From monoliths to microservices

in the context of Cloud Computing

Origins of service-oriented architecture

- No standard for microservices
- SOA = Service-Oriented Architecture
  - Principles
  - Encourages reusable software components
  - Components provide well-defined interfaces
  - Unit = self-contained service + interface
  - SOA: services should be standalone process
    - Service = discrete unit of functionality, that can be accessed remotely and independently
    - Nothing about protocols
    - Vague regarding organization/deployment
    - First pub 2009 @https://www.soa-manifesto.org/
      - No longer available

SOA : service?

- The 4 properties of a service
  - It logically represents a repeatable business activity with a specified outcome
  - It is self-contained
  - It is a black box for its consumers, meaning the consumer does not have to be aware of the service's inner workings
  - It may be composed of other services
- Communication between services
  - Inter-Process Communication (IPC)
    - Sockets on the same machine
    - Shared memory for message queues
  - Remote Procedure Calls (RPC)
  - etc.
- Microservices
  - A specialization of SOA

The monolithic approach

- Concentration
  - "Everything about the service is in one place"
  - The same code base for:
    - The API
    - (Optional) database related ops
    - Handling the communication with external services
- Benefits
  - Single code base
  - Code management simplicity
  - Testing simplicity
  - The possibility of single-package deployment
- Extra benefits if using cloud-based technologies
  - To scale-up, run plural instances of the application
  - Use some database replication solution to have the multiple databases in sync
- Examples
  - Most solutions on LAMP architecture

The monolithic approach (2)

- If the app stays small
  - This is a sensible approach
  - Maintainable by a single team
- If big changes are needed, such as...
  - The need to work with new or different external services
  - Significant changes in the database layer
- The changes will impact the entire solution
  - Need for testing
  - Risks of collateral damage
  - Uneven scale needs
    - Limited scaling solutions
  - With time, it gets harder and harder to decouple parts
- Mitigate some of the above
  - With modular design
    - The problem of dependencies management arises

The microservices approach

- "Componentize" the solution
  - Organize the code in separate components
  - That can run in separate processes
  - Communicating via some protocol
    - e.g. HTTP, with functionality available via RESTful Web Services
- For example
  - Authentication, searching and reporting
    - Can usually be developed as components
    - These components have an API
    - Can possibly work with specific databases
  - The overall app communicates with them via their APIs
- How? Think on the internal interactions in the mono code
  - And explicit them as visible parts
  - Then question about the corresponding functions and data

The microservices approach (2)

- Tentative definition of microservices
  - Lightweight application, providing a focused list of features with a well-defined contract. It can be developed and deployed independently.
    - No compromise with HTTP and XML, JSON, whatever
      - It could be binary data being exchanged via UDP
    - But, many, many times, the protocol indeed is HTTP and the data is represented in XML and/or JSON
- Benefits
  - Code separation
  - Focus: goals and responsibility separation
    - Philosophically similar to the "single responsibility principle" (Robert Martin)
  - More scaling and deployment options
  - AKA "loose coupling"

The microservices approach (3)

- Risks of microservices
  - Not so good componentization
    - You'll need several iterations!
    - To add/remove microservices can be harder than to just work on the monolith
    - Hints: if changing one service always requires changing other(s), may be they should be together
  - Increased network relevance
    - Latency, speed, protocol choice become more relevant
    - New choices: async or blocking calls?
    - New questions: what happens if... ?
  - Increased challenges in data storage and data sharing
    - DB multiplication and questions regarding rights
    - To (not) duplicate?
    - How to remove?
  - Compatibility issues
    - Between microservices
    - Between technologies of the different stacks in use
  - Testing
    - Inferno? Several new technologies try to help: K8s, Terraform, CloudFormation

microservices in Python

- Focus on dealing with
  - Incoming requests
  - Structuring a response
  - Respond the response
- Abstract all the complicated parts
  - Protocol negotiation
  - Certificates
  - All this is handled by the web server
    - Apache HTTPD, nginx, Microsoft IIS
- Python can go beyond CGI = Common Gateway Interface
  - WSGI = Web Server Gateway Interface
    - There are WSGI extensions for most web servers
  - ASGI = Asynchronous Server Gateway Interface

Asynchronous Python

- Built in Python, since v3.5+
  - The asyncio library
  - https://docs.python.org/3/library/asyncio.html
- Asynchronous framework
  - Quart (very similar to Flask, but asynchronous)
  - Aiohttp
- Synchronous frameworks
  - Flask
  - Bottle
  - Pyramid
  - Cornice
- These frameworks...
  - Are mostly to route requests + some helpers
  - So, the microservices per se should be easy to rewrite in any framewwork
- There is a risk in using synch libraries in asynch code

Referências

- TODO