Übung 2

In RabbitMQ ermöglichen Routing-Keys die gezielte Zustellung von Nachrichten an Queues über Exchanges.

**Direct Exchange:** Nachrichten werden an Queues weitergeleitet, deren Binding-Key exakt mit dem Routing-Key übereinstimmt.  
*Beispiel:* Eine Direct Exchange logs\_direct verteilt Nachrichten mit error, info oder warning an entsprechende Queues.

**Topic Exchange:** Nachrichten werden basierend auf Musterabgleichen zwischen Routing-Keys und Binding-Keys verteilt. Platzhalter wie \* (ein Wort) und # (null oder mehr Wörter) erlauben komplexere Routing-Szenarien.  
*Beispiel:* Eine Topic Exchange logs\_topic verteilt kern.\* an Queues für Kernel-Logs und \*.critical an Queues für kritische Logs.

**Anwendungsbeispiel:** In einem Logging-System können Topic Exchanges verwendet werden, um Nachrichten nach Ursprung (auth, kernel) und Schwere (info, error) zu sortieren, was die Analyse erleichtert.

Übung 3

Das PACELC-Theorem erweitert das CAP-Theorem, indem es nicht nur die Trade-offs bei Netzwerkpartitionen (Partition Tolerance, Availability, Consistency) betrachtet, sondern auch die Abwägungen zwischen Latenz (Latency) und Konsistenz (Consistency) in partitionstoleranten Systemen ohne aktuelle Partitionen einbezieht.

[Wikipedia](https://en.wikipedia.org/wiki/PACELC_theorem?utm_source=chatgpt.com)

**Typisches Einsatzgebiet:** Das PACELC-Theorem wird in der verteilten Systemarchitektur verwendet, um die Kompromisse zwischen Konsistenz, Verfügbarkeit und Latenz sowohl während als auch außerhalb von Netzwerkpartitionen zu analysieren. Es hilft Entwicklern und Architekten, fundierte Entscheidungen bei der Gestaltung von verteilten Datenbanken und Systemen zu treffen, indem es die Auswirkungen dieser Kompromisse auf die Systemleistung und -zuverlässigkeit verdeutlicht.

[Dremio](https://www.dremio.com/wiki/pacelc-theorem/?utm_source=chatgpt.com)

**Vier mögliche Konfigurationsoptionen:**

1. **PA/EL (Partition Tolerance & Availability / Else Latency):**
   * **Bei Partitionierung:** Das System priorisiert Verfügbarkeit und Partitionstoleranz, was zu Lasten der Konsistenz gehen kann.
   * **Ohne Partitionierung:** Das System legt Wert auf geringe Latenz, möglicherweise auf Kosten der Konsistenz.
2. **PA/EC (Partition Tolerance & Availability / Else Consistency):**
   * **Bei Partitionierung:** Verfügbarkeit und Partitionstoleranz werden priorisiert, wodurch die Konsistenz beeinträchtigt werden kann.
   * **Ohne Partitionierung:** Das System priorisiert Konsistenz über Latenz.
3. **PC/EL (Partition Tolerance & Consistency / Else Latency):**
   * **Bei Partitionierung:** Konsistenz und Partitionstoleranz werden priorisiert, was die Verfügbarkeit beeinträchtigen kann.
   * **Ohne Partitionierung:** Das System legt Wert auf geringe Latenz, möglicherweise auf Kosten der Konsistenz.
4. **PC/EC (Partition Tolerance & Consistency / Else Consistency):**
   * **Bei Partitionierung:** Konsistenz und Partitionstoleranz werden priorisiert, was die Verfügbarkeit beeinträchtigen kann.
   * **Ohne Partitionierung:** Das System priorisiert weiterhin Konsistenz, unabhängig von der Latenz.

Diese Konfigurationsoptionen verdeutlichen die unterschiedlichen Ansätze, die bei der Gestaltung verteilter Systeme in Bezug auf Konsistenz, Verfügbarkeit und Latenz berücksichtigt werden müssen.

Übung 4

Das SAGA-Pattern ermöglicht verteilte Transaktionen mit schlussendlicher Konsistenz. Es gibt zwei Ansätze:

**Choreografie:**  
Dienste steuern sich selbst durch Events.

* **Vorteile:** Lose Kopplung, kein zentraler Ausfallpunkt.
* **Nachteile:** Schwieriges Monitoring, komplexe Fehlerbehandlung.

**Orchestrierung:**  
Ein zentraler Orchestrator koordiniert die Abläufe.

* **Vorteile:** Klare Kontrolle, einfache Kompensation.
* **Nachteile:** Single Point of Failure, engere Kopplung.

**Komplexität:**  
Choreografie verteilt die Logik auf Dienste, erschwert jedoch die Nachvollziehbarkeit. Orchestrierung vereinfacht die Steuerung, erhöht aber die Abhängigkeit vom Koordinator.

Die Wahl hängt von Anforderungen wie Flexibilität oder Kontrollbedarf ab.

Übung 5

**Kriterien zur Auswahl des geeigneten Architekturmusters**

1. **Skalierbarkeit:**
   * **Monolith:** Gut geeignet für Anwendungen mit geringer bis moderater Skalierungsanforderung. Skalierung erfolgt durch Hochskalierung (vertikale Skalierung).
   * **Modularer Monolith:** Bietet eine bessere Skalierung als ein reiner Monolith durch klar definierte Module. Vertikale Skalierung bleibt dominant.
   * **Microservices:** Ideal für hohe Skalierungsanforderungen, da einzelne Services unabhängig horizontal skaliert werden können.
2. **Teamgröße:**
   * **Monolith:** Optimal für kleine Teams, da weniger Koordinationsaufwand erforderlich ist.
   * **Modularer Monolith:** Geeignet für mittelgroße Teams, die an verschiedenen Modulen arbeiten können.
   * **Microservices:** Erfordert größere, spezialisierte Teams, die unabhängig an verschiedenen Services arbeiten.
3. **Komplexität:**
   * **Monolith:** Einfach zu entwickeln, zu testen und bereitzustellen. Geringe Komplexität, aber eingeschränkte Flexibilität.
   * **Modularer Monolith:** Erhöhte Komplexität durch Modularisierung, aber weiterhin einfacher als Microservices.
   * **Microservices:** Höchste Komplexität durch verteilte Systeme, Kommunikation zwischen Diensten und Deployment-Strategien.
4. **Technische Rahmenbedingungen:**
   * **Monolith:** Benötigt keine fortschrittlichen Tools oder Infrastrukturen.
   * **Modularer Monolith:** Erfordert gute Architekturprinzipien und klare Modulgrenzen.
   * **Microservices:** Setzt DevOps-Kultur, Containerisierung (z. B. Docker), Orchestrierung (z. B. Kubernetes) und Monitoring-Tools voraus.
5. **Änderungs- und Erweiterungsbedarf:**
   * **Monolith:** Weniger flexibel bei Änderungen; Änderungen können das gesamte System betreffen.
   * **Modularer Monolith:** Flexibler als ein Monolith, Änderungen können auf Module beschränkt werden.
   * **Microservices:** Höchste Flexibilität, da Services unabhängig aktualisiert oder ersetzt werden können.
6. **Time-to-Market:**
   * **Monolith:** Schnelle Entwicklungs- und Bereitstellungszeit für kleinere Projekte.
   * **Modularer Monolith:** Etwas längere Entwicklungszeit durch Modularisierung, aber bessere Wartbarkeit.
   * **Microservices:** Längere Entwicklungszeit durch erhöhte Komplexität.
7. **Kosten:**
   * **Monolith:** Geringste Kosten für Entwicklung und Infrastruktur.
   * **Modularer Monolith:** Moderate Kosten durch erhöhte Entwicklungsaufwände.
   * **Microservices:** Höchste Kosten durch komplexe Infrastruktur und spezialisierte Teams.

**Beispielanwendungen**

* **Monolith:** Eine kleine E-Commerce-Website mit begrenzten Benutzerzahlen und Funktionen.
* **Modularer Monolith:** Ein mittelgroßes CRM-System, bei dem verschiedene Teams Module wie Kundenverwaltung und Reporting entwickeln.
* **Microservices:** Eine globale Streaming-Plattform (z. B. Netflix), die hohe Skalierbarkeit und unabhängige Serviceentwicklung erfordert.