Microservice Design Patterns

Neda Mohammadi

Latest update: August 2024

Introduction

This description of the microservice design pattern is based on our research and represents the results of our study in the field of microservice design patterns. If you find this information useful and wish to cite it, please refer to the following publication:

1. Mohammadi, N., & Rasoolzadegan, A. (2022). *A Pattern-aware Design and Implementation Guideline for Microservice-based Systems.* In *2022 27th International Computer Conference, Computer Society of Iran (CSICC)*, 1–6. IEEE.
2. Mohammadi, N., & Rasoolzadegan, A. & Hosseinpour, S., & Sabeghi, M. & Shahrestani. A. *BenchPDM : Benchmarking Pattern Detection Methods in Microservice-Based Systems Using Automatically Generated Pattern-Assisted Testbeds.*

For the description of this pattern, multiple references were consulted, including [1], [2], [4], and [3].

Contact Information

Email GitHub LinkedIn ORCID

1 Pattern Name: Service Discovery

1.1 Textual Description

The **Service Discovery** pattern addresses the challenge of dynamically locating microservices that are often deployed on virtual machines or containers with changing addresses. Given the dynamic nature of microservices, fixed access points cannot be defined, making it difficult for users and developers to access these services via APIs. The Service Discovery pattern introduces a centralized management approach through the **Service Registry** microservice, which maintains and updates the addresses of all microservices within a system.

1.2 Pattern Signature

1. **Roles**:

   - **User**: Needs a specific microservice to perform a function within an application.
   - **Worker Microservice**: Performs the functional tasks required by the system.
   - **Load Balancer Microservice**: Acts as a mediator between the user and the worker microservices. It provides the address of each worker microservice by querying the **Service Registry Management** microservice.
   - **Service Registry Management Microservice**: Stores and updates the addresses of all microservices in the system.

2. **Communication Flow**:

   - The user sends a request through the Load Balancer microservice using HTTP methods such as GET, POST, DELETE, and PUT.
   - The Load Balancer microservice retrieves the address of the appropriate Worker microservice by invoking the GET operation on the Service Registry Management microservice.
   - Once the address is obtained, the Load Balancer forwards the user's request to the relevant Worker microservice.

1.3 Formal and Mathematical Description

Let S denote the Service Discovery pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, n\}$
- **Set of Worker Microservices**: $W \subseteq M$
- **Set of Load Balancer Microservices**: $L \subseteq M$
- **Set of Service Registry Management Microservices**: $R \subseteq M$
- **Set of Users**: U

Define relations:

- $F : U \times W \to L$ where $F(u, w)$ represents the Load Balancer $l \in L$ responsible for handling user u's request to worker microservice w.
- $G : L \to R$ where $G(l)$ represents the Service Registry Management microservice $r \in R$ that Load Balancer l queries to obtain the address of worker microservice w.
- $H : R \to W$ where $H(r)$ provides the mapping between the Service Registry Management microservice r and the worker microservices w whose addresses it manages.

1.4 Pattern Properties

1. **Dynamic Addressing**:

   The Service Registry Management microservice $r \in R$ dynamically updates the addresses of worker microservices $w \in W$.

2. **Scalability**:

   The pattern supports scalable deployment as new worker microservices can be added to the Service Registry Management without disrupting the existing services.

3. **Fault Tolerance**:

   In case a worker microservice w changes its address or goes offline, the Service Registry Management updates its records, ensuring minimal disruption in service availability.

1.5 User-Load Balancer Communication

Each user $u \in U$ interacts with the Load Balancer microservice $l \in L$ via HTTP methods (GET, POST, PUT, DELETE) to request services. Formally, $\forall u \in U, \exists l \in L$ such that $u \to l$ where $\to$ denotes the communication.

1.6 Load Balancer-Service Registry Communication

Each Load Balancer $l \in L$ queries the Service Registry Management microservice $r \in R$ to retrieve the address of the appropriate worker microservice. This is represented as $\forall l \in L, \exists r \in R$ such that $l \to r$ and $r \to w$ where $w \in W$.

1.7 Service Registry-Worker Microservice Communication

The Service Registry Management microservice r maintains the addresses of all worker microservices w. This is represented as $H(r) = W$, meaning r knows the address of all $w \in W$.

1.8 Service Dependency Graph

The Service Dependency Graph (SDG) for the Service Discovery pattern is as Figure 1.

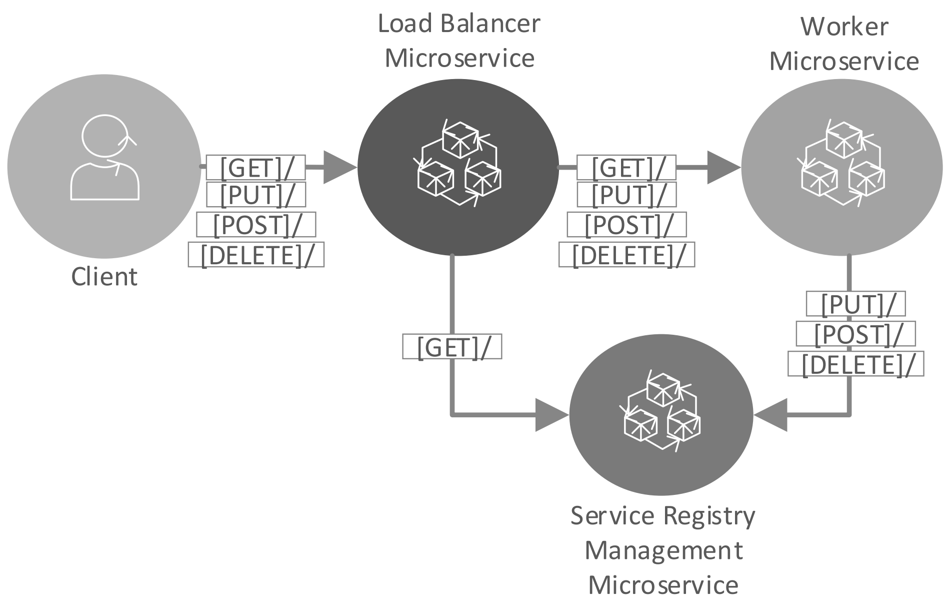

Figure 1: Dependency graph of the Service Discovery pattern.

2 Pattern Name: Event Sourcing

2.1 Textual Description

The **Event Sourcing** pattern introduces an approach to managing operations on data that is driven by a sequence of events. All occurred events are stored in an append-only database. In the approach introduced by the Event Sourcing pattern, a user or other services send requests to an **Event Manager** microservice. This microservice communicates with a **Database Manager** microservice to perform and satisfy the request, while simultaneously storing the occurred events in an append-only database via the **Event Database Manager** microservice.

2.2 Pattern Signature

1. **Roles**:
    - **User**: Sends requests for data operations.
    - **Event Manager Microservice**: Handles requests from users and coordinates with the Database Manager and Event Database Manager microservices.
    - **Event Database Manager Microservice**: Manages the append-only database where all events are stored.
    - **Database Manager Microservice**: Manages the database where the actual data is stored and retrieved.
2. **Databases**:
    - **Data Database**: Stores the actual data.
    - **Event Database**: Stores all events that occurred in the system.
3. **Communication**:
    - Users or other services interact with the Event Manager Microservice using HTTP methods such as GET, POST, DELETE, and PUT.
    - The Event Manager Microservice sends requests to the Database Manager Microservice using GET APIs to retrieve the required data.
    - Simultaneously, the Event Manager Microservice stores the occurred events in the Event Database via POST APIs provided by the Event Database Manager Microservice.

4. **Interactions**:
    - Users do not have direct access to the databases.
    - The Event Manager Microservice acts as a mediator between the User and the Databases.

2.3 Formal and Mathematical Description

Let ES denote the Event Sourcing pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, n\}$
- **Set of Event Manager Microservice**: $E \subseteq M$
- **Set of Database Manager Microservice**: $D \subseteq M$
- **Set of Event Database Manager Microservice**: $ED \subseteq M$
- **Set of Users**: U

Define a relation $R_{DB} \subseteq E \times D$ where $(e, d) \in R_{DB}$ indicates that the Event Manager Microservice e sends requests to the Database Manager Microservice d.

Define a relation $R_{ED} \subseteq E \times ED$ where $(e, ed) \in R_{ED}$ indicates that the Event Manager Microservice e stores events in the Event Database Manager Microservice ed.

2.4 Pattern Properties

1. **Event Storage**: All occurred events are stored in an append-only database managed by the Event Database Manager Microservice.
2. **Decoupled Data Management**: Users interact with the Event Manager Microservice which manages both the event logging and data retrieval processes.

2.5 User-Event Manager Communication

- For each User $u \in U$, the User interacts with the Event Manager Microservice $e \in E$ using HTTP methods $\text{GET}, \text{POST}, \text{PUT}, \text{DELETE}$. Formally, $\forall u \in U$, $\exists e \in E$ such that $u \to e$ where $\to$ denotes the communication for service requests.

2.6 Event Manager-Database Manager Communication

- Each Event Manager Microservice $e \in E$ sends data retrieval requests to the Database Manager Microservice $d \in D$. Formally, $\forall e \in E, \exists d \in D$ such that $(e, d) \in R_{DB}$.

2.7 Event Manager-Event Database Manager Communication

- Each Event Manager Microservice $e \in E$ stores events in the Event Database Manager Microservice $ed \in ED$. Formally, $\forall e \in E, \exists ed \in ED$ such that $(e, ed) \in R_{ED}$.

2.8 Service Dependency Graph

The Service Dependency Graph (SDG) for the Event Sourcing pattern is Figure 2.

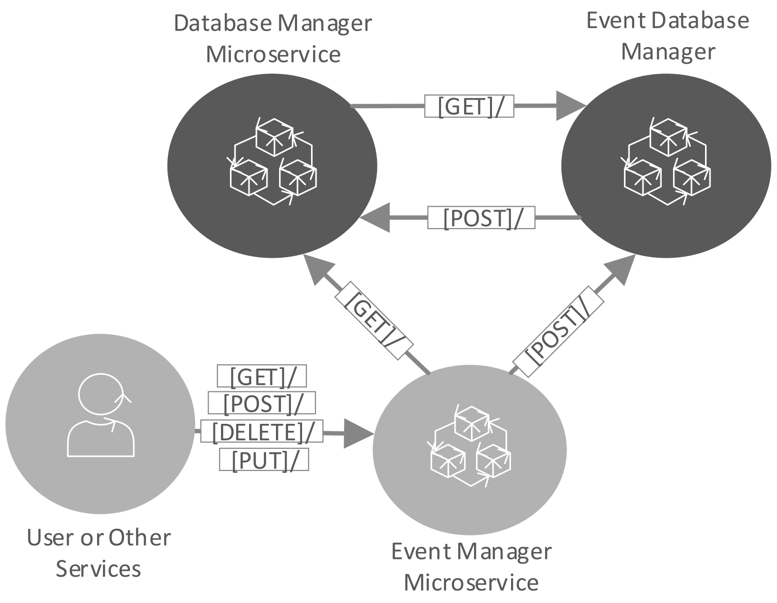

Figure 2: Dependency graph of the Event Sourcing pattern.

3 Pattern Name: API Gateway

3.1 Textual Description

The **API Gateway** pattern is generally similar to the aggregation pattern. It involves receiving user requests, identifying the required worker microservices, and invoking them, which are common management and non-functional tasks in both patterns. However, the API Gateway pattern provides additional fundamental capabilities such as routing, network address translation protocol (NATP), and others that are essential for many applications. The process starts with the user request. Upon receiving this request, the API Gateway microservice analyzes it, identifies the appropriate worker microservices needed to fulfill the request, and then routes the request to these worker microservices.

3.2 Pattern Signature

1. **Roles**:
   - **API Gateway Microservice**: Receives user requests, performs necessary operations such as routing, and forwards requests to worker microservices.
   - **Worker Microservices**: Execute tasks as requested by the API Gateway Microservice.
   - **User**: Initiates requests that are handled by the API Gateway Microservice.
2. **APIs**:
   - The API Gateway Microservice provides APIs for the user to interact using GET, POST, PUT, and DELETE methods.
   - Worker Microservices also provide APIs using GET, POST, PUT, and DELETE methods, which are utilized by the API Gateway Microservice to perform requests.

3.3 Formal and Mathematical Description

Let AG denote the API Gateway pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, n\}$
- **Set of API Gateway Microservice**: $G \subseteq M$
- **Set of Worker Microservices**: $W \subseteq M$

- **Set of Users**: U

Define a relation $R_{GW} \subseteq G \times W$ where $(g, w) \in R_{GW}$ indicates that the API Gateway Microservice g forwards requests to the Worker Microservice w.

3.4 Pattern Properties

1. **Request Handling**: The API Gateway Microservice handles requests by analyzing and routing them to the appropriate Worker Microservices.
2. **Routing Capabilities**: The API Gateway Microservice provides additional functionalities such as routing and network address translation (NAT).

3.5 User-API Gateway Communication

- For each User $u \in U$, the User interacts with the API Gateway Microservice $g \in G$ using HTTP methods $\text{GET}, \text{POST}, \text{PUT}, \text{DELETE}$. Formally, $\forall u \in U$, $\exists g \in G$ such that $u \rightarrow g$ where $\rightarrow$ denotes the communication using the specified HTTP methods.

3.6 API Gateway-Worker Communication

- Each API Gateway Microservice $g \in G$ sends requests to multiple Worker Microservices $w \in W$. Formally, $\forall g \in G, \exists w \in W$ such that $(g, w) \in R_{GW}$.

3.7 Service Dependency Graph

The Service Dependency Graph (SDG) for the API Gateway pattern is Figure 3.

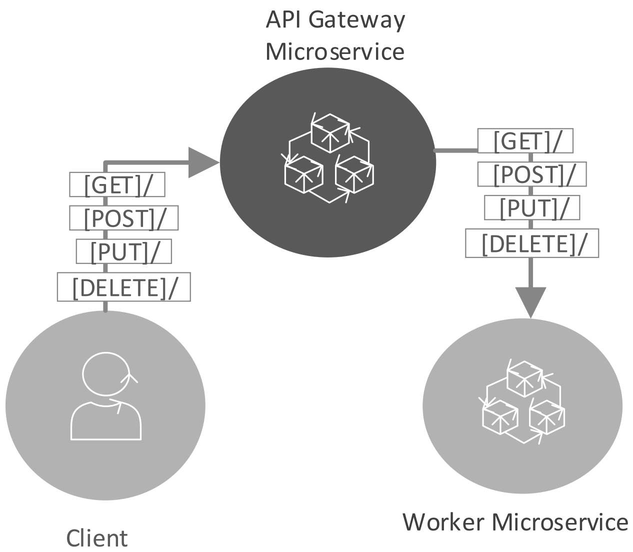

Figure 3: Dependency graph of the API Gateway pattern.

4 Pattern Name: Saga Distributed Transactions

4.1 Textual Description

The **Saga Distributed Transactions** pattern addresses the challenges of maintaining ACID (Atomicity, Consistency, Isolation, Durability) properties in a distributed microservice architecture. In traditional databases, a transaction ensures data validity despite errors, power failures, or other issues by adhering to ACID properties. The Saga pattern aims to replicate these properties in distributed systems by breaking a transaction into a series of smaller, manageable operations (sagas) that can be coordinated to maintain consistency and handle failures.

4.2 Pattern Signature

1. **Roles**:
   - **Worker Microservices**: Each worker microservice executes a part of the system's functionality within the Saga. They handle specific operations and are responsible for coordinating with other microservices to complete a saga.
2. **Microservice Count**:
   - The system must have at least two worker microservices to demonstrate the functionality of the Saga pattern effectively.
3. **Communication**:
   - Each worker microservice establishes a bidirectional relationship with its adjacent worker microservices via POST APIs. This relationship includes sending output to the next microservice and receiving compensation commands in case of failures.
4. **Knowledge of Relationships**:
   - Each worker microservice maintains knowledge of its predecessor and successor microservices, storing their addresses in a list for communication purposes.

4.3 Formal and Mathematical Description

Let S denote the Saga pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, n\}$
- **Set of Worker Microservices**: $W \subseteq M$

Define a relation $R_S \subseteq W \times W$ where $(w_i, w_j) \in R_S$ indicates that Worker Microservice w_i communicates with Worker Microservice w_j via POST API.

4.4 Pattern Properties

1. **Bidirectional Communication**: Each Worker Microservice w_i communicates bidirectionally with its adjacent microservices. It sends output to w_{i+1} and receives compensation commands from w_{i-1} if a rollback is required.
2. **Transaction Management**: The Saga pattern ensures that a series of operations are completed successfully or compensated for, maintaining transactional consistency across distributed microservices.
3. **Microservice Awareness**: Each Worker Microservice is aware of its predecessor and successor in the saga sequence and maintains their addresses.

4.5 Worker Communication

- For each Worker Microservice $w_i \in W$, it interacts with the next Worker Microservice w_{i+1} and receives compensation commands from the previous Worker Microservice w_{i-1}. Formally, $\forall w_i \in W, \exists w_{i+1} \in W$ such that $(w_i, w_{i+1}) \in R_S$ and $\exists w_{i-1} \in W$ for compensation.

4.6 Service Dependency Graph

The Service Dependency Graph (SDG) for the Saga Distributed Transactions pattern is Figure 4.

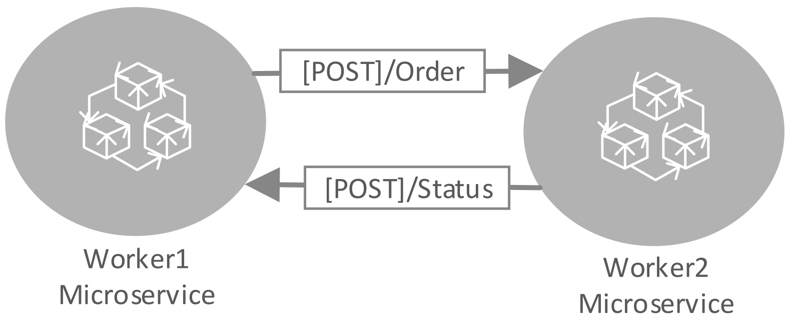

Figure 4: Dependency graph of the Saga Distributed Transactions pattern.

5 Pattern Name: Ambassador

5.1 Textual Description

The **Ambassador** pattern is a special case of the Sidecar pattern. Similar to the Sidecar pattern, the Ambassador pattern is useful for commonly used functional components that can be easily shared across services. These components, such as logging, metering, alerting, markdown functionality, etc., are separately deployable. The difference between the Sidecar pattern and the Ambassador pattern lies in the remote communication with a legacy application. In scenarios where a remote legacy application does not provide a proper interface, the Ambassador pattern is employed as a wrapper to create a remote connection between client applications and the legacy application.

5.2 Pattern Signature

1. **Roles**:
   - **Client Application**: Located on the host and may use remote microservices to perform part of its tasks.
   - **Ambassador Microservice**: Responsible for safely sending user requests to remote microservices, handling routing, authentication, receiving responses from remote microservices, and delivering these responses to the client application.
   - **Worker Microservices**: Perform the main functionality of the system.
2. **Communication**:
   - The Ambassador microservice communicates with worker microservices through POST, PUT, GET, and DELETE APIs.
3. **Interactions**:
   - The client application has no direct connection with the worker microservices.
   - The Ambassador microservice and the client application are on the same host.
   - There is only one Ambassador microservice.
   - There is at least one worker microservice.
4. **Communication Direction**:
   - There is a directional edge from the Ambassador microservice to worker microservices, indicating unidirectional communication from the Ambassador microservice to the worker microservices.

5.3 Formal and Mathematical Description

Let A denote the Ambassador pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, n\}$

- **Set of Worker Microservices**: $W \subseteq M$
- **Set of Ambassador Microservices**: $A \subseteq M$
- **Set of Clients**: C

Define a relation $R \subseteq A \times W$ where $(a, w) \in R$ indicates that Ambassador microservice a can communicate with Worker microservice w.

5.4 Pattern Properties

1. **Independence**: For any microservice $m \in M$, changes to m do not affect other microservices. Formally, if $m_i, m_j \in M$ and $i \neq j$, then m_i and m_j operate independently.
2. **Language Agnostic**: Microservices are language-agnostic, i.e., the functionality of $m \in M$ is independent of the programming language used.

5.5 Client-Ambassador Communication

- For each client $c \in C$, the client interacts with Ambassador microservice $a \in A$ via HTTP methods $\text{GET}, \text{POST}, \text{PUT}, \text{DELETE}$. Formally, $\forall c \in C$, $\exists a \in A$ such that $c \to a$ where $\to$ denotes the communication using the specified HTTP methods.

5.6 Ambassador-Worker Communication

- Each Ambassador microservice $a \in A$ communicates with multiple worker microservices $w \in W$. This is represented as $\forall a \in A, \exists w \in W$ such that $(a, w) \in R$.

5.7 Unidirectional Communication

- There is a directional edge from Ambassador microservices to Worker microservices. Formally, $\forall a \in A, \forall w \in W$, if $(a, w) \in R$, then communication from a to w is unidirectional.

5.8 Dependency Graph

The Dependency Graph for Ambassador pattern is Figure 5.

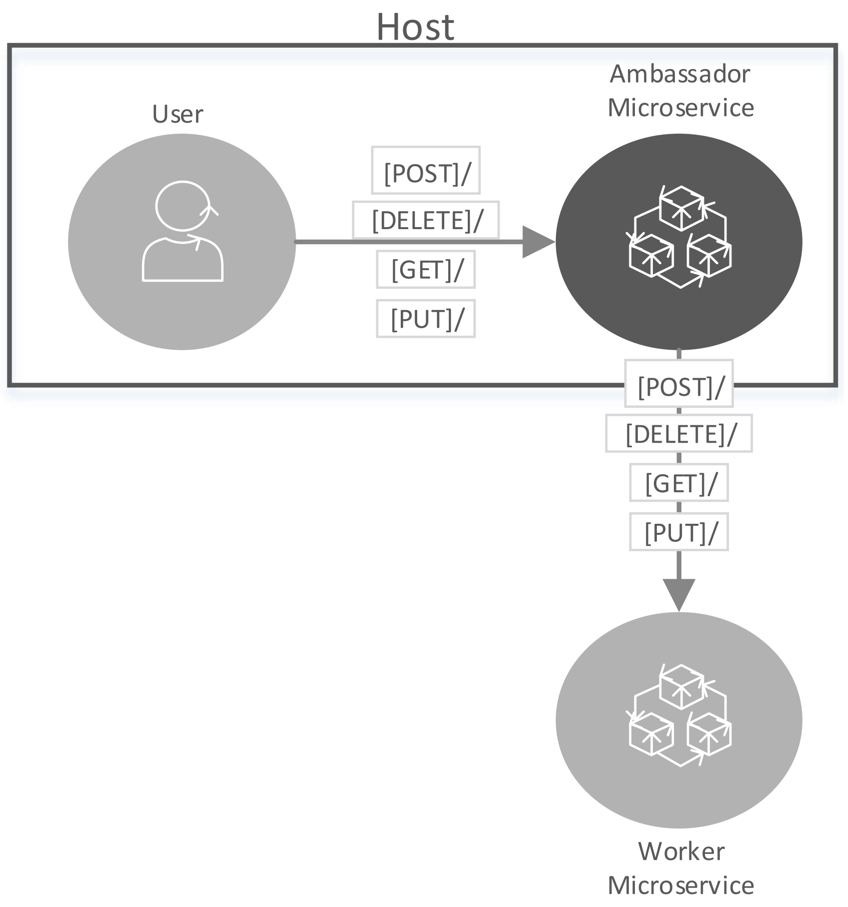

Figure 5: Dependency graph of the Ambassador pattern.

6 Pattern Name: Compute Resource Consolidation

6.1 Textual Description

The **Compute Resource Consolidation** pattern aims to optimize the use of computational resources in cloud environments. In such environments, tasks are often divided into multiple sub-tasks, each executed on different geographically distributed remote services. If a web service is underutilized and resides on a physical machine that remains idle for most of the time, it results in inefficient use of resources. To address this issue, the Compute Resource Consolidation pattern consolidates processing tasks onto a minimal number of physical or virtual machines, allowing some machines to be turned off when they are not needed.

6.2 Pattern Signature

1. **Roles**:
   - **Consolidation Microservice**: Responsible for monitoring, moving tasks in progress, and consolidating them onto other physical machines. This microservice ensures that tasks are efficiently distributed and resources are optimized.
   - **Worker Microservices**: Execute the main functionality of the application on physical or virtual machines.
2. **Microservice Count**:
   - The pattern involves at least three microservices: two worker microservices and one consolidation microservice.
   - The consolidation microservice is a singleton, while the number of worker microservices is at least two and can be up to N, where $N > 2$.

3. **Knowledge and Communication**:
    - The consolidation microservice maintains a list of all worker microservices and continually updates it.
    - Communication between microservices is conducted using GET and POST APIs.

6.3 Formal and Mathematical Description

Let C denote the Compute Resource Consolidation pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, N\}$
- **Set of Worker Microservices**: $W \subseteq M$, where $|W| \geq 2$
- **Consolidation Microservice**: $C \in M$, and C is a singleton.

Define a relation $R_C \subseteq W \times W$ where $(w_i, w_j) \in R_C$ indicates that Worker Microservice w_i communicates with Worker Microservice w_j via POST API.

6.4 Pattern Properties

1. **Singleton Consolidation Microservice**: There is only one consolidation microservice in the system.
2. **Dynamic Worker List**: The consolidation microservice maintains and updates a dynamic list of all worker microservices.
3. **Efficient Resource Utilization**: Tasks are consolidated onto a minimal number of machines to optimize resource usage and potentially allow some machines to be turned off.

6.5 Microservice Communication

- The consolidation microservice interacts with worker microservices using GET and POST APIs for monitoring and task redistribution. Formally, $\forall w_i, w_j \in W$, communication between w_i and w_j is managed by the consolidation microservice C using the specified APIs.

6.6 Service Dependency Graph

The Service Dependency Graph (SDG) for the Compute Resource Consolidation pattern is Figure 6.

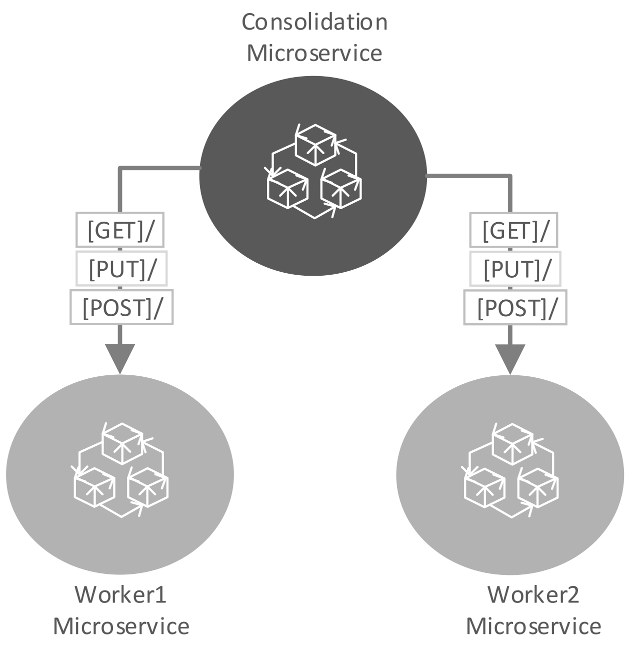

Figure 6: Dependency graph of the Compute Resource Consolidation pattern.

7 Pattern Name: Priority Queue

7.1 Textual Description

The **Priority Queue** pattern focuses on prioritizing tasks and utilizing more powerful hardware resources for higher-priority tasks. To achieve this, several priority queues are created, each handling tasks of a specific priority. Higher-priority tasks are sent to higher-priority queues, which then allocate them to more powerful hardware resources. Each computing resource is managed by a worker microservice.

7.2 Pattern Signature

1. **Roles**:
    - **Publisher Microservice**: Maintains a list of primary tasks with varying priorities and sends each task to a corresponding priority queue.
    - **Queue Management (QM) Microservice**: Receives and manages tasks with specific priorities. The number of QM microservices should correspond to the number of priority levels.
    - **Worker Microservices**: Execute the primary tasks on their respective hardware resources.
2. **Microservice Count**:
    - The minimum number of worker microservices is one, and the maximum number is M, where $M > 1$. These worker microservices have varying infrastructure and processing power.
    - There is only one publisher microservice.
    - The minimum number of QM microservices is one, and the maximum number is N, where $N > M$.
3. **Communication**:
    - The publisher microservice communicates with the QM microservice through a directional edge.
    - The publisher microservice exposes a GET API, which the QM microservice uses to fetch tasks.
    - There is a directional edge from the QM microservice to worker microservices. Each worker microservice communicates with the QM microservice using POST and GET APIs to inform about its resource status (busy or idle) and to receive tasks for processing.

7.3 Formal and Mathematical Description

Let P denote the Priority Queue pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, N\}$
- **Set of Publisher Microservice**: $P \in M$
- **Set of Queue Management Microservices**: $Q \subseteq M$ where $|Q|$ corresponds to the number of priority levels.
- **Set of Worker Microservices**: $W \subseteq M$ where $|W| \geq 1$

Define the following relations:

- $R_{PQ} \subseteq P \times Q$: The publisher microservice P communicates with QM microservices Q to send tasks to the appropriate queue.
- $R_{QW} \subseteq Q \times W$: The QM microservices Q communicate with worker microservices W to send tasks for processing.

7.4 Pattern Properties

1. **Priority-Based Task Management**: Tasks are prioritized and allocated to queues and worker microservices based on their priority.
2. **Singleton Publisher**: There is only one publisher microservice responsible for task distribution.
3. **Dynamic Queue Management**: The number of QM microservices is proportional to the number of priority levels.

7.5 Microservice Communication

- The publisher microservice communicates with the QM microservices via GET API to provide tasks.
- The QM microservices use POST and GET APIs to communicate with worker microservices regarding task allocation and resource status.

7.6 Service Dependency Graph

The Service Dependency Graph (SDG) for the Priority Queue pattern is Figure 7.

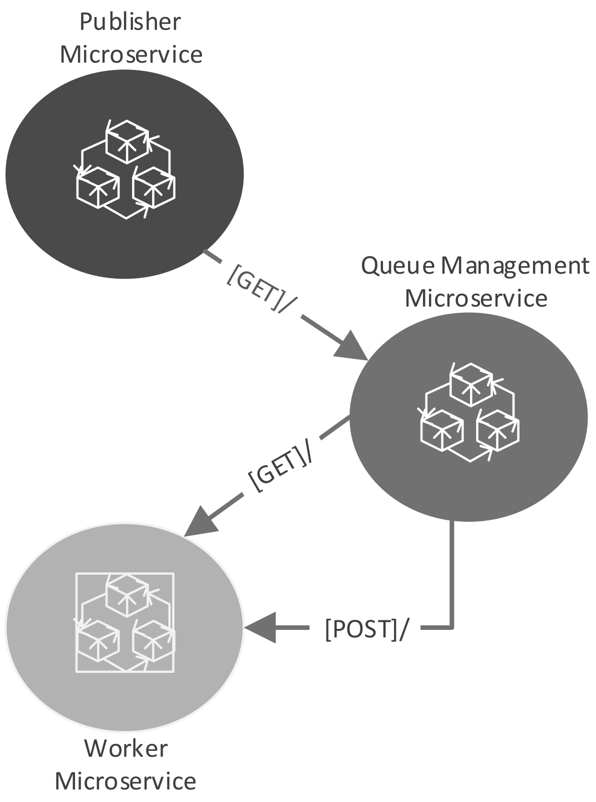

Figure 7: Dependency graph of the Priority Queue pattern.

8 Pattern Name: Leader Election

8.1 Textual Description

The **Leader Election** pattern introduces a leader microservice to manage access to shared datasets, reduce the complexity of relationships among worker microservices, and establish coordination and interaction between them. To avoid inconsistencies in shared datasets, the leader microservice assumes the role of coordinating these interactions. A widely used algorithm for implementing the Leader Election pattern is the Bully algorithm, which is applied in this context.

8.2 Pattern Signature

1. **Roles**:
   - **Leader Microservice**: Manages relationships across worker microservices, oversees access to shared memories and datasets, and simplifies the complexity of interactions among worker microservices.
   - **Worker Microservices**: Perform the main functionality of the application.
2. **Communication**:
   - All microservices use the POST API to communicate with each other.
3. **Microservice Knowledge**:
   - Each microservice is aware of other microservices and maintains a list of their addresses.
4. **Singleton Leader**:
   - The leader microservice is a singleton, but there must be at least two worker microservices in the pattern.

8.3 Formal and Mathematical Description

Let L denote the Leader Election pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, N\}$
- **Set of Leader Microservice**: $L \in M$
- **Set of Worker Microservices**: $W \subseteq M$ where $|W| \geq 2$

Define the following relations:

- $R_{LW} \subseteq L \times W$: The leader microservice L communicates with worker microservices W to manage access and coordination.
- $R_{WW} \subseteq W \times W$: Worker microservices W may communicate with each other indirectly through the leader.

8.4 Pattern Properties

1. **Leader Singleton**: There is only one leader microservice responsible for coordination.
2. **Post API Communication**: All microservices communicate using the POST API.
3. **Address List Maintenance**: Each microservice maintains a list of the addresses of other microservices.

8.5 Microservice Communication

- All microservices use POST APIs for communication, facilitated through the leader microservice.

8.6 Service Dependency Graph

The Service Dependency Graph (SDG) for the Leader Election pattern is as Figure 8.

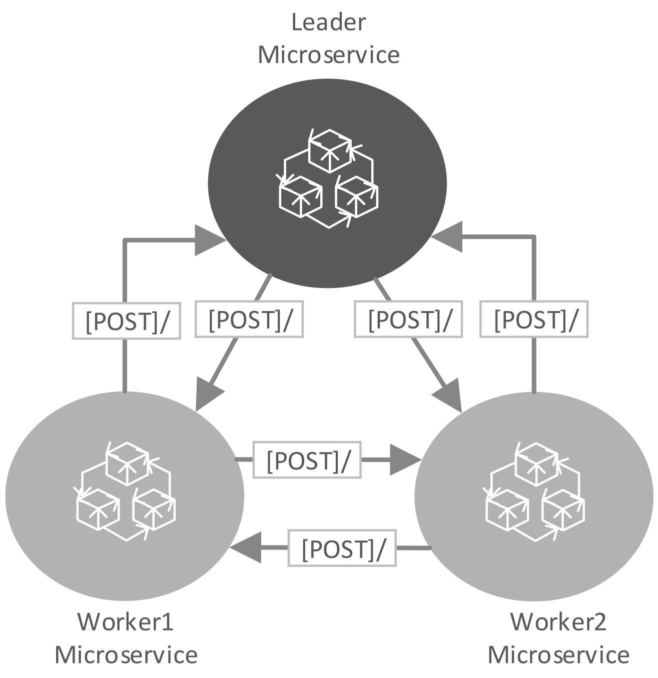

Figure 8: Dependency graph of the Leader Election pattern.

9 Pattern Name: Cache-Aside

9.1 Textual Description

The **Cache-Aside** pattern involves transferring frequently used data from a data store to a cache to enhance data access performance. This pattern aims to improve data retrieval speed by storing frequently accessed data in a cache, while ensuring data consistency between the cached data and the original data maintained in the worker microservice's database.

9.2 Pattern Signature

1. **Roles**:
   - **Master Microservice**: Caches frequently used data and manages data consistency between cached data and the original data in the worker microservice's database.
   - **Worker Microservice**: Maintains the datastore and transfers frequently used data to the master microservice.
   - **Client**: Sends requests to the master microservice to insert new data, update existing data, or read specified data.
2. **Relationships**:
   - The master microservice has a one-to-many relationship with worker microservices.
   - Each master microservice holds the address of its associated worker microservice.
   - In a hierarchical cache structure, a cache can reference its high-level cache and worker caches.
3. **Communication Directions**:
   - Communication from the client to the master microservice occurs via GET and POST APIs.
   - Communication from the master microservice to the worker microservice occurs via GET and POST APIs.
   - There is no direct communication between the client and the worker microservice.

9.3 Formal and Mathematical Description

Let C denote the Cache-Aside pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, N\}$
- **Set of Master Microservices**: $M_c \subseteq M$
- **Set of Worker Microservices**: $W \subseteq M$
- **Set of Clients**: C

Define the following relations:

- $R_{MCW} \subseteq M_c \times W$: Each master microservice $m_c \in M_c$ communicates with worker microservices W to maintain data consistency.
- $R_{MCC} \subseteq M_c \times C$: Each client $c \in C$ communicates with master microservices M_c for data operations.

9.4 Pattern Properties

1. **Cache Hierarchy**: Caches can reference higher-level caches and worker caches in a hierarchical structure.
2. **Communication Restrictions**: There is no direct communication between the client and worker microservices.

9.5 Client-Master Communication

- The client interacts with master microservices via GET and POST APIs to perform data operations.

9.6 Master-Worker Communication

- The master microservice communicates with worker microservices via GET and POST APIs to maintain and update data.

9.7 Service Dependency Graph

The Service Dependency Graph (SDG) for the Cache-Aside pattern is Figure 9.

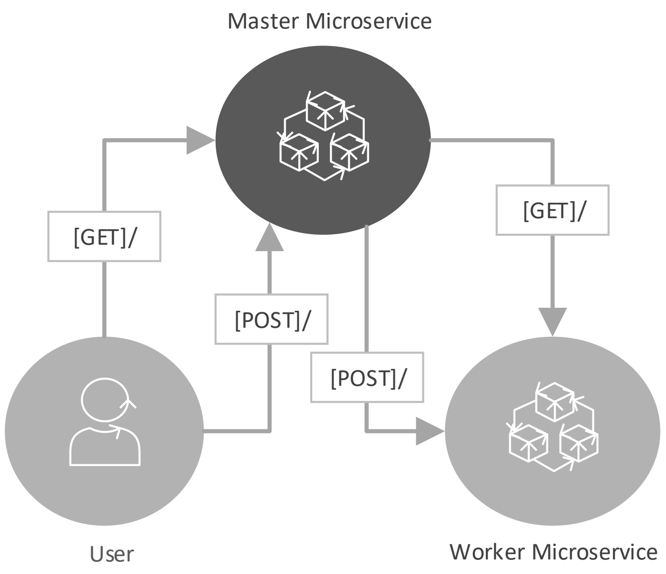

Figure 9: Dependency graph of the Cache-Aside pattern.

10 Pattern Name: Sidecar

10.1 Textual Description

The **Sidecar** pattern involves identifying all the independent functionalities of a system and implementing each functionality through a single microservice. The sidecar microservice is loosely coupled with the main application, ensuring that dependencies between microservices are weak. This allows for changes in one microservice without affecting others. Additionally, since each feature is implemented in its own microservice, it is independent of the programming language and can communicate with other microservices implemented in different languages. On each host, common operations such as routing, authentication, load balancing, traffic management, and health checks can be managed by a manager microservice called the sidecar.

10.2 Pattern Signature

1. **Roles**:
   - **Client**: Initiates a request to perform a task.
   - **Sidecar Microservice**: Located on the client host, it acts as an intermediary between the client and remote microservices.
   - **Worker Microservices**: Responsible for executing the main tasks and functionalities of an application. These microservices can be located on the client host or used remotely.
2. **Communication**:
   - The client sends requests to the Sidecar microservice via GET, POST, PUT, and DELETE APIs to access remote worker microservices.
   - Each Sidecar microservice communicates with several worker microservices.
3. **Communication Direction**:
   - There is a directional edge from the Sidecar microservice to worker microservices, indicating unidirectional communication from the Sidecar microservice to the worker microservices.

10.3 Formal and Mathematical Description

Let S denote the Sidecar pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, N\}$
- **Set of Sidecar Microservices**: $S \subseteq M$
- **Set of Worker Microservices**: $W \subseteq M$
- **Set of Clients**: C

Define the following relations:

- $R_{SW} \subseteq S \times W$: Each Sidecar microservice $s \in S$ communicates with worker microservices W to perform its tasks.
- $R_{SC} \subseteq S \times C$: Each client $c \in C$ communicates with Sidecar microservices S for request handling.

10.4 Pattern Properties

1. **Loose Coupling**: Sidecar microservices are loosely coupled with the main application, ensuring that changes in one microservice do not affect others.
2. **Language Agnostic**: Each microservice operates independently of the programming language used, allowing for communication between microservices implemented in different languages.

10.5 Client-Sidecar Communication

- The client interacts with the Sidecar microservice via GET, POST, PUT, and DELETE APIs.

10.6 Sidecar-Worker Communication

- Each Sidecar microservice communicates with multiple worker microservices.

10.7 Unidirectional Communication

- There is a directional edge from Sidecar microservices to worker microservices, indicating unidirectional communication.

10.8 Service Dependency Graph

The Service Dependency Graph (SDG) for the Sidecar pattern is Figure 10.

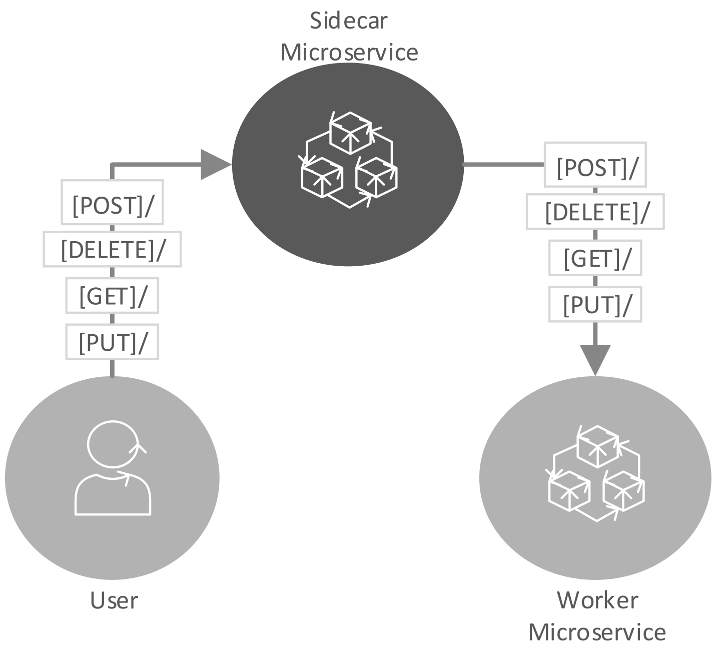

Figure 10: Dependency graph of the Sidecar pattern.

11 Pattern Name: Pipes and Filters

11.1 Textual Description

The **Pipes and Filters** pattern aims to divide a complex task into several smaller tasks to improve efficiency, scalability, reusability, and maintainability. The pattern organizes the system into a pipeline where each microservice (filter) processes data and passes it to the next microservice through a predefined sequence (pipe).

11.2 Pattern Signature

1. **Roles**:
   - **Master Microservices**: Located at the beginning and end of the pipeline. The master microservice at the beginning receives requests and sends them through the pipeline. The master microservice at the end receives outputs from the pipeline and sends them to the user.
   - **Worker Microservices**: Perform the main functionality of the system within the pipeline. Each worker microservice processes data and passes it to the next microservice.
2. **Communication**:
   - Communication from the beginning to the end of the pipeline is unidirectional, with each microservice sending information to the next microservice using POST APIs.
3. **Minimum Number of Worker Microservices**:
   - To effectively illustrate the use of pipelines, there should be at least two worker microservices in the pipeline.

11.3 Formal and Mathematical Description

Let P denote the Pipes and Filters pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_i \mid i = 1, 2, \ldots, N\}$
- **Set of Master Microservices**: $M_m \subseteq M$
- **Set of Worker Microservices**: $W \subseteq M$

Define the following relations:

- $R_{PW} \subseteq M_m \times W$: Each master microservice $m_m \in M_m$ communicates with worker microservices W in the pipeline.
- $R_{WW} \subseteq W \times W$: Each worker microservice $w_i \in W$ communicates with the next worker microservice $w_j \in W$ through POST API.

11.4 Pattern Properties

1. **Pipeline Efficiency**: By dividing a complex task into smaller tasks, the pattern increases system efficiency and scalability.
2. **Reusability**: Each worker microservice (filter) can be reused in different pipelines or contexts.
3. **Maintainability**: The pipeline structure makes it easier to update or replace individual microservices without affecting the entire system.

11.5 Master-Worker Communication

- The master microservices at the beginning and end of the pipeline communicate with worker microservices and handle the initiation and termination of the pipeline processing.

11.6 Worker-Worker Communication

- Worker microservices communicate with each other through POST APIs, sending processed data to the next microservice in the pipeline.

11.7 Service Dependency Graph

The Service Dependency Graph (SDG) for the Pipes and Filters pattern is Figure 11.

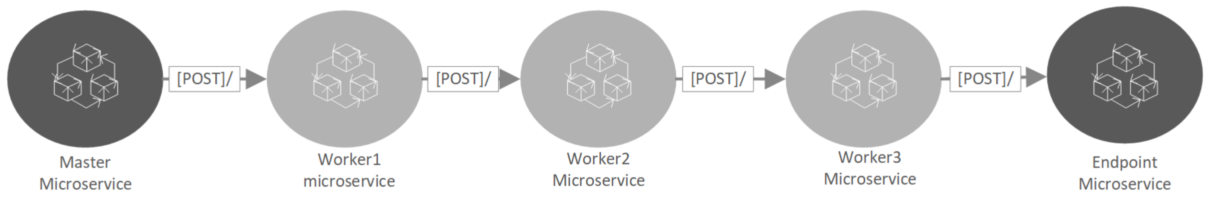

Figure 11: Dependency graph of the Pipes and Filters pattern.

12 Pattern Name: Static Content Hosting

12.1 Textual Description

The **Static Content Hosting** pattern focuses on the separation of static data from dynamic data and managing this static data using a dedicated microservice known as the Storage microservice. The goal of this pattern is to reduce the overhead of computational resources by handling static content separately from dynamic processing tasks.

12.2 Pattern Signature

1. **Roles**:
   - **User**: An entity that interacts with the Storage microservice to either retrieve or update static content.
   - **Storage Microservice**: Responsible for managing and serving static content. It handles requests from both clients and admins.
2. **Communication**:
   - **Client Requests**: The client submits a request to the Storage microservice via a GET API to retrieve static data. This creates a directional edge from the client to the Storage microservice.
   - **Admin Requests**: The admin submits a request to the Storage microservice via a POST API to insert new static data or update existing data. This creates a directional edge from the admin to the Storage microservice.
3. **Types of Users**:
   - **Client**: Sends requests to the Storage microservice to access static content.
   - **Admin**: Sends requests to the Storage microservice to manage static content.

12.3 Formal and Mathematical Description

Let S denote the Static Content Hosting pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_s\}$ where M_s is the Storage microservice.
- **Set of Users**: $U = \{U_c, U_a\}$ where U_c represents the Client and U_a represents the Admin.

Define the following relations:

- $R_{UC} \subseteq U_c \times M_s$: The relationship where each client U_c communicates with the Storage microservice M_s through GET APIs.
- $R_{UA} \subseteq U_a \times M_s$: The relationship where each admin U_a communicates with the Storage microservice M_s through POST APIs.

$$R_{UC} = \{(U_c, M_s) \mid U_c \text{ is a client and } M_s \text{ is the Storage microservice}\}$$

$$R_{UA} = \{(U_a, M_s) \mid U_a \text{ is an admin and } M_s \text{ is the Storage microservice}\}$$

12.4 Pattern Properties

1. **Resource Efficiency**: By offloading static content management to a dedicated Storage microservice, the pattern minimizes computational overhead and optimizes resource usage.
2. **Separation of Concerns**: Static content is managed independently from dynamic processing, leading to a cleaner architecture and easier management of static data.
3. **Simplified Access**: Clients and admins interact with a single microservice for static content, simplifying the system architecture and reducing potential points of failure.

12.5 Master-Worker Communication

- The **Storage microservice** receives requests from clients to serve static content and from admins to manage static content. Communication is managed through GET and POST APIs.

12.6 Service Dependency Graph

The Service Dependency Graph (SDG) for the Static Content Hosting pattern is Figure 12.

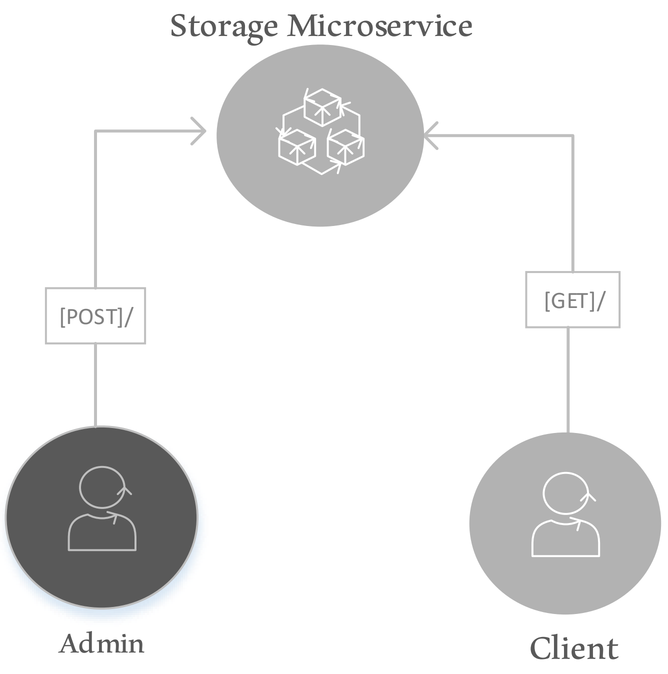

Figure 12: Dependency graph of the Static Content Hosting pattern.

13 Pattern Name: Aggregation

13.1 Textual Description

The **Aggregation** pattern addresses the need to manage a set of microservices to perform a specific business task. In this pattern, a central microservice, known as the Aggregator, is responsible for managing the task execution by coordinating various worker microservices. The Aggregator microservice receives user requests, identifies the required worker microservices, invokes them, collects their responses, and integrates these responses to produce the final output. This pattern enhances transparency in business logic by centralizing the coordination and aggregation process.

13.2 Pattern Signature

1. **Roles**:
   - **Aggregator Microservice**: Receives user requests, identifies the necessary worker microservices, invokes them, aggregates their responses, and delivers the final output to the user.
   - **Worker Microservices**: Perform the core functionality required to process the business logic.
2. **Communication**:
   - Worker microservices provide GET, POST, PUT, and DELETE APIs to interact with the Aggregator microservice.
   - The Aggregator microservice has one-sided connections to worker microservices.
3. **Number of Aggregator Microservices**:
   - In a basic scheme of this pattern, there is exactly one Aggregator microservice.

13.3 Formal and Mathematical Description

Let A denote the Aggregation pattern. Define the following sets and relations:

- **Set of Microservices**: $M = \{M_a, M_w\}$ where M_a is the Aggregator microservice and M_w represents the set of Worker microservices.
- **Set of Worker Microservices**: $W \subseteq M$ where W contains all worker microservices involved in the pattern.

Define the following relations:

- $R_{AW} \subseteq M_a \times W$: Each Aggregator microservice M_a communicates with worker microservices W to perform the aggregation.

$$R_{AW} = \{(M_a, w_i) \mid M_a \text{ is the Aggregator microservice and } w_i \text{ is a worker microservice}\}$$

13.4 Pattern Properties

1. **Centralized Coordination**: The Aggregator microservice manages the task execution and coordination of worker microservices, simplifying the process of aggregating results.
2. **Transparency**: By consolidating the management and aggregation of outputs, the pattern increases transparency in the business logic.
3. **Scalability**: The pattern allows for the integration of multiple worker microservices, making it scalable for complex tasks.

13.5 Master-Worker Communication

- The Aggregator microservice receives requests from users, delegates tasks to worker microservices, and integrates their responses to provide the final output.

13.6 Worker-Worker Communication

- Worker microservices do not communicate directly with each other; all interactions are mediated by the Aggregator microservice.

13.7 Service Dependency Graph

The Service Dependency Graph (SDG) for the Aggregation pattern is Figure 13.

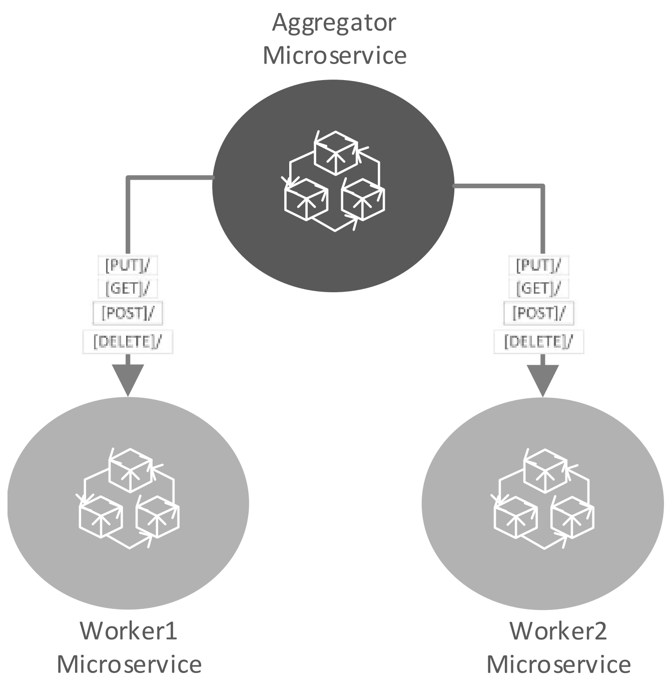

Figure 13: Dependency graph of the Aggregation pattern.

References

[1] Rajesh Bhojwani. Design patterns for microservices. https://dzone.com/articles/design-patterns-for-microservices, Accessed: 2024-08-21. [Online].

[2] Design patterns for microservices. https://learn.microsoft.com/en-us/azure/architecture/microservices/design/patterns, Accessed: 2024-08-21. [Online].

[3] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[4] Chris Richardson. Microservice architecture pattern. https://microservices.io/patterns/microservices.html, Accessed: 2024-08-21. [Online].