SERVERLESS360

Logic App Patterns and Best Practices

Steef-Jan Wiggers, Microsoft Azure MVP

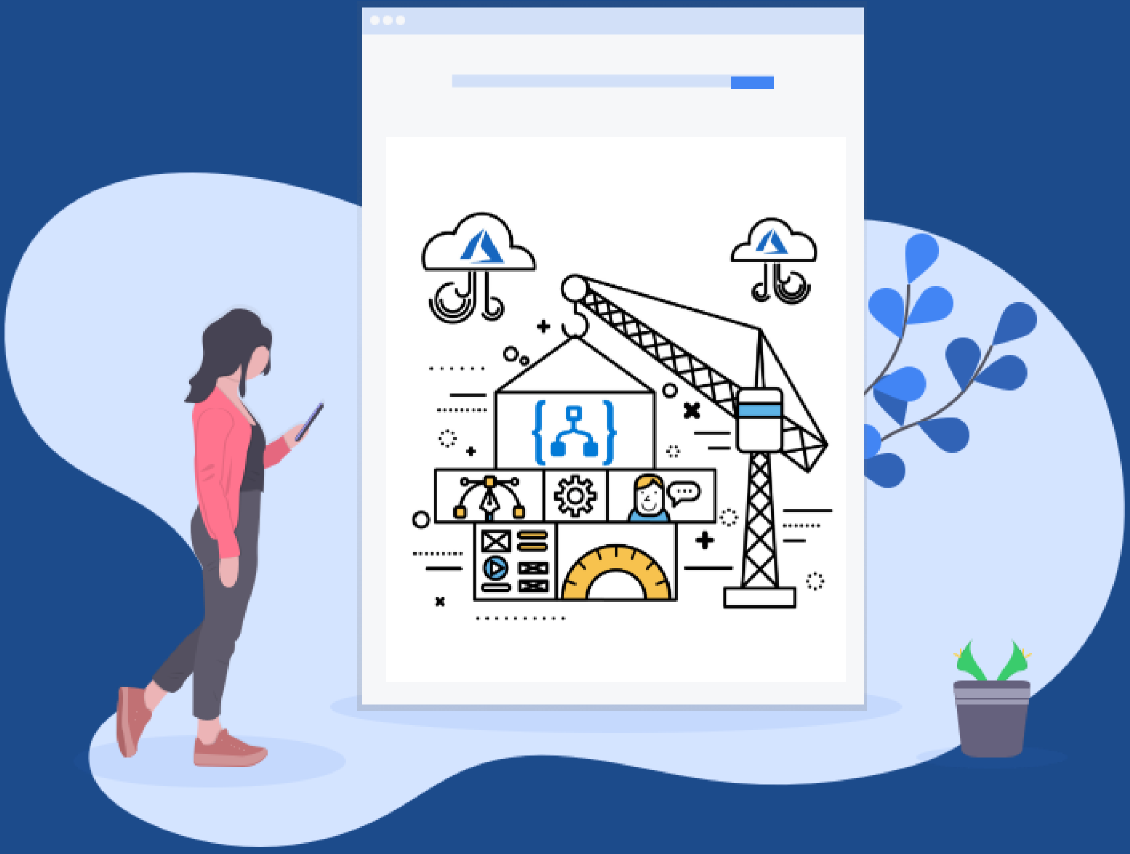


Table of Contents

1. Introduction 03
2. About the Author 04
3. Logic Apps 05
4. Messaging Patterns 11
5. Messaging Communication Patterns 17
6. Messaging Handling Patterns 23
7. Best Practices 28
8. Summary 34

Introduction

Logic Apps was released in July 2016 for the public and has since then evolved into a leader in the integration Platform as a Service (iPaaS) space in two years. Moreover, Logic Apps succeeded Microsoft's first attempt at offering an iPaaS in the Cloud – BizTalk Services. It later received some updates and branded as Microsoft Azure BizTalk Service, MABS for short – a version 2 of BizTalk Services. However, MABS was deprecated, and both the VETER pipelines and the EDI/B2B functionality Microsoft moved to Logic Apps.

In this paper, we will discuss some of the patterns you can implement with Logic Apps. Furthermore, we present some of the best practices when developing Logic App workflows.

About the Author

Steef-Jan Wiggers

Steef-Jan Wiggers is Azure Technology Consultant at Codit, is all in on Microsoft Azure, IoT, Integration, and Data Science. He has almost 20 years' experience in a wide variety of scenarios such as custom .NET solution development, overseeing large enterprise integrations, designing and building API's and cloud solutions, managing projects, experimenting with data, SQL Server database administration, and consulting. Steef-Jan loves challenges in the Microsoft playing field combining it with his domain knowledge in energy, utility, banking, insurance, healthcare, agriculture, (local) government, bio-sciences, retail, travel, and logistics. He is very active in the community as a blogger, book author, InfoQ editor, and global public speaker. For these efforts, Microsoft has recognized him as a Microsoft MVP for the past nine years. Steef-Jan's can be found on twitter at @SteefJan.

Logic Apps

The successor to MABS is Logic Apps - based on a different model. First of all, it is Platform as a Service offering, where you do not have to worry about managing any infrastructure - unlike MABS you do not have to choose any SKU's. The infrastructure is abstracted away, and you can build a flow using triggers and actions by creating a workflow definition. Furthermore, Logic Apps are interoperable with other Azure services like Service Bus, Azure Functions, Event Grid, and API Management. Besides these services Logic Apps also offers various connectors for other Azure Services and SaaS solutions. In case a connector doesn't exist you can build one yourself.

With Logic Apps you can focus more on the business requirement and deliver value faster as you do not have to deal with any infrastructure. Furthermore, you can quickly build mashups of data, connect services and systems in the cloud together. The connectors provide connectivity with other Azure services, SaaS solutions, and the Microsoft Cloud. Moreover, Microsoft targets Logic Apps for enterprise integration in the cloud. However, it is also suitable for hybrid scenarios when leveraging the on-premise data gateway or service bus queue and topics.

iPaaS

You can view Logic Apps as a true iPaaS offering as it adheres to the definition given by Wikipedia:

- Deployed on multi-tenant, elastic cloud infrastructure – the Microsoft Azure platform.
- Subscription model pricing (operating expense, not a capital expenditure) - provided by the consumption model.
- No software development (required connectors should already be available) – in case of Logic Apps these are the managed out-of-the-box connectors.
- Users do not perform deployment or manage the platform itself – managed by Microsoft.
- Presence of integration management & monitoring features – the offering of Operation Management Suite (OMS) and third-party products like Serverless360.

Logic App is an iPaaS offering by Microsoft and is an enterprise-grade service, which within two after its general availability became a leader in Enterprise Integration Platform as a Service (eiPaaS) by Gartner in April 2018. See https://aka.ms/eipaasmq for the full reprint Gartner report.

Workflow definition

Developers can build a Logic App, i.e. design a workflow using the designer in the browser or Visual Studio – leveraging the Azure Logic Apps Visual Studio Tools.

You can view a Logic App itself as a logical container or host of a workflow definition – when running the flow it will consume resources on the Azure Platform (resources are abstracted away from you). With the designer, you can define a business process workflow, or integration workflow by choosing one of the many predefined templates or an empty one. The predefined templates offer something like:

- 'Peek-lock receive a Service Bus message and complete it'
- 'Peek-lock receives a Service Bus message with exception handling'.

Beside the pre-built template, you can start with an empty template and commence dragging a trigger and actions yourself to the canvas – each trigger, action, condition, and so you place in the Visual Designer are captured as JSON. Furthermore, in the code-behind, you can examine the JSON.

Workflow features

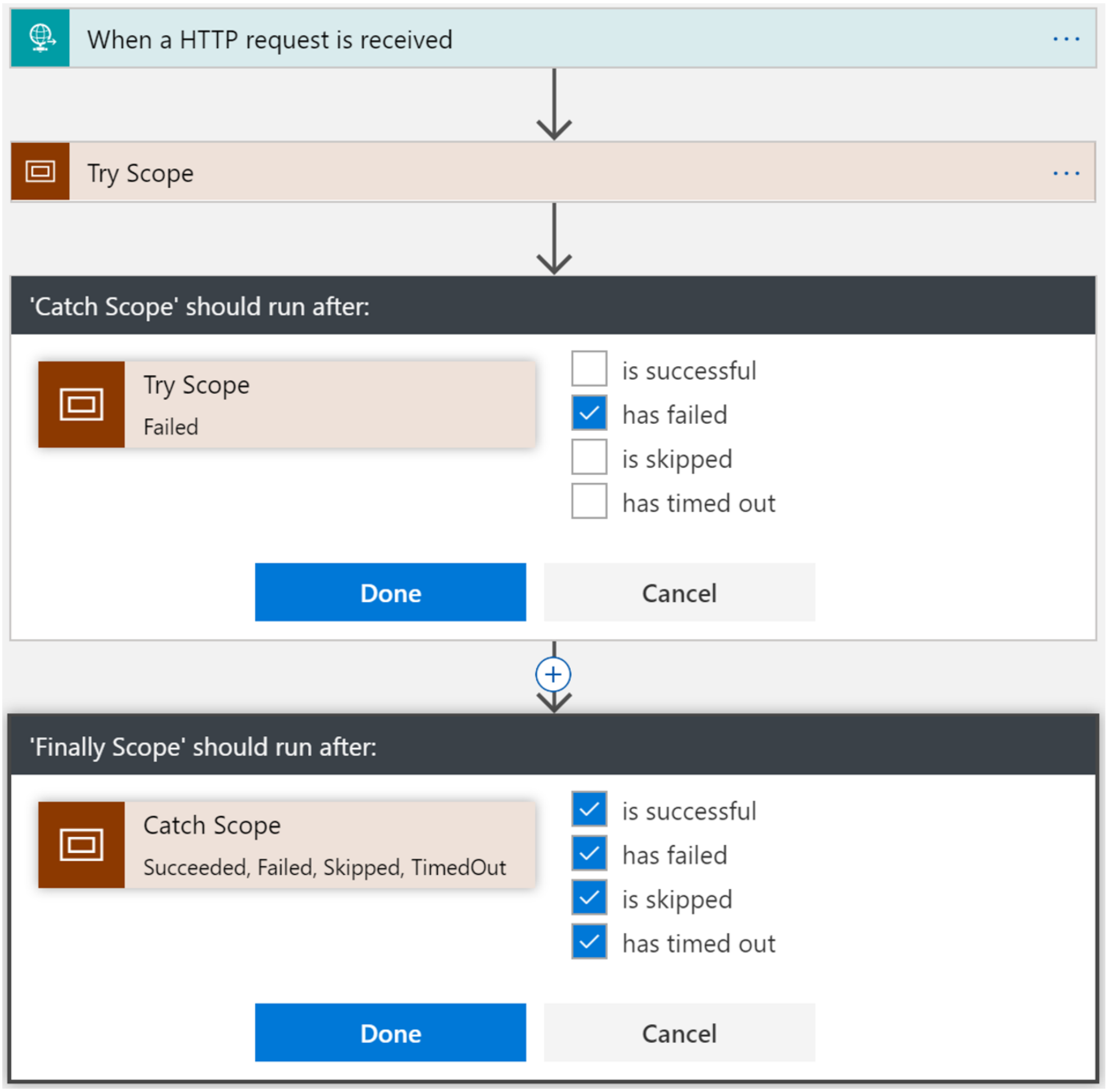
In the previous paragraphs, we discussed what Logic Apps are and how you start building a workflow definition. The connectors, actions, conditions, and expression available in the designer can provide you with the means to construct many types of workflows. With these workflows, you can leverage some of the features within the workflow to control the behavior such as:

- **Retry policy** - either set by default, custom or disabled
- **Run after** - a conditional dependency control through status setting (Succeeded, Failed, TimedOut, Skipped)
- **Limit** - limit the duration of execution
- **Terminate** - early termination by failed or successful status
- **Scopes** – encapsulate set of actions

These features can aid in applying error handling in a workflow, have concurrency control, schedule executions, and perform run once jobs. In the following paragraphs, we will discuss some of them.

Try-Catch-Finally

With any application, service or workflow, you build error handling. It is a crucial part to enhance robustness. In a Logic App, you can apply some of the behavior control features like scopes, retry policies, and the Run after. You can use scopes to group actions together, and you can act upon the success or failures of actions within the scope using the **run after**. For instance, you can implement a try-catch-finally in Logic App, similar to what you can do with C#. In the 'Try Scope' you place the actions, the 'Catch' is the run after the 'Try Scope', and 'Finally' is a run after of the 'Try Scope' (see picture below).



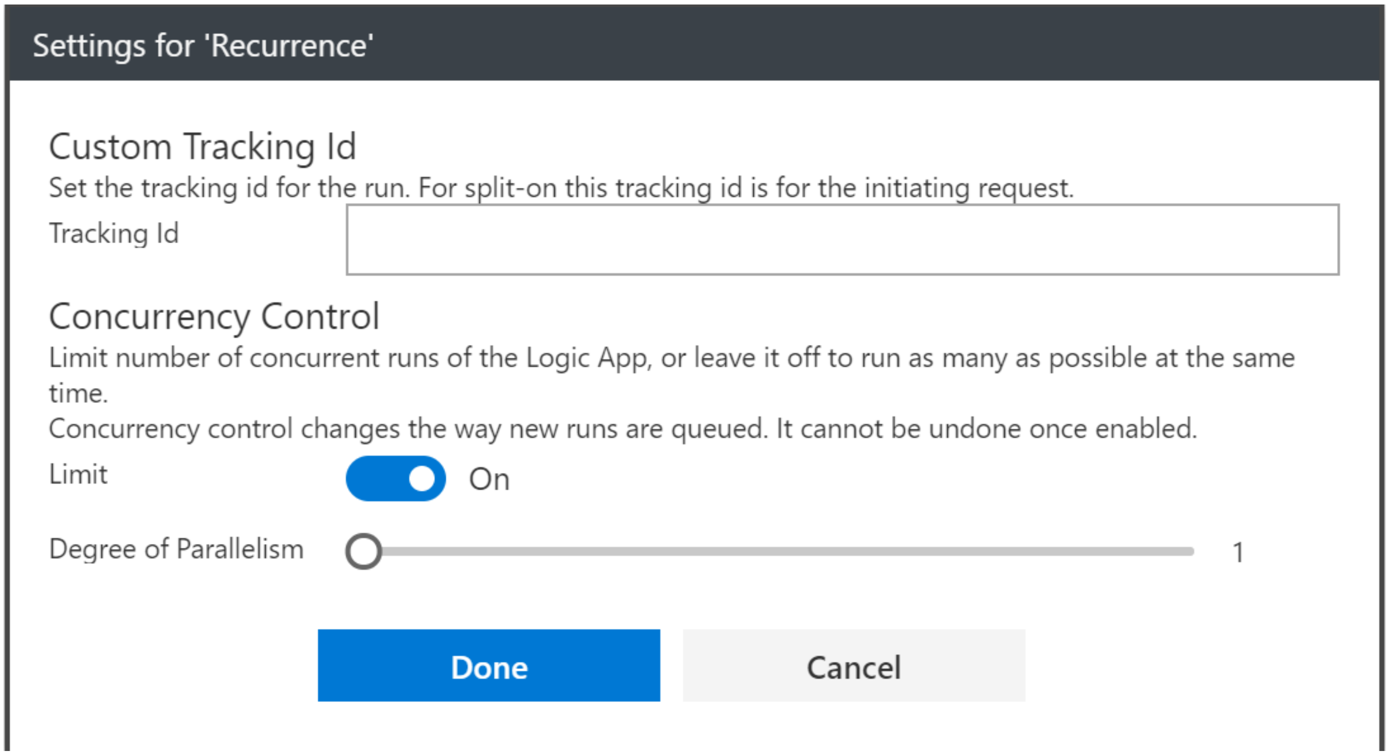
More details about error handling in Logic Apps see Handle errors and exceptions in Logic Apps.

Concurrency Control

One of the critical aspects you need to consider with any Azure Service is the workload you assign to it. The engine of Logic Apps is capable of handling large amounts of requests, however, has limits to throughput and the connectors (see connector reference) you use. However, Logic Apps offer concurrency control feature you can use in the designer, which enables you to **limit the runs of the Logic App** - the number of workflow instances.

Furthermore, with this feature you can prevent that backend systems will be overwhelmed with requests. Note that feature is only available for recurrence and polling triggers.

Another feature you can use for concurrency control is the **single instance setting**. This feature enables you to limit the number of concurrent instances to 1 for recurrence triggers and have your workflow only run once at a time – thus run Logic App as a singleton.

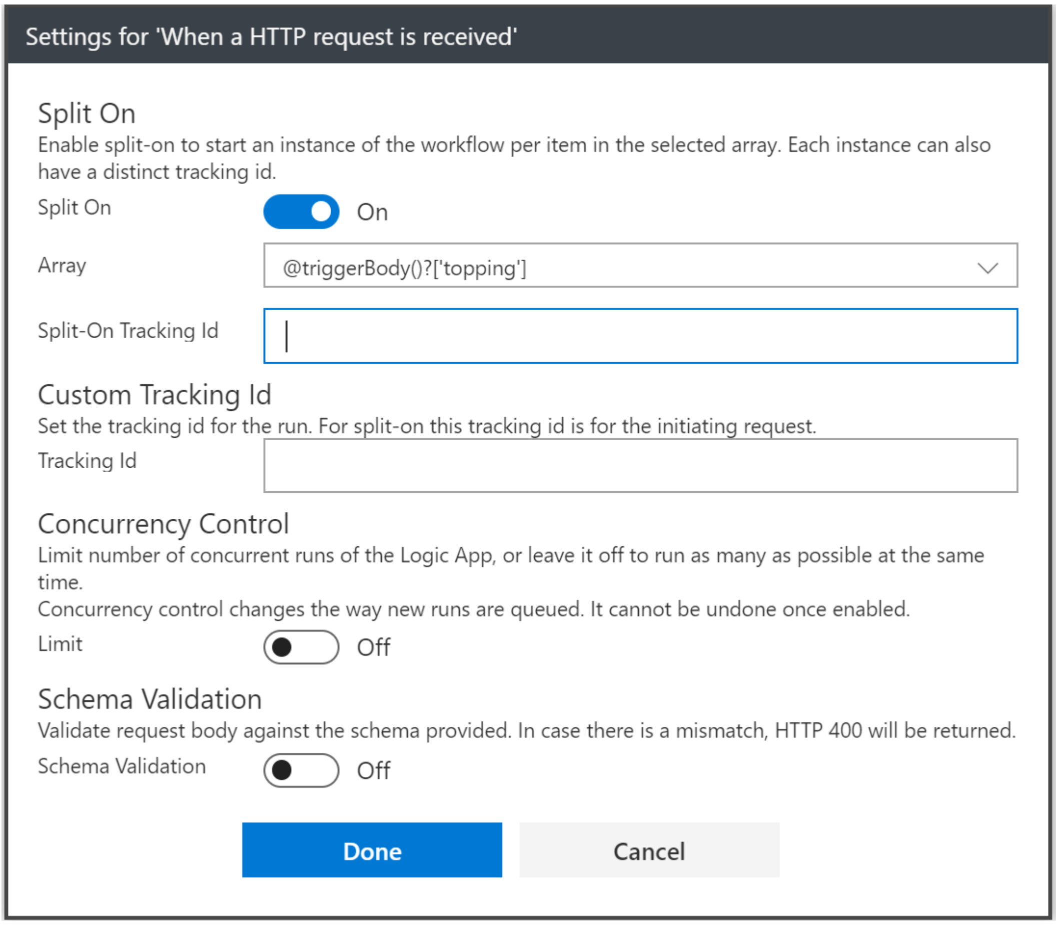

Settings for 'Recurrence'

Custom Tracking Id
Set the tracking id for the run. For split-on this tracking id is for the initiating request.
Tracking Id

Concurrency Control
Limit number of concurrent runs of the Logic App, or leave it off to run as many as possible at the same time.
Concurrency control changes the way new runs are queued. It cannot be undone once enabled.
Limit: On
Degree of Parallelism: 1

Split-On command

A widespread integration scenario is debatching a message - split a message into smaller messages based upon repeating elements. In Logic Apps you can loop through the repeating elements in a message, or you can use the split-on feature. You define a split-on command in a trigger in a Logic App.


Settings for 'When a HTTP request is received'

Split On

Enable split-on to start an instance of the workflow per item in the selected array. Each instance can also have a distinct tracking id.

Split On: On

Array: @triggerBody()?['topping']

Split-On Tracking Id:

Custom Tracking Id

Set the tracking id for the run. For split-on this tracking id is for the initiating request.

Tracking Id:

Concurrency Control

Limit number of concurrent runs of the Logic App, or leave it off to run as many as possible at the same time.

Concurrency control changes the way new runs are queued. It cannot be undone once enabled.

Limit: Off

Schema Validation

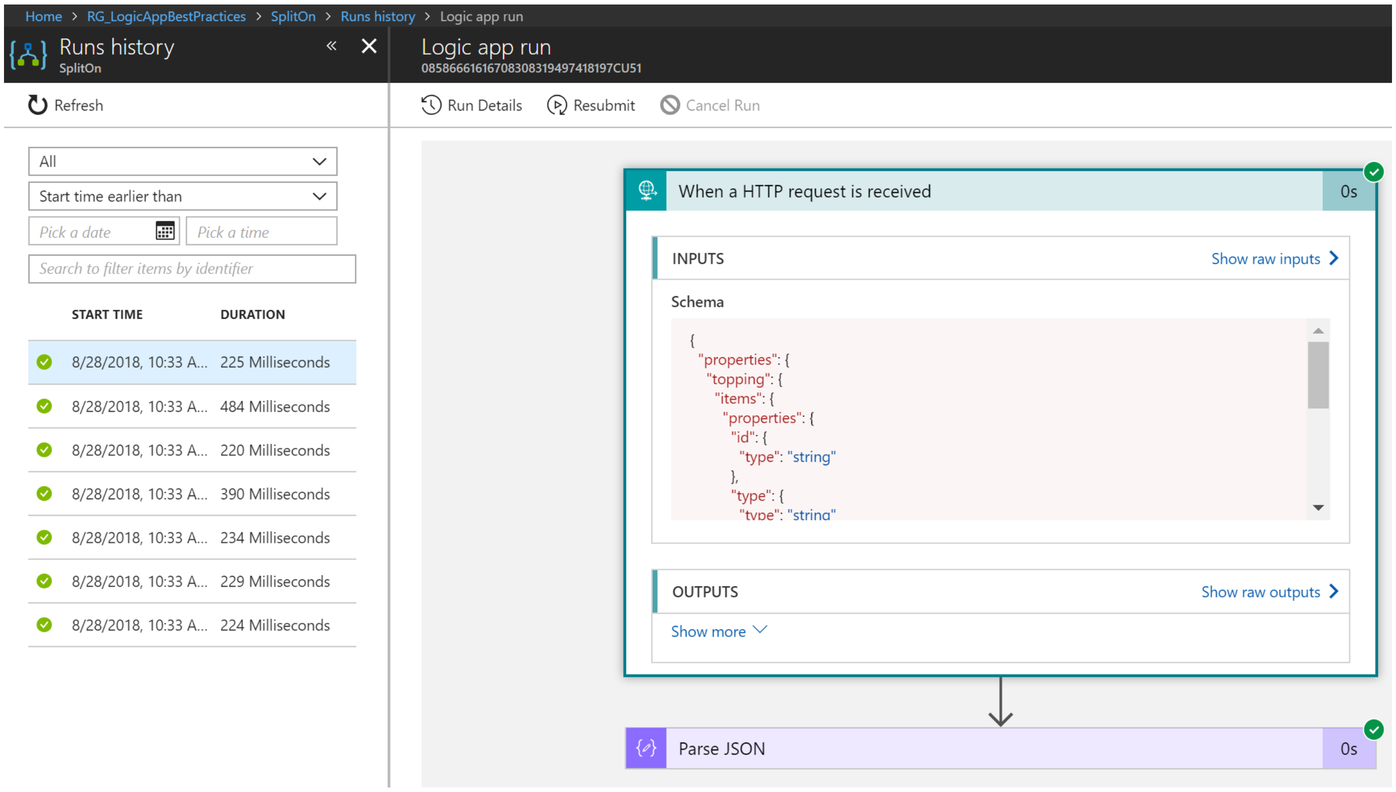
Validate request body against the schema provided. In case there is a mismatch, HTTP 400 will be returned.

Schema Validation: Off

In the screenshot above, you can see a Split-On command defined for an incoming request message – a request containing an array. When you send an array of several repeating elements like below to the Logic App endpoint than an instance of a Logic App is created for each element.

```
{
"topping":
[
{ "id": "5001", "type": "None" },
{ "id": "5002", "type": "Glazed" },
{ "id": "5005", "type": "Sugar" },
{ "id": "5007", "type": "Powdered Sugar" },
{ "id": "5006", "type": "Chocolate with Sprinkles" },
{ "id": "5003", "type": "Chocolate" },
{ "id": "5004", "type": "Maple" }
]
}
```

The request above is sent to an HTTP trigger configured with the Split-On command will result in seven instances of the Logic App.



START TIME	DURATION
8/28/2018, 10:33 A...	225 Milliseconds
8/28/2018, 10:33 A...	484 Milliseconds
8/28/2018, 10:33 A...	220 Milliseconds
8/28/2018, 10:33 A...	390 Milliseconds
8/28/2018, 10:33 A...	234 Milliseconds
8/28/2018, 10:33 A...	229 Milliseconds
8/28/2018, 10:33 A...	224 Milliseconds

```
{
  "properties": {
    "topping": {
      "items": {
        "properties": {
          "id": {
            "type": "string"
          },
          "type": {
            "type": "string"
```

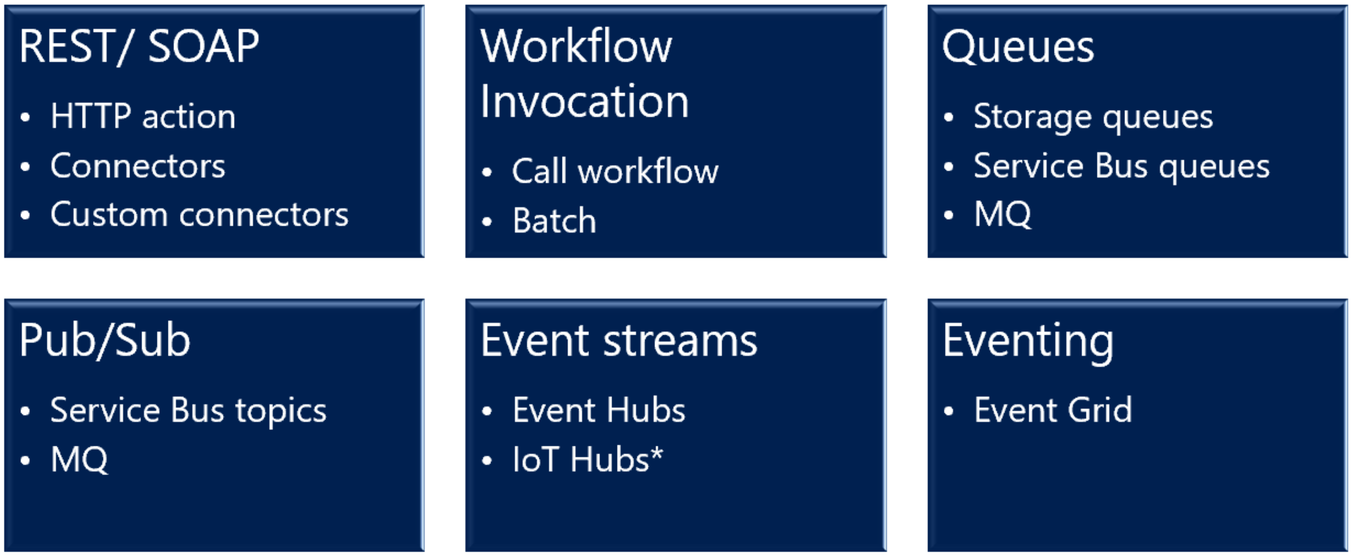
The Split-On command provides you with a way to process each element in array independently from the other. If one of the Logic App instances fails, it will not impact the other and handling of an exception can be done in the instance itself. Furthermore, you can make better use of the resubmit functionality.

Scheduling executions

The recurrence trigger (schedule) in Logic Apps is useful to perform workflows that need to run by a schedule. For instance, you can think of cleanup jobs, data synchronization, or running reports. You can set the recurrence to run every 4 hours per day to poll data from a public data source like weather data. Furthermore, you can set a the start time for a recurrence trigger or run the Logic App only once. For an example of using a recurrence trigger see the tutorial - Check traffic with a scheduler-based logic app.

Messaging Patterns

With Logic Apps you can support various messaging patterns when interacting with other Azure Services. You can expose a workflow by an HTTP trigger, handle messages from a service bus queue, support publish-subscribe with service bus topics, and subscribe to events.



REST/ SOAP
- HTTP action
- Connectors
- Custom connectors

Workflow Invocation
- Call workflow
- Batch

Queues
- Storage queues
- Service Bus queues
- MQ

Pub/Sub
- Service Bus topics
- MQ

Event streams
- Event Hubs
- IoT Hubs*

Eventing
- Event Grid

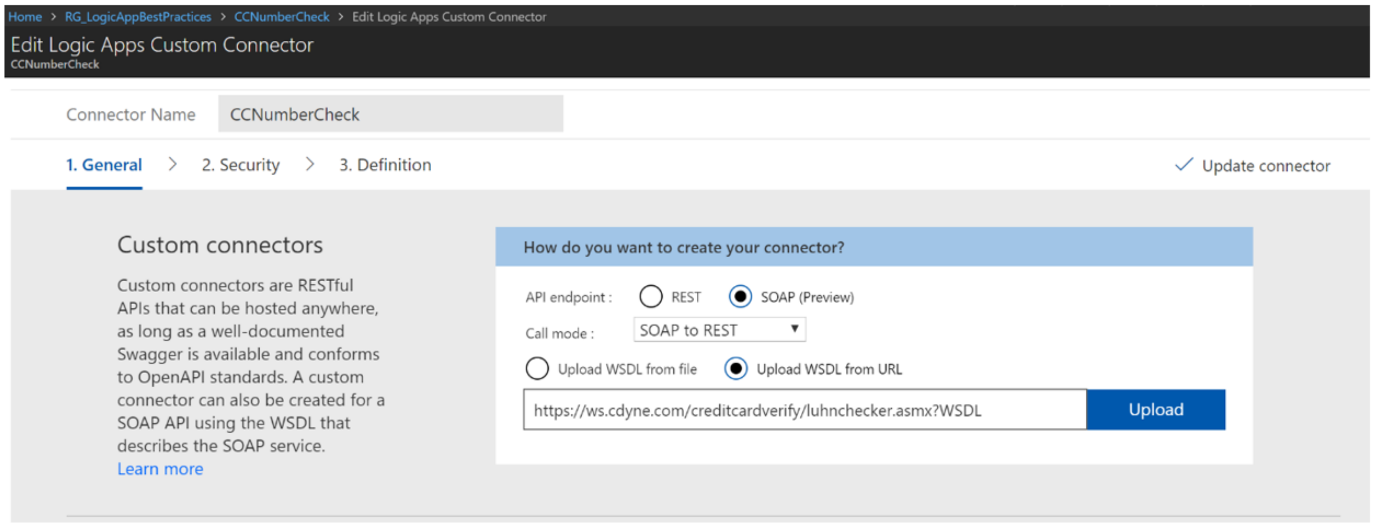
In the following paragraphs, we will discuss messaging patterns with Logic Apps.

REST and SOAP

Logic Apps provide native support for REST and SOAP. By default, Logic App supports REST, and each managed connector by Microsoft supports JSON. Moreover, Microsoft has developed over 200 connectors for Logic Apps enabling you to connect to various SaaS solutions like Salesforce, a wide variety of Azure Services, and on-premise connectors like File-connector.

In case connector is available you can create a custom connector, or call HTTP or HTTPS endpoints using the HTTP connectors as described in Call HTTP or HTTPS endpoints with Azure Logic Apps Microsoft docs. Furthermore, the native SOAP support is offered through the custom OpenAPI connectors. By importing the WSDL of a web service, you can create a connector that enables you to communicate with SOAP Services in your Logic App. In the Azure Marketplace, you can find the ***Logic App Custom Connector service***, which provides you with a wizard enabling you to import, define, and create your Logic App.

You can, for instance, create a custom connector to a public SOAP service that will check a given credit card number: https://ws.cdyne.com/creditcardverify/luhnchecker.asmx. You can find **the Logic App Custom connector service** in the Azure Marketplace. Create (provision) it and go through the steps of defining the custom connector.


Home > RG_LogicAppBestPractices > CCNumberCheck > Edit Logic Apps Custom Connector

Edit Logic Apps Custom Connector
CCNumberCheck

Connector Name: CCNumberCheck

1. General > 2. Security > 3. Definition — Update connector

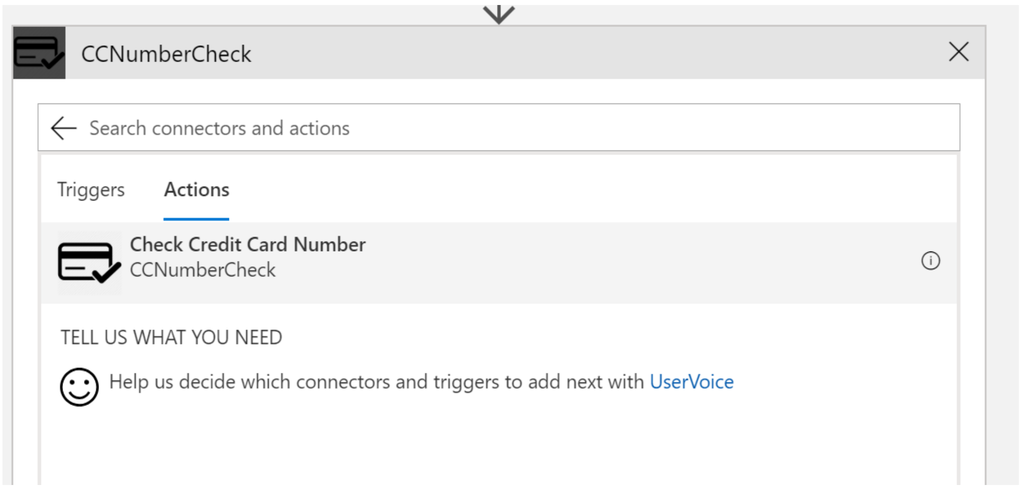
Custom connectors

Custom connectors are RESTful APIs that can be hosted anywhere, as long as a well-documented Swagger is available and conforms to OpenAPI standards. A custom connector can also be created for a SOAP API using the WSDL that describes the SOAP service.
Learn more

How do you want to create your connector?

API endpoint: REST [✓ SOAP (Preview)]
Call mode: SOAP to REST
Upload WSDL from file [✓ Upload WSDL from URL]
https://ws.cdyne.com/creditcardverify/luhnchecker.asmx?WSDL
Upload

Once you have gone through the wizard and create your custom connector supporting a SOAP service you can use the connector in a Logic App.


CCNumberCheck

Search connectors and actions

Triggers Actions

Check Credit Card Number
CCNumberCheck

TELL US WHAT YOU NEED

Help us decide which connectors and triggers to add next with UserVoice

The call to the service https://ws.cdyne.com/creditcardverify/luhnchecker.asmx by the Logic App will result in a request from REST to SOAP and response from SOAP to REST.

Workflow Invocation

A trigger instantiates a Logic App flow. When you define your workflow you start by a trigger - either one will be available for you at the start when choosing one of the pre-built templates. Otherwise, you choose one yourself when picking a blank template.

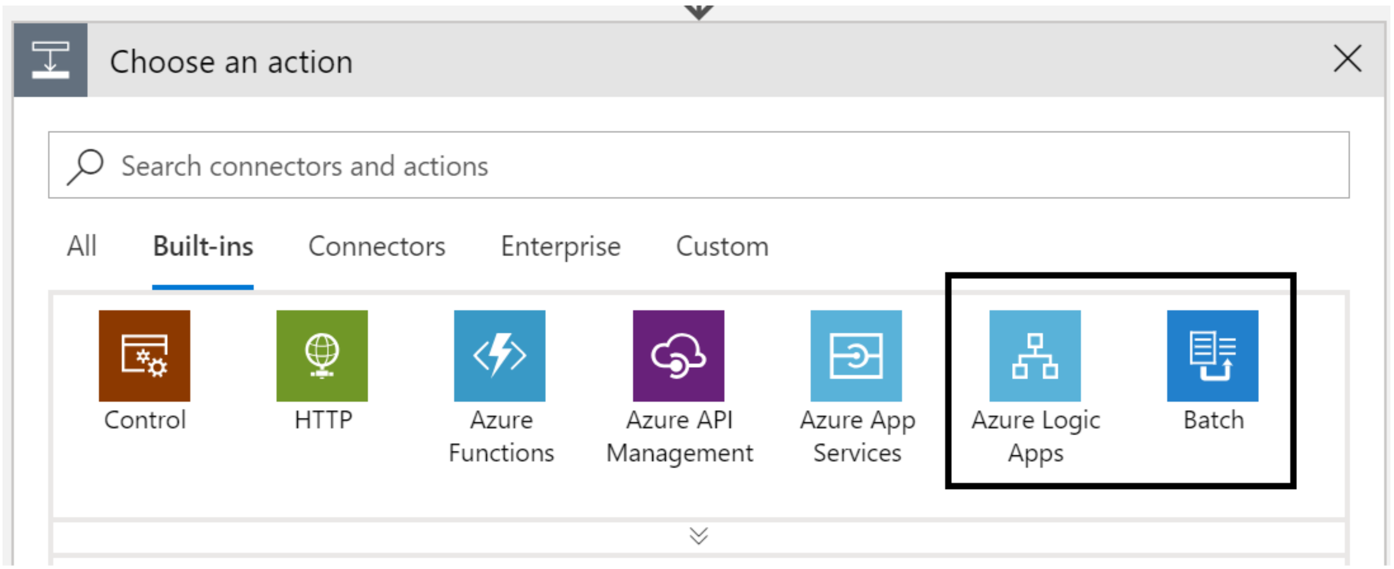
A trigger can either be a polling type, which means it polls a service like service bus, for instance, to see if any new messages are available. Or the trigger can be a push, a directly callable endpoint like HTTP trigger provides, or through a call-back mechanism by, for instance, an HTTP - Webhook or Event Grid trigger (see also the previous paragraph on REST and SOAP).

Generally, there are six types of trigger mechanisms leading to an instantiation of a Logic App:

- **Request**: Makes the Logic app an endpoint for you to call.
- **Recurrence**: Fires based on a defined schedule.
- **HTTP - Polls an HTTP web endpoint**. The HTTP endpoint must conform to a specific triggering contract - either by using a 202-async pattern or by returning an array.
- **API Connection:** Polls like the HTTP trigger, however, it takes advantage of the Microsoft managed APIs.
- **HTTP Webhook**: Opens an endpoint, similar to the Manual trigger. However, it also calls out to a specified URL to register and unregister.
- **API connection Webhook**: Operates like the HTTP Webhook trigger by taking advantage of the Microsoft managed APIs.

Furthermore, a trigger can be set via a condition, when it is met or with Split-On. If Split On is placed on a trigger and if an array is passed to the trigger - the logic apps engine will split that array into one logic apps instance per item in the array. Furthermore, the trigger will have a single instance output instead of an array (see also the Split-On paragraph). A trigger can also occur through a custom created connector.

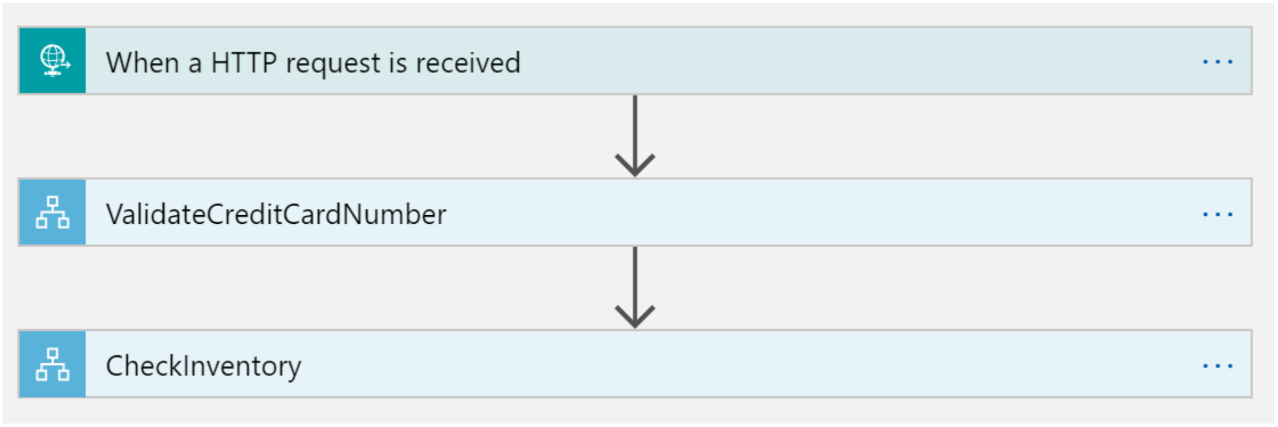
You can also trigger a workflow from another workflow – you can call a Logic App from another Logic App. The built-in capabilities in a workflow provide you with a way to call another Logic App (Child) or Batch.



For Batch see also the Middleware Friday Episode - Batching and Debatching with Logic Apps.

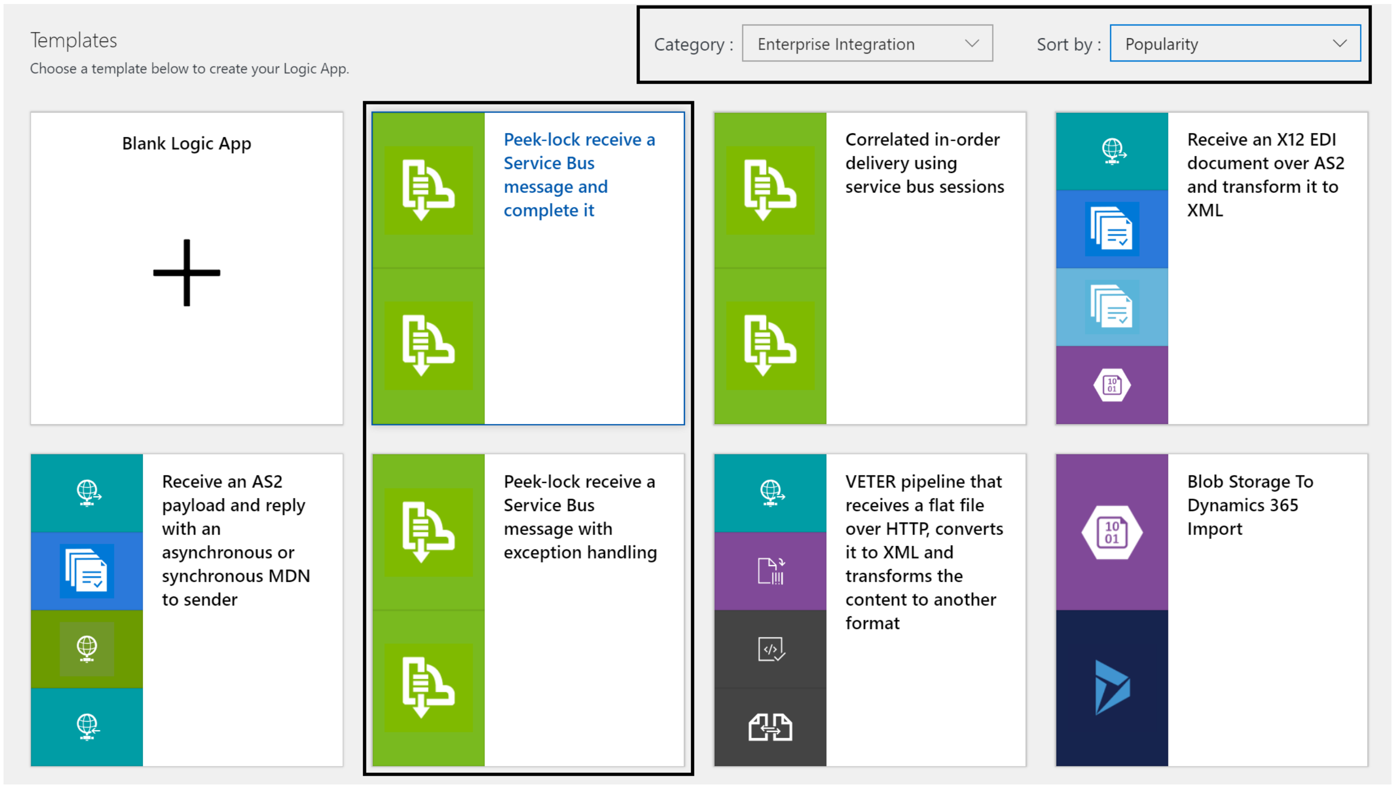
When you want to call another Logic, you will need to select that from the **Azure Logic Apps** action.

Assume you want to validate an incoming order and require to do this is in a few independent steps - you build a Validate Order Logic App that can receive the order via an HTTP request – subsequently you call Check Credit Card Logic App and Check Inventory Logic App. The benefit of calling other Logic Apps can be reuse or prevent putting too much complexity into one Logic App.



Queues

One of the most used Logic App connectors is the Service Bus connector. With the Service Bus connector, you can listen to a queue or topic subscription – bringing asynchronous communication to Logic Apps. Furthermore, Logic Apps provide a few useful prebuilt templates for the Service Bus Connector.


Templates
Choose a template below to create your Logic App.
Category : Enterprise Integration
Sort by : Popularity
Blank Logic App
Peek-lock receive a Service Bus message and complete it
Correlated in-order delivery using service bus sessions
Receive an X12 EDI document over AS2 and transform it to XML
Receive an AS2 payload and reply with an asynchronous or synchronous MDN to sender
Peek-lock receive a Service Bus message with exception handling
VETER pipeline that receives a flat file over HTTP, converts it to XML and transforms the content to another format
Blob Storage To Dynamics 365 Import

With the Service Bus connector in Logic Apps, you can support many scenarios, including asynchronous processing of messages, implement a publish-subscribe pattern, and fire-and-forget (send a message to a queue). Furthermore, you can chain Logic Apps together with a queue to decouple to achieve loose coupling. In the previous paragraph, we discussed calling a Logic App directly from another Logic App. Also, you can use the service bus to correlate messages going through Logic Apps. Two good samples and explanations for correlation are:

- Correlating messages over Logic Apps using Service Bus
- Logic Apps: Correlation and Message Dependency Management on Logic Apps with Service Bus

See also Microsoft docs for communication with Service Bus: Exchange messages in the cloud with Azure Service Bus and Azure Logic Apps

Pub/Sub

A very well-known messaging pattern is the publish-subscribe pattern. The foundation of BizTalk is based upon this pattern. According to Wikipedia:

"publish-subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are."

The Logic App Service Bus connector provides support for you to subscribe to service bus topic subscriptions and thus process messages you're interested in your Logic App. You can create a Service Bus Topic in Azure and generate one or multiple subscriptions based upon filter expressions.

Event Streams

With Logic Apps you can receive and send events to Azure Event Hub – a telemetry ingestion service that can collect, transform, and store millions of events. The Event Hub connector in Logic Apps can enable you to send and receive events – thus process or send out events. A good example is described in the Azure Logic Apps and Azure Event Hubs blog post and Middleware Friday Episode.

By leveraging the service bus queue connector in a Logic App, you can indirectly pick up events from the IoT Hub. In IoT hub, you can create a routing rule to send events to a service bus queue.

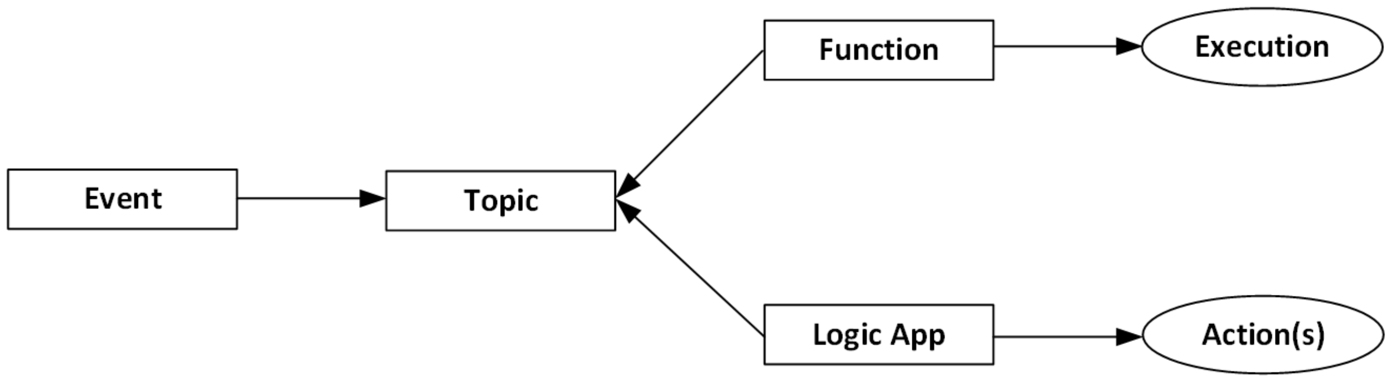
A Logic App can listen to that particular queue and pick up any message that arrives on that queue as explained in the paragraph on queues. An example of working with IoT Hub and Logic Apps is available on git hub: IoT remote monitoring and notifications with Azure Logic Apps connecting your IoT hub and mailbox.

Eventing

Event Grid is a Platform Service in Azure, which provides intelligent event routing through filters and event types. Moreover, it offers a uniform publish-subscribe model similar to the model of the BizTalk runtime.

However, we are talking about events here and not messaging - Event Grid is a managed service in Azure with service fabric underneath to support autoscale. Furthermore, the service is billed per operation, and no management of any servers is required.

Event Grid offers custom event routing capabilities with an Event Grid Topic. Consequently, a Topic can be provisioned through the Azure Portal. And once the Topic becomes available, you can hook it up with one or more subscribers, for instance, a Logic App. The subscription is set for either Event Type and/or filters (Prefix, Suffix).



The consumption of the events can be through the Event Grid Topic as visualized above. Furthermore, consumers can be a Function, Logic App, WebHook or Azure Automation - and the mechanism of subscriptions in Logic App and Functions is through WebHooks. On Serverless360 blog you find a good sample of consuming events from Event Grid: How To Build A Reactive Solution With Azure Event Grid.

Messaging Communication Patterns

The messaging communication patterns like synchronous, asynchronous, and ordered delivery are applicable for Logic Apps. You can expose a workflow via HTTP, process the request, and return a response synchronously, or accept a request to place it on a queue and start asynchronous communication. In the following paragraphs, we will discuss the message communication patterns applicable to Logic Apps.

Direct Synchronous

You can expose a Logic App to the outside world via an HTTP endpoint leveraging the HTTP trigger action in the Logic App definition. The Logic App can consume a request, do some work, and return the result to the caller (client). A trigger can be a push, a directly callable HTTP endpoint (trigger), or through a call-back mechanism by, for instance, an HTTP - Webhook or Event Grid trigger (see also Workflow Invocation). Direct synchronous messaging for Logic Apps can be summarized as:

- Request/Reply (blocking), explained above, and
- Call workflow (synchronous)

You can develop Request-Reply Logic App by creating a Logic App by starting with an HTTP trigger. Subsequently, you define the payload of the incoming request and generate the JSON-schema. Next, you follow with one or more actions, and you finish with a response action.

For instance, you can create a Logic App that receives a simple request like below:

{

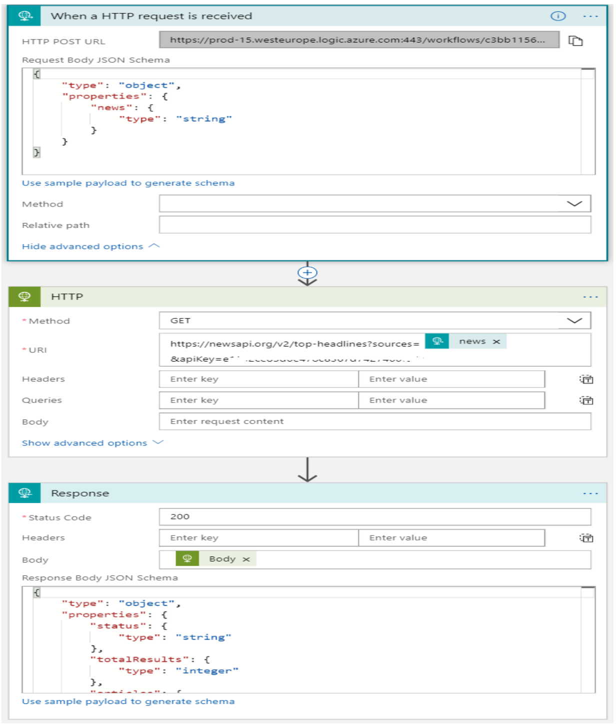
"news" : "techcrunch"

}

The request above also serves as payload in the HTTP trigger to define the JSON-schema. Next, you follow with an HTTP action that will call a news API like https://newsapi.org/. You specify a GET operation for this public API to get top stories based on a source (techcrunch):

https://newsapi.org/v2/top-headlines?sources= techcrunch&apikey=..................

You define the response action after the HTTP action and return the response from the HTTP action. Furthermore, you create a JSON-schema of the response message by testing the HTTP GET operation with, for instance, Postman. The last step is to save the Logic App workflow definition, and it will generate an HTTP endpoint you can call.



When a HTTP request is received

HTTP POST URL: https://prod-15.westeurope.logic.azure.com:443/workflows/c3bb1156...

Request Body JSON Schema

```
{
    "type": "object",
    "properties": {
        "news": {
            "type": "string"
        }
    }
}
```

HTTP

* Method: GET

* URI: https://newsapi.org/v2/top-headlines?sources= news
&apiKey=e[illegible]

Response

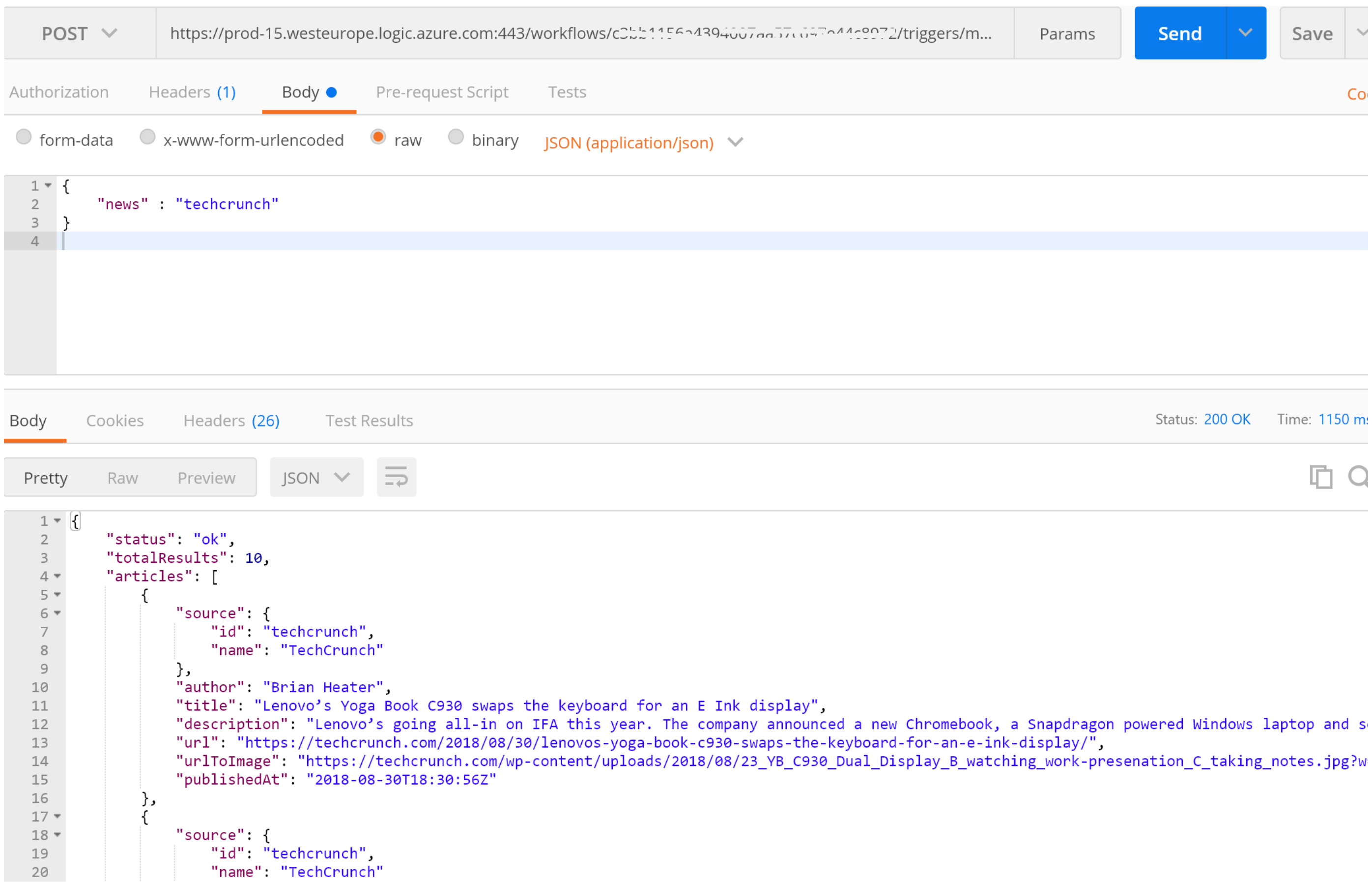
* Status Code: 200

Body: Body

Response Body JSON Schema

```
{
    "type": "object",
    "properties": {
        "status": {
            "type": "string"
        },
        "totalResults": {
            "type": "integer"
        },
```

You can test the Logic App by calling the endpoint using Postman.



```
{
    "news" : "techcrunch"
}

```

```
{
    "status": "ok",
    "totalResults": 10,
    "articles": [
        {
            "source": {
                "id": "techcrunch",
                "name": "TechCrunch"
            },
            "author": "Brian Heater",
            "title": "Lenovo’s Yoga Book C930 swaps the keyboard for an E Ink display",
            "description": "Lenovo’s going all-in on IFA this year. The company announced a new Chromebook, a Snapdragon powered Windows laptop and s
            "url": "https://techcrunch.com/2018/08/30/lenovos-yoga-book-c930-swaps-the-keyboard-for-an-e-ink-display/",
            "urlToImage": "https://techcrunch.com/wp-content/uploads/2018/08/23_YB_C930_Dual_Display_B_watching_work-presenation_C_taking_notes.jpg?w
            "publishedAt": "2018-08-30T18:30:56Z"
        },
        {
            "source": {
                "id": "techcrunch",
                "name": "TechCrunch"
```

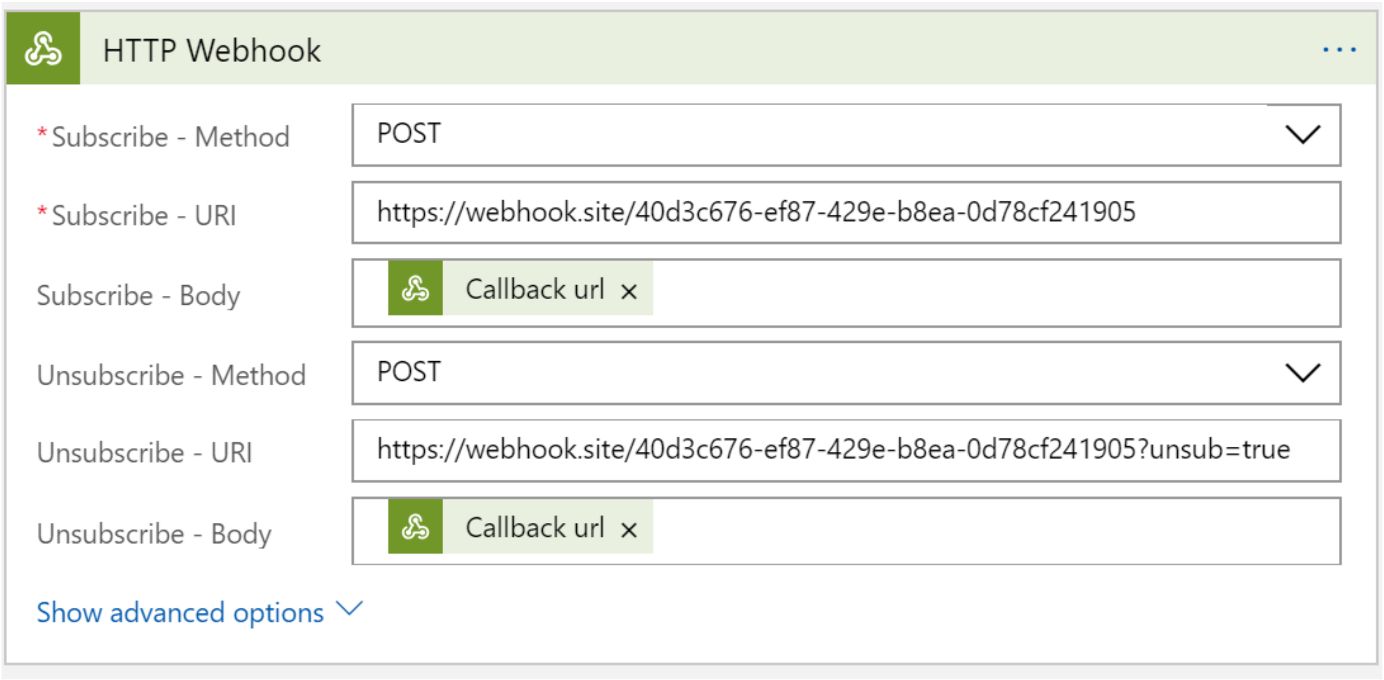
Thus, here you have an easy example of how to create a Request/Reply Logic App.

Direct Asynchronous

Asynchronous communication with Logic Apps is through the use of a trigger that pulls, for instance, a message from a Service Bus queue (see also Queues). Think direct asynchronous communication with Logic Apps about:

- Fire and forget (201)
- Call Workflow (async) and Send to Batch
- Fire and Wait/Poll (202 with retry-after and location header)
- Webhooks

Assume you need to implement a call-back mechanism with Logic Apps. You can apply this mechanism by creating a Logic App starting with a Webhook trigger and specify the details for the webhook subscribe and unsubscribe calls. For example, we can use the HTTP WebHook and subscribe and subscribe to WebHook tester.

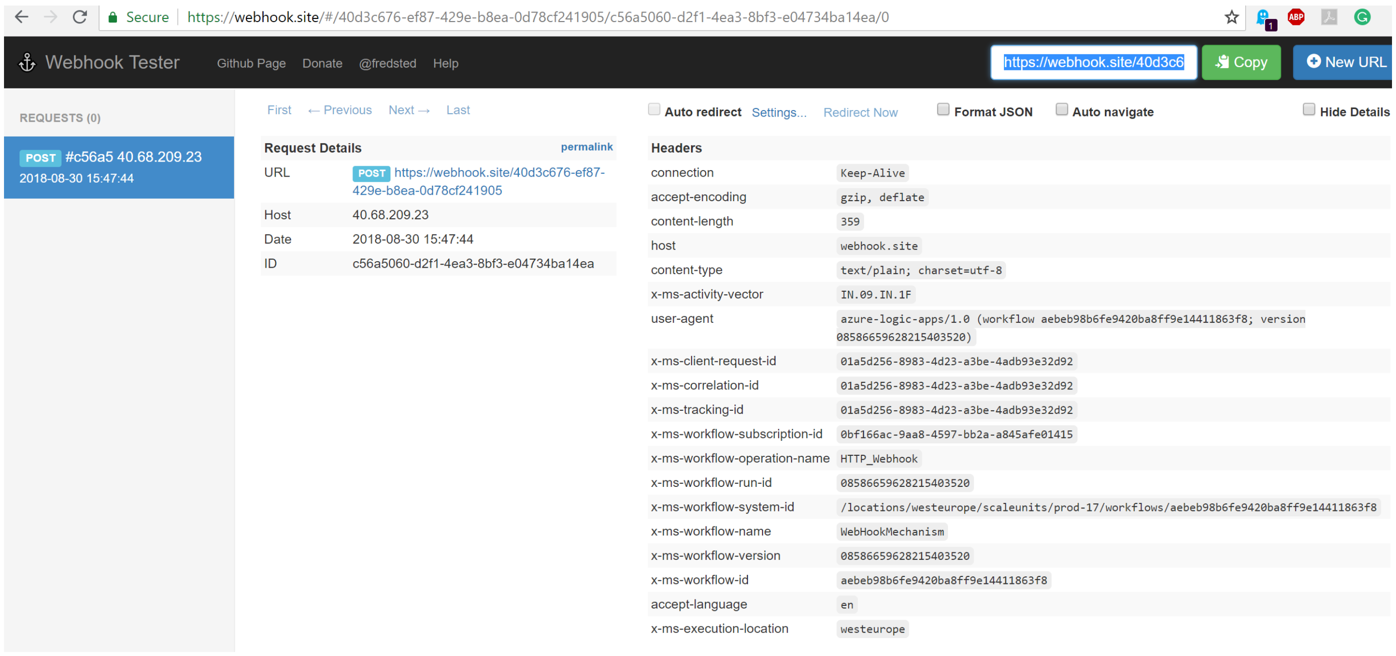


HTTP Webhook

* Subscribe - Method	POST
* Subscribe - URI	https://webhook.site/40d3c676-ef87-429e-b8ea-0d78cf241905
Subscribe - Body	Callback url
Unsubscribe - Method	POST
Unsubscribe - URI	https://webhook.site/40d3c676-ef87-429e-b8ea-0d78cf241905?unsub=true
Unsubscribe - Body	Callback url

Show advanced options

In the Webhook tester, you can see the result of registration (subscription).



Secure https://webhook.site/#/40d3c676-ef87-429e-b8ea-0d78cf241905/c56a5060-d2f1-4ea3-8bf3-e04734ba14ea/0

Webhook Tester Github Page Donate @fredsted Help https://webhook.site/40d3c6 Copy New URL

REQUESTS (0)

POST #c56a5 40.68.209.23
2018-08-30 15:47:44

First ← Previous Next → Last

Auto redirect Settings... Redirect Now Format JSON Auto navigate Hide Details

Request Details	permalink
URL	POST https://webhook.site/40d3c676-ef87-429e-b8ea-0d78cf241905
Host	40.68.209.23
Date	2018-08-30 15:47:44
ID	c56a5060-d2f1-4ea3-8bf3-e04734ba14ea

Headers	
connection	Keep-Alive
accept-encoding	gzip, deflate
content-length	359
host	webhook.site
content-type	text/plain; charset=utf-8
x-ms-activity-vector	IN.09.IN.1F
user-agent	azure-logic-apps/1.0 (workflow aebeb98b6fe9420ba8ff9e14411863f8; version 08586659628215403520)
x-ms-client-request-id	01a5d256-8983-4d23-a3be-4adb93e32d92
x-ms-correlation-id	01a5d256-8983-4d23-a3be-4adb93e32d92
x-ms-tracking-id	01a5d256-8983-4d23-a3be-4adb93e32d92
x-ms-workflow-subscription-id	0bf166ac-9aa8-4597-bb2a-a845afe01415
x-ms-workflow-operation-name	HTTP_Webhook
x-ms-workflow-run-id	08586659628215403520
x-ms-workflow-system-id	/locations/westeurope/scaleunits/prod-17/workflows/aebeb98b6fe9420ba8ff9e14411863f8
x-ms-workflow-name	WebHookMechanism
x-ms-workflow-version	08586659628215403520
x-ms-workflow-id	aebeb98b6fe9420ba8ff9e14411863f8
accept-language	en
x-ms-execution-location	westeurope

The pictures above are an illustration of how to set up a Webhook in a Logic App. You can after specifying the Webhook trigger follow with several actions depending on your requirements. For more details, see Create event-based workflows or actions by using webhooks and Azure Logic Apps on Microsoft docs.

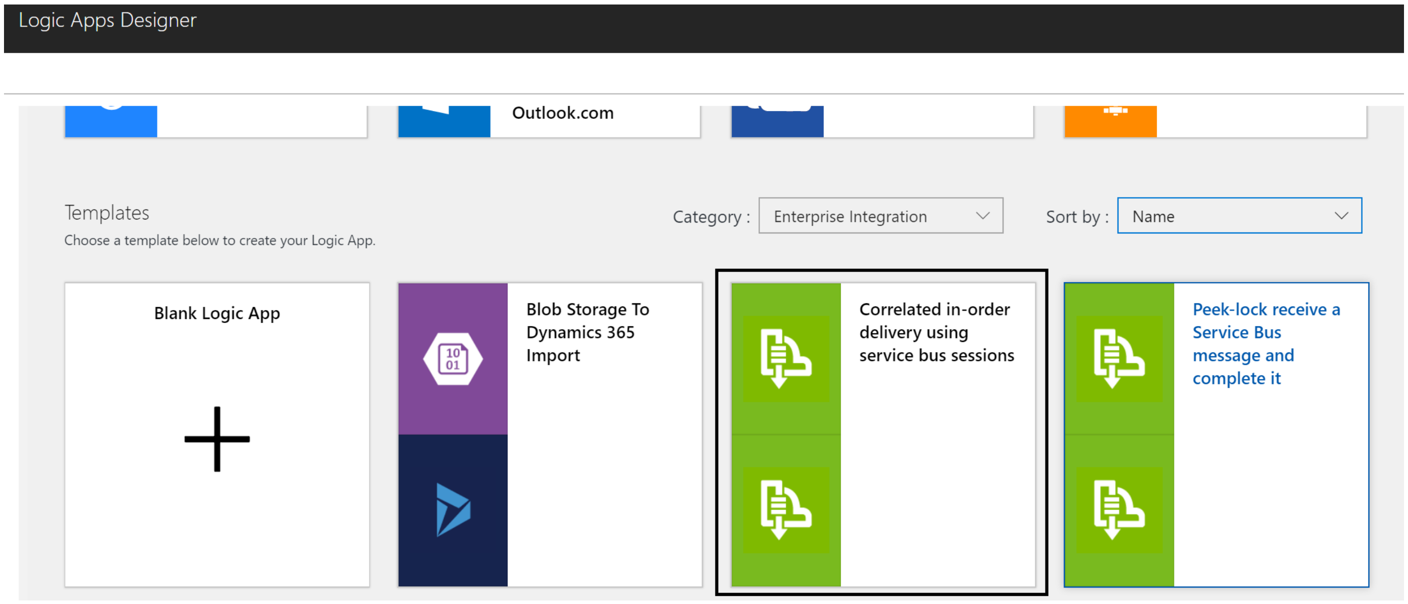
Correlation

Business processes can run short or for a long time. For instance, you could use Logic Apps to support a multiple approval step process, or wait for an external process to finish using a Webhook, or communicate with queues. In these scenarios, we deal with correlation. Generally, we can summarize them in:

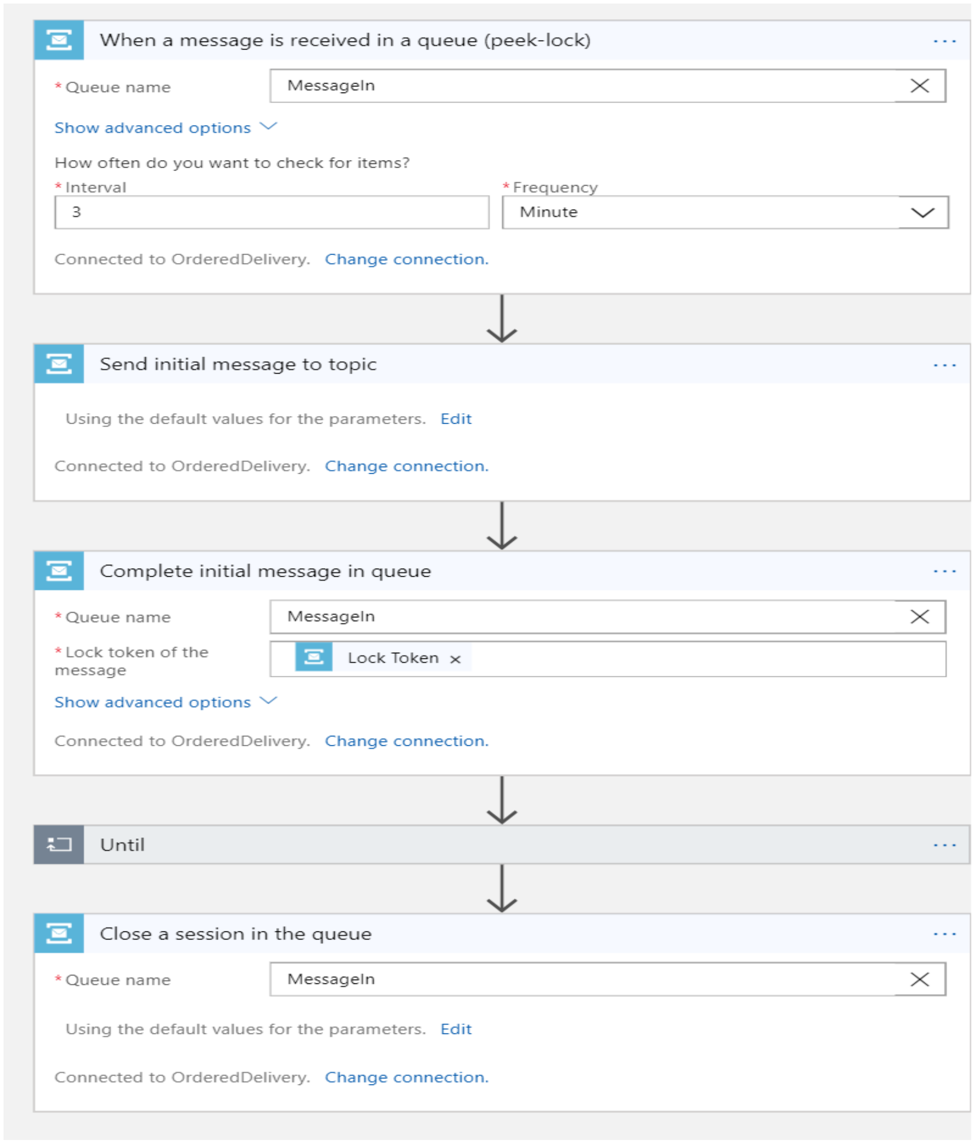
- Send – Async Response
- Webhook (Approval)
- Response Queues
- Session queues

Correlating messages are useful in a scenario of where you want to send out messages in the same order as they came. Logic Apps can support this scenario or what you might know as a sequential convoy from the BizTalk world. The Service Bus connector supports sessions- thus based on the session id you can have all the correlated messages together send them out in the same order as they came in. You basically can follow the following steps for implementing in-order delivery in Logic Apps:

- Create a Service Bus Namespace.
- Create a session-aware queue.
- Create topics which have names based on the values you will use for the **sessionid** property set on each message.
- Create a Logic App.
- Choose the "*Correlated in-order delivery using service bus sessions*" template


Logic Apps Designer
Outlook.com
Templates
Choose a template below to create your Logic App.
Category : Enterprise Integration
Sort by : Name
Blank Logic App
Blob Storage To Dynamics 365 Import
Correlated in-order delivery using service bus sessions
Peek-lock receive a Service Bus message and complete it

- Connect to the service bus; the first trigger is the service bus – you need to select the appropriate namespace (first step).
- Specify details in the workflow definition.



When a message is received in a queue (peek-lock)

* Queue name: MessageIn

Show advanced options

How often do you want to check for items?

* Interval: 3

* Frequency: Minute

Connected to OrderedDelivery. Change connection.

Send initial message to topic

Using the default values for the parameters. Edit

Connected to OrderedDelivery. Change connection.

Complete initial message in queue

* Queue name: MessageIn

* Lock token of the message: Lock Token

Show advanced options

Connected to OrderedDelivery. Change connection.

Until

Close a session in the queue

* Queue name: MessageIn

Using the default values for the parameters. Edit

Connected to OrderedDelivery. Change connection.

- Send messages to the queue by setting the session id property.

A more detailed description can be found on the MSDN blog In-order delivery of correlated messages in Logic Apps by using Service Bus sessions. Note that the Logic App designer has evolved over time and that the screenshots look different. Other resources you can look at with regards to correlation are:

- Message Re-sequencing using Azure Functions and Table storage (includes Logic Apps)
- The Logic Apps Webhook Action and the Correlation Identifier Pattern
- Enforcing Ordered Delivery using Azure Logic Apps and Service Bus

Partitions

In the previous paragraph, we discussed correlation and ordered delivery using session queues. You can also with Logic Apps group messages together as a batch. To implement batching messages together, you will need to Logic Apps:

- A **batch** logic app, which accepts and collects messages into a batch until your specified criteria are met for releasing and processing those messages.
- One or if you feel necessary multiple Logic Apps, which will send messages to the previously created **batch** Logic App.

How to set this up is described in Send, receive, and batch process messages in Azure Logic Apps and demonstrated in the Middleware Friday episode Batching and Debatching with Logic Apps.

Messaging Handling Patterns

There are several ways how you can handle a message in Logic Apps similar to how you do this with BizTalk Server. Splitting a message, for instance, is something you can do using a BizTalk pipeline or with the earlier described Split-On command in a Logic App trigger. For Batching and Debatching see also the Middleware Friday episode: Batching and Debatching with Logic Apps.

Other messages handling patterns are

- Claim Check
- Data Transformation and Enricher
- Content-based router
- Message pipelines (VETER)

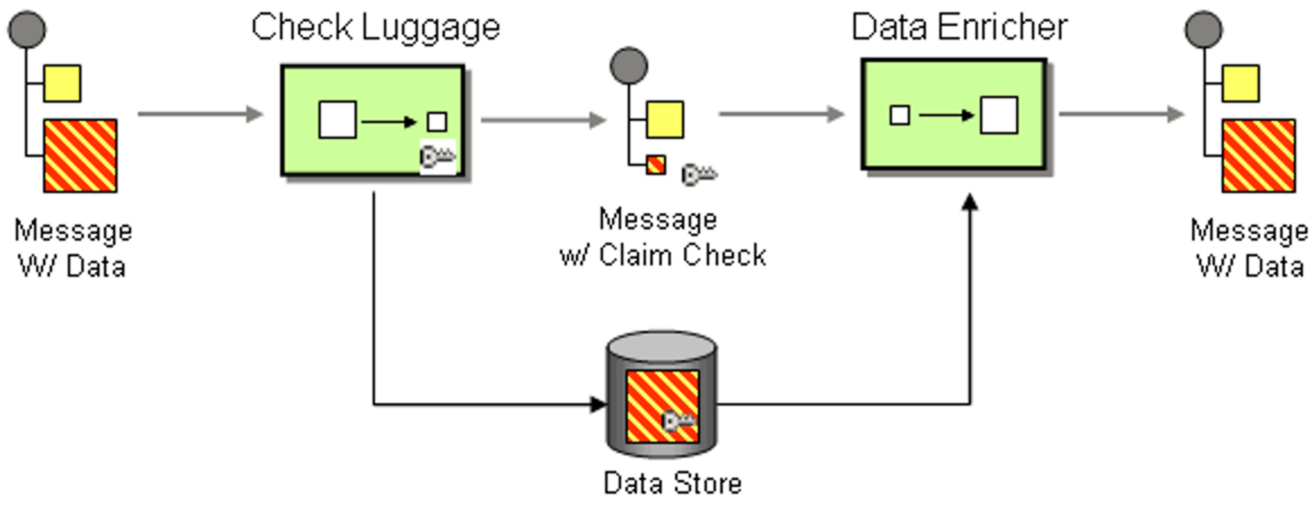
In the following paragraphs, we will discuss these message handling patterns for Logic Apps.

Claim Check

Like any service in Azure, Logic Apps have boundaries or limits. The size of a message can be a limitation – the standard tier of Service Bus only allows messages up to 256 kB.

Furthermore, with the premium tier, the limit is 1 Mb. The message size for a Logic App can be up to 100 Mb for a single HTTP request. Hence you cannot send message payload larger than 100 Mb to a Logic App exposed via HTTP(s) or have Logic App pick up a payload bigger than 100 Mb from the storage container. However, you can overcome this limitation by leveraging the Claim Check Pattern, which according to enterpriseintegrationpatterns.com is:

"a message may contain a set of data items that may be needed later in the message flow, but that is not necessary for all intermediate processing steps. We may not want to carry all this information through each processing step because it may cause performance degradation and makes debugging harder because we carry so much extra data."



> *"Store message data in a persistent store and pass a Claim Check to subsequent components. These components can use the Claim Check to retrieve the stored information."*
>
> https://www.enterpriseintegrationpatterns.com/patterns/messaging/StoreInLibrary.html

Toon VanHoutte wrote an excellent blog post Service Bus Claim Check API App for Logic Apps - explaining how to apply the Claim Check Pattern with Logic Apps.

Data Transformation and Enricher

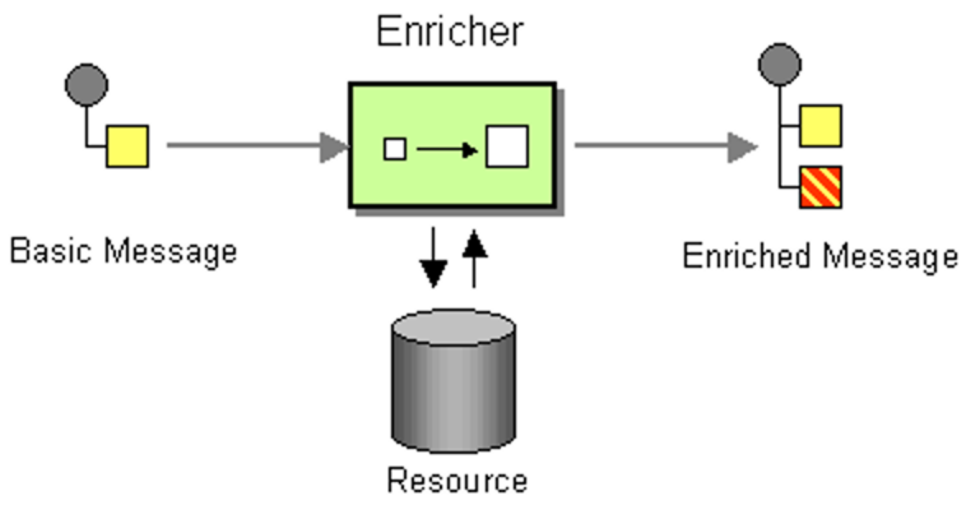
Another general pattern within integration is transforming or enriching of data. It will be of no surprise that you can transform and enrich data using Logic Apps. With an integration account as service for storing your maps, you can perform a mapping with XML and JSON messages. For instance, you can upload an XML or flat file schemas to an integration account to use them to convert a flat file payload to XML, or XML payload to JSON.

Furthermore, you can upload a Map to do the mapping from one format to the other. Note that once you want to do the transformation and leverage the integration account, you will increase the TCO (Total Cost of Ownership) of your Logic App, see also the Cost Estimation section later in this paper.

With the Microsoft Azure Logic Apps Enterprise Integration Tools for Visual Studio 2015 installed on your machine, you can create Schema's and Maps for your Logic Apps. Or in case you have a BizTalk Development environment at your disposal, you can create your schemas and maps in Visual Studio 2015. Not that you do not need to install the previously mentioned tools. Once installed, you can open Visual Studio 2015 and create an empty Integration Account solution. Subsequently, you can create a flat file, or XML schema, or a Map.

From Visual Studio 2015 you define XML and Flat File schemas and build a Map. The mapping experience is similar to creating mappings for BizTalk Server. Once you have created schemas and maps, you can upload them to your integration account. Now you will have a link the integration account to your Logic App and leverage the transformation capabilities.

Besides transforming a message in a Logic App, you can enrich a message. The enrichment of a message in Logic App can be accomplished in various ways. You can call an external endpoint directly using HTTP trigger, call a function, or API via API Management. With the information of the message entering a Logic App, you can retrieve data from a resource and append to the message data – thus enriched the message. The picture below shows the enricher pattern from enterpriseintegrationpatterns.com.

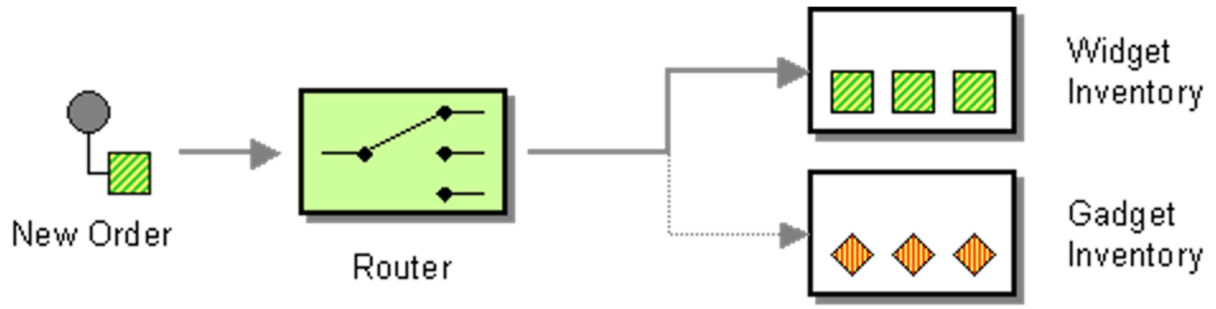


> *"Use a specialized transformer, a Content Enricher, to access an external data source to augment a message with missing information."*
>
> https://www.enterpriseintegrationpatterns.com/patterns/messaging/DataEnricher.html

Content-based router

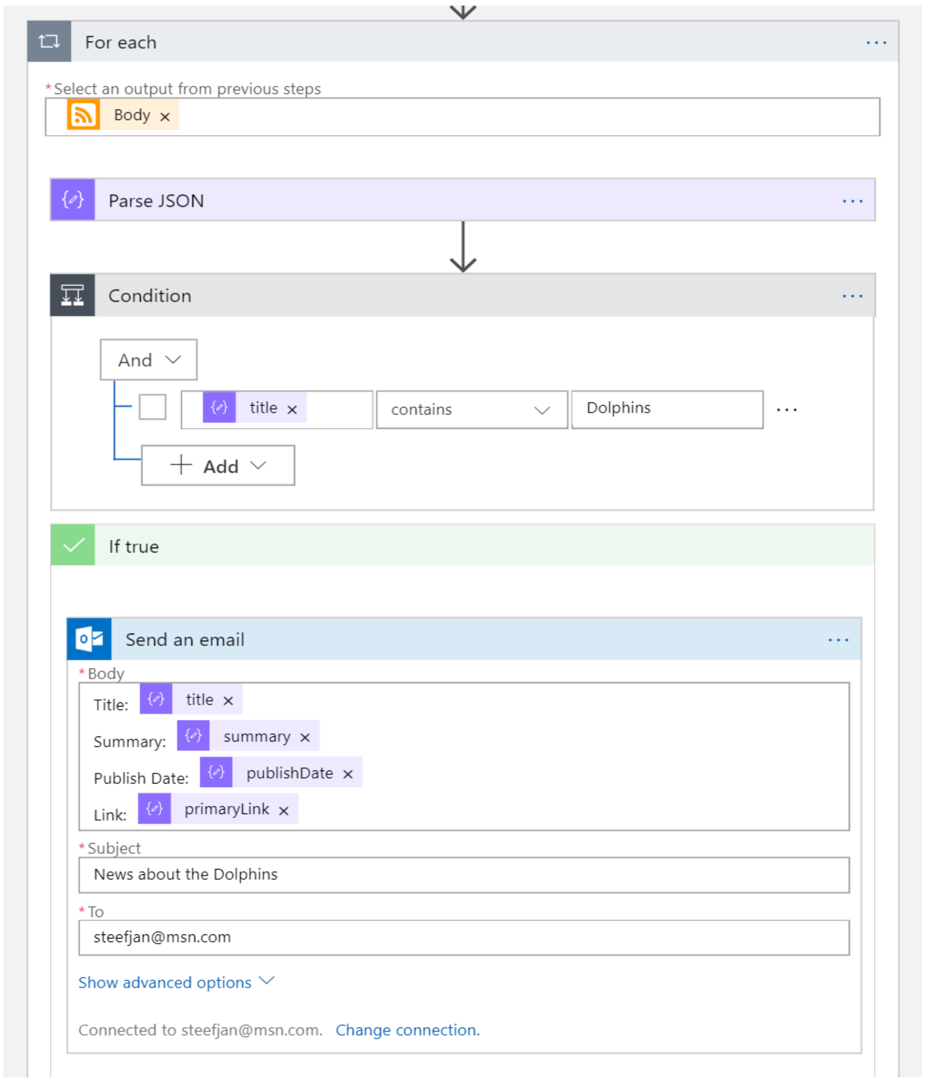
A favorite pattern in enterprise integration is content-based routing. A pattern that is widely applied in BizTalk solutions. You can also implement this pattern with Logic Apps. Let's first understand what content-based routing means and the best way to do so is by a picture - below shows the content-based router pattern from enterpriseintegrationpatterns.com.

> *"Use a Content-Based Router to route each message to the correct recipient based on message content."*
>
> https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html



In a Logic App, you can parse a JSON message (tokenize) and use the token (message elements/properties) in a condition to send a message to the correct recipient. The recipient can be Logic App action, sending the message to a queue or HTTP action using some of the tokens to call a restful endpoint.

Assume you consume an RSS-feed every day on NFL news, and you are interested in news about the Miami Dolphins. You can build a Logic App with a schedule trigger set for every day to consume the RSS-feed. Next, you parse the feed with Parse action and use the token 'title' in a condition action to verify if the title contains 'Dolphins'. If it does, then send an email to notify yourself about the news otherwise do something else.


For each

*Select an output from previous steps

Body

Parse JSON

Condition

And

title contains Dolphins

Add

If true

Send an email

*Body

Title: title

Summary: summary

Publish Date: publishDate

Link: primaryLink

*Subject

News about the Dolphins

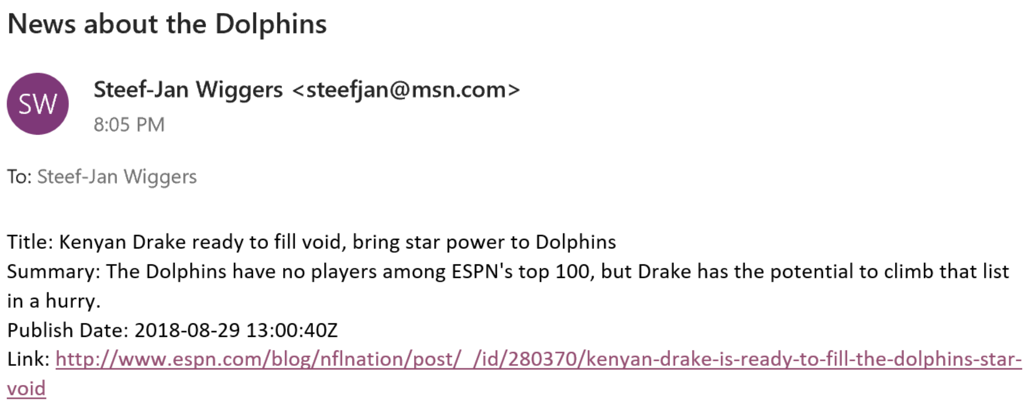
*To

steefjan@msn.com

Show advanced options

Connected to steefjan@msn.com. Change connection.

When you test such a solution, you will see an email notification when the condition is met as shown in the picture above.


News about the Dolphins

Steef-Jan Wiggers <steefjan@msn.com>

8:05 PM

To: Steef-Jan Wiggers

Title: Kenyan Drake ready to fill void, bring star power to Dolphins
Summary: The Dolphins have no players among ESPN's top 100, but Drake has the potential to climb that list in a hurry.
Publish Date: 2018-08-29 13:00:40Z
Link: http://www.espn.com/blog/nflnation/post/_/id/280370/kenyan-drake-is-ready-to-fill-the-dolphins-star-void

You can expand such a scenario by using a switch statement – thus sending news about different teams to different recipients.

Message Pipelines (VETER)

Logic Apps support handling of XML messages and even provides a template for it to Validate, Extract, Transform, Enrich and Route – the VETER template. The template is based upon the VETER pattern, which was first available in the retired MABS service, the processor or Logic Apps. You can leverage this template straight away through GitHub – you can provide a solution with this template. It creates an integration account, adds schema/map into it, creates a Logic App and associates it with the integration account. The Logic App implements a VETER pipeline using XML Validation, XPath Extract, and Transform XML operations. You can also start implementing a VETER pattern from scratch – you provision a Logic App and choose the template.

Best Practices

With Logic Apps, you need to consider a few best practices like working with variables, arrays, looping, security, monitoring, and cost estimations. In the following paragraphs, we will discuss these.

Working with variables

Within a Logic App definition, you can work with variables, which have a global scope. Variables can be used to count the number of times that a loop runs, or checking an array for a specific item - you can use a variable to reference the index number for each array item.

A variable in Logic Apps can be of datatype string, integer, float, boolean, array, and an object. You need to initialize the variable at the top of the workflow definition after the trigger action that instantiates the flow. Furthermore, you can initialize the value as well if necessary.

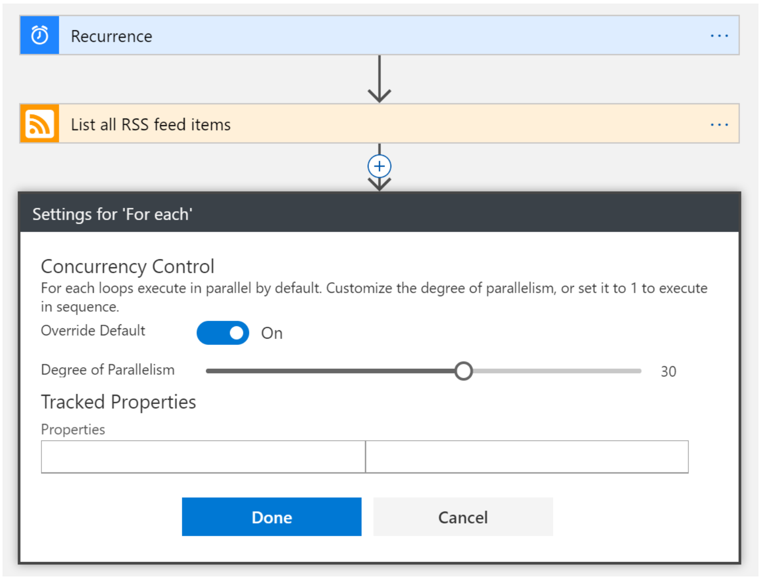
It is a best practice always to **assign a value** to a variable. Later in the flow, you can retrieve or edit the value that increments the value or set another value.

A sample is discussed in this blog https://connectedcircuits.wordpress.com/2017/10/08/integration-scatter-gather-pattern-using-azure-logic-apps-and-service-bus/

Arrays and looping

It is not unusual that you will encounter an array of elements in your message. Moreover, you might have to deal with arrays when developing Logic Apps. You can interact with arrays by going through each element using a for-each loop, or you can instantiate a separate flow (Logic App) for each element using the Split-On command. With the for-each loop, you can choose between running it sequential or in parallel.

Assume you have a news RSS-feed you like to consume every day and each feed contains one or more news items. You loop through the news items with the for-each control action in a Logic App. When configuring the For-Each, you can set the concurrency control to 1 up to 50 degrees of parallelism. With one, you run the for-each sequentially.



Settings for 'For each'

Concurrency Control

For each loops execute in parallel by default. Customize the degree of parallelism, or set it to 1 to execute in sequence.

Override Default On

Degree of Parallelism 30

Tracked Properties

Properties

In case you need the iteration to stop after a failure, you need to use the do-until control action. With the parallelism behavior, each iteration will execute as a separate thread – thus you can't terminate a logic app instance within a for each statement. When you do need to iterate through all the elements within an array whether you execute them in parallel or sequential a for-each is a good fit. Or when you need more isolation, do not need to aggregate results, and require faster execution you could use a child Logic App with the Split-On command. Below you'll find a table describing the difference between a For-Each action and calling a child Logic App with Split-On.

For-each action	**Call child Logic App with Split-On**
Run multiple iterations in parallel (50 max)	No upper limit on how many child Logic Apps can run in parallel after the split
Wait for all iterations to complete	Parent Logic App fire-and-forget
Aggregate results from all iterations to determine the for-each status	No aggregation in parent Logic App

There is a blog post by Toon VanHoutte about Working with collections in Logic Apps that provide you with extra details about handling arrays.

Security

A Logic App can be made directly accessible using an HTTP action, HTTP WebHook or Request/Response Trigger/Action. It will be triggered by a REST type call, i.e. methods like GET or POST. The HTTP Trigger and HTTP WebHook support authentication using login yet with an HTTP Request trigger the authentication has been provided by a SAS signature in the query string. The latter can be considered less secure as the signature can be sniffed as it is exposed in the query string.

A more secure way for an HTTP Request trigger in a Logic App can be restricting the incoming IP address using API Management. The only IP address allowed to call the HTTP Request trigger generated an address, is a specified API Management instance with a known IP address. Furthermore, by using Azure API Management you can add a layer of security on top of the Logic App endpoint.

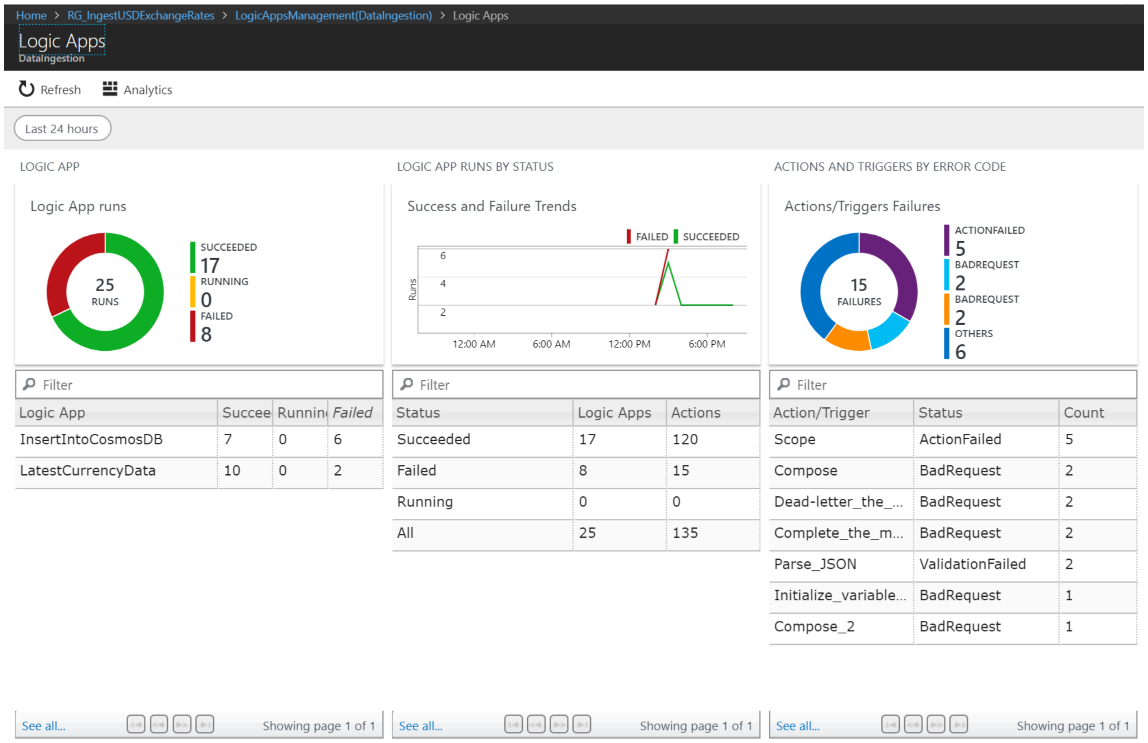
These other layers may include OAuth 2.0, Azure Active Directory, connecting to a virtual network, Mutual Certificate authentication and Basic authentication. See also the blog post Securing Logic App direct access endpoint HTTP Request using API Management.

Monitoring

When developing your Logic App, you need to think about monitoring as well. Fortunately, Azure provides you with a few services to monitor your Logic Apps. One of them is Azure Log Analytics, a monitoring service in Azure that collects and analyses log files from various Azure- and on-premise resources including Logic Apps.

Moreover, the service can collect all the data into a single workspace (Operation Management (OMS) Workspace) and provides a query language to query the ingested data.

Operations Management Suite (OMS) is a service in Azure that bundles capabilities like log analytics, IT automation, backup and recovery, and security and compliance tasks. An IT-pro can manage on-premises and cloud IT assets from one console. You can leverage OMS by setting up a workspace and collect data from other Azure Resources like for instance Logic Apps. Tying an OMS Workspace with Logic Apps provides you straight away get a view on your Logic App runs.



Logic App	Succee	Runnin	*Failed*
InsertIntoCosmosDB	7	0	6
LatestCurrencyData	10	0	2

Status	Logic Apps	Actions
Succeeded	17	120
Failed	8	15
Running	0	0
All	25	135

Action/Trigger	Status	Count
Scope	ActionFailed	5
Compose	BadRequest	2
Dead-letter_the_...	BadRequest	2
Complete_the_m...	BadRequest	2
Parse_JSON	ValidationFailed	2
Initialize_variable...	BadRequest	1
Compose_2	BadRequest	1

To start working with OMS and Logic Apps, see the Monitor and get insights about logic app runs with Log Analytics in Microsoft docs and Middleware Friday episode Leveraging the Logic Apps Management solution for Logs Analytics.

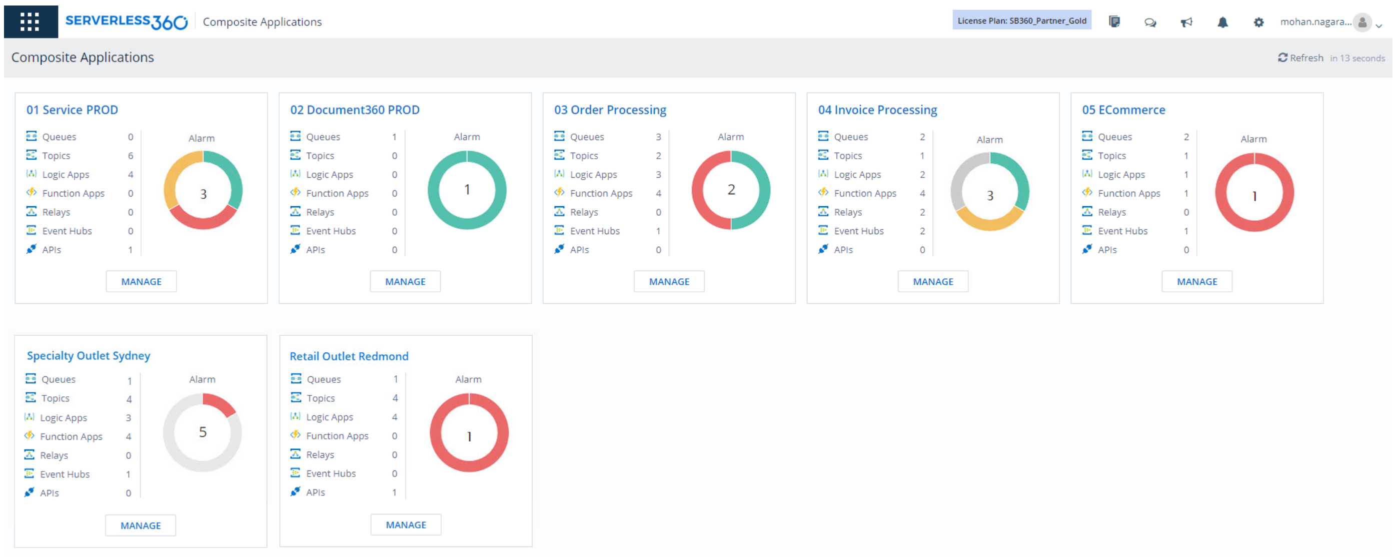
Another monitoring solution is Azure Monitor, which provides a base-level infrastructure metrics and logs for most services in Microsoft Azure. Currently, not all Azure services have Azure Monitor, but will be soon. The solution offers the following capabilities:

- **Activity Log**: information about all types of an event occurring within an Azure subscription – for instance, VM activities such as starting or stopping of a VM. Furthermore, the events will be stored for a max period of 90 days. You can query the events, save and pin to the Azure Dashboard. Also, you can push it to a storage account for a more extended period or to Event Hubs for real-time streaming or Log Analytics.
- **Diagnostic Settings**: information about events occurring inside a particular resource in an Azure subscription – for instance retrieving a secret from Key Vault. By default, these events will not be collected unless you enable them manually inside a resource (Portal), inside an ARM Template, using either PowerShell or REST API.
- **Metrics**: time-based metric points of your Azure resources similar to performance counters in a Windows Server. These metrics are available by default and have a retention period of 90 days. Furthermore, you can examine the performance of an Azure resource and track used or available credits. Also, you can push the metrics to Event Hubs, Stream Analytics, and retrieve and query metric data using PowerShell or REST API.

- **Alerts**: the alerts section, accessible in various Azure Resources, is where you can view and manage all Azure alerts. Alerts coming from the Activity Log, Metrics, Application Insights, and Log Analytics are visible here. Furthermore, you can create alert rules, which you send out via email, SMS, WebHook, or to a third party IT Service Management (TISM) application. Also, you can call an Automation Runbook if you want.

In the blade of a Logic App, you can find the Azure monitor features, which you can leverage for your Logic App.

An alternative or third-party solution you can leverage is Serverless360, a SaaS solution which allows you to manage and monitor composite cloud-native solutions at one place. This solution monitors your Azure integration services like Logic Apps, Functions, Event Hubs, Service Bus, and API endpoints.



	01 Service PROD	02 Document360 PROD	03 Order Processing	04 Invoice Processing	05 ECommerce	Specialty Outlet Sydney	Retail Outlet Redmond
Queues	0	1	3	2	2	1	1
Topics	6	0	2	1	1	4	4
Logic Apps	4	0	3	2	1	3	4
Function Apps	0	0	4	4	1	4	0
Relays	0	0	0	2	0	0	0
Event Hubs	0	0	1	2	1	1	0
APIs	1	0	0	0	0	0	1
Alarm	3	1	2	3	1	5	1

Cost Estimation

In every design with Logic App, you will have to look at costs. With Logic Apps the costs are determined by usage of the triggers and actions. Execution of a trigger and actions is charged – even retries are charged. Running Logic Apps is consumption based – thus the model is 'pay-as-you-go'.

The pricing according to pricing details from Microsoft is:

Every time a Logic App definition runs the triggers, action, and connector executions are metered.

	Price Per Execution
Actions	€0.000022
Standard Connector	€0.000106
Enterprise Connector	€0.000844

Data retention: €0.11 GB/month

For more details on connector see Microsoft documents - Connectors for Azure Logic Apps.

Another aspect of pricing you need to consider when using Logic Apps is the integration with the Integration Account in case that applies to your solution. With the Integration account, you can take advantage of Logic Apps B2B / EDI and XML processing capabilities. The integration account has two tiers – **basic** and **standard**.

	Basic	Standard
EDI Trading Agreements	1	500
EDI Trading Partners	2	500
Maps	500	500
Schemas	500	500
Assemblies	25	50
Certificates	2	500
Batch Configs	1	50
Price per hour	€0.35	€1.16

Assume a Logic App with a built-in connector (schedule), followed by 10 actions, including mapping actions – thus including an integration account (basic) running 100 times a day results in the following monthly costs:

- 10 actions X €0.000022 X 100 (frequency) X 30 (days) = €0.66
- 1 trigger (built-in) X €0.000106 X 100 (frequency) X 30 (days) = €0.318
- 1 basic integration account X 24 (hours) X €0.35 X 30 (days) = €252

TCO for one of such a Logic App is €253. The initial cost of an integration account is high for one Logic App and increases when you start to ramp up to the number of Logic Apps. Assume you will have 20 Logic Apps and the combined number of triggers is 30, some actions around 60, and the number of executions is 10000, and a standard integration account – the monthly costs are:

- 10000 actions X €0.000022 X 30 (days) = €6.6
- 1000 trigger executions X €0.000106 X 30 (days) = €3.18
- 1 standard integration account X 24 (hours) X €1.16 X 30 (days) = €835.2

The TCO for such a scenario is ~ €846 – around 10K per year. These are operational costs for your Logic Apps (OPEX). These cost estimations exclude a charge of monitoring and managing the Logic Apps.

Summary

In this paper, we discussed various patterns and best practices regarding Logic Apps. Logic Apps have matured over time to a first class service on the Azure Platform. Furthermore, the service is a first-class citizen in the Microsoft Cloud Integration Offering (Azure Integration Services) and a leader in the iPaaS Magic Quadrant of Gartner.

These patterns and practices have a developer focus. Once you establish the need for a Logic App in your cloud integration solution the patterns and practices described in this paper will be of value to you. Most of them include links for more details or Middleware Friday videos explaining the pattern or practice. With this paper, we hope you have a good view and understanding of the capabilities of Logic Apps and reference for your future Logic App solution designs.

About Serverless360

Serverless360 is a one platform tool to operate, manage and monitor Azure Serverless components. It provides efficient tooling that is not and likely to be not available in Azure Portal. Manageability is one of the key challenges with Serverless implementations. Hundreds of small, discrete Serverless functionalities are scattered in various places – managing and operating such solutions is complex. Serverless360 solves these challenges with a rich set of sophisticated tools.

Start your free trial now

TRY IT FOR FREE

SERVERLESS360

Visit www.serverless360.com for more information.