Software Architectures

Microservices

41492 – Engenharia de Software, Nuno Sá Couto e Rafael Direito
October 2nd 2023

Agenda

01

The 9 Pillars of Microservices (by Martin Fowler)

The 9 Pillars of Microservices (by Martin Fowler)

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

To know more

https://martinfowler.com/articles/microservices.html

Componentization via Services

- A **component** is a unit of software that is independently replaceable and upgradeable.

- **Modular design is always a good idea**. Each component is responsible for a specific part of the software.

- **Modularity can be achieved using:**

  - **Libraries** – called directly within the process

  - **Services** – called by out-of-process mechanism (Web API, RPC)

- In **Microservices** we **prefer using Services for the componentization. Libraries** can be **used inside the service**

- One main reason for using services as components (rather than libraries) is that services are **independently deployable**

Organized Around Business Capabilities

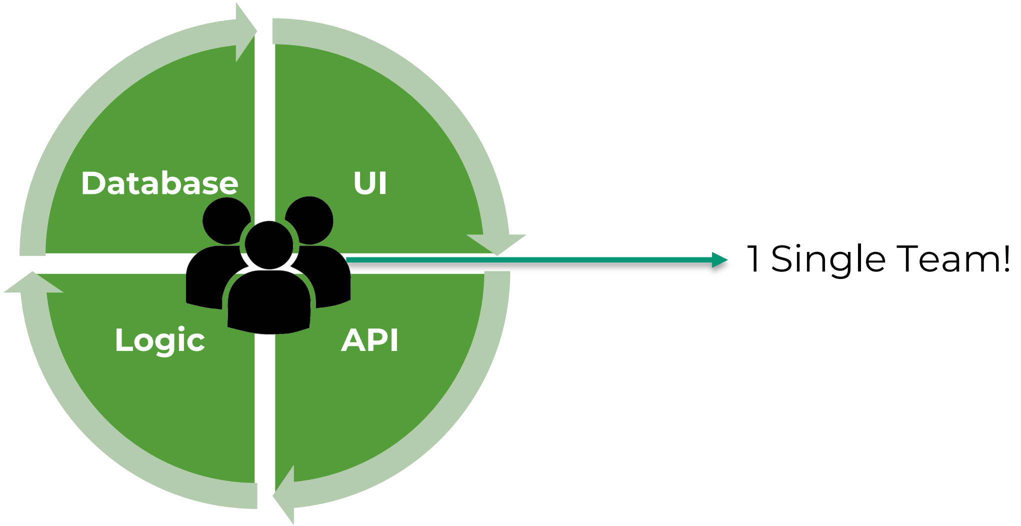
- **Monolithic Solutions had several horizontal teams:**
  - UI, Database, and server-side logic teams. At least…
  - This involved continuous (and critical) communication between all teams
  - **Time consuming!**

- With **Microservices**, **every service is handled by a single team, responsible for all aspects**
- **Every service** handles a **well-defined business capability**

Organized Around Business Capabilities

- The team has **only one goal and one goal alone**: to **make the service and its functionality work as best as possible**

- The **team is also autonomous** in making decisions about how functionality must be implemented without any concern of internal politics. **The speed of development and progress is higher, and the time to market is much lower.**



Products not Projects

- Project model: the aim is **to deliver some pieces of software which is then considered to be completed**. Then, the software is handed over to the maintenance team and the **project's team move to the next project**
    - The development team has **little to no interaction with the customer**

- Microservice/Product model: The team **should deliver a working product** and own it throughout all its lifecycle! **The team is responsible for maintaining the microservice after it is delivered**
    - The development team **continuously interacts with the customer**
    - AWS mentality: ***"You built it, you own it!"*** (Werner Vogels, 2006)
    - **The team's success is not only related to the software they produced, but also with its maintainability and operation!**

Smart Endpoints and Dumb Pipes

- **Traditional SoA architectures use complex mechanisms to manage the communication between Services.**

- A good example is the **ESB:**

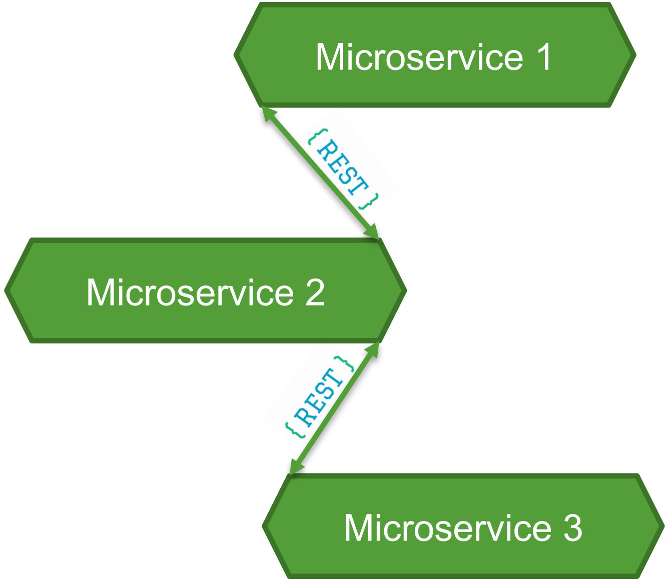
  - ESB had to be **“very smart”** to deal with message routing, choreography, message transformation and applying business rules

  - **ESBs got too difficult to maintain**

- Result: **The communication between services in an SoA architecture became a real challenge!**

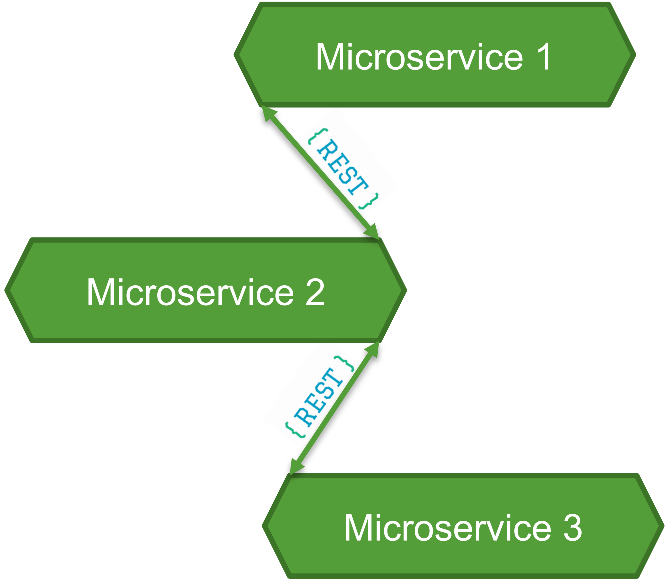
Smart Endpoints and Dumb Pipes

- **Microservices** use **Smart Endpoints and Dumb Pipes** (simple message buses)
  - The **complexity** should be **encapsulated in the Service itself, and not on the message buses**
  - Services should **communicate via simple protocols**. Example: HTTP
  - **Services can interact via RESTish protocols**, for instance through a REST API (built on top of HTTP and very simple to operate!)



Smart Endpoints and Dumb Pipes

- **Some Considerations:**
    - Direct connections between services **is not a good** idea (although it is on the image)
    - Better use **discovery service or a gateway**
    - In recent years more protocols were introduced (GraphQL, gRPC)…
    - … some of them quite complex (*contrary to best practice*), nevertheless we must consider them as they do a great job in exposing specific functionality



Decentralized Governance

- In **traditional projects there is a standard for almost anything:**
  - Which dev platform to use
  - Which database to use
  - How logs are created, etc.
- **In Microservices, things are a little different…**

Decentralized Governance

- With Microservices **each team makes its own decisions:**
  - Which dev platform to use
  - Which database to use
  - How logs are created, etc
- **Each team is fully responsible for its service**
  - Enabled by the loosely coupled nature of Microservices
  - Multi dev platform is called ***Polyglot***

Decentralized Data Management

- **Traditional systems have a single database**
  - **Stores all the system's data, from all components** (Monolithic and, many times, SOA based systems)

- In a Microservices Architecture, **each Microservice has its own database**
  - This is **controversial**, because, many times, **several services need to share a database.**
  - Having its own database, each microservice has **increased autonomy and control of its logic**
  - Several databases add **complexity** and may lead to **data duplication**
  - But... it also allows the development team to **use different types of databases for different services (SQL vs NoSQL)**

Infrastructure Automation

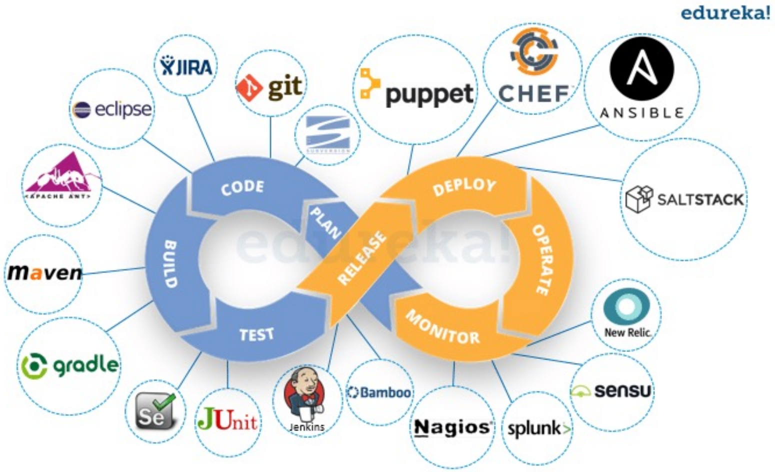
- The **SOA** paradigm suffered from **lack of tooling and automation**

- This **aspects are critical in Microservices Architecture!**

    - Automated Testing

    - Automated Deployment

    - **DevOps!**



- **Automation enables faster development and deployment cycle!**

- **There are a lot of tools available to do this! Testing and deployment can’t be done manually!**

Design for Failure

- With microservices, there are a lot of **independent components that may fail**. We have to **deal with this gracefully!**

- A microservices **team should constantly reflect on how service failures affect the user experience**

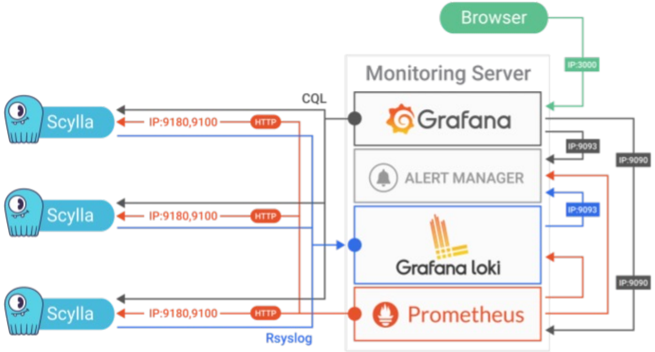
    - E.g.: Netflix usually induces failures of services and datacenters to test the system's resilience and how it behaves in case of a failure

    - If you use **external services, always assume they may fail**, and write code to **deal with it**

- Extensive **logging and monitoring should be in place** to **catch the failures** when they happen and **deal with them**

Design for Failure

- **Logging and Monitoring:**
  - You should **monitor all the system's logs** and have a **central log storage**, that aggregates and processes all these logs.
  - You should **monitor system's metrics**
  - When faced with unusual situations, (e.g. failures) you **should send notifications** to the right team members and **have code to deal with these situations**
  - Example: **Kubernetes** has code to deal with failures and automatically acts in order to keep the system at its desired state



Evolutionary Design

- The **adoption** of Microservices should be **gradual**, and **well analyzed before a decision is made**

- There’s **no need to break everything apart**

- **Start small and upgrade each part separately**

The 9 Pillars of Microservices - Summary

- **The most important attributes:**
  - Componentization
  - Organized around business capabilities
  - Decentralized governance
  - Decentralized data management (when possible)
  - Infrastructure automation

- **These are guidelines, not mandatory instructions**
  - Adopt always what works for your service
  - The Microservices world is rapidly changing

    In a 2 years period some of this information will be deprecated

    Follow and Study new APIs, monitoring tools , cloud services etc.