Christoffer Perch Nielsen & Jonathan Gjetting Braad

System Integrations - OLA 2

Q1 – *What are the main differences between Enterprise and Solution Architecture? Describe the different roles that these play in designing large-scale systems.*

- **Enterprise Architecture (EA)** provides an broad framework that aligns IT infrastructure with the business strategy. It focuses on the big picture, ensuring that all systems across the organization work together cohesively to achieve long-term business objectives.

- **Solution Architecture (SA)** focuses on specific projects or systems within the enterprise framework. It deals with the technical design and implementation of solutions that address individual business needs, ensuring they integrate with existing systems.

Differences:

- **EA** is more broad, addressing the entire organization, while **SA** is more focused on specific technical solutions.
- **EA** ensures alignment with business goals, while **SA** is responsible for executing specific implementations within that larger structure.

Q2 – *In his video called "Practical Architecture," what does Stefan Tilkov say about the role of teams in modern architecture? He refers to a book, "Team Topologies" – what "team topologies" does the book describe? Do you agree from your own experience that the team is a core part of a successful project?*

- **Stefan Tilkov** emphasizes that teams are central to successful modern architecture. He highlights how aligning the architecture with team structures leads to better outcomes because small, autonomous teams can make decisions faster and adapt to changes more effectively.

- The book **"Team Topologies"** describes four main team structures:
    1. **Stream-Aligned Teams**: Focused on delivering a continuous stream of business value.

2. **Platform Teams**: Provide internal services that support other teams.
3. **Complicated Subsystem Teams**: Handle specialized, complex parts of the system.
4. **Enabling Teams**: Help other teams by providing expertise in specific areas.

- From our experience, smaller teams often lead to more efficient development, better communication, and quicker problem solving. The bigger the teams, the more time is wasted on communication and understanding each other. Dividing projects into smaller teams also allows for each individual group to not have their focus spread across too many different objectives, which can lead to better focus on the tasks.

Q3 – *In the video "Practical Architecture," Stefan Tilkov explains why some things are best done centrally (at 23 min 30 seconds). Why do you think he is saying this? What does he claim are the benefits?*

- **Stefan Tilkov** argues that some functions are better handled centrally because they provide a **shared service** across multiple teams or systems. Centralized components, such as authentication, logging, and monitoring, prevent teams from working on something that has already been done, and ensures consistency across the organization.

- The key benefits he highlights are:
    1. **Consistency**: Centralized services enforce uniform standards and practices.
    2. **Efficiency**: Teams avoid reinventing the wheel by using shared services.
    3. **Scalability**: Centralizing common functions makes it easier to scale systems and ensures that crucial parts of the infrastructure are reliable.

This centralization allows for a balance between autonomy and standardization in distributed systems.

Q4 – *In the video "Architecting for Outcomes," Simon Rohrer describes old-fashioned and modern enterprise architecture. He talks about the ABCDE of modern architecture to compare modern and legacy ways of working. Do you find his arguments persuasive and if so why?*

- **Simon Rohrer** explains the **ABCDE** framework for modern enterprise architecture:
    1. **A**utonomous Teams
    2. **B**usiness-Aligned Architecture
    3. **C**ross-Functional Teams
    4. **D**ecoupled Systems

5. **E**vent-Driven Architecture

- Compared to legacy systems, this approach allows organizations to be more **agile**, **scalable**, and **responsive** to business changes. We find his arguments persuasive because modern architecture encourages decentralized decision-making and faster development cycles, both of which are critical for maintaining competitiveness in today's fast-paced environments.

Q5 – *In the same video, Simon Rohrer explains the concept of the "Continuous Conversation." In what ways does he say it benefits Saxo Bank? How does he connect the Continuous Conversation to the DevOps Infinity Loop?*

- **Continuous Conversation** refers to the repeating feedback sessions between development teams, operations, and business stakeholders, ensuring that the project/architecture evolves in response to real-time needs.

- Saxo Bank benefits from this approach as it keeps the development in line with business goals and customer demands, leading to faster iterations and better alignment between product delivery and business outcomes.

- Rohrer connects this concept to the **DevOps Infinity Loop** by explaining how Continuous Conversation integrates feedback into every phase of development, from planning through to operations. This ensures that feedback is rapidly acted upon, promoting agile development and continuous improvement in the development process.

Q6 – *In the course, we focus on building integrated enterprise-scale applications using messaging, and there are a number of core integration patterns for messaging. Using the PowerPoint presentation for this week, describe 5 integration patterns for messaging and provide a use case for them. Include Pipes and Filters and the Request-Reply patterns.*

1. **Pipes and Filters**:
    - **Pattern**: Breaks a process into smaller steps (filters), with data passed between them via channels (pipes).
    - **Use Case**: Data transformation in a message pipeline where each filter applies a specific transformation.
2. **Request-Reply**:

    - **Pattern**: A sender sends a request and waits for a response from the receiver.
    - **Use Case**: Querying a service for specific information, such as retrieving a customer profile from a database.
3. **Message Broker**:
    - **Pattern**: A mediator handles the communication between multiple services, ensuring messages are delivered to the correct destination.
    - **Use Case**: A centralized broker like RabbitMQ managing messages between services in a microservices architecture.
4. **Publish-Subscribe**:
    - **Pattern**: A sender publishes messages that multiple subscribers can consume.
    - **Use Case**: Event notifications, where multiple systems need to be notified when a particular event occurs.
5. **Message Router**:
    - **Pattern**: Directs messages to the correct channel based on specific criteria.
    - **Use Case**: Routing messages to different microservices based on message content, such as customer service requests being routed based on priority.

Q7 – *In Gregor Hohpe's talk "Enterprise Integration Patterns 2," he describes the idea of conversations in a messaging architecture (from 18 minutes). What differences are there between messaging and conversation architectures, and what challenges does he describe for conversation-based solutions (for example, what is the difference between pub-sub and subscribe-notify)?*

- **Messaging architecture** focuses on simple, one-way communication where messages are sent without expecting immediate responses. It's more suited for systems that need to exchange data asynchronously.
- **Conversation architecture** is a series of exchanges (a conversation) between systems, where each step may depend on the previous one, requiring more coordination and management.
- **Challenges of Conversation-Based Solutions**:
    - Increased **complexity** in managing state and dependencies between messages.
    - **Timing**: Conversations need to manage when and how messages are sent and received, requiring more effort with handling.
- **Difference between Pub-Sub and Subscribe-Notify**:
    - **Pub-Sub**: A publisher sends messages to many subscribers at once, without targeting individual receivers.
    - **Subscribe-Notify**: Subscribers register for specific notifications, and they are notified when a relevant event occurs.

Q8 – *In Gregor Hohpe's video, he explains the history and role of the idea of Patterns and Pattern Languages. How would you summarize what he explains – what do you see as the core requirements for patterns to be useful?*

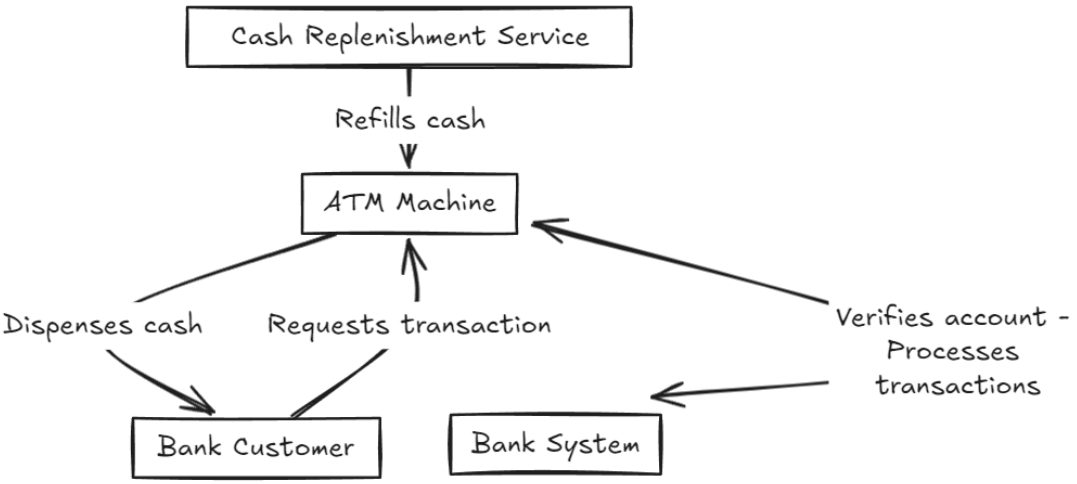
- **Summary of Patterns and Pattern Languages**:
  Gregor Hohpe explains that patterns are repeatable solutions to common problems in software architecture. They provide a structured approach to solving recurring challenges by offering proven solutions, making system design more efficient and reliable.
- **Core Requirements for Useful Patterns**:
  1. **Reusability**: Patterns must solve a problem that recurs in different contexts.
  2. **Simplicity**: They should be easy to understand and apply, providing clear benefits without overcomplicating the solution.
  3. **Flexibility**: Patterns should be adaptable to various specific needs and contexts.
  4. **Scalability**: Patterns must be applicable to both small and large systems.

Q9 – *In Simon Rohrer's article on modern enterprise architecture, he discusses the 'strangler' pattern for moving away from legacy systems. Why is this seen as a complex problem, and what does the strangler pattern do to help?*

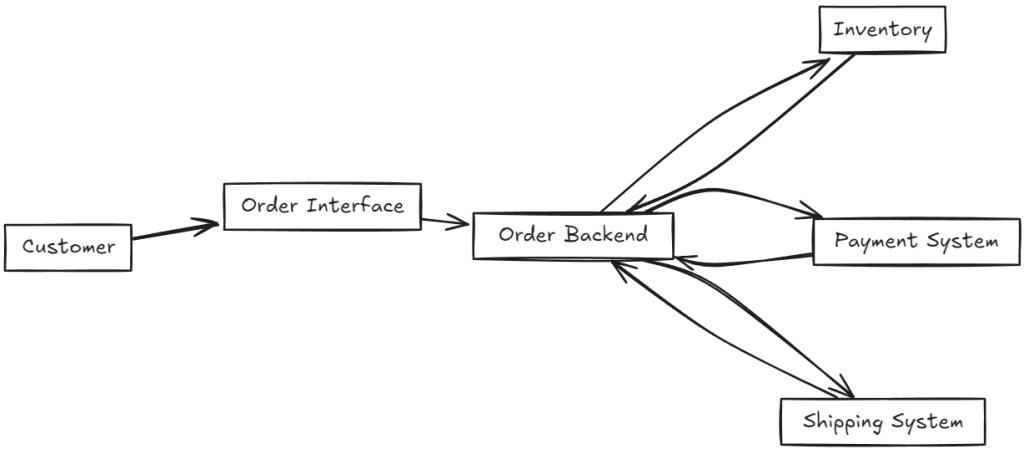
- **Complexity of Moving Away from Legacy Systems**:
  Legacy systems are deeply integrated into an organization, making them difficult to replace without disrupting operations. The challenge lies in gradually transitioning from old systems while maintaining functionality.
- **Strangler Pattern**:
  The **strangler pattern** addresses this by incrementally replacing parts of the legacy system with new functionality. Over time, the new system "strangles" the old one, allowing for a smoother transition without requiring a full system rewrite all at once.

Q10 – *In Jesper Lowgren's video "Solution vs Enterprise Architecture Tutorial," he describes three core diagrams that can be used to describe an architecture. What are they, and what role do they play? Include examples.*

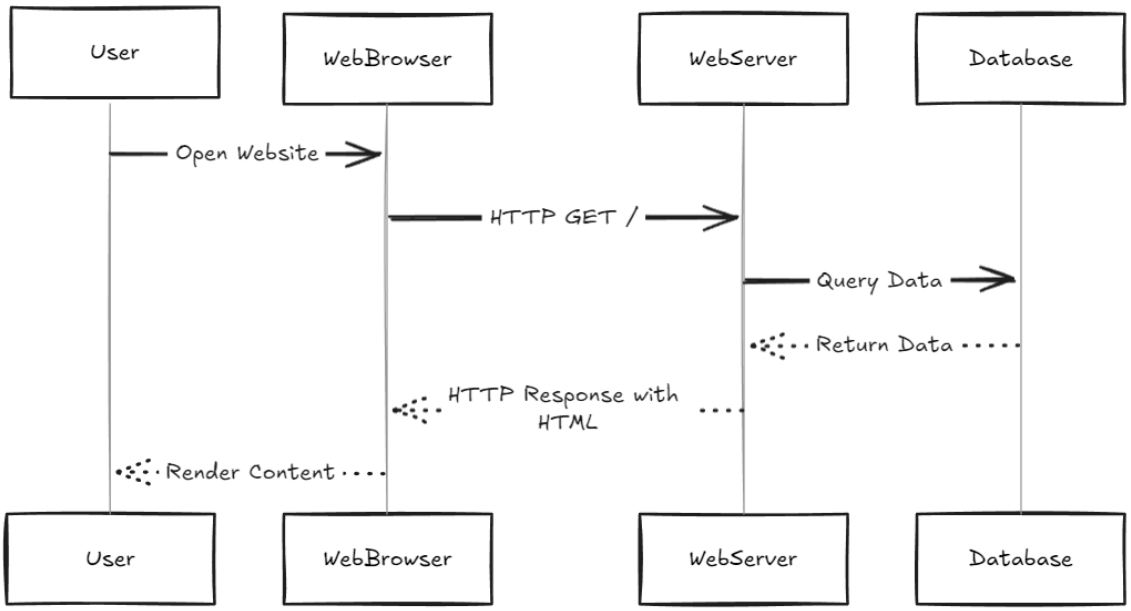
- **Core Diagrams**:
  1. **Context Diagram**: Shows how the system interacts with external entities.
     - **Example**: A diagram showing how a payment atm system:



2. **Component Diagram**: Illustrates the internal structure of a system, showing its components and how they interact.
    - **Example**: A diagram displaying a customer ordering a product



3. **Sequence Diagram**: Focuses on how components interact over time, detailing the sequence of messages or operations.
    - **Example**: A sequence diagram showing the interaction between a user, web server, and database



- **What role do they play?**: These diagrams help teams visualize the system architecture at different levels of detail, aiding in communication, design decisions, and ensuring that all team members understand the system's structure.