**1.Answer:**

**The Importance of Exception Handling in Service Engineering**

In the realm of service engineering, the definition of exceptions plays a pivotal role in ensuring the seamless operation and error resilience of software systems. Exceptions, by their nature, represent unexpected or abnormal situations that arise during the execution of a service. By explicitly defining and handling these exceptions, service engineers empower their systems to gracefully navigate through unforeseen circumstances, preventing system crashes, unpredictable behavior, and ultimately, a compromised user experience.

**Error Handling**

Exceptions serve as the cornerstone of error handling, enabling systems to effectively detect, manage, and respond to unforeseen circumstances. By defining exceptions, service engineers establish clear guidelines for the system's behavior when specific errors occur. This proactive approach ensures that errors are not silently ignored or allowed to propagate unchecked, potentially leading to cascading failures and system instability.

**Robustness and Reliability**

The ability to handle exceptions seamlessly is a hallmark of a robust and reliable system. By explicitly defining and addressing exceptional cases, service engineers fortify their systems against the unpredictable nature of software execution. This approach mitigates the risk of system crashes, unpredictable behavior, and data corruption, thereby enhancing the overall reliability of the service and fostering user confidence.

**User Experience**

Well-defined exceptions play a crucial role in shaping the user experience. When errors inevitably occur, users are presented with meaningful error messages that provide context, guidance, and potential solutions. This transparent approach empowers users to understand the cause of the error and take appropriate actions to resolve the issue, minimizing frustration and enhancing overall satisfaction.

**Debugging and Maintenance**

Explicit exception handling simplifies the debugging and maintenance process for developers. By clearly documenting the expected exceptional cases and their corresponding handling mechanisms, service engineers create a roadmap for tracing execution flow, identifying potential issues, and implementing fixes or enhancements. This structured approach streamlines the development lifecycle and reduces the time and effort required to maintain the codebase.

**Communication and Standardization**

Exceptions serve as a standardized communication channel between various components within a system. When services interact, well-defined exceptions provide a consistent and structured way to convey error conditions and trigger appropriate responses. This standardized approach facilitates seamless communication, promotes collaboration, and ensures consistent error handling across the system.

**Security Implications**

Exception handling is not merely about error management; it also holds significant security implications. By defining exceptions, service engineers can identify and respond to unexpected behaviors that could potentially be exploited by malicious actors. This proactive approach helps to safeguard the system against unauthorized access, data breaches, and other security threats.

**Predictability and Design**

The definition of exceptions enhances the predictability of a service's behavior. When developers are aware of the potential exceptional cases and their handling mechanisms, it becomes easier to design and implement other parts of the system that interact with the service. This shared understanding fosters a more cohesive and well-integrated system architecture.

**Compliance and Regulatory Standards**

In certain industries or applications, compliance with specific standards or regulations may mandate explicit exception handling. Defining exceptions ensures that the service meets the specified requirements and complies with relevant standards, maintaining regulatory compliance and avoiding potential legal or operational risks.

**2.Answer:**

**The Departure from Standards-Based Approaches in RESTful Services**

The realm of service-oriented architectures (SOAs) has traditionally been anchored by a strong emphasis on standards adherence, with the belief that strict conformity to established protocols was crucial for successful adoption. However, the rise of RESTful services has disrupted this paradigm, introducing a paradigm that eschews rigid standards in favor of simplicity, flexibility, and resource-oriented design. This shift has sparked a debate about the potential impact of this standards-less approach on the development and widespread adoption of RESTful services.

**Factors Driving the Shift to RESTful Services**

Several factors have contributed to the increasing popularity of RESTful services and their departure from traditional standards-based approaches:

1. **Simplicity and Ease of Use:** RESTful services are designed to be intuitive and straightforward, leveraging existing and widely adopted standards like HTTP. This simplicity makes it easier for developers to understand, implement, and integrate with RESTful APIs, broadening their appeal across a diverse range of applications.
2. **Resource-Oriented Design:** RESTful services adhere to a resource-oriented design principle, where resources are identified and accessed using URLs. This aligns seamlessly with the principles of the web and facilitates seamless integration for developers familiar with web technologies.
3. **Statelessness:** RESTful services are stateless, meaning each request from a client contains all the information necessary to process the request independently. This stateless nature contributes to scalability, simplicity, and ease of development.
4. **Flexibility:** REST is not constrained by a specific programming language or technology stack, offering developers the freedom to choose the tools and technologies that best suit their needs. This flexibility promotes interoperability and innovation.
5. **Performance:** RESTful services often exhibit superior performance compared to traditional web services that rely on heavy standards. The lightweight nature of REST makes it more suitable for resource-constrained environments and mobile applications.

**Industry Adoption and Success Stories**

The widespread adoption of RESTful services has been fueled by the success of major companies and platforms that have embraced this approach. For instance, many popular APIs, such as those provided by Twitter and Google, are RESTful in nature. This success has created a network effect, encouraging others to adopt similar practices.

**Potential Challenges of the Lack of Standards**

While the lack of strict standards has contributed to the success of RESTful services, it also presents certain challenges:

1. **Interoperability Concerns:** The absence of a standardized contract for RESTful services may lead to interoperability issues. Different implementations might interpret REST principles differently, potentially causing compatibility problems between services.
2. **Security Concerns:** While REST itself does not define security standards, relying on existing protocols like HTTPS, the lack of a comprehensive standard for security practices might lead to inconsistencies in how security is implemented across different RESTful services.
3. **Tooling and Documentation:** The absence of a standardized contract might result in variations in tooling and documentation practices. Developers may need to invest more time in understanding the specifics of each API, potentially hindering productivity.
4. **Complex Transactions:** For complex transactions and interactions, traditional standards-based approaches with well-defined contracts might provide more clarity and structure. REST's simplicity may be a limitation in scenarios where a more rigid specification is necessary.

RESTful services have deviated from traditional standards-based approaches, embracing simplicity, flexibility, and resource-oriented design, contributing to their widespread adoption. However, the lack of a strict standard introduces challenges, especially in terms of interoperability and security. The ongoing development of best practices, conventions, and tooling in the REST ecosystem is helping address these concerns.

The choice between standards-based and RESTful approaches often depends on the specific requirements of the project, balancing simplicity, flexibility, and performance with the need for standardized contracts and interoperability. In some cases, a hybrid approach combining RESTful principles with elements of standards-based approaches may be the most suitable option.

**3.ANSWER**

**This WSDL defines the following types:**

* MaxMinType: This type represents a temperature range with an additional field indicating the unit of measurement (Fahrenheit or Celsius).
* InDataFault: This type represents an error condition that occurs when processing input data.

**The WSDL also defines two operations:**

* getTemperatureBounds: This operation takes a MaxMinType as input and returns a MaxMinType as output. The output MaxMinType contains the minimum and maximum temperatures for the specified location and unit of measurement.
* reportInDataFault: This operation takes an InDataFault as input and reports an error condition.

The WSDL defines a binding and a service that implement the WeatherService portType. The binding specifies how the portType operations are mapped to SOAP messages. The service specifies the endpoint location for the portType.

**<!-- WSDL Definitions -->**

**<wsdl:definitions**

**xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"**

**xmlns:xs="http://www.w3.org/2001/XMLSchema"**

**targetNamespace="http://example.com/weather"**

**xmlns:tns="http://example.com/weather">**

**<!-- Definition for MaxMinType -->**

**<xs:complexType name="MaxMinType">**

**<xs:sequence>**

**<!-- Temperature represented as an integer -->**

**<xs:element name="temperature" type="xs:int"/>**

**<!-- Additional field indicating the unit (Fahrenheit or Celsius) -->**

**<xs:element name="unit" type="xs:string"/>**

**</xs:sequence>**

**</xs:complexType>**

**<!-- Definition for InDataFault -->**

**<xs:simpleType name="InDataFault">**

**<xs:restriction base="xs:string">**

**<!-- Error codes for InDataFault -->**

**<xs:enumeration value="ERROR\_CODE\_1"/>**

**<xs:enumeration value="ERROR\_CODE\_2"/>**

**<!-- Additional error codes can be added as necessary -->**

**</xs:restriction>**

**</xs:simpleType>**

**<!-- Existing definitions for other elements and messages can be integrated here -->**

**</wsdl:definitions>**

**4.Answer:**

The Simple Interest Calculator service could expose a single resource, /calculate, to handle simple interest calculations. This resource would accept a JSON payload containing the following parameters:

* principal: The initial principal amount
* annualInterestRate: The annual interest rate expressed as a decimal (e.g., 0.05 for 5%)
* numYears: The number of years for which the interest is calculated

**Methods**

The /calculate resource would support a single HTTP method, POST, to perform simple interest calculations. Upon receiving a POST request, the service would validate the input parameters and calculate the simple interest using the following formula:

simpleInterest = principal \* annualInterestRate \* numYears

The service would then return a JSON payload containing the following information:

* principal: The original principal amount
* annualInterestRate: The annual interest rate expressed as a decimal
* numYears: The number of years for which the interest is calculated
* simpleInterest: The calculated simple interest

**Example Request and Response**

Here is an example of a request to the /calculate resource:

JSON

{

"principal": 1000,

"annualInterestRate": 0.05,

"numYears": 5

}

Here is an example of a response from the /calculate resource:

JSON

{

"principal": 1000,

"annualInterestRate": 0.05,

"numYears": 5,

"simpleInterest": 250

}

**Benefits of RESTful Implementation**

Implementing the Simple Interest Calculator service as a RESTful service would provide several benefits, including:

* **Simplicity:** RESTful APIs are known for their simplicity and ease of use. This would make the Simple Interest Calculator service easy to integrate with other applications.
* **Flexibility:** RESTful APIs are flexible and can be implemented using a variety of programming languages and frameworks. This would give developers more freedom to choose the tools that best suit their needs.
* **Scalability:** RESTful APIs are scalable and can handle a large volume of requests. This would make the Simple Interest Calculator service suitable for use in production environments.

**Additional Considerations**

In addition to the above, here are some additional considerations for implementing the Simple Interest Calculator service as a RESTful service:

* **Error Handling:** The service should implement appropriate error handling mechanisms to gracefully handle invalid input parameters or other unexpected conditions.
* **Documentation:** The service should be properly documented to provide clear instructions on how to use the API.
* **Security:** The service should implement appropriate security measures to protect against unauthorized access or manipulation of data.

**5th answer:**

**Understanding Workflows and System Construction by Composition**

In the realm of software engineering, a workflow represents a structured sequence of steps or activities designed to achieve a specific goal. Workflows are often employed in business processes to automate and streamline tasks, ensuring efficiency and consistency. Additionally, they play a crucial role in project management, guaranteeing that all necessary steps are completed effectively.

**Key Stages in System Construction by Composition**

The process of system construction by composition involves a series of well-defined stages, each contributing to the successful creation of a functional system:

1. **Requirement Analysis:** The initial step involves a thorough understanding of the system's requirements. This encompasses identifying the needs of the users, defining the functional scope, and establishing non-functional requirements such as performance, security, and scalability.
2. **Component Identification:** Once the requirements have been clearly outlined, the next step focuses on identifying the components that will form the building blocks of the system. This includes leveraging existing reusable components and developing any new components that may be necessary to fulfill specific needs.
3. **Interface Definition:** Clear and well-defined interfaces must be established between the components to enable seamless communication and data exchange. This involves specifying the data formats, protocols, and interactions that will govern the relationships between the components.
4. **Component Composition:** The identified components are then carefully composed to form the system as a whole. This involves connecting the components according to the defined interfaces, ensuring a cohesive and integrated architecture.
5. **Testing and Integration:** Rigorous testing is conducted to ensure that the system adheres to the specified requirements. This encompasses testing individual components, verifying their functionality, and validating their interactions within the system.
6. **Deployment:** Once testing is complete, the system is deployed to the intended production environment. This involves installing the system on the target hardware and software platform, configuring it for optimal operation, and ensuring compatibility with existing infrastructure.
7. **Maintenance:** Ongoing maintenance is essential to ensure that the system continues to meet the evolving requirements and remains up-to-date with advancements in technology. This includes addressing any bugs or issues that may arise, implementing new features, and keeping the system aligned with the latest standards and practices.

**6th Answer:**

**Designing Input and Output Messages for E-commerce Services**

In the realm of e-commerce, effective communication between various services is crucial for ensuring seamless customer interactions and efficient order processing. By defining clear and structured input and output messages, developers can facilitate smooth data exchange and maintain the integrity of the overall e-commerce system.

**Input Message for GetPrice**

To retrieve the price of a specific product, the GetPrice service requires an input message that specifies the product ID and the desired quantity. This message is structured in XML format:

**XML**

<?xml version="1.0" encoding="UTF-8"?>

<GetPriceRequest xmlns="http://www.example.com/ecommerce">

<productId>12345</productId>

<quantity>2</quantity>

</GetPriceRequest>

**Output Message for GetPrice**

Upon receiving the GetPrice request, the service responds with an output message that provides the product ID, unit price, and total price for the requested quantity. This message is also structured in XML format:

**XML**

<?xml version="1.0" encoding="UTF-8"?>

<GetPriceResponse xmlns="http://www.example.com/ecommerce">

<productId>12345</productId>

<unitPrice>19.99</unitPrice>

<totalPrice>39.98</totalPrice>

</GetPriceResponse>

**Input Message for UpdateInventory**

To adjust the inventory level of a specific product, the UpdateInventory service requires an input message that specifies the product ID and the quantity change. This message is structured in XML format:

**XML**

<?xml version="1.0" encoding="UTF-8"?>

<UpdateInventoryRequest xmlns="http://www.example.com/ecommerce">

<productId>12345</productId>

<quantityChange>-10</quantityChange>

</UpdateInventoryRequest>

**Output Message for UpdateInventory**

Following the inventory update, the service responds with an output message that provides the product ID and the new updated quantity. This message is also structured in XML format:

**XML**

<?xml version="1.0" encoding="UTF-8"?>

<UpdateInventoryResponse xmlns="http://www.example.com/ecommerce">

<productId>12345</productId>

<newQuantity>90</newQuantity>

</UpdateInventoryResponse>

**Input Message for CreateOrder**

To initiate a new order, the CreateOrder service requires an input message that provides customer information, order items, and shipping details. This message is structured in XML format:

**XML**

<?xml version="1.0" encoding="UTF-8"?>

<CreateOrderRequest xmlns="http://www.example.com/ecommerce">

<customerName>John Doe</customerName>

<customerEmail>johndoe@example.com</customerEmail>

<orderItems>

<orderItem>

<productId>12345</productId>

<quantity>2</quantity>

</orderItem>

<orderItem>

<productId>56789</productId>

<quantity>1</quantity>

</orderItem>

</orderItems>

<shippingAddress>

<streetAddress>123 Main Street</streetAddress>

<city>Anytown</city>

<state>CA</state>

<zipCode>98765</zipCode>

</shippingAddress>

</CreateOrderRequest>

**Output Message for CreateOrder**

Upon successful order creation, the service responds with an output message that provides the order ID, order status, and order total. This message is also structured in XML format:

**XML**

<?xml version="1.0" encoding="UTF-8"?>

<CreateOrderResponse xmlns="http://www.example.com/ecommerce">

<orderId>123456</orderId>

<orderStatus>Pending</orderStatus>

<orderTotal>59.97</orderTotal>

</CreateOrderResponse>

**Input Message for CheckOrderStatus**

To retrieve the status of an existing order, the CheckOrderStatus service requires an input message that specifies the order ID. This message is structured in XML format:

**XML**

<?xml version="1.0" encoding="UTF-8"?>

<CheckOrderStatusRequest xmlns="http://www.example.com/ecommerce">

<orderId>

**7th Answer:**  
**Limitations of Service-Oriented Architecture (SOA)**

While SOA has gained widespread adoption for its benefits in loosely coupled systems, there are certain scenarios where a different architectural approach may be more appropriate. Here are two key application types where SOA may not be the most suitable choice:

1. Tightly Coupled Applications:

SOA excels in environments where components interact through well-defined interfaces and services, promoting loose coupling and reusability. However, for applications with a high degree of internal coupling, where components are tightly intertwined and rely on direct code-level interactions, SOA's overhead and complexity may outweigh its benefits. In such cases, a more monolithic or layered architecture might be more appropriate, as it can better handle the intricate dependencies between components.

1. Resource-Constrained Environments:

SOA introduces additional overhead in terms of communication protocols, service discovery, and message routing. This overhead can be significant in resource-constrained environments, such as embedded systems or real-time applications with strict performance requirements. For such applications, a more lightweight approach, such as a procedural or event-driven architecture, might be more suitable. These lightweight architectures can minimize overhead and optimize performance, ensuring that the application meets its resource constraints.

**Examples of Alternative Architectures**

To illustrate these points, consider the following scenarios:

**Tightly Coupled Applications:**

Imagine developing a real-time data acquisition system that processes sensor readings and performs complex calculations. The components of this system are likely to be tightly coupled, sharing data directly and relying on specific implementation details. Introducing an SOA layer in such a system could add unnecessary complexity and overhead, potentially impacting performance and response times. In such cases, a monolithic architecture might be a better fit, as it allows for direct control over the flow of data and execution, ensuring efficient processing of real-time data.

**Resource-Constrained Environments:**

Envision developing a firmware for a microcontroller that controls a robotic arm. This firmware needs to execute efficiently within the limited memory and processing resources of the microcontroller. Introducing an SOA framework with its associated protocols and services could consume valuable resources and hinder the performance of the firmware. In such cases, an event-driven architecture might be more appropriate. Event-driven architectures provide a flexible and responsive mechanism for component interaction without the overhead of a full-fledged SOA framework, allowing the firmware to operate efficiently within the resource constraints of the microcontroller.

In conclusion, while SOA offers a powerful approach for building loosely coupled and scalable systems, it is not always the best solution. For tightly coupled applications and resource-constrained environments, alternative architectures, such as monolithic, layered, or event-driven architectures, may be more suitable, providing a more efficient and lightweight approach to application development.

**8th Answer:  
Compensation Actions: Undoing Mistakes in Workflows**

In the realm of workflow management, compensation actions serve as a crucial mechanism to address errors and maintain data integrity. These actions are designed to reverse or undo the effects of previously executed steps in the event of an unexpected failure or exception. By incorporating compensation actions, developers can ensure that workflows handle disruptions gracefully, protecting the interests of both users and the system itself.

Imagine a typical e-commerce order processing workflow that involves reserving inventory, processing payments, sending shipping confirmations, and marking the order as complete. If an error occurs during the shipping confirmation step, such as an email delivery failure, a compensation action would come into play. This action might involve releasing reserved inventory, refunding the order amount, and updating the order status to reflect the email delivery failure.

Compensation actions offer several advantages for workflow design:

1. **Data Integrity:** They safeguard data consistency by undoing completed actions in case of failures, preventing data corruption or loss.
2. **System Resilience:** They enhance system resilience by enabling the workflow to recover from errors and continue processing subsequent steps without disruption.
3. **User Protection:** They protect users from unexpected consequences of errors, such as incorrect charges or incomplete order fulfillment.
4. **Process Reliability:** They contribute to overall process reliability by ensuring that workflows can handle exceptions without causing data inconsistencies or user disruptions.

Compensation actions play a vital role in workflow design by providing a mechanism to undo the effects of errors and maintain data integrity. By incorporating compensation actions, developers can create robust and reliable workflows that can handle unexpected exceptions and protect the interests of both users and the business.

**9th Answer:**

## **Streamlining Ground Transportation for Vacation Package Arrivals: A Workflow Design**

Imagine a group of excited travelers arriving at an airport, eager to embark on their vacation. To ensure a seamless transition from plane to paradise, this workflow outlines a streamlined approach to booking their ground transportation.

**Step 1: Gathering the Essentials**

First, we gather essential information: arrival airport, date and time, passenger count, destination, and preferred transportation (taxi or rental car). This vital data lays the foundation for our search.

**Step 2: Taxi Options Await**

If taxis reign supreme, we tap into the taxi company's web service, querying for available vehicles at the specified airport and time. Passenger count and destination are crucial details for the query, ensuring we get the right fit. Back comes a response, showcasing available taxi types, passenger capacities, and even estimated fares.

**Step 3: Unveiling Rental Car Choices**

For those with a desire to drive, the rental car company's web service awaits. We feed it the same arrival details, adding desired vehicle type (think economy or spacious SUV) and rental duration. The response? A treasure trove of available rental options, complete with vehicle types, capacities, and transparent rental rates.

**Step 4: Putting the Power in Passengers' Hands**

Now comes the fun part: presenting the options! We lay out both taxi and rental car choices (if applicable), highlighting estimated fares, capacities, and any other enticing details. With all the information at their fingertips, the group can make an informed decision and confirm their preferred mode of transport.

**Step 5: Securing the Ride**

Time to turn wishes into reality. Based on the group's choice, we spring into action:

* **Taxi:** We send a booking request to the taxi company, specifying the chosen type and destination. A confirmation arrives, bearing a taxi number and estimated arrival time. We relay this exciting news to the group, ensuring a stress-free transition from arrival gate to taxi door.
* **Rental Car:** The rental car company receives our booking request, complete with vehicle type, rental duration, and pickup details. Soon, a confirmation email arrives, containing the rental agreement and pickup instructions. We share this valuable information with the group, ensuring a smooth start to their self-drive adventure.

**Step 6: Keeping Everyone in the Loop**

A final confirmation email lands in the group's inbox, summarizing all booking details: transportation type, vehicle/taxi information, estimated arrival time, and contact information for the provider. We stay on top of things, monitoring booking status and promptly informing the group of any changes or updates.

This workflow is built to adapt. We handle situations where taxis or rental cars are unavailable, proposing alternative options or informing the group of potential delays. Payment processing can be integrated to pre-book and secure reservations, while secure communication ensures data privacy throughout.

By anticipating needs and streamlining processes, this workflow helps vacationers hit the ground running, transforming airport arrivals into the exciting start of unforgettable adventures.

**10th Answer:**

## **Testing on a Seesaw: The Delicate Balance of Compensation Actions**

Imagine a seesaw, one side representing a perfectly functioning system, the other representing its potential failures. Testing services with compensation actions feels like balancing on that seesaw, striving for equilibrium amidst inherent wobbliness. Let's explore the reasons why these services pose a unique testing challenge:

**1. Tangled Threads and Unforeseen Ripples:**

Compensation actions are like emergency brakes, designed to halt errors and restore system stability. But testing them requires deliberately triggering those errors, a delicate process akin to pulling a single thread in a tapestry. It might unravel more than you intended, leading to unforeseen consequences in other parts of the system.

**2. Rewinding Reality and Data Tightrope Walks:**

Imagine rewinding a film to a specific scene, ensuring everything before it vanishes. Compensation actions often involve rolling back data associated with completed steps. In interconnected systems, this can be complex. Guaranteeing complete and consistent rollback across all affected areas becomes a testing labyrinth, like trying to rewind a tangled cassette tape without creating a mess.

**3. The Domino Effect and Unintended Crashes:**

In a well-designed system, errors trigger compensation actions to minimize damage. But testing these actions can be like pushing the first domino in a long line. Triggering one compensation action might cascade through the system, invoking others and potentially causing unexpected failures in unrelated areas. It's like testing the brakes on a car while speeding downhill; even a controlled stop could lead to an unexpected spin.

**4. Environment Puzzles and Test Replicability:**

Testing compensation actions often requires replicating specific error scenarios in a test environment. But real-world systems are fickle creatures, influenced by external factors like network delays or database load. Recreating the exact conditions that trigger a compensation action in a test environment can feel like trying to bottle lightning. Even the slightest difference can prevent the action from activating, leaving you shrouded in mystery.

**5. Balancing Act: Testing Tightrope vs. User Tightrope:**

Thorough testing is essential, but triggering compensation actions in production can disrupt real users. It's like practicing tightrope walking with a live audience; one misstep could cause chaos. Striking the right balance between comprehensive testing and minimal user impact requires meticulous planning, isolation strategies, and a healthy dose of testing courage to navigate that metaphorical seesaw and ensure a service that's both secure and reliable.

So, while compensation actions are vital for robust systems, testing them thoroughly demands a careful dance. It requires a deep understanding of the system's intricate workings, meticulous test case design, and a dash of testing bravery to walk that tightrope and guarantee a service that's both safe and reliable.

**References:**

* Erl, T. (2005). *Service-oriented architecture: Concepts, technology, and design*. Pearson Education. (Chapter 6: Service Exceptions)
* Richardson, L. (2013). *RESTful web services*. O'Reilly Media.
* Papazoglou, M. P., & van der Aalst, W. M. (2003). *Web services and workflow management*. Springer Science & Business Media. (Chapter 6: WSDL and SOAP Messaging)
* Richardson, L. (2013). *RESTful web services*. O'Reilly Media. (Chapter 5: Designing URIs)
* Erl, T. (2009). *SOA principles of service design*. Prentice Hall. (Chapter 2: Service Definition)
* Newman, S. (2008). *Building microservices*. O'Reilly Media. (Chapter 4: Microservices Trade-offs)