Microsoft

End-to-end case management integration in the utilities industry with Azure services

August 21, 2017, Version 1.0, prepared by Wagner Silveira

Note: To activate the links in this article, please download this PDF file.

Introduction

In this scenario, a client created Contoso Utilities, a new business that provided utilities services across the country. These services included gas line installation, faults maintenance with emergency services, plus preventive and corrective maintenance. Contoso acquired the rights for providing these services from another company. For this reason, Contoso had to transfer the management for existing and new service requests over a short time -- about four months -- from the previous owner.

In this article

What was the business problem?

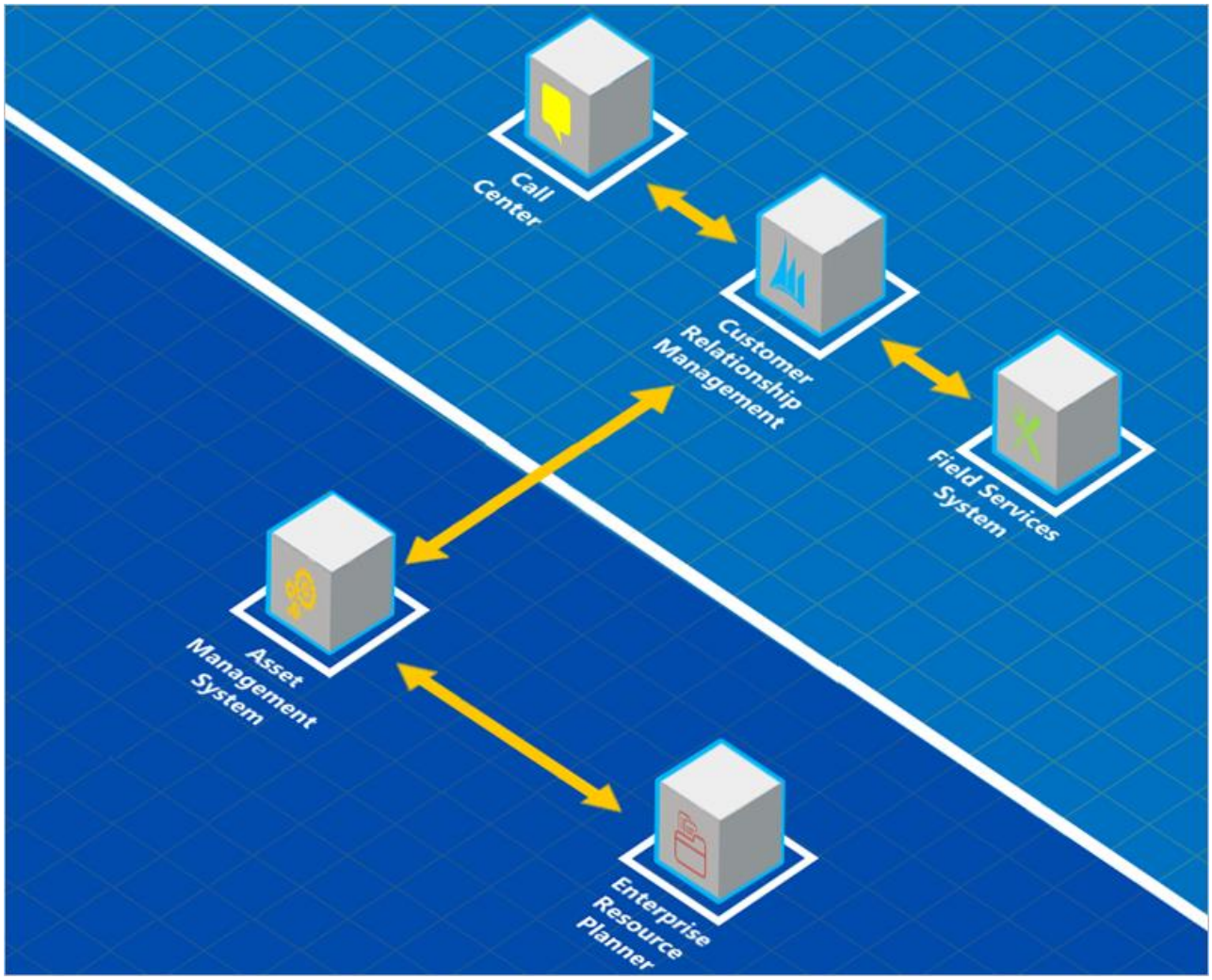
To provide these management services, including the operation, training, and setup for new IT infrastructure along with systems and services, Contoso had to implement and integrate these line-of-business (LOB) systems:

- A Call Center system for initiating work order requests and emergency services on behalf of Contoso Utilities
- A central Customer Relationship Management (CRM) system that served as the central repository for these work orders and emergency services cases. This CRM also needed a self-service portal for creating new work orders and providing information about the latest status for each order.
- A Field Services system that managed field teams, expedited emergency services, and improved team productivity for other work order requests
- An Enterprise Asset Management (EAM) system that managed the state for each asset installed at the client
- An Enterprise Resource Planner (ERP) system that managed the financial aspects for the whole operation

For this setup, Contoso had their ERP and the EAM systems installed on premises, while they provisioned all the other systems outside the company's boundaries, meaning in their partners' datacenters or in the cloud:

- The Call Center and Field Services systems were outsourced to external providers.
- Contoso owned and provisioned the CRM system in the cloud with a Platform as a Server (PaaS) offering.

All these systems needed to integrate with each other, so they could share crucial information for fulfilling work orders and emergency services.



As a new company with a small IT footprint and seeking to minimize investment in hardware and systems maintenance, Contoso based their IT strategy on a cloud-first, PaaS-first approach. Their IT team focused mainly on managing the core infrastructure and services and relied heavily on managed services such as Office 365 for providing day-to-day IT services to end users.

What were the requirements?

Contoso needed a solution that integrated all their LOB systems. With only four months to set up the entire company and define their business processes, Contoso's solution had to quickly adapt to changes. Their partners also required an easy-to-use integration interface that guided them during the development process.

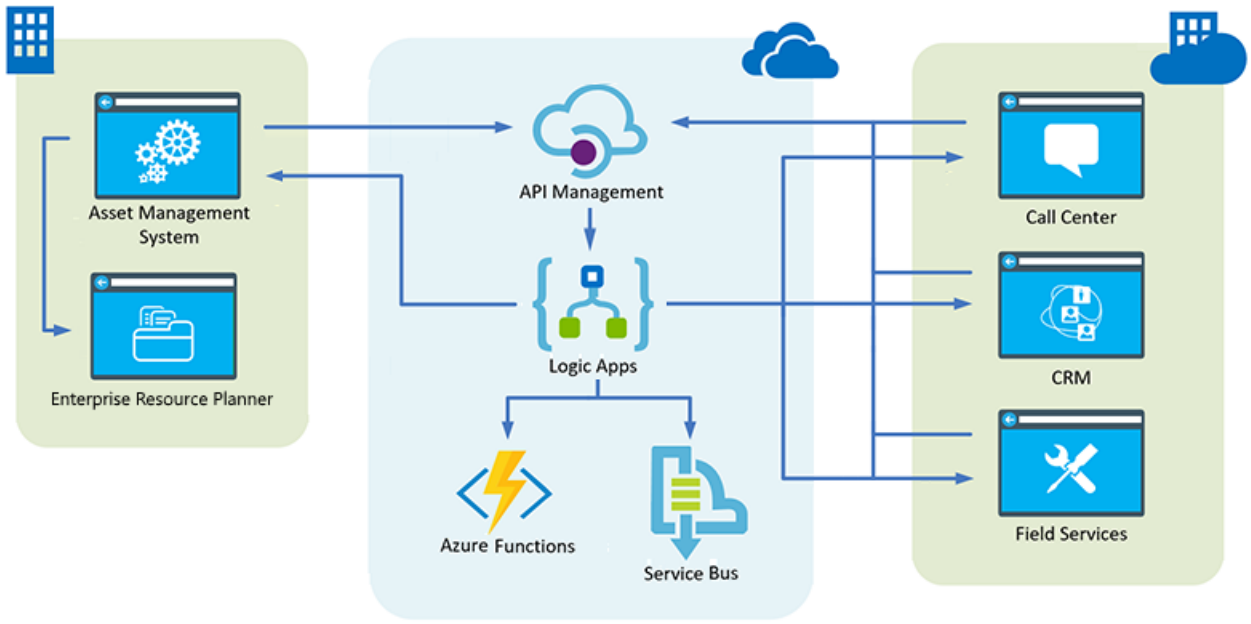
Contoso built their integration platform, which included services and workflows, while the integrated systems implemented their own business processes. This meant that Contoso's partners and internal developers couldn't wait for Contoso to complete their workflows before the end systems started implementing their interfaces.

Contoso's end-to-end integration solution had to support these requirements:

- Integrate seamlessly with on-premises and cloud systems.
- Interface with LOB systems quickly and with as little custom code as possible.
- Adapt easily to changes in requirements.

What was the solution's design and architecture?

Contoso designed their solution using Microsoft Azure ecosystems, based on this high-level conceptual design:



This design combined cloud integration technologies in Azure for a full end-to-end solution, which not only interfaces with on-premises systems on both client and partner networks, but also with PaaS apps hosted in the public cloud. This solution used these Azure services and technologies:

- Azure API Management
- Azure Logic Apps
- Azure Functions
- Azure Service Bus

Azure API Management

Azure API Management provides a platform for organizations to define and publish APIs that internal and external developers can consume. This platform lets API designers focus on defining the best API structure by providing codeless solutions for common requirements like security, throttling, mocking, and API composition. This service played a crucial role in Contoso's solution by providing a common integration point for all connected systems, including:

- A familiar and secure communication protocol: TLS over HTTP
- A common service framework: REST based services API
- A simple but efficient security mechanism: Shared Access Signatures (SAS) tokens

API Management sped up development by letting all teams start implementing their own pieces of the full, complex, end-to-end integration puzzle. To meet this goal, the teams published the agreed-on operation contracts as soon as they were finalized.

From the start, published APIs had full security functionality. Usually, teams leave this aspect of integration until the last minute, which can cause headaches. API Management provides responses from the ground up. The initially mocked responses let the client systems effectively integrate with APIs before the backend services were fleshed out. Live responses replaced mocked responses when the backend workflows, implemented as logic apps, became available. And because Contoso defined the migration of mocked responses to live responses at the configuration level, rather than in code, they enabled live services quickly and transparently to client endpoints.

Azure Logic Apps

The Azure Logic Apps service simplifies automation for business processes and connections to systems both in the cloud and on premises. You can orchestrate calls to system APIs through many out-of-the-box connectors and other Azure resources. Logic Apps also lets you use your own APIs and legacy web services, so that you can use your existing services and API ecosystem.

At the core of Contoso's integration solution, Logic Apps mapped integration requirements from evolving business processes to workflows. During workflow design, Contoso identified common patterns and translated them, as much as possible, into reusable logic apps. These apps created simpler artifacts that Contoso could compose into more complex workflows. Here are some key Logic Apps features that sped up and simplified the implementation process:

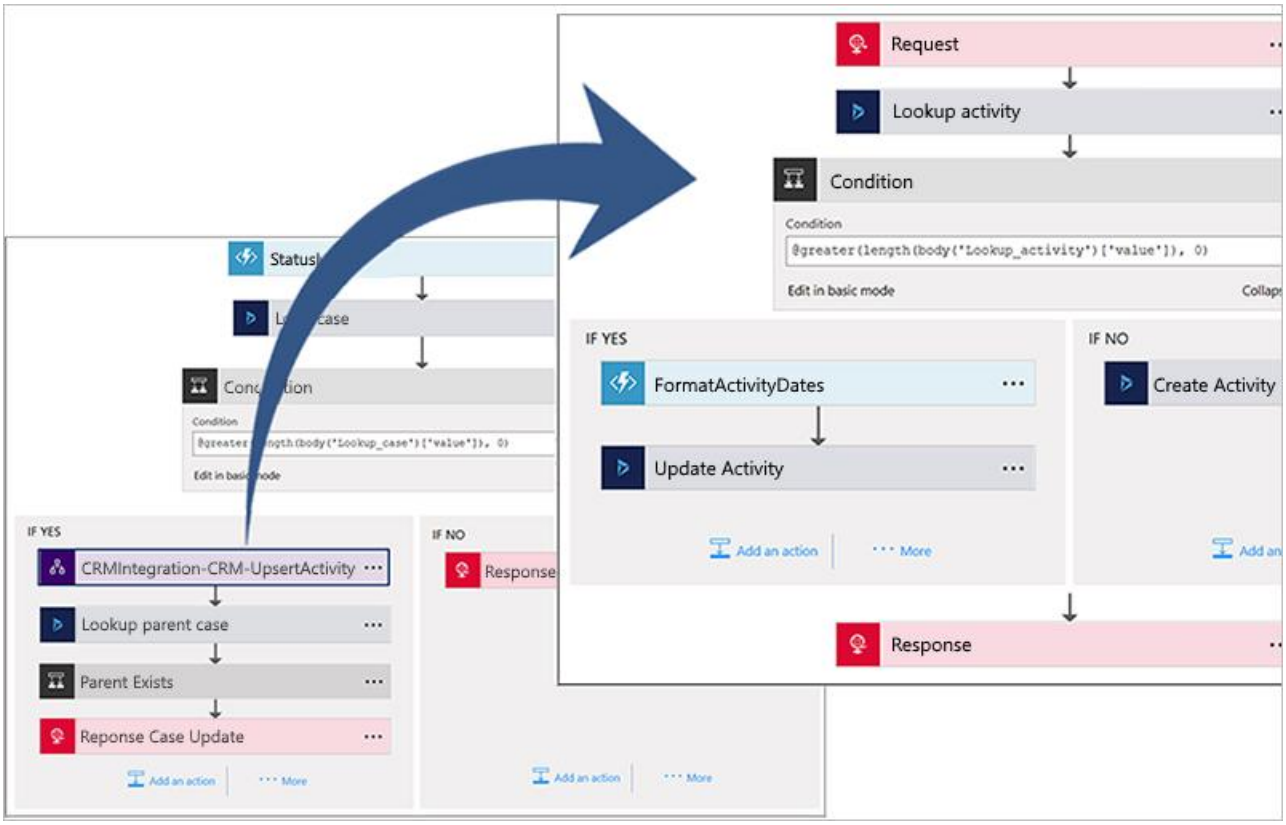
- **Security and connectivity**: API Connection abstracts connectivity to the PaaS solution into its own component, which you can reuse in any action that the managed API exposes. This simplifies tasks such as authentication, token management, and deployment.
- **Operations and entity discovery**: In most cases, API Management presents a set of available operations, and where required, also discovers the entities and associated properties that can be used by that operation. The CRM system was a classic where any entity exposed by the CRM could execute each operation. The capability to discover entities available for each operation, and each property sped up the integration with that system.
- **Common interface**: After mastering one action in a managed API, Contoso found that other actions in the same API usually followed the same patterns and principles, which simplified workflow development.

Managed APIs as accelerators

Azure Logic Apps provides APIs managed by Microsoft that you can use as out-of-the-box adapters, or connectors. These APIs expose operations for the most common PaaS solutions and were important for Contoso's required agile implementation approach. These APIs sped up integration with partner systems by simplifying the most time-consuming aspects of integration:

Logic apps as actions

Azure Logic Apps has the capability to execute a workflow as an action inside another workflow. So, you can use logic apps to break down large and long business processes into smaller, reusable components that you can compose when you implement bigger scenarios. You can reuse any workflow that's implemented with the request-response pattern as an action. A logic app workflow implements the request-response pattern by using a request type trigger, finalized with a response action. That way, you can implement and test smaller pieces of integration in parallel, speeding up development and simplifying the main workflows.



Logic apps as "execution pipelines"

BizTalk has been the defacto Microsoft Enterprise tool for over ten years now and has the concept of three major "execution pipelines" that play specific roles in an integration workflow:

- Receive ports
- Orchestrations
- Send ports

In Logic Apps, these concepts are less rigid and might lead to monolithic workflows where inbound logic, workflow processing, and outbound logic are all tied into a single logic app. This flexibility might be welcome in simple projects, but might lead to duplicate effort with large projects because each logic app could recreate common processes that you could otherwise reuse, leading to more development and testing effort than necessary.

As an alternative to the monolithic approach, you could identify the inbound, outbound, and workflow processing functionality and create different logic apps for each purpose. Contoso used these reusable components in the main workflow logic by using logic apps as actions and with Service Bus Topics to create true publisher-subscriber scenarios. For more information about publisher-subscriber scenarios with Service Bus, see the sections below.

Azure Functions

Extending the Azure platform, Azure Functions provides on-demand computing and can run code snippets when triggered by events in other systems, Azure services, Software-as-a-Service (SaaS) products, or on-premises systems. Azure Functions supports first-class integration with Azure Logic Apps and crucially extends that platform by implementing services that aren't available out of the box. The following sections describe patterns that Contoso implemented with Azure Functions.

Cached Lookup

In this scenario, each source system provides fields, which are represented as name-value pairs, as either only names or only values for some messages. The integration workflow needs to translate these fields into the correct values. Here are examples where this scenario happened in this solution:

- The source system provides only numeric values, which aren't human readable, but the workflow must show specific fields to the end user.
- The source system provides either numeric or character values, but the workflow must map those values to related values in another system.

To bridge this gap, Contoso created an Azure function that created cached versions of the lookup entities and mapping matrix where required. Because you can dissociate deployment for Azure functions and logic apps, you can update mapping or lookup changes without redeploying logic apps, minimizing the risks in updating those maps.

Transforming JSON to JSON and JSON to XML

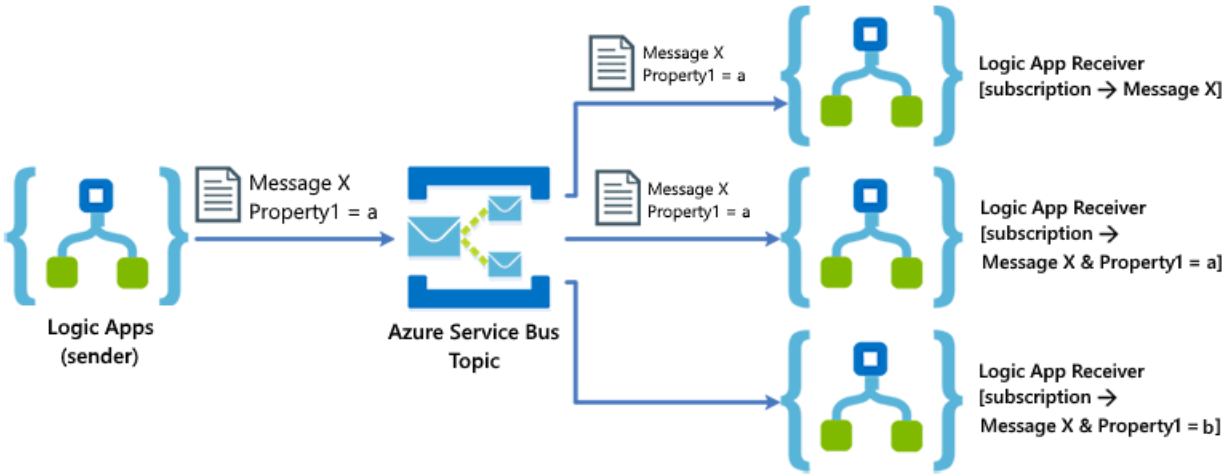
When Contoso needed complex JSON-JSON or JSON-XML transforms, for example, in aggregations or complex mapping logic, they implemented Azure functions that made the transformation process simpler and more cost effective. In some cases, they used an Azure function to replace large numbers of logic app actions. This approach worked very effectively in workflows that bridged JSON-based REST services to classic XML-based, SOAP services.

Azure Service Bus Topics

The Azure Service Bus service provides messaging technologies that let on-premises and cloud systems exchange messages by using reliable messaging queueing and durable publisher-subscriber messaging. Service Bus Topics complemented the capabilities in Logic Apps by providing these important aspects for end-to-end integration:

- A decoupled mechanism for exchanging messages between systems so that systems could go down for some time without losing those messages
- A publisher-subscriber mechanism that lets multiple systems consume copies of the same message, which streamlines integration workflow.

Contoso heavily used the second pattern in their solution. Their CRM system managed service requests for maintenance and emergency faults. While the Enterprise Asset Management system cared about both message types in any state, the Field Services and Call Center systems cared only about fault messages, and sometimes, only specific states. A Service Bus Topic subscription lets these systems evaluate a single message published by CRM and added that message to each subscription queue based on a series of properties. The logic apps that interfaced with each LOB system then monitored that specific subscription and process messages accordingly.



What was the strategy and its challenges?

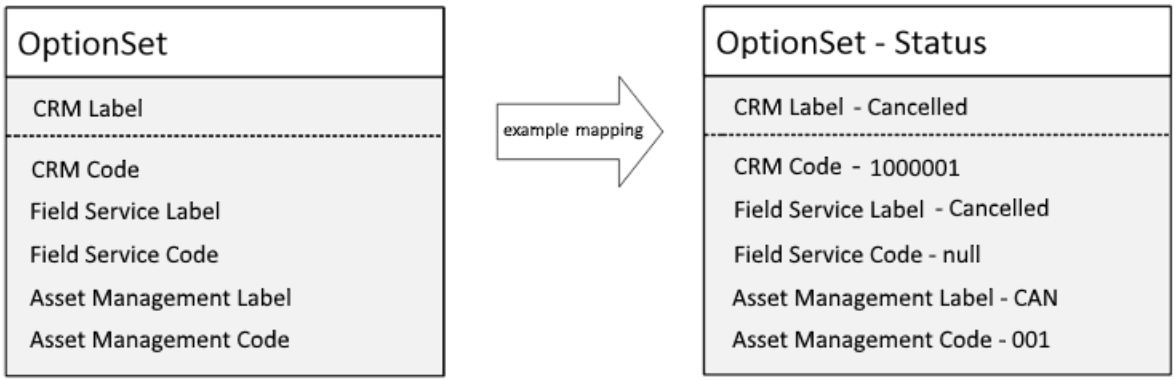
Like most integration projects, developing this solution had challenges. Some challenges were purely technical - Contoso just needed to identify known patterns and apply them to the new technology stack. Other challenges reflected the maturity level around some aspects of the technology.

Create a lookup mapping between systems

The CRM system used *option sets*, which are sets of business-defined name-value pairs, as a data type. Option sets improve your business's records quality by restricting the data that fields can have. But from a logic apps integration perspective, this capability might create problems for these reasons:

- CRM actions, like queries, inserts, and updates, return only the value associated with the option set, not the name-value pair. If users had to view the record, this task required translating the value to human-readable format.
- Usually, destination systems have lookup values that differ from the values in CRM and require translation.

So, the challenge here lay in creating a generic method that translated the required information across multiple systems in a processor or cost effective way. Contoso created an Azure function that held a collection of matrix objects matching the values for each option set in CRM, Field Services, and Enterprise Asset Management. Here's an example that shows the matrix structure:



The Azure function receives these inputs:

- The OptionSet category that needed translation, for example, "Status"
- The SourceType, such as "CRM Label", "CRM Code", and so on
- The actual value, for example "Cancelled"

With these inputs, the function finds the object, in the appropriate OptionSet collection, where the SourceType value matches the value that's passed as a parameter, for example, "Status.CRMLabel == Cancelled". The function then returns the object, so the logic app could execute any required translation. Or, the function could pass an object from a source system like CRM Record, to a specific end system like the EAM system, but at the same time, translate any option sets for that element. The alternative advantage lets discrete functions create translations that the CRM logic app's default actions don't provide, and separate maintenance for option sets from logic apps maintenance.

During the initial implementation and due to time constraints, Contoso implemented option set lists as JSON arrays stored in a single CS Script file. Ideally, Contoso would implement the JSON arrays as a file in a storage account and as function inputs. The OptionSet list used a singleton pattern, so only one list instance was instantiated.

Handle transient errors and exceptions

Contoso also had the challenge in defining a good exception management and retry policy for a new technology without much guidance. Logic app actions have a built-in retry policy: four retries at 20 seconds apart. This policy is useful in some cases and lets logic apps overcome transient errors. But this policy can also create

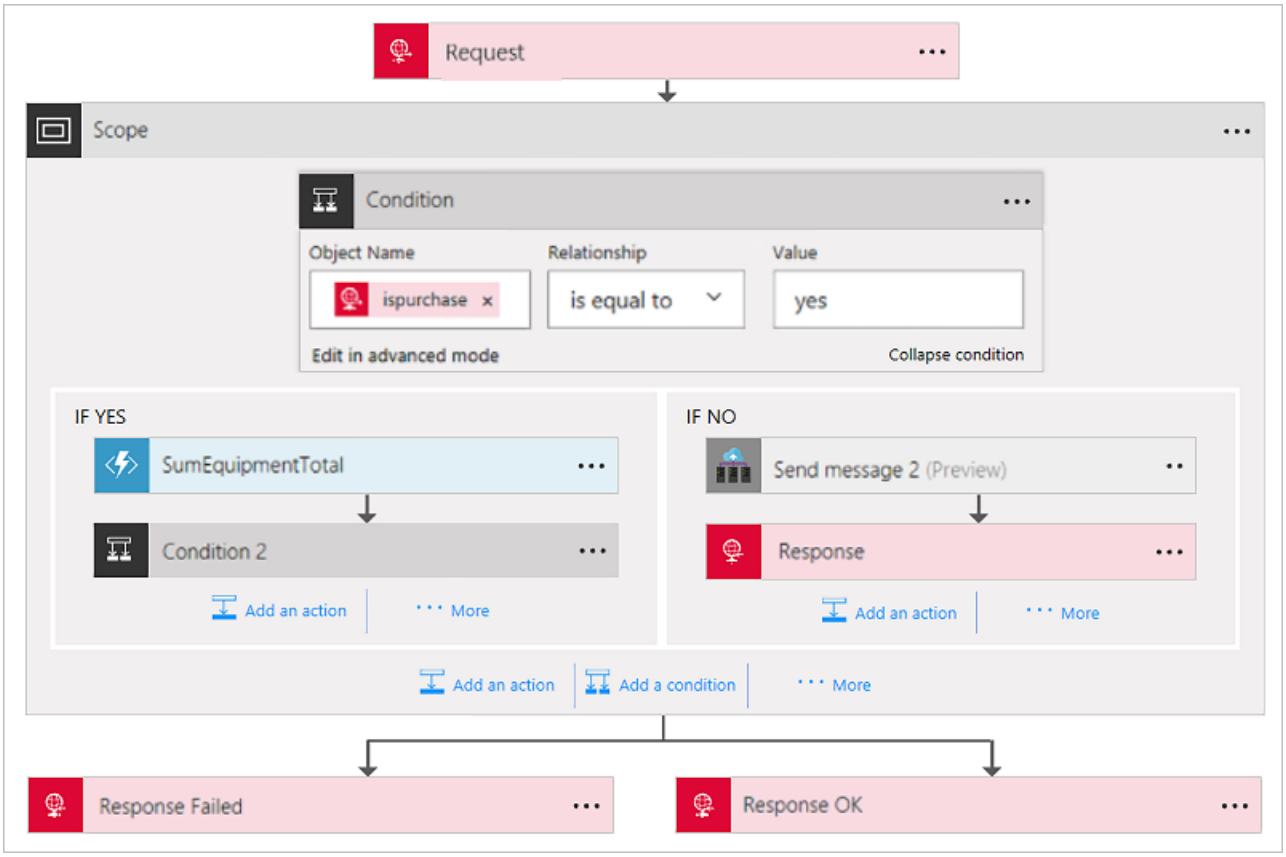
problems in request-response synchronous workflows where a response must return within a specific time. You can't define try-catch blocks through the visual Logic Apps Designer, so you must edit the logic app's JSON definition in Logic App Code View to handle requirements for managing exceptions. To minimize errors, Contoso applied the same patterns, described below, to all logic apps they implemented from scratch.

Create exception management with scopes and conditional execution

Always put a logic app's main workflow inside a *scope*, which gets its own status after the inner workflow runs. That way, you can evaluate the scope's result and define two actions:

- An action that runs when the scope succeeds, which returns a successful response
- An action that runs when the scope fails, which returns a failure response

Here's an example that shows the scope pattern:

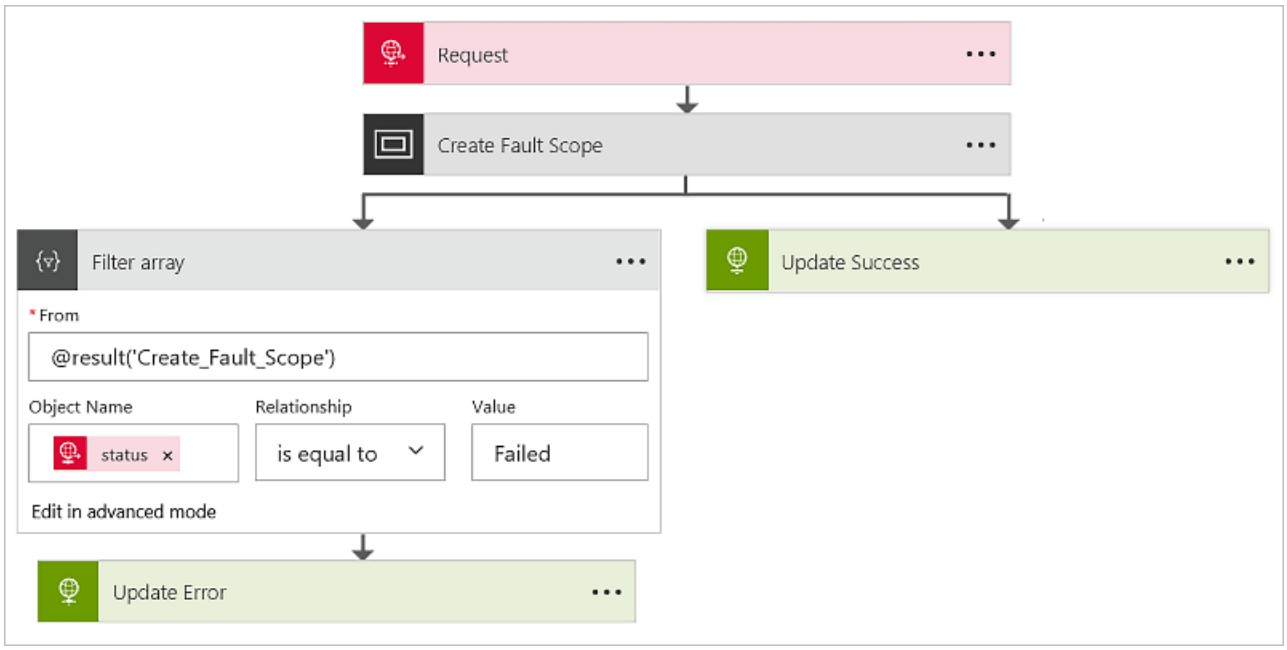


Here, the drawback is that a failed scope provides a list of action results, not an "exception message". So, when Contoso needed more exception information, they used a different technique, described below.

Note: By default, logic app actions run after previous actions successfully finish. To change this behavior, update the **`runAfter`** parameter in the logic app's JSON definition through Logic App Code View. For more information about the **`runAfter`** parameter, see Handle errors and exceptions in Azure Logic Apps.

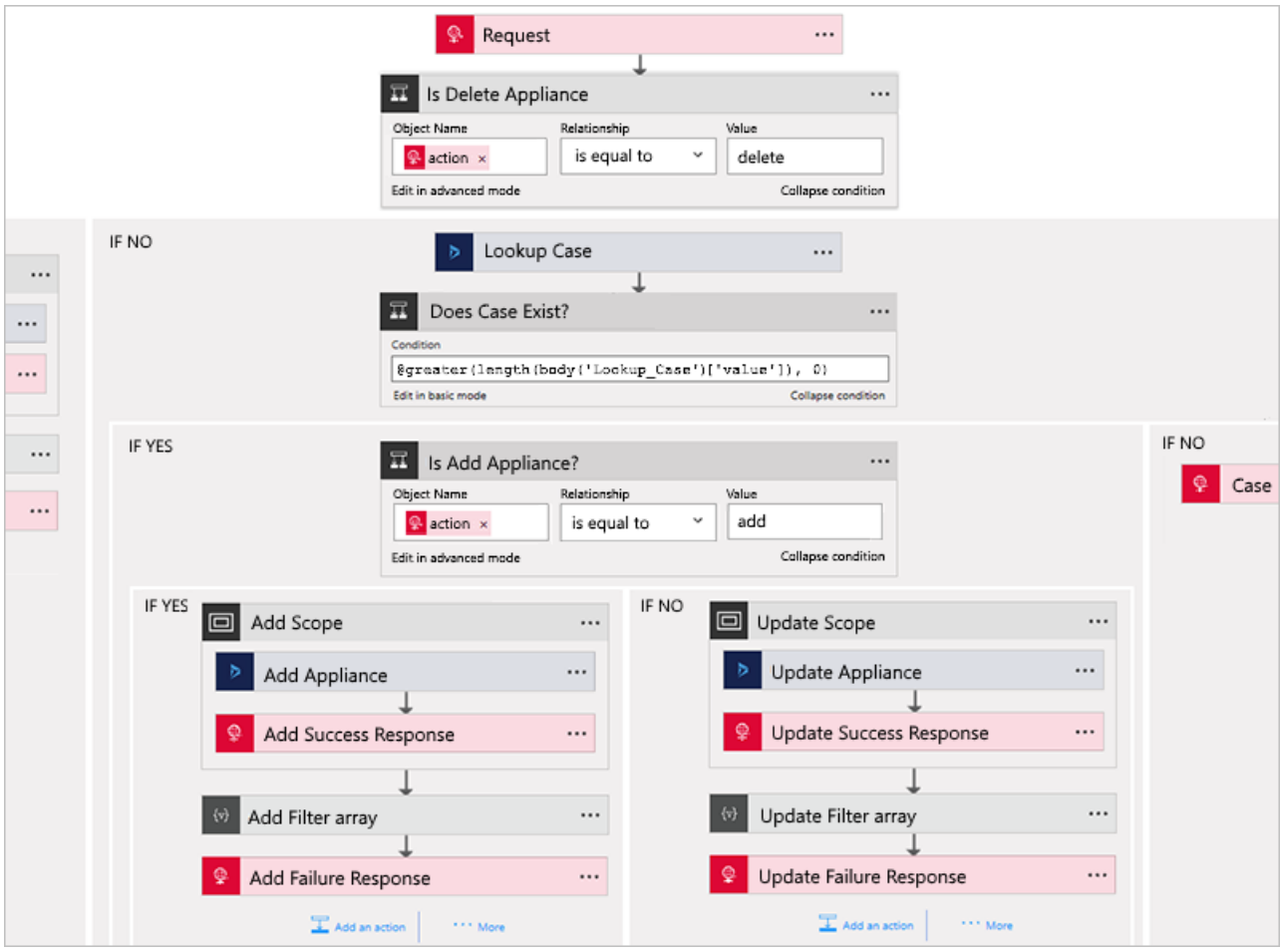
Filter the scope output

To get more data about the causes behind errors, Contoso added a step in the "failed branch." This step filtered the scope's output for a list of failed results and passed that data along with the failed result. This example shows this filtering technique where you can use a filter array to implement this pattern:



Individual scope actions for fine-grained responses or compensation

Sometimes, Contoso needed refined responses or compensation actions for subsets of logic app actions. So, they used a variation of the exception management technique. They created individual scopes where failed blocks either compensated for exceptions or returned failed messages. This example shows how each branch needs a scope to run:



Avoid timeouts on errors

Previously, we mentioned that the Logic Apps automatic retry feature can prove very useful in some scenarios, but might also cause problems in synchronous workflows. When run over an extended amount of time, retries can cause these problems:

- **False negatives**: When retries run for a long time, they might cause the initiating system to time out, close the communication channel, and report even a successful workflow execution as an "unknown failure due to timeout."
- **Lack of information about exceptions**: When retries run due to a valid exception, such as data or logic errors, and are run for a long time, these errors won't be reported back to the initiating system. This scenario creates situations where the poisoned message is repeatedly sent.

To avoid these situations, make sure to analyze logic apps that implement synchronous workflows and apply one of the techniques in this section.

Minimize retries

When you remove unnecessary retries from your logic apps, you can avoid timeouts that might result from extended retry periods. Try to identify reliable services that can have a small number of retries or no retries at all. For Contoso's project, CRM Online was a highly available PaaS system with the biggest impact in the workflows, due to the number of calls required. Contoso made the design decision to remove retries from calls to CRM Online so they could minimize timeouts. Usually, errors that came from those calls were real exceptions, rather than transient errors.

Contoso associated a retry policy with each action in a logic app and used this format:

```
"retryPolicy": {
    "type": "<type-of-retry-policy>",
    "interval": <retry-interval>,
    "count": <number-of-retry-attempts>
}
```

To avoid any retries, you can implement this retry policy instead:

```
"retryPolicy": {
    "type": "None"
}
```

Note: For more information about retry policies, see Workflow actions and triggers.

Handle large workflows

In some cases, even after you remove unnecessary retries, a workflow might just too big for the default timeout to handle. This scenario was especially true for some CRM integration workflows, due to the number of entities that needed creating, updating, or looking up. So, Contoso applied a mix of these patterns:

- **Increase the timeout in initiating systems**: For some cases, this solution was simple and pragmatic enough, especially when the initiating system was running a batch, overnight jobs, or scheduled jobs.
- **Change the synchronous call to fire and forget**: This pattern was particularly useful for scheduled or overnight jobs that couldn't handle exceptions anyway when sent.
- **Change the synchronous call to an asynchronous two-way call or webhook**: This pattern completely removes the time dependency between each system. This option is preferred when you can control the integration endpoints on the LOB system side.

Deployment and source control

Contoso found a new challenge when developing a cloud integration solution that used multiple technologies, a PaaS model, and configured each solution in a portal-like environment. Specifically, they needed to have all items in one repository for source control and deployment.

Source control

From source control perspective, Azure services vary in the capability for associating generated artifacts with a source control repository. In this area, Azure Functions seems the most advanced and can associate an Azure function app with a Git repository. Developers can perform all the usual repository actions such as fetch, commit, pull, and branch code. Azure Functions also supports Continuous Integration (CI) by letting you associate an Azure function app with a repository branch. So, any changes pushed to that repository are automatically published to Azure functions. With the correct branching strategy, you can make sure that Azure Functions code moves through development, test, and production by committing code to the correct branch.

API Management lets you store its policies in a Git repository. But unlike Azure Functions, API Management has no CI capability. Also, sometimes policy configuration depends on environment-specific configuration. Although both Azure Functions and API Management partially integrate with Git repositories, their actual definitions are not stored in those repositories. So, to deploy the same function in another environment, some scripting is required. Here, you can use Azure Resource Manager templates, which define Azure resource configurations. These templates use parameters and variables to create templates that you can deploy to an Azure subscription. You can configure parameter files for each environment, so that you can use the same template to deploy in multiple environments and maintain consistency between deployed environments.

To get all the technologies into one source control repository, Contoso exported Resource Manager templates from the Azure portal, adjusted the templates for any parameterization not generated during export, and stored the templates in a source repository. For the last two steps, you can use the Resource Manager Template project in Visual Studio to create and parameterize Resource Manager templates. You can then save those templates to your source repository, thanks to Visual Studio's out-of-the-box integration with all mainstream source repositories.

Logic Apps and source control

For source control, Logic Apps created a special challenge compared to the other technologies. In Azure Functions, API Management, and Service Bus, the boundary is clearly defined between resource provisioning and code, or configuration in the case of API Management. But, for Logic Apps, the resource definition is the code. So, during the project development phases, Contoso needed a good strategy for maintaining logic app definitions and making sure to correctly capture definition changes in source control. Unfortunately, at the time, Visual Studio tooling for Logic Apps was still under development. To enable source control for Logic Apps, Contoso adopted these patterns:

- **Use the "nested" templates capability in Resource Manager templates**. Each logic app was a self-contained Resource Manager template, which let Contoso trace changes to a specific Resource Manager template file, and not a monolithic file, in the project.
- **As much as possible, parameterize information that might change across environments.** This pattern simplified creating the final Resource Manager template and storing the template in source control. For example, an HTTP address or the credentials for one or more actions had its own entry in a logic app's parameters section and was referenced from that section in the logic app. So, when Contoso generalized the template in Visual Studio, they only had to change that information in a one place, not multiple places.
- **Identify changes between logic app versions with a diff tool.** Contoso found this technique useful when they "reapplied" the previously described generalization approach.

Deployment

Contoso found that deploying to other environments posed these challenges:

- Catering for initial provisioning and code updates.
- Creating one process to deploy all technologies, correctly synchronized, and avoid deploying breaking changes.

Initial provisioning x code updates

Redeploying the entire stack every time isn't practical for these reasons:

- Some Azure technologies, such as Service Bus and Functions, need a unique address for deployment. Monolithic deployment required first deleting those items.
- The system required 24/7 uptime because faults could happen any time. Bringing down the whole system was impractical, especially for Service Bus, which Contoso could use as a message cache when other parts of the system were down.
- Reapplying the entire deployment required more regression and smoke testing.

So, Contoso performed initial provisioning with a Resource Manager template, which they can use each time they had to create or recreate an environment, for example, after a catastrophic failure. To execute the template, they can use either a scripted procedure, for example, in PowerShell, which has good support for executing Resource Manager templates, or include the template in another Resource Manager template when deploying the full solution.

Synchronizing deployments

Synchronizing technologies for patches or partial deployments in an environment created another challenge. Contoso couldn't follow only one process because each technology dealt differently with initial resource provisioning in Azure and with code, or configuration, changes. With time and technology constraints, Contoso could only combine as many pieces of deployment scripts as possible, along with Resource Manager templates, PowerShell, and manual steps. For example, a typical bug update might include these tasks:

- Create a script to deploy each logic app affected by the bug and as Resource Manager templates.
- Manually push Azure function changes to the appropriate branch.

Deploying updates to logic app definitions doesn't change or redefine their trigger URLs, so Contoso didn't need to make changes in API Management. In fact, API Management changed only when Contoso required new operations, so usually, Contoso manually made these changes. In this project, Service Bus rarely changed and could define subscriptions and filters in Resource Manager templates. That way, Service Bus changes followed the same principles applied to logic apps.

With improving Visual Studio support for Logic Apps and Azure Functions, it's worth exploring a solution that relies on Visual Studio, or even better, builds in Visual Studio Team Services for deploying changes to an environment, and uses alternatives such as Azure Automation for provisioning changes in API Management configuration. If Contoso still needed a manual process, they can use the logic app import wizards for API Management.

Conclusion and lessons learned

Contoso used the Azure cloud integration stack extremely successfully for this project, mainly due to the Azure's ability to cope with changes, and the pace at which Contoso could provision and implement their solution. That said, Contoso found many areas for improvement, especially around the development lifecycle and continuous integration.

Here are some lessons learned for future projects:

- **Make sure that each logic app has its own Resource Manager template** in the Resource Manager template project, especially if you don't use Visual Studio for template development. That way, you can more easily compare and perform partial deployments when necessary.
- **Create different Functions branches for testing and production**. Make sure that you push only into production in your deployment script. This pattern avoids scenarios where logic apps and Azure functions get out of sync.
- **Invest time and resources in an automated deployment process** when possible.