Microservices: Design

acarlstein.com/

Posted by Alejandro G. Carlstein Ramos Mejia on December 9, 2016 January 18, 2017 About Programming / Architecture / Microservices

 Micro-services: Design Principles Introduction | Microservices: Technology

Summary of Principles to Implement

- High Cohesion: Small micro-service focused and single functionality.
    - Single focus.
    - Do a single thing and do it well done.
- Autonomous: Allow upgrade of different part without risking other parts in the system.
    - Independently changeable.
    - Independently deployable.
- Business Domain Centric: aligned with the overall organization structure.

    Represent a business function or domain.
- Resilience:
    - Embrace failure.
    - Degrade or default functionality when failure detected.
- Observable: so we can have an overall view of the health of the system.;
    - Centralized Monitoring.
    - Centralized Logging.
- Automation: in order to administrate the complex system.

    Tools:
    - Testing.
    - Feedback.
    - Deployment.

High Cohesion Design Principle

In order to implement a micro-service with high cohesion, we need to:

1. Identify a single focus.

    It might be in a form of a business function.
    - e.: A function to generate invoice for the account system.

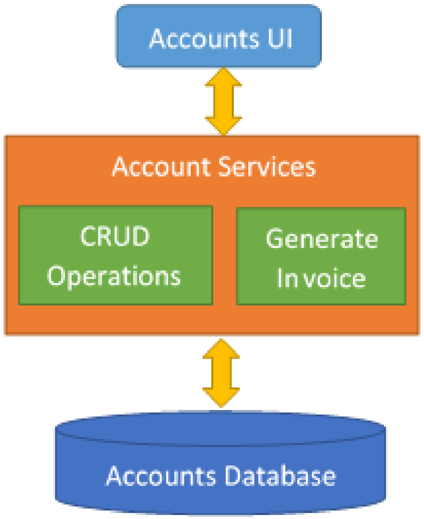
        Notices that it has clear inputs and outputs.
    - It might be in a form of a business domain.

        e. micro-service focus in creating, updating, retrieving, and deleting data related with a part of the organization such as the accounting deparment.
    - We should not crowd the micro-service with both types of focus.

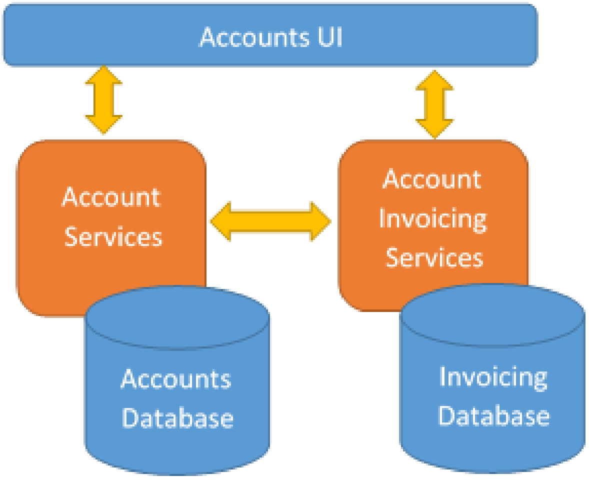
        - Example of no high cohesion:
        - Example of high cohesion:

2. Split into more smaller services

   ⚠ Avoid the thinking of “It’s kind of the same” mentality and begin
   coupling multiple business functions into one micro-service.



    1. We wish to avoid one business function breaking another business function.
    2. A micro-service should only have one reason to change.
    3. Don’t be lazy. Even if it requires an extra effort, make sure to create an extra micro-service or split an existent micro-service.



    4. Remember the overall objective which is to have a system which is reliable, flexible and scalable.
    5. We want our system being in separate parts so we can deploy them as specific parts.
    6. Laziness will only lead us to a system which would be monolithic in nature; therefore, introducing all the disadvantage of such.
    7. As you create micro-services in an incremental way, you will be learning how to maintain them and monitor these micro-services.
3. Ensure your micro-services have high cohesion.
    1. Continuously question the design of micro-services.
        1. i.e.: Is there a reason why a new micro-service has to change?

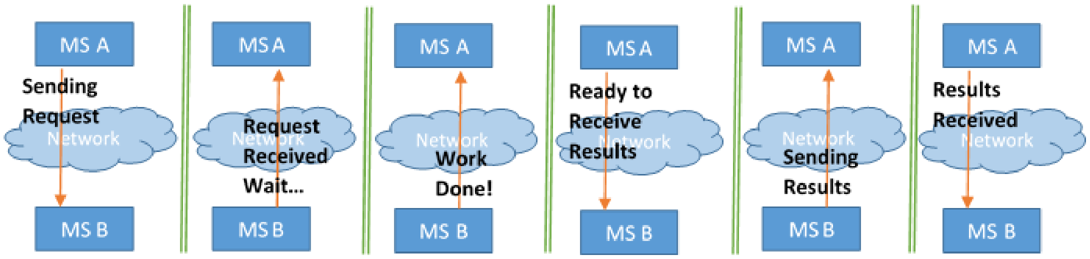
Autonomous

Our micro-services must be independently deployable and changeable.

Autonomous: Loosely Couple

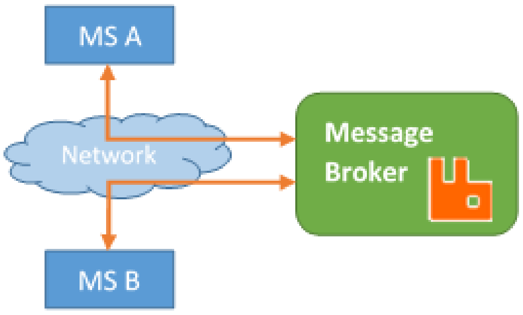
- Each micro-services should depend on each other in the most minimal way possible.
- They should have the least amount of knowledge of each other.
- They shouldn’t be connected physically to each other directly, but use a medium such as the network in order to talk to each other.

- Communication can be synchronous.



  - A micro-service calls another and waits for a reply.
  - Advantage is to know that if our communication was successful or not due the status of response.
  - In order to work, the micro-service receiving the request should respond right away even before it perform and completed its actual task for the request.
    - This allow the micro-service doing the request to carry out its own task while waiting.
    - When the micro-service doing the work, finish, it will call back the micro-service requestor indicating that the task is completed.
    - The original request should include a callback address.

      In this way, the micro-service knows who to notify when the job is done.
- Communication can be asynchronous.



  - Instead of having the micro-services making requests between each other, the micro-services public events in a form of messages.
    - These events are queue by a message broker such as RabbitMQ.
    - All micro-services listen out for these events and carry out tasks if any of those event correspond to them.

      If they are interested in the message they pick it up. Process them. Then send an event so the other micro-services interested pick the result.
    - Micro-services subscribe to events
  - Use open communication protocols in such way that we obtain a technology agnostic API.
    - Example of communication protocol: REST over HTTP and data in JSON
    - This allow for micro-services to work on different technology stacks

instead of forcing them to work on the same technology stack.

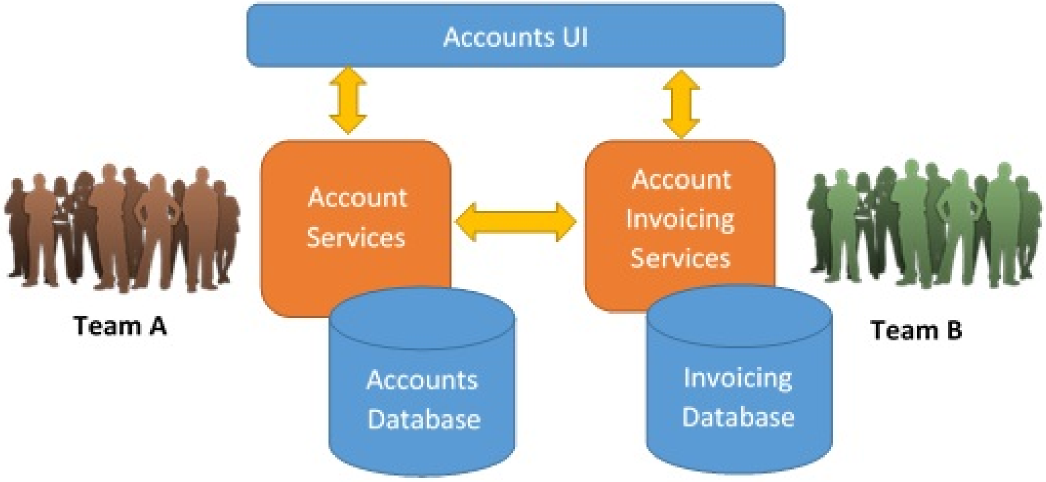
i.e.: Using REST JSON, we can have a .NET-based service communicate with a Java-based service.

- Avoid the use of client libraries.
  - A consumer of your micro-services requires the implementation of a client library in order for the consumer to talk to your micro-service.
  - Client libraries increase coupling because it force your micro-services and clients to change when its client library changes.
  - It forces the use of a specific technology platform at the consuming end.
- Micro-services should implement Order Shared Model which means that the micro-services should have a contract between them.
  - Fixed and agreed interfaces between the services.

    Method signatures and the format of the data that is exchanged.
  - We always use Share models of the data that are unlikely to change when any of the micro-services is enhanced.
    - The shared models should be different from the internal representation of the data within the micro-service.
    - Keep the internal representation of data separate from data that is going to be exchanges using the shared model.
  - These contracts and interfaces are important for multiple teams since it help to have a clear view of the known inputs and outputs of each micro-service.
- Avoid chatty exchange between micro-services.
- The sharing of things like databases between two micro-services should avoided.
  - While it may seems like a good idea to share data such a database, but a change in the shared database will result in both micro-services having to change (i.e. new schema change)

    This can lead to have to deploy both micro-services instead of one.
  - Force both micro-services to use the same database technology.
- Minimize the use of shared libraries within the microservice.
  - i.e.: A bug fixed in a shared library would force us to deploy both micro-services.
  - Perhaps that shared library should be a micro-service itself serving other micro-services.

Autonomous: Ownership and Versioning



- Each micro-service is owned by a team
- Small micro-services allow for small teams

  Small teams will be better retention of knowledge about the micro-service.
- It encourage small teams to build and maintain the micro-service autonomous.
- Teams are responsible to:
  - Design a micro-service that is independently changeable and deployable.
  - Agreeing the contract between the micro-services.
  - How the micro-services interact.
  - Maintain the contract so future changes don't break contracts with other micro-services.
  - Long-term maintenance of the micro-service.
- Ownership encourage to:
  - Collaborate with other teams.
  - Communicate contract requirements.
  - Communicate data requirements.
  - Concurrent development.
- Multiple teams can work on different micro-services at the same time plus agree in the interaction between these micro-services.
- When creating a new version of the micro-service, think about the versioning strategy for that micro-service.

  Create a new version of the micro-service avoiding breaking other micro-services by changing the contract.
- All new changes should be backwards compatible.
  - Other micro-services should be able to continue working without any change.
  - Honor the original contract that was agreed.
  - Ensure your new micro-services is not and will not break any existing contracts.
- Use integration tests to test the change of the micro-service for inputs and outputs, plus shared models.

  Test if the original contract is still intact.
- If a new version of your micro-service includes breaking changes, then you have

concurrent versions of your micro-service running.
    - An old and new version of your micro-service could be running at the same time.
    - This allow a period of transition from the old micro-service and the new micro-service.
- Use semantic versioning where the version number is made up of three numbers: Major.Minor.Patch
    - The major number increments if the new version of the micro-service is not backward compatible.
    - The minor number increments while the new version of the micro-service is backward compatible.
    - The path number increments if the new version of the micro-service have a defect fix

        Plus, the overall micro-service is still backwards compatible.
- When you with to include both old and new code in the new version of the micro-service, we can have coexisting endpoints.
    - The original endpoint which points at the original code (old version), and have a new endpoint pointing at the new version.
    - Consumer can slowly migrate from the old endpoint to the new endpoint.
    - We can have a new version of a micro-service which has the old endpoint; however, the old endpoint can have a wrapper for the new endpoint.

        In other words, you could have the old endpoint redirect the calls to the new endpoint.

Business Domain Centric

Micro-services should represent a business function or business domain.

- Define these business domains in a coarse manner.
- These business domains should represent departments or areas of the organization.
- Split each area into business functions or business areas.
- Have in consideration to review the benefits of splitting the micro-service further.
- Remember to have high cohesion.

    A micro-service must
    - Do one thing and do it well.
    - Have a single focus.
    - Only one reason for it to change.
- See micro-services as components
    - Maps to different components.
    - Functions within the organization.
- When parts of the organization change, we know which specific micro-service should be affected.
- Agree to a common language.

- Fix incorrect boundaries.

    Be ready to split a micro-service further.
- Merge two or more micro-service into one if they are doing the same thing.
- Consider the inputs and outputs and the contracts existent between the micro-services.
- We can split the system by technical boundaries.

    For example, we need a special micro-service for accessing data or improve performance.

Resilience

The entire system shouldn't go down for one failure; therefore, we must design our micro-services for all known failures.

- Known Failures:

    Downstream systems: Micro-services that carry our specific task
        - Internal and/or external services.
        - Network outages and network latencies.
        - Timeouts.
- Micro-service should degrade or default functionality on failure detection.
- Do not hang or delay a transaction. System should fail fast and recover fast.
- Use standard timeout length functionality between services communication.
- Our system should continuously monitor our timeouts and log our timeouts.

    This can help to workout specific behaviors related.
- Make issues transparent for health checks.

Observable: Centralized Monitor

Our system will consist of multiple micro-services and instances of micro-services; therefore, we must implement a centralized monitor system that allows us to see the system health.

- Monitor data in real time.
- Monitor health of the host
    - CPU usage, memory usage, and disk usage.
    - Response times.
    - Timeouts and number of timeout errors.
    - Exceptions and errors.
- Monitor service itself. Expose metrics within your service.
- Expand to include business data related metrics.
    - Number of orders.
    - Average time from basket to checkout.
- Collect and aggregate monitoring data.
    - From trends and history to details.
    - Drill down options.

- Visualize trends to spot patterns and potential problems.
- Compare data across servers.
- Trigger alerts

  For example, trigger alarm when a measures exceeds a threshold.

Observable: Centralized Logging

We are recording detailed information about events. It is key for problem solving in a system of distributed transactions.

- Log when our micro-services start up and/or shut down.
- Log code path milestones. For example:
  - Received a request
  - Code decisions
  - Give responses
- Log timeouts, exceptions and errors
- Information logged should be structured and be consistent across the system.

  A log may contain:
    - Level of information.
    - Information state.
    - Information regarding an error
    - Debug information.
    - Statistics that's have being recorded.
    - Date and time when event happened.
    - Correlation ID so we can trace distributed transactions across our logs.
      - A unique ID which is assigned to every transactions.
      - When the transaction becomes distributed, we can follow that transaction across our micro-services.
    - Host name so we know where the log entry came from.
    - Service name and service instance so we know which micro-service made the log entry.
    - A message which is the key information whic is associated with the event.

      i.e.: Callstack details regarding the exception.
- Keep structured logging format consistent.
  - This allows us to query the logging information.
  - We can search for specific patterns and specific issues.
- It allows to make transactions more traceable.

Automation: Continuous Integration Tools

The Continuous Integration tools provide an automatic way to do testing and feedback of your software changes.

- These tools work with the source control.

- Test software after check-in and change into the source control.
- Run unit tests and integration tests that have begin written.
  - Unit test and integration test are designed to test our production code.
  - They test that a change or enhancement in the code hasn't break the existing and new requirements.
- Provide quick feedback.

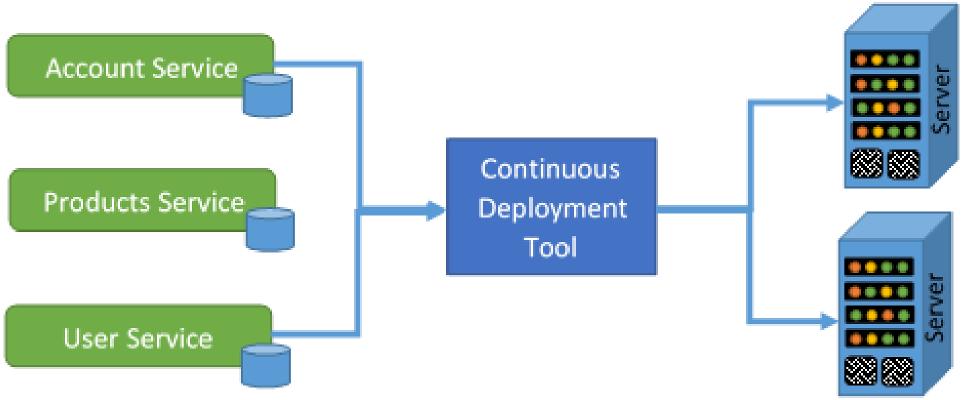
  If a micro-services breaks (itself or anything else that may use any of those micro-services), we will receive a quick feedback so we can fix it.
- Provide useful Information on the quality of integration.
- Prevent issues to pile up.

  Automatic feedback is sent to its respective teams so they can quickly fix the issue.
- Culture Note: All teams should stop development until all the issues reported have being fixed.
- Integration tools can be use to build our software

  Test Driven Development

Automation: Continuous Deployment Tools

These tools automate the software deployment.



- The Continuous Integration tool creates the build that the Continuous Deployment tool will deploy.
- There could be multiple micro-services and multiple instances of those micro-services on different servers.

  Each server could be running a different technology stack (i.e. Microsoft, Linux, Unix, etc)
- It is time consuming to configure this tool; however, it is done once. In the long run, this tools saves a ton of time.

  When a new version of a micro-service is available, the same configuration is used to re-deploy automatically.
- As long as all the continuous integration test pass, the new version will be deployed.
- This tools provide the ability to release anytime upgrades.
- It allows to deploy new version of your software to the market in a quick and

reliable way.

This improve customer experience.

© 2016 – 2017, Alejandro G. Carlstein Ramos Mejia. All rights reserved.