SUSE

Cloud Native Fundamentals Scholarship Program

Prepared by Sani Kamal

Lesson 2

Architecture Consideration for Cloud Native Applications

Learn about monoliths and microservices, their differences, and their pros and cons.

CONCEPTS

Introduction

Summary

Welcome to the Cloud Native Fundamentals course!

Before building an application, it is common to go through a design phase to identify the main requirements and structure of an application. In correlation with the available resources, a team will choose the most suitable project architecture.

In the industry, usually, the two main approaches that are usually referenced are **monoliths and microservices**. In this lesson, we will explore each architectural model and the implied trade-offs. As well, we will cover good development practices to be considered if an application is targeted for containerization. These practices are valid for both monolithic and microservice architectures.

Architecture Considerations lesson outline

In this lesson, we will cover:

- Monoliths and Microservices
- Trade-offs for Monoliths and Microservices
- Practices for Application Development

Design Considerations for Cloud-Native Applications

Summary

When building an application it is important to allocate time for context discovery. This includes listing all necessary functionalities of the application and enumerating any resources that can enable its build-out. This phase sets the fundamentals of the project. If properly implemented, it can enable the creation of services that are scalable, resilient, and extensible.

The first step in the context discovery process is to list the **functional requirements**, or what application capabilities should deliver to the end-users. For example, a good starting point is to expand on the following:

- Stakeholders
- Functionalities
- End users
- Input and output process
- Engineering teams

The second step is to enumerate the **available resources** that facilitates the implementation of the project. For example, a good starting point is to list available:

- Engineering resources
- Financial resources
- Timeframes
- Internal knowledge

Having a good understanding of functional requirements and available resources can lead to a simpler choice between monolithic and microservice-based architectures.

Monoliths and Microservices

Summary

Prior to an organization delivers a product, the engineering team needs to decide on the most suitable application architecture. In most of the cases 2 distinct models are referenced: monoliths and microservices. Regardless of the adopted structure, the main goal is to design an application that delivers value to customers and can be easily adjusted to accommodate new functionalities.

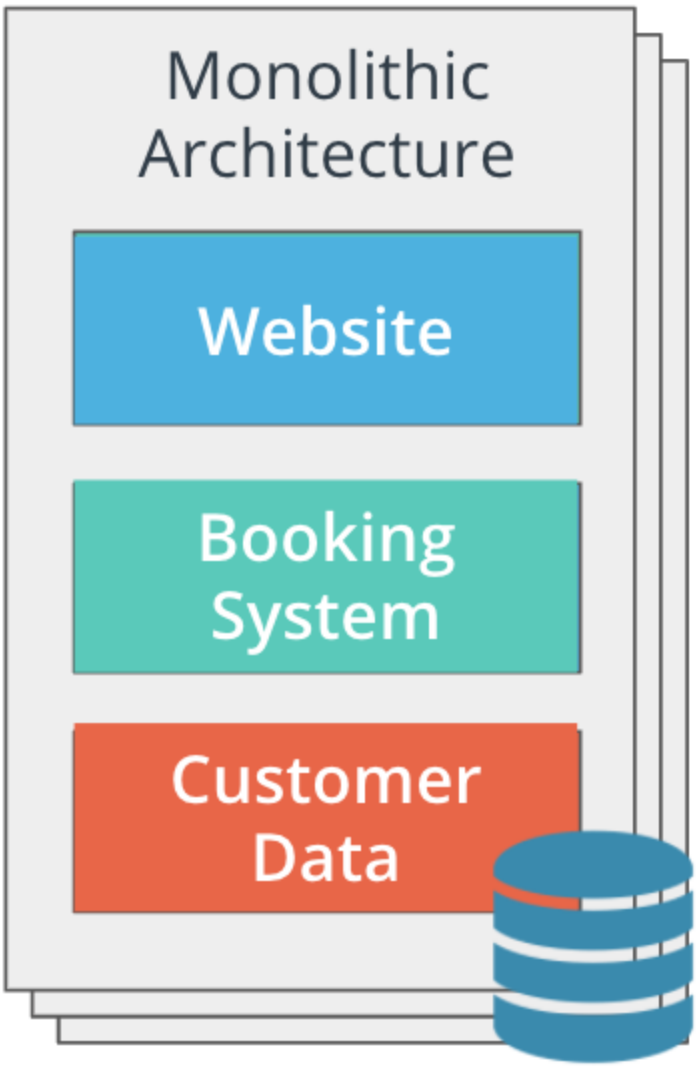
Also, each architecture encapsulates the 3 main tires of an application:

- UI (User Interface) - handles HTTP requests from the users and returns a response
- Business logic - contained the code that provides a service to the users
- Data layer - implements access and storage of data objects

Monoliths

In a monolithic architecture, application tiers can be described as:

- part of the same unit
- managed in a single repository
- sharing existing resources (e.g. CPU and memory)
- developed in one programming language
- released using a single binary



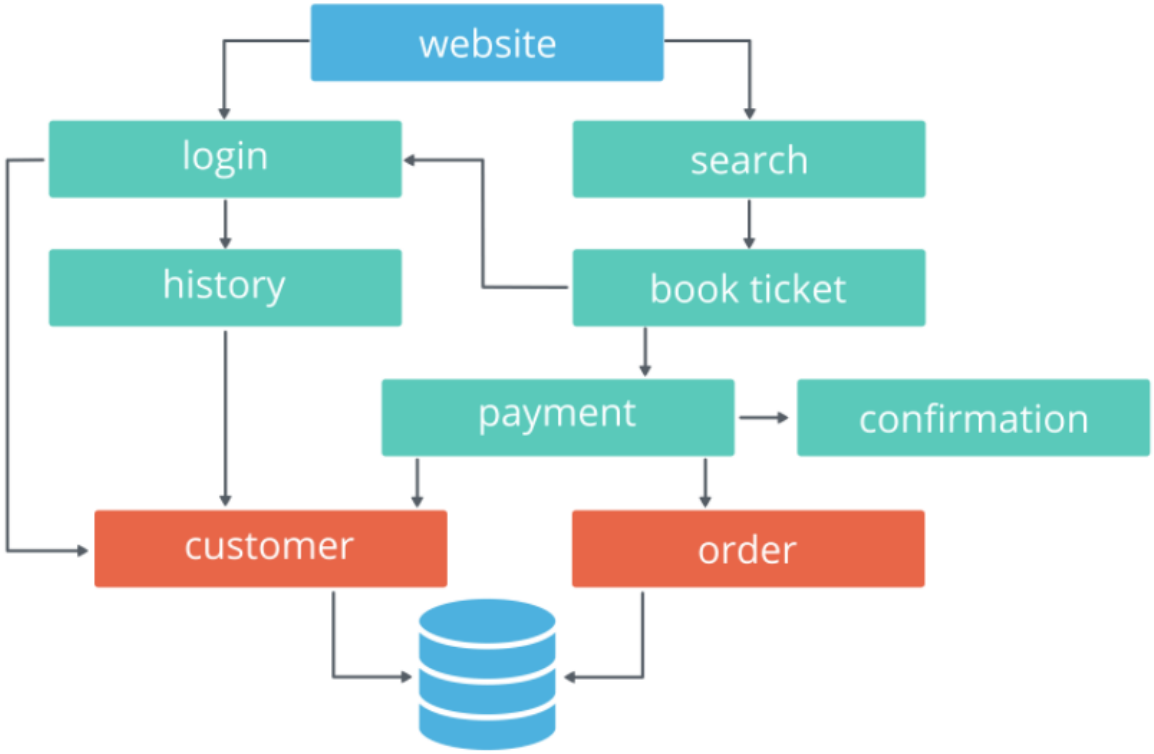
A booking application referencing the monolithic architecture

Imagine a team develops a booking application using a monolithic approach. In this case, the UI is the website that the user interacts with. The business logic contains the code that provides the booking functionalities, such as search, booking, payment, and so on. These are written using one programming language (e.g. Java or Go) and stored in a single repository. The data layer contains functions that store and retrieve customer data. All of these components are managed as a unit, and the release is done using a single binary.

Microservices

In a microservice architecture, application tiers are managed independently, as different units. Each unit has the following characteristics:

- managed in a separate repository
- own allocated resources (e.g. CPU and memory)
- well-defined API (Application Programming Interface) for connection to other units
- implemented using the programming language of choice
- released using its own binary



A booking application referencing the microservice architecture

Now, let's imagine the team develops a booking application using a microservice approach.

In this case, the UI remains the website that the user interacts with. However, the business logic is split into smaller, independent units, such as login, payment, confirmation, and many more. These units are stored in separate repositories and are written using the programming language of choice (e.g. Go for the payment service and Python for login service). To interact with other services, each unit exposes an API. And lastly, the data layer contains functions that store and retrieve customer and order data. As expected, each unit is released using its own binary.

New terms

- **Monolith**: application design where all application tiers are managed as a single unit
- **Microservice**: application design where application tiers are managed as independent, smaller units

Further reading

- What's the Difference Between Monolith and Microservices?
- Microservices vs Monolithic Architecture

Quizzes: Monoliths and Microservices

Trade-offs for Monoliths and Microservices

Summary

An application can be designed using both, monolithic or microservice-based architectures. So far, we have looked at how the structure of an application is contoured by functional requirements and available resources. However, each architecture has a set of trade-offs, that need to be thoroughly examined before deciding on the final structure of the application.

These trade-offs cover development complexity, scalability, time to deploy, flexibility, operational cost, and reliability. Let's examine each trade-off in more detail!

Summary

Development Complexity

Development complexity represents the effort required to deploy and manage an application.

- **Monoliths** - one programming language; one repository; enables sequential development
- **Microservice** - multiple programming languages; multiple repositories; enables concurrent development

Scalability

Scalability captures how an application is able to scales up and down, based on the incoming traffic.

- **Monoliths** - replication of the entire stack; hence it's heavy on resource consumption
- **Microservice** - replication of a single unit, providing on-demand consumption of resources

Time to Deploy

Time to deploy encapsulates the build of a delivery pipeline that is used to ship features.

- **Monoliths** - one delivery pipeline that deploys the entire stack; more risk with each deployment leading to a lower velocity rate
- **Microservice** - multiple delivery pipelines that deploy separate units; less risk with each deployment leading to a higher feature development rate

Flexibility

Flexibility implies the ability to adapt to new technologies and introduce new functionalities.

- **Monoliths** - low rate, since the entire application stack might need restructuring to incorporate new functionalities
- **Microservice** - high rate, since changing an independent unit is straightforward

Operational Cost

Operational cost represents the cost of necessary resources to release a product.

- **Monoliths** - low initial cost, since one code base and one pipeline should be managed. However, the cost increases exponentially when the application needs to operate at scale.
- **Microservice** - high initial cost, since multiple repositories and pipelines require management. However, at scale, the cost remains proportional to the consumed resources at that point in time.

Reliability

Reliability captures practices for an application to recover from failure and tools to monitor an application.

- **Monoliths** - in a failure scenario, the entire stack needs to be recovered. Also, the visibility into each functionality is low, since all the logs and metrics are aggregated together.
- **Microservice** - in a failure scenario, only the failed unit needs to be recovered. Also, there is high visibility into the logs and metrics for each unit.

Each application architecture has a set of trade-offs that need to be considered at the genesis of a project. But more importantly, it is paramount to understand how the application will be maintained in the future e.g. at scale, under load, supporting multiple releases a day, etc.

There is no "golden path" to design a product, but a good understanding of the trade-offs will provide a clear project roadmap. Regardless if a monolith or microservice architecture is chosen, as long as the project is coupled with an efficient delivery pipeline, the ability to adopt new technologies, and easily add features, the path to could-native deployment is certain.

Quizzes: Trade-offs for Monoliths and Microservices

Exercise: Trade-offs for Monoliths and Microservices

Outline the architecture of an application

From the early stages of application development, it is fundamental to understand the requirements and available resources. Overall, these will contour the architecture decisions.

Imagine this scenario: you are part of the team that needs to outline the structure of a centralized system to book flight tickets for different airlines. At this stage, the clients require the *front-end(UI), payment, and customer functionalities to be designed*. Also, these are the *individual requirements* of each airline:

- Airline A - payments should be allowed only through PayPal
- Airline B - payments should be disabled (bookings will be exclusively in person or via telephone)

- Airline C - payments should be allowed to use PayPal and debit cards

Using the above requirements, outline the application architecture. Also, elaborate your reasoning on choosing a microservice or monolith based approach.

Solution: Monoliths and Microservices

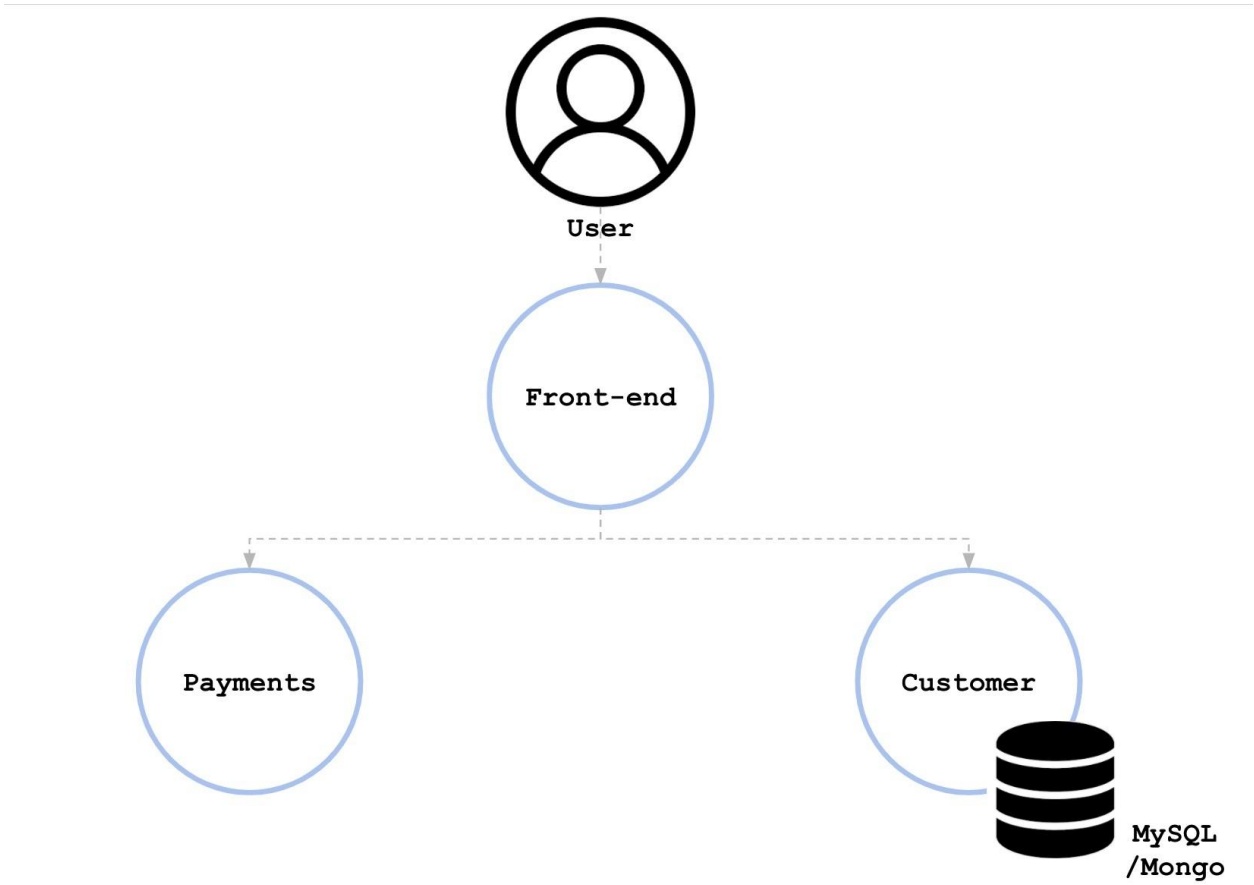
Summary

Given the scenario, it is paramount to choose an architecture that would be **replicable and scalable**. For example, if thousands of customers access the payment service in the same timeframe, then this particular service should be scaled up. In a monolith architecture, scaling up creates a replica of everything, including front-end and customer services, in addition to the payment service. This will also consume more resources on the platform, such as CPU and memory, and takes longer to spin up.

On the other side, a microservice is a lightweight component that requires fewer resources (CPU and memory) and less time for provisioning.

For this example, a microservice-based architecture is chosen, based on considerations that the application is a central booking system for multiple airlines, that implies a high load. The main components are:

- Front-end - entry-point for the user, where they will choose their airline or choice
- Customer - requires a database (MySQL or Mongo) to store the customer details
- Payments - to implement PayPal and Debit based operations

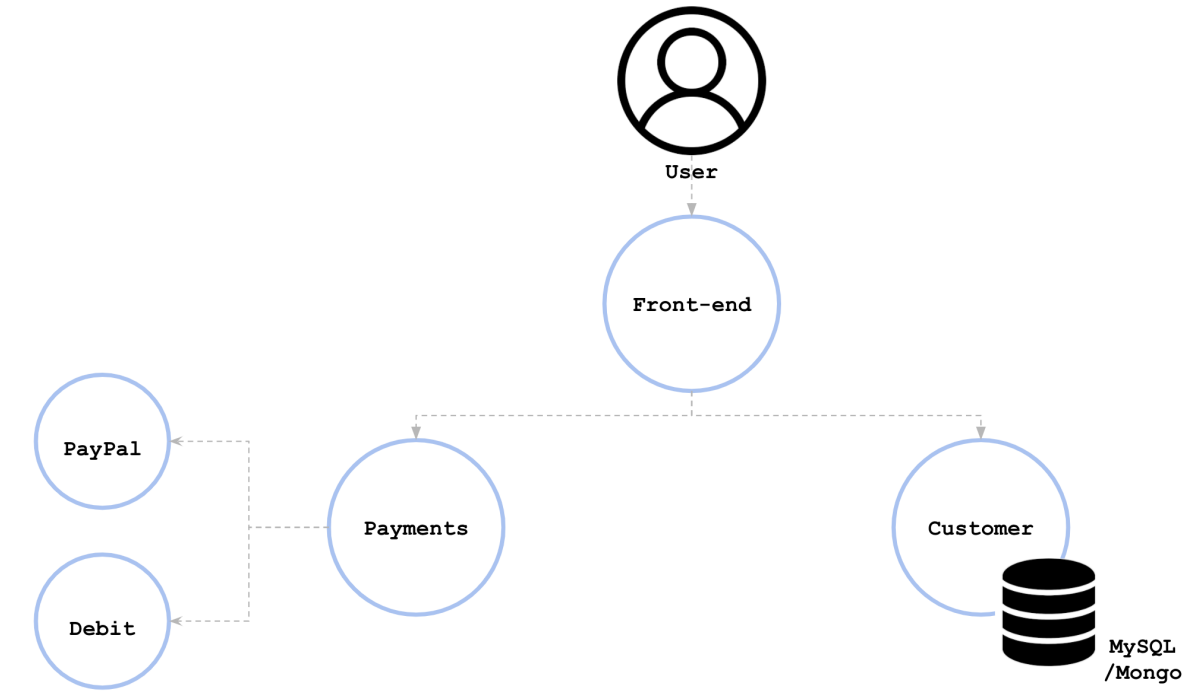


Flight booking application using microservice architecture

Additionally, the "payments" microservice is capable of handling multiple payment systems. Interaction with the PayPal interface and management of debit card APIs are fundamentally different. The "payments" component is a monolith that can be divided into multiple parts.

Payments:

- PayPal - handling all the PayPal payments
- Debit - handling all the debit card payments



Payment service being split into PayPal and Debit microservices

Best Practices For Application Deployment

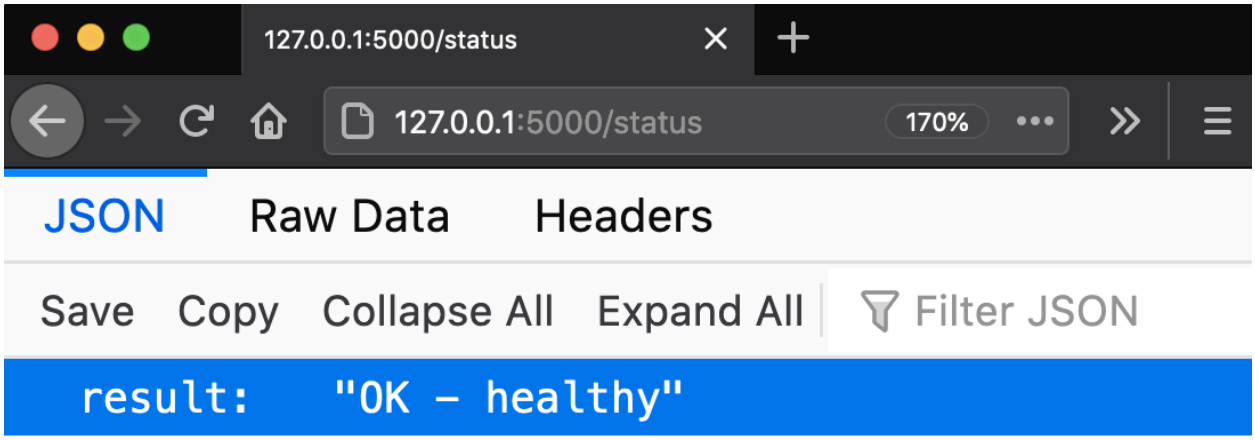
Summary

Using the knowledge acquired so far, you should be able to choose the most suitable architecture for an application, based on requirements, available resources, and involved trade-offs. The next stage consists of building the application. Regardless of the chosen architecture, a set of good development practices can be applied to improve the application lifecycle throughout the release and maintenance phases. Adopting these practices increases resiliency, lowers the time to recovery, and provides transparency of how a service handles incoming requests.

These practices are focused on health checks, metrics, logs, tracing, and resource consumption.

Health Checks

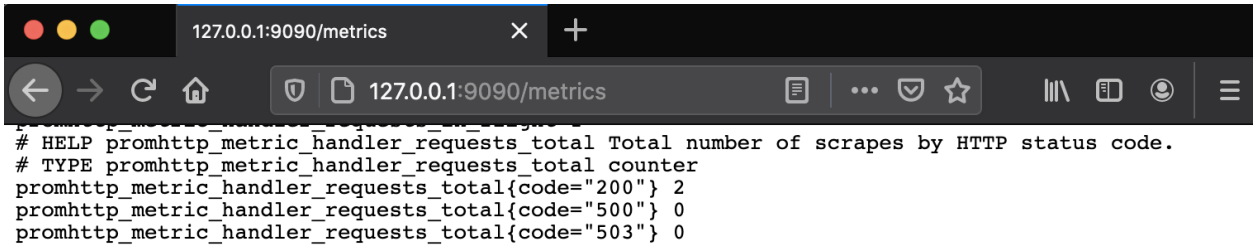
Health checks are implemented to showcase the status of an application. These checks report if an application is running and meets the expected behavior to serve incoming traffic. Usually, health checks are represented by an HTTP endpoint such as /healthz or /status. These endpoints return an HTTP response showcasing if the application is healthy or in an error state.



/status health check that showcases that the application is healthy

Metrics

Metrics are necessary to quantify the performance of the application. To fully understand how a service handles requests, it is mandatory to collect statistics on how the service operates. For example, the number of active users, handled requests, or the number of logins. Additionally, it is paramount to gather statistics on resources that the application requires to be fully operational. For example, the amount of CPU, memory, and network throughput. Usually, the collection of metrics are returned via an HTTP endpoint such as /metrics, which contains the internal metrics such as the number of active users, consumed CPU, network throughput, etc.



/metrics endpoint that list of metrics counting the amount requests by the HTTP code returned

Logs

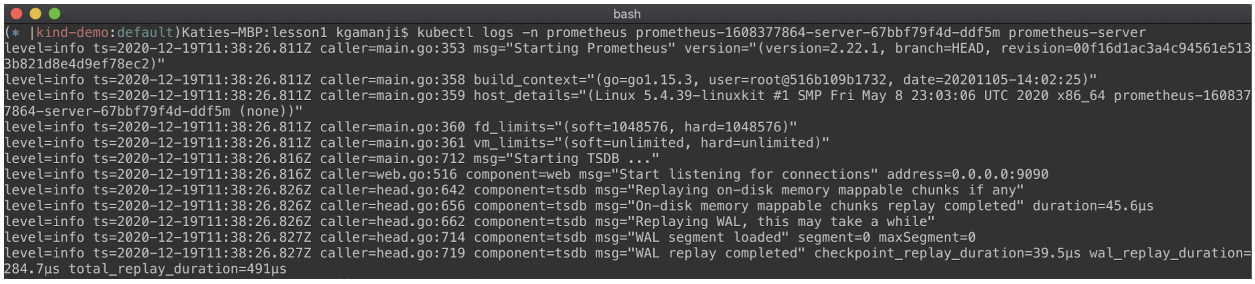
Log aggregation provides valuable insights into what operations a service is performing at a point in time. It is the nucleus of any troubleshooting and debugging process. For example, it is essential to record if a user logged in successfully into a service, or encountered an error while performing a payment.

Usually, the logs are collected from STDOUT (standard out) and STDERR (standard error) through a passive logging mechanism. This means that any output or errors from the application are sent to the shell. Subsequently, these are collected by a logging tool, such as Fluentd or Splunk, and stored in backend storage. However, the application can send the logs directly to the backend storage. In this case, an active logging technique is used, as the log transmission is handled directly by the application, without a logging tool required.

There are multiple logging levels that can be attributed to an operation. Some of the most widely used are:

- **DEBUG** - record fine-grained events of application processes
- **INFO** - provide coarse-grained information about an operation
- **WARN** - records a potential issue with the service
- **ERROR** - notifies an error has been encountered, however, the application is still running
- **FATAL** - represents a critical situation, when the application is not operational

As well, it is common practice to associate each log line with a **timestamp**, that will exactly record when an operation was invoked.



Multiple INFO log lines recorded when a Prometheus service started

Tracing

Tracing is capable of creating a full picture of how different services are invoked to fulfill a single request. Usually, tracing is integrated through a library at the application layer, where the developer can record when a particular service is invoked. These records for individual services are defined as spans. A collection of spans define a trace that recreates the entire lifecycle of a request.

Resource Consumption

Resource consumption encapsulates the resources an application requires to be fully operational. This usually refers to the amount of CPU and memory that is consumed by an application during its execution. Additionally, it is beneficial to benchmark the network throughput, or how many requests can an application handle concurrently. Having awareness of resource boundaries is essential to ensure that the application is up and running 24/7.

A graph showcasing the CPU consumption of the **coredns** container

Further reading

- Health Checks - explore the core reasons to introduce health checks and implementations examples
- Prometehus Best Practices on Metrics Naming - explore how to name, label, and define the type of metrics
- Application Logging Best Practices - read more on how to define what logs should be collected by an application
- Logging Levels - explore possible logging levels and when they should be enabled
- Enabling Distributed Tracing for Microservices With Jaeger in Kubernetes - learn what tools can be used to implement tracing in a Kubernetes cluster

Quizzes: Best Practices For Application Deployment

Exercise: Endpoints for Application Status

This exercise aims to extend a Python Flask web application with status and metrics endpoints.

Environment Setup

Set up your environment to extend a Python Flask application:

Task List

Once all the pre-requisites are completed, you can get started on developing endpoints to describe the application state.

Exercise

Extend the Python Flask application with /status and /metrics endpoints, considering the following requirements:

- Both endpoints should return an HTTP 200 status code
- Both endpoints should return a JSON response e.g. {"user": "admin"}. (Note: the JSON response can be hardcoded at this stage)
- The /status endpoint should return a response similar to this example: result: OK - healthy
- The /metrics endpoint should return a response similar to this example: data: {UserCount: 140, UserCountActive: 23}

Tips: If you get stuck, feel free to check the solution video where detailed operations are demonstrated.

Solution: Endpoints for Application Status

The following snippet showcases an example of a Python Flask application, with /metrics, /status and main page (/) endpoints:

```
    from flask import Flask
    from flask import json

    app = Flask(__name__)

    @app.route('/status')
    def status():
        response = app.response_class(
                response=json.dumps({"result":"OK - healthy"}),
                status=200,
                mimetype='application/json'
        )

        return response

    @app.route('/metrics')
    def metrics():
        response = app.response_class(

response=json.dumps({"status":"success","code":0,"data":{"UserCount":140,"
UserCountActive":23}}),
                status=200,
                mimetype='application/json'
        )

        return response

    @app.route("/")
    def hello():
        return "Hello World!"

    if __name__ == "__main__":
        app.run(host='0.0.0.0')
```

Exercise: Application Logging

Logging is a core factor in increasing the visibility and transparency of an application. When in troubleshooting or debugging scenarios, it is paramount to pin-point the functionality that impacted the service. This exercise will focus on bringing the logging capabilities to an application.

At this stage, you have extended the Hello World application to handle different endpoints. Once an endpoint is reached, a log line should be recorded showcasing this operation.

In this exercise, you need to further develop the Hello World application collect logs, with the following requirements:

- A log line should be recorded the timestamp and the requested endpoint e.g. "{{TIMESTAMP}}, {{ ENDPOINT_NAME}} endpoint was reached"
- The logs should be stored in a file with the name app.log. Refer to the logging Python module for more details.
- Enable the collection of Python logs at the DEBUG level. Refer to the logging Python module for more details.

Note: For the environment setup, follow the instructions in the previous exercise.

Tips: If you get stuck, feel free to check the solution video where detailed operations are demonstrated.

Solution: Application Logging

Summary

The following snippet showcases how logging would be implemented for each endpoint:

```
from flask import Flask
from flask import json
import logging

app = Flask(__name__)

@app.route('/status')
```

```
    def healthcheck():
        response = app.response_class(
                response=json.dumps({"result":"OK - healthy"}),
                status=200,
                mimetype='application/json'
        )

        ## log line
        app.logger.info('Status request successfull')
        return response

    @app.route('/metrics')
    def metrics():
        response = app.response_class(

response=json.dumps({"status":"success","code":0,"data":{"UserCount":140,"
UserCountActive":23}}),
                status=200,
                mimetype='application/json'
        )

        ## log line
        app.logger.info('Metrics request successfull')
        return response

    @app.route("/")
    def hello():
        ## log line
        app.logger.info('Main request successfull')

        return "Hello World!"

    if __name__ == "__main__":

        ## stream logs to app.log file
        logging.basicConfig(filename='app.log',level=logging.DEBUG)

        app.run(host='0.0.0.0')
```

Edge Case: Amorphous Applications

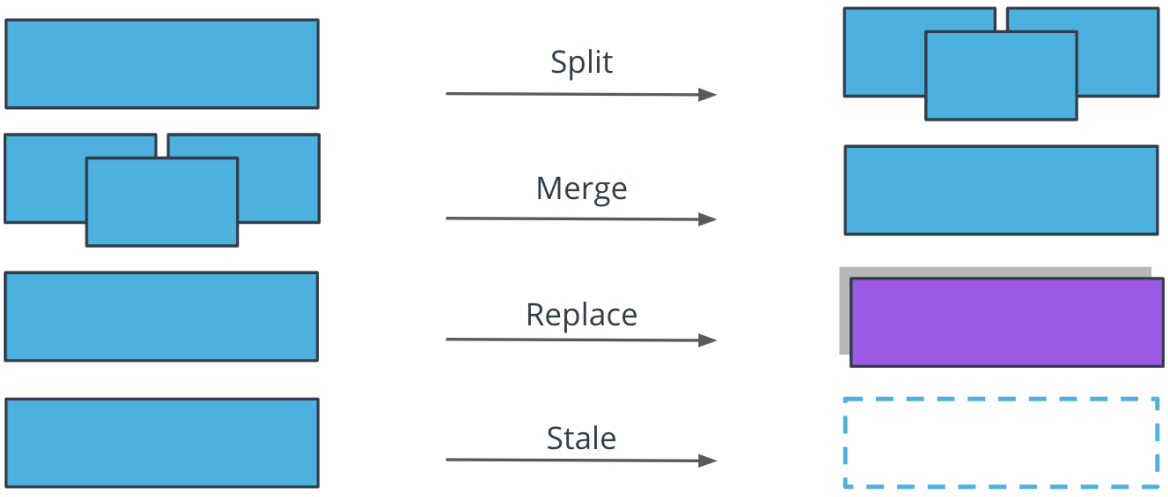
In the previous sections, we have explored how to choose a suitable architecture for an application and how to apply some of the best development practices. However, this is only the start of the application lifecycle. After an engineering team has successfully released a product, with both monolith and microservices, the next phase in the application lifecycle is **maintenance**. In this edge case, we will explore commonly used maintenance operations after a product is released.

Summary

Throughout the maintenance stage, the application structure and functionalities can change, and this is expected! The architecture of an application is not static, it is amorphous and in constant movement. This represents the organic growth of a product that is responsive to customer feedback and new emerging technologies.

Both monolith and microservice-based applications transition in the maintenance phase after the production release. When considering adding new functionalities or incorporating new tools, it is always beneficial to focus on **extensibility rather than flexibility**. Generally speaking, it is more efficient to manage multiple services with a well-defined and simple functionality (as in the case of microservices), rather than add more abstraction layers to support new services (as we've seen with the monoliths). However, to have a well-structured maintenance phase, it is essential to understand the reasons an architecture is chosen for an application and involved trade-offs.

Some of the most encountered operations in the maintenance phase are listed below:



Application operations to occur in the maintenance phase

- A **split** operation - is applied if a service covers too many functionalities and it's complex to manage. Having smaller, manageable units is preferred in this context.
- A **merge** operation- is applied if units are too granular or perform closely interlinked operations, and it provides a development advantage to merge these together. For example, merging 2 separate services for log output and log format in a single service.
- A **replace** operation - is adopted when a more efficient implementation is identified for a service. For example, rewriting a Java service in Go, to optimize the overall execution time.
- A **stale** operation - is performed for services that are no longer providing any business value, and should be archived or deprecated. For example, services that were used to perform a one-off migration process.

Performing any of these operations increases the longevity and continuity of a project. Overall, the end goal is to ensure the application is providing value to customers and is easy to manage by the engineering team. But more importantly, it can be observed that the structure of a project is not static. It amorphous and it evolves based on new requirements and customer feedback.

Further reading

- Modern Banking in 1500 Microservices - watch how Monzo is managing thousands of microservices and evolves their ecosystem

Lesson Conclusion

Summary

In this lesson, we have covered how to build an application using monolith and microservice-based architecture. The choice of an application structure is highly impacted by available resources, requirements, and involved trade-offs. But more importantly, we have covered development practices that should be considered to optimize an application's resilience, time to recovery, and traceability.

Overall, in this lesson, we covered:

- Monoliths and Microservices
- Trade-offs for Monoliths and Microservices
- Practices for Application Development

Glossary

- **Monolith**: application design where all application tiers are managed as a single unit
- **Microservice**: application design where application tiers are managed as independent, smaller units