Azure Cloud Architecture Design Pattern

| **Pattern Name** | **Comment** |
| --- | --- |
| **Ambassador** | A proxy service to handle some generic and common utility task like:   * Circle Breaking * Retry * Monitoring * Security * Logging   Diagram of the Ambassador pattern |
| **Anti-corruption Layer** | Loose coupling modern system with legacy system by providing a façade to isolate them in order to prevent legacy system to “pollute” the modern system  Diagram of the Anti-Corruption Layer pattern |
| **Asynchronous Request-Reply** | Turn time consuming synchronous Api call into polling an Asynchronous Api call using HTTP response status code |
| **Backends for Fronts** | Create separate backend services for different front-end applications due to UI and reactive consideration, e.g., different backend APIs for Desktop and Mobile.  Context-and-problem diagram of the Backends for Frontends pattern |
| **Bulkhead** | Bulkhead is identical to create an **isolation** and **redundant** protection in case of failure, it includes:   * Isolation, divide service to different partitions (connection pools) based on their workloads etc. * Redundant, create on-demand connection pools to handle workload balancing.  |  |  | | --- | --- | | First diagram of the Bulkhead pattern | Diagram showing multiple clients calling a single service. | |
| **Cache-Aside** | Cache non-transient data can improve performance  Using the Cache-Aside pattern to store data in the cache |
| **Choreography** | Each component of a system tries to communicate with each other without a centralized Orchestrator by using **Asynchronous Messaging Pattern** (also called **Publisher-Subscriber Pattern** or **Broadcast** or Azure **Topic**)  Processing a request using a central orchestrator  Processing a request using a choreographer  Publish-subscribe pattern using a message broker |
| **Circuit Breaker** | It works with **Retry Pattern** and manages the following 3 states:   * **Closed**: operation normal, return result * **Open**: operation failed, return failure or exception * **Half-Open**: Retry in a limited scale to see whether to switch to Closed or Open   Circuit Breaker states |
| **Claim Check** | Message holds data, but split large message into a claim check (message id) and the big payload, just send the claim-check to the messaging platform (e.g., Azure Service Bus) and store the payload to an external service (e.g., database) and later claim back the payload using claim check (the message id) to re-construct the original message data.  Diagram of the Claim-Check pattern. |
| **Compensating Transaction** | The difference between **Traditional Transaction** and **Compensting Transaction** is: Compensating Transaction is not Transactional at all, there is no 2 phase commit at all, it is just an eventual consistency model - a typical business operation consists of a series of separate steps. While the operation performs these steps, the overall view of the system state might be inconsistent. But when the operation finishes and all the steps have run, the system should become consistent again.  Diagram that shows the steps for creating an itinerary. The steps of the compensating transaction that cancels the itinerary are also shown.  The compensation logic seems case by case, better to use a local database to lock and store each transaction step and later call an idempotent method to guarantee commit.   * **Traditional Transaction** is if I say I do then I will do for sure. It uses locking mechanism. * **Transaction Compensation** is I will do first and rewind (**undo**) in case of failure. It does not lock, so there will be concurrent conflicts. Similar to **Saga Pattern**. |
| **Saga** | A saga is a sequence of local transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, the saga executes compensating transactions that counteract the preceding transactions.  It works like commit one transactional step at one time and continue to next step until it completes or rollback in a First In Last Out (stack) pattern.    Because Saga has a Local Transaction in Database so it’s safer than **Compensating Transaction**.  Here is an example of Bank Transfer (Credit and Debit) and Sending Receipt Saga Transaction workflow. Notice that the Saga Orchestrator is a durable Function App which means it saves the state into a durable storage like database. |
| **Competing Consumer** | Count on Azure cloud platform to automatically to create Azure Function App Instances to auto scale out competing message consumers to pull messages simultaneously from Azure Service. The Azure Functions runtime receives a message in PeekLock mode, if the function finishes successfully, it calls Complete on the message, or it may call Abandon if the function fails, and the message will become visible again, allowing another consumer to retrieve it. If the function runs for a period longer than the PeekLock timeout, the lock is automatically renewed as long as the function is running.  Azure Functions can scale out/in based on the depth of the queue, all acting as competing consumers of the queue. If multiple instances of the functions are created, they all compete by independently pulling and processing the messages.  Using a message queue to distribute work to instances of a service |
| **Compute Resource Consolidation** | Group computing intensive resources together and host them in powerful cloud resources. This is more a configuration rather than programming pattern. |
| **CQRS** | **CQRS** means **Command and Query Responsibility Segregation**  That is to separate **Write** (Command) and **Read** (Query) data store in order to reduce contention (transactional locks, dead locks etc.)  CQRS often works with Event Sourcing pattern, where application state is stored as a sequence of events. Each event represents a set of changes to the data. The current state is constructed by replaying the events. In a CQRS context, one benefit of Event Sourcing is that the same events can be used to notify other components — in particular, to notify the read model. The read model uses the events to create a snapshot of the current state, which is more efficient for queries.  When used with the Event Sourcing pattern, the store of events is the write model, and is the official source of information. The read model of a CQRS-based system provides materialized views of the data, typically as highly denormalized views. These views are tailored to the interfaces and display requirements of the application, which helps to maximize both display and query performance.  Using the stream of events as the write store, rather than the actual data at a point in time, avoids update conflicts on a single aggregate and maximizes performance and scalability. The events can be used to asynchronously generate materialized views of the data that are used to populate the read store.  Because the event store is the official source of information, it is possible to delete the materialized views and replay all past events to create a new representation of the current state when the system evolves, or when the read model must change. The materialized views are in effect a durable read-only cache of the data.  When used with the Event Sourcing pattern, the store of events is the write model, and is the official source of information. The read model of a CQRS-based system provides materialized views of the data, typically as highly denormalized views. These views are tailored to the interfaces and display requirements of the application, which helps to maximize both display and query performance.  Using the stream of events as the write store, rather than the actual data at a point in time, avoids update conflicts on a single aggregate and maximizes performance and scalability. The events can be used to asynchronously generate materialized views of the data that are used to populate the read store.  Because the event store is the official source of information, it is possible to delete the materialized views and replay all past events to create a new representation of the current state when the system evolves, or when the read model must change. The materialized views are in effect a durable read-only cache of the data.When using CQRS combined with the Event Sourcing pattern, consider the following:  As with any system where the write and read stores are separate, systems based on **this pattern are only eventually consistent**. There will be some delay between the event being generated and the data store being updated.  The pattern adds complexity because code must be created to initiate and handle events, and assemble or update the appropriate views or objects required by queries or a read model. The complexity of the CQRS pattern when used with the Event Sourcing pattern can make a successful implementation more difficult, and requires a different approach to designing systems. However, event sourcing can make it easier to model the domain, and makes it easier to rebuild views or create new ones because the intent of the changes in the data is preserved.  Generating materialized views for use in the read model or projections of the data by replaying and handling the events for specific entities or collections of entities can require significant processing time and resource usage. This is especially true if it requires summation or analysis of values over long periods, because all the associated events might need to be examined. Resolve this by implementing snapshots of the data at scheduled intervals, such as a total count of the number of a specific action that has occurred, or the current state of an entity.  A CQRS architecture with separate read and write stores |
| **Event Sourcing** | Instead of storing just the current state of the data in a domain, use an **append-only store** to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects. This can simplify tasks in complex domains, by avoiding the need to synchronize the data model and the business domain, while improving performance, scalability, and responsiveness. It can also provide consistency for transactional data, and maintain full audit trails and history that can enable compensating actions.  Limitation of **CRUD**:   * CRUD systems perform update operations directly against a data store. These operations can slow down performance and responsiveness and can limit scalability, due to the processing overhead it requires. * In a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data. * Unless there's another auditing mechanism that records the details of each operation in a separate log, history is lost.   The events are persisted in an event store that acts as the system of record (the authoritative data source) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action, that's required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events. **Events are immutable**. **Event Sourcing is also eventual consistent**.  An overview and example of the Event Sourcing pattern |
| **Materialized View** | This is more focused on the Query/Read model, it’s similar to relational database view. It may be deformalized for speed and closed to required information. This pattern works closely with **CQRS** and **Event Sourcing** pattern.  Figure 1 shows an example of how the Materialized View pattern might be used |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |