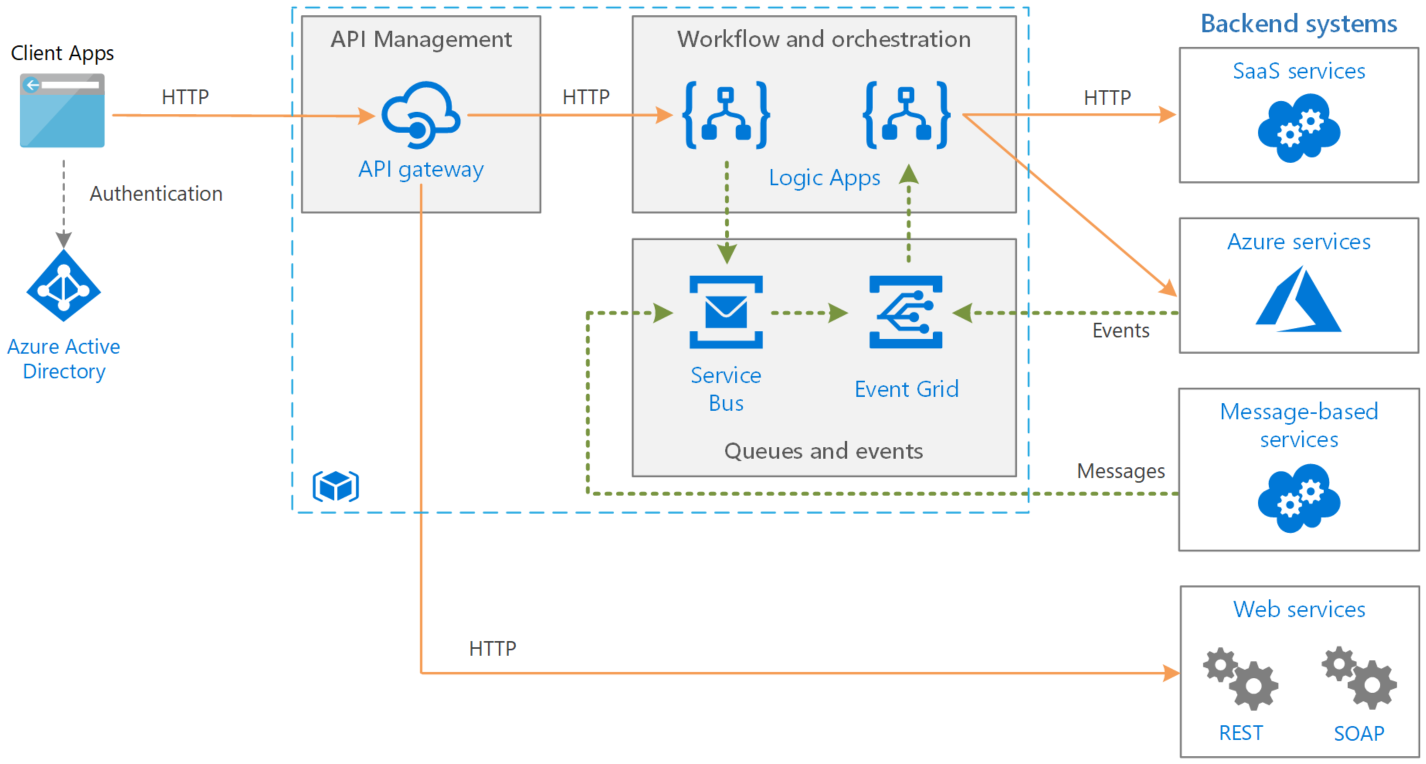
# Enterprise integration on Azure using message queues and events

12/03/2018 • 4 minutes to read • Contributors

**In this article**

Architecture
Recommendations
Scalability considerations
Availability considerations
Security considerations

This reference architecture integrates enterprise backend systems, using message queues and events to decouple services for greater scalability and reliability. The backend systems may include software as a service (SaaS) systems, Azure services, and existing web services in your enterprise.



## Architecture

The architecture shown here builds on a simpler architecture that is shown in Basic enterprise integration. That architecture uses Logic Apps to orchestrate workflows and API Management to create catalogs of APIs.

This version of the architecture adds two components that help make the system more reliable and scalable:

- **Azure Service Bus**. Service Bus is a secure, reliable message broker.
- **Azure Event Grid**. Event Grid is an event routing service. It uses a publish/subscribe (pub/sub) eventing model.

Asynchronous communication using a message broker provides a number of advantages over making direct, synchronous calls to backend services:

- Provides load-leveling to handle bursts in workloads, using the Queue-Based Load Leveling pattern.
- Reliably tracks the progress of long-running workflows that involve multiple steps or multiple applications.

- Helps to decouple applications.
- Integrates with existing message-based systems.
- Allows work to be queued when a backend system is not available.

Event Grid enables the various components in the system to react to events as they happen, rather than relying on polling or scheduled tasks. As with a message queue, it helps decouple applications and services. An application or service can publish events, and any interested subscribers will be notified. New subscribers can be added without updating the sender.

Many Azure services support sending events to Event Grid. For example, a logic app can listen for an event when new files are added to a blob store. This pattern enables reactive workflows, where uploading a file or putting a message on a queue kicks off a series of processes. The processes might be executed in parallel or in a specific sequence.

## Recommendations

The recommendations described in Basic enterprise integration apply to this architecture. The following recommendations also apply:

### Service Bus

Service Bus has two delivery modes, *pull* or *push*. In the pull model, the receiver continuously polls for new messages. Polling can be inefficient, especially if you have many queues that each receive a few messages, or if there a lot of time between messages. In the push model, Service Bus sends an event through Event Grid when there are new messages. The receiver subscribes to the event. When the event is triggered, the receiver pulls the next batch of messages from Service Bus.

When you create a logic app to consume Service Bus messages, we recommend using the push model with Event Grid integration. It's often more cost efficient, because the logic app doesn't need to poll Service Bus. For more information, see Azure Service Bus to Event Grid integration overview. Currently, Service Bus Premium tier is required for Event Grid notifications.

Use PeekLock for accessing a group of messages. When you use PeekLock, the logic app can perform steps to validate each message before completing or abandoning the message. This approach protects against accidental message loss.

### Event Grid

When an Event Grid trigger fires, it means *at least one* event happened. For example, when a logic app gets an Event Grid triggers for a Service Bus message, it should assume that several messages might be available to process.

Event Grid uses a serverless model. Billing is calculated based on the number of operations (event executions). For more information, see Event Grid pricing. Currently, there are no tier considerations for Event Grid.

## Scalability considerations

To achieve higher scalability, the Service Bus Premium tier can scale out the number of messaging units. Premium tier configurations can have one, two, or four messaging units. For more information about scaling Service Bus, see Best practices for performance improvements by using Service Bus Messaging.

## Availability considerations

Review the SLA for each service:

- API Management SLA
- Event Grid SLA

- Logic Apps SLA
- Service Bus SLA

To enable failover if a serious outage occurs, consider implementing geo-disaster recovery in Service Bus Premium. For more information, see Azure Service Bus geo-disaster recovery.

## Security considerations

To secure Service Bus, use shared access signature (SAS). You can grant a user access to Service Bus resources with specific rights by using SAS authentication. For more information, see Service Bus authentication and authorization.

If you need to expose a Service Bus queue as an HTTP endpoint, for example, to post new messages, use API Management to secure the queue by fronting the endpoint. You can then secure the endpoint with certificates or OAuth authentication as appropriate. The easiest way to secure an endpoint is using a logic app with an HTTP request/response trigger as an intermediary.

The Event Grid service secures event delivery through a validation code. If you use Logic Apps to consume the event, validation is automatically performed. For more information, see Event Grid security and authentication.