Class #3

02. Data privacy exercise web app

Software Architectures

Master in Informatics Engineering

Cláudio Teixeira (claudio@ua.pt)

From last class:
Deploying Observability

60 minutes
10 min class discussion

https://docs.google.com/document/d/10VDvDJZLJkAXhWVBG2sBTn7_QiOZRDhdikUhY5y0KpI/edit?usp=sharing

Agenda

- Loose coupling, High cohesion, SOC and SOLID
- Distributed Systems
  - Choreography
  - Security
  - Observability
- Work on Assignment #1
  - individual

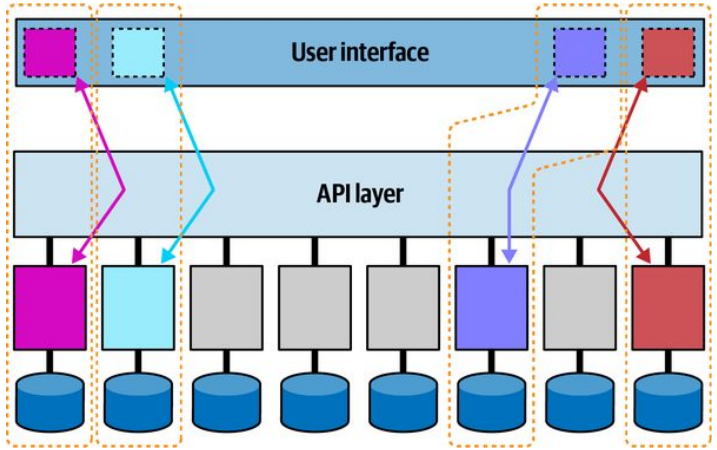
My O'Reilly Playlist for Software Architecture Course
https://learning.oreilly.com/playlists/e4bbd2ea-5604-419c-8f40-2b0a06e7f8aa

Micro-frontends

The concept of a loosely coupled architecture evolved to include a related concept: **micro-frontends**. As the name suggests, it aims to bring the same principles used in microservices to the realm of frontend development.

In a micro-frontend environment, different teams build different parts of the UIs that connect to different microservices. For example, in a digital banking portal, a bank may provide a number of customer offerings, such as online accounts, credit card, loans, and investment. Each of these business offerings could be evolved independently by different teams that work in isolation.

Micro-frontends give front-end development teams the freedom they need to address those differences quickly and effectively — keeping teams focused and productive, and giving the bank an important competitive differentiator.



Principle of least knowledge
Law of Demeter (LoD)

The LoD promotes loose coupling in software design by limiting an object's interactions to a close set of related objects.

Specifically, an object should only call methods of objects it directly holds, objects passed as parameters, or objects it creates.

This principle discourages reaching through objects to access others, aiming to reduce dependencies and enhance modularity.

Following LoD can make software more maintainable and adaptable by minimizing knowledge of other objects' internal structures

Aim for loose coupling
Get High cohesion

Loose coupling and high cohesion are fundamental principles in software architecture, aiming to create flexible, maintainable systems.

Coupling refers to the degree of direct knowledge one component has of another, where loose coupling minimizes dependencies, allowing changes with minimal impact.

Cohesion denotes how closely related and focused the responsibilities of a single module are.

High cohesion means that a module's operations are closely related to each other, improving readability, reusability, and updateability.

Achieving high cohesion within modules, while maintaining loose coupling between them, enhances the quality and longevity of software.

Cohesion types

Cohesion types, from the lowest (least desirable) to the highest (most desirable):

1. Coincidental Cohesion (Least Desirable)
2. Logical Cohesion
3. Temporal Cohesion
4. Procedural Cohesion

(acceptable threshold barrier should be here)

5. Communicational Cohesion
6. Sequential Cohesion
7. Functional Cohesion

Coincidental Cohesion

A random collection of unrelated functions within the same module.

Example: A Miscellaneous class that combines unrelated functionalities like managing application settings, parsing user input, and sending email notifications.

How to improve cohesion: Divide the Miscellaneous class into distinct classes where each handles a specific domain, e.g., WindowManager for window operations, EmailSender for email functionalities, and FileWriter for file writing tasks.

```
public class Miscellaneous {
    public void OpenWindow() { /* Open a window */ }
    public void SendEmail(string to, string subject) { /* Send an email
*/ }
    public void WriteToFile(string content) { /* Write to a file */ }
}
```

Logical Cohesion

Parts of a module are grouped logically by category, but are not necessarily related by time, sequence, or task.

Example: A Utility class that offers a mix of helper methods (like file operations, string manipulations) without them being directly related.

How to improve cohesion: Refactor the Utilities class by separating functionalities into specific classes, like FileOperations for file-related actions, ensuring each class has a single, focused responsibility.

```
public class UtilityServices
{
    // Method for logging messages
    public void LogEvent(string message)
    {
        // Imagine this method logs messages to some logging service
        Console.WriteLine($"Log: {message}");
    }
    // Method for sending emails
    public void SendEmail(string to, string subject, string body)
    {
        // Imagine this method sends an email
        Console.WriteLine($"Sending email to {to} with subject
{subject}");
    }
    // Method for processing user input
    public void ProcessUserInput(string input)
    {
        // Process input in some way
        Console.WriteLine($"Processing input: {input}");
    }
}
```

Temporal Cohesion

Activities are related by the time at which they are done.

Example: A Logger class that performs all logging operations at the same time during application execution.

How to improve cohesion: Instead of having a Logger class handle various unrelated time-based functions, create dedicated classes such as ProcessLogger for start/end logs and ErrorLogger for errors to enhance functional cohesion.

```
public class Logger {
   public void LogStart() => Console.WriteLine("Start");
   public void LogEnd() => Console.WriteLine("End");
   public void LogError(string message) => Console.WriteLine($"Error:
{message}");
}
```

Procedural Cohesion

Components are grouped because they always follow a certain sequence of execution.

Example: A Startup class that initializes application components in a specific order.

This is ok..ish, but can be improved for cohesion: Split the Startup class into classes focused on specific initialization aspects, like ConfigurationInitializer, DatabaseInitializer, and ServiceStarter, each handling its own domain.

```
public class Startup {
    public void Initialize() {
        ConfigureSettings();
        SetupDatabase();
        StartServices();
    }

    private void ConfigureSettings() { /* Configuration logic */ }
    private void SetupDatabase() { /* Database setup logic */ }
    private void StartServices() { /* Start essential services */ }
}
```

Communicational Cohesion

Parts of a module work on the same data (or closely related data).

Example: A ReportGenerator class where different methods generate various parts of a report using the same data source.

```
public class ReportGenerator {
    private List<int> data = new List<int>();

    public void AddData(int newData) => data.Add(newData);
    public int Sum() => data.Sum();
    public double Average() => data.Average();
}
```

Sequential Cohesion

Output from one part of a component serves as input to another part.

Example: A DataProcessor class where one method processes data and passes it to another method for further processing.

```
public class DataProcessor {
    public string ProcessData(string input) {
        var cleaned = CleanData(input);
        return TransformData(cleaned);
    }

    private string CleanData(string data) => data.Trim();
    private string TransformData(string data) => $"Transformed {data}";
}
```

Functional Cohesion

Components are grouped because they all contribute to a single well-defined task.

Example: A Calculator class with methods like Add, Subtract, Multiply.

```
public class Calculator {
    public int Add(int a, int b) => a + b;
    public int Subtract(int a, int b) => a - b;
}
```

Designing for high cohesion

Software architects should design modules to have the highest cohesion possible.

Each module should have a single, well-defined purpose.

The contained elements should be related and contribute to that purpose.

Separation of Concerns

Separation of Concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern.

A concern is a set of information that affects the code of a program. SoC helps in making the software more organized, easier to understand, and maintain.

Example: Consider an application that handles user management and issue tracking. SoC would involve separating the user management logic from the issue tracking logic into different modules or services.

```
// User management concern
public class UserManager {
    public void AddUser(string username) {
        // Add user logic
    }
}

// Issue tracking concern
public class IssueTracker {
    public void CreateIssue(string issueDescription) {
        // Create issue logic
    }
}
```

SOLID

SOLID is an acronym that represents five key design principles in object-oriented programming and design, aimed at making software more understandable, flexible, and maintainable.

These principles help developers manage dependencies in their code, making it easier to extend and modify without introducing bugs or complexities.

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

This principle states that a class should have only one reason to change, meaning it should have only one job or responsibility. By following SRP, developers can create classes that are focused on a single functionality, leading to code that is easier to understand and maintain.

Example: A class that handles both user authentication and user data management violates SRP.

Solution: Split in two classes.

```
// Handles user authentication
public class Authenticator {
    public bool AuthenticateUser(string username,
string password) {
        // Authentication logic
        return true;
    }
}

// Handles user data management
public class UserDataManager {
    public void SaveUserData(string userData) {
        // Save user data logic
    }
}
```

Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) **should be open for extension, but closed for modification**.

This means you should be able to add new functionality to an entity without changing its existing code, which is often achieved through the use of interfaces or abstract classes.

OCP helps in building flexible systems that can grow over time without requiring extensive rework.

Example: Implementing a reporting system that can be extended to support new report types without modifying the existing code.

```
public abstract class Report {
    public abstract void GenerateReport();
}

public class SalesReport : Report {
    public override void GenerateReport() {
        // Generate sales report
    }
}

public class InventoryReport : Report {
    public override void GenerateReport() {
        // Generate inventory report
    }
}
```

Liskov Substitution Principle (LSP)

Objects of a superclass shall be replaceable with objects of its subclasses without affecting the correctness of the program.

It ensures that a subclass can stand in for its superclass without causing errors or unexpected behavior, promoting the use of polymorphism and inheritance more safely and effectively.

Example: Extending a base class with a subclass that doesn't alter the behavior expected by the base class's clients.

```
public class Bird {
    public virtual void Fly() {
        // Default fly behavior
    }
}

public class Duck : Bird {
    public override void Fly() {
        // Duck-specific fly behavior
    }
}
```

Interface Segregation Principle (ISP)

No client should be forced to depend on methods it does not use.

This principle encourages the segmentation of large interfaces into smaller, more specific ones, so that implementing classes only need to be concerned with the methods that are relevant to them.

This reduces the impact of changes and enhances code manageability.

Example: Splitting a large interface into smaller, more specific ones so that clients only need to know about the methods that are of interest to them.

```
public interface IWorkable {
    void Work();
}

public interface IFeedable {
    void Eat();
}

public class Worker : IWorkable {
    public void Work() {
        // Work implementation
    }
}

public class Animal : IFeedable {
    public void Eat() {
        // Eat implementation
    }
}
```

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules, but both should depend on abstractions. Furthermore, abstractions should not depend on details; details should depend on abstractions.

DIP leads to the decoupling of software modules, making the system easier to refactor, scale, and test

Example: A service that communicates with a data repository should not be directly dependent on a specific implementation but rather an abstract interface.

```
public interface IDataRepository {
    void Save(object data);
}

public class DataRepository : IDataRepository {
    public void Save(object data) {
        // Save data to the database
    }
}

public class DataService {
    private readonly IDataRepository _dataRepository;

    public DataService(IDataRepository dataRepository) {
        _dataRepository = dataRepository;
    }

    public void SaveData(object data) {
        _dataRepository.Save(data);
    }
}
```

Distributed Systems

Distributed Systems

Loose coupling brings us to Distributed Systems (more on this later on, during the course).

Interesting topics:

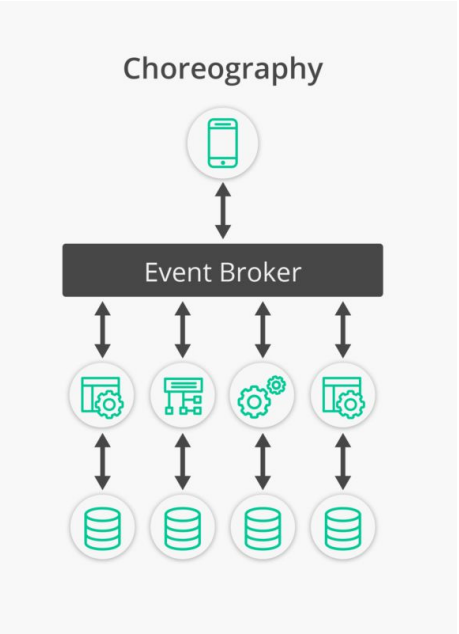
- Choreography vs orchestration
- Security
- Observability

Choreography

Choreography deals with how individual services within a distributed system interact with one another without a central point of control.

Designing services to be as loosely coupled as possible and as high cohesive as possible, enhances the scalability and resilience of distributed systems.

In a nutshell: It's a decentralized, event-driven approach where services communicate through events.



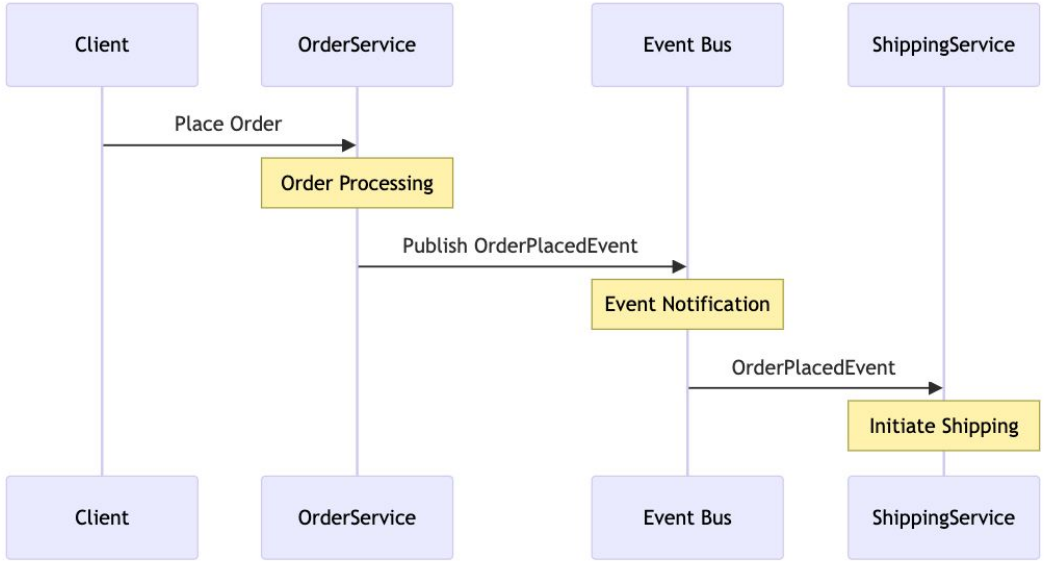
Choreography

Key Concept: Services independently decide on their actions based on events.

Example Scenario: *OrderService* publishes an *OrderPlacedEvent*, and *ShippingService* reacts by initiating shipping without direct orders.

Benefits: Enhances service decoupling, promotes flexibility, and reduces single points of failure.

Challenges: Requires robust event management and can lead to complexity in tracking service interactions.

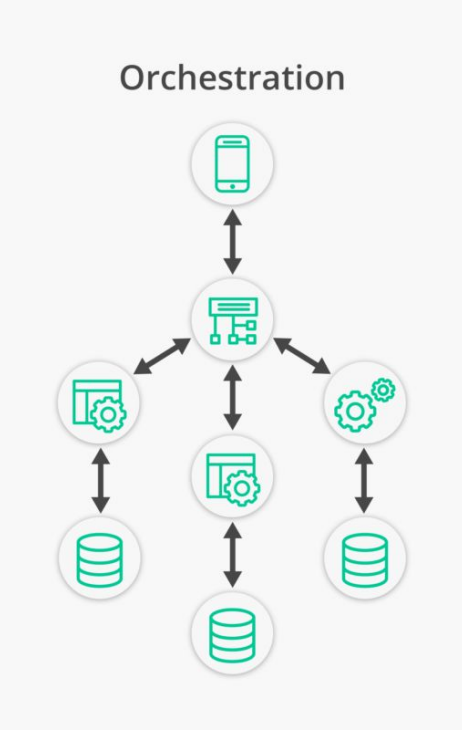


Orchestration

A central orchestrator (e.g., workflow engine) controls the interaction between services, dictating the process flow.

Orchestration focuses on a central orchestrator or coordinator that dictates the workflow and interaction between services in a distributed system. This approach allows for precise control over the processes and interactions.

In a nutshell: Orchestration represents a centralized, controlled approach to managing service interactions. It relies on a designated orchestrator to direct the flow of operations, making it easier to monitor, manage, and modify complex processes within distributed systems.



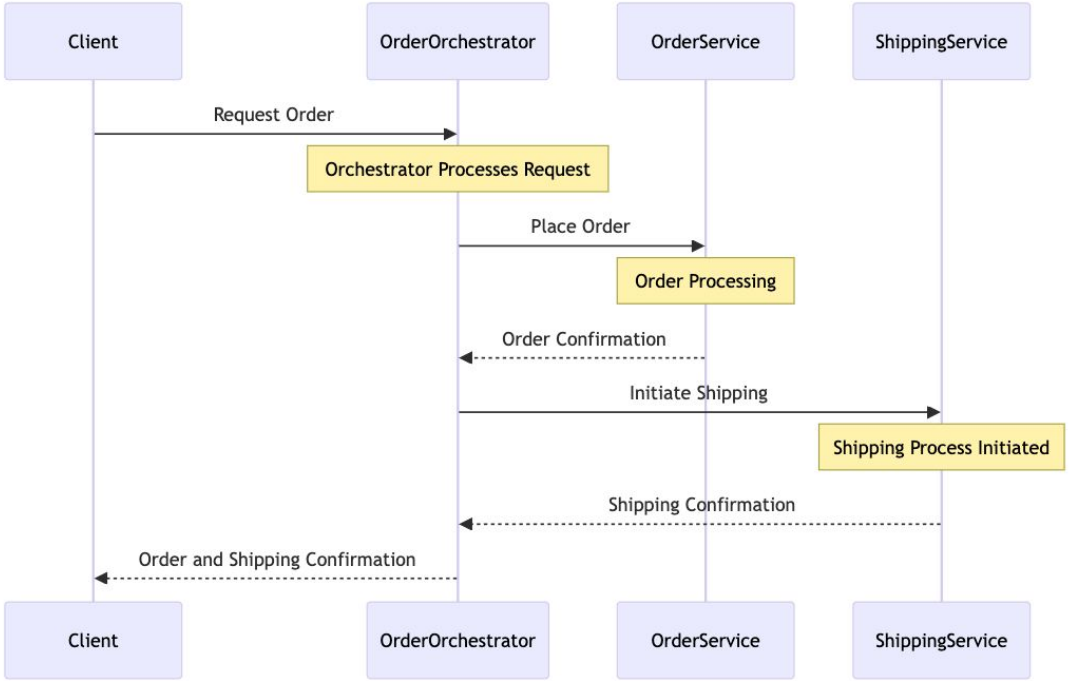
Orchestration

Key Concept: A central orchestrator (e.g., workflow engine) controls the interaction between services, dictating the process flow.

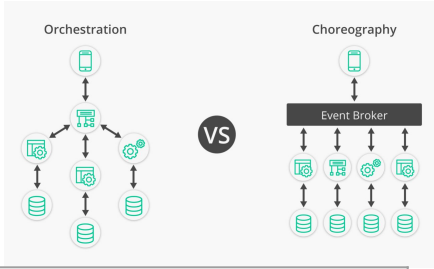
Example Scenario: An *OrderOrchestrator* service directly calls *OrderService* to place an order, then commands *ShippingService* to ship the order.

Benefits: Simplifies interaction logic, centralizes control, and makes processes easier to monitor and adjust.

Challenges: Can create bottlenecks and increase the risk of a single point of failure.



Choreography vs Orchestration



	Choreography	Orchestration
Control	Decentralised control	Centralised Control
Complexity	Higher complexity in event management	Simpler interaction logic
Coupling	Low coupling	Potential for higher coupling
Scalability	Highly scalable	Scalable with potential bottlenecks
Failure Impact	Distributed failure impact	Centralised failure impact

Consider also: System size, team capability, real-time scalability needs, and failure tolerance

Choreography	Orchestration
Pros	
Decentralization: Enhances resilience and avoids single points of failure.	**Centralized Control:** Simplifies process management and makes workflows easy to understand and monitor.
Flexibility: Easier to add or modify services without impacting the overall flow.	**Coordination**: Allows for complex processes involving multiple services to be coordinated effectively.
Scalability: Services can scale independently, improving system responsiveness and efficiency.	**Visibility**: Provides a clear overview of process flows and their current state, aiding in debugging and monitoring.
Cons	
Complexity: Managing and monitoring distributed events can become challenging as the system grows.	**Bottleneck Risk:** The orchestrator can become a bottleneck, affecting system performance and resilience.
Debugging Difficulty: Tracing problems through asynchronous events across services can be complex.	**Coupling**: Services may become more tightly coupled to the orchestrator, potentially hindering flexibility.
Event Consistency: Ensuring event consistency and handling failure scenarios require careful design.	**Single Point of Failure:** The orchestrator's central role can make it a single point of failure, impacting the entire system if it goes down.

Security in (Enterprise) Software Systems

Enterprise security:

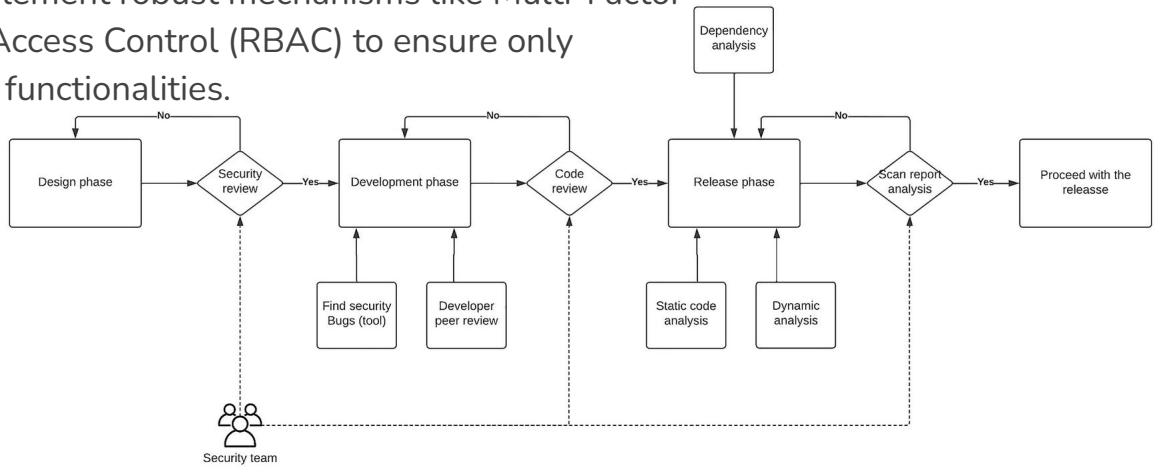
- includes the techniques, processes, and strategies that secure data and IT assets from unauthorized access
- prevent risks that may interfere with confidentiality, integrity, and availability (CIA) enterprise security of the system.
- is a combination of technology, people, and processes that secure the most valuable asset of an enterprise, which is data.
- Comprises of two main components (based on the required techniques to secure those):
    - Data
    - Applications

Data Security

- At rest: encryption of databases and storage systems to secure data at rest
- In transit: SSL and TLS
- Data privacy: need to know basis - cypher/mask database fields for specific operators. Comply with data protection regulations

Application Security Strategies

- **Secure Development Lifecycle (SDLC):** Integrate security practices at every phase of software development to identify and mitigate vulnerabilities early.
- **Dependency Management:** Regularly audit and update third-party libraries and dependencies to protect against known vulnerabilities.
- **Authentication and Authorization:** Implement robust mechanisms like Multi-Factor Authentication (MFA) and Role-Based Access Control (RBAC) to ensure only authorized users can access application functionalities.



Observability

Distributed computer systems increase the number of communication links required to perform a certain computing task.

These communication links can fail due to many reasons such as hardware failures, firmware bugs, or software bugs.

The more distributed the system is, the more chance that a particular component can fail, and hence, the overall system stability is impacted.

But this is not something we can avoid since the benefits of distributed systems outrank the challenges they pose. Instead, we can design these systems to withstand failure and continue.

There are three main aspects that we need to deal with when implementing and maintaining distributed systems in a robust manner:

- Make the system observable.
- Monitor the system continuously.
- Take remedial measures to fix the failures.

Observability

Definition: Observability is the ability to understand the internal states of a system based on its external outputs. It's crucial for detecting, diagnosing, and resolving issues within complex software architectures.

The Three Pillars: Logs, Metrics, and Traces - key data types that provide insights into system performance and health.

Importance: Observability enables proactive issue resolution, performance optimization, and improved system reliability.

Benefits:

- Ability to provide better user experience by **mitigating failures quickly** and avoiding possible failures of the system
- Ability to allow teams to **improve the efficiency of the system** and fuel innovation and growth using metrics
- Ability to **keep operational data in one place** so that audit and compliance requirements can be met without much hassle

Observability - Logs

- Structured or unstructured records generated by applications, services, and infrastructure, detailing events and operations.
- Suitable for identifying errors, understanding application flow, and security monitoring.
- **Best Practices:** Implement consistent log formats, use log aggregation tools, and ensure logs are searchable and actionable.

Observability - Metrics

- Quantitative data that measures various aspects of system performance over time, such as CPU usage, memory consumption, and request latency.
- Quantitative data that measures business and operational performance, such as active users, abandoned carts, number of purchases in the last hour, etc.
- Suitable for monitoring system health, alerting on anomalies, and performance benchmarking.
- **Best Practices:** Use time-series databases for metrics storage, establish meaningful alerts, and visualize metrics for easy interpretation.

Observability - Traces

- Detailed, contextualized records of individual requests as they travel through distributed systems, capturing the path and latency of inter-service calls.
- Suitable for pinpointing bottlenecks, understanding dependencies, and optimizing performance.
- **Best Practices:** Adopt distributed tracing standards (e.g., OpenTelemetry), ensure consistent trace ID propagation, and integrate tracing with logs and metrics for comprehensive insights.

Implementing Observability

- use existing tooling! Prometheus, Grafana, ELK Stack, Jaeger, ...
- Strategy: Prioritize instrumenting critical paths and services first. Align with business objectives and system complexity
- Integrate & Automate: dynamic thresholding, anomaly detection, and integrating observability tools with CI/CD pipelines for real-time insights

Observability Best practices

- Enrich logs, metrics, and traces with contextual information to facilitate quicker diagnosis and resolution.
- Implement techniques and tools that allow correlation between logs, metrics, and traces for a unified view of system behavior.
- Be mindful of **data storage costs and retention policies**. Implement data lifecycle management to balance accessibility and cost.

Bibliography

- (Garlan, 1992) – David Garlan, Mary Shaw. “An Introduction to Software Architecture”
- (Mens, T. , Demeyer, S. , 2008) – Tom Mens and Serge Demeyer, “Software Evolution”, ISBN: 978-3-540-76440-3, http://www.springerlink.com/content/978-3-540-76439-7/#section=200232&page=1
- (Bass et al, 1998) – L. Bass, P. Clements, R. Kazman (1998). Software Architecture in Practice. Reading, MA: Addison Wesley Longman, Inc.
- (Garlan et al, 1993) – D. Garlan, M. Shaw (1993). "An Introduction to Software Architecture." In V. Ambriola, G. Tortora, eds., Advances in Software Engineering and Knowledge Engineering, Vol. 2, pp. 1—39. New Jersey: World Scientific Publishing Company.
- Solution Architecture Patterns for Enterprise: A Guide to Building Enterprise Software Systems
- https://camunda.com/blog/2023/02/orchestration-vs-choreography/
- https://learn.microsoft.com/en-us/azure/architecture/patterns/choreography

Data privacy exercise web app

https://docs.google.com/document/d/1A9Lr4qx8_phgnCYX-rWDPQkUhGacDsy4_P9mIpOh6Ew/edit?usp=sharing

Full Class individual Presentation: 11th and 12th March