Commonly Used Patterns

Objectives

Key objectives of this chapter

- Commonly used patterns with microservices

1.1 Why Use Patterns?

- Design Patterns offer high-level description of proven solutions in various problem domains
- Learning patterns (and anti-patterns, for that matter), helps software designers avoid common pitfalls and become more productive
  - At the very least, you can save yourself time not reinventing the wheel or inventing a square wheel
- In this module, we will list some of the commonly used patterns with microservices
- Selecting a pattern that best suits your needs may require some investigation / PoC projects
- Some patterns are discussed in detail in other modules, so here those are just briefly mentioned

1.2 Performance-Related Patterns

- **Caching**
  - Helps with data availability (sometimes at the expense of data consistency)
    - This aspect is discussed in subsequent modules
  - Mostly, read-through and write-through caching used
  - Caching can be implemented locally or by using an external caching solution (e.g. Redis, Memcached, etc.)
- **Concurrent request processing**
  - Scaling out within a process is more elastic than scaling across processes
    - Essentially, this is multi-threaded request processing using a shared state model
    - Involves low-level usage of software / hardware synchronization primitives like locks, barriers, thread pools and the like
  - Reactive and asynchronous concurrency models include Event-based models, Promises, the Actor Model, and Reactive Streams

1.3 More Performance-Related Patterns

- **HTTPS Connection Pooling**
  - Establishing an HTTPS connection has an overhead due to the initial SSL handshake
  - Some solutions use NGINX web server as an external HTTP reverse proxy that natively handles connection pooling and SSL termination for your microservices
    - The basic idea is to avoid SSL renegotiation
- **HTTP Timeout**
  - Supported by the HTTP protocol through the 408 Request Timeout response status code
    - A rather aggressive way of dealing with modern browsers' feature of using HTTP pre-connection mechanism to speed up (concurrent) resource retrieval

1.4 Pagination vs. Infinite Scrolling - UX Lazy Loading

- **UX Lazy Loading**
  - On-demand piecemeal loading of the required data from the server to minimize an impact on user experience in front-end applications in case of large data sets
  - Lazy loading usually consists of two phases:
    - (Optional) Fast initial fetch of data from the server (e.g. loading and showing only one screenfull of transaction history), followed by

- The pagination phase which is on-demand loading of the next data batch when the user clicks a link or a button saying *Next*, *Load More*, and such like visuals
  - Non-paginated data fetch of large data sets is sometimes referred to as *infinite scrolling*

Notes:

For a good discussion of Infinite Scrolling vs. Pagination, visit https://uxplanet.org/ux-infinite-scrolling-vs-pagination-1030d29376f1

1.5 Integration Patterns

- **Ambassador**
  - This pattern is about offloading cross-cutting client connectivity concerns to an intermediate infrastructure service fronting the back-end service
  - Ambassador is deployed as a proxy to the remote service to help standardize and extend instrumentation (request routing, security, retries, monitoring, circuit breaker mechanism, etc.)
  - Bears some similarities to the Service Mesh pattern
- **Backends for Frontends**
  - Create a set of separate single-purpose back-end services to be consumed by specific front-end applications or interfaces
  - Main motivating factor:
    - Creating generic / universal / shared server-side service may be too complex or would create a burden on the back-end team

- As the obvious disadvantage is potentially massive code and development effort duplication across single-purpose services
  - To mitigate such situations, you may want to implement a hybrid solution with a core shared back-end service that is used by specialized single-purpose services

1.6 More Integration Patterns

- **Command Query Responsibility Segregation (CQRS)** (discussed later in the course)
- **Façade** (discussed later in the course)
- **Service Mesh** (discussed later in the module)

1.7 The Service Mesh Integration Pattern

- **Service Mesh** is an inter-service communication infrastructure that takes over the job previously done by an ESB
  - In this integration pattern, microservices do not communicate with each other directly, but rather through a software component called **mesh** (or side-car proxy) to which the core network functions (like resiliency in the form of circuit breaker capability or time-outs, routing, service discovery, etc.), as well as distributed cross-cutting concerns (like security, tracing, logging, etc.) are offloaded from each microservice
- Communication via mesh is done using standard protocols such as HTTP, gRPC (modern general-purpose cross-platform RPC infrastructure), and so on
- Bears similarities to the Ambassador pattern

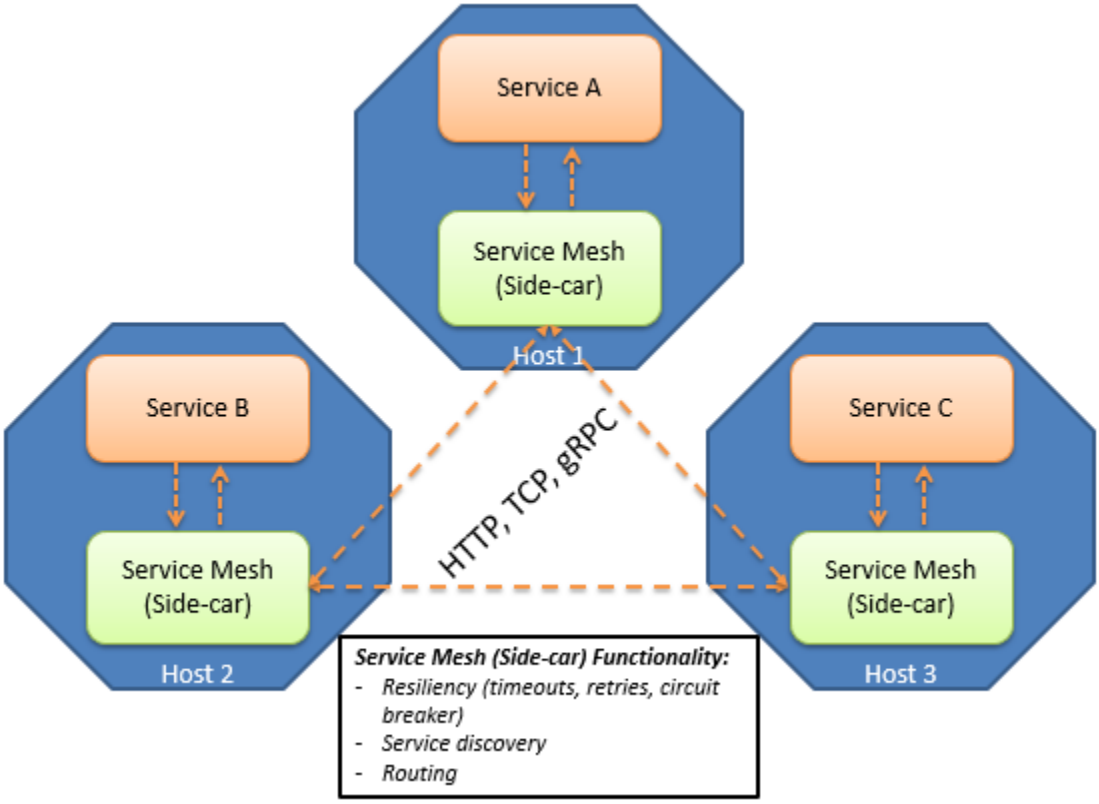
1.8 Mesh Pros and Cons

- Pros of this pattern include:
  - Offloading core network logic and cross-cutting concerns from each microservice to the infrastructure layer
- Cons:
  - Potentially poor performance
  - Dependency on the mesh component

Notes:

Linkerd and Istio are two popular open source service mesh implementations.

1.9 Service-to-Service Communication with Mesh



1.10 Resilience-Related Patterns

- **Circuit Breaker** (discussed later in the course)
  - This pattern prevents cascading failures when one failing system may trigger a sequence of failures of dependent services (the so-called domino effect)
- **Anti-Corruption Layer**
  - Introduced by Eric Evans in his book Domain-Driven Design (DDD)
  - An isolation layer created between interacting microservices that prevents direct access to their internals from each other across the layer; the layer performs the needed (bi-directional) request translation
  - Usually modeled after Façade or Adapter software patterns
  - The Ambassador and Mesh patterns can be used for the Anti-Corruption Layer pattern as well

1.11 Summary

- Asynchronous / reactive communication patterns become critical in designing microservices-based solutions
- In this module we listed some of the more important patterns used when working with microservices
- Selecting a pattern that best suits your needs may require some investigation / PoC projects