**Handelsplattform 1**

|  |  |
| --- | --- |
| **Vorname** | **Nachname** |
| **Mariam** | **Abdullah** |
| **Yasin** | **Aktas** |
| **Adam** | **Dolgos** |
| **Akram** | **El-Kormany** |
| **Christopher** | **Heyda** |
| **David** | **Hitthaler** |
| **Bartosz** | **Kujat** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | 1. **Analyse: Machen Sie sich mit dem Ausgangs-Source-Code „SolHandelsplattform V.01“ vertraut. Publizieren Sie die beiden Services „BlackFriday“ und „IEGEasyCreditCardService“ in die Microsoft Azure Cloud und Testen Sie die Funktionalität (0 Punkte)** | |  | | 1. **2 weitere Microservice Produktkataloge: Erstellen Sie ein Microservice, welches eine Liste von Produkten anbietet. Der Inhalt der Liste soll dabei aus einem „microservice local datastore“ kommen – (Decentralized Data Management). Ersetzen Sie die hard codierten Werte im BlackFriday/ProductList-Controller durch den Aufruf des soeben erstellen Services. Ein weiterer Produktkatalog-Service soll Produkte aus einem Text File auf einem FTP-Server auslesen und zur Verfügung stellen. (10 Punkte):** | |  | | 1. **Skalierung, Ausfallssicherheit und Logging (Design for failure) für CreditPaymentService. Detailsbeschreibung: Publizieren Sie das Service „IEGEasyCreditCardService“ mehrfach in die Cloud und rufen Sie die Services im „Round Robin“ Stil auf. Falls es beim Aufruf eines Service zu einem Fehler kommt, soll es eine Retry-Logik geben, außerdem soll der aufgetretene Fehler mit Hilfe eines zentralen Logging-Service protokolliert werden. Nach n erfolglosen Versuchen, soll das nächste Service aufgerufen werden. (10 Punkte)** | |  | | 1. **(theoretische) Überlegungen zum Einsatz von Asynchronen Kommunikationsstilen in der Handelsplattform (10 Punkte)** <http://soapatterns.org/design_patterns/reliable_messaging><http://soapatterns.org/design_patterns/event_driven_messaging> **http://soapatterns.org/design\_patterns/asynchronous\_queuing** | | **Reliable Messaging**  Garantierte Servicekommunikation kann nur dann ermöglicht werden, wenn nicht „reliable Messaging Protokolle“ verwendet werden. Um eine garantierte Kommunikation zu ermöglichen, wird die Architektur angepasst. Es werden Middleware, Service Agents und Datenspeicher eingesetzt. Die Nachrichtenlieferungen werden verfolgt, die Ausgabe von Bestätigungen wird verwaltet, um während der Fehlerbedingungen diese einzuspeichern.  Ablauf der Kommunikation:   1. Der Sender übergibt seine Nachricht an seinen Vermittler. 2. Dieser Vermittler stellt eine Verbindung zum Vermittler auf Empfängerseite her. 3. Die Nachricht wird von Vermittler zu Vermittler übertragen. 4. Der Nachrichteneingang wird vom Empfängervermittler bestätigt. 5. Der Vermittler auf Empfängerseite liefert die Nachricht an den eigentlichen Empfänger aus.   Durch diesen Ablauf ist sichergestellt, dass a) der Empfänger die Nachricht wirklich bekommen hat und b) der Sender auch darüber Bescheid weiß.    **Event-Driven-Messaging**  Wenn wir Runtime Service Ereignisse haben und automatisch benachrichtigt werden wollen, dann hilft hier ein Messaging Framework, das das Publish-and-Subscribe-MEP und die damit verbundene komplexe Ereignisverarbeitung und -verfolgung unterstützen kann. Das ereignisgesteuerte Messaging ist ein Entwurfsmuster, das innerhalb des Designparadigmas für Serviceorientierungen angewendet wird, um Serviceanwendern, die an Ereignissen innerhalb der Peripherie eines Serviceanbieters interessiert sind, Benachrichtigungen zu diesen Ereignissen zu geben. Das ereignisgesteuerte Messaging ist ein Entwurfsmuster, das innerhalb des Designparadigmas für Serviceorientierungen angewendet wird, um Serviceanwendern, die an Ereignissen innerhalb der Peripherie eines Serviceanbieters interessiert sind, Benachrichtigungen zu diesen Ereignissen zu geben.  Eine Event-Driven Architecture besteht aus einer Instanz, die die Ereignisse erzeugt, und einer weiteren, die sie verbraucht. Die Ereignis-erzeugende Instanz ist der Event-Creator, die verbrauchende der Event-Consumer. Während der Creator für die Generierung des Ereignisses sorgt, ist es für den Consumer wichtig zu wissen ob das Ereignis erzeugt wurde, es in die Prozesse involviert ist und diese beeinflusst. Der Entscheidungsalgorithmus für die Ereignis-basierten Aufgaben ist: If This Then That (IFTTT).  In Ereignis-gesteuerten Architekturen können auftretende Ereignisse mittels Event Stream Processing (ESP) nahezu in Echtzeit analysiert und die Informationen in die verteilten Umgebungen weitergeleitet werden. Durch die niedrige Latenz können Organisationen und Unternehmen schnell auf jede Geschäftsaktivität reagieren und dadurch traditionelle Techniken der Datenintegration wie bei der Replikation von Daten verbessern. Die Modellierung von Geschäftsprozessen in diskrete Zustandsänderungen bietet eine höhere Flexibilität in Bezug auf sich ändernde Bedingungen.  Eine Event-Driven Architecture kann eine Software Oriented Architecture (SOA) insofern ergänzen, weil die die Services der SOA-Architektur durch die Ereignisse ausgelöst werden können.  **Asynchronous Queuing**  Wenn eine Servicefähigkeit erfordert, dass Verbraucher synchron mit ihr interagieren, kann dies die Leistung beeinträchtigen und die Zuverlässigkeit beeinträchtigen.  Ein Dienst kann Nachrichten mit seinen Verbrauchern über einen Zwischenpuffer austauschen, so dass Dienst und Verbraucher Nachrichten unabhängig verarbeiten können, indem sie zeitlich entkoppelt bleiben.  Asynchrones Messaging ist ein Kommunikationsverfahren, bei dem das System eine Nachricht in eine Nachrichtenwarteschlange einfügt und keine sofortige Antwort benötigt, um mit der Verarbeitung fortzufahren. Beispiele umfassen eine Anfrage nach Informationen, Erklärungen oder Daten, die benötigt werden, aber nicht sofort benötigt werden.  Anwendung: Warteschlangen-Technologie muss in die umgebende Architektur integriert werden, und Backup-Speicher können ebenfalls erforderlich sein. | |  | | 1. **Schreiben Sie ein zusätzliches „Paymentservice“. Dieses Payment-Service soll sowohl JSON, XML-Nachrichten als auch Nachrichten im Format CSV verarbeiten und erzeugen können. Orientieren Sie sich an dem Pattern - http://soapatterns.org/design\_patterns/content\_negotiation  (10 Punkte)** | |  | |  |  |  | | --- | | 1. **(theoretische) Überlegungen zu einem PaymentService-Broker. Dieses Service soll zwischen Shops und Payment-Services „vermitteln“.** <http://serviceorientation.com/serviceorientation/service_loose_coupling><http://serviceorientation.com/serviceorientation/service_discoverability> **http://soapatterns.org/design\_patterns/data\_format\_transformation**   **http://www.enterpriseintegrationpatterns.com/patterns/messaging/CanonicalDataModel.html**  **http://soapatterns.org/design\_patterns/canonical\_schema**  **http://soapatterns.org/design\_patterns/canonical\_protocol  (10 Punkte)** | |  | |  |   Es handelt sich dabei um Software Architecture Pattence. Das Ziel des Broker Pattence ist eine möglichst effiziente Kommunikation zwischen den Services, diese voneinander zu entkoppeln. In diesem konkreten Beispiel wäre jeder Shop ein Frontend, mit dem User interagieren können. Zur Abwicklung des Payment-Prozesses werden eigene Services verwendet, die beispielsweise über Netzwerke verbunden sind. Jedes Payment-Service registriert sich beim Broker und stellt ihm ein definiertes Interface zur Verfügung. Bei jeder Anfrage für die Abwicklung eines Payment-Prozesses an den Broker, wählt dieser den geeigneten Payment-Service aus der Liste der registrierten Services. Wenn ein oder mehrere Services ausfallen, werden die anderen nicht durch den Ausfall beeinträchtigt. Auch wenn man bestimmte Services austauschen/löschen/updaten möchte, hat dies keinen Einfluss auf die anderen Services, noch auf den Broker oder die Webshops.   |  | | --- | | 1. **Webhook-Subscriber: Überlegen und implementieren Sie ein mögliches Webhook-Szenario** | |  | |  | |