Microservices and Cloud Based Architecture

Rafael Morales Angoa

Summary

- Overview of Micro-services architecture
- Overview of 12 factor apps methodology
- Introduction to benefits of Cloud architecture

Overview of Micro-services architecture

- Pattern: Monolithic Architecture
- Pattern: Microservice Architecture
- Characteristics of a Microservice Architecture

Context

- You are developing a server-side enterprise application.
- It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications.
- The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker.
- The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response.
- There are logical components corresponding to different functional areas of the application.

Problem

What’s the application’s deployment architecture?

Forces

- There is a team of developers working on the application
- New team members must quickly become productive
- The application must be easy to understand and modify
- You want to practice continuous deployment of the application
- You must run multiple copies of the application on multiple machines in order to satisfy scalability and availability requirements
- You want to take advantage of emerging technologies (frameworks, programming languages, etc)

Pattern: Monolithic Architecture

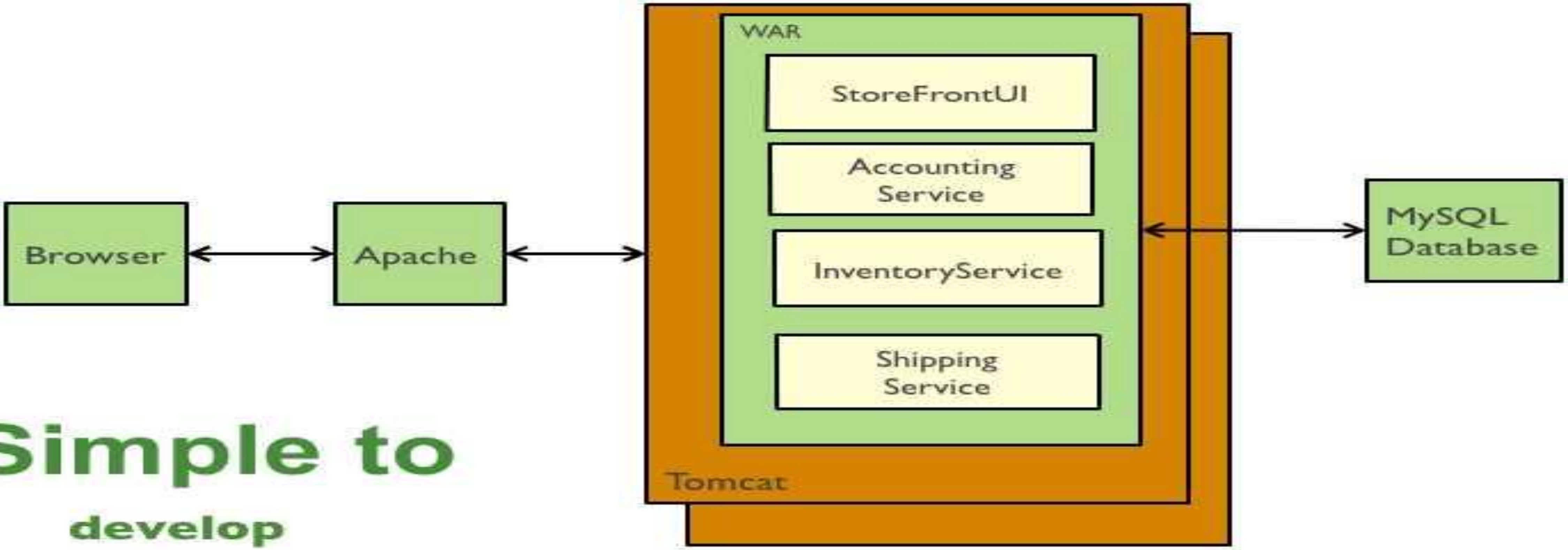
Solution

- Build an application with a monolithic architecture.

For example:

  - a single Java WAR file.
  - a single directory hierarchy of Rails or NodeJS code.

Traditional web application architecture



Simple to

develop
test
deploy
scale

Pattern: Monolithic Architecture

Resulting context

- Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications
- Simple to deploy - you simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime
- Simple to scale - you can scale the application by running multiple copies of the application behind a load balancer

Pattern: Monolithic Architecture

However, once the application becomes large and the team grows in size, this approach has a number of drawbacks that become increasingly significant:

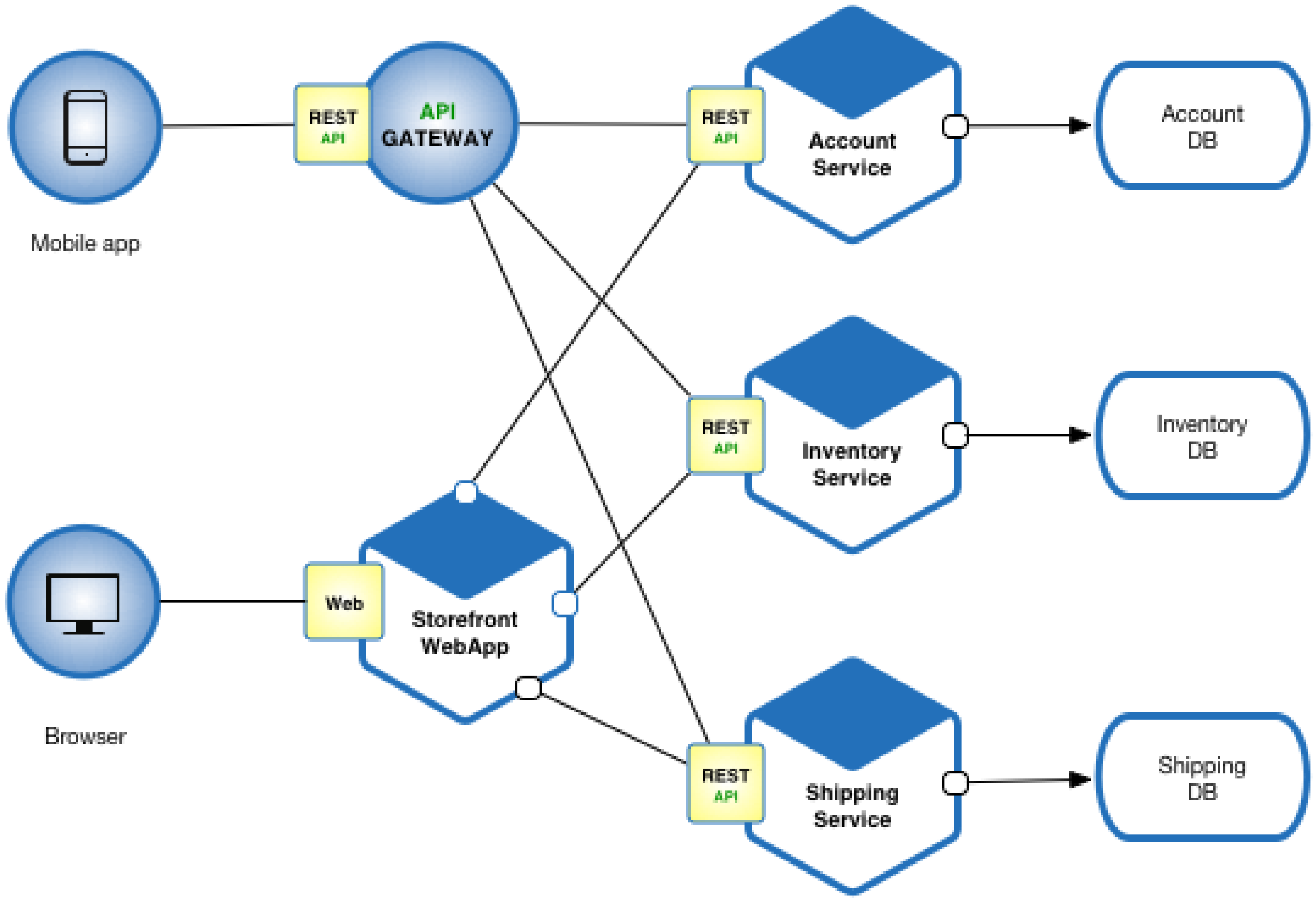
- The large monolithic code base intimidates developers, especially ones who are new to the team.
- The application can be difficult to understand and modify.
- Overloaded IDE - the larger the code base the slower the IDE and the less productive developers are.
- Overloaded web container - the larger the application the longer it takes to start up.
- Continuous deployment is difficult - a large monolithic application is also an obstacle to frequent deployments.
- Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application.
- Obstacle to scaling development - A monolithic application is also an obstacle to scaling development.
- Requires a long-term commitment to a technology stack - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development .
- With a monolithic application, can be difficult to incrementally adopt a newer technology.

Pattern: Microservice Architecture

Solution

- Define an architecture that structures the application as a set of loosely coupled, collaborating services. For example, an application might consist of services such as the order management service, the customer management service etc.
- Services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP.
- Services can be developed and deployed independently of one another.
- Each service has its own database in order to be decoupled from other services.

Pattern: Microservice Architecture



Pattern: Microservice Architecture

This solution has a number of benefits:

- Enables the continuous delivery and deployment of large, complex applications.
- Better testability - services are smaller and faster to test
- Better deployability - services can be deployed independently
- It enables you to organize the development effort around multiple, auto teams. It enables you to organize the development effort around multiple teams. Each team is owns and is responsible for one or more single service. Each team can develop, deploy and scale their services independently of all of the other teams.
- Each microservice is relatively small
- Easier for a developer to understand
- The IDE is faster making developers more productive
- The application starts faster, which makes developers more productive, and speeds up deployments
- Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
- Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack.

What are microservices?

The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

Martin Fowler

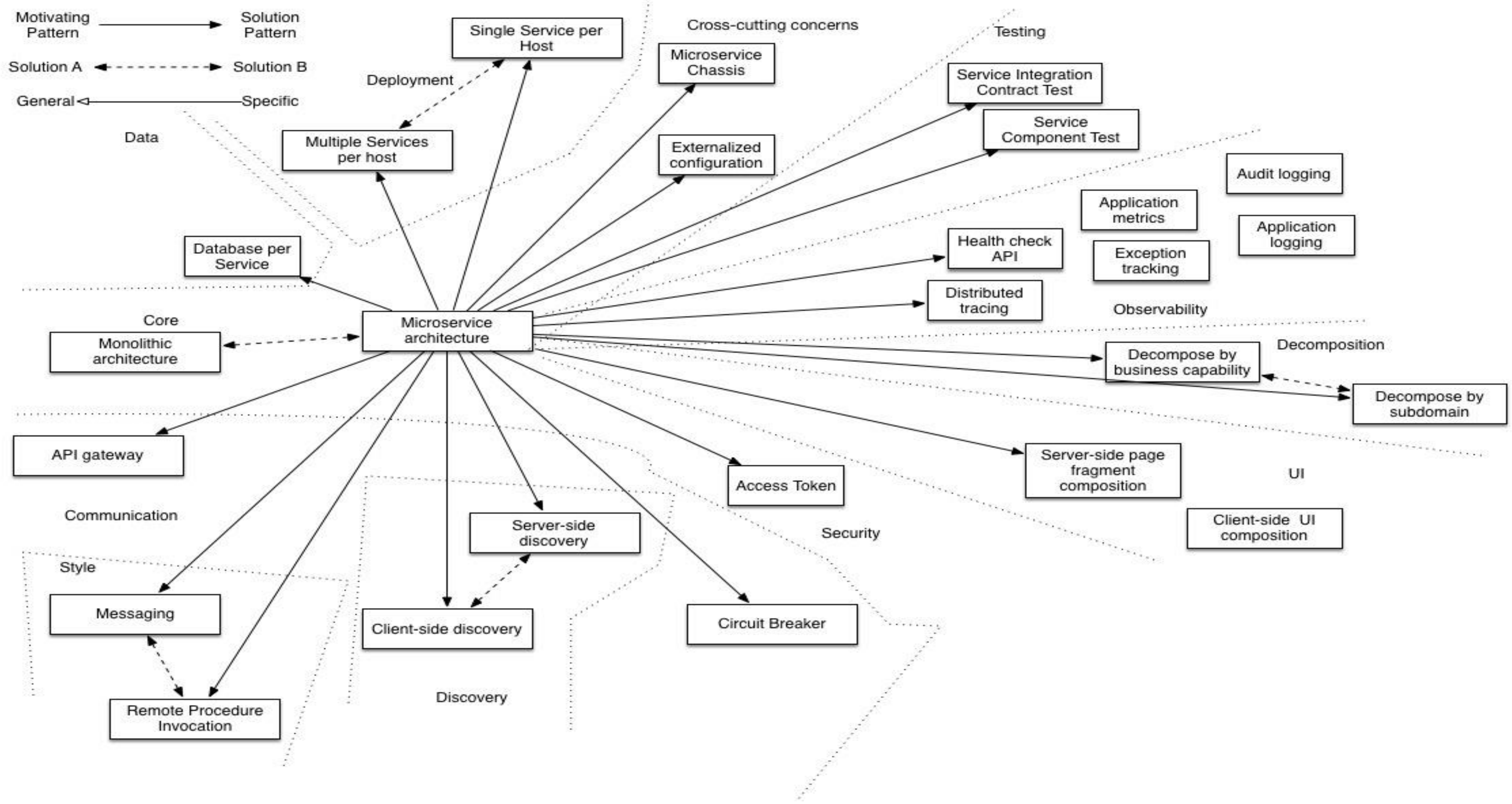
What are microservices?

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack.

microservices.io

Characteristics of a Microservice Architecture

- 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


Motivating Pattern
Solution Pattern
Solution A
Solution B
General
Specific
Data
Deployment
Single Service per Host
Multiple Services per host
Database per Service
Cross-cutting concerns
Microservice Chassis
Externalized configuration
Testing
Service Integration Contract Test
Service Component Test
Audit logging
Application metrics
Application logging
Health check API
Exception tracking
Distributed tracing
Observability
Core
Microservice architecture
Monolithic architecture
Decomposition
Decompose by business capability
Decompose by subdomain
API gateway
Access Token
Server-side page fragment composition
UI
Communication
Server-side discovery
Security
Client-side UI composition
Style
Messaging
Client-side discovery
Circuit Breaker
Remote Procedure Invocation
Discovery

Overview of 12 factor apps methodology

Twelve-Factor App methodology

- The Twelve-Factor App methodology is a methodology for building software as a service applications.
- These best practices are designed to enable applications to be built with portability and resilience when deployed to the web.
- As made apparent by the title, the 12-Factor App methodology is a list of principles, each explaining the ideal way to handle a subset of your application.

The 12-Factor App principles

1. Codebase – One codebase tracked in revision control, many deploys
2. Dependencies – Explicitly declare and isolate dependencies
3. Configuration – Store config in the environment
4. Backing Services – Treat backing services as attached resources
5. Build, release, run – Strictly separate build and run stages
6. Processes – Execute the app as one or more stateless processes
7. Port binding – Export services via port binding
8. Concurrency – Scale out via the process model
9. Disposability – Maximize robustness with fast startup and graceful shutdown
10. Dev/prod Parity – Keep development, staging, and production as similar as possible
11. Logs – Treat logs as event streams
12. Admin processes – Run admin/management tasks as one-off processes

The 12-Factor App principles

Codebase

- The first principle of the 12-Factor App methodology is related to your application's codebase.
- The most important point here is to ensure that your application is tracked with revision control, and that it sits in a central repository that is accessible to your developers.
- This is most commonly handled by using Git or SVN to store your code.

Dependencies

- There is no room for assumptions when it comes to dependencies.
- Anything your applications rely on to run should be controlled and managed to minimize — if not completely eliminate — conflicts.

The 12-Factor App principles

Configuration

- Configuration, as it relates to API keys, services, and database credentials, should never be hardcoded.
- This prevents your application from being at risk from both production data leaks and production errors. Instead of hardcoding this information, rely on environment variables to handle this sensitive information.

Backing Services

- A backing service is one that requires a network connection to run.
- This is a very popular paradigm found in modern application development, especially prevalent with the rise in popularity of microservice architecture.
- The methodology advises developers to treat these services agnostically, meaning changes or modifications should occur without having to make any code changes.
- Typically, this factor is best handled by calling each backing service through an API, with credentials stored in a configuration file that lives in your runtime environment.

The 12-Factor App principles

Build, release, run

- Build, release, and run stages should be treated as completely distinct from one another. Automation and tooling will help to make this principle simpler.
- This can be accomplished by using existing tools to fully automate your build process.
- A tool like git or RTC can be used to tag your latest build, while Jenkins can be used to automate your release stage.

Processes

- Stateless applications are designed to degrade gracefully.
- That means if a dependency fails, the app itself does not become a failure.
- Single points of failure may be difficult, but not impossible, to avoid.
- The methodology recommends storing data outside of running code in order to prevent operational headaches and debugging nightmares.

The 12-Factor App principles

Port binding

- All application services should be accessible via a URL.
- For web applications, this process happens automatically.
- This enables 12-Factor Apps to be fully self-contained, avoiding the need to rely on various methods of runtime injection in order to create web facing services.

Concurrency

- Every process inside your application should be treated as a first-class citizen.
- That means that each process should be able to scale, restart, or clone itself when needed.
- This approach will improve the sustainability and scalability of your application as a whole.

The 12-Factor App principles

Disposability

- As noted in the previous factor, treating processes as first-class citizens translates to an easier startup and shutdown process.
- Compare this to an application where all process are bundled together, where startup and shutdown processes can take up to several minutes depending on their size.

Dev/prod Parity

- Consistency is key for meeting this factor. When your environments are similar, testing and developing gets much simpler.
- Similar environments means ensuring that areas such as your infrastructure stack, config management processes, software and runtime versions and deployment tools are the same everywhere.
- With this approach, fewer bugs will find their way into your production environment, since your test cases can be applied on production-level data.

The 12-Factor App principles

Logs

- Logging is important for debugging and checking up on the general health of your application.
- At the same time, your application shouldn’t concern itself with the storage of this information. Instead, these logs should be treated as a continuous stream that is captured and stored by a separate service.

Admin processes

- One-off admin processes are essentially data collection jobs that are used to gather key information about your application.
- This information will be needed to asses the state of your production environment, so it’s important to ensure these one-off processes occur in your production environment.
- That way there can be no discrepancies between the data you need and the data coming from the visible long running production application.

The twelve-factor app

Today, software is commonly delivered as a service: called *web apps*, or *software-as-a-service*.
The twelve-factor app is a methodology for building software-as-a-service apps that:

- Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project.
- Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments.
- Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration.
- **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility.
- And can **scale up** without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

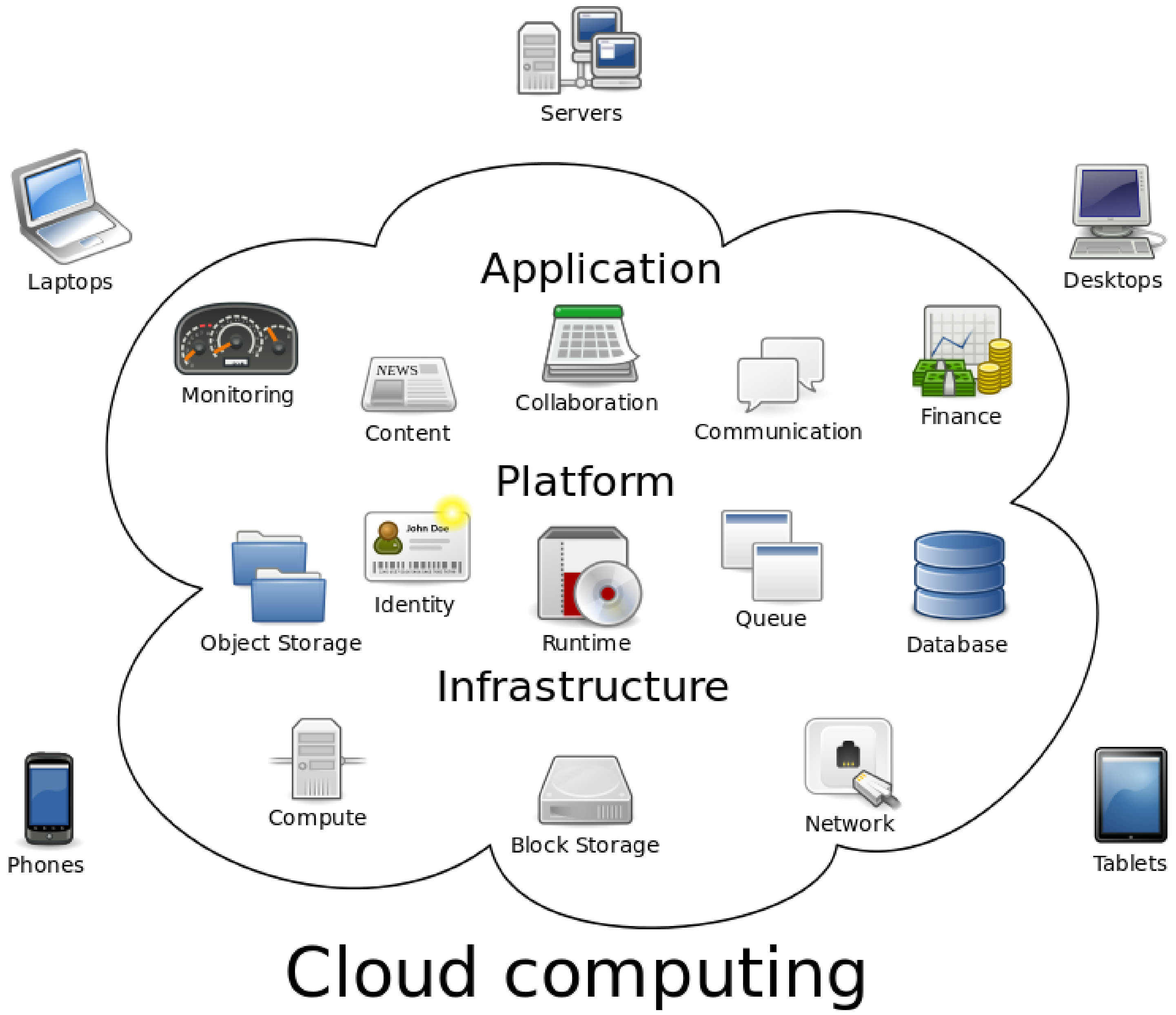
https://12factor.net/

Introduction to benefits of Cloud architecture

Cloud computing

Is an information technology (IT) paradigm that enables **ubiquitous** access to shared pools of configurable system resources and higher-level services that can be rapidly provisioned with minimal management effort, often over the Internet.

Cloud computing relies on sharing of resources to achieve coherence and economies of scale, similar to a public utility.



Benefits of Cloud based architecture

- Your application is experiencing increased traffic and it’s becoming difficult to scale resources on the fly to meet the increasing demand.
- You need to reduce operational costs, while increasing the effectiveness of IT processes.
- Your clients require fast application implementation and deployment, and thus want to focus more on development while reducing infrastructure overhead.
- Your clients want to expand their business geographically, but you suspect that setting up a multi-region infrastructure – with all the associated maintenance, time, human, and error control effort – is going to be a challenge.

Benefits of Cloud based architecture

- It’s becoming more difficult and expensive to keep up with your growing storage needs.
- You’d like to build a widely distributed development team. Cloud computing environments allow remotely located employees to access applications and work via the Internet.
- You need to establish a disaster recovery system but setting it up for an entire data center could double the cost. It would also require a complex disaster recovery plan. Cloud disaster recovery systems can be implemented much more quickly and give you much better control over your resources.

- Cloud computing shifts IT expenditure to a pay-as-you-go model, which is an attractive benefit, especially for startups.

- Tracking and upgrading underlying server software is a time consuming, yet essential process that requires periodic and sometimes immediate upgrades. In some cases, a cloud provider will take care of this automatically. Some cloud computing models similarly handle many administration tasks such as database backup, software upgrades, and periodic maintenance.

Drive down costs

- Avoid large capital expenditure on hardware and upgrades. Cloud can also improve cost efficiency by more closely matching your cost pattern to your revenue/demand pattern, moving your business from a capital-intensive cost model to an Opex model.

Cope with demand

- You know what infrastructure you need today, but what about your future requirements? As your business grows, a cloud environment should grow with you. And when demand is unpredictable or you need to test a new application, you have the ability spin capacity up or down, while paying only for what you use.

Run your business; don’t worry about your IT

- Monitoring your infrastructure 24/7 is time consuming and expensive when you have a business to run. A managed cloud solution means that your hosting provider is doing this for you. In addition to monitoring your infrastructure and keeping your data safe, they can provide creative and practical solutions to your needs, as well as expert advice to keep your IT infrastructure working efficiently as your needs evolve.

Innovate and lead

- Ever-changing business requirements mean that your IT infrastructure has to be flexible. With a cloud infrastructure, you can rapidly deploy new projects and take them live quickly, keeping you at the vanguard of innovation in your sector.

Improved security and compliance

- You have to protect your business against loss of revenue and brand damage. In addition, many organizations face strict regulatory and compliance obligations. A cloud environment means that this responsibility no longer rests entirely on your shoulders. Your cloud hosting provider will build in resiliency and agility at an infrastructure-level to limit the risk of a security breach, and will work with you to help address compliance and regulatory requirements.

Reduce your carbon footprint

- Hosting in a data center rather than onsite allows you to take advantage of the latest energy-efficient technology. Additionally, as cloud service providers host multiple customers on shared infrastructure, they can drive higher and more efficient utilization of energy resources.

References