Microservices: Design Principles Introduction

acarlstein.com/

Posted by Alejandro G. Carlstein Ramos Mejia on December 2, 2016 January 18, 2017 About Programming / Architecture / Microservices

Microservices | Microservices: Design

Design Criteria

In order for a service to be a microservice, it must match the following criteria:

- It needs to be have high cohesion.
- It needs to be autonomous.
- It must be business domain centric.
- It must have resilience.
- It must be observable.
- Automation should be used throughout the development process.

High Cohesion

- The microservices functionality and content in terms of input and output must be coherent.
    - It must have a single focus, and do it well.
        - This principle allows to control the size of the service.
        - It prevent to create a monolithic service by attaching other behaviors into the microservice which may not be related.
    - It must follow the Single Responsibility Principle.
        - A class can only change for one reason.
            - A business function.
            - A business domain.
        - Check SOLID principles for more information.
    - e.: If your microservice takes care of the postage, then all the input, output and process should be focus around the postage.
- It is like encapsulation principle from OOP programming principles.
  Take all the data and functionality that's related and put them together in one package.
- This principle makes the microservice easier to rewrite.
- This makes the system highly scalable, flexible and reliable.
    - We can scale up individual microservices which represent a specific business function or business domain.
    - The system is more flexible because we can change or upgrade the functionality of specific business functions or domains.
    - The system have reliability because we are changing specific small parts with

in the system without affecting the other parts.

Autonomus

- The interaction with an external systems should not make the microservice subject of change.
- There should be loose coupling between the microservices and/or clients
    - A change to a microservice should not force other microservices and/or clients to change.
        - Microservices should honor interfaces and contracts to other services and/or clients.
        - The way the input and output are formatted should not change between versions.
        - There should be a clear definition of the input and output of the microservice.
    - A microservice should be stateless,
        - No need to remember previous interactions in order to carry out the current request.
        - They should be independently deployable (if they honor the contracts and interfaces).
    - Services should always be backwards compatible.

Business Domain Centric

A service should represent a business function and/or business domain.

- Scope of service.
- This idea have being taken from domain driven design.
- Define a bounded context which contains all the functionality related to a specific part of the business to a business function or business domain.
- Define bounded context by defining boundaries and seams within code.
- Shuffle code when it is required so that code ends in the right place where it makes sense and belongs in term of business function and/or business domain.
- Aim to high cohesion.
- Microservices should be business domain centrist or responsible to business change.

Resilience

Embrace failure.

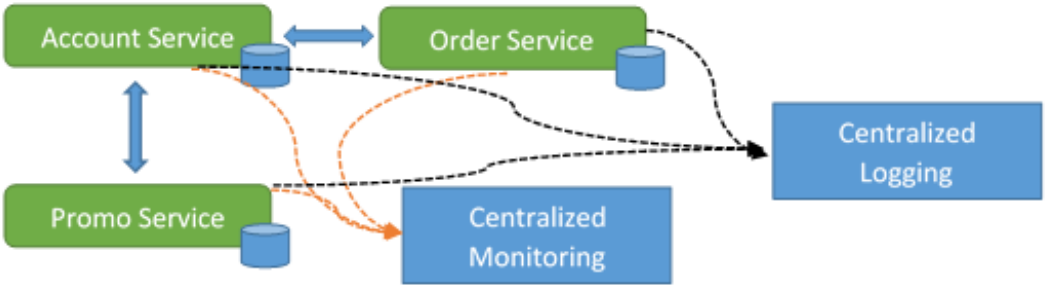
- Failure takes many forms:
  - It can be a service not responding to your service.
  - A connection line gone down.
  - A third party system failing.
- When failure is found, the microservice should degrade the functionality within or use a default functionality. It shouldn't cascade the failure. It shouldn't crash.
  - Decide when you should degrade functionality and/or when you should default functionality.
  - Example of degrade functionality: The service can decide to not display a page
  - Example of default functionality: The service decide to display a default page with a message.
- Make the microservice resilient by having multiple instance of microservices.
  - Each microservice should register themselves as they star up.
  - If any of these microservices fails, they will deregister themselves.
- Be aware of different types of failures:
  - Exceptions and/or errors.
  - Delays in replying request.
  - Unavailability of a microservices.
  - Network failures.

    Remember, we are using distributed transactions. One service may use several other services before it actually completes.
  - Validate microservice input from services and clients.

    The microservice shouldn't fail due wrong formatted data.

Observable

Microservices should being observable. We need a way to observe our system health and status so we can quick solve any problems.



- Current system activity.
- Error currently happened.
- Monitoring and logging (logs) needs to be centralized so there in one place to go in order to view information about the system health.
    - Centralized monitoring
    - Centralized Logging
    - Remember the system use distributed transactions across the network and several services.
    - System health:
        - Logs
        - Status
        - Information
        - Warnings
        - Errors
- Need a quick way of getting feedback in response to deployment.
- Data collected can be used for capacity planning and scaling up system.
  Monitor business data.
- Demand can be easily spotted.
- Easy to make specific measurements.

Automation

Automation is done in a form of tools since micro-services have many “moving parts”; therefore, testing can become complex. So, we create automated tools for testing. For example, the purpose of automating testing is to reduce the amount of time:

- Required for manual regression testing.
- For test integration between services and clients.
- To set up test environments.

These automated testing tools should provide us with a quick feedback, confirm that changes integrates with the entire system, to test integrations (also known as continuous integrations), to help with the pipeline to deployment, to indicate our microservice deployment status (is it ready?), to check in a microservice, to provide a way a way to move the build to the target machine (or the target cloud system), and any other functionality that may help.

The concept of using tools for deployments belongs to the continuous deployment category. The use of continuous integration tools allows the microservice architecture, which is a complex distributed system with multiple instances of the services, to be organized.Since manual deployment would be too time consuming and unreliable.