usermapfilename="UserMap.xml" />
<cmtdatafile filename="data_1041.zip" lcid="1041" usermapfilename="" />
</cmtdatafiles>
```

You can define your custom logic in the `OverrideConfigurationDataFileLanguage` method (explained later) to import a localized configuration data file instead of the default one (specified in crmmigdataimportfile) based on the locale ID (LCID) value specified using the runtime settings (explained later).

2. Click **Save All**.

   The following represents the contents of a sample `ImportConfig.xml` file.

```
<?xml version="1.0" encoding="utf-16"?>
<configdatastorage xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="https://www.w3.org/2001/XMLSchema"
installsampledata="true"
waitforsampledatatoinstall="true"
agentdesktopzipfile=""
agentdesktopexename=""
crmmigdataimportfile="data_1033.zip">
<solutions>
<configsolutionfile solutionpackagefilename="SampleSolutionOne_1_0_managed.zip"
overwriteunmanagedcustomizations="false"
publishworkflowsandactivateplugins="true"/>
<configsolutionfile solutionpackagefilename="SampleSolutionTwo_1_0_managed.zip"
overwriteunmanagedcustomizations="false"
publishworkflowsandactivateplugins="true"/>
<configsolutionfile solutionpackagefilename="SampleSolutionThree_1_0_managed.zip" />
</solutions>
<filestoimport>
<configimportfile filename="SampleOption.csv"
filetype="CSV"
associatedmap="SampleOption"
importtoentity="sample_option"
datadelimiter=""
fielddelimiter="comma"
enableduplicatedetection="true"
isfirstrowheader="true"
isrecordownerateam="false"
owneruser=""
waitforimporttocomplete="false"/>
<configimportfile filename="File.zip"
filetype="ZIP"
associatedmap="FileMapName"
importtoentity="FileEntity"
datadelimiter=""
fielddelimiter="comma"
enableduplicatedetection="true"
isfirstrowheader="true"
isrecordownerateam="false"
owneruser=""
waitforimporttocomplete="true"/>
<zipimportdetails>
<zipimportdetail filename="subfile1.csv"
filetype="csv"
importtoentity="account" />
<zipimportdetail filename="subfile2.csv"
filetype="csv"
importtoentity="contact" />
</zipimportdetails>
</filestoimport>
<filesmapstoimport>
<configimportmapfile filename="SampleOption.xml" />
</filesmapstoimport>
<cmtdatafiles>
<cmtdatafile filename="data_1033.zip"
lcid="1033"
usermapfilename="UserMap.xml" />
<cmtdatafile filename="data_1041.zip"
lcid="1041"
usermapfilename="" />
</cmtdatafiles>
</configdatastorage>
```

Step 5: Define custom code for your package

1. In the Solution Explorer pane, double-click the **PackageTemplate.cs** file at the root to edit it.

2. In the PackageTemplate.cs file, you can:

   a. Enter custom code to execute when the package is initialized in the override method definition of `InitializeCustomExtension`.

      This method can be used to let users use the runtime parameters while running a package. As a developer, you can add support for any runtime parameter to your package by using the RuntimeSettings property as long as you have code to process it based on the user input.

      For example, the following sample code enables a runtime parameter called `SkipChecks` for the package that has two possible values: true or false. The sample code checks if the user has specified any runtime parameters while running Package Deployer (either by using the command line or PowerShell), and then accordingly processes the information. If no runtime parameter is specified by the user while running the package, the value of the RuntimeSettings property will be null.

```
public override void InitializeCustomExtension()
{
// Do nothing.

// Validate the state of the runtime settings object.
if (RuntimeSettings != null)
{
PackageLog.Log(string.Format("Runtime Settings populated.  Count = {0}",
RuntimeSettings.Count));
foreach (var setting in RuntimeSettings)
{
PackageLog.Log(string.Format("Key={0} | Value={1}", setting.Key, setting.Value.ToString()));
}

// Check to see if skip checks is present.
if ( RuntimeSettings.ContainsKey("SkipChecks") )
{
bool bSkipChecks = false;
if (bool.TryParse((string)RuntimeSettings["SkipChecks"], out bSkipChecks))
OverrideDataImportSafetyChecks = bSkipChecks;
}
}
else
PackageLog.Log("Runtime Settings not populated");
}
```

      This lets the administrator use the command line or the Import-CrmPackage cmdlet to specify whether to skip the safety checks while running the Package Deployer tool to import the package. More information: Deploy packages using Package Deployer and Windows PowerShell

   b. Enter custom code to execute before the solutions are imported in the override method definition of `PreSolutionImport` to specify whether to maintain or overwrite customizations while updating the specified solution in a target Dataverse instance, and whether to automatically activate plug-ins and workflows.

   c. Use the override method definition of `RunSolutionUpgradeMigrationStep` to perform data transformation or upgrade between two versions of a solution This method is called only if the solution you are importing is already present in the target Dataverse instance.

      This function expects the following parameters:

PARAMETER	DESCRIPTION
`solutionName`	Name of the solution
`oldVersion`	Version number of the old solution
`newVersion`	Version number of the new solution
`oldSolutionId`	GUID of the old solution.
`newSolutionId`	GUID of the new solution.

d. Enter custom code to execute before the solution import completes in the override definition of the `BeforeImportStage` method. The sample data and some flat files for solutions specified in the `ImportConfig.xml` file are imported before the solution import completes.

e. Override the currently-selected language for configuration data import using the override method definition of `OverrideConfigurationDataFileLanguage`. If the specified locale ID (LCID) of the specified language is not found in the list of available languages in the package, the default data file is imported.

   You specify the available languages for the configuration data in the `<cmtdatafiles>` node in the `ImportConfig.xml` file. The default configuration data import file is specified in the `crmmigdataimportfile` attribute in the `ImportConfig.xml` file.

   Skipping data checks (OverrideDataImportSafetyChecks = true) can be effective here if you are sure that the target Dataverse instance does not contain any data.

f. Enter custom code to execute after the import completes in the override definition of `AfterPrimaryImport` >method. The remaining flat files that were not imported earlier, before the solution import started, are imported now.

g. Change the default name of your package folder from PkgFolder to the package name that you want. To do so, rename the `PkgFolder` >folder in the **Solution Explorer** pane, and then edit the return value under the `GetImportPackageDataFolderName` property.

```
public override string GetImportPackageDataFolderName
{
get
{
// WARNING this value directly correlates to the folder name in the Solution Explorer where
the ImportConfig.xml and sub content is located.
// Changing this name requires that you also change the correlating name in the Solution
Explorer
return "PkgFolder";
}
}
```

h. Change the package name by editing the return value under the `GetNameOfImport` property.

```
public override string GetNameOfImport(bool plural)
{
return "Package Short Name";
}
```

   This is the name of your package that will appear on the package selection page in the Dynamics

365 Package Deployer wizard.

i. Change the package description by editing the return value under the `GetImportPackageDescriptionText` property.

```
public override string GetImportPackageDescriptionText
{
get { return "Package Description"; }
}
```

This is the package description that will appear alongside the package name on the on the package selection page in the Package Deployer wizard.

j. Change the package long name by editing the return value under the `GetLongNameOfImport` property.

```
public override string GetLongNameOfImport
{
get { return "Package Long Name"; }
}
```

The package long name appears on the next page after you have selected the package to install.

3. Additionally, the following function and variables are available to the package:

NAME	TYPE	DESCRIPTION
CreateProgressItem(String)	Function	Used to create a new progress item in the user interface (UI).
RaiseUpdateEvent(String, ProgressPanelItemStatus)	Function	Used to update the progress created by the call to CreateProgressItem(String). ProgressPanelItemStatus is an enum with the following values: Working = 0 Complete = 1 Failed = 2 Warning = 3 Unknown = 4
RaiseFailEvent(String, Exception)	Function	Used to fail the current status import with an exception message.
IsRoleAssoicatedWithTeam(Guid, Guid)	Function	Used to determine if a role is associated with a specified team.
IsWorkflowActive(Guid)	Function	Used to determine if a specified workflow is active.

NAME	TYPE	DESCRIPTION
PackageLog	Class Pointer	This is a pointer to the initialized logging interface for the package. This interface is used by a package to log messages and exceptions to the package log file.
RootControlDispatcher	Property	This is a dispatcher interface used to allow your control to render its own UI during package deployment. Use this interface to wrap any UI elements or commands. It is important to check this variable for null values before using it as it may or may not be set to a value.
CrmSvc	Property	This is a pointer to CrmServiceClient class that allows for a package to address Dynamics 365 from within the package. Use this to execute SDK methods and other actions in the overridden methods.
DataImportBypass	Property	Use this to specify whether Dynamics 365 Package Deployer skips all data import operations such as importing Dataverse sample data, flat file data, and data exported from the Configuration Migration tool. Specify true or false. Default is `false`.
OverrideDataImportSafetyChecks	Property	Use this to specify whether Dynamics 365 Package Deployer will bypass some of its safety checks, which helps in improving the import performance. Specify `true` or `false`. Default is `false`. You should set this to `true` only if the target Dataverse instance does not contain any data.

4. Save your project, and then build it (**Build** > **Build Solution**) to create the package. Your package is the following files under the *<Project>*\Bin\Debug folder

   - **<PackageName> folder**: The folder name is the same as the one you changed for your package folder name in step 2.g of this section (Step 5: Define custom code for your package). This folder contains all solutions, configuration data, flat files, and the contents for your package.
   - **<PackageName>.dll**: The assembly contains the custom code for your package. By default, the name of the assembly is the same as your Visual Studio project name.

     The next step is to deploy your package.

Deploy a package

After you create a package, you can deploy it on the Dataverse instance by using either the Package Deployer

tool or Windows PowerShell.

The package deployer tool is distributed as part of the Microsoft.CrmSdk.XrmTooling.PackageDeployment NuGet package. To download the Package Deployer tool, see Download tools from NuGet.

For detailed information, see Deploy packages using Package Deployer or Windows PowerShell.

Best practices for creating and deploying packages

While creating packages, developers must ensure that the package assemblies are signed.

While deploying the packages, Dataverse administrators must:

- Insist on a signed package assembly so that you can track an assembly back to its source.
- Test the package on a pre-production instance (preferably a mirror image of the production instance) before running it on a production instance.
- Back up the production instance before deploying the package.

See also

Solution Packager tool

SolutionPackager tool

7/15/2022 • 9 minutes to read • Edit Online

SolutionPackager is a tool that can reversibly decompose a Microsoft Dataverse compressed solution file into multiple XML files and other files so that these files can be easily managed by a source control system. The following sections show you how to run the tool and how to use the tool with managed and unmanaged solutions.

Where to find the SolutionPackager tool

The SolutionPackager tool is distributed as part of the Microsoft.CrmSdk.CoreTools NuGet package. See Download tools from NuGet for information about how to download it.

SolutionPackager command-line arguments

SolutionPackager is a command-line tool that can be invoked with the parameters identified in the following table.

ARGUMENT	DESCRIPTION
/action: {Extract|Pack}	Required. The action to perform. The action can be either to extract a solution .zip file to a folder, or to pack a folder into a .zip file.
/zipfile: <file path>	Required. The path and name of a solution .zip file. When extracting, the file must exist and will be read from. When packing, the file is replaced.
/folder: <folder path>	Required. The path to a folder. When extracting, this folder is created and populated with component files. When packing, this folder must already exist and contain previously extracted component files.
/packagetype: {Unmanaged|Managed|Both}	Optional. The type of package to process. The default value is Unmanaged. This argument may be omitted in most occasions because the package type can be read from inside the .zip file or component files. When extracting and Both is specified, managed and unmanaged solution .zip files must be present and are processed into a single folder. When packing and Both is specified, managed and unmanaged solution .zip files will be produced from one folder. For more information, see the section on working with managed and unmanaged solutions later in this topic.
/allowWrite:{Yes|No}	Optional. The default value is Yes. This argument is used only during an extraction. When /allowWrite:No is specified, the tool performs all operations but is prevented from writing or deleting any files. The extract operation can be safely assessed without overwriting or deleting any existing files.

ARGUMENT	DESCRIPTION
/allowDelete:{Yes|No|Prompt}	Optional. The default value is Prompt. This argument is used only during an extraction. When /allowDelete:Yes is specified, any files present in the folder specified by the /folder parameter that are not expected are automatically deleted. When /allowDelete:No is specified, no deletes will occur. When /allowDelete:Prompt is specified, the user is prompted through the console to allow or deny all delete operations. Note that if /allowWrite:No is specified, no deletes will occur even if /allowDelete:Yes is also specified.
/clobber	Optional. This argument is used only during an extraction. When /clobber is specified, files that have the read-only attribute set are overwritten or deleted. When not specified, files with the read-only attribute aren't overwritten or deleted.
/errorlevel: {Off|Error|Warning|Info|Verbose}	Optional. The default value is Info. This argument indicates the level of logging information to output.
/map: <file path>	Optional. The path and name of an .xml file containing file mapping directives. When used during an extraction, files typically read from inside the folder specified by the /folder parameter are read from alternate locations as specified in the mapping file. During a pack operation, files that match the directives aren't written.
/nologo	Optional. Suppress the banner at runtime.
/log: <file path>	Optional. A path and name to a log file. If the file already exists, new logging information is appended to the file.
@ <file path>	Optional. A path and name to a file that contains command-line arguments for the tool.
/sourceLoc: <string>	Optional. This argument generates a template resource file, and is valid only on extract. Possible values are `auto` or an LCID/ISO code for the language you want to export. When this argument is used, the string resources from the given locale are extracted as a neutral .resx file. If `auto` or just the long or short form of the switch is specified, the base locale or the solution is used. You can use the short form of the command: /src.
/localize	Optional. Extract or merge all string resources into .resx files. You can use the short form of the command: /loc. The localize option supports shared components for .resx files. More information: Using RESX web resources

Use the /map command argument

The following discussion details the use of the /map argument to the SolutionPackager tool.

Files that are built in an automated build system, such as .xap Silverlight files and plug-in assemblies, are typically not checked into source control. Web resources may already be present in source control in locations that are not directly compatible with the SolutionPackager tool. By including the /map parameter, the

SolutionPackager tool can be directed to read and package such files from alternate locations and not from inside the Extract folder as it would typically be done. The /map parameter must specify the name and path to an XML file containing mapping directives that instruct the SolutionPackager to match files by their name and path, and indicate the alternate location to find the matched file. The following information applies to all directives equally.

- Multiple directives may be listed including those that will match identical files. Directives listed early in the file take precedence over those listed later.
- If a file is matched to any directive, it must be found in at least one alternative location. If no matching alternatives are found, the SolutionPackager will issue an error.
- Folder and file paths may be absolute or relative. Relative paths are always evaluated from the folder specified by the /folder parameter.
- Environment variables may be specified by using a %variable% syntax.
- A folder wildcard "**" may be used to mean "in any sub-folder". It can only be used as the final part of a path, for example: "c:\folderA**".
- File name wildcards may be used only in the forms "*.ext" or "*.*". No other pattern is supported.

  The three types of directives mappings are described here, along with an example that shows you how to use them.

Folder mapping

The following provides detailed information on folder mapping.

Xml Format

```
<Folder map="folderA" to="folderB" />
```

Description

File paths that match "folderA" will be switched to "folderB".

- The hierarchy of subfolders under each must exactly match.
- Folder wildcards are not supported.
- No file names may be specified.

  ##### Examples

  ```
  <Folder map="folderA" to="folderB" />
  <Folder map="folderA\folderB" to="..\..\folderC\" />
  <Folder map="WebResources\subFolder" to="%base%\WebResources" />
  ```

File To file mapping

The following provides detailed information on file-to-file mapping.

Xml Format

```
<FileToFile map="path\filename.ext" to="path\filename.ext" />
```

Description

Any file matching the `map` parameter will be read from the name and path specified in the `to` parameter.

For the `map` parameter:

- A file name must be specified. The path is optional. If no path is specified, files from any folder may be matched.
- File name wildcards are not supported.
- The folder wildcard is supported.

For the `to` parameter:

- A file name and path must be specified.
- The file name may differ from the name in the `map` parameter.
- File name wildcards are not supported.
- The folder wildcard is supported.

Examples

```
  <FileToFile map="assembly.dll" to="c:\path\folder\assembly.dll" />
  <FileToFile map="PluginAssemblies\**\this.dll" to="..\..\Plugins\**\that.dll" />
  <FileToFile map="Webresrouces\ardvark.jpg" to="%SRCBASE%\CrmPackage\WebResources\JPG format\aardvark.jpg"
/>
```

File to path mapping

The following provides detailed information on file-to-path mapping.

Xml Format

```
<FileToPath map="path\filename.ext" to="path" />
```

Description

Any file matching the `map` parameter is read from the path specified in the `to` parameter.

For the `map` parameter:

- A file name must be specified. The path is optional. If no path is specified, files from any folder may be matched.
- File name wildcards are supported.
- The folder wildcard is supported.

For the `to` parameter:

- A path must be specified.
- The folder wildcard is supported.
- A file name must not be specified.

Examples

```
  <FileToPath map="assembly.dll" to="c:\path\folder" />
  <FileToPath map="PluginAssemblies\**\this.dll" to="..\..\Plugins\bin\**" />
  <FileToPath map="*.jpg" to="%SRCBASE%\CrmPackage\WebResources\JPG format\" />
  <FileToPath map="*.*" to="..\..\%ARCH%\%TYPE%\drop" />
```

Example mapping

The following XML code sample shows a complete mapping file that enables the SolutionPackager tool to read any web resource and the two default generated assemblies from a Developer Toolkit project named CRMDevTookitSample.

```
<?xml version="1.0" encoding="utf-8"?>
<Mapping>
       <!-- Match specific named files to an alternate folder -->
       <FileToFile map="CRMDevTookitSamplePlugins.dll"
to="..\..\Plugins\bin\**\CRMDevTookitSample.plugins.dll" />
       <FileToFile map="CRMDevTookitSampleWorkflow.dll"
to="..\..\Workflow\bin\**\CRMDevTookitSample.Workflow.dll" />
       <!-- Match any file in and under WebResources to an alternate set of sub-folders -->
       <FileToPath map="WebResources\*.*" to="..\..\CrmPackage\WebResources\**" />
       <FileToPath map="WebResources\**\*.*" to="..\..\CrmPackage\WebResources\**" />
</Mapping>
```

Managed and unmanaged solutions

A Dataverse compressed solution (.zip) file can be exported in one of two types as shown here.

Managed solution

A completed solution ready to be imported into an organization. Once imported, components can't be added or removed, although they can optionally allow further customization. This is recommended when development of the solution is complete.

Unmanaged solution

An open solution with no restrictions on what can be added, removed, or modified. This is recommended during development of a solution.

The format of a compressed solution file will be different based on its type, either managed or unmanaged. The SolutionPackager can process compressed solution files of either type. However, the tool can't convert one type to another. The only way to convert solution files to a different type, for example from unmanaged to managed, is by importing the unmanaged solution .zip file into a Dataverse server and then exporting the solution as a managed solution.

The SolutionPackager can process unmanaged and managed solution .zip files as a combined set via the /PackageType:Both parameter. To perform this operation, it is necessary to export your solution twice as each type, naming the .zip files as follows.

Unmanaged .zip file: AnyName.zip	Managed .zip file: AnyName_managed.zip

The tool will assume the presence of the managed zip file in the same folder as the unmanaged file and extract both files into a single folder preserving the differences where managed and unmanaged components exist.

After a solution has been extracted as both unmanaged and managed, it is possible from that single folder to pack both, or each type individually, using the /PackageType parameter to specify which type to create. When specifying both, two .zip files will be produced using the naming convention as above. If the /PackageType parameter is missing when packing from a dual managed and unmanaged folder, the default is to produce a single unmanaged .zip file.

Troubleshooting

If you use Visual Studio to edit resource files created by the solution packager, you may receive a message when

you repack similar to this:

```
“Failed to determine version id of the resource file <filename>.resx the resource file must be exported from the solutionpackager.exe tool in order to be used as part of the pack process.”
```

This happens because Visual Studio replaces the resource file's metadata tags with data tags.

Workaround

1. Open the resource file in your favorite text editor and locate and update the following tags:

   ```
   <data name="Source LCID" xml:space="preserve">
   <data name="Source file" xml:space="preserve">
   <data name="Source package type" xml:space="preserve">
   <data name="SolutionPackager Version" mimetype="application/x-microsoft.net.object.binary.base64">
   ```

2. Change the node name from `<data>` to `<metadata>`.

   For example, this string:

   ```
   <data name="Source LCID" xml:space="preserve">
     <value>1033</value>
   </data>
   ```

   Changes to:

   ```
   <metadata name="Source LCID" xml:space="preserve">
     <value>1033</value>
   </metadata>
   ```

   This allows the solution packager to read and import the resource file. This problem has only been observed when using the Visual Studio Resource editor.

See also

Use Source Control with Solution Files

Solution concepts

Source control with solution files

7/15/2022 • 6 minutes to read • Edit Online

The SolutionPackager tool can be used with any source control system. After a solution .zip file has been extracted to a folder, simply add and submit the files to your source control system. These files can then be synchronized on another computer where they can be packed into a new identical solution .zip file.

An important aspect when using extracted component files in source control is that adding all the files into source control may cause unnecessary duplication. See the Solution Component File Reference to see which files are generated for each component type and which files are recommended for use in source control.

As further customizations and changes are necessary for the solution, developers should edit or customize components through existing means, export again to create a .zip file, and extract the compressed solution file into the same folder.

> **IMPORTANT**
> Except for the sections described in When to edit the customizations file, manual editing of extracted component files and .zip files is not supported.

When the SolutionPackager tool extracts the component files it will not overwrite existing component files of the same name if the file contents are identical. In addition, the tool honors the read-only attribute on component files producing a warning in the console window that particular files were not written. This enables the user to check out, from source control, the minimal set of files that are changing. The `/clobber` parameter can be used to override and cause read-only files to be written or deleted. The `/allowWrite` parameter can be used to assess what impact an extract operation has without actually causing any files to be written or deleted. Use of the `/allowWrite` parameter with verbose logging is effective.

After the extract operation is completed with the minimal set of files checked out from source control, a developer may submit the changed files back into source control, as is done with any other type of source file.

Team development

When there are multiple developers working on the same solution component a conflict might arise where changes from two developers result in changes to a single file. This occurrence is minimized by decomposing each individually editable component or subcomponent into a distinct file. Consider the following example.

1. Developer A and B are both working on the same solution.
2. On independent computers, they both get the latest sources of the solution from source control, pack, and import an unmanaged solution .zip file into independent Microsoft Dataverse organizations.
3. Developer A customizes the "Active Contacts" system view and the main form for the Contact entity.
4. Developer B customizes the main form for the Account entity and changes the "Contact Lookup View".
5. Both developers export an unmanaged solution .zip file and extract.
    a. Developer A will need to check out one file for the Contact main form, and one file for the "Active Contacts" view.
    b. Developer B will need to check out one file for the Account main form, and one file for the "Contact

Lookup View".

6. Both developers may submit, in any order, as their respective changes touched separate files.
7. After both submissions are complete, they can repeat step #2 and then continue to make further changes in their independent organizations. They each have both sets of changes, with no overwrites of their own work.

The previous example works only when there are changes to separate files. It is inevitable that independent customizations require changes within a single file. Based on the example shown above, consider that developer B customized the "Active Contacts" view while developer A was also customizing it. In this new example, the order of events becomes important. The correct process to reconcile this predicament, written out in full, is as follows.

1. Developer A and B are both working on the same solution.
2. On independent computers, they both get the latest sources of the solution from source control, pack, and import an unmanaged solution .zip file into independent organizations.
3. Developer A customizes the "Active Contacts" system view and the main form for the Contact entity.
4. Developer B customizes the main form for the Account entity and changes the "Active Contacts".
5. Both developers export an unmanaged solution . zip file and extract.
    a. Developer A will need to check out one file for the Contact main form, and one file for the "Active Contacts" view.
    b. Developer B will need to check out one file for the Account main form, and one file for the "Active Contacts" view.
6. Developer A is ready first.
    a. Before she submits to source control she must get latest sources to ensure no prior check-ins conflict with her changes.
    b. There are no conflicts so she is able to submit.
7. Developer B is ready next following developer A.
    a. Before he submits he must get the latest sources to ensure no prior check-ins conflict with his changes.
    b. There is a conflict because the file for "Active Contacts" has been modified since he last retrieved the latest sources.
    c. Developer B must reconcile the conflict. It is possible the capabilities of the source control system in use may aide this process; otherwise the following choices are all viable.
        a. Developer B, through source control history, if available, can see that the developer A made the prior change. Through direct communication they can discuss each change. Then developer B only has to update his organization with the agreed resolution. He then exports, extracts, and overwrites the conflicting file and submits.
        b. Allow source control to overwrite his local file. Developer B packs the solution and imports it into his organization, then assesses the state of the view and re-customizes it as necessary. Next, he may export, extract, and overwrite the conflicting file.
        c. If the prior change can be deemed unnecessary, developer B allows his copy of the file to overwrite the version in source control and submits.

Whether working on a shared organization or independent organizations, team development of Dataverse solutions requires those actively working on a common solution to be aware of the work of others. The SolutionPackager tool does not fully remove this need but it does enable easy merging of non-conflicting changes at the source control level, and it proactively highlights the concise components where conflicts have arisen.

The next sections are the generic processes to effectively use the SolutionPackager tool in source control when developing with teams. These work equally with independent organizations or shared development organizations, though with shared organizations the export and extract will naturally include all changes present within the solution, not just those made by the developer performing the export. Similarly, when importing a solution .zip file the natural behavior to overwrite all components will occur.

Create a solution

The following procedure identifies the typical steps used when first creating a solution.

1. In a clean organization, create a solution on Dataverse server, and then add or create components as necessary.
2. When you are ready to check in, do the following.
    a. Export the unmanaged solution.
    b. Using the SolutionPackager tool, extract the solution into component files.
    c. From those extracted component files, add the necessary files to source control.
    d. Submit these changes to source control.

Modify a solution

The following procedure identifies the typical steps used when modifying an existing solution.

1. Synchronize or get the latest solution component file sources.
2. Using the SolutionPackager tool, pack component files into an unmanaged solution .zip file.
3. Import the unmanaged solution file into an organization.
4. Customize and edit the solution as necessary.
5. When you are ready to check the changes into source control, do the following.
    a. Export the unmanaged solution.
    b. Using the SolutionPackager tool, extract the exported solution into component files.
    c. Synchronize or get the latest sources from source control.
    d. Reconcile if any conflicts exist.
    e. Submit the changes to source control.

    Steps 2 and 3 must be done before further customizations occur in the development organization. Within step 5, step b must be completed before step c.

See also

Solution Component File Reference (SolutionPackager)
SolutionPackager tool

Use the Power Apps checker web API

7/15/2022 • 8 minutes to read • Edit Online

The Power Apps checker web API provides a mechanism to run static analysis checks against customizations and extensions to the Microsoft Dataverse platform. It is available for makers and developers to perform rich static analysis checks on their solutions against a set of best practice rules to quickly identify problematic patterns. The service provides the logic for the solution checker feature in the Power Apps maker portal and is included as part of the automation for applications submitted to AppSource. Interacting with the service directly in this manner allows for analysis of solutions that are included as part of on-premise (all supported versions) and online environments.

For information about using the checker service from PowerShell code see Work with solutions using PowerShell.

NOTE

- Use of Power Apps checker does not guarantee that a solution import will be successful. The static analysis checks performed against the solution do not know the configured state of the destination environment and import success may be dependent on other solutions or configurations in the environment.

Alternative approaches

Before reading through the details of how to interact at the lowest level with the web APIs, consider leveraging our PowerShell module, Microsoft.PowerApps.Checker.PowerShell, instead. It is a fully supported tool that is available in the PowerShell Gallery. The current restriction is that it does require Windows PowerShell. If unable to meet this requirement, then interacting with the APIs directly will likely be the best approach.

Getting started

It is important to note that a solution analysis can result in a long running process. It can typically take sixty seconds to upwards of five minutes depending on a variety of factors, such as number, size, and complexity of customizations and code. The analysis flow is multi-step and asynchronous beginning with initiating an analysis job with the status API being used to query for job completion. An example flow for an analysis is as follows:

1. Obtain an OAuth token
2. Call upload (for each file in parallel)
3. Call analyze (initiates the analysis job)
4. Call status until finished (looping with a pause in between calls until the end is signaled or thresholds are met)
5. Download the result(s) from the provided SAS URI

A few variations are:

- Include a lookup of the ruleset or rules as a pre-step. However, it would be slightly faster to pass in a configured or hard coded ruleset ID. It is recommended that you use a ruleset that meets your needs.
- You can opt to not use the upload mechanism (see the upload for limitations).

You will need to determine the following:

- Which geography?
- Which version?
- Which rulesets and rules?
- What is your tenant ID?

Refer to the following topics for documentation on the individual APIs:

Retrieve the list of rulesets
Retrieve the list of rules
Upload a file
Invoke analysis
Check for analysis status

Determine a geography

When interacting with the Power Apps checker service, files are temporarily stored in Azure along with the reports that are generated. By using a geography specific API, you can control where the data is stored. Requests to a geography endpoint are routed to a regional instance based on best performance (latency to the requestor). Once a request enters a regional service instance, all processing and persisted data remains within that particular region. Certain API responses will return regional instance URLs for subsequent requests once an analysis job has been routed to a specific region. Be aware that each geography may have a different version of the service deployed at any given point in time due to the multi-stage safe deployment process, which ensures full version compatibility. Thus, the same geography should be used for each API call in the analysis lifecycle and may reduce overall execution time as the data may not have to travel as far over the wire. The following are the available geographies:

AZURE DATACENTER	NAME	GEOGRAPHY	BASE URI
Public	Preview	United States	unitedstatesfirstrelease.api.advisor.powerapps.com
Public	Production	United States	unitedstates.api.advisor.powerapps.com
Public	Production	Europe	europe.api.advisor.powerapps.com
Public	Production	Asia	asia.api.advisor.powerapps.com
Public	Production	Australia	australia.api.advisor.powerapps.com
Public	Production	Japan	japan.api.advisor.powerapps.com
Public	Production	India	india.api.advisor.powerapps.com
Public	Production	Canada	canada.api.advisor.powerapps.com
Public	Production	South America	southamerica.api.advisor.powerapps.com

AZURE DATACENTER	NAME	GEOGRAPHY	BASE URI
Public	Production	United Kingdom	unitedkingdom.api.advisor.powerapps.com
Public	Production	France	france.api.advisor.powerapps.com
Public	Production	Germany	germany.api.advisor.powerapps.com
Public	Production	United Arab Emirates	unitedarabemirates.api.advisor.powerapps.com
Public	Production	US Government	gov.api.advisor.powerapps.us
Public	Production	US Government L4	high.api.advisor.powerapps.us
Public	Production	US Government L5 (DOD)	mil.api.advisor.appsplatform.us
Public	Production	China operated by 21Vianet	china.api.advisor.powerapps.cn

> **NOTE**
> You may choose to use the preview geography to incorporate the latest features and changes earlier. However, note that the preview uses United States Azure regions only.

Versioning

While not required, it is recommended to include the api-version query string parameter with the desired API version. The current API version is 1.0. For example, below is a ruleset HTTP request specifying to use the 1.0 API version:

```
https://unitedstatesfirstrelease.api.advisor.powerapps.com/api/ruleset?api-version=1.0
```

If not provided, the latest API version will be used by default. Using an explicit version number is recommended as the version will be incremented if breaking changes are introduced. If the version number is specified in a request, backward compatibility support in later (numerically greater) versions will be maintained.

Rulesets and rules

Power Apps checker requires a list of rules when run. These rules can be provided in the form of individual rules or a grouping of rules, referred to as a *ruleset*. A ruleset is a convenient way to specify a group of rules instead of having to specify each rule individually. For example, the solution checker feature uses a ruleset named *Solution Checker*. As new rules are added or removed, the service will include these changes automatically without requiring any change by the consuming application. If you require that the list of rules not change automatically as described above, then the rules can be specified individually. Rulesets can have one or more rules with no limit. A rule can be in no or multiple rulesets. You can get a list of all rulesets by calling the API as follows: `[Geographical URL]/api/ruleset`. This endpoint is open and does not require authentication.

Solution checker ruleset

The solution checker ruleset contains a set of impactful rules that have limited chances for false positives. If running analysis against an existing solution, it is recommended that you start with this ruleset. This is the ruleset used by the solution checker feature.

AppSource certification ruleset

When publishing applications on AppSource, you must get your application certified. Applications published on AppSource are required to meet a high quality standard. The AppSource certification ruleset contains the rules that are part of the solution checker ruleset, plus additional rules to ensure only high quality applications are published on the store. Some of AppSource certification rules are more prone to false positives and may require additional attention to resolve.

Find your tenant ID

The ID of your tenant is needed to interact with the APIs that require a token. Refer to this article for details on how to obtain the tenant ID. You can also use PowerShell commands to retrieve the tenant ID. The following example leverages the cmdlets in the AzureAD module.

```
# Login to AAD as your user
Connect-AzureAD

# Establish your tenant ID
$tenantId = (Get-AzureADTenantDetail).ObjectId
```

The tenant ID is the value of the `ObjectId` property that is returned from `Get-AzureADTenantDetail`. You may also see it after logging in using the Connect-AzureAD cmdlet in the cmdlet output. In this case it will be named `TenantId`.

Authentication and authorization

Querying for rules and rulesets do not require an OAuth token, but all of the other APIs do require the token. The APIs do support authorization discovery by calling any of the APIs that require a token. The response will be an unauthorized HTTP status code of 401 with a WWW-Authenticate header, the authorization URI, and the resource ID. You should also provide your tenant ID in the `x-ms-tenant-id` header. Refer to Power Apps Checker authentication and authorization for more information. Below is an example of the response header returned from an API request:

```
WWW-Authenticate →Bearer authorization_uri="https://login.microsoftonline.com/0082fff7-33c5-44c9-920c-
c2009943fd1e", resource_id="https://api.advisor.powerapps.com/"
```

Once you have this information, you can choose to use the Azure Active Directory Authentication Library (ADAL) or some other mechanism to acquire the token. Below is an example of how this can be done using C# and the ADAL library, version 4.5.1:

```
// Call the status URI as it is the most appropriate to use with a GET.
// The GUID here is just random, but needs to be there.
Uri queryUri = new Uri($"{targetServiceUrl}/api/status/4799049A-E623-4B2A-818A-3A674E106DE5");
AuthenticationParameters authParams = null;

using (var client = new HttpClient())
{
    var request = new HttpRequestMessage(HttpMethod.Get, queryUri);
    request.Headers.Add("x-ms-tenant-id", tenantId.ToString());

    // NOTE - It is highly recommended to use async/await
    using (var response = client.SendAsync(request).GetAwaiter().GetResult())
    {
        if (response.StatusCode == System.Net.HttpStatusCode.Unauthorized)
        {
            // NOTE - It is highly recommended to use async/await
            authParams =
AuthenticationParameters.CreateFromUnauthorizedResponseAsync(response).GetAwaiter().GetResult();
        }
        else
        {
            throw new Exception($"Unable to connect to the service for authorization information.
{response.ReasonPhrase}");
        }
    }
}
```

Once you have acquired the token, it is advised that you provide the same token to subsequent calls in the request lifecycle. However, additional requests will likely warrant a new token be acquired for security reasons.

Transport security

For best-in-class encryption, the checker service only supports communications using Transport Layer Security (TLS) 1.2 and above. For guidance on .NET best practices around TLS, refer to Transport Layer Security (TLS) best practices with the .NET Framework.

Report format

The result of the solution analysis is a zip file containing one or more reports in a standardized JSON format. The report format is based on static analysis results referred to as Static Analysis Results Interchange Format (SARIF). There are tools available to view and interact with SARIF documents. Refer to this web site for details. The service leverages version two of the OASIS standard.

See also

Retrieve the list of rulesets
Retrieve the list of rules
Upload a file
Invoke analysis
Check for analysis status

Invoke analysis

7/15/2022 • 2 minutes to read • Edit Online

Initiating an analysis job is done by submitting a `POST` request to the `analyze` route. Analysis can be a long running process that usually lasts longer than a minute. The API first does some basic validation, initiates the request on the backend by submitting a job, and then responds with a status code of 202 and a `Location` header or with the appropriate error details. The `Location` header value is a URL that can be used to check on the status of the request and to obtain the URL(s) of the result(s). There are various options through the `POST` action to tailor the job based on your criteria, such as the list of rules or rulesets, files to exclude from the analysis, and more. You can initiate the analysis using the following `[Geographical URL]/api/analyze?api-version=1.0`.

> **NOTE**
>
> It is recommended to wait between 15 to 60 seconds between status checks. Analysis usually takes between 1 to 5 minutes to run.
> This API does require an OAuth token.

Headers

NAME	TYPE	EXPECTED VALUE	REQUIRED?
Authorization	string	The OAuth 1 bearer token with Azure Active Directory (AAD) Application ID claim.	yes
x-ms-tenant-id	GUID	The ID of the tenant for the application.	yes
x-ms-correlation-id	GUID	The Identifier for the analysis run. You should provide the same ID for the entire execution (upload, analyze, status).	yes
Accept	object	`application/json, application/x-ms-sarif-v2`	yes
Accept-Language	string	The language code or codes (e.g,. en-US). The default is en-US. If multiple languages are provided, the first will be the primary. However, all translations (if the language is supported) will be included.	no

Body

Commonly used options:

PROPERTY	TYPE	EXPECTED VALUE	REQUIRED?
sasUriList	array of strings	A list of URIs that provides the service access to download a single solution, a zip file containing multiple solution files, or a package.	Yes
ruleSets	array of custom	0 or more	No
ruleSets.id	guid	The ID of the ruleset, which can be found by querying the ruleset API.	No, but this is usually what you would want to use. You must use either this or ruleCodes.
ruleCodes.code	string	The ID of the desired rule, which can be found by querying the ruleset and rule APIs.	No, you must use either this or ruleSets.
fileExclusions	array of strings	A list of file names or file name patterns to exclude. Support exists for using "*" as a wildcard in the beginning and/or end of a file name (e.g.,*jquery.dll and *jquery*).	No

Expected responses

HTTP STATUS CODE	SCENARIO	RESULT
202	Request for analysis was accepted and the status check URI was returned in the `Location` header	No result body
400	A non-zip file was sent, incorrect parameters, or a file was included with a virus	No result body
409	A request with a duplicate `x-ms-correlation-id` header value was sent	No result body

Expected response headers

NAME	TYPE	EXPECTED VALUE	REQUIRED?
Location	Uri	URL to use in querying for the current status and to obtain the results	yes

Example: initiate an analysis

This is an example of initiating an analysis job with the *AppSource Certification* ruleset, a single file, and excluding files that contain the text *jquery* and *json* in the name.

Request

```
POST [Geographical URI]/api/analyze?api-version=1.0
Accept: application/json
Content-Type: application/json; charset=utf-8
x-ms-correlation-id: 9E378E56-6F35-41E9-BF8B-C0CC88E2B832
x-ms-tenant-id: F2E60E49-CB87-4C24-8D4F-908813B22506

{
    "ruleSets": [{
        "id": "0ad12346-e108-40b8-a956-9a8f95ea18c9"
    }],
    "sasUriList": ["https://testenvfakelocation.blob.core.windows.net/mySolution.zip"],
    "fileExclusions": ["*jquery*", "*json*"]
}
```

Response

```
HTTP/1.1 202 Accepted
Content-Type: application/json; charset=utf-8
Location: [Geographical URI]/api/status/9E378E56-6F35-41E9-BF8B-C0CC88E2B832&api-version=1.0
```

See also

Use the Power Apps checker web API
Retrieve the list of rulesets
Retrieve the list of rules
Upload a file
Check for analysis status

Check for analysis status

7/15/2022 • 3 minutes to read • Edit Online

A URL is returned as part of the `Location` header in response to a request to the `analyze` API. It is to be used to query via HTTP `GET` for the analysis job's status. When the analysis job is finished the response body will include the URL or list of URLs in which the results output can be downloaded. Keep calling this URI until an HTTP status code of 200 is returned. While the job is still running, an HTTP status code of 202 will be returned with the `Location` header containing this same URI that was returned from `analyze`. Once a 200 response is returned, the `resultFileUris` property will include the single or list of downloadable locations of the output, which is contained in a zip file. A Static Analysis Results Interchange Format (SARIF) V2 formatted file is included within this zip download that is a `JSON` formatted file containing the results of the analysis. The response body will contain an `IssueSummary` object that contains a summary of the count of issues found.

> **NOTE**
> It is recommended to wait between 15 to 60 seconds between status checks. Analysis usually takes between 1 to 5 minutes to run.
> This API does require an OAuth token that must be a token for the same client application that initiated the analysis job.

Headers

NAME	TYPE	EXPECTED VALUE	REQUIRED?
Authorization	string	The OAuth 1 bearer token with AAD Application ID claim.	yes
x-ms-tenant-id	GUID	The ID of the tenant for the application.	yes
x-ms-correlation-id	GUID	The identifier for the analysis run. You should provide the same Id for the entire execution (upload, analyze, status)	yes

Expected responses

HTTP STATUS CODE	SCENARIO	RESULT
200	One or more results were found	See the example below. One result will be returned.
202	Still processing	See the example below. One result will be returned.
403	Forbidden	The requestor is not the same as the originator of the request for analysis.

HTTP STATUS CODE	SCENARIO	RESULT
404	Not found	Unable to find the analysis request with the reference provided in the URL.

Expected response headers

NAME	TYPE	EXPECTED VALUE	REQUIRED?
Location	uri	URI to use in querying for the current status and to obtain the results	yes

Expected response body

The following table outlines the structure of the response for each request (HTTP 200 or 202 response only).

PROPERTY	TYPE	EXPECTED VALUE	REQUIRED?
privacyPolicy	string	The URI of the privacy policy.	Yes
progress	int	A value ranging from 0-100 percentage complete, where 10 means that processing is approximately 10% complete.	Yes
runCorrelationId	GUID	The request identifier that is included in each request. This can be used to correlate to the request, if needed.	Yes
status	string	`InProgress` is returned when the job is still being processed. `Failed` is returned when there was a catastrophic issue processing the job on the server. There should be more details in the error property. `Finished` is returned when the job has completed successfully without issues. `FinishedWithErrors` is returned when the job has completed successfully, however, one or more rules failed to complete without error. This is purely a signal for you to know that the report may not be complete. Microsoft is aware of these issues in the backend and will work to get things diagnosed and addressed.	Yes

PROPERTY	TYPE	EXPECTED VALUE	REQUIRED?
resultFileUris	array of strings	A list of URIs that allow for direct download of the output. There should be one per file that was included in the original analyze API call.	No. This is only included when processing has completed.
issueSummary	IssueSummary	Properties listed below	No. This is only included when processing has completed.
issueSummary.criticalIssueCount	int	Count of issues identified having a critical severity in the result	Yes
issueSummary.highIssueCount	int	Count of issues identified having a high severity in the result	Yes
issueSummary.mediumIssueCount	int	Count of issues identified having a medium severity in the result	Yes
issueSummary.lowIssueCount	int	Count of issues identified having a low severity in the result	Yes
issueSummary.informationalIssueCount	int	Count of issues identified having an informational severity in the result	Yes

Example: status check when done

This example issues a status check call with the result being a completion.

Request

```
GET [Geographical URI]/api/status/9E378E56-6F35-41E9-BF8B-C0CC88E2B832&api-version=1.0
Accept: application/json
Content-Type: application/json; charset=utf-8
x-ms-correlation-id: 9E378E56-6F35-41E9-BF8B-C0CC88E2B832
x-ms-tenant-id: F2E60E49-CB87-4C24-8D4F-908813B22506
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{
    "privacyPolicy":"https://go.microsoft.com/fwlink/?LinkID=310140",
    "progress":100,
    "resultFileUris":["https://fakeblob.blob.core.windows.net/report-files/mySolution.zip?sv=2017-11-
09&sr=b&sig=xyz&se=2019-06-11T20%3A27%3A59Z&sp=rd"],"runCorrelationId":"9E378E56-6F35-41E9-BF8B-
C0CC88E2B832","status":"Finished","issueSummary":
    {
        "informationalIssueCount":0,
        "lowIssueCount":0,
        "mediumIssueCount":302,
        "highIssueCount":30,
        "criticalIssueCount":0
    }
}
```

See also

Use the Power Apps checker web API
Retrieve the list of rulesets
Retrieve the list of rules
Upload a file
Invoke analysis

Retrieve the list of rules

7/15/2022 • 4 minutes to read • Edit Online

Rules are grouped together using a ruleset. A rule can be in no ruleset, or multiple rulesets. Use a `GET` request to obtain a list of all rules available, rules in a ruleset, or rulesets by calling the API `[Geographical URI]/api/rule`. There are a few variations to calling this API, however, the most common usage is to retrieve the list of rules for a specific ruleset.

> **NOTE**
> This API does not require an OAuth token, but can accept one if provided.

Headers

NAME	TYPE	EXPECTED VALUE	REQUIRED?
Accept-Language	string	The language code (e.g., en-US). The default is en-US.	no

Parameters

NAME	TYPE	EXPECTED VALUE	REQUIRED?
ruleset	string	The name or ID of the ruleset or a list of ruleset IDs, or names separated by a comma or semicolon (e.g., "Solution Checker").	no
includeMessageFormats	bool	When set to `true`, the list of possible message variations are included in the results of the language(s) requests, if available. This is useful for translations into multiple languages. If not needed, then do not provide this parameter or provide `false` as the value as this parameter will increase the size of the response and can increase processing time.	no

Expected responses

HTTP STATUS CODE	SCENARIO	RESULT
200	One or more results were found	See the example below. One or more results may be returned.
204	No results were found	No results in the response body.

Expected response body

The following table outlines the structure of the response for each request (HTTP 200 response only).

PROPERTY	TYPE	EXPECTED VALUE	REQUIRED?
code	string	The identifier of the rule, sometimes referred to as the Rule ID.	Yes
summary	string	A summary of the the rule.	Yes
description	string	More detailed description of the rule.	Yes
guidanceUrl	URI	The URL in which to find published guidance. There may be some cases where there is not a dedicated supporting guidance article.	Yes
include	boolean	Signals to the service that the rule is to be included in the analysis. This will be `true` for this API.	No
messageTemplates	array	This property value is included only when `includeMessageFormats` is `true`.	No
messageTemplates.ruleId	string	Returns the same ID value as the `code` property.	Yes
messageTemplates.messageTemplateId	string	An identifier used in the Static Analysis Results Interchange Format (SARIF) report to signal an issue message variation for the rule.	Yes
messageTemplates.messageTemplate	string	The text of the message variation for the issue scenario that the rule reports. This is a format string that may contain tokens in which arguments provided in the SARIF report can be used to construct a detailed message.	Yes

Example: retrieve rules for a ruleset in another language

This example returns data for all of the rules in the *Solution Checker* ruleset in the French language. If the desired language is English, then just remove the Accept-Language header.

Request

```
GET [Geographical URI]/api/rule?ruleset=083A2EF5-7E0E-4754-9D88-9455142DC08B&api-version=1.0
x-ms-correlation-id: 9E378E56-6F35-41E9-BF8B-C0CC88E2B832
Accept: application/json
Content-Type: application/json; charset=utf-8
Accept-Language: fr
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

[
    {
        "description": "Ne pas implémenter d’activités de workflow Microsoft Dynamics CRM 4.0",
        "guidanceUrl": "https://go.microsoft.com/fwlink/?LinkID=398563&error=il-avoid-crm4-
wf&client=PAChecker",
        "include": true,
        "code": "il-avoid-crm4-wf",
        "summary": "Ne pas implémenter d’activités de workflow Microsoft Dynamics CRM 4.0",
        "howToFix": {
            "summary": ""
        }
    },
    {
        "description": "Utiliser InvalidPluginExecutionException dans des plug-ins et activités de
workflow",
        "guidanceUrl": "https://go.microsoft.com/fwlink/?LinkID=398563&error=il-use-standard-
exception&client=PAChecker",
        "include": true,
        "code": "il-use-standard-exception",
        "summary": "Utiliser InvalidPluginExecutionException dans des plug-ins et activités de workflow",
        "howToFix": {
            "summary": ""
        }
    },
...
]
```

Example: retrieve all

This example returns data for all of the rules available.

Request

```
GET [Geographical URI]/api/rule?api-version=1.0
Accept: application/json
Content-Type: application/json; charset=utf-8
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

[
    {
        "description": "Retrieve specific columns for an entity via query APIs",
        "guidanceUrl": "https://go.microsoft.com/fwlink/?LinkID=398563&error=il-specify-
column&client=PAChecker",
        "include": true,
        "code": "il-specify-column",
        "summary": "Retrieve specific columns for an entity via query APIs",
        "howToFix": {
            "summary": ""
        }
    },
    {
        "description": "Do not duplicate plug-in step registration",
        "guidanceUrl": "https://go.microsoft.com/fwlink/?LinkID=398563&error=meta-remove-dup-
reg&client=PAChecker",
        "include": true,
        "code": "meta-remove-dup-reg",
        "summary": "Do not duplicate plug-in step registration",
        "howToFix": {
            "summary": ""
        }
    },
...
]
```

Example: retrieve for a ruleset with message formats

This example returns data for all of the rules in the *Solution Checker* ruleset in the French language. If the desired language is English, then just remove the Accept-Language header.

Request

```
GET [Geographical URI]/api/rule?ruleset=083A2EF5-7E0E-4754-9D88-9455142DC08B&includeMessageFormats=true&api-
version=1.0
Accept: application/json
Content-Type: application/json; charset=utf-8
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

[
    {
        "description": "Do not implement Microsoft Dynamics CRM 4.0 workflow activities",
        "guidanceUrl": "https://go.microsoft.com/fwlink/?LinkID=398563&error=il-avoid-crm4-
wf&client=PAChecker",
        "include": true,
        "code": "il-avoid-crm4-wf",
        "summary": "Do not implement Microsoft Dynamics CRM 4.0 workflow activities",
        "howToFix": {
            "summary": ""
        },
        "messageTemplates": [
            {
                "ruleId": "il-avoid-crm4-wf",
                "messageTemplateId": "message1",
                "messageTemplate": "Update the {0} class to derive from
```

```
System.Workflow.Activities.CodeActivity, refactor Execute method implementation, and remove
Microsoft.Crm.Workflow.CrmWorkflowActivityAttribute from type"
            },
            {
                "ruleId": "il-avoid-crm4-wf",
                "messageTemplateId": "message2",
                "messageTemplate": "Change the {0} property's type from {1} to {2} Argument &lt;T&gt; type"
            },
            {
                "ruleId": "il-avoid-crm4-wf",
                "messageTemplateId": "message3",
                "messageTemplate": "Replace the Microsoft.Crm.Workflow.Crm{0}Attribute with
Microsoft.Xrm.Sdk.Workflow.{0}Attribute"
            },
            {
                "ruleId": "il-avoid-crm4-wf",
                "messageTemplateId": "message4",
                "messageTemplate": "Remove the {0} System.Workflow.ComponentModel.DependencyProperty type
field"
            }
        ]
    },
    {
        "description": "Use InvalidPluginExecutionException in plug-ins and workflow activities",
        "guidanceUrl": "https://go.microsoft.com/fwlink/?LinkID=398563&error=il-use-standard-
exception&client=PAChecker",
        "include": true,
        "code": "il-use-standard-exception",
        "summary": "Use InvalidPluginExecutionException in plug-ins and workflow activities",
        "howToFix": {
            "summary": ""
        },
        "messageTemplates": [
            {
                "ruleId": "il-use-standard-exception",
                "messageTemplateId": "message1",
                "messageTemplate": "An unguarded throw of type {0} was detected. Refactor this code to
either throw an exception of type InvalidPluginExecutionException or guard against thrown exceptions of
other types."
            },
            {
                "ruleId": "il-use-standard-exception",
                "messageTemplateId": "message2",
                "messageTemplate": "An unguarded rethrow of type {0} was detected. Refactor this code to
either throw an exception of type InvalidPluginExecutionException or guard against thrown exceptions of
other types."
            }
        ]
    },
...
]
```


See also

Use the Power Apps checker web API
Retrieve the list of rulesets
Upload a file
Invoke analysis
Check for analysis status

Retrieve the list of rulesets

7/15/2022 • 2 minutes to read • Edit Online

Rules are grouped together using a ruleset. Rulesets can have one or more rules with no limit. A rule can be in no ruleset, or multiple rulesets. Use a `GET` request to obtain a list of all rulesets available by calling the API, [Geographical URI]/api/ruleset.

> **NOTE**
> This API does not require an OAuth token, but can accept one if provided.

Expected responses

HTTP STATUS CODE	SCENARIO	RESULT
200	One or more results were found	See example below. One or more results may be returned.
204	No results were found	No results response body is returned.

Expected response body

The following table outlines the structure of the response for each request (HTTP 200 response only).

PROPERTY	TYPE	EXPECTED VALUE	REQUIRED?
id	Guid	Identifier of the ruleset	Yes
name	string	Friendly name of the ruleset	Yes

Example: retrieve all rulesets

This example returns data for all of the rulesets available.

Request

```
GET [Geographical URI]/api/ruleset?api-version=1.0
Accept: application/json
x-ms-correlation-id: 9E378E56-6F35-41E9-BF8B-C0CC88E2B832
Content-Type: application/json; charset=utf-8
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

[
    {
        "id": "083a2ef5-7e0e-4754-9d88-9455142dc08b",
        "name": "AppSource Certification"
    },
    {
        "id": "0ad12346-e108-40b8-a956-9a8f95ea18c9",
        "name": "Solution Checker"
    }
]
```

See also

Use the Power Apps checker web API
Retrieve the list of rules
Upload a file
Invoke analysis
Check for analysis status

Upload a file for analysis

7/15/2022 • 2 minutes to read • Edit Online

The initiation of an analysis job requires a path to an Azure blob that is accessible by URL. The ability to upload a file to Azure blob storage in the specified geography using the upload service is provided. It is not required that the upload API be used in order to run analysis. You can upload using a `POST` request to the following: `[Geographical URI]/api/upload?api-version=1.0`. Uploading a file up to 30 MB in size is supported. For anything larger you will need to provide your own externally accessible Azure storage and SAS URI.

> **NOTE**
> This API does require an OAuth token.

Headers

NAME	TYPE	EXPECTED VALUE	REQUIRED?
Authorization	string	The OAuth 1 bearer token with Azure Active Directory (AAD) Application ID claim.	yes
x-ms-tenant-id	GUID	The ID of the tenant for the application.	yes
x-ms-correlation-id	GUID	The Identifier for the analysis run. You should provide the same ID for the entire execution (upload, analyze, status).	yes
Content-Type	object	multipart/form-data	yes
Content-Disposition	object	Include the name and filename parameters, for example: `form-data; name="solution1.zip"; filename="solution1.zip"`	yes

Expected responses

HTTP STATUS CODE	SCENARIO	RESULT
200	Upload was a success	No result body
400	A non zip file was sent, incorrect parameters, or a file was included with a virus	No result body

HTTP STATUS CODE	SCENARIO	RESULT
413	File is too large	No result body

Example: upload a file

This example demonstrates how a file can be uploaded that is to be analyzed.

Request

```
POST [Geographical URI]/api/upload
Accept: application/json
x-ms-correlation-id: 9E378E56-6F35-41E9-BF8B-C0CC88E2B832
x-ms-tenant-id: F2E60E49-CB87-4C24-8D4F-908813B22506
Content-Type: multipart/form-data
Content-Disposition: form-data; name=mySolution.zip; filename=mySolution.zip
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

["https://mystorage.blob.core.windows.net/solution-files/0a4cd700-d1d0-4ef8-8318-
e4844cc1636c/mySolution.zip?sv=2017-11-09&sr=b&sig=xyz&se=2019-06-11T19%3A05%3A20Z&sp=rd"]
```

See also

Use the Power Apps checker web API
Retrieve the list of rulesets
Retrieve the list of rules
Invoke analysis
Check for analysis status

Work with solutions using PowerShell

7/15/2022 • 2 minutes to read • Edit Online

The Power Apps checker web API provides a mechanism to run static analysis checks against customizations and extensions to the Microsoft Dataverse platform. It's available for makers and developers to perform rich static analysis checks on their solutions against a set of best practice rules to quickly identify problematic patterns. To begin using the Power Apps checker Web API, see Getting started.

The checker service provides the logic for the solution checker feature in the Power Apps maker portal and is included as part of the automation for applications submitted to AppSource. In this section, we describe how to run a best practice solution analysis of your customizations and extensions in your DevOps pipeline to verify the quality of your solution component.

TIP

Tip #1: Consider using the PowerShell module, Microsoft.PowerApps.Checker.PowerShell, instead of using the web API. The module is a community supported tool that's available in the PowerShell Gallery. The current restriction is that it does require Windows PowerShell in your project pipeline. If you're unable to meet this requirement, interacting with the web APIs directly will likely be the best approach.

Tip #2: You can easily manage solutions using PowerShell as part of your custom automation. Refer to the Microsoft.Xrm.Data.PowerShell module, which is also a community created and supported tool. See sample code here. For example:

```
Export-CrmSolution "MySolution"
```

```
Export-CrmSolution -conn $conn -SolutionName "MySolution" -Managed -SolutionFilePath "C:\temp" -
SolutionZipFileName "MySolution_Managed.zip"
```

```
Import-CrmSolutionAsync -SolutionFilePath c:\temp\mysolution.zip -ActivateWorkflows -
OverwriteUnManagedCustomizations -MaxWaitTimeInSeconds 600
```

In addition, there is a checker task in the Azure DevOps build tools include a checker task. For more information about using that task in your build pipeline, see Quality check.

See also

PowerShell modules
Online Management API (REST) to manage environments

Work with solutions using the Dataverse SDK

7/15/2022 • 10 minutes to read • Edit Online

As part of your development to production lifecycle you may want to create custom automation to handle certain tasks. For example, in your DevOps project pipeline you might want to execute some custom code or script that creates a sandbox environment, imports an unmanaged solution, exports that unmanaged solution as a managed solution, and, finally, deletes the environment. You can do this and more by using the APIs that are available to you. Below are some examples of what you can accomplish using the Dataverse SDK for .NET and custom code.

> **NOTE**
>
> You can also perform these same operations using the Web API. The related actions are: ImportSolution, ExportSolution, CloneAsPatch, and CloneAsSolution.

Create, export, or import an unmanaged solution

Let's see how to perform some common solution operations by using C# code. To view the complete working C# code sample that demonstrates these types of solution operations (and more), see Sample: Work with solutions.

Create a publisher

Every solution requires a publisher, represented by the Publisher entity. A publisher requires the following:

- A customization prefix
- A unique name
- A friendly name

> **NOTE**
>
> For a healthy ALM approach, always use a custom publisher and solution, not the default solution and publisher, for deploying your customizations.

The following code sample first defines a publisher and then checks to see whether the publisher already exists based on the unique name. If it already exists, the customization prefix might have been changed, so this sample seeks to capture the current customization prefix. The `PublisherId` is also captured so that the publisher record can be deleted. If the publisher isn't found, a new publisher is created using the IOrganizationService.Create method.

```
// Define a new publisher
Publisher _myPublisher = new Publisher
{
   UniqueName = "contoso-publisher",
   FriendlyName = "Contoso publisher",
   SupportingWebsiteUrl =
      "https://docs.microsoft.com/powerapps/developer/data-platform/overview",
   CustomizationPrefix = "contoso",
   EMailAddress = "someone@contoso.com",
   Description = "This publisher was created from sample code"
};

// Does the publisher already exist?
QueryExpression querySamplePublisher = new QueryExpression
{
   EntityName = Publisher.EntityLogicalName,
   ColumnSet = new ColumnSet("publisherid", "customizationprefix"),
   Criteria = new FilterExpression()
};

querySamplePublisher.Criteria.AddCondition("uniquename", ConditionOperator.Equal,
   _myPublisher.UniqueName);

EntityCollection querySamplePublisherResults =
   _serviceProxy.RetrieveMultiple(querySamplePublisher);

Publisher SamplePublisherResults = null;

// If the publisher already exists, use it
if (querySamplePublisherResults.Entities.Count > 0)
{
   SamplePublisherResults = (Publisher)querySamplePublisherResults.Entities[0];
   _publisherId = (Guid)SamplePublisherResults.PublisherId;
   _customizationPrefix = SamplePublisherResults.CustomizationPrefix;
}

// If the publisher doesn't exist, create it
if (SamplePublisherResults == null)
{
   _publisherId = _serviceProxy.Create(_myPublisher);

   Console.WriteLine(String.Format("Created publisher: {0}.",
   _myPublisher.FriendlyName));

   _customizationPrefix = _myPublisher.CustomizationPrefix;
}
```

Create an unmanaged solution

After you have a custom publisher available, you can then create an unmanaged solution. The following table lists the fields with descriptions that a solution contains.

FIELD LABEL	DESCRIPTION
Display Name	The name for the solution.
Name	Microsoft Dataverse generates a unique name based on the **Display Name**. You can edit the unique name. The unique name must only contain alphanumeric characters or the underscore character.
Publisher	Use the **Publisher** lookup to associate the solution with a publisher.

FIELD LABEL	DESCRIPTION
Version	Specify a version by using the following format: *major.minor.build.revision* (for example, 1.0.0.0.
Configuration Page	If you include an HTML Web resource in your solution, you can use this lookup to add it as your designated solution configuration page.
Description	Use this field to include any relevant details about your solution.

Below is sample code to create an unmanaged solution that uses the publisher we created in the previous section.

```
// Create a solution
Solution solution = new Solution
{
   UniqueName = "sample-solution",
   FriendlyName = "Sample solution",
   PublisherId = new EntityReference(Publisher.EntityLogicalName, _publisherId),
   Description = "This solution was created by sample code.",
   Version = "1.0"
};

// Check whether the solution already exists
QueryExpression queryCheckForSampleSolution = new QueryExpression
{
   EntityName = Solution.EntityLogicalName,
   ColumnSet = new ColumnSet(),
   Criteria = new FilterExpression()
};

queryCheckForSampleSolution.Criteria.AddCondition("uniquename",
   ConditionOperator.Equal, solution.UniqueName);

// Attempt to retrieve the solution
EntityCollection querySampleSolutionResults =
   _serviceProxy.RetrieveMultiple(queryCheckForSampleSolution);

// Create the solution if it doesn't already exist
Solution SampleSolutionResults = null;

if (querySampleSolutionResults.Entities.Count > 0)
{
   SampleSolutionResults = (Solution)querySampleSolutionResults.Entities[0];
   _solutionsSampleSolutionId = (Guid)SampleSolutionResults.SolutionId;
}

if (SampleSolutionResults == null)
{
   _solutionsSampleSolutionId = _serviceProxy.Create(solution);
}
```

After you create an unmanaged solution, you can add solution components by creating them in the context of this solution or by adding existing components from other solutions. More information: Add a new solution component and Add an existing solution component

Export an unmanaged solution

This code sample shows how to export an unmanaged solution or package a managed solution. The code uses the ExportSolutionRequest class to export a compressed file representing an unmanaged solution. The

option to create a managed solution is set by using the Managed property. This sample saves a file named *samplesolution.zip* to the output folder.

```
// Export a solution
ExportSolutionRequest exportSolutionRequest = new ExportSolutionRequest();
exportSolutionRequest.Managed = false;
exportSolutionRequest.SolutionName = solution.UniqueName;

ExportSolutionResponse exportSolutionResponse =
   (ExportSolutionResponse)_serviceProxy.Execute(exportSolutionRequest);

byte[] exportXml = exportSolutionResponse.ExportSolutionFile;
string filename = solution.UniqueName + ".zip";

File.WriteAllBytes(outputDir + filename, exportXml);

Console.WriteLine("Solution exported to {0}.", outputDir + filename);
```

Import an unmanaged solution

Importing (or upgrading) a solution by using code is accomplished with ImportSolutionRequest.

```
// Install or upgrade a solution
byte[] fileBytes = File.ReadAllBytes(ManagedSolutionLocation);

ImportSolutionRequest impSolReq = new ImportSolutionRequest()
{
   CustomizationFile = fileBytes
};

_serviceProxy.Execute(impSolReq);
```

Tracking import success

You can use the ImportJob entity to capture data about the success of the solution import. When you specify an `ImportJobId` for the ImportSolutionRequest, you can use that value to query the ImportJob entity about the status of the import. The `ImportJobId` can also be used to download an import log file using the RetrieveFormattedImportJobResultsRequest message.

```
// Monitor solution import success
byte[] fileBytesWithMonitoring = File.ReadAllBytes(ManagedSolutionLocation);

ImportSolutionRequest impSolReqWithMonitoring = new ImportSolutionRequest()
{
   CustomizationFile = fileBytes,
   ImportJobId = Guid.NewGuid()
};

_serviceProxy.Execute(impSolReqWithMonitoring);

ImportJob job = (ImportJob)_serviceProxy.Retrieve(ImportJob.EntityLogicalName,
   impSolReqWithMonitoring.ImportJobId, new ColumnSet(new System.String[] { "data",
   "solutionname" }));

System.Xml.XmlDocument doc = new System.Xml.XmlDocument();
doc.LoadXml(job.Data);

String ImportedSolutionName =
   doc.SelectSingleNode("//solutionManifest/UniqueName").InnerText;

String SolutionImportResult =
   doc.SelectSingleNode("//solutionManifest/result/\@result").Value;

Console.WriteLine("Report from the ImportJob data");

Console.WriteLine("Solution Unique name: {0}", ImportedSolutionName);

Console.WriteLine("Solution Import Result: {0}", SolutionImportResult);

Console.WriteLine("");

// This code displays the results for Global Option sets installed as part of a
// solution.

System.Xml.XmlNodeList optionSets = doc.SelectNodes("//optionSets/optionSet");

foreach (System.Xml.XmlNode node in optionSets)
{
   string OptionSetName = node.Attributes["LocalizedName"].Value;
   string result = node.FirstChild.Attributes["result"].Value;

   if (result == "success")
   {
      Console.WriteLine("{0} result: {1}",OptionSetName, result);
   }
   else
   {
      string errorCode = node.FirstChild.Attributes["errorcode"].Value;
      string errorText = node.FirstChild.Attributes["errortext"].Value;

      Console.WriteLine("{0} result: {1} Code: {2} Description: {3}",OptionSetName,
      result, errorCode, errorText);
   }
}
```

The content of the `Data` property is a string representing the solution XML file.

Add and remove solution components

Learn how to add and remove solution components using code.

Add a new solution component

This sample shows how to create a solution component that is associated with a specific solution. If you don't

associate the solution component to a specific solution when it is created it will only be added to the default solution and you will need to add it to a solution manually or by using the code included in the Add an existing solution component.

This code creates a new global option set and adds it to the solution with a unique name equal to `_primarySolutionName`.

```
OptionSetMetadata optionSetMetadata = new OptionSetMetadata()
{
   Name = _globalOptionSetName,
   DisplayName = new Label("Example Option Set", _languageCode),
   IsGlobal = true,
   OptionSetType = OptionSetType.Picklist,
   Options =
{
   new OptionMetadata(new Label("Option 1", _languageCode), 1),
   new OptionMetadata(new Label("Option 2", _languageCode), 2)
}
};
CreateOptionSetRequest createOptionSetRequest = new CreateOptionSetRequest
{
   OptionSet = optionSetMetadata
};

createOptionSetRequest.SolutionUniqueName = _primarySolutionName;
_serviceProxy.Execute(createOptionSetRequest);
```

Add an existing solution component

This sample shows how to add an existing solution component to a solution.

The following code uses the AddSolutionComponentRequest to add the `Account` entity as a solution component to an unmanaged solution.

```
// Add an existing Solution Component
// Add the Account entity to the solution
RetrieveEntityRequest retrieveForAddAccountRequest = new RetrieveEntityRequest()
{
   LogicalName = Account.EntityLogicalName
};
RetrieveEntityResponse retrieveForAddAccountResponse =
(RetrieveEntityResponse)_serviceProxy.Execute(retrieveForAddAccountRequest);
AddSolutionComponentRequest addReq = new AddSolutionComponentRequest()
{
   ComponentType = (int)componenttype.Entity,
   ComponentId = (Guid)retrieveForAddAccountResponse.EntityMetadata.MetadataId,
   SolutionUniqueName = solution.UniqueName
};
_serviceProxy.Execute(addReq);
```

Remove a solution component

This sample shows how to remove a solution component from an unmanaged solution. The following code uses the RemoveSolutionComponentRequest to remove an entity solution component from an unmanaged solution. The `solution.UniqueName` references the solution created in Create an unmanaged solution.

```
// Remove a Solution Component
// Remove the Account entity from the solution
RetrieveEntityRequest retrieveForRemoveAccountRequest = new RetrieveEntityRequest()
{
   LogicalName = Account.EntityLogicalName
};
RetrieveEntityResponse retrieveForRemoveAccountResponse =
(RetrieveEntityResponse)_serviceProxy.Execute(retrieveForRemoveAccountRequest);

RemoveSolutionComponentRequest removeReq = new RemoveSolutionComponentRequest()
{
   ComponentId = (Guid)retrieveForRemoveAccountResponse.EntityMetadata.MetadataId,
   ComponentType = (int)componenttype.Entity,
   SolutionUniqueName = solution.UniqueName
};
_serviceProxy.Execute(removeReq);
```

Delete a solution

The following sample shows how to retrieve a solution using the solution `uniquename` and then extract the `solutionid` from the results. The sample then uses the `solutionid` with theIOrganizationService. Delete method to delete the solution.

```
// Delete a solution

QueryExpression queryImportedSolution = new QueryExpression
{
    EntityName = Solution.EntityLogicalName,
    ColumnSet = new ColumnSet(new string[] { "solutionid", "friendlyname" }),
    Criteria = new FilterExpression()
};


queryImportedSolution.Criteria.AddCondition("uniquename", ConditionOperator.Equal, ImportedSolutionName);

Solution ImportedSolution = (Solution)_serviceProxy.RetrieveMultiple(queryImportedSolution).Entities[0];

_serviceProxy.Delete(Solution.EntityLogicalName, (Guid)ImportedSolution.SolutionId);

Console.WriteLine("Deleted the {0} solution.", ImportedSolution.FriendlyName);
```

Cloning, patching, and upgrading

You can perform additional solution operations by using the available APIs. For cloning and patching solutions use the CloneAsPatchRequest and CloneAsSolutionRequest. For information about cloning and patching, see Create solution patches.

When performing solution upgrades use the StageAndUpgradeRequest and DeleteAndPromoteRequest. For more information about the process of staging and upgrades, see Upgrade or update a solution.

Detect solution dependencies

This sample shows how to create a report showing the dependencies between solution components.

This code will:

- Retrieve all the components for a solution.
- Retrieve all the dependencies for each component.

- For each dependency found display a report describing the dependency.

```
// Grab all Solution Components for a solution.
QueryByAttribute componentQuery = new QueryByAttribute
{
    EntityName = SolutionComponent.EntityLogicalName,
    ColumnSet = new ColumnSet("componenttype", "objectid", "solutioncomponentid", "solutionid"),
    Attributes = { "solutionid" },

    // In your code, this value would probably come from another query.
    Values = { _primarySolutionId }
};

IEnumerable<SolutionComponent> allComponents =
    _serviceProxy.RetrieveMultiple(componentQuery).Entities.Cast<SolutionComponent>();

foreach (SolutionComponent component in allComponents)
{
    // For each solution component, retrieve all dependencies for the component.
    RetrieveDependentComponentsRequest dependentComponentsRequest =
        new RetrieveDependentComponentsRequest
        {
            ComponentType = component.ComponentType.Value,
            ObjectId = component.ObjectId.Value
        };
    RetrieveDependentComponentsResponse dependentComponentsResponse =
        (RetrieveDependentComponentsResponse)_serviceProxy.Execute(dependentComponentsRequest);

    // If there are no dependent components, we can ignore this component.
    if (dependentComponentsResponse.EntityCollection.Entities.Any() == false)
        continue;

    // If there are dependencies upon this solution component, and the solution
    // itself is managed, then you will be unable to delete the solution.
    Console.WriteLine("Found {0} dependencies for Component {1} of type {2}",
        dependentComponentsResponse.EntityCollection.Entities.Count,
        component.ObjectId.Value,
        component.ComponentType.Value
        );
    //A more complete report requires more code
    foreach (Dependency d in dependentComponentsResponse.EntityCollection.Entities)
    {
        DependencyReport(d);
    }
}
```

The `DependencyReport` method is in the following code sample.

Dependency report

The `DependencyReport` method provides a friendlier message based on information found within the dependency.

> **NOTE**
> In this sample the method is only partially implemented. It can display messages only for attribute and option set solution components.

```
/// <summary>
/// Shows how to get a more friendly message based on information within the dependency
/// <param name="dependency">A Dependency returned from the RetrieveDependentComponents message</param>
/// </summary>
public void DependencyReport(Dependency dependency)
{
```

```
{
 // These strings represent parameters for the message.
    String dependentComponentName = "";
    String dependentComponentTypeName = "";
    String dependentComponentSolutionName = "";
    String requiredComponentName = "";
    String requiredComponentTypeName = "";
    String requiredComponentSolutionName = "";

 // The ComponentType global Option Set contains options for each possible component.
    RetrieveOptionSetRequest componentTypeRequest = new RetrieveOptionSetRequest
    {
     Name = "componenttype"
    };

    RetrieveOptionSetResponse componentTypeResponse =
(RetrieveOptionSetResponse)_serviceProxy.Execute(componentTypeRequest);
    OptionSetMetadata componentTypeOptionSet = (OptionSetMetadata)componentTypeResponse.OptionSetMetadata;
 // Match the Component type with the option value and get the label value of the option.
    foreach (OptionMetadata opt in componentTypeOptionSet.Options)
    {
     if (dependency.DependentComponentType.Value == opt.Value)
     {
      dependentComponentTypeName = opt.Label.UserLocalizedLabel.Label;
     }
     if (dependency.RequiredComponentType.Value == opt.Value)
     {
      requiredComponentTypeName = opt.Label.UserLocalizedLabel.Label;
     }
    }
 // The name or display name of the compoent is retrieved in different ways depending on the component type
    dependentComponentName = getComponentName(dependency.DependentComponentType.Value,
(Guid)dependency.DependentComponentObjectId);
    requiredComponentName = getComponentName(dependency.RequiredComponentType.Value,
(Guid)dependency.RequiredComponentObjectId);

 // Retrieve the friendly name for the dependent solution.
    Solution dependentSolution = (Solution)_serviceProxy.Retrieve
     (
      Solution.EntityLogicalName,
      (Guid)dependency.DependentComponentBaseSolutionId,
      new ColumnSet("friendlyname")
     );
    dependentComponentSolutionName = dependentSolution.FriendlyName;

 // Retrieve the friendly name for the required solution.
    Solution requiredSolution = (Solution)_serviceProxy.Retrieve
      (
       Solution.EntityLogicalName,
       (Guid)dependency.RequiredComponentBaseSolutionId,
       new ColumnSet("friendlyname")
      );
    requiredComponentSolutionName = requiredSolution.FriendlyName;

 // Display the message
     Console.WriteLine("The {0} {1} in the {2} depends on the {3} {4} in the {5} solution.",
     dependentComponentName,
     dependentComponentTypeName,
     dependentComponentSolutionName,
     requiredComponentName,
     requiredComponentTypeName,
     requiredComponentSolutionName);
}
```

Detect whether a solution component may be deleted

Use the RetrieveDependenciesForDeleteRequest message to identify any other solution components which would prevent a given solution component from being deleted. The following code sample looks for any

attributes using a known global optionset. Any attribute using the global optionset would prevent the global optionset from being deleted.

```
// Use the RetrieveOptionSetRequest message to retrieve
// a global option set by it's name.
RetrieveOptionSetRequest retrieveOptionSetRequest =
    new RetrieveOptionSetRequest
    {
     Name = _globalOptionSetName
    };

// Execute the request.
RetrieveOptionSetResponse retrieveOptionSetResponse =
    (RetrieveOptionSetResponse)_serviceProxy.Execute(
    retrieveOptionSetRequest);
_globalOptionSetId = retrieveOptionSetResponse.OptionSetMetadata.MetadataId;
if (_globalOptionSetId != null)
{
 // Use the global OptionSet MetadataId with the appropriate componenttype
 // to call RetrieveDependenciesForDeleteRequest
 RetrieveDependenciesForDeleteRequest retrieveDependenciesForDeleteRequest = new
RetrieveDependenciesForDeleteRequest
{
 ComponentType = (int)componenttype.OptionSet,
 ObjectId = (Guid)_globalOptionSetId
};

 RetrieveDependenciesForDeleteResponse retrieveDependenciesForDeleteResponse =
  (RetrieveDependenciesForDeleteResponse)_serviceProxy.Execute(retrieveDependenciesForDeleteRequest);
 Console.WriteLine("");
 foreach (Dependency d in retrieveDependenciesForDeleteResponse.EntityCollection.Entities)
 {

  if (d.DependentComponentType.Value == 2)//Just testing for Attributes
  {
   String attributeLabel = "";
   RetrieveAttributeRequest retrieveAttributeRequest = new RetrieveAttributeRequest
   {
    MetadataId = (Guid)d.DependentComponentObjectId
   };
   RetrieveAttributeResponse retrieveAttributeResponse =
(RetrieveAttributeResponse)_serviceProxy.Execute(retrieveAttributeRequest);

   AttributeMetadata attmet = retrieveAttributeResponse.AttributeMetadata;

   attributeLabel = attmet.DisplayName.UserLocalizedLabel.Label;

    Console.WriteLine("An {0} named {1} will prevent deleting the {2} global option set.",
   (componenttype)d.DependentComponentType.Value,
   attributeLabel,
   _globalOptionSetName);
  }
 }
}
```

Solution staging, with asynchronous import and export

7/15/2022 • 4 minutes to read • Edit Online

Have you ever run into the situation during the import or export of a large solution where the operation times out? If so, you may be a candidate for performing the solution import/export asynchronously. This topic describes how to initiate the asynchronous import or export using the Dataverse SDK for .NET and Web APIs.

Staging a solution

In comparison to importing a solution where the solution is imported and available in the environment right away, staging breaks the import process into more controllable phases. The staging process imports the solution as a "holding" solution where the administrator can decide when to make the staged solution available to users, or to perform an upgrade (in the case of a solution upgrade) in the target environment. Part of the staging process is validation of the staged solution. In this way you can stage the solution, know that the solution is valid, and schedule when to apply that solution or upgrade to the target environment.

OPERATION	WEB API	DATAVERSE SDK FOR .NET
Stage a solution	StageSolution	StageSolutionRequest

The result of staging the solution will be a collection of validation results indicating success or failure and (if successful) a `StageSolutionUploadId` to be used in the `ImportSolutionAsync` call. See the import solution Web API sample code above for an example of how this is done.

```
public static StageSolutionResults StageSolution(
    IOrganizationService service,
    string solutionFilePath)
{
  // Stage the solution
  var req = new StageSolutionRequest();

  byte[] fileBytes = File.ReadAllBytes(solutionFilePath);
  req["CustomizationFile"] = fileBytes;
  var res = service.Execute(req);

  return (res["StageSolutionResults"] as StageSolutionResults);
}
```

Solution import

`ImportSolution` is the action (or message) that performs the synchronous import operation. To execute the import operation asynchronously use `ImportSolutionAsync`.

OPERATION	WEB API	DATAVERSE SDK FOR .NET
Import a solution	ImportSolutionAsync	ImportSolutionAsyncRequest

Now let's take a look at some example code that demonstrates `ImportSolutionAsync`.

```
public static ImportSolutionAsyncResponse ImportSolution(
    IOrganizationService service,
    StageSolutionResults stagingResults,
    Dictionary<string,Guid> connectionIds,
    Dictionary<string,string> envarValues )
{
    // Import the staged solution
    var componentDetails = stagingResults.SolutionComponentsDetails;

    // TODO These are not referenced in the code but are usefull to explore
    var missingDependencies = stagingResults.MissingDependencies;   // Contains missing dependencies
    var solutionDetails = stagingResults.SolutionDetails;           // Contains solution details

    var connectionReferences = componentDetails.Where(x => string.Equals(x.ComponentTypeName,
"connectionreference"));
    var envVarDef = componentDetails.Where(x => string.Equals(x.ComponentTypeName,
"environmentvariabledefinition"));
    var envVarValue = componentDetails.Where(x => string.Equals(x.ComponentTypeName,
"environmentvariablevalue"));

    var componentParams = new EntityCollection();

    // Add each connection reference to the component parmameters entity collection.
    foreach (var conn in connectionReferences)
    {
        var e = new Entity("connectionreference")
        {
            ["connectionreferencelogicalname"] =
conn.Attributes["connectionreferencelogicalname"].ToString(),
            ["connectionreferencedisplayname"] =
conn.Attributes["connectionreferencedisplayname"].ToString(),
            ["connectorid"] = conn.Attributes["connectorid"].ToString(),
            ["connectionid"] = connectionIds[conn.ComponentName]
        };
        componentParams.Entities.Add(e);
    }

    // Add each environment variable to the component parmameters entity collection.
    foreach (var value in envVarValue)
    {
        var e = new Entity("environmentvariablevalue")
        {
            ["schemaname"] = value.Attributes["schemaname"].ToString(),
            ["value"] = envarValues[value.ComponentName]
        };

        if (value.Attributes.ContainsKey("environmentvariablevalueid"))
        {
            e["environmentvariablevalueid"] = value.Attributes["environmentvariablevalueid"].ToString();
        }
        componentParams.Entities.Add(e);
    }

    // Import the solution
    var importSolutionReq = new ImportSolutionAsyncRequest();
    importSolutionReq.ComponentParameters = componentParams;
    importSolutionReq.SolutionParameters = new SolutionParameters { StageSolutionUploadId =
stagingResults.StageSolutionUploadId };
    var response = service.Execute(importSolutionReq) as ImportSolutionAsyncResponse;

    return (response);
}
```

`ImportSolutionAsync` shares many input parameters with `ImportSolution` but adds `ComponentParameters` and `SolutionParameters`. `ComponentParameters` can be used to overwrite the component data in the solution's customization XML file. `SolutionParameters` can be used to pass the `StageSolutionUploadId` of a staged solution

as was shown in the example Web API code. More information: Staging a solution

The response returned from `ImportSolutionAsync` contains `ImportJobKey` and `AsyncOperationId`. The `ImportJobKey` value can be used to obtain the import result and the `AsyncOperationId` value can be used to track the import job status.

```
public static void CheckImportStatus(
    IOrganizationService service,
    Guid asyncOperationId,
    Guid importJobKey)
{
    // Get solution import status
    var finished = false;
    Entity asyncOperation = null;
    // Wait until the async job is finished
    while (!finished)
    {
        asyncOperation = service.Retrieve("asyncoperation", asyncOperationId, new ColumnSet("statecode",
"statuscode"));
        OptionSetValue statecode = (OptionSetValue)asyncOperation["statecode"];
        if (statecode.Value == 3)
        {
            finished = true;
        }
        else
        {
            Thread.Sleep(10000);
        }
    }
    // Solution import completed successfully
    OptionSetValue statuscode = (OptionSetValue)asyncOperation["statuscode"];
    if (statuscode.Value == 30)
    {
        Console.WriteLine("The solution import completed successfully.");
    }
    else if (asyncOperation["statuscode"].ToString() == "31")  // Solution import failed
    {
        Console.WriteLine("The solution import failed.");
        var getLogReq = new RetrieveFormattedImportJobResultsRequest { ImportJobId = importJobKey };
        var importJob = service.Execute(getLogReq) as RetrieveFormattedImportJobResultsResponse;
        // TODO Do something with the import job results
    }
}
```

Solution export

`ExportSolution` is the action (or message) that performs the synchronous export operation. To execute the export operation asynchronously use `ExportSolutionAsync`.

OPERATION	WEB API	DATAVERSE SDK FOR .NET
Export a solution	ExportSolutionAsync	ExportSolutionAsyncRequest
Download an exported solution file	DownloadSolutionExportData	DownloadSolutionExportDataRequest

Now let's take a look at some example code that demonstrates `ExportSolutionAsync`.

```
// Where 'service' is a pre-configured Organization service instance.
var service = (OrganizationServiceProxy)xsc.CreateOrganizationService();

var req = new OrganizationRequest("ExportSolutionAsync");
req.Parameters.Add("SolutionName", "ExportSolutionAsyncTest");
req.Parameters.Add("Managed", false);
var response = service.Execute(req);
```

In the response are the `AsyncOperationId` and `ExportJobId` parameter values. Use the `AsyncOperationId` in the response to verify the success (`statecode` == 3; `statuscode` == 30) of the asynchronous job. Next, use the `DownloadSolutionExportData` action (or message) with the `ExportJobId` value from the export response to download the exported solution file, which is returned in the `ExportSolutionFile` parameter.

```
// Where 'service' is a pre-configured Organization service instance.
var service = (OrganizationServiceProxy)xsc.CreateOrganizationService();

var req = new OrganizationRequest("DownloadSolutionExportData");
req.Parameters.Add("ExportJobId", Guid.Parse("a9089b53-a1c7-ea11-a813-000d3a14420d");
var response = service.Execute(req);
```

See Also

Sample: Solution staging with asynchronous import

Create patches to simplify solution updates

7/15/2022 • 6 minutes to read • Edit Online

If you add an entity to a solution and export the solution, the entity and all of its related assets are exported in that solution. These assets include attributes, forms, views, relationships, and visualizations, and any other assets that are packaged with the entity. Exporting all objects means that you can unintentionally modify objects on the target deployment, or carry over unintended dependencies.

To address this, you can create and publish solution patches that contain subcomponents of entities rather than publishing the entire entity and all of its assets. The original solution and one or more related patches can be rolled up (merged) at a later time into an updated version of the solution, which then can replace the original solution in the target Microsoft Dataverse organization.

Patches

You can apply patches to either managed or unmanaged solutions and include only changes to entities and related entity assets. Patches do not contain any non-customized system components or relationships that it dependents upon because these components already exist in the deployed-to organization. At some point in your development cycle, you can roll up all the patches into a new solution version to replace the original solution that the patches were created from.

Patches are stored in the Dataverse database as `Solution` entity records. A non-null `ParentSolutionId` attribute indicates that the solution is a patch. Patches can be created and managed through the Organization Service or Web APIs, which are useful for developing automation such as a product install script. However, the Dataverse web application provides various web forms that enable you to interactively create and manage patches.

- Patches can only be created from a parent solution using CloneAsPatchRequest or CloneAsPatch Action.
- The patch parent can't be a patch.
- Patches can only have one parent solution.
- A patch creates a dependency (at the solution level) on its parent solution.
- You can only install a patch if the parent solution is present.
- You can't install a patch unless the unique name and major/minor version number of the parent solution, as identified by `ParentSolutionId`, do not match those of the parent solution installed in the target organization.
- A patch version must have the same major and minor number, but a higher build and release number, than the parent solution version number. The display name can be different.
- If a solution has patches, subsequent patches must have a numerically higher version number than any existing patch for that solution.
- Patches support the same operations as solutions, such as additive update, but not removal. You cannot remove components from a solution using a patch. To remove components from a solution perform an upgrade.
- Patches exported as managed must be imported on top of a managed parent solution. The rule is that patch protection (managed or unmanaged) must match its parent.
- Don't use unmanaged patches for production purposes.

- Patches are only supported in Dataverse organizations of version 8.0 or later.

  The SolutionPackager and PackageDeployer tools in this release support solution patches. Refer to the tool's online help for any command-line options that are related to patches.

Create a patch

Create a patch from an unmanaged solution in an organization by using the CloneAsPatchRequest message or the CloneAsPatch Action, or by using the web application. Once you create the patch, the original solution becomes locked and you can't change or export it as long as there are dependent patches that exist in the organization that identify the solution as the parent solution. Patch versioning is similar to solution versioning and specified in the following format: *major.minor.build.release*. You can't make changes to the existing major or minor solution versions when you create a patch.

Export and import a patch

You can use the Organization Service or Web APIs, the web application, or the Package Deployer tool to export and import a patch. The relevant Organization Service message requests are ImportSolutionRequest and ExportSolutionRequest. The relevant actions For the Web API are ImportSolution Action and ExportSolution Action.

Patching examples

The following table lists the details of a patching example. Note that in this example, the solution and patches are imported in numerical order and are additive, which is consistent with solution import in general.

PATCH NAME	DESCRIPTION
SolutionA, version 1.0 (unmanaged)	Contains entityA with 6 fields.
SolutionA, version 1.0.1.0 (unmanaged)	Contains entityA with 6 fields (3 updated), and adds entityB with 10 fields.
SolutionA, version 1.0.2.0 (unmanaged)	Contains entityC with 10 fields.

The import process is as follows.

1. The developer or customizer first imports the base solution (SolutionA 1.0) into the organization. The result is entityA with 6 fields in the organization.
2. Next, the SolutionA patch 1.0.1.0 is imported. The organization now contains entityA with 6 fields (3 have been updated), plus entityB with 10 fields.
3. Finally, SolutionA patch 1.0.2.0 is imported. The organization now contains entityA with 6 fields (3 updated), entityB with 10 fields, plus entityC with 10 fields.

Another patching example

Let's take a look at another patching example, with the details listed in the following table.

PATCH NAME	DESCRIPTION
SolutionA, version 1.0 (unmanaged, base solution)	Contains the `Account` entity where the length of the account number field is adjusted from 20 to 30 characters.
SolutionB, version 2.0 (unmanaged, different vendor)	Contains the `Account` entity where the length of the account number field is adjusted to 50 characters.

PATCH NAME	DESCRIPTION
SolutionA, version 1.0.1.0 (unmanaged, patch)	Contains an update to the `Account` entity where the length of the account number field is adjusted to 35 characters.

The import process is as follows:

1. The developer or customizer first imports the base solution (SolutionA 1.0) into the organization. The result is an `Account` entity with an account number field of 30 characters.
2. SolutionB is imported. The organization now contains an `Account` entity with an account number field of 50 characters.
3. SolutionA patch 1.0.1.0 is imported. The organization still contains an `Account` entity with an account number field of 50 characters, as applied by SolutionB.
4. SolutionB is uninstalled. The organization now contains an `Account` entity with an account number field of 35 characters, as applied by the SolutionA 1.0.1.0 patch.

Delete a patch

You can delete a patch or base (parent) solution by using DeleteRequest or, for the Web API, use the `HTTP DELETE` method. The delete process is different for a managed or unmanaged solution that has one or more patches existing in the organization.

For an unmanaged solution, you must uninstall all patches to the base solution first, in reverse version order that they were created, before uninstalling the base solution.

For a managed solution, you simply uninstall the base solution. The Dataverse system automatically uninstalls the patches in reverse version order before uninstalling the base solution. You can also just uninstall a single patch.

Update a solution

Updating a solution involves rolling up (merging) all patches to that solution into a new version of the solution. Afterwards, that solution becomes unlocked and can once again be modified (unmanaged solution only) or exported. For a managed solution, no further modifications of the solution are allowed except for creating patches from the newly updated solution. To rollup patches into an unmanaged solution, use CloneAsSolutionRequest or the CloneAsSolution Action. Cloning a solution creates a new version of the unmanaged solution, incorporating all its patches, with a higher *major.minor* version number, the same unique name, and a display name.

For a managed solution things are handled slightly differently. You first clone the unmanaged solution (A), incorporating all of its patches and then exporting it as a managed solution (B). In the target organization that contains the managed version of the (A) solution and its patches, you import managed solution (B) and then execute DeleteAndPromoteRequest or the DeleteAndPromote Action to replace managed solution (A) and its patches with the upgraded managed solution (B) that has a higher version number.

See also

Use segmented solutions

When to edit the customizations file

7/15/2022 • 2 minutes to read • Edit Online

The customizations.xml file that is exported as part of an unmanaged solution can be edited to perform specific customization tasks. After editing the file you can compress the modified file together with the other files exported in the unmanaged solution. You apply the changes by importing that modified unmanaged solution.

Editing a complex XML file like the customizations.xml file is much easier and less prone to errors if you use a program designed to support schema validation. While it is possible to edit this file using a simple text editor like Notepad, this is not recommended unless you are very familiar with editing this file. For more information, see Edit the Customizations file with Schema Validation.

> **IMPORTANT**
> Invalid XML or incorrect definition of solution components can cause errors that will prevent importing a manually edited unmanaged solution.

Supported tasks

You can edit the customization.xml file to perform the following tasks.

Editing the ribbon

This documentation describes the process of editing the ribbon by editing the customization.xml file directly. Several people have created ribbon editors that provide a user interface to make editing the ribbon easier. The most popular one so far is the Ribbon Workbench. For support using this program, contact the program publisher.

For more information about editing the ribbon by editing the customization.xml manually, see Customize commands and the ribbon.

Editing the SiteMap

The SDK describes the process of editing the SiteMap by editing the customization.xml file directly. However, its recommended that you use the site map designer in Microsoft Dataverse to create or update site maps. More information: Create a site map for an app using the site map designer

You can also use one of the community-developed site map editors, such as the XrmToolBox Site Map Editor.

For more information, see Change Application Navigation using the SiteMap

Editing FormXml

FormXml is used to define entity forms and dashboards. The form editor and dashboard designer in the application are the most commonly used tools for this purpose. Editing the customizations.xml file is an alternative method. For more information, see Customize entity formsand Create a Dashboard.

Editing saved queries

Definitions of views for entities are included in the customizations.xml file and may be manually edited. The view editor in the application is the most commonly used tool for this purpose. Editing customizations.xml is an alternative method. For more information, see Customize entity views.

Editing the ISV.config

In earlier versions of Dynamics 365 Dataverse, ISV.Config was the way to add client application extensions as well as some other configuration options. For Microsoft Dynamics CRM 2011 and Microsoft Dynamics 365

Online, the Ribbon provides the way to extend the application. The only remaining capability left in ISV.Config is to customize the appearance of the Service Calendar. For more information, see Service Calendar Appearance Configuration

Unsupported tasks

Defining any other solution components by editing the exported customizations.xml file is not supported. This includes the following:

- Entities
- Attributes
- Entity Relationships
- Entity Messages
- Option Sets
- Web Resources
- Processes (Workflows)
- Plugin Assemblies
- SDK Message Processing steps
- Service Endpoints
- Reports
- Connection Roles
- Article Templates
- Contract Templates
- E-mail Templates
- Mail Merge Templates
- Security Roles
- Field Security Profiles

See also

Customization XML Reference
Customization Solutions File Schema
Ribbon core schema
Ribbon types schema
Ribbon WSS schema
SiteMap schema
Form XML schema
Schema Support for the Customization File