
agap2IT
TRAINING CENTER
Domain Driven Design
Module 6
Application Patterns

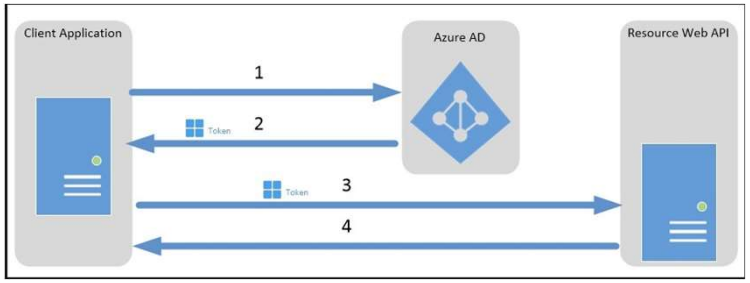
Domain Driven Design

Module Overview

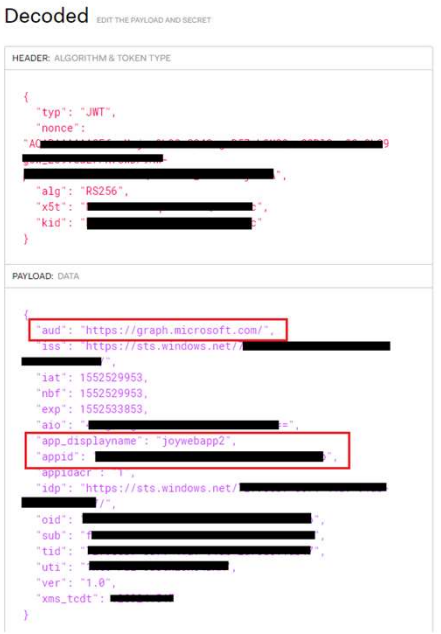
- Authentication Pattern
- Sync vs Async Communication
- API Gateway Pattern
- API Gateway Routing and Offloading Pattern
- Backends for Frontends Pattern
- Message Queuing
- Publish/Subscribe Communication
- Event-Driven Architecture

Authentication Pattern

“Application Integration: AAD integrates with a wide range of applications and services, including Microsoft 365, Azure services, and thousands of third-party applications. It supports standard protocols such as OAuth 2.0 and OpenID Connect, enabling seamless integration and secure access to these resources.”



Step	Description
1	Client Application sent the System Credentials to Azure AAD who will validate this information and generate a valid token (valid within some time)
2	Azure AAD respond to Client Application the generated token
3	Client Application, finally can request some endpoint in APIM passing the token as a header request
4	If the token is valid, the resource Web API will respond the requested data



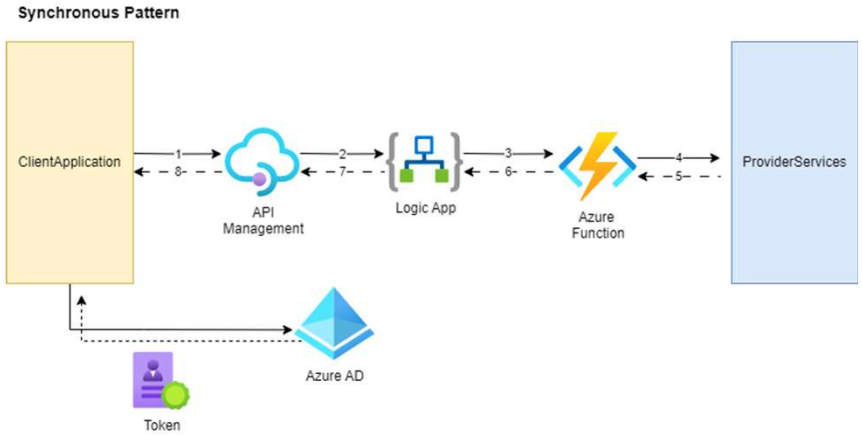
Synchronous vs asynchronous communication

“In the context of domain-driven design (DDD), synchronous and asynchronous communication refer to different ways of exchanging information and coordinating actions between software components or bounded contexts.”

“In DDD, the choice between synchronous and asynchronous communication depends on the specific context and requirements of the system. It's important to consider factors such as performance, scalability, fault tolerance, coupling, and the nature of the business domain. Often, a combination of both communication styles may be employed within different bounded contexts or subsystems to achieve the desired system characteristics.”

Synchronous Pattern

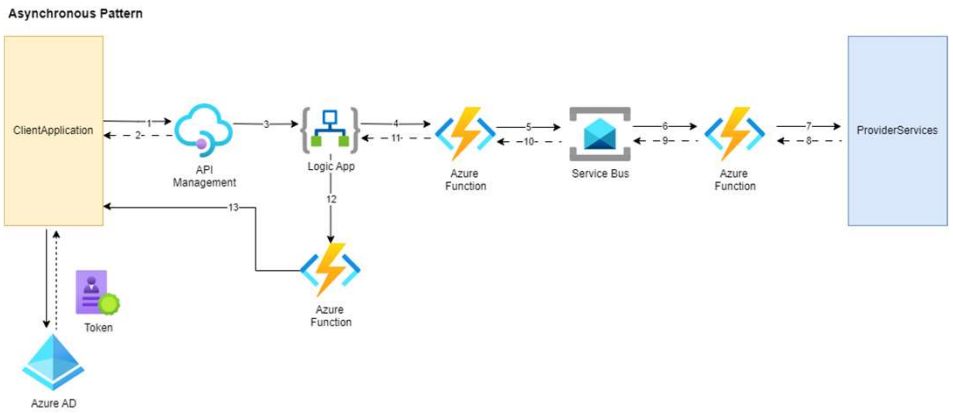
“Synchronous communication is a straightforward approach where a component or service makes a request to another component and waits for a response before proceeding further. It typically involves blocking, where the caller pauses execution until it receives a reply. Synchronous communication is often achieved through method calls, remote procedure calls (RPC), or synchronous HTTP requests.”



Step	Description
1	The API Management receives an HTTP request from the ClientApplication, validates the OAuth 2.0 Token received in the Authorization header, and forwards the received request to the Logic App.
2	An HTTP request from the API Management triggers the Logic App workflow that forwards the received request to the Azure Function.
3	An HTTP request from the Logic App triggers the Azure Function that computes the HMAC or prepare the adapter token using the API secret.
4	The received request is forwarded to the ProviderServices endpoint alongside the previously computed HMAC token and the API key.
5	The Azure Function receives the ProviderServices response and ignores some of the returned values to simplify the response that is sent to ClientApplication (another adapter or an ACL).
6	The Azure Function forwards the response to the Logic App.
7	The Logic App forwards the received response to the API Management.
8	The API Management forwards the received response to ClientApplication.

Asynchronous Pattern

“Asynchronous communication is a decoupled approach where the caller does not wait for an immediate response but continues with its execution. The caller sends a message to the receiver, which can process it at its own pace. Asynchronous communication can be implemented using message queues, publish-subscribe patterns, event-driven architectures, or asynchronous HTTP requests.”



Step	Description
1	The API Management receives an HTTP request from the ClientApplication, validates the OAuth 2.0 Token received in the Authorization header, and forwards the received request to the Logic App.
2	The API respond to Client Application with an HTTP 202 - Accepted
3	An HTTP request from the API Management triggers the Logic App workflow that forwards the received request to the Azure Function.
4, 5	The Logic App workflow returns a status code 202 (accepted), if all the process went as expected. If there was an error during the previous steps the status code returned could be 400 or 500, depending on the origin of the error, and the process would finish. Up to this step the behaviour has been synchronous. Now that the message is in the queue, waiting to be processed asynchronously, any response will be sent back through the callback endpoint.
6, 7, 8, 9,10,11	Messages in the Service Bus Queue trigger the second workflow of the Logic App, which orchestrates the interactions with ProviderServices
12	The Azure Function receives the ProviderServices response and ignores some of the returned values to simplify the response that is sent to ClientApplication (another adapter or an ACL).
13	The Azure Function forwards the response to the ClientApplication.

Synchronous vs Asynchronous

“Should the service communicate synchronously or asynchronously?”

Synchronous…

- The client sends a request and waits for a response from the service
- While waiting, the thread may or may not be blocked
- The client can only continue when it receives the response

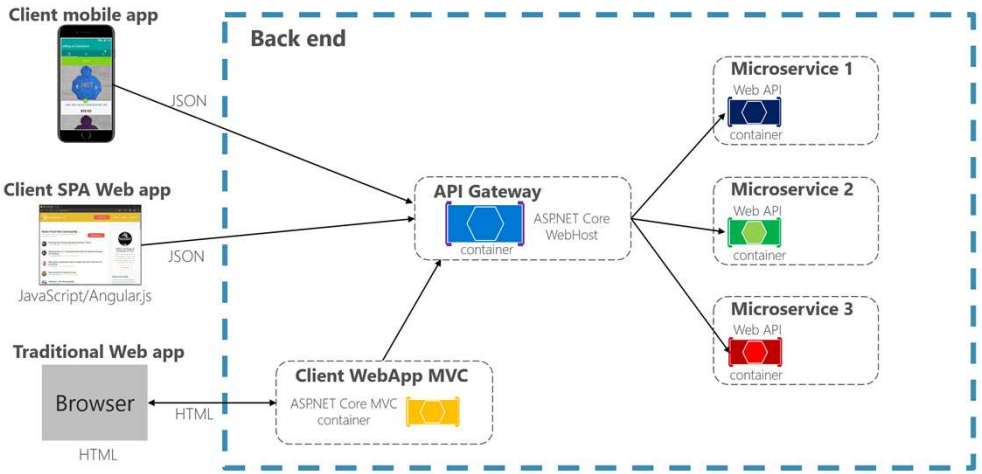
Asynchronous…

- The client sends a request to an intermediary message broker
- It does not wait for a response

API Gateway Pattern

“In the context of domain-driven design (DDD), the API Gateway pattern is a design pattern that serves as a mediator or entry point for client applications to interact with the backend services of a system. It acts as a single point of entry for all client requests, providing a unified interface and encapsulating the complexities of the underlying microservices or bounded contexts.”

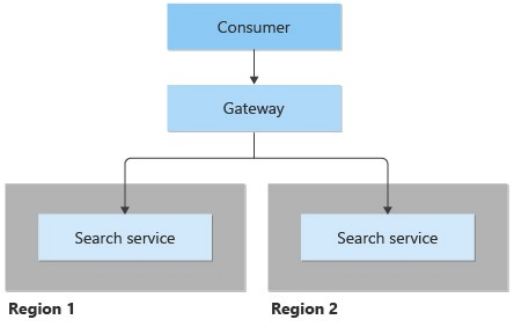
Using a single custom **API Gateway service**

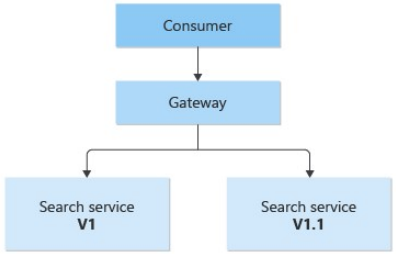


#	Advantages
1	Front-end service encapsulate core backend services
2	Exposes single point of entry
3	Routes requests to backend services
4	Can fan requests across multiple backend services
5	Isolate ClientApplication from internal partitioning and refactoring

API Gateway Routing and Offloading Pattern

"API Gateway Routing and Offloading pattern is an extension of the API Gateway pattern that focuses on intelligent request routing and offloading processing tasks from backend services to the API Gateway. It aims to optimize the performance, scalability, and resilience of the system by leveraging the capabilities of the gateway."

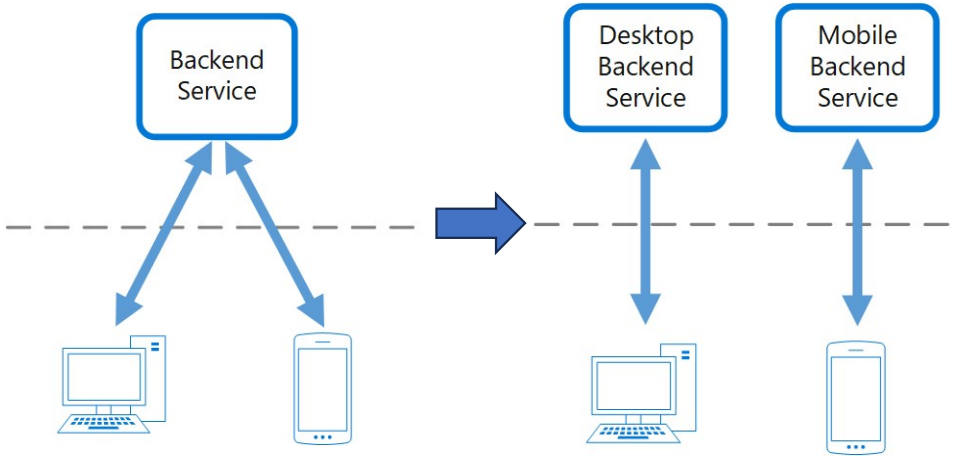




#	Advantages
1	It routes the requests to the appropriate backend services or microservices, ensuring that each request is directed to the most suitable destination.
2	The API Gateway can distribute incoming requests across multiple instances of backend services to achieve load balancing
3	Can offload certain processing tasks from backend services to reduce their workload and improve overall system performance
4	Can implement caching mechanisms to cache responses from backend services and serve subsequent requests directly from the cache
5	Can perform protocol translation, enabling clients to use different communication protocols while internally converting them to the protocol understood by backend services
6	Can enhance the fault tolerance and resilience of the system by implementing

Backends for Frontends Pattern

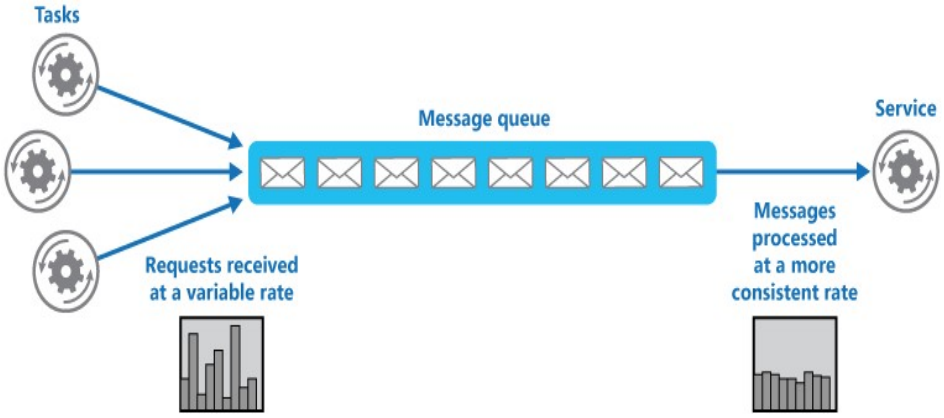
"Backends for Frontends (BFF) pattern is an architectural approach that addresses the complexities of building user interfaces (frontends) that require different types of backend services to support their specific needs. The BFF pattern promotes the idea of tailoring backend services to optimize the user experience and enhance development efficiency."



#	Advantages
1	With the BFF pattern, each frontend or group of frontends has its own dedicated backend service.
2	The BFF pattern encourages the development of backend services that align with the domain of the frontend.
3	Can optimize data retrieval and processing to enhance performance
4	The architecture becomes more modular and easier to scale
5	Different frontends may have varying security requirements, and by having dedicated BFFs, it becomes easier to manage authentication, authorization, and other security-related concerns specific to each frontend.

Message Queing

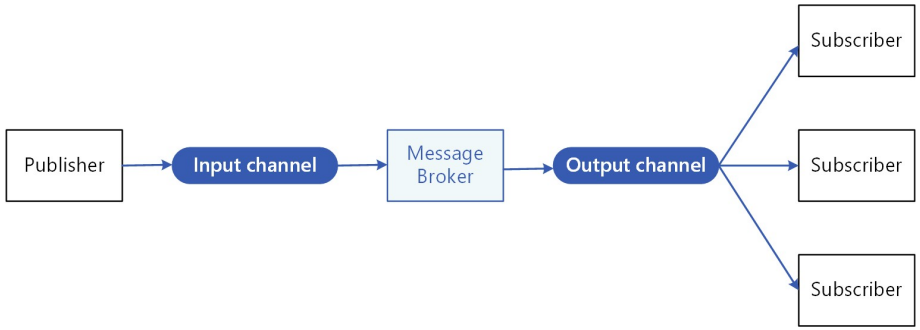
"The queue decouples the tasks from the service, and the service can handle the messages at its own pace regardless of the volume of requests from concurrent tasks. Additionally, there's no delay to a task if the service isn't available at the time, it posts a message to the queue."



#	Advantages
1	It can help to maximize availability because delays arising in services won't have an immediate and direct impact on the application, which can continue to post messages to the queue even when the service isn't available or isn't currently processing messages.
2	It can help to maximize scalability because both the number of queues and the number of services can be varied to meet demand.
3	It can help to control costs because the number of service instances deployed only have to be adequate to meet average load rather than the peak load.
4	Some services implement throttling when demand reaches a threshold beyond which the system could fail. Throttling can reduce the functionality available. You can implement load leveling with these services to ensure that this threshold isn't reached.

Publish/Subscribe communication

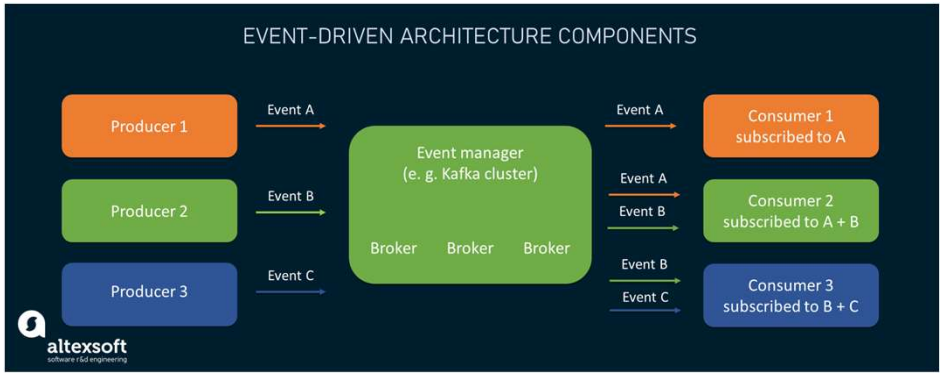
"In the context of domain-driven design (DDD), the Publish/Subscribe communication pattern is a messaging pattern that facilitates loose coupling and asynchronous communication between components or bounded contexts. It enables the broadcasting of messages from publishers to multiple subscribers without the publishers needing to have explicit knowledge of the subscribers."



#	Advantages
1	It decouples subsystems that still need to communicate. Subsystems can be managed independently, and messages can be properly managed even if one or more receivers are offline.
2	It increases scalability and improves responsiveness of the sender. The sender can quickly send a single message to the input channel, then return to its core processing responsibilities. The messaging infrastructure is responsible for ensuring messages are delivered to interested subscribers.
3	It improves reliability. Asynchronous messaging helps applications continue to run smoothly under increased loads and handle intermittent failures more effectively.
4	It allows for deferred or scheduled processing. Subscribers can wait to pick up messages until off-peak hours, or messages can be routed or processed according to a specific schedule.
5	It enables simpler integration between systems using different platforms, programming languages, or communication protocols, as well as between on-premises systems and applications running in the cloud.
6	It facilitates asynchronous workflows across an enterprise.
7	It improves testability. Channels can be monitored and messages can be inspected or logged as part of an overall integration test strategy.
8	It provides separation of concerns for your applications. Each application can focus on its core capabilities, while the messaging infrastructure handles everything required to reliably route messages to multiple consumers.

Event-Driven architecture

"In an event-driven architecture, components are loosely coupled and interact with each other by producing and consuming events. Events are immutable, time-stamped messages that carry information about a specific occurrence or state change. They represent facts or notifications about something that has happened in the system or the domain."



#	Advantages
1	Components are decoupled, as they interact through events without direct knowledge of each other. Components can evolve independently, making it easier to introduce new functionalities or modify existing ones without impacting the entire system.
2	Event-driven architecture enables horizontal scalability by distributing event processing across multiple consumers. Components can scale individually based on their specific needs and handle events concurrently, leading to improved performance and responsiveness.
3	Event-driven systems are inherently flexible and extensible. New components can be easily added by subscribing to relevant events, and existing components can be modified or replaced without affecting the overall system.
4	Event-driven architectures often embrace eventual consistency, where different components eventually reach a consistent state by processing events in an asynchronous manner. This allows for handling complex, distributed business processes and enables fault tolerance and resilience.

Who else needs some coffee ?

Domain Driven Design

Technical Training

Let’s do it