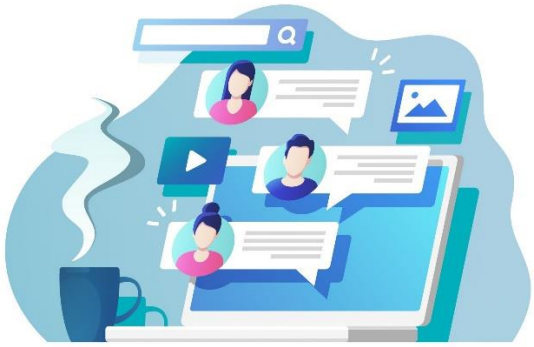
Messaging Summary Module

Summary made regarding Messaging module on Canvas



Name: Victoria C. A. Fong

Student number: 488384

Introduction

In this section you will learn to use messaging to integrate applications such that they can operate more independently from one another. The end result will be a more scalable solution.

messaging is an **asynchronous** form of communication. The advantage is clear: the receiver can process the message when it suits best. When the receiving application is down, the message will remain in storage to be delivered when the receiver is up again. The catch is that you need to rethink about communication flows in your system. If you apply messaging wisely, you will end up with a more scalable and better performant system.

Table of Contents

Design principles

Design principles serve as a base for IT architecture, development policies, and standards. Here are four principles that form a good starting point for the application of messaging:

- Design to scale out
- Minimize coordination
- Make all things redundant
- Design for evolution

I have summarized two design principle (Design to scale out & Design for evolution) that took interest in me for myself to look back on when designing for a scalable application design.

Design to scale out

Link to article: [Design to scale out]

the design principle “Design to scale out” is based on designing your application with the ability to be able to scale horizontally when new instances are added or removed.

The article gives a few recommendations to be able to design your application in a “scale out” design way. Such recommendations are:

- **Avoid instance stickiness:** when requests from the same client are always routed to the same server. To avoid this, you would have to ensure that any instance can handle any request.
- **Identify bottlenecks:** look for the bottlenecks in the system before adding more instances to the issue. The most common bottlenecks are stateful parts of a system.
- Decompose workloads by scalability requirements: noticing and decomposing workloads facing sudden loads can benefit your application.
- Offload resource-intensive tasks: identifying tasks that uses quite a lot of CPU or I/O resources and moving them to background jobs will minimize the load on the front end side which is handling user requests.
- Use built-in autoscaling features: predictable workload for applications can be scaled out on a schedule for efficiency. However, if the workload is not predictable, you can autoscale the application by using performance metrics where some services have built in support for autoscaling.
- Consider aggressive autoscaling for critical workloads: For example, adding new instances under fast under heavy load to handle added traffic and then scaling it back.
- Design for scale in: the application must be able to handle instances being removed. Suggestions to do so are:
    - Listen for shutdown events and shut down gracefully.
    - Clients/consumers of a service should support the temporary unavailability of a service and be able to retry.
    - For long duration tasks, you should consider breaking up the work by using checkpoints or the Pipes and Filters pattern.
    - Putting work items on queue so that another instance can pick up the work if an instance is removed during processing a task.

Design for evolution

Link to article: [Design for evolution]

Each developer is aware that applications overtime come accurse bug fixes, adding new features, bringing new technologies or scaling the existing system for scalability and resilience.

When an application is tightly coupled is adding such changes quite challenging without breaking other parts of the system.

Services can be more easy to design to evolve over time where developers can innovate and continuously deliver new features. This article address consideration to integrating this design.

Considerations:

- Enforce high cohesion and loose coupling: having a cohesive service where updating one service would not require changes to another services makes adding features or any code changes easier.
- Encapsulate domain knowledge: making sure domain knowledge of the business rules are encapsulated under the responsibility of the designated services will avoid domain knowledge spreading across different parts of the system.
- Use asynchronous messaging: decoupling the message producer from the consumer makes sure that the producer is not dependent on the consumer responding where the system can still function without any responds.
- Don't build domain knowledge into a gateway: avoiding implementing domain knowledge into gateways, you make sure that gateways are not a heavy dependency in the system.
- Expose open interfaces: by avoiding creating custom translation layers that sit between services, you prevent situations where you will have to update all upstream services that depend on updating one service.
- Design and test against service contracts: testing APIs will make sure of its functionality.
- Abstract infrastructure away from domain logic: avoiding combining infrastructure related functionality with domain logic will prevent tightly coupled layers.
- Offload cross-cutting concerns to a separate service: an example here is moving a functionality such as requests always needing to be authenticated to its own service, is evolving the a new authentication flow be easy without touching any of the services that use it.
- Deploy services independently: designing the application to work independently from one another will improve efficiency in adding new features, updates and bug fixes.

What is messaging solving?

Messaging solves “lost requests” that are sent to a service being down. Messaging helps glue the system parts together while keeping non-functional requirements intact. Design the system with the non-functional requirements in mind will with designing the messaging infrastructure will make sure of having these requirements in place.

a useful article that touches down on what non-functional requirements are, how to identify them and document them can be found here: “What are non-functional requirements?”

Design Patterns of Integration

Integration patterns are design patters based on messaging.

Brandan Jones YouTube video “What are Enterprise Integration Patterns” gives a clear overview of what enterprise integration patterns are with a clear explanation of usefulness. Further, he touches down on enterprise integration terms and lastly, different sources and destination besides other microservices.

A very in-depth written article of enterprise integration can be found at “Messaging Patterns Overview”. It gives a general understanding of design patterns, the number of different types of design patterns and categories they fall under.

Messaging - How to Apply?

This chapter helps you critically in your enterprise application design. By giving a few steps and ways to do so. By analyzing the to be build application and its non-functional requirements is it visible to what you as a developer will need / want for your system. With the help of sources provided is it easy to get lost but looking into a few designs and choosing one to implement is better than none. Critically analyzing your system not only helps you to be more certain into what you will be building more also thinking for future tasks and how decisions made now can benefit for future decisions.

During this module I used the “What are non-functional requirements?” document to recreate my non-functional requirements and better analyzing what message broker and messaging integration pattern I would like to follow.