**Microservices and things that come with it (Functional Decomposition)**

**Benefits:**

1. Each service can be scaled independently with a different tech stack – polyglot
2. Ideally different data stores whose schema can be changed without affecting other services. Doesn’t happen all the time in practice.
3. Loosely coupled – boundaries between services are stronger than components in the same process. Overtime Monolith leads to `spaghetti code`. Example, usage of shared libraries could be much more rampant with monliths. Updating class models in shared libraries could affect multiple services. Also, same data store. In practice, multiple components lead to using same tables so it gets difficult to change schemas etc.
4. Human benefit – Less code of individual services maintained by smaller teams

**Disadvantages or Considerations:**

1. Eventual Consistency: Different data stores for microservices. Atomically updating records in such a scenario – distributed transactions that offer strong consistency are slow and expensive. Embracing eventual consistency.
2. Debugging becomes harder. You can’t just load an application and step through it with a debugger. Distributed tracing is required because of more moving parts.
3. CI, CD pipeline is necessary. Testing the integration of all the microservices as one system is harder. Challenges when they interact with each other.
4. Communication: Remote calls, a lot of them. Need to look into async calls and batching.

**API Gateway:**

* **Why do we need one:** A client needs to be aware of the services and their DNS names. What if those services need to make changes? This would cause clients to be upgraded? How do you force apps on mobiles to be updated? API gateway acts as façade for internal services. Public API which rarely change.
* **Features:**
  + **Routing**: Maps external API to internal ones. If internal path changes, continue to expose the old public path for backward compatibility. Example, krakend in GNS has 1:1 mapping.

|  |
| --- |
| {  **"endpoint": "/api/domainparking/{level1}",**  "method": "GET",  "output\_encoding": "no-op",  "concurrent\_calls": 1,  "querystring\_params": [  "\*"  ],  "headers\_to\_pass": [  "Cookie",  "xtoken",  "username",  "Content-Type"  ],  **"backend": [**  **{**  **"host": [**  **"{{ .env.domainparking\_url }}**" => this resolves to the IP and PORT of domain parking Kubernetes service |

* + **Composition:** A public API that calls multiple internal services, composes their responses and returns it to the client. Reduces the number of requests to be sent by the client.
  + **Translation:**
    - Translate a HTTP Rest request call to internal gRPC call.
    - Different APIs for different use cases or devices and then translate those different APIs to internal ones intelligently.
    - Graph based APIs: A *graph-based API* exposes a schema composed of types, fields, and relationships across types. The API allows a client to declare what data it needs and let the gateway figure out how to translate the request into a series of internal API calls.
  + **Cross Cutting Concerns:** Implement functionalities that would otherwise need to be reimplemented in each service:
    - Caching frequently accessed resources
    - Rate limiting
    - Authentication: Since it’s the point of entry. Centralized logic to support different authentication mechanisms. Authorization is left to the internal service to prevent API gateway be aware of the domain logic of the services. It prevents coupling. Authz: Token Based (OAuth2.0 JWT) and API-Key based (Fastly-Key with Domain Parking). With API Key, the application server needs to look into DB to validate the API Key value for every request. JWT is faster since it can be validated without needing to call an external service as it’s signed by a trusted entity.

Diagram

Description automatically generated

* **Disadvantages:**
  + Every new service needs to be hooked up with it
  + Whenever an internal service changes, API Gateway changes are needed
  + API Gateway needs to scale as well.

**Messaging:**

1. **Why do we need it?**

* As the number of services increases, network calls increase, probability that a destination could be momentarily available. Messaging provides the necessary buffer for messages would could otherwise be lost if we rely on synchronous request response behaviour.
* Client can execute operations on a service asynchronously. Particularly useful for operations that take a long time. For example, conversion of a video to multiple formats. Clients just write a message to the channel and move on. There is no need for client to wait on the service to acknowledge the completion of the conversion.
* Consumers can read messages at their own pace without getting overloaded– useful for spike
* Load balancing of messages across a pool consumers which can be dynamically added or removed. For example, in kafka if a new consumer is added to the existing groups, it takes some partitions off other consumers.2.
* Producers and Consumers are decoupled. Producers don’t need to know the addresses of consumers.

1. **Disadvantages?**
   1. Higher latency because of an additional hop and more so if there is a backlog in the channel.
   2. More system complexity as one more system needs to be managed – message broker.
2. **Types of Message Channels:**
   1. **Point to Point**: A specific message is delivered to one consumer only
   2. **Publish Subscribe**: A copy of message is delivered to all consumers.
3. **Communication Styles:**
   1. **One way messaging:** Point to Point channel. Producer is not interested in the response. It just hopes for the consumer to process it.
   2. **Request Response Messaging:** Consumer has a point to point request channel. Every producer has a separate response channel. Each request has a request ID and a reference to the producer channel. Consumer writes responses to the producer channel using the reference ID present in the request message and tags the response with the requestId
   3. **Broadcast:** One way Broadcast. Publish Subscribe Channel.
4. **Tradeoffs or guarantees that a message broker provides:**
   1. **Delivery guarantees** like atleast once or atmost once. Exactly once is not possible ideally. Consumer deletes the message from the channel after processing it to prevent it from being read by other consumers. When to delete? If deleted right after reception and before processing it, message can be lost forever if the consumer crashes before completing the processing. If deleted after processing, there is a risk that the consumer crashes after processing but before deleting the message from the channel. In this case, the message could be read again. Best you can do is simulate exactly once processing by requiring messages to be idempotent.
   2. **Latency**
   3. **Message durability guarantees.** Example**:** When a message is being processed by the consumer, it remains in the channel but other consumers can’t see it for the duration of visibility timeout. Why is there a timeout? If the consumer crashes during processing, the message can eventually be visible and read by other consumers.
   4. **Maximum supported size of messages**
   5. **Ordering guarantees:** Amazon SQS doesn’t provide ordering guarantees. Kafka provides ordering at partition level. Since a single node holds all the messages of a partition, it is trivial to provide ordering guarantees.
5. **How to deal with Failures?**
   1. If the consumer fails to process a message, the visibility timeout triggers and it can be processed by other consumers
   2. If there is some problem with message and it repeatedly fails while processing, limit the retries. Broker can stamp the messages with a counter that indicates the number of times a message has been delivered to a consumer. If the threshold is reached, consumer can delete the message from channel and add it to some low priority channel which acts as buffer for such error prone messages which can then be inspected later.
6. **How to deal with Backlogs?**
   1. Backlogs build when the rate at which consumers process message is less than the rate of arrival of messages. More producers can come online, some messages took so long to get processed, retries of certain messages before they end up in the “dead letter” channel
   2. Detect backlogs early. Measure the average time a message waits in the channel before being read. Brokers can attach a timestamp to message when the message arrives. If backlog is detected, scale the number of consumers in consumer group.
7. **Fault Isolation, how to deal with faulty producers?** Some producers can degrade the whole system. If messages from them fail most of the time, it can build backlogs in the channel since the error prone messages are retried multiple times before they end up in the dead letter channel. It’s important to identify such producers. Broker can stamp the messages with the identifier of producers. If they are identified as belonging to some faulty producers, consumers can write them to some other low priority channel without processing them. Consumers can later read them from such a channel with less frequency.
8. **How to deal with blobs?** Messages in a channel have a size limit**.** Azure Storage queues limit messages to 64 KB, AWS Kinesis to 1 MB, etc Store them in a blob storage like S3. Send the URL of the blob with the message.