Service Oriented Architecture

Version 2.0

Updated
04 Mar 2025

Lecture Notes, Discussion Points and References

Table of Contents

SOA Syllabus

Revision 2.0, Mar 2025 (Updates in **Bold**)

Unit 1: Introduction to Service-Oriented Architecture	
1.1	Overview of Service-Oriented Architecture - Idea of a Service, Key Characteristics, Historical Context, **AI-Driven Services**
1.2	Principles and Concepts of SOA - Service Loose Coupling, Service Reusability, Service Abstraction, **AI Considerations**
1.3	Evolution and History of SOA - Early Web Services, Emergence of SOA Standards, **Transition to Microservices & Containerization, DevOps and MLOps, Role of Cloud Providers & AI Services:**
1.4	Benefits and Challenges of SOA - Business Agility, Interoperability, Challenges in Implementation, **AI Integration Challenges**
1.5	Contemporary Trends in SOA - Microservices Architecture, Cloud Computing and SOA - Serverless Computing and SOA, **AI/ML in the Service Ecosystem**
Unit 2: SOA Design and Modeling	
2.1	Service Design Principles and Patterns - Service Cohesion, Granularity, Design for Change, **Service Design in AI**
2.2	Service Contract Design and Management - Interface Definition Languages (IDLs), Contract-First Design, Versioning and Evolution, **AI and gRPC**
2.3	Designing for Scalability and Resilience - Load Balancing, Fault Tolerance, Circuit Breaker Pattern, **AI Workloads**
Unit 3: SOA Implementation Technologies	
3.1	Web Services Standards - Simple Object Access Protocol (SOAP), Representational State Transfer (REST) - GraphQL, **gRPC**
3.2	Microservices Architecture and its Relationship with SOA - Decentralized Data Management, Independent Deployment, **MLOps & Microservices**: - Infrastructure Automation, **Automated Model Deployment**
3.3	Containerization and Orchestration

	- Docker Container, Kubernetes Orchestration, **Specialized AI/ML Orchestration,** Service Mesh Technologies, **Observing AI Microservices**
3.4	Event-Driven Architecture (EDA) and SOA - Event Sourcing, Command Query Responsibility Segregation (CQRS), **AI Use Case** - Event-Driven Messaging Systems, **Pub/Sub patterns, Streaming Pipelines for AI**
3.5	API Management and Governance - API Design Principles, **AI-Specific API Considerations** - Developer Portals, **AI "Model Catalog"** - Rate Limiting and Quotas, **AI Endpoint Limits**
Unit 4: Security and Governance in SOA	
4.1	Security Considerations in SOA - Understanding Threat Models, Common Security Risks in SOA **Threats in AI-Driven Services** - Security Design Patterns, **Zero Trust for distributed microservices (including AI endpoints)**
4.2	Data Encryption and Integrity - Message-Level Encryption (XML Encryption), Digital Signatures (XML Signature), **- JSON & gRPC, Data at Rest for AI Models** - Secure Hash Algorithms (SHA), Securing APIs and Web Services,
4.3	API Security Best Practices - Securing RESTful APIs, Web Service Security Standards (WS-Security) **- Securing AI Inference APIs**
4.4	XML Security and SAML Assertions - XML Security Considerations, Introduction to SAML **- JSON-based Security** - SAML Assertions and Assertions Consumers, **Modern Alternatives with JWT/OAuth 2.0 vs. SAML usage in microservices and AI service endpoints**
Unit 5: SOA Emerging Trends	
5.1	Serverless Computing and its Impact on SOA - Function-as-a-Service (FaaS), Event-Driven Architectures, **AI Use Cases in Serverless** - Operational Characteristics, **Observability in AI-Driven Serverless**
5.2	Artificial Intelligence (AI) and Machine Learning (ML) in SOA - Intelligent Agents, Predictive Analytics, Natural Language Processing (NLP), **AI Orchestration and Workflow**
5.3	Edge Computing and SOA Integration, - **,** Edge Gateway Architectures, Low-Latency Data Processing. Offline Capabilities, **AI Workloads at the Edge**

SOA Lab Exercises

Exercise 1: Overview of Service-Oriented Architecture

Objective:

Introduce students to SOA basics by creating a simple, containerized web service that clients can consume.

Steps / Tasks

1. **Set Up a Basic Service**
   - Choose a framework (Flask/Python, Spring Boot/Java, Express/Node.js).
   - Create a simple endpoint (e.g., `/products` or `/users`) that returns or manipulates data (e.g., CRUD operations in-memory or in a small database like SQLite).
2. **Containerization**
   - Create a `Dockerfile` for your service.
   - Use Docker to containerize and run your service locally (e.g., `docker build`, `docker run`).
3. **Client Consumption**
   - Write a simple client script or another microservice to call your web service.
   - Demonstrate basic operations (GET, POST, PUT, DELETE).
4. **Documentation and Testing**
   - Produce a small `OpenAPI/Swagger` specification to define your API.
   - Use a tool like **Postman** or **curl** to test endpoints.

Key Learning Points:

- Foundational SOA concepts (service exposure, discoverability, loose coupling).
- Introduction to containerization for easy deployment and scaling.

Exercise 2: Principles and Concepts of SOA

Objective:
Implement a loosely coupled service architecture using **asynchronous messaging**.

Steps / Tasks

1. **Choose a Message Broker**
    - Use **RabbitMQ**, **Apache Kafka**, or **ActiveMQ**.
    - Explain how messaging decouples the producer from the consumer.
2. **Publisher Service**
    - Create a simple service that sends messages (e.g., JSON payload) to the broker whenever an event occurs (e.g., new order placed, data update).
    - Containerize it if desired (using Docker) for consistency.
3. **Consumer Service**
    - Implement a separate service that subscribes to the broker and processes messages independently (e.g., logs them, stores them in a DB, triggers a workflow).
    - Ensure no direct coupling between publisher and consumer beyond the message format.
4. **Observability**
    - Introduce logging and monitoring for your publisher and consumer services (e.g., using **Elastic Stack**, **Prometheus**, or built-in broker metrics).

Key Learning Points:

- Asynchronous communication for **loose coupling**.
- Event-driven design principles and decoupled service interactions.
- Understanding of **microservices** patterns.

Exercise 3: Contemporary Trends in SOA

Objective:
Explore Serverless Computing and integrate it into an existing SOA/microservice ecosystem.

Steps / Tasks

1. Set Up a Simple Serverless Function
    - Pick an open source serverless platform like (**OpenFaaS, Apache OpenWhisk, Knative, Kubeless, Fission, or KEDA**).
    - Create a function that performs a specific task, e.g., image resizing, simple text processing, or a quick calculation.
2. Expose the Function as a REST Endpoint
    - Use API Gateway (AWS), Azure Function's HTTP trigger, or Cloud Functions' HTTPS endpoint to make your function externally callable.
    - Verify your function can accept inputs and return outputs.
3. Integration with Other Services
    - Invoke your serverless function from a previously created microservice or a simple client.
    - Demonstrate that the function can be part of a broader SOA. For example, upload an image via a REST service, which triggers the serverless function to resize it and then store the result in a storage service.
4. Observability and Cost Monitoring (Optional Enhancement)
    - Show how to monitor invocation counts, latency and cost metrics for your serverless function.
    - Highlight the ephemeral nature of serverless (cold starts, concurrency limits, etc.).

Key Learning Points:

- Basics of Function-as-a-Service (FaaS).
- Serverless integration with existing services for scalability and event-driven operations.

- Challenges like cold starts, limited runtime environment, debugging in a serverless context.

Exercise 4: Artificial Intelligence (AI) and Machine Learning (ML) in SOA:

Objective:
Build and integrate a **simple AI-driven service** (e.g., classification, sentiment analysis, or basic prediction) within an SOA-based architecture.

Steps / Tasks

1. **Develop/Obtain a ML Model**
   - Use a small classification model (e.g., scikit-learn or TensorFlow).
   - Pre-train or load a pretrained model (e.g., for text sentiment or an image classification dataset like MNIST).
2. **Build a Service for ML Inference**
   - Wrap the model in a REST endpoint (Flask, FastAPI, or any framework).
   - Accept input data (text, image, numeric features) and return inference results.
3. **Containerize**
   - Package the model service with **Docker** for easy deployment.
   - Show how the model can be scaled independently, if needed.
4. **Integration and Testing**
   - Integrate your AI service with a front-end client or another microservice. For instance, the client sends text or image data and the AI service returns a prediction.
   - Demonstrate how updates to the model (newer version, better accuracy) can be swapped in with minimal disruption to the rest of the SOA.
5. **Expand with MLOps**
   - Briefly mention or demonstrate how to track model versions, use a model registry (e.g., MLflow).
   - Automated testing: ensure new model versions do not break the interface or degrade performance.

Key Learning Points:

- Basic AI model serving in a **service-oriented** environment.
- Handling model versioning, data input/output formats and performance considerations.
- Implementation of MLOps

Reference Books

1. Service-Oriented Architecture: Concepts, Technology and Design by Thomas Erl

2. Building Microservices" by Sam Newman

3. Microservices Patterns: With examples in Java by Chris Richardson

4. SOA Security by Ramarao Kanneganti and Prasad Chodavarapu

5. Designing Data-Intensive Applications – Martin Kleppmann

6. Practical MLOps – Noah Gift, Alfredo Deza

7. Kubernetes Patterns – Bilgin Ibryam, Roland Huß

Terminology

Enterprise	Enterprise refers to an organization or a business
Service	A basic granular unit of a system that provide a specific function
Architecture	An organization or design pattern of an software system
IT Systems	Refers to hardware and software components of an Enterprise
Applications	A software designed to specific functions or services
Web Service	An HTTP based application used over internet
Security	Refer to protecting user, data, infrastructure and applications of an enterprise

Unit-1 Introduction to Service Oriented Architecture

1.1	Overview of Service-Oriented Architecture - Why learn SOA, Key Characteristics, Historical Context, **AI-Driven Services**

Motivations of SOA

Consider **modern applications** that we use in our daily lives:

- Streaming - JioHotstar, Netflix, Prime Video
- Payment (NCPI: BHIM, Paytm, Google Pay, etc)
- Cab Booking applications (Ola, Uber, Rapido, Namma Yatri, etc)
- e-Commerce (Amazon, Flipkart)
- Quick Commerce (Zepto, Blinkit)
- Navigation (Google Maps, Open Street Maps)
- Food Delivery (Swiggy, Zomato, etc)
- Social Media (Instagram, Twitter)
- Communication (WhatsApp, Telegram, Jabber)

IND vs AUS, Champions Trophy 2025: Jio Hotstar live streaming viewership surges over 66.9 cr as India wins semi-finals

IND vs AUS, Champions Trophy 2025: India won the semi-final against Australia scoring 267 with 4 wickets and 11 balls remaining on Tuesday. Viewership on Jio Hotstar surged to over 66.9 crore as of the end of the cricket match.

Livemint

Updated • 4 Mar 2025, 09:40 PM IST

Daily Product Statistics | Monthly Product Statistics

Filter:

Product	Year	Month	Metric
UPI	2025	February	Volume

UPI Daily Product Statistics Trended February 2025

Day	Volume (in Mn.)
February 1, 2025	604.75
February 2, 2025	591.60
February 3, 2025	582.39
February 4, 2025	576.42
February 5, 2025	585.68
February 6, 2025	579.32

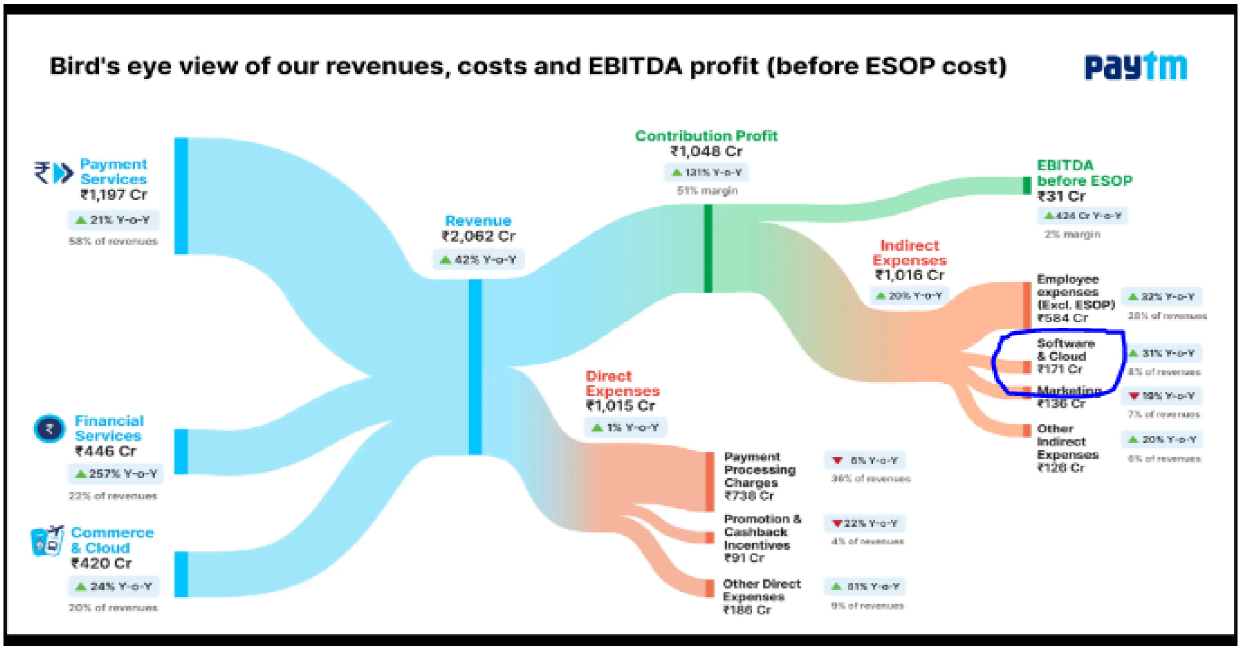
NPCI Transactions: https://www.npci.org.in/statistics/monthly-metrics

Day	Volume (in Mn.)
February 1, 2025	604.75
February 2, 2025	591.60
February 3, 2025	582.39
February 4, 2025	576.42
February 5, 2025	585.68
February 6, 2025	579.32

Challenges of building such applications

- **Scalability** - very high number of users (thousands, millions of active users, transactions)
- **Multiple and Heterogeneous components** - each application has many components interconnected (front-end, backend, databases) and each component is designed differently and coded in different languages (e.g. Java, Go-lang, JS, Python, etc)
- **Security** - authentication (e.g. login), protecting data (encryption), data privacy
- **High Availability and Business Continuity** - making applications available with minimal down time.
- **Cost and Operational Expenses (OpEx)** -
  - Development Cost - developing new products and features

- ○ Deployment Cost - high cost of running these applications in data centers and cloud.
- ○ Paytm OpEx: Software, Cloud and Data Center costs were ₹171 Cr, up 31% YoY in Feb 2023, ₹188 Cr in Mar 2023



Source:https://paytm.com/blog/investor-relations/how-paytm-achieved-operational-profitability/

What is SOA

Home Work References
1. Read about original publication on SOA: https://www.opengroup.org/soa/source-book/soa/index.htm   2. Read Martin Fowler (ThoughtWorks) take on SOA for an article in 2005 https://martinfowler.com/bliki/ServiceOrientedAmbiguity.html

Modern take on SOA (Inspired by Martin Fowler article)

- For some, SOA is all about **exposing software**—like AI models—**through web services**.
- Others see SOA as ending “big apps,” **focusing on small core services** plus aggregator UIs for business and AI tasks.

- Some treat SOA as a **universal messaging backbone**—HTTP, Kafka, whatever—for all systems, including AI pipelines.
- Another crowd **uses SOA for asynchronous data flows**, letting AI workflows happen behind the scenes without blocking.
- Overall, **there's no single "right" SOA**—it's just about building flexible, maintainable systems, often mixing microservices, serverless, and AI.

Formal Definition

- **SOA refers to** → Service Oriented Architecture
    - Service-Oriented Architecture (SOA) is an architectural style that supports **service-orientation.**
    - **It's a Design Pattern:** a way to build modern complex applications using granular, reusable services.
    - **It's an approach** to build software systems that are based on <u>distributed systems</u>.
    - It's an approach to build software systems based on <u>loosely coupled service</u> components
- A *service*:
    - Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)
    - Is self-contained
    - *May be* composed of other services
    - Is a "black box" to consumers of the service
- **Idea of a Service -** A service is defined as a specific **granular, functional, self-contained, reusable** component or code consumed by other services or applications (e.g. Login Service, Order History, Map APIs)
    - **Service interface -** provides interface to invoke a service and define formats to pass and receive data from a service. For example, user of RESTAPI for request and response, XML, JSON for sending and receiving data from service.

- **Service is technology independent and interoperable** - consumers of the service can invoke the service on any hardware or software platform or code. For example, a Cab booking app running in AWS can invoke Google Map API services to from source to destination
- **Service is discoverable** - consumers of the service can easily detect the purpose and use of the service. For example, a E-Commerce app can discover various payment methods.
- **Service is stateless** - a service doesn't maintain any specific state of a service call. For example, a QR scanning service takes a QR code, just returns the value of code and doesn't maintain any other context of service invocation.

Key Characteristics of SOA

Service-Oriented Architecture (SOA) is defined by several key characteristics that shape its design and implementation. These characteristics include:

1. **Loosely Coupled**: SOA promotes loose coupling between software components, allowing them to interact independently without tight dependencies. This enables flexibility and agility in system design, as services can be modified or replaced without impacting other components.

2. **Interoperable**: SOA facilitates interoperability between heterogeneous systems and technologies. By adhering to open standards and protocols, such as XML, SOAP and REST, services can communicate seamlessly across different platforms and programming languages.

3. **Flexible:** SOA is inherently flexible, allowing for the composition and recomposition of services to meet changing business requirements. Services can be combined and orchestrated in various ways to create new functionalities, enabling organizations to adapt to evolving needs.

4. Scalable: SOA provides scalability by distributing functionality across multiple services, each capable of running independently and horizontally scaling to accommodate increased demand. This allows systems to handle varying workloads and scale resources efficiently.

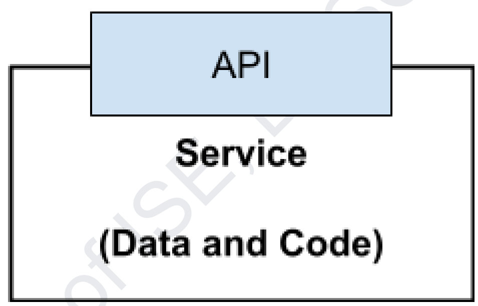
5. Stateless: SOA promotes statelessness, where services do not maintain session state between requests. This enhances scalability and fault tolerance by allowing services to handle each request independently, without relying on previous interactions.

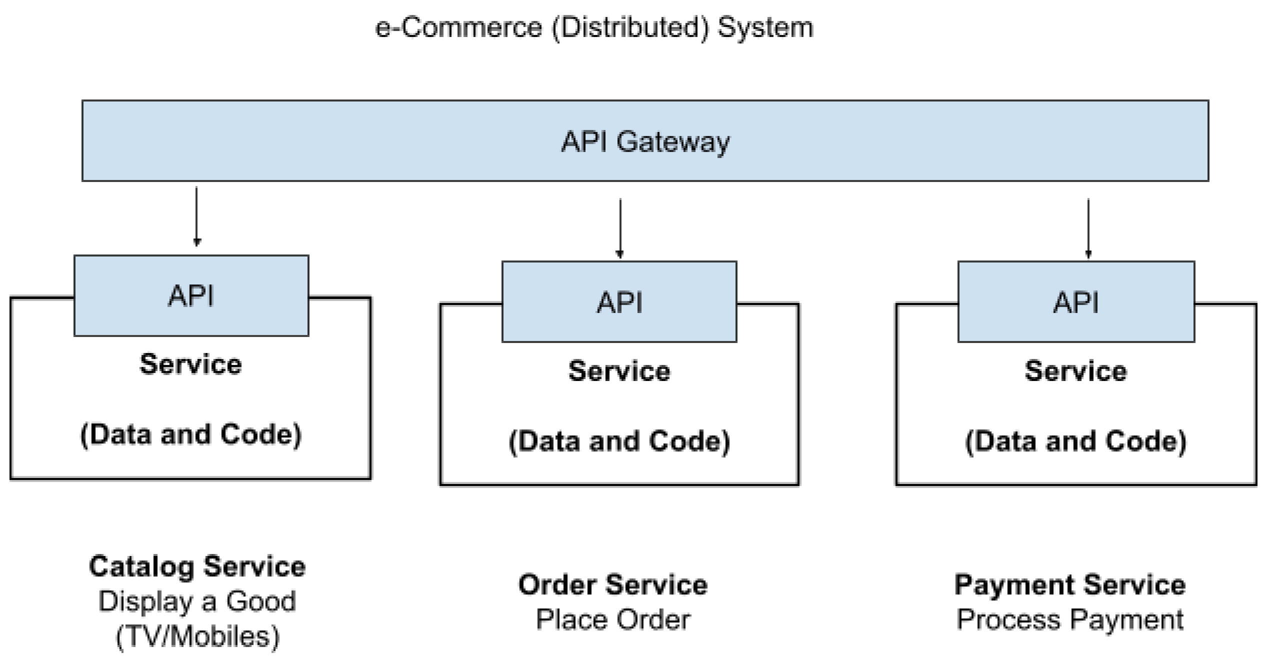
These key characteristics of SOA—loose coupling, interoperability, flexibility, scalability and statelessness—lay the foundation for building resilient, adaptable and efficient **complex and distributed** software systems.

SOA Components

The basic unit of Service Oriented Architecture is a **Service**

- It's a self-contained software system
- Accessible via APIs





Evolution and Historical Context of SOA

Late 1990s - Early 2000s: Emergence of Web Services

- *Key Events*:
  - Rise of Internet technologies - Dot Com Boom! Web based growth
  - The term first appeared in 1998
  - Need for interoperability between systems
- *Technologies*:
  - Introduction of SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language)

Early to Mid-2000s: Paradigm Shift with SOA

- *Key Events*:
  - Transition towards modular, loosely coupled architectures
  - Emergence of SOA as a new architectural paradigm
- *Technologies*:
  - Adoption of SOA principles in enterprise integration

Standardization Efforts and Industry Consortia

- *Key Events*:
  - Development of SOA-related standards and specifications
  - Role of organizations like W3C and OASIS
- *Technologies*:
  - Definition of XML, SOAP, WSDL and WS-* standards

Challenges and Criticisms in SOA Adoption

- *Key Challenges*:
  - Complexity in service design and governance
  - Cultural resistance to change
  - Concerns about ROI
- *Critiques*:
  - Effectiveness of SOA in delivering promised benefits

Legacy and Impact

- *Key Influences*:
  - Evolution towards microservices and cloud-native architectures
  - **Serverless** and **API-first** approaches also align with SOA's idea of exposing discrete functionality.
  - Continued relevance in modern software design practices
  - The concept of **service boundaries** remains crucial, especially in modern **AI/ML** model-serving scenarios.
- **Continued Relevance:**
  - Even as technology stacks shift (REST, GraphQL, gRPC), SOA's core vision of **interoperable services** still underpins **enterprise integration** and **distributed architectures**.

Early Web Services

Early web services emerged in the late 1990s and early 2000s as a means of enabling interoperability and communication between disparate systems over the Internet.

- **Key Features of Early Web Services**
  - Utilized technologies such as HTTP, XML and SOAP (Simple Object Access Protocol) for communication.
  - SOAP provided a standardized protocol for exchanging structured information between systems.
  - Web services allow for the integration of applications across different platforms and programming languages.

Here are use cases of SOA in early web services:

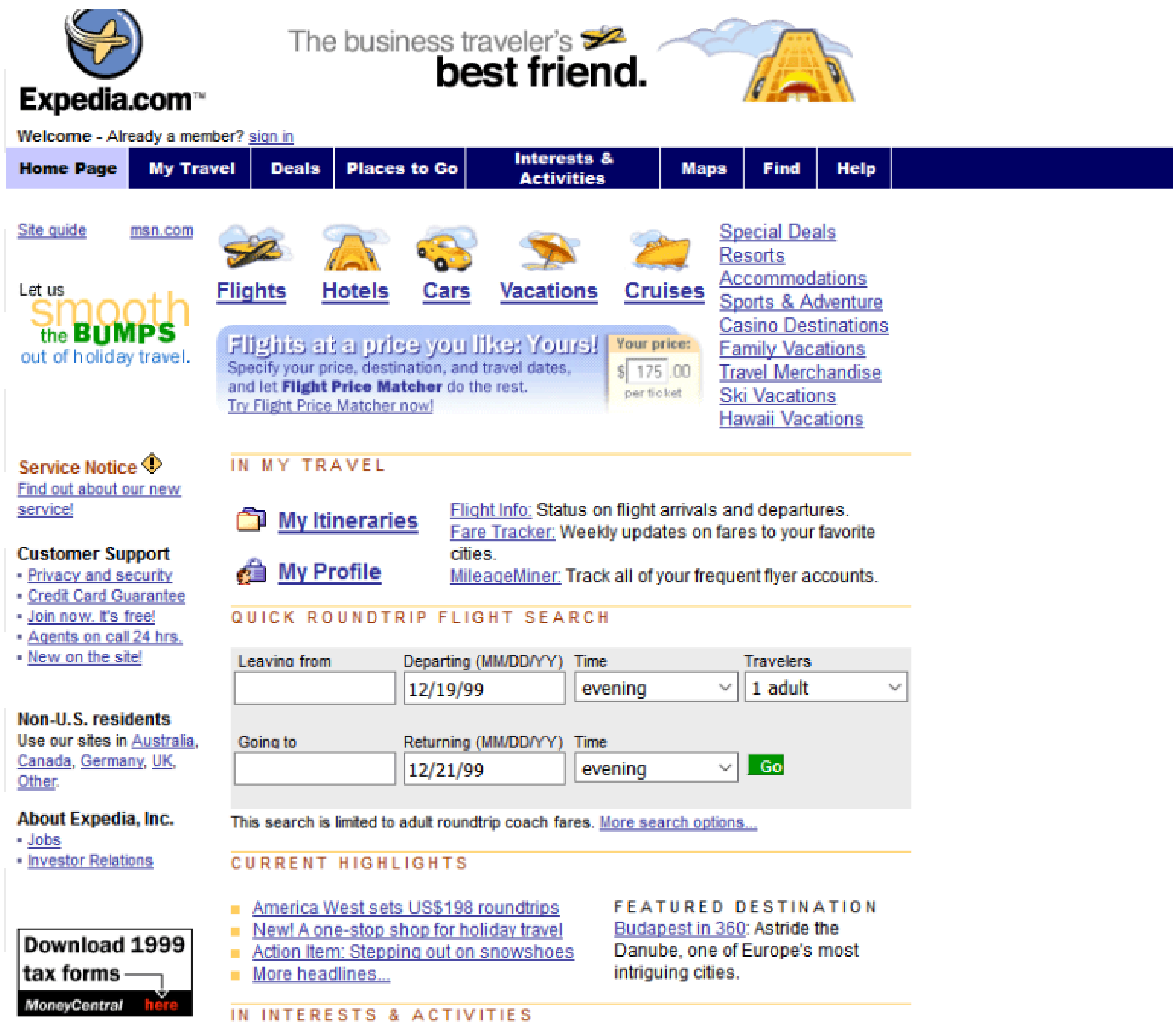
- ***E-commerce Integration***: Early web services facilitated the integration of e-commerce platforms with payment gateways, enabling secure transactions and real-time order processing.
- ***Travel Booking Systems***: Travel agencies use web services to integrate with airline reservation systems, allowing customers to search for flights, book tickets and receive real-time updates on flight availability.
- ***Enterprise Resource Planning (ERP)***: Organizations utilise web services to integrate ERP systems with third-party vendors for tasks such as inventory management, supply chain optimization and financial reporting.

E-commerce Integration - Amazon.com

- **Use Case *Scenario:*** In the mid-90s, Amazon.com began its journey as an online bookstore, but it quickly evolved into a platform selling a wide range of products. To handle the complex nature of its e-commerce operations, Amazon likely employed service-oriented principles, even before SOA was formally recognized.

- **SOA Characteristics**: Amazon's platform likely utilised loosely coupled services to handle various aspects of its operations, such as inventory management, order processing, payment processing and customer relationship management (CRM). These services would have communicated with each other using standardized protocols such as HTTP and XML

Travel Booking System - Expedia.com



- **Use Case Scenario**: Expedia, founded in the late 1990s, revolutionized the travel industry by offering an online platform for booking flights, hotels, rental cars and vacation packages. To provide a seamless booking experience to its users, Expedia likely employed SOA principles in its system architecture.

- **SOA Characteristics**: Expedia's platform likely consisted of various services responsible for different functions, such as flight search, hotel booking, payment processing and itinerary management. These services would have been loosely coupled, allowing for flexibility and scalability. For example, the flight search service could communicate with airline reservation systems via standardized interfaces, such as SOAP or XML over HTTP.

Enterprise Resource Planning - SAP ERP

- **Use Case Scenario:** SAP, a leading provider of ERP software, has been in the industry since the early 1970s. While its early systems may not have fully embraced SOA principles, SAP likely transitioned towards a more service-oriented approach in the late 1990s and early 2000s to address the growing complexity of enterprise operations.
- **SOA Characteristics:** SAP's ERP system would consist of various modules or services responsible for different business functions, such as finance, human resources, supply chain management and customer relationship management. These services would have been designed to be reusable and interoperable, allowing organizations to customize their ERP implementations based on their specific needs.

Emergence of SOA Standards

The emergence of SOA standards occurred as organizations sought more structured and scalable approaches to integrating systems and building software applications.

Key Standards	Original Role	Modern View
SOAP (Simple Object Access Protocol): A protocol for exchanging structured information in	Defined message formats, error handling, and communication protocols for **web services**.	Still used in many enterprise environments, especially where **WS-***

Key Standards	Original Role	Modern View
the implementation of web services.		standards (e.g., WS-Security) are critical.
WSDL (Web Services Description Language): A standard for describing the functionality of web services, facilitating their discovery and invocation.	Provided a **contract-first** approach to describe web service interfaces and operations, facilitating **discovery** and **invocation** in SOAP-based contexts.	Inspired later mechanisms in RESTful environments, such as **OpenAPI/Swagger** for describing REST APIs.
XML (eXtensible Markup Language): A markup language used for encoding documents in a format that is both human-readable and machine-readable	Became the go-to format for **structuring data** in SOAP messages and configuration.	**Shift to JSON:** Over time, many services adopted **JSON** due to its lighter weight and better fit for **web-based** communication.
REST (Representational State Transfer) & JSON	Gained popularity as a simpler alternative to SOAP, using HTTP verbs (GET, POST, etc.) and **JSON** payloads	Encouraged **stateless** interactions and easier integration, forming the backbone of many microservices and modern APIs.
gRPC & Protocol Buffers	A high-performance, language-agnostic RPC	Well-suited for low-latency or **AI/ML** scenarios, often

Key Standards	Original Role	Modern View
	framework that uses **Protocol Buffers** for data serialization.	used for microservices at scale.

AI-Driven Services: A Modern Extension of SOA

Modern **Service-Oriented Architecture (SOA)** isn't limited to traditional web services that simply exchange XML or JSON over HTTP. With the rise of **artificial intelligence (AI) and machine learning (ML)**, organizations increasingly expose **trained models** as **independently deployable services**—often referred to as **"Model as a Service"** (MaaS). This trend significantly extends the original SOA concepts by emphasizing:

1. **Data-Centric Interfaces**
   - Traditional SOA focuses on functional operations (e.g., "create order," "update account").
   - **AI-driven** endpoints often revolve around **model inference**: accepting data (e.g., images, text) and returning predictions (e.g., classifications, recommendations).
   - Data formats can be more complex—images, audio, time series—requiring careful consideration of **serialization** (JSON, Protocol Buffers, etc.).
2. **Loose Coupling for Continuous Model Updates**
   - A key SOA principle is **loose coupling**—ensuring that changes in one service do not break others.
   - When models are treated as **services**, updates (like retraining or swapping a model) can happen **independently** without disrupting downstream clients.
   - This aligns with **DevOps/MLOps** practices, where continuous integration and deployment (CI/CD) extend to **model retraining** and **re-deployment**.
3. **Scalability and High-Performance Requirements**

- AI inference can be **resource-intensive**, especially for large models (e.g., deep learning) that may run on GPUs.
- **SOA** must account for dynamic **scaling** and specialized hardware needs to handle fluctuating inference workloads.
- Auto Scaling microservices (on Kubernetes, for instance) or employing serverless frameworks helps manage spikes in AI requests.

4. **Model Discovery and Lifecycle Management**
    - Traditional SOA might use a **service registry** to discover endpoints (e.g., UDDI, or more modern API gateways).
    - **AI services** add the need for **model registries**, versioning tools, and performance monitoring. A new version of a model can be published and discovered similarly to any service update.
    - Model metadata (e.g., accuracy, training data date) can be just as critical as WSDL or OpenAPI specs were for standard web services.
5. **Impact on Traditional SOA Concepts**
    - **Contracts and Interfaces**: AI endpoints tend to have **input/output schemas** for inference, but can evolve over time (e.g., new features or label sets). Contract versioning remains essential.
    - **Security and Governance**: Data privacy, user consent, and model explainability bring new layers of governance. Sensitive data used for inference requires strict controls and audits.
    - **Orchestration vs. Choreography**: AI services might be orchestrated in a **pipeline** (e.g., data cleaning → model inference → results aggregator) or use event-based triggers for asynchronous ML workflows.
6. **Example Use Cases**
    - **Image Recognition Service**: A separate microservice wraps a CNN (Convolutional Neural Network). A front-end application calls this service via an API to identify objects in uploaded images.
    - **Chatbot or NLP Endpoint**: A text analysis or large language model (LLM) is exposed as a REST or gRPC endpoint. Multiple consumer applications can tap into it for summarization, sentiment analysis, or Q&A.

    ○ **Fraud Detection Pipeline**: Transaction data is streamed to an **AI microservice** that continuously scores for fraud risk, returning responses to a parent billing or payment service in near real-time.

Benefits and Challenges of SOA

Benefits of SOA

- **Business Agility:**
    - SOA enables organizations to respond quickly to changing market conditions and business requirements.
    - ***Real Use Case* Netflix**: Netflix employs SOA to continuously innovate its streaming platform. With SOA, Netflix can rapidly introduce new features, personalize recommendations and scale its infrastructure to accommodate fluctuations in viewer demand. For example, Netflix's recommendation service analyzes user preferences in real-time, leveraging microservices to deliver personalized content recommendations instantly.
- **Interoperability:**
    - SOA promotes interoperability by standardizing communication protocols and data formats.
    - ***Real Use Case*: Salesforce.com:** Salesforce.com leverages SOA to integrate its cloud-based CRM platform with various third-party applications and services. Through standardized APIs and web services, Salesforce enables seamless data exchange between its CRM system and other business systems, such as marketing automation tools, ERP systems and customer support platforms.

Challenges in Implementation

- **Cultural Resistance to Change:**
    - Implementing SOA often requires cultural shifts within organizations, as it may disrupt traditional development practices and organizational structures.

    - ***Real Use Case*: Banking Industry:** Large banks often face cultural resistance when transitioning to SOA due to the legacy nature of their systems and the hierarchical structure of their IT departments. Developers may be accustomed to working in silos and there may be resistance from management to adopt new development methodologies. Overcoming this resistance requires strong leadership, effective communication and a focus on the benefits of SOA for delivering customer-centric solutions.
- **Complexity in Governance and Management:**
    - SOA introduces complexity in governance, management and lifecycle management of services.
    - ***Real Use Case*: Government Services**: Government agencies implementing SOA face challenges in managing service lifecycles, ensuring data security and maintaining compliance with regulations. For example, a government agency responsible for citizen services may struggle with governing access to sensitive data across multiple departments and agencies. Implementing robust governance frameworks and security policies is essential to address these challenges and ensure the integrity and confidentiality of citizen data.

Realizing the Benefits

- **Best Practices for Implementation:**
    - ***Real Use Case*: Amazon Web Services (AWS):** AWS provides a comprehensive set of cloud services built on SOA principles. By offering a wide range of modular services, such as computing, storage and databases, AWS enables organizations to build scalable and resilient applications. Best practices include leveraging AWS services in a decoupled manner, implementing auto-scaling and fault-tolerant architectures and continuously monitoring and optimizing performance.
- **Continuous Improvement and Adaptation:**
    - SOA is an iterative process that requires continuous improvement and adaptation to changing business needs and technology landscapes.

- ***Real Use Case*: Uber:** Uber continually evolves its platform using SOA principles to meet the demands of its global user base. By breaking down its monolithic architecture into microservices, Uber can deploy new features independently, optimize performance and scale its infrastructure dynamically. Continuous improvement involves gathering feedback from users, monitoring system performance and iteratively enhancing services to deliver a seamless and Microservices Architecture

Contemporary Trends of SOA

Microservices architecture is an architectural style that structures an application as a collection of loosely coupled services, each responsible for a specific business function and independently deployable.

- **Key Characteristics:**
  - **Service Decomposition**: Applications are decomposed into smaller, independently deployable services, each responsible for a specific business capability.
  - **Decentralized Data Management:** Each service manages its own database or data store, enabling greater autonomy and scalability.
  - **Polyglot Persistence**: Services can use different databases or data storage technologies based on specific requirements.
  - **Infrastructure Automation**: Microservices rely on automation for deployment, scaling and monitoring to ensure resilience and reliability.
- **Real Use Case: Netflix**
  - Netflix transitioned from a monolithic architecture to a microservices-based architecture to support its rapid growth and global expansion.
  - Each microservice at Netflix handles a specific function, such as user authentication, content recommendation, billing and streaming.
  - This architecture enables Netflix to scale its services independently, deploy updates faster and deliver personalized experiences to millions of users worldwide.

Cloud Computing and SOA

Cloud computing is the delivery of computing services—including servers, storage, databases, networking, software and analytics—over the internet to offer faster innovation, flexible resources and economies of scale.

- **Key Characteristics:**
  - **On-Demand Self-Service:** Users can provision and manage computing resources, such as servers and storage, without human intervention.
  - **Resource Pooling**: Cloud providers pool and dynamically allocate resources to multiple users, optimizing resource utilization and scalability.
  - **Pay-Per-Use Billing**: Users pay only for the resources they consume, enabling cost-effective and scalable solutions.
  - **Scalability and Elasticit**y: Cloud services can scale up or down based on demand, ensuring performance and availability.
- **Real Use Case: Airbnb**
  - Airbnb leverages cloud computing services, such as Amazon Web Services (AWS), to power its online marketplace for lodging and tourism experiences.
  - By using cloud infrastructure, Airbnb can quickly scale its services to accommodate spikes in demand during peak booking seasons or events.
  - Additionally, cloud-based analytics and machine learning services enable Airbnb to personalize search results, recommend listings and optimize pricing for hosts.

Serverless Computing and SOA

Serverless computing is a cloud computing model where cloud providers manage the infrastructure, dynamically allocating resources as needed and users only pay for the compute resources consumed by their applications.

- **Key Characteristics:**

- **No Server Management:** Users do not need to provision, manage, or maintain servers or infrastructure, allowing for faster development and deployment.
- **Event-Driven Architecture**: Serverless applications are event-driven and respond to triggers or events, such as HTTP requests, database changes, or messages from queues.
- **Auto-Scaling**: Serverless platforms automatically scale resources based on demand, ensuring high availability and performance without user intervention.
- **Pay-Per-Use Billing**: Users are billed based on the actual resources consumed by their applications, offering cost savings and efficiency.

- **Real Use Case: Lyft**
  - Lyft utilizes serverless computing for its backend infrastructure to handle millions of ride requests and data-intensive operations in real-time.
  - By adopting a serverless architecture on AWS Lambda, Lyft can dynamically scale its backend services in response to user demand, ensuring low-latency responses and optimal performance.
  - Serverless computing enables Lyft to focus on building and improving its core ride-sharing platform without worrying about managing servers or infrastructure.

Unit-2 SOA Design and Modeling

Service Design Principles and Patterns

Service Design Principles and Patterns form the foundations of effective Service-Oriented Architecture (SOA). This unit delves into the essential concepts and strategies for designing services that are cohesive, granular and adaptable to change. Understanding these principles and patterns will provide insight into creating robust, scalable and maintainable service-oriented systems.

Service Coupling

Definition: Service Coupling refers to the degree of interdependence between any two business processes or services within a system.

Preferable State: In SOA, **weak coupling is preferred**, indicating lower dependency for increased flexibility, scalability and maintainability.

- Explanation:
  - Weak coupling allows services to evolve independently, reducing the risk of unintended consequences when modifications or updates are made.
  - Services with weak coupling can adapt more easily to changes in business requirements, ensuring that adjustments in one part of the system do not propagate unexpectedly to other interconnected services.
- Example:
  - A service employing standardized interfaces and protocols can interact with other services more loosely, minimizing the impact of changes in one service on others.
  - Example: Consider an e-commerce platform where a "Checkout Service" encapsulates functionalities such as processing payment, updating inventory and sending order confirmation emails. This service demonstrates strong cohesion by focusing on a cohesive set of operations related to completing the checkout process.

Service Cohesion

Definition: Service Cohesion refers to the degree of functional relatedness and focus of operations within a service.

Preferable State: **Strong cohesion is preferred in SOA**, indicating that a service should encapsulate closely related and well-defined functionalities.

- Explanation:
  - Strong cohesion ensures that a service encapsulates a well-defined and closely related set of functionalities, enhancing clarity, maintainability and usability.
  - Cohesive services promote reusability and contribute to a modular and extensible architecture.
- Example:
  - A service responsible for order processing should encapsulate functionalities such as order validation, payment processing and inventory management, exhibiting strong cohesion.

Applying Coupling and Cohesion to SOA

- The principles of coupling and cohesion remain relevant in modern service-oriented systems.
- Analyzing different approaches, such as WS-* versus REST, reveals differences in coupling and cohesion. For example, in systems based on WS-, *interfaces often exhibit higher degrees of coupling due to their ad hoc and variable nature. Each service endpoint may have its own unique interface, leading to increased complexity and tighter coupling between services. Conversely, RESTful systems adhere to uniform interfaces, promoting loose coupling and greater cohesion. For instance, consider a banking application where WS-* services handle transactions with varying interfaces for different account types. In contrast, a RESTful approach may use a uniform interface for all account-related operations, such as

GET, POST, PUT and DELETE methods, leading to more cohesive service interactions and easier integration across the system.

- Creating understandable and maintainable Web service orchestrations requires considering the cohesion of services being orchestrated.

Service Granularity

Service design principles such as cohesion, granularity and design for change are fundamental to creating effective and maintainable service-oriented architectures. By adhering to these principles, organizations can develop robust and adaptable systems capable of meeting the dynamic needs of the business environment.

Definition: Service Granularity denotes the scope of functionality exposed by a service.

Preferable State: Coarse granularity is recommended in SOA, suggesting that services should provide broad functionalities to address specific needs, promoting reusability.

- Coarse-grained services encapsulate broader and more encompassing functionalities, reducing the number of service invocations and promoting simplicity.
- Coarse granularity enhances service reuse, reduces the impact of changes on service interfaces and aligns with the goal of creating a modular and scalable architecture.
    - Example: A coarse-grained service responsible for customer management provides functions such as creating, updating and deleting customer profiles, providing a comprehensive set of functionalities within a single service interface.
    - Example: In a travel booking system, a "Reservation Service" might provide coarse-grained functionalities such as booking flights, hotels and rental cars in a single service call, enabling customers to make comprehensive travel arrangements efficiently.

Design for Change

- Designing services for change is essential in SOA to ensure adaptability to evolving business requirements.
- Service-oriented systems should be designed with flexibility and agility in mind, allowing services to evolve independently without impacting other parts of the architecture.
- Example: Consider a healthcare management system where a "Patient Information Service" is designed to accommodate changes in medical record formats or regulatory requirements. By encapsulating data access and manipulation logic within the service, it can adapt to evolving standards without affecting other components of the system.

Service Contract Design and Management

This section explores critical aspects of designing and managing service contracts in a Service-Oriented Architecture (SOA).

A **service contract** serves as the interface between service providers and consumers, defining the obligations, responsibilities and expectations of both parties. This enables creation of interoperable and extendable service contracts that facilitate seamless integration and collaboration within distributed systems.

Example: In a modern e-commerce platform, the use of OpenAPI Specification (known as **Swagger**) allows developers to define clear and standardized interfaces for various microservices responsible for product catalog, user authentication and payment processing. By utilizing OpenAPI Specification, developers can ensure consistent communication between services and enable seamless integration with third-party applications.

Contract-First Design

- Understanding the concept of Contract-First Design as a methodology for designing services from the perspective of their contracts.
- Discussing the advantages of Contract-First Design in promoting loose coupling, interoperability and alignment with business requirements.
- Case studies demonstrating the implementation of Contract-First Design principles in real-world service development projects.

Example: A telecommunications company adopts Contract-First Design when developing a new API for their billing system. By defining the contract (API specifications) first, based on the requirements gathered from stakeholders, the development team ensures that the API meets the exact needs of the consumers. Any changes or updates to the API contract are communicated and agreed upon before implementation, reducing the risk of compatibility issues.

Versioning and Evolution

- Addressing the challenges of versioning and evolution in service contracts over time.
- Strategies for managing backward and forward compatibility while introducing changes to service contracts.
- Best practices for versioning service contracts to ensure seamless migration and coexistence of multiple service versions.

Example: A cloud storage provider (e.g. AWS S3, DropBox) introduces a new version of its API to support additional features and improve performance. To ensure backward compatibility, the provider maintains support for the previous API version while allowing clients to migrate to the new version at their own pace. Through versioning and effective communication of changes, the provider minimizes disruptions for existing clients and facilitates the adoption of new features by offering clear migration paths.

Interface Definition Languages (IDLs)

Interface Definition Languages (IDLs) are formal languages used to describe the interfaces of software components, enabling communication and interaction between distributed systems. This lecture explores the role of IDLs in service-oriented architectures (SOAs), their key features and their application in modern software development.

Definition of IDLs

- IDLs provide a standardized way to define the structure, operations and data types of interfaces between software components.
- They facilitate interoperability by enabling communication between heterogeneous systems implemented in different programming languages or running on different platforms.

Features of IDLs

- **Interface Specification**: IDLs allow application developers to specify the methods, parameters and data types exposed by a software component's interface.
- **Language Neutrality**: IDLs are independent of programming languages, allowing components written in different languages to communicate seamlessly.
- **Platform Independence:** IDLs abstract away platform-specific details, enabling components running on different operating systems or hardware architectures to interact.

Types of IDLs

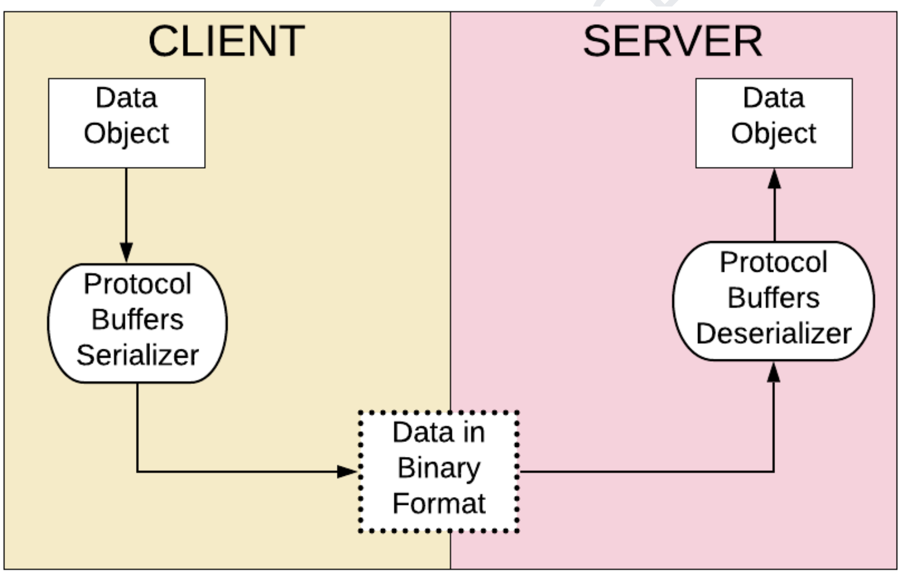
- Operation-Oriented IDLs: Focus on defining remote procedure calls (RPCs) and method invocations between distributed components. Examples include CORBA IDL and DCOM IDL.

- Data-Oriented IDLs: Primarily used for defining data structures and messages exchanged between systems. Examples include Google Protocol Buffers, Apache Thrift and Apache Avro.

Application of IDLs in SOA

- **Contract-First Design:** IDLs promote a contract-first approach to service design, where interfaces are defined and agreed upon before implementation.
- **Versioning and Evolution:** IDLs support versioning mechanisms, allowing services to evolve over time while maintaining backward compatibility.
- **Interoperability**: IDLs enable interoperability between services implemented in different languages or running on different platforms, fostering a heterogeneous and distributed ecosystem.

Protocol Buffers (protobuf)

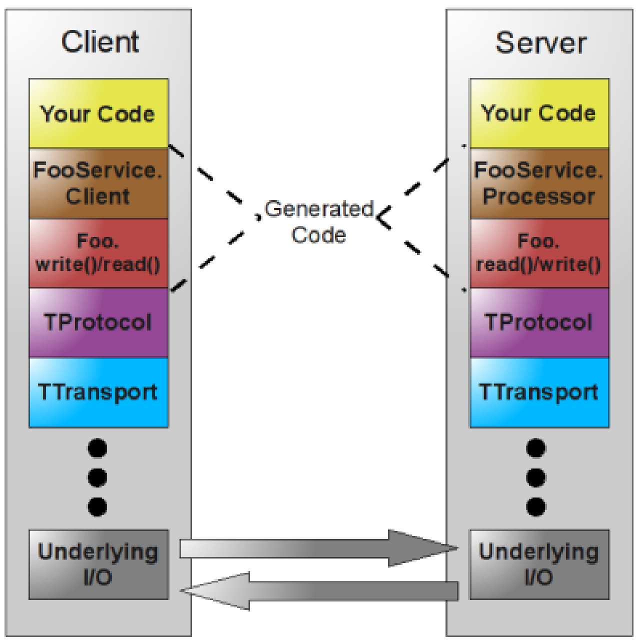


- Developed by Google, Protocol Buffers is a widely-used IDL for serializing structured data, particularly in microservices and cloud-native applications.

- Example: In a distributed messaging system, Protocol Buffers is used to define message formats for communication between microservices, ensuring efficient data serialization and deserialization.

Apache Thrift

- Apache Thrift is a cross-language IDL framework developed by Facebook, used for defining and communicating between services in diverse environments.



- Example: In a scalable web application, Apache Thrift is employed to define service interfaces for handling user authentication, session management and data storage, allowing seamless communication between backend services written in different languages.

Designing for Scalability and Resilience

Designing for Scalability and Resilience is essential for building robust and adaptable service-oriented architectures (SOAs) that can handle varying workloads and maintain availability under challenging conditions. This section describes key design principles and patterns, including Load Balancing, Fault Tolerance and the Circuit Breaker Pattern, along with modern application examples to illustrate their practical implementation.

- **Load Balancing in Cloud-Native Applications:**
  - Example: In a containerized microservices application deployed on **Kubernetes**, an ingress controller acts as a load balancer, distributing incoming HTTP traffic to pods running the same service. Kubernetes dynamically adjusts the load balancing configuration based on resource availability and service health.
- **Fault Tolerance in Serverless Computing:**
  - Example: In a serverless architecture for a real-time analytics platform, functions are deployed across multiple cloud providers to ensure fault tolerance and high availability. If one cloud provider experiences an outage, the platform automatically scales up instances in alternative regions to maintain service uptime.
  - Fault-tolerant systems include monitoring tools, such as **Netflix's Eureka** and stress-testing tools, like **Chaos Monkey.** They help to discover issues earlier by testing in pre-deployment environments, like integration (INT), quality assurance (QA) and user acceptance testing (UAT), to prevent potential problems before moving to the production environment.
- **Circuit Breaker Pattern in API Gateways:**
  - Example: In a **modern API gateway** (Kong, Envoy, Apigee) for a mobile banking application, circuit breakers are implemented to protect against backend service failures. If the authentication service experiences errors, the circuit breaker opens, temporarily routing requests to a cached authentication token to maintain user session integrity.

Load Balancing

- **Definition**: Load Balancing is the process of distributing incoming network traffic across multiple servers to ensure optimal resource utilization and prevent overload on any single server.
- **Application:** In SOA, load balancers are used to evenly distribute requests among service instances, improving scalability and responsiveness.
- **Example:** In a cloud-based e-commerce platform, a load balancer distributes incoming web traffic across multiple instances of the Product Catalog Service, ensuring that no single instance becomes overwhelmed during peak shopping periods.
    - Nginx - https://www.youtube.com/watch?v=MxPVAaBb-wA
    - HAProxy - https://www.youtube.com/watch?v=qYnA2DFEELw

Fault Tolerance

- **Definition**: Fault Tolerance refers to the ability of a system to continue operating properly in the event of component failures or disruptions.
- **Application**: In SOA, fault-tolerant designs incorporate redundancy, error handling and failover mechanisms to mitigate the impact of failures on system availability.
- **Example**: In a financial trading application, redundant instances of the Order Execution Service are deployed across geographically distributed data centers. If one data center experiences an outage, traffic is automatically redirected to the backup data center to maintain service continuity.
- **Reference**: https://opensource.com/article/19/3/tools-fault-tolerant-system

Circuit Breaker Pattern

- **Definition**: The Circuit Breaker Pattern is a design pattern used to handle faults and failures in distributed systems by temporarily suspending requests to a failing service.
- b: In SOA, circuit breakers monitor the health of downstream services and prevent cascading failures by quickly detecting and isolating faulty components.

- **Example**: In a microservices architecture for a social media platform, a circuit breaker is implemented in the Notification Service to prevent excessive retries when sending notifications to users. If the Notification Service experiences a high error rate, the circuit breaker opens, temporarily halting requests to the service and preventing overload.

Unit-3 SOA Implementation Techniques

Web Services Standards

Web services standards define the protocols and formats used for communication between different software applications over the internet. These standards enable interoperability and integration between heterogeneous systems, allowing them to exchange data and invoke functionality seamlessly.

Simple Object Access Protocol

- SOAP is a protocol used for exchanging structured information between systems.
- It defines a standard XML format for messages, which typically include headers and bodies.
- SOAP messages are typically transported over HTTP. Other protocols like SMTP (Simple Mail Transfer Protocol) and JMS (Java Message Service) can also be used.
- SOAP provides a robust messaging framework with features such as security, reliability and transactionality.
- It follows a contract-based approach, where the structure of messages and operations is defined in a WSDL (Web Services Description Language) document.

Representational State Transfer (REST)

- REST is an architectural style for designing networked applications, emphasizing simplicity, scalability and statelessness.
- It relies on standard HTTP methods such as GET, POST, PUT, DELETE for performing CRUD (Create, Read, Update, Delete) operations on resources.
- RESTful APIs expose resources as URIs (Uniform Resource Identifiers) and use HTTP status codes for indicating the outcome of operations.
- REST APIs are lightweight, easy to understand and widely adopted for building web services, especially for public-facing APIs.

Graph QL

- GraphQL is a query language and runtime for APIs developed by Facebook.
- It allows clients to specify exactly what data they need, enabling more efficient and flexible data retrieval compared to traditional REST APIs.
- With GraphQL, clients can request multiple resources in a single query and receive only the data they ask for, reducing over-fetching and under-fetching of data.
- GraphQL APIs are introspective, meaning they expose a schema that describes the types of data available and the operations that can be performed.

Examples and Code Snippets

SOAP:

- Example: Integrating a payment gateway API into an e-commerce platform.
- **Code Snippet: SOAP**

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="http://www.example.com/webservice">
   <soapenv:Header/>
   <soapenv:Body>
      <web:ProcessPayment>
         <web:Amount>100.00</web:Amount>
         <web:CardNumber>1234567890123456</web:CardNumber>
         <!-- Additional Payment Details -->
      </web:ProcessPayment>
   </soapenv:Body>
</soapenv:Envelope>
```

REST:

- Real-life Example: Retrieving weather data from a public API.
- **Code Snippet (Python using requests library)**

```
import requests

url = "https://api.weather.com/data"
params = {"city": "Bangalore", "format": "json"}
```

```
response = requests.get(url, params=params)
weather_data = response.json()
print(weather_data)
```

GraphQL:

- Example: Fetching user profile data from a social media platform API.
- Code Snippet (GraphQL query)

```
query {
  user(id: "123@fb.com") {
    id
    name
    email
    posts {
      id
      title
      content
    }
  }
}
```

Microservices Architecture and its Relationship with SOA

Microservices architecture is an approach to developing software applications as a collection of small, independently deployable services. Each service is self-contained, focused on a specific business capability and communicates with other services through well-defined APIs.

Decentralized Data Management:

- In microservices architecture, each service manages its own data store, which is often optimized for the service's specific requirements.
- This decentralized approach to data management allows services to be more autonomous and reduces dependencies between services.
- Services can choose the most suitable data storage technology for their needs, such as relational databases, NoSQL databases, or in-memory caches.

Example:

Consider a social media platform where each microservice handles a specific functionality, such as user management, post management and notification handling. Each service manages its own database tailored to its requirements, enabling flexibility and scalability.

Code Snippet

```
# Example of a microservice handling user management

class UserService:
    def __init__(self, db):
        self.db = db

    def create_user(self, user_data):
        # Code to create a new user in the user database
        pass

    def get_user(self, user_id):
        # Code to retrieve user information from the user database
        pass

# Example usage
user_db = UserDatabase()
user_service = UserService(user_db)
user_service.create_user(user_data)
```

Independent Deployment:

- Microservices can be independently deployed, updated and scale
- s and fixes more frequently, improving agility and time-to-market.
- Each service can have its own deployment pipeline, testing strategy and release schedule, reducing coordination overhead.

Example:

In a retail application, the product catalog service can be updated with new product information independently of the checkout service. This allows the product team to release updates to the catalog without waiting for the checkout team, enabling faster innovation.

Code Snippet

```
# Example deployment configuration for a microservice
services:
  - name: product-catalog
    version: v1.2.0
    replicas: 3
    image: product-catalog:v1.2.0
    ports:
      - 8080
    environment:
      - ENVIRONMENT=production
      - DATABASE_URL=postgres://user:password@10.2.2.3:5432/catalog
```

Infrastructure Automation

- Microservices architecture relies heavily on automation for provisioning, scaling and managing infrastructure.
- Infrastructure is often defined as code using tools like Terraform or Kubernetes, allowing for consistent and repeatable deployments.
- Automation enables efficient resource utilization, improves system reliability and reduces manual overhead.

Example:

In a cloud-native microservices application, infrastructure resources such as virtual machines, containers and networking are provisioned and managed automatically using Infrastructure as Code (IaC) tools like Terraform or AWS CloudFormation.

Code Snippet:

```
# Example Terraform configuration for provisioning AWS resources
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = {
    Name = "example-instance"
  }
}
```

Relationship with SOA

- Microservices architecture shares similarities with Service-Oriented Architecture (SOA) in its focus on modularization, loose coupling and service autonomy.
- Both architectures aim to improve agility, scalability and maintainability by breaking down monolithic systems into smaller, more manageable components.
- However, microservices tend to be more fine-grained and decentralized compared to traditional SOA, which often relies on heavyweight middleware and centralized governance.

Example

A comparison between a traditional SOA implementation and a microservices-based approach in a banking application. While SOA might involve large, monolithic services managed by a central ESB (Enterprise Service Bus), microservices would consist of smaller, independently deployable services handling specific banking functions like account management, transactions and customer notifications.

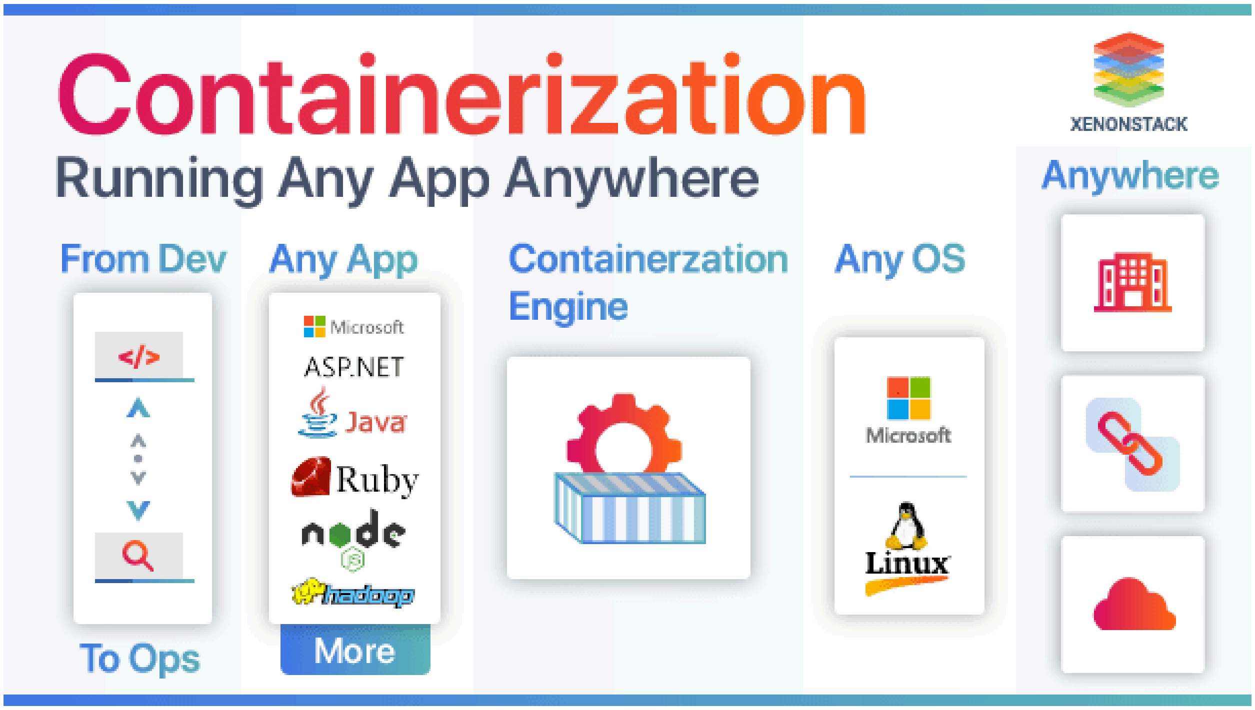
Code Snippet

```
// Example microservice handling transaction processing
@RestController
@RequestMapping("/transactions")
public class TransactionController {

    @Autowired
    private TransactionService transactionService;

    @PostMapping("/process")
    public ResponseEntity<Transaction>
processTransaction(@RequestBody TransactionRequest request) {
        Transaction transaction =
transactionService.processTransaction(request);
        return ResponseEntity.ok(transaction);
    }
}
```

Containerization and Orchestration

Containerization

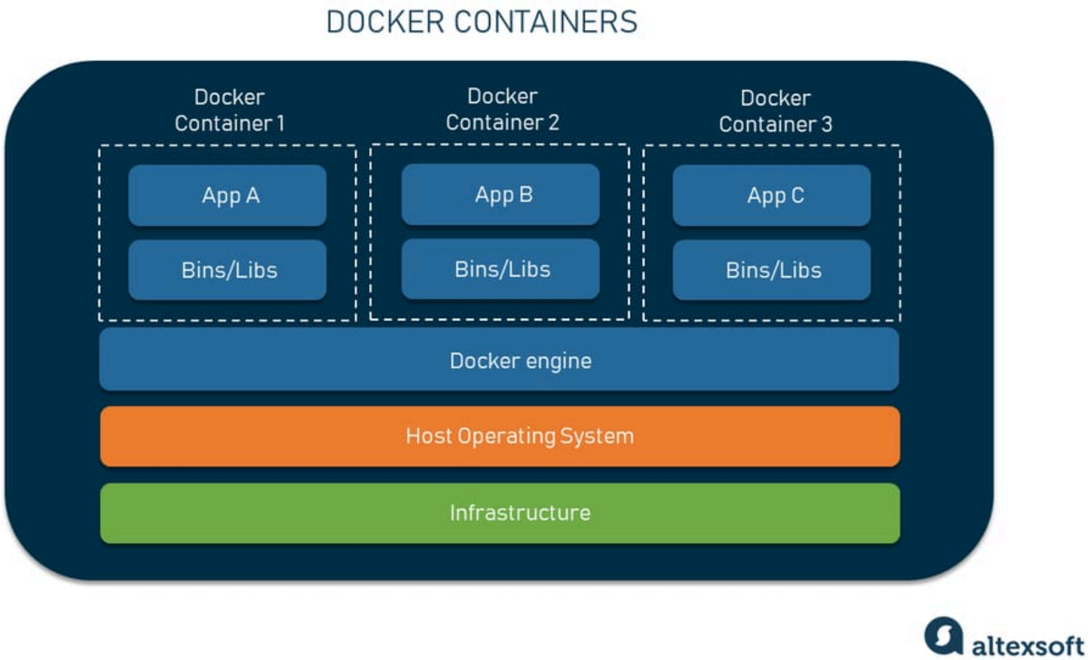
Containerization is a lightweight, portable and efficient method for packaging, distributing and running applications. Containers encapsulate everything needed to run an application, including the code, runtime, libraries and dependencies, into a single unit.



Source: https://www.xenonstack.com/insights/containerization

Docker Container

- Docker is a leading containerization platform that allows developers to build, ship and run applications in containers.
- Docker containers are isolated environments that share the host operating system's kernel, providing consistency across different environments.
- Docker uses Dockerfiles to define container configurations and Docker images to package applications and their dependencies.



Source: Alexsoft

Containerization - Pros and Cons

Reference: https://www.xenonstack.com/insights/containerization

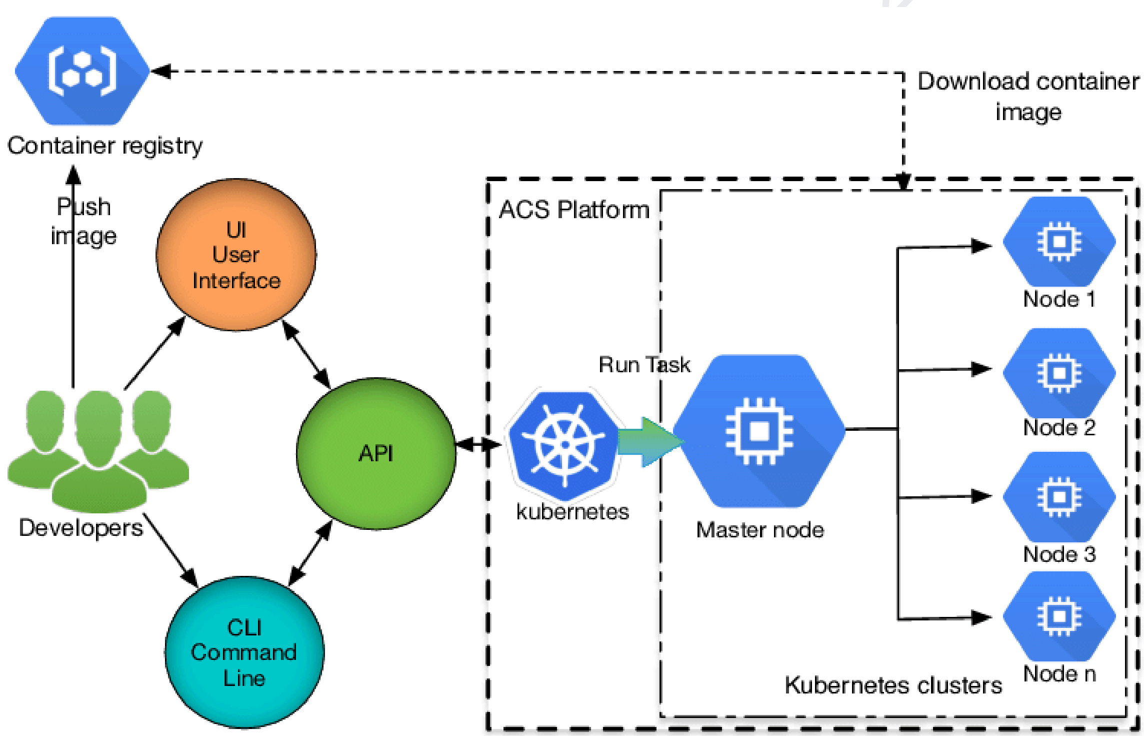
- Pros
  - Portability - no dependency on hardware, containers (dockers) abstracts running of application on any host
  - Lightweight - contains only application specific requirements and no unnecessary OS overhead, keep it lightweight
  - Speed - more faster and efficient in application bring up
  - Cost-effective - cost of running containers is much lower than running virtual machines
- Cons
  - Security - vulnerability of container engine and poor access control has associated risks
  - Manageability - managing large number of containers is challenging
  - Monitoring - needs a good monitoring system for effective maintenance and troubleshooting.

Example: A web application running in a Docker container:

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

Kubernetes Orchestration

- Kubernetes is an open-source container orchestration platform for automating the deployment, scaling and management of containerized applications.
- Kubernetes abstracts away underlying infrastructure complexities and provides features like automatic scaling, load balancing and self-healing.
- Kubernetes organizes containers into logical units called pods, which are the smallest deployable units in Kubernetes.



Example: A Kubernetes deployment manifest for a web application:

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: my-webapp:latest
        ports:
        - containerPort: 80
```

Service Mesh Technologies

- Service mesh technologies like Istio and Linkerd provide a dedicated infrastructure layer for handling service-to-service communication within a containerized environment.
- Service meshes offer features like traffic management, load balancing, encryption and observability to improve reliability, security and performance.
- Service mesh components, such as sidecar proxies, intercept and manage communication between services transparently.

Example- Istio service mesh

Istio configuration for implementing mutual TLS encryption between services:

```yaml
# destination-rule.yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: default-mtls
spec:
  host: "*.default.svc.cluster.local"
```

```
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

Event-Driven Architecture

Event-Driven Architecture (EDA) is an architectural pattern where the production, detection, consumption and reaction to events are central to the design. EDA enables decoupled, scalable and responsive systems by promoting loose coupling between components and allowing them to communicate asynchronously through events.

Event Sourcing

- Event Sourcing is a pattern where changes to an application's state are captured as a sequence of immutable events.
- Instead of storing the current state of an entity, Event Sourcing stores a log of events that represent state transitions over time.
- Event Sourcing enables reconstructing the current state of an entity at any point in time by replaying the events.

Example: FinTech app

In a banking application or a FinTech application, each transaction, such as deposits, withdrawals, UPI payments, Wallet updates, is recorded as an event. The current account balance is derived by replaying these events.

Command Query Responsibility Segregation (CQRS)

- Command Query Responsibility Segregation (CQRS) is a pattern that separates the responsibility of handling commands (write operations) from queries (read operations).
- In CQRS, different models are used to process commands and queries, allowing each model to be optimized for its respective use case.
- CQRS simplifies scalability, as read-heavy and write-heavy operations can be scaled independently.

Example: e-commerce platform

In an e-commerce platform, the command model handles order creation, modification and cancellation, while the query model handles product catalog queries and order history retrieval.

Event-Driven Messaging Systems

- Event-Driven Messaging Systems facilitate communication between decoupled components by sending and receiving events.
- Event messages contain information about a specific event, such as its type, timestamp and payload data.
- Messaging systems like Apache Kafka, RabbitMQ and Amazon SNS/SQS provide reliable, scalable and fault-tolerant event delivery.

Example: Ride-sharing App

A ride-sharing application uses event-driven messaging to notify drivers of ride requests, update the status of ongoing rides and handle payment transactions.

Code Snippet: Publishing an event to a message broker (using Apache Kafka):

```
from kafka import KafkaProducer
import json

producer = KafkaProducer(bootstrap_servers='localhost:9092',
                         value_serializer=lambda v:
json.dumps(v).encode('utf-8'))

event = {'type': 'amount_credited', 'transaction_id': '12345',
         'amount': 100.00}
producer.send('orders', value=event)
producer.flush()
```

API Management and Governance

API Management and Governance involve the planning, design, deployment and monitoring of APIs to ensure they meet business objectives, adhere to standards and provide a positive developer experience. It encompasses various aspects such as API design, documentation, security, versioning and usage policies.

API Design Principles

- API Design Principles focus on creating APIs that are intuitive, consistent and easy to use.
- Principles include using descriptive and meaningful endpoint URLs, following RESTful design principles, using HTTP methods appropriately and providing clear and concise documentation.

Example: Design an API for a weather service

Designing an API for a weather service that provides endpoints like `/weather/{city}` to retrieve weather information for a specific city and `/forecast/{city}` to get a weather forecast.

Developer Portals

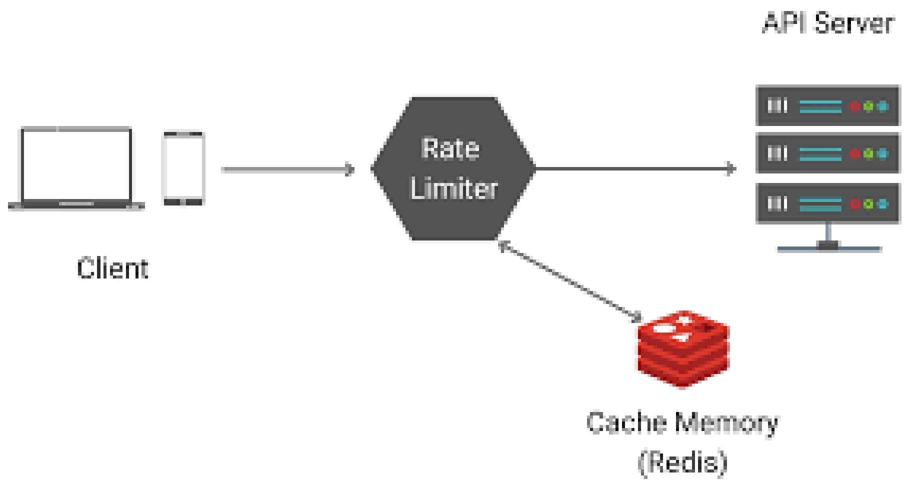
- Developer Portals are web-based platforms that provide developers with resources, documentation and tools for consuming APIs.
- Portals typically include API documentation, getting started guides, code samples, SDKs and interactive API explorers to facilitate API consumption.

Example*: Github Developer Portal*: https://github.com/topics/developer-portal

The GitHub Developer Portal offers comprehensive documentation, tutorials and API reference guides for developers integrating with GitHub's APIs.

Rate Limiting and Quotas

- Rate Limiting and Quotas control the number of requests an API consumer can make within a specific time frame to prevent abuse and ensure fair usage.
- Rate limits are typically enforced based on factors such as API keys, user authentication, IP addresses or subscription plans.



Source: https://systemsdesign.cloud/SystemDesign/RateLimiter

Example: **Implementing rate limiting for a social media API to restrict users to 1000 requests per hour to prevent spamming and ensure server stability.**

Code Snippet: Implementing rate limiting using Flask and Redis:

```
from flask import Flask, jsonify, request
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
from redis import Redis

app = Flask(__name__)
limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["1000 per hour"]
)
redis = Redis(host='localhost', port=6379)

@app.route('/api/resource')
@limiter.limit("10 per minute")
def get_resource():
    return jsonify({'data': 'Resource data'})

if __name__ == '__main__':
    app.run(debug=True)
```

Unit-4 Security and Governance in SOA

Security Considerations in SOA

Security is a critical aspect of Service-Oriented Architecture (SOA) as it involves multiple interconnected services communicating over networks. Understanding and mitigating security risks is essential to protect sensitive data, maintain integrity and ensure compliance with regulations, such as India Data Privacy Data Protection, Europe GDPR, HIPPA and so on.

Understanding Threat Models

- Threat Models identify potential security threats and vulnerabilities that could compromise the confidentiality, integrity, or availability of services and data.
- Common threats include unauthorized access, data breaches, injection attacks, denial-of-service (DoS) attacks and man-in-the-middle (MitM) attacks.

Example: Threats in Fin-Tech

Identifying threat models for a banking application's SOA, including risks such as SQL injection attacks on database services, unauthorized access to customer account information and DoS attacks targeting transaction processing services.

Threat	Security Property Violated
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-repudiation
Information Disclosure	Confidentiality
Denial of Service	Availability
Elevation of Privilege	Authorization

Common Security Risks in SOA include:

- Insecure Authentication and Authorization: Weak authentication mechanisms or inadequate access controls can lead to unauthorized access to services.Methods to address the same are multi-factor authentication (MFA) ,Role-Based Access Control and centralised Identity and Access Management (IAM)
- Insecure Communication: Lack of encryption or improper configuration of transport layer security (TLS) can expose sensitive data to interception.Methods to address the same are to Ensure TLS is properly configured to use strong ciphers and protocols ,Encryption and Secure Protocols.
- Injection Attacks: Improper input validation and sanitization can result in injection attacks such as SQL injection or XML External Entity (XXE) injection and Secure Parsing.Methods to address the same are Validate all input data against a whitelist of allowed values,Use prepared statements and parameterized queries to prevent SQL injection and Use secure XML parsers that disable external entity resolution to prevent XXE attacks.
- Data Exposure: Inadvertent exposure of sensitive data through misconfigured APIs or insecure storage mechanisms.Methods to address the same are to Implement strict access controls on APIs to ensure only authorized users can access sensitive data,Encrypt sensitive data at rest using strong encryption standards and Conduct regular security audits and configuration reviews to identify and correct data exposure vulnerabilities
- Denial-of-Service (DoS) Attacks: Overloading services with excessive requests to disrupt normal operations.Methods to address the same are to Implement rate limiting to restrict the number of requests a client can make within a specified time frame, Use load balancers to distribute traffic across multiple servers, preventing any single server from becoming overwhelmed, Deploy IDPS to monitor network traffic and detect potential DoS attacks.

Example: Healthcare App

A healthcare organization's SOA faces security risks such as unauthorized access to patient records, interception of sensitive medical data during transmission between services and injection attacks targeting healthcare APIs.

key security risks specific to healthcare apps and methods to address them:

Security Risks in SOA for Healthcare Apps

1. **Data Breaches and Unauthorized Access:**
   - **Risk:** Sensitive patient information, including personal health information (PHI), is at risk of being accessed by unauthorized users.
   - **Addressing:**
     - **Strong Authentication:** Use multi-factor authentication (MFA) to ensure that only authorized personnel can access sensitive data.
     - **Access Controls:** Implement role-based access control (RBAC) to restrict access to data based on user roles and responsibilities.
2. **Insecure Communication:**
   - **Risk:** Data transmitted between services can be intercepted, leading to the exposure of sensitive information.
   - **Addressing:**
     - **Encryption:** Use TLS/SSL to encrypt data in transit. Ensure that all communications between services are encrypted.
     - **Secure Protocols:** Use secure communication protocols like HTTPS for all service interactions.
3. **Injection Attacks:**
   - **Risk:** Improper input validation can lead to injection attacks such as SQL injection or XML External Entity (XXE) injection, compromising data integrity and confidentiality.
   - **Addressing:**
     - **Input Validation:** Validate and sanitize all inputs to ensure they conform to expected formats and values.
     - **Prepared Statements:** Use prepared statements and parameterized queries to prevent SQL injection.

        - **Secure XML Parsing:** Use secure XML parsers that disable external entity resolution to prevent XXE attacks.

4. **Data Exposure through APIs:**
    - **Risk:** Misconfigured APIs can inadvertently expose sensitive data to unauthorized users.
    - **Addressing:**
        - **API Security:** Implement strict access controls and authentication mechanisms for all APIs.
        - **Data Masking:** Mask sensitive data in API responses where full exposure is not necessary.
        - **Regular Audits:** Conduct regular security audits and reviews of API configurations.
5. **Denial-of-Service (DoS) Attacks:**
    - **Risk:** Attackers can overwhelm services with excessive requests, causing disruption of normal operations.
    - **Addressing:**
        - **Rate Limiting:** Implement rate limiting to control the number of requests that can be made to a service within a specified period.
        - **Throttling:** Throttle requests to prevent any single client from overloading the system.
        - **DDoS Protection:** Use DDoS protection services to filter and manage malicious traffic.
6. **Insufficient Logging and Monitoring:**
    - **Risk:** Lack of proper logging and monitoring can hinder the detection and response to security incidents.
    - **Addressing:**
        - **Comprehensive Logging:** Implement comprehensive logging of all access and activity within the system.
        - **Monitoring:** Use Security Information and Event Management (SIEM) systems to monitor logs and detect anomalies in real-time.
7. **Weak Service Discovery Mechanisms:**
    - **Risk:** Insecure service discovery can lead to the discovery and use of unauthorized or rogue services.

- **Addressing:**
    - **Secure Service Registry:** Implement secure access controls and encryption for the service registry.
    - **Authentication:** Ensure that only authenticated services can register and discover other services.

8. **Compliance and Regulatory Risks:**
    - **Risk:** Failure to comply with healthcare regulations such as HIPAA can result in legal and financial penalties.
    - **Addressing:**
        - **Compliance Audits:** Conduct regular compliance audits to ensure all services adhere to regulatory requirements.
        - **Data Protection Policies:** Implement and enforce policies for data protection, access control and data handling in accordance with healthcare regulations.
9. **Integration with Legacy Systems:**
    - **Risk:** Legacy systems may have outdated security measures, posing risks when integrated with modern SOA services.
    - **Addressing:**
        - **Secure Integration:** Use secure integration methods and protocols to interface with legacy systems.
        - **Security Patches:** Ensure that all legacy systems are updated with the latest security patches and upgrades.
10. **Insufficient Data Validation and Sanitization:**
    - **Risk:** Poor data validation can lead to vulnerabilities such as injection attacks and data corruption.
    - **Addressing:**
        - **Data Validation:** Implement rigorous data validation and sanitization processes.
        - **Frameworks:** Use secure coding frameworks that enforce data validation standards.

Security Design Patterns:

- Security Design Patterns are reusable solutions to common security problems in software architecture.
- Patterns such as these help address security concerns and reduce threats:
  - Multi-factor Authentication
  - Least Privilege Authorization Policy and Role based access control
  - Secure Communication using TLS
  - Input Validation to check for malicious input and threat injection
  - Continuous Audit Logging

Data Encryption and Integrity

In Service-Oriented Architecture (SOA), data encryption (**refers to providing confidentiality**) and integrity **(refers to protection against tampering)** are crucial for ensuring that sensitive information remains confidential and unaltered during transmission between services. This section covers message-level encryption, digital signatures, secure hash algorithms and best practices for securing APIs and web services.

Message-Level Encryption and Digital Signatures

Message-Level Encryption (XML Encryption)

- Definition: XML Encryption is a standard for encrypting XML data to ensure that the information is only accessible to authorized parties.
- Use Cases:
  - Protecting sensitive data in XML documents, such as credit card numbers or personal information.
  - Ensuring confidentiality in SOA where messages traverse multiple intermediaries.

Example:

```xml
<EncryptedData>
  <CipherData>
    <CipherValue>A23B45C67D89E0...</CipherValue>
  </CipherData>
</EncryptedData>
```

- Process:
    1. Generate a Symmetric Key: For encrypting the data.
    2. Encrypt the Data: Using the symmetric key.
    3. Encrypt the Symmetric Key: With the recipient's public key for secure transmission.

Digital Signatures (XML Signature)

- Definition: XML Signature is a standard for digitally signing XML data to ensure data integrity and authenticity.
- Use Cases:
    - Verifying that the data has not been altered during transmission.
    - Authenticating the sender of the XML message.

```xml
Example:
<Signature>
  <SignedInfo>
    <SignatureMethod Algorithm="..."/>
    <Reference URI="...">
      <DigestMethod Algorithm="..."/>
      <DigestValue>abc123...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>def456...</SignatureValue>
  <KeyInfo>...</KeyInfo>
```

```
</Signature>
```

- Process:
    1. Hash the Data: Using a secure hash algorithm.
    2. Sign the Hash: With the sender's private key.
    3. Attach the Signature: To the XML message.

Secure Hash Algorithms (SHA)

- Definition: Secure Hash Algorithms (SHA) are cryptographic hash functions used to generate a fixed-size hash value from variable input data, ensuring data integrity.
- Types:
    - SHA-1: Produces a 160-bit hash value. (Not recommended due to vulnerabilities)
    - SHA-256: Produces a 256-bit hash value. (Part of the SHA-2 family)
    - SHA-3: The latest member of the Secure Hash Algorithm family

```
// Example of computation of SHA-256 hash
import hashlib
data = "Hello, World!"
hash_object = hashlib.sha256(data.encode())
hex_dig = hash_object.hexdigest()
print(hex_dig)  # Outputs the SHA-256 hash of the input data
```

Ensuring Data Integrity in SOA

1. **Message Digest Generation:**
    - When a service sends data (message), it generates a hash value (digest) using a SHA algorithm.
    - This hash value represents the original data in a fixed-size string, typically 256 bits for SHA-256.
2. **Transmission of Data and Digest:**

    - The data and its corresponding hash value are transmitted to the receiving service.
    - The hash value can be sent along with the data or through a separate secure channel.
3. **Verification at the Receiving End**:
    - Upon receiving the data, the receiving service generates a new hash value from the received data using the same SHA algorithm.
    - The newly generated hash value is then compared with the original hash value sent by the sender.
4. **Integrity Check:**
    - If the two hash values match, it confirms that the data has not been altered during transmission. This ensures data integrity.
    - If the hash values do not match, it indicates that the data has been tampered with or corrupted and appropriate actions can be taken.

Implementation in SOA

1. **Digital Signatures**:
    - SHA is often used in conjunction with digital signatures to ensure data integrity and authenticity.
    - The sender creates a hash of the data and encrypts it with their private key to create a digital signature.
    - The recipient decrypts the signature using the sender's public key and compares the hash value with the hash of the received data.
2. **Message Authentication Code (MAC)**:
    - SHA can be used to generate a Message Authentication Code (MAC) when combined with a secret key (HMAC - Hash-based Message Authentication Code).
    - The sender computes an HMAC of the data using SHA and a shared secret key and the recipient verifies it using the same key and algorithm.
3. **WS-Security:**

- In SOA, Web Services Security (WS-Security) standards often use SHA algorithms to ensure message integrity.
- WS-Security allows for the inclusion of security headers in SOAP messages, which can contain hash values and digital signatures.

Uses in SOA:

- Ensuring data integrity by generating and comparing hash values before and after transmission.
- Creating digital signatures by hashing data before signing it.

SOA API Security

APIs (Application Programming Interfaces) are critical components of modern software architecture, enabling communication between different services and applications. Ensuring the security of APIs, especially in a Service-Oriented Architecture (SOA) environment, is crucial to protect sensitive data and maintain the integrity of services.

RESTful APIs are widely used in SOA due to their simplicity and scalability. Here are some best practices for securing RESTful APIs:

1. **Authentication and Authorization**
    - Use OAuth2 and OpenID for authentication.
    - OAuth2 is an industry-standard protocol for authorization. OpenID Connect builds on OAuth2 to add authentication.
2. **Data Encryption**
    - Always use HTTPS to encrypt data in transit between the client and the API server.
    - This prevents eavesdropping and man-in-the-middle attacks.
3. **Message-Level Encryption**:
    - Use message-level encryption for sensitive data within the API payload.

- Encrypt specific fields or the entire message body.

4. **Input Validation**
    - **Sanitize Inputs**: Validate and sanitize all inputs to prevent injection attacks such as SQL injection and cross-site scripting (XSS).
    - Use libraries or frameworks that provide built-in input validation.
5. **Rate Limiting**:
    - Implement rate limiting to prevent abuse and denial of service attacks.
    - Set thresholds for the number of requests a client can make in a given time period.
6. **Error Handling**
    - Avoid exposing detailed error messages that might reveal internal server information.
    - Use generic error messages and log detailed errors internally.
7. **Logging and Monitoring**
    - **Log Requests and Responses**: Keep detailed logs of API requests and responses for auditing and troubleshooting.
    - Ensure logs do not contain sensitive information.
8. **Monitor API Traffic**:
    - Use monitoring tools to track API usage and detect unusual patterns or potential security breaches.

Unit-5 SOA Emerging Trends

Some of the emerging trends in SOA are:
1. Serverless Computing
2. AI/ML in SOS
3. Edge computing and SOA integration

Serverless Computing

Serverless computing is a cloud computing execution model where the cloud provider dynamically manages the allocation and provisioning of servers. It abstracts the server management from developers, allowing them to focus on writing code.

Serverless computing includes services like:

- Function-as-a-Service (FaaS)
- Event-driven architectures

Function-as-a-Service

Function-as-a-Service (FaaS) is a serverless computing service that allows developers to execute individual functions, pieces of business logic, in response to events without managing the underlying infrastructure.

Reference: https://blog.back4app.com/what-are-serverless-functions-in-cloud-computing/

Key Features

1. **Event-Driven Execution**:
    - Functions are triggered by events such as HTTP requests, database changes, or message queue events.
    - Example: AWS Lambda, Google Cloud Functions, Azure Functions.
2. **Automatic Scaling**:
    - Functions scale automatically with the volume of incoming requests.
    - No need for manual intervention to handle varying loads.
3. **Pay-per-Use**:
    - Billing is based on the actual usage, such as the number of requests and the duration of function execution.
    - Cost-effective compared to always-on server instances.

Operational Characteristics of Serverless Computing

1. **No Server Management**:
    - Cloud provider handles server maintenance, patching and scaling.
    - Developers focus on writing code.
2. **Scalability**:
    - Automatic scaling to handle varying loads without manual intervention.
    - Suitable for applications with unpredictable traffic patterns.
3. **High Availability**:
    - Built-in redundancy and fault tolerance provided by cloud providers.
    - Functions run across multiple availability zones.
4. **Cost Efficiency**:
    - Pay only for actual usage (execution time and resource consumption).
    - No charges for idle resources.
5. **Quick Deployment**:
    - Rapid deployment of functions without lengthy provisioning times.
    - Accelerates development cycles and time-to-market.

Challenges

1. **Cold Starts**:
    - Initial latency when a function is invoked after being idle.
    - Can affect performance for latency-sensitive applications.

2. **Vendor Lock-In**:
    - Dependence on specific cloud provider services and APIs.
    - Challenges in migrating functions across different platforms.
3. **Complexity in Debugging**:
    - Distributed nature of serverless applications complicates debugging and monitoring.

Introduction to AI and ML in SOA

Artificial Intelligence (AI) and Machine Learning (ML) are transforming Service-Oriented Architectures (SOA) by enabling services to become more intelligent, adaptive and capable of handling complex tasks. Integrating AI and ML into SOA enhances decision-making, automates processes and improves user interactions.

Artificial Intelligence (AI) and Machine Learning (ML) are increasingly being integrated into Service-Oriented Architecture (SOA) to significantly enhance service capabilities. This integration leverages the strengths of both technologies to create more intelligent, efficient and adaptive service frameworks. Here's how AI and ML are being used within SOA:

1. **Service Optimization and Automation**:
    - **Predictive Analytics**: ML algorithms analyze historical data to predict future trends and behaviors, enabling proactive service adjustments and optimizations.
    - **Automated Decision-Making**: AI systems automate routine decisions and processes, reducing manual intervention and speeding up service delivery.
2. **Enhanced Data Processing**:
    - **Natural Language Processing (NLP)**: NLP capabilities enable services to understand and process human language, enhancing customer interaction through chatbots and virtual assistants.
    - **Data Integration and Management**: AI improves data integration from various sources, ensuring more accurate and real-time data availability for services.
3. **Personalization and Customization**:
    - **User Behavior Analysis**: ML models analyze user behavior and preferences, allowing services to offer personalized experiences and recommendations.
    - **Adaptive Services**: AI-driven services can adapt in real-time to user needs and contexts, providing more relevant and dynamic responses.
4. **Improved Security and Compliance**:

    - **Anomaly Detection**: AI algorithms detect unusual patterns and potential security threats, enhancing the security of the service ecosystem.
    - **Compliance Monitoring**: ML models continuously monitor service activities for compliance with regulations and standards, ensuring adherence and reducing risks.
5. **Operational Efficiency**:
    - **Resource Management**: AI optimizes the allocation and utilization of resources, improving operational efficiency and reducing costs.
    - **Performance Monitoring**: ML continuously monitors service performance, identifying bottlenecks and recommending improvements.
6. **Advanced Analytics and Insights**:
    - **Real-Time Analytics**: AI enables real-time analysis of service data, providing instant insights and enabling quick responses to emerging issues.
    - **Predictive Maintenance**: ML predicts when service components are likely to fail, allowing for preemptive maintenance and minimizing downtime.

By integrating AI and ML into SOA, organizations can create smarter, more responsive and efficient service architectures that better meet the evolving needs of users and businesses. This fusion not only enhances current capabilities but also opens up new possibilities for innovation and growth in service delivery.

Intelligent Agents

Intelligent agents **are autonomous entities that observe and act upon** an environment to achieve specific goals. In SOA, intelligent agents can enhance services by performing tasks such as decision-making, monitoring and automation.

Characteristics

1. **Autonomy**:
    - Operate without human intervention.
    - Make decisions based on predefined rules or learned behaviors.
2. **Reactivity**:
    - Respond to changes in the environment in real-time.

    - Example: Monitoring system performance and alerting when anomalies are detected.
3. **Proactivity**:
    - Take initiative to perform tasks or make recommendations.
    - Example: Proactively scaling resources based on predicted load.
4. **Learning Ability**:
    - Adapt and improve over time through learning from data and interactions.
    - Example: Personalizing user experiences based on behavior analysis.

Example Use Case

- **Service Health Monitoring**:
    - An intelligent agent monitors the health of various services in an SOA.
    - Uses ML models to predict failures and automatically takes corrective actions (e.g., restarting a service, scaling resources).

Predictive Analytics

Predictive analytics involves using statistical algorithms and ML techniques to analyze historical data and make predictions about future events. In SOA, predictive analytics can optimize services by anticipating needs and preventing issues.

Key Techniques

1. **Regression Analysis**:
    - Predicts a continuous outcome based on one or more predictor variables.
    - Example: Forecasting demand for a service based on usage patterns.
2. **Classification**:
    - Assigns items to predefined categories based on input data.
    - Example: Classifying incoming support tickets to route them to the appropriate department.
3. **Time Series Analysis**:
    - Analyzes sequential data points to forecast future values.
    - Example: Predicting server load to preemptively allocate resources.

Example Use Case

- **Customer Behaviour Prediction**:

- Analyze customer interaction data to predict next intents of a customer
- Use these insights to proactively engage at-risk customers and reduce drop rates.

Natural Language Processing (NLP)

Natural Language Processing (NLP) is a branch of AI that focuses on the interaction between computers and humans through natural language. In SOA, NLP can enhance services by enabling them to understand and generate human language.

Key Applications

1. **Chatbots and Virtual Assistants**:
   - Automate customer support and interactions.
   - Example: A customer service bot that answers queries and processes requests.
2. **Text Analysis**:
   - Extract meaningful information from text data.
   - Example: Analyzing customer feedback to identify common issues and sentiments.
3. **Machine Translation**:
   - Automatically translate text from one language to another.
   - Example: Translating service documentation for global users.

Example Use Case

- **Automated Customer Support**:
  - Implement an NLP-powered chatbot that interacts with customers, answers common questions and escalates complex issues to human agents.

lecture notes on: Edge Computing and SOA Integration - Edge Gateway Architectures, Low-Latency Data Processing. Offline Capabilities

Introduction to Edge Computing and SOA Integration

Edge computing refers to the practice of processing data near the edge of the network, where the data is generated, rather than in a centralized data center or cloud. Integrating edge computing with Service-Oriented Architectures (SOA) can enhance service performance by reducing latency, improving reliability and enabling offline capabilities.

Edge computing plays a crucial role in the evolution of Service-Oriented Architecture (SOA) by enhancing its capabilities and addressing some of its inherent limitations. Here are the key points highlighting the significance of edge computing in this context:

1. **Reduction in Latency**:
   - **Significance**: Edge computing processes data closer to its source, reducing the time it takes to send data to a central server and back. This is critical for SOA applications requiring real-time processing and low latency.
   - **Impact**: Improved responsiveness of services, making SOA more suitable for time-sensitive applications like IoT, autonomous vehicles and real-time analytics.
2. **Enhanced Scalability**:
   - **Significance**: By distributing computing tasks across multiple edge devices, edge computing supports horizontal scaling.
   - **Impact**: SOA can handle larger volumes of data and more complex service requests without overburdening centralized infrastructure, facilitating the growth of IoT networks and large-scale distributed systems.
3. **Improved Reliability and Resilience**:
   - **Significance**: Edge computing enhances system reliability by decentralizing processing power, reducing the impact of any single point of failure.
   - **Impact**: SOA systems become more robust and resilient, ensuring continuous service delivery even in the face of localized failures or network issues.
4. **Bandwidth Optimization**:
   - **Significance**: By processing and filtering data at the edge, only essential information is sent to the central servers, optimizing bandwidth usage.
   - **Impact**: Reduced network congestion and lower operational costs for SOA implementations, particularly beneficial for applications involving large data volumes, such as video streaming and sensor networks.
5. **Enhanced Security and Privacy**:
   - **Significance**: Edge computing allows sensitive data to be processed locally, reducing the need to transmit it over potentially insecure networks.
   - **Impact**: Increased data security and privacy for SOA services, making it easier to comply with data protection regulations and safeguard user information.
6. **Localized Decision-Making**:
   - **Significance**: Edge computing enables real-time, localized decision-making by processing data at the source.

    - **Impact**: SOA can support applications that require immediate responses, such as industrial automation, smart grids and healthcare monitoring systems, enhancing the overall effectiveness and applicability of SOA.
7. **Cost Efficiency**:
    - **Significance**: Reducing the need for extensive centralized computing resources and minimizing data transmission can lead to significant cost savings.
    - **Impact**: More cost-effective SOA implementations, particularly for businesses with extensive remote operations or those relying heavily on data-driven services.
8. **Improved User Experience**:
    - **Significance**: By reducing latency and ensuring more reliable service delivery, edge computing enhances the end-user experience.
    - **Impact**: SOA applications can provide faster, more reliable and context-aware services, improving customer satisfaction and engagement.

Edge Gateway Architectures

An edge gateway is a device that connects edge devices (sensors, IoT devices) to the cloud or data center. It acts as an intermediary that processes data locally, making decisions, filtering and aggregating data before sending it to the central systems.

Key Components

1. **Data Collection**:
    - Collects data from edge devices and sensors.
    - Example: An edge gateway collecting temperature data from IoT sensors.
2. **Local Processing**:
    - Processes data locally to reduce the amount of data sent to the cloud.
    - Example: Aggregating sensor data and performing initial analysis to detect anomalies.
3. **Connectivity**:
    - Provides communication between edge devices and the cloud.
    - Supports various protocols such as MQTT, HTTP and CoAP.
4. **Security**:
    - Ensures secure data transmission and storage.

    - Implements encryption, authentication and access control mechanisms.

Types of Edge Gateway Architectures

1. **Centralized Edge Gateway**:
    - **Architecture**: A single, powerful gateway that aggregates data from multiple edge devices and performs significant local processing.
    - **Use Case**: Suitable for environments where a central point can efficiently manage and process data, such as industrial automation or smart buildings.
2. **Distributed Edge Gateway**:
    - **Architecture**: Multiple smaller gateways distributed across various locations, each handling local data processing and communication.
    - **Use Case**: Ideal for large-scale, geographically dispersed networks like smart cities or wide-area IoT deployments.
3. **Hierarchical Edge Gateway**:
    - **Architecture**: Combines centralized and distributed approaches, with primary gateways aggregating data from secondary gateways or edge devices.
    - **Use Case**: Useful in complex environments requiring multiple levels of data processing and aggregation, such as multi-site industrial facilities.
4. **Mesh Edge Gateway**:
    - **Architecture**: Gateways form a mesh network, communicating with each other directly to share processing loads and data.
    - **Use Case**: Effective in scenarios requiring high resilience and flexibility, such as disaster recovery operations or military communications.
5. **Cloud-Integrated Edge Gateway**:
    - **Architecture**: Edge gateways closely integrated with cloud services, leveraging cloud resources for additional processing and storage as needed.
    - **Use Case**: Suitable for applications that benefit from both local processing and the extensive capabilities of cloud computing, like hybrid cloud environments in retail or healthcare.

Benefits of Edge Gateway Architectures

- **Reduced Latency**: Local data processing minimizes the time delay associated with sending data to central servers.

- **Improved Bandwidth Efficiency**: Only essential data is sent to the cloud, optimizing bandwidth usage.
- **Enhanced Security**: Local data processing and encryption reduce the risk of data breaches during transmission.
- **Scalability**: Distributed and hierarchical architectures support scalable deployment across large and diverse environments.
- **Reliability**: Mesh and hierarchical configurations enhance system resilience and reliability, ensuring continuous operation even if some gateways fail.

Example Architecture

1. **Smart City Traffic Management**

Architecture: Distributed Edge Gateway

Description: In a smart city, traffic management systems use distributed edge gateways to collect and process data from traffic sensors, cameras and IoT devices deployed across the city.

Components:

- **Edge Gateways**: Deployed at key intersections and traffic hubs, equipped with processors for local data analytics.
- **Sensors and Cameras**: Collect real-time data on vehicle movement, traffic density and environmental conditions.
- **Communication Interfaces**: 5G and Wi-Fi for real-time data transmission between sensors, edge gateways and central traffic management systems.
- **Local Storage**: Temporary storage for traffic data to buffer and manage network inconsistencies.
- **Security Modules**: Encrypt data and manage access control to ensure secure communication.

Functionality:

- **Real-time Traffic Analysis**: Edge gateways process data locally to manage traffic signals dynamically based on real-time conditions.
- **Anomaly Detection**: Immediate identification of traffic incidents or anomalies, such as accidents or congestion.
- **Data Aggregation**: Periodically sends aggregated data to the central traffic management system for long-term analysis and city-wide optimization.

2. Healthcare Remote Monitoring

Architecture: Cloud-Integrated Edge Gateway

Description: In healthcare, cloud-integrated edge gateways enable remote patient monitoring by collecting and processing health data from wearable devices and home medical equipment.

Components:

- **Edge Gateways**: Installed in patients' homes, connected to various medical devices.
- **Wearable Devices**: Track vital signs such as heart rate, blood pressure and glucose levels.
- **Communication Interfaces**: LTE/5G for real-time data transmission to healthcare providers and cloud services.
- **Local Storage**: Temporary storage of patient data for buffering and immediate processing.
- **Security Modules**: Ensure data privacy and secure communication.

Functionality:

- **Real-time Health Monitoring**: Local processing of health data for immediate alerts and notifications to patients and caregivers.
- **Data Aggregation**: Periodically sends aggregated health data to cloud services for comprehensive analysis and long-term health tracking.
- **Adaptive Treatment Plans**: Enables healthcare providers to adjust treatment plans based on real-time data insights.

- **Sensors and IoT Devices**:
  - Generate data continuously.
- **Edge Gateway**:
  - Collects and preprocesses data.
  - Applies local business logic and decision-making.
  - Sends relevant data to the cloud for further processing and storage.
- **Cloud**:
  - Provides centralized processing, analytics and long-term storage.
  - Manages and coordinates multiple edge gateways.

Low-Latency Data Processing

Low-latency data processing is crucial for applications requiring real-time or near-real-time responses, such as autonomous vehicles, industrial automation and augmented reality.

Techniques for Low-Latency Processing

1. **Local Data Processing**:
   - Processing data locally at the edge reduces the time needed to transmit data to a central server and wait for a response.
   - Example: Real-time video analytics on surveillance cameras.
2. **Edge Caching**:
   - Storing frequently accessed data locally to reduce retrieval time.
   - Example: Caching recent sensor readings to quickly provide historical context for new data.
3. **Event-Driven Architectures**:
   - Using event-driven models to trigger actions immediately when specific conditions are met.
   - Example: Triggering an alert when a sensor detects an abnormal condition.

Example Use Case

- **Autonomous Vehicles**:
  - Process sensor data (e.g., LIDAR, cameras) locally to make immediate driving decisions.
  - Only send summarized data to the cloud for long-term analysis and learning.

Offline Capabilities

Offline capabilities are essential for ensuring continuous operation in environments with intermittent or no connectivity, such as remote locations, transportation systems and disaster recovery scenarios.

Techniques for Enabling Offline Capabilities

1. **Local Data Storage**:
   - Store data locally during offline periods and synchronize with the cloud once connectivity is restored.
   - Example: A field device recording environmental data locally and uploading it when back online.
2. **Edge Computing Workloads**:
   - Run essential workloads locally to ensure continuous operation during connectivity outages.

    - Example: Local processing of critical alarms and alerts in an industrial setup.
3. **Graceful Degradation**:
    - Design systems to degrade gracefully by maintaining core functionalities when offline.
    - Example: An application that provides limited functionalities offline and full features online.

Example Use Case

- **Remote Monitoring Systems**:
    - A remote environmental monitoring system that collects data from various sensors.
    - Operates independently during connectivity outages and synchronizes data with the central server once connectivity is available.

Lab Exercises - Solution

Exercise 1: Overview of SOA: Implement a REST Web Service

Code Example: Develop a simple web service using a framework like Flask (Python), Spring Boot (Java), or Express (Node.js). Demonstrate how clients can consume this service to retrieve or manipulate data.

REST Web Service - Python Implementation (GET and POST Methods)

Prerequisites:

- Python Installation version 3.X
- Install Flask - use pip or pip3 based on your installation of python
- Update host firewall (if configured) to allow
- curl command

Install Flask using pip:

```
pip3 install flask
```

 # Save the following a file called webserver.py
 # Create a flask application

```
from flask import Flask, jsonify, request

app = Flask(__name__)

# Sample data
books = [
    {"id": 1, "title": "Book 1", "author": "Author 1"},
    {"id": 2, "title": "Book 2", "author": "Author 2"},
    {"id": 3, "title": "Book 3", "author": "Author 3"}
]

# Endpoint to get all books
@app.route('/books', methods=['GET'])
def get_books():
    return jsonify(books)
```

```
# Endpoint to get a specific book by id
@app.route('/books/<int:id>', methods=['GET'])
def get_book(id):
    book = next((book for book in books if book['id'] == id),
None)
    if book:
        return jsonify(book)
    else:
        return jsonify({"error": "Book not found"}), 404

# Endpoint to add a new book
@app.route('/books', methods=['POST'])
def add_book():
    data = request.json
    new_book = {
        "id": len(books) + 1,
        "title": data['title'],
        "author": data['author']
    }
    books.append(new_book)
    return jsonify(new_book), 201

if __name__ == '__main__':
    app.run(debug=True)
```

Run the application using python

```
python app.py
```

The Flask application should be running. Invoke the service using You can consume this service using HTTP client such as curl

To get all books, this demonstrates GET method of Web Service

```
curl http://localhost:5000/books
```

To get a specific book by id:

```
curl http://localhost:5000/books/1
```

To add a new book:

```
curl -X POST -H "Content-Type: application/json" -d
'{"title":"New Book","author":"New Author"}'
http://localhost:5000/books
```

Alternate command

curl -X POST -H "Content-Type: application/json" -d "{\"title\":\"New Book\", \"author\":\"New Author\"}" http://localhost:5000/books

REST Web Service - Spring Boot (Java) Implementation

First, make sure you have Spring Boot installed. Prerequisites:

- Install Spring Boot: https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started.installing
- Install Maven: https://maven.apache.org/
- Install TomCat server: https://tomcat.apache.org/download-10.cgi
- Configure Spring Boot and Tomcat https://www.baeldung.com/spring-boot-configure-tomcat
- Add Spring Boot dependencies to Maven by creating **`pom.xml`**:

// Create pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

// Create Java code

```
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;

import java.util.List;
import java.util.Optional;
```

```java
@SpringBootApplication
@RestController
public class Application {

    private List<Book> books = new ArrayList<>();

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @GetMapping("/books")
    public List<Book> getBooks() {
        return books;
    }

    @GetMapping("/books/{id}")
    public Book getBook(@PathVariable int id) {
        Optional<Book> result = books.stream().filter(book ->
book.getId() == id).findFirst();
        return result.orElse(null);
    }

    @PostMapping("/books")
    public Book addBook(@RequestBody Book book) {
        book.setId(books.size() + 1);
        books.add(book);
        return book;
    }
}

class Book {
    private int id;
    private String title;
    private String author;

    // Getters and setters

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
```

```
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

Run the application. Spring Boot will automatically start an embedded Tomcat server on port 8080 by default. You can now access the service using "curl" HTTP client.

// To get all books:

```
curl http://localhost:8080/books
```

// To get specific book

```
curl http://localhost:8080/books/1
```

// To add a new book:

```
curl -X POST -H "Content-Type: application/json" -d
'{"title":"New Book","author":"New Author"
```

Exercise 2: Principles and Concepts of SOA

- Code Example: Implement a basic service demonstrating loose coupling by using asynchronous messaging (e.g., RabbitMQ or Kafka). Create a publisher service that sends messages to a message broker and a consumer service that receives and processes these messages independently.

Pub-Sub: Demonstrate a Publisher-Subscriber message exchange using RabbitMQ.

Terminology

- A **message broker** is an intermediary service that helps reliable exchange messages from one service called "producer" or "publisher" to another service called "consumer" or "subscriber".
- **RabbitMQ** is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol (AMQP).
- One of the real time use cases of a message broker is for "**communication (comms)" service** used in banking, ticketing and e-commerce applications to post SMS or WhatsApp message to users about a transaction (e.g. credit / debit amount, order booking, shipment details, etc), In this case, a order booking service will post a message to "comms" service via message broker, this allows asynchronous and non-blocking communication between producer and consumer.

Prerequisites:

- Install Erlang
  - Erlang is a programming language developed by Ercisson in 1986. Erlang is the programming language used to code WhatsApp
  - RabbitMQ is written in Erlang
  - Erlang Installation:
    - Windows Installer: https://github.com/erlang/otp/releases/download/OTP-26.2.3/otp_win64_26.2.3.exe

- Install RabbitMQ
    - https://www.rabbitmq.com/docs/install-windows#installer

RabbitMQ tutorial - "Hello world!"

- This consists of two programs in Python; a producer (sender) that sends a single message and a consumer (receiver) that receives messages and prints them out. It's a "Hello World" of messaging.
- https://www.rabbitmq.com/tutorials/tutorial-one-python

Exercise 3: Demonstrate a Content Delivery Network (CDN)

Design a simple Content Delivery Network (CDN) using Python with focus on distributing content efficiently to users from multiple edge servers

Tech Stack:

- Python programming language
- Flask framework (for building HTTP servers)
- Requests library (for making HTTP requests)
- Create a folder called **content** and store a short video file or an image.

Step 1: Setup Edge Servers

```
# edge_server.py

from flask import Flask, send_file
import os

app = Flask(__name__)

@app.route('/content/<path:path>')
def serve_content(path):
    content_dir = 'content'
    file_path = os.path.join(content_dir, path)
    return send_file(file_path)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Step 2: Load Balancer

```
# load_balancer.py

from flask import Flask, request, redirect
import random

app = Flask(__name__)

edge_servers = ['http://localhost:5000', 'http://localhost:5001',
'http://localhost:5002']
```

```
@app.route('/')
def load_balancer():
    # randomly select one of the servers
    selected_server = random.choice(edge_servers)
    return redirect(selected_server)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

Step 3: Run Edge Servers and Load Balancer

- **Open three terminal windows and run the edge servers:**

```
python edge_server.py
```

- **In another terminal window, run the load balancer:**

```
python load_balancer.py
```

Step 4: Accessing Content through CDN

- Open a web browser and access **`http://localhost:8000/content/image.jpg`**
- Refresh the page multiple times to observe content served from different edge servers.

Sample Results:

- When accessing content through the load balancer, you'll notice that the requests are redirected to different edge servers randomly, demonstrating load balancing.
- Each time you refresh the page, the image file **(`image.jpg`)** will be served from a different edge server, showcasing content distribution.
- You can add more content to the `content` directory and access them through the CDN to observe the distribution of different content files.

This basic setup demonstrates the concept of a Content Delivery Network (CDN) using Python and Flask, focusing on load balancing and content distribution among multiple edge servers.

Exercise 4: Build a AI-driven Customer Sentiment analysis service

Design a simple AI driven Customer Sentiment analysis service using ML models and integrate it into a SOA application.

To build a simple AI-driven service using machine learning models and integrate it into a Service-Oriented Architecture (SOA), you can follow these steps and use the following tools and code snippets:

1. **Objective**: Develop a sentiment analysis service that analyzes customer reviews and provides feedback on product sentiment.
2. **Machine Learning Models**: Use a pre-trained natural language processing (NLP) model for **sentiment analysis**. For this example, we'll use the Hugging Face Transformers library with a pre-trained **BERT** mode **(Bidirectional Encoder Representations from Transformers)**
3. Refer: https://huggingface.co/docs/transformers/en/model_doc/bert
4. Tools:
    - Python3 for coding
    - **Hugging Face Transformers library for NLP models**
    - **Website: https://huggingface.co/docs/transformers/quicktour**
    - Flask for creating the web service
    - Docker for containerization
5. Implementation:
    - Install libraries: `pip3 install transformers flask`
    - Create a Python script for the sentiment analysis service (`sentiment_service.py`)
    - Code: save this code as **sentiment_service.py**

```
from transformers import pipeline
from flask import Flask, request, jsonify

app = Flask(__name__)
nlp = pipeline("sentiment-analysis")

@app.route("/analyze_sentiment", methods=["POST"])
def analyze_sentiment():
    data = request.json
    text = data["text"]
    result = nlp(text)[0]
    return jsonify({"text": text, "sentiment": result["label"],
"confidence": result["score"]})
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

- Above script defines a Flask application with a single endpoint **/analyze_sentiment** that accepts POST requests. When a request is received with a JSON payload containing the text to analyze, the sentiment analysis model is invoked to analyze the text and the result is returned as JSON containing the analyzed text, sentiment label and confidence score. RESTful API endpoint server is accessible via HTTP.
- Create a Dockerfile to containerize the service save it as: **Dockerfile**

```
FROM python:3.9-slim
WORKDIR /app
COPY . .
RUN pip install --no-cache-dir -r requirements.txt
CMD ["python", "sentiment_service.py"]
```

- Build and run the Docker container:

```
docker build -t sentiment-service .
docker run -p 5000:5000 sentiment-service
```

6. **Integration with SOA:**
   - Use 'curl' to invoke sentiments API:

Input: (Good Sentiment)

```
curl -X POST http://localhost:5000/analyze_sentiment \
-H "Content-Type: application/json" \
-d '{"text": "I like this product! It's awesome."}'
```

Output:

```
{
  "text": "I like this product! It's awesome.",
  "sentiment": "POSITIVE",
  "confidence": 0.9998
}
```

Input: (Bad Sentiment)

```
curl -X POST http://localhost:5000/analyze_sentiment \
-H "Content-Type: application/json" \
-d '{"text": "This product is bad! Don’t buy it."}'
```

Output:

```json
{
  "text": "This product is bad! Don’t buy it.",
  "sentiment": "NEGATIVE",
  "confidence": 0.9985
}
```

Exercise 5: Contemporary Trends in SOA

Build a serverless function using a platform like AWS Lambda or Azure Functions.

Create a simple function that performs a specific task (e.g., image resizing, data processing) and expose it as a RESTful endpoint. Integrate this function into an existing SOA architecture to demonstrate its interoperability with other services

- **Objective**: The objective of this lab exercise is to create a serverless function using AWS Lambda that resizes images. The function will be exposed as a RESTful endpoint using AWS API Gateway. Additionally, you will integrate this function into an existing Service-Oriented Architecture (SOA) by making HTTP requests to the API endpoint. This exercise aims to familiarize students with serverless computing, RESTful APIs and integrating services within an SOA.
- **Tools Used**
    - AWS Lambda: A serverless computing service to run code without provisioning or managing servers.
    - AWS API Gateway: A service to create, publish, maintain, monitor and secure APIs.
    - Pillow: A Python Imaging Library (PIL) fork that adds image processing capabilities.
    - Python: The programming language used to write the Lambda function and integration script.
    - Requests Library: A simple HTTP library for Python to make API requests.
- **Prerequisites**
    - AWS Account: Access to **an AWS account** with permissions to create Lambda functions, API Gateway and IAM roles.
    - Note:
        - Try to use your college AWS account, if available, if not, create a free AWS trial account:
            - How to create a free trial AWS Account: https://k21academy.com/amazon-web-services/aws-solutions-architect/create-aws-free-tier-account/
            - !! CAUTION !!
                - ONCE THIS EXERCISE IS COMPLETE, REMEMBER TO DELETE ALL AWS RESOURCES CREATED AS PART OF THIS EXERCISE, ELSE AWS WILL CONTINUE TO “**BILL**” USAGE OF YOUR RESOURCE.
    - Basic Knowledge of Python: Understanding of Python programming, including handling JSON and HTTP requests.

- Basic Knowledge of AWS Services: Familiarity with AWS Lambda and API Gateway.
- Python and Pip Installed: Python 3.x and pip installed on your local machine.

- **Prerequisites**
  - **Step 1: Set Up AWS Lambda Function**
    1. Create the Lambda Function:
       - Log in to the AWS Management Console.
       - Navigate to AWS Lambda.
       - Click "Create function".
       - Choose "Author from scratch".
       - Set Function name: `ImageResizer`.
       - Set Runtime: Python 3.9 (or the latest available).
       - Set Permissions: Create a new role with basic Lambda permissions.
       - Click "Create function".
    2. Write the Lambda Function Code:
       - Install dependencies locally:

```
mkdir lambda_image_resizer
cd lambda_image_resizer
virtualenv venv
source venv/bin/activate
pip install Pillow
mkdir python
cp -r venv/lib/python3.x/site-packages/*
python/
zip -r9 function.zip python
```

    3. Create your function code (`lambda_function.py`):

```
import json
import base64
from io import BytesIO
from PIL import Image

def lambda_handler(event, context):
    try:
        body = json.loads(event['body'])
        image_data = base64.b64decode(body['image'])
        target_width = int(body['width'])
        target_height = int(body['height'])

        image = Image.open(BytesIO(image_data))
```

```
        resized_image = image.resize((target_width,
target_height))

        byte_stream = BytesIO()
        resized_image.save(byte_stream, format='JPEG')
        byte_stream.seek(0)

        resized_image_base64 =
base64.b64encode(byte_stream.read()).decode('utf-8')

        response = {
            'statusCode': 200,
            'body': json.dumps({'resized_image':
resized_image_base64}),
            'headers': {'Content-Type':
'application/json'}
        }
 return response

    except Exception as e:
        return {
            'statusCode': 500,
            'body': json.dumps({'error': str(e)}),
            'headers': {'Content-Type':
'application/json'}
        }
```

4. Add the function code to the ZIP file and then In the AWS Lambda Console, upload the `function.zip` file.

   zip -g function.zip lambda_function.py

- **Step 1: Create API Gateway**
  1. Create a New API:
     - Navigate to API Gateway.
     - Click "Create API".
     - Choose "REST API" and then "Build".
     - **Set API name: `ImageResizerAPI`.**
     - Click "Create API".
  2. Create a Resource and Method:
     - Create a new resource:
       - Click "Actions" and select "Create Resource".
       - Set Resource Name: `images`.
       - Set Resource Path: `/images`.
       - Click "Create Resource".

- Create a POST method:
    - With the **`/images`** resource selected, click "Actions" and select "Create Method".
    - Choose "POST" from the dropdown and click the checkmark.
    - In the Method Execution pane, set the Integration type to Lambda Function.
    - Select the region and enter the name of the Lambda function (`ImageResizer`).
    - Click "Save" and "OK" to give API Gateway permission to invoke your Lambda function.

3. Deploy the API:
    - Click "Actions" and select "Deploy API".
    - Set Deployment stage: Create a new stage called **`lab`**.
    - Click "Deploy".
    - Note the Invoke URL of the deployed API
4. **Step 3: Integration with SOA**
    1. Integrate with an Existing Service:
        - The existing service will call this API by making an HTTP POST request to the API endpoint with the image data and desired dimensions.

```
import requests
import base64
import json

def resize_image(image_path, width, height):
    with open(image_path, 'rb') as
image_file:
        image_data = image_file.read()

    image_base64 =
base64.b64encode(image_data).decode('utf-8')

    payload = {
        'image': image_base64,
        'width': width,
        'height': height
    }

    api_url =
'https://{api-id}.execute-api.{region}.amazo
naws.com/prod/images'
```

```
    response = requests.post(api_url,
data=json.dumps(payload),
headers={'Content-Type':
'application/json'})

    if response.status_code == 200:
        resized_image_base64 =
response.json()['resized_image']
        resized_image_data =
base64.b64decode(resized_image_base64)
        with open('resized_image.jpg', 'wb')
as resized_image_file:

resized_image_file.write(resized_image_data)
        print("Image resized successfully!")
    else:
        print("Failed to resize image:",
response.json()['error'])

resize_image('path/to/your/image.jpg', 100,
100)
```

- **Calling the Lambda Function**
    - **Step-1:** Convert the Image to Base64:
        - Use a tool or a script to encode your image to base64. Here's a simple way to do it using Python:

```
import base64

def encode_image_to_base64(image_path):
    with open(image_path, 'rb') as image_file:
        image_data = image_file.read()
    return
base64.b64encode(image_data).decode('utf-8')

encoded_image =
encode_image_to_base64('path/to/your/image.jpg')
```

            - **Copy the output of this script (the base64 encoded image).**

- **Step-2: Prepare JSON payload**
    - Create a JSON payload file (`payload.json`) with the base64 encoded image and desired width and height.

```
{
    "image": "base64-encoded-image-here",
    "width": 100,
    "height": 100
}
```

- **Step-3: Run curl command**
    - **Use the following `curl` command to make the POST request. Replace `{api-id}`, `{region}` and `base64-encoded-image-here` with your actual API ID, region and base64 encoded image data.**

```
curl -X POST \

https://{api-id}.execute-api.{region}.amazonaws.com/prod/images \
  -H "Content-Type: application/json" \
  -d @payload.json
```

Reference Articles

1. **Why Amazon Retail Went to SOA Architecture**
   https://highscalability.com/why-amazon-retail-went-to-a-service-oriented-architecture/
2. **Hugging Face:**
   https://huggingface.co/docs/transformers/quicktour