**FH JOANNEUM - University of Applied Sciences**

**IST Analyse**

**Dokumentation**

**Eingereicht für die Lehrveranstaltung “E-Business Anwendungen”**

**Autoren:**

**Florian Reinprecht, BSc**

**Elisabeth Fellner, BSc**

**Labinot Jashanica, BSc**

**Kristian Ndou, BSc**

**Yannick Collasius, BSc**

**Supervisor:**

**FH-Prof. Dipl.-Ing. Dr. techn. Erwin Zinser**

**FH-Prof. Mag. Dr. Robert Singer**

**Graz, 2019**



Inhaltsverzeichnis

[1 Microservices 3](#_Toc10911354)

[1.1 GUI 3](#_Toc10911355)

[1.2 ConfigurationService 4](#_Toc10911356)

[1.3 Gateway 5](#_Toc10911357)

[1.4 ServiceDiscovery 6](#_Toc10911358)

[1.5 ExternalCommunicator 7](#_Toc10911359)

[1.6 ProcessModelStorage 7](#_Toc10911360)

[1.7 ProcessEngine 8](#_Toc10911361)

[1.8 Eventlogger 9](#_Toc10911362)

[1.9 ProcessStore 9](#_Toc10911363)

[1.10 Modeling Plattform 9](#_Toc10911364)

[2 ArchiMate Ist-Analyse Modellierung 10](#_Toc10911365)

[2.1 Business- und Strategy-Layer 12](#_Toc10911366)

[2.2 Application Layer 13](#_Toc10911367)

[2.3 Technology- und Physical-Layer 15](#_Toc10911368)

[3 Workflow Management Coalition 17](#_Toc10911369)

[3.1 EasyBiz vs WfMC 17](#_Toc10911370)

[3.1.1 Workflow Enactment Service 17](#_Toc10911371)

[3.1.2 Process Definition 18](#_Toc10911372)

[3.1.3 Client Anwendung (GUI) 18](#_Toc10911373)

[3.1.4 Invoking and interacting with External Application 18](#_Toc10911374)

[3.1.5 Interface für andere Workflow Engines 18](#_Toc10911375)

[3.1.6 Administration and Monitoring 18](#_Toc10911376)

[4 Twelve-Factor App Vergleich 20](#_Toc10911377)

[4.1.1 Twelve-Factor App Grundlagen 20](#_Toc10911378)

[4.1.2 Vergleich mit bestehender Software 21](#_Toc10911379)

[4.1.3 Zusammenfassung & Verbesserungen 30](#_Toc10911380)

[5 Cloud Design Patterns 31](#_Toc10911381)

[5.1 Availability 31](#_Toc10911382)

[5.2 Data Management 31](#_Toc10911383)

[5.3 Design and Implementation 33](#_Toc10911384)

[5.4 Messaging Patterns 36](#_Toc10911385)

[5.5 Management and Monitoring Patterns 36](#_Toc10911386)

[5.6 Performance and Scalability Patterns 36](#_Toc10911387)

[5.7 Resilliency Patterns 37](#_Toc10911388)

[5.8 Security Patterns 37](#_Toc10911389)

[6 SOLL-Szenarien Cloud Migration 39](#_Toc10911390)

[6.1 Managed Service 39](#_Toc10911391)

[6.2 Container Instances 39](#_Toc10911392)

[7 IST-Szenarien Cloud Migration 40](#_Toc10911393)

[7.1 Lift & Shift Migration 40](#_Toc10911394)

[7.2 Azure for Docker Migration 41](#_Toc10911395)

Abbildungsverzeichnis

[Abbildung 1 - EasyBiz Microservice Architektur 3](#_Toc10911396)

[Abbildung 2: Configuration Service 4](#_Toc10911397)

[Abbildung 3: Gateway 6](#_Toc10911398)

[Abbildung 4: Service Discovery 6](#_Toc10911399)

[Abbildung 5: External Communicator 7](#_Toc10911400)

[Abbildung 6: Process Model Storage 8](#_Toc10911401)

[Abbildung 7: Process Engine 8](#_Toc10911402)

[Abbildung 8: ArchiMate - Ist-Architektur 11](#_Toc10911403)

[Abbildung 10: ArchiMate - Business- und Strategy Layer 12](#_Toc10911404)

[Abbildung 11: ArchiMate - Application Layer 13](#_Toc10911405)

[Abbildung 12: ArchiMate – Technology- und Physical-Layer 15](#_Toc10911406)

[Abbildung 13: Workflow Enactment Service 19](#_Toc10911407)

[Abbildung 14: 12 Factor - Codebase 22](#_Toc10911408)

[Abbildung 15: 12 Factor - Unterstützende Dienste 24](#_Toc10911409)

[Abbildung 16: 12 Factor - Build, release, run 25](#_Toc10911410)

[Abbildung 17: 12 Factor - Nebenläufigkeit 27](#_Toc10911411)

[Abbildung 18: Cloud Design Patterns 31](#_Toc10911412)

[Abbildung 19: Index Table Pattern 1 32](#_Toc10911413)

[Abbildung 20: Index Table Pattern 2 32](#_Toc10911414)

[Abbildung 21: Valet Key Pattern 33](#_Toc10911415)

[Abbildung 22: Gateway Aggregation Pattern 34](#_Toc10911416)

[Abbildung 23: Gateway Routing Pattern 35](#_Toc10911417)

[Abbildung 24: Strangler Pattern 35](#_Toc10911418)

[Abbildung 25: Bulkhead pattern 37](#_Toc10911419)

[Abbildung 26 - Lift & Shift Migration 40](#_Toc10911420)

[Abbildung 27 - Docker in Azure VM 41](#_Toc10911421)

Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

# Microservices

Die Wahl der Architektur fiel bei EasyBiz auf Microservices. Diese besteht aus einer Sammlung von kleinen, autonomen Services. Jeder Service ist in sich geschlossen und sollte eine einzige Geschäftsfähigkeit implementieren. Die nachfolgende Abbildung bildet die verschiedenen Microservices von EasyBiz ab.

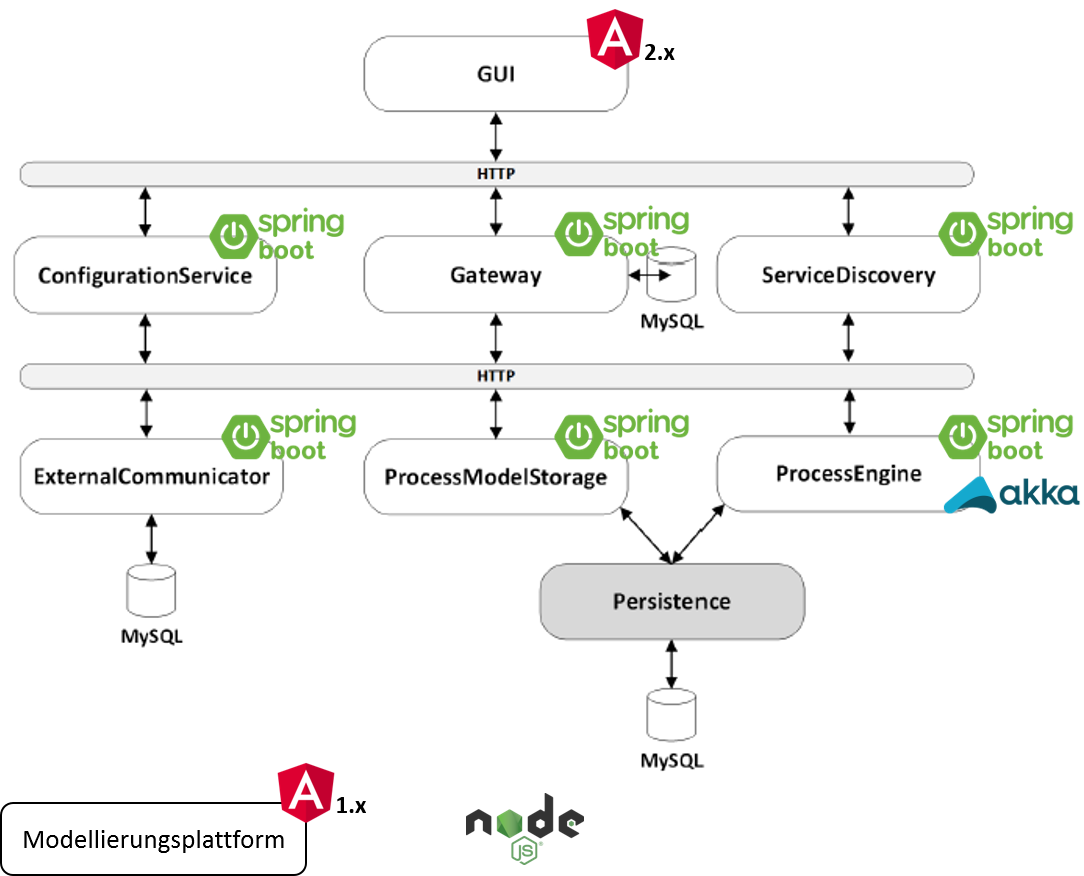


Abbildung 1 - EasyBiz Microservice Architektur

Der aktuelle IST Zustand der Architektur lässt sich eher als SOA Architektur beschreiben, da die Services nicht so schlank und nicht so locker verbunden sind, wie es eigentlich für eine Microservices Architektur notwendig wäre. So wurde zum Beispiel bei den Microservices teilweise im Code „localhost“ angegeben, obwohl die IP Adressen automatisch durch die Services eingetragen werden sollte. Für eine richtige Microservice Architektur müsste ein Reengineering des Codes erfolgen, was im aktuellen Stand nicht Teil der Zielsetzung ist.

## GUI

Die Angular GUI stellt den Benutzerinnen bzw. Benutzern eine grafische Oberfläche bereit, damit diese die Komponenten des Prototyps bedienen können. Demgemäß werden  
unter anderem Menüs bzw. Dialoge für den Import von Prozessmodellen sowie für die Ausführung von Prozessinstanzen bereitgestellt. Dabei kann man die möglichen Operationen der GUI in drei Kategorien einteilen:

* Market Place
* Management Portal
* S-BPMN Modelling Environment

Im Marketplace kann man sich registrieren und einloggen was der Authentifizierung dient. Des Weiteren kann man einen Prozess hochladen oder auch andere Prozesse kaufen. Die verschiedenen Prozesse im Market Place können auch bewertet und kommentiert werden.

Das Management Portal hat zwei zentrale Funktionen. Zum einen werden dort die entsprechenden Rollen für die User vergeben. Zum anderen werden dort die hochgeladenen Modelle validiert und bei positiver Bewertung für den Market Place freigegeben. Diese Aufgabe übernimmt der User mit der Berechtigung create\_approval.

Die S-BPMN Umgebung befindet sich noch im Aufbau. Später sollen hier S-BPMN Modelle erstellt, bearbeitet und als OWL-File exportiert werden können.

## ConfigurationService

Der ConfigurationService ermöglicht die Verwaltung von Konfigurationsdateien in verteilten Systemen. Er ist die zentrale Stelle, in der alle Konfigurationen der Microservices gespeichert sind. Ohne diesen Service müssten alle Konfigurationsdateien in den Microservices selbst gespeichert werden. Das würde das Deployment komplizierter machen. Außerdem ist es durch einen zentralen Konfigurationsserver möglich, Konfigurationen während der Laufzeit eines Microservices zu ändern. Für den ConfigurationService wird Spring Cloud Config verwendet.

Die Dateinamen der Konfigurationsdateien müssen einem definierten Namensschema folgen:

* {application}: Namensangabe der Applikation (festgelegt mittels spring.application.name)
* {profile}: Deklaration des aktuellen Profils der Anwendung (z.B. Test, Produktion, etc.)
* {label}: Versionierung der Anwendung (nützlich für die Ausführung von Microservices

in verschiedenen Versionen)

Diese Konfigurationen werden beim Start der Microservices vom ConfigurationService geladen. Außerdem kann eine POST-http-Anfrage an die Microservices gesendet werden. Darauf laden diese die Konfigurationen neu vom ConfigurationService. Einige Informationen (z.B. Name der Anwendung) müssen jedoch lokal in den Microservices hinterlegt werden.

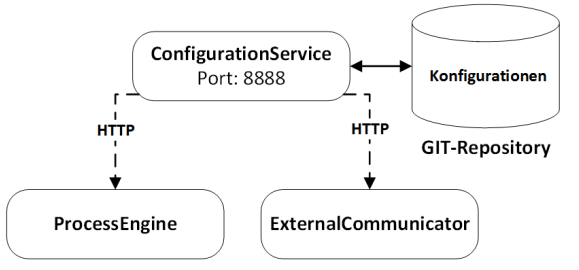


Abbildung 2: Configuration Service

## Gateway

Die GUI sendet alle Anfragen an den Microservice Gateway, der die Anfragen dann an die internen Microservices weiterleitet. Es wird dabei durch einen Authentication Service sichergestellt, dass alle Anfragen, welche weitergeleitet werden, die entsprechenden Berechtigungen besitzen. Wäre dieser Service dort nicht in dieser Form implementiert müsste jeder Microservice über einen eigenen Authentifizierungsmechanismus verfügen.

Das Gateway besteht aus zwei Hauptkomponenten. Die Zuul Gateway API, welcher der Weiterleitung von Anfragen dient und dem der AuthenticationService, welcher die Authentifizierung aller Anfragen sicherstellt. Alle externen Anfragen (z.B. von der GUI) werden an das Gateway gesendet und es entscheidet dann an welchen Mircoservice (z.B. Process Engine) die Anfrage weitergeleitet wird. Zusätzlich nutzt das Gateway REST (Representational State Transfer) Controller bzw. den Feign Client zur Kommunikation.

Das Zuul Gateway dient der Weiterleitung aller Anfragen. Dabei ist es wichtig, dass nur die notwenigen APIs der Microservices veröffentlich werden. Nach dem Empfang der Anfrage und der Zuordnung eines passenden Pfades wird in der Service-Liste geprüft, ob eine verfügbare Serviceinstanz vorhanden ist. Sind mehrere Serviceinstanzen vorhanden wird ein Load Balancing durchgeführt, oft mit Round Robin-Algorithmus um die Anfragen auf die verschiedenen Serviceinstanzen zu verteilen.

Der AuthenticationService ermöglicht die Anbindung an externe Authentifizierungssysteme. Nach Einbindung des externen Authentifizierungssystems, muss sichergestellt werden, dass die externen APIs abgesichert werden und folglich keine Anfragen ohne gültige Authentifizierung akzeptiert werden. Der AuthenticationService basiert auf Role Based Access Control (RBAC). Dieses Prinzip gibt eine Struktur für die Vergabe von Berechtigungen vor. Der AuthenticationService ermöglicht auch die Anbindung an externe Authentifizierungssysteme wie z.B. Active Directory. Es werden also Authentifizierungsanfragen an externe Systeme weiterdelegiert.

Mittels JSON Web Tokens (JWT) wird sichergestellt, dass die externen APIs abgesichert werden und folglich keine Anfragen mehr ohne gültige Authentifizierung entgegengenommen werden. Der Token besteht dabei aus drei Teilen: Header, Payload und Signature. Der nachfolgenden Abbildung kann das Zusammenspiel der Zuul Gateway API und dem Authenticationservice sowie anderer beteiligter Services entnommen werden.

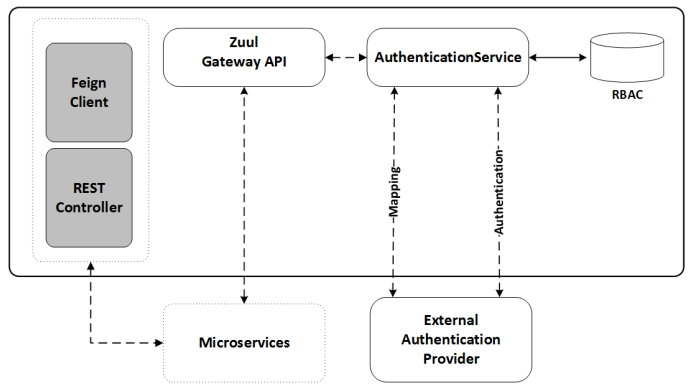


Abbildung 3: Gateway

## ServiceDiscovery

Service Registry und Discovery ist eine der zentralen Komponenten eines typischen Ökosystems von Microservices. Bei einer hohen Anzahl von Microservices, mit dazugehörigen Start- und Stop-Vorgängen ist eine manuelle Vorkonfiguration nur schwer realisierbar. Zusätzlich kann es der Fall sein, dass nicht immer alle URI (Unique Ressource Identifier) schon vor dem Starten der Microservices bekannt sind. Daher muss eine Automatisierung dieses Vorganges implementiert werden. Dies wird mittels des ServiceDiscovery Microservices umgesetzt. An dieser zentralen Stelle tragen sich die Microservices beim Start selbstständig ein. Weiters werden die Services entfernt, welche nicht mehr verfügbar sind. Somit enthält die Service Registry & Discovery die aktuellen Informationen über die Verfügbarkeit der Services sowie deren Metadaten (z.B. URI, Port, etc.). Alle weiteren Microservices können Anfragen an die Service Registry & Discovery senden, um Informationen über die anderen Services zu erhalten.

Im Projekt wird die Spring-Cloud Komponente Eureka, welche von Netflix entwickelt wurde, dafür verwendet. Beim Start registriert sich der Microservice (Client) in einer Service-Liste am Server. Nach der Registrierung der Microservices sendet dieser alle 30 Sekunden einen Ping an den Server. Empfängt der Server keinen Ping eines Clients mehr, dann wird der Client aus der Service-List entfernt. Die Service-Liste wird von den anderen Clients abgerufen und im Cache gespeichert. Somit weiß jeder Service wo der andere zu finden ist.

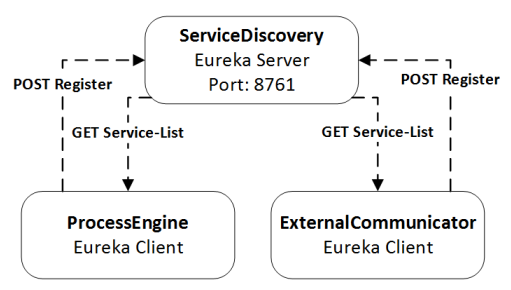


Abbildung 4: Service Discovery

## ExternalCommunicator

Dieser Service bietet die Möglichkeit der Integration von externen Subjekten (Subjekte deren internes Verhalten nicht bekannt ist). Durch den ExternalCommunicator können somit Anfragen an externe Services versendet werden sowie, von diesen Anfragen empfangen und verarbeitet werden. Dafür wird das Execution Framework basierend auf Akka verwendet, dieses verwendet Konfigurationen und Plugins, um das Senden und Empfangen der Nachrichten zu ermöglichen. Die Nachrichten werden mittels Parser und Composer transformiert und anschließend an externe Subjekte oder entsprechende Microservices weitergeleitet. Einen detaillierteren Überblick über den ExternalCommunicator, der während des Projektes gegenüber dem früheren Stand nur geringfügig bzw. gar nicht verändert wurde, liefert die nachfolgende Abbildung.

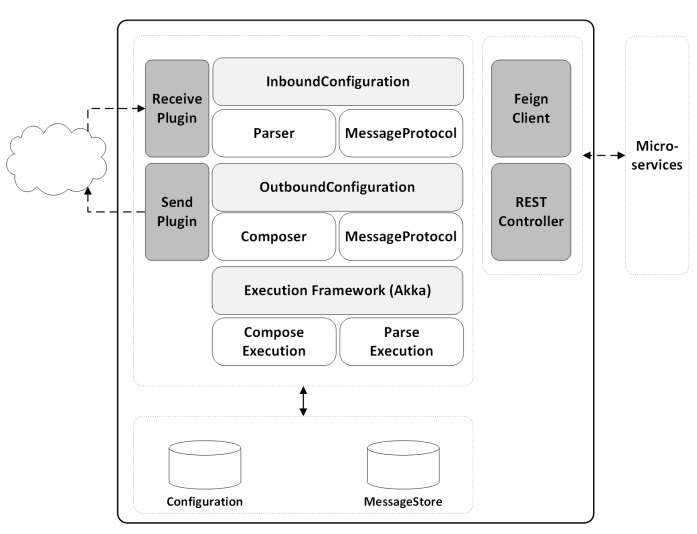


Abbildung 5: External Communicator

## ProcessModelStorage

Der ProcessModelStorage legt den Fokus auf die Implementierung von S-BPM. Die Hauptaufgabe ist die Speicherung und Bereitstellung von subjekt-orientierten Prozessen. Diese Prozesse sollen dann von der ProcessEngine aufgerufen und ausgeführt werden. Der ProcessModelStorage besteht auf zwei Komponenten, dem Parser und dem Import.

Die wichtigste Aufgabe des Parsers ist es, die erstellten Prozessmodelle so zu parsen, dass diese importiert werden können. Geparst wird mittels Apache Jena (einem Java-basierten Framework für Semantic Web). Der Importprozess erfolgt in zwei Schritten. Die GUI stellt die Schnittstelle zur Verfügung um das Web Ontology Language (OWL) File hochzuladen, dieses wird dann an den ProcessModelStorage weitergeleitet. Hier beginnt dann der eigentliche Parsing-Prozess. Das Ergebnis wird anschließend an die GUI zurückgesendet, wo der User weitere Einstellungen vornehmen sollte wie z.B. die Definition einer Versionsnummer sowie eine optionale Beschreibung des Prozessmodells. Anschließend wird das File wieder an den ProcessModelStorage gesendet und die Komponenten (Elemente) in der Datenbank persistiert. Als Resultat dieses Prozesses steht das Prozessmodell nun der ProcessEngine zur Verfügung. Weiter erhält der User noch eine Rückmeldung ob der Import erfolgreich war oder ein Fehler auftrat. Die nachfolgende Abbildung aus der Masterarbeit zeigt die Architektur des ProcessModelStorage Mircoservices.

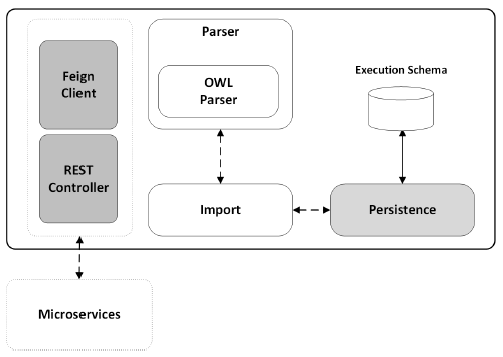


Abbildung 6: Process Model Storage

## ProcessEngine

Die ProcessEngine ist für Ausführung von subjekt-orientierten Prozessen zuständig. Die importierten Prozessmodelle im ProcessModelStorage werden von der ProcessEngine geladen und in Prozessinstanzen überführt. Diese Prozessinstanzen sind dann für den/die Benutzer/in ausführbar. Die Prozessinstanzen werden über die ExecutionEngine gesteuert, hierfür wird das Akka Framework verwendet. Die ProcessEngine enthält auch noch das Java Package Persistance, welches Datenbankobjekte und Builder enthält. Um mit anderen Services kommunizieren zu können, werden die Komponenten REST Controller und Feign Client verwendet. Die nachfolgende Abbildung gibt einen generellen Überblick über die Funktionsweise bzw. Architektur der ProcessEngine.

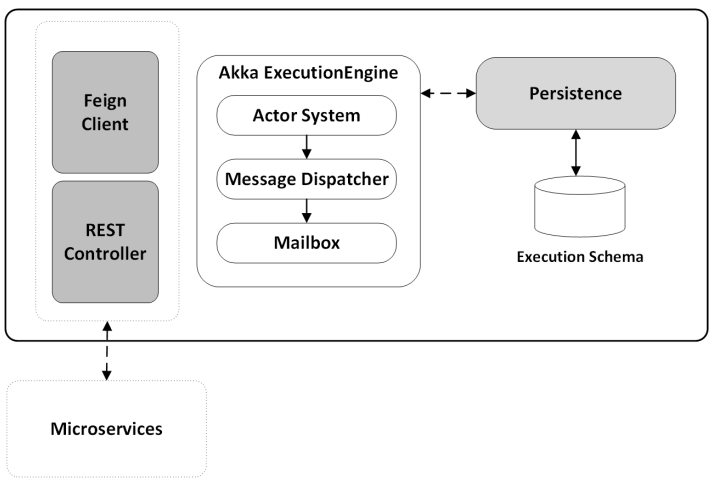


Abbildung 7: Process Engine

Der Microservice wurde nach dem Single-Responsibility-Prinzip aufgebaut, somit hat jeder Aktor nur eine Aufgabe. Weiter wurde auch das Supervisor-Prinzip verwendet, durch dieses werden die Aktoren in Hierarchien gegliedert. Der Supervisor bestimmt den Ablauf, die untergeordneten Aktoren unterstützen. Weiter gibt es eine TaskManager Komponente, diese ermöglicht die Ausführung von Tasks (komplexe Operationen).

## Eventlogger

Der EventLogger speichert verschiedene Events der Prozessausführungen der ProcessEngine und kann somit ggf. für eine Pay-as-you-go Umsetzung genutzt werden, indem die Ausführzeiten der Prozesse getrackt werden. Diese Funktionalität ist aber im aktuellen Stand nicht implementiert.

## ProcessStore

Im ProcessStore werden Meta-Daten der Webplattform gespeichert. Genauer gesagt alle Daten für das Kaufen und Verkaufen der Prozesse sowie der Freigabe und Bewertung dieser. So werden Prozessmodelle, die auf die Webplattform EasyBiz hochgeladen werden, mittels eines Approval-Prozesses geprüft. Wenn sie eine Freigabe erhalten werden sie im Store veröffentlicht und andere User können diese erwerben. Im ProcessStore wird also auch festgehalten, welcher Prozess welchem User gehört und welcher User welchen Prozess erworben hat. Des Weiteren können gekaufte Prozesse bewertet und kommentiert werden. Die Kommentare und Bewertungen werden ebenfalls im ProcessStore persistiert.

## Modeling Plattform

Die S-BPMN Modelling Plattform befindet sich noch in einer Entwicklungsphase und ist nur begrenzt verfügbar/funktional. Später sollen darin auch S-BPMN Prozesse modelliert und anschließend an den ProcessStore zur Freigabe als OWL-File exportiert werden. Es soll ebenfalls eine Änderung bereits existierender Prozessmodelle möglich sein.

# ArchiMate Ist-Analyse Modellierung

Das nachfolgende ArchiMate-Modell repräsentiert die Gesamtarchitektur aller vorhandenen Elemente des Projektes. Es sind dabei nicht alle Elemente im übergebenen Projekt funktional. Beispielsweise ist der BPMN-Modeller nur beschränkt einsetzbar.

Des Weiteren gilt es anzumerken, dass das Modell der Architektur auf verschiedene Weise hätte modelliert werden können. Wir haben den Fokus darauf gelegt die komplette Architektur einmal abzubilden. Es wurden dabei beispielsweise auch Gruppen gebildet, wie das Management-Portal, welche dann von den darunterliegenden Application Services aufgerufen wird.

Das Modell wurde modelliert nach dem Konzept, das der User immer aktiv eine Aktion ausführen muss, welche dann entsprechend von den Services weiterverarbeitet / bedient werden. Die Services reagieren daher passiv auf aktive Interaktionen der User.

Das ArchiMate Modell ist in drei unterschiedlichen Layer gegliedert, den Strategie- Business-Layer, den Application Layer und den Technology- und Physical Layer. Nachfolgend wird auf jeden Layer einzeln eingegangen und die enthaltenen Elemente sowie deren Interagieren erläutert. Begonnen wird dabei beim Business- und Strategy-Layer.

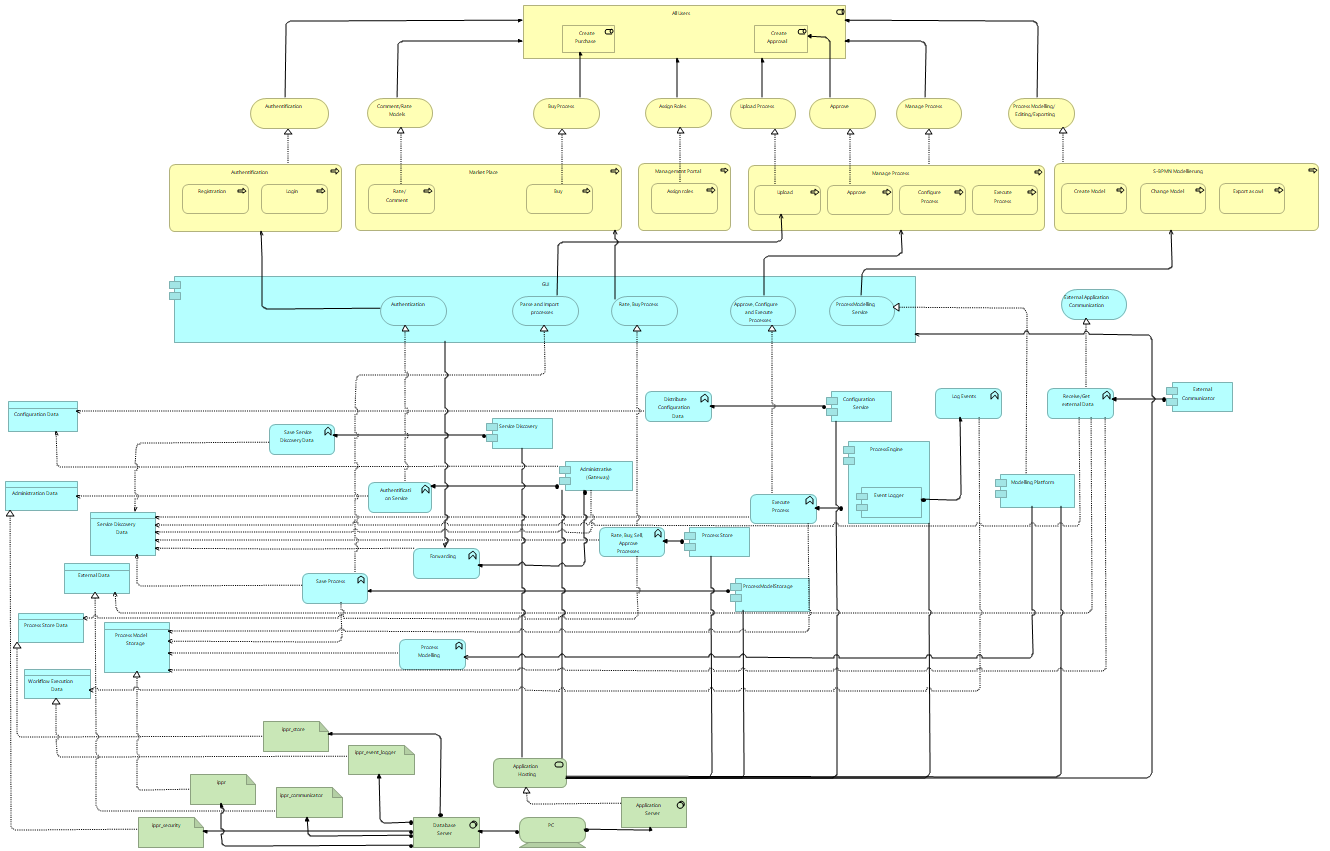


Abbildung : ArchiMate - Ist-Architektur

## Business- und Strategy-Layer

Nachfolgend wird der Business- und Strategy-Layer beschrieben. Die folgende Abbildung soll aufzeigen um welchen Teil des ArchiMate-Modells es sich handelt. Grafisch wird dieser Layer mit der Farbe Gelb dargestellt.

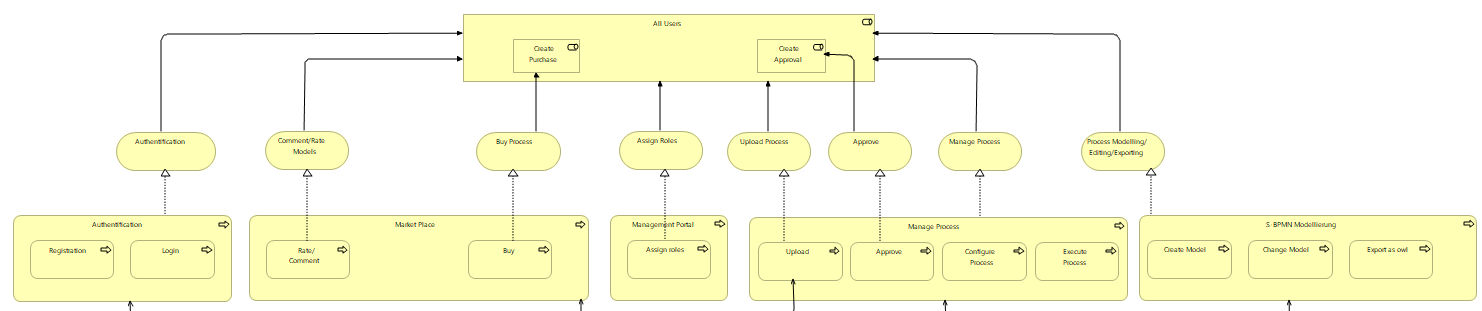


Abbildung 9: ArchiMate - Business- und Strategy Layer

Die oberste Ebene stellt die Rollen von EasyBiz dar. Zum Ist-Stand gibt es zwei verschiedene Berechtigungen, die vollständig implementiert sind:

* Create Purchase
* Create Approval

Wenn keine dieser Berechtigungen vergeben ist, haben alle User Zugriff auf die jeweiligen Prozesse und Funktionen. Weitere Berechtigungen können von den Usern selbstständig hinzugefügt und verwaltet werden.

Die nächste Schicht sind die Business Services von EasyBiz. Diese verbinden die Rollen bzw. Berechtigungen mit den Business Processes. Diese Services sind:

* Authentication
* Comment/Rate Model
* Buy Process
* Assign Roles
* Upload Process
* Approve
* Manage Process

Durch die Pfeile von den Rollen zu den Services wird dargestellt, welche Rolle zu welchen Service und somit Businessprozess führt.

Der unterste Layer besteht aus mehreren Business Prozesse:

* Authentication (Registration, Login)
* MarketPlace (Buy/Comment, Buy)
* ManagementPortal (Assign Roles)
* Manage Process (Upload, Approve, Configure Process, Execute Process)
* S-BPMN-Modellierung (Create Model, Change Model, Export als OWL)

Durch die Berechtigung „Create Purchase“ wird es möglich Einkäufe zu tätigen und mit der „Create Approval“ können Prozesse freigegeben werden. Auf alle anderen Prozesse kann mit allen Usern zugegriffen werden.

## Application Layer

Im Application Layer werden die Microservices und ihre Interaktion miteinander dargestellt. Die Elemente in dieser Schicht sind blau markiert und werden unterteilt in Application Components, Processes, Services und Data Objects. Eine detaillierte Beschreibung der Elemente befindet sich in der ArchiMate Dokumentation.

Die Application Components repräsentieren die einzelnen Microservices von EasyBiz. Die Microservices werden im Kapitel Microservices beschrieben. Das folgende Kapitel befasst sich mit der Repräsentation in ArchiMate und ihre Interaktion untereinander.

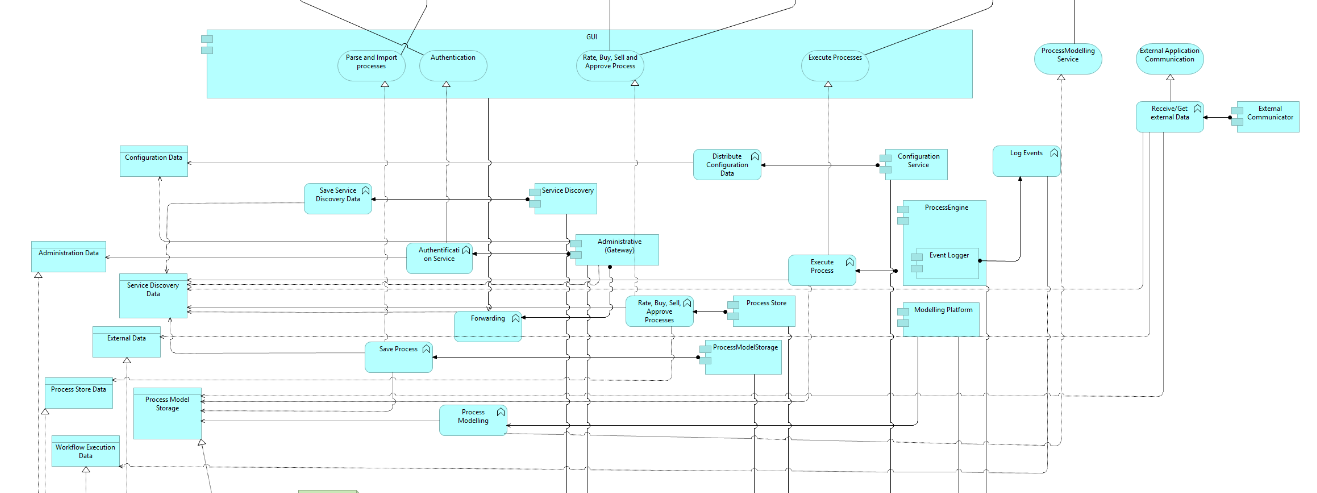


Abbildung 10: ArchiMate - Application Layer

Im Allgemeinen kommuniziert jede Komponente mit den anderen über die Data Objects. Die Data Objects befinden sich alle auf der linken Seite des ArchiMate-Modells. Die Anwendungskomponenten, die die Microservices repräsentieren, greifen über eine Business Function auf die Datenobjekte zu. Diese Business Functions stellen dann die Application Services zur Verfügung, die als Schnittstellen für die Business Layer (Users und Business Functions) beschrieben werden können.

**Gateway**

Wie im Kapitel Microservices beschrieben, bietet das Gateway eine sichere Kommunikation zwischen den Komponenten und leitet die Anfragen aus der GUI weiter. Diese Interaktion wird durch die Service Discovery Data ermöglicht, die die internen Anwendungsdaten darstellen. Wie im Modell ersichtlich, können alle Anwendungskomponenten auf dieses vom Gateway verwaltete Datenobjekt zugreifen.

**GUI**

Wie bereits erwähnt, werden die Anforderungen der GUI als Application Services modelliert und von der GUI Application Component realisiert. Zu den Diensten, die nach den Components gruppiert sind, die sie realisieren, gehören:

* Prozesse analysieren und importieren
* Authentifizierung
* Bewerten, Kaufen, Verkaufen und Genehmigen von Prozessen
* Prozesse ausführen

Diese zwei weiteren Services sind nicht Teil der GUI.

* External Application Communication
* Process Modelling Service

All diese Services dienen als Interface für den Business Layer dieses Modells.

**ProcessEngine**

Diese Komponente ist für die Ausführung der Prozessinstanzen verantwortlich und protokolliert mit der Event Logger Component die Instanzen und deren Ausführungszeit. Der Event Logger ist als Teil der ProcessEngine modelliert, da seine Funktionalität eng mit den Aufgaben der Engine verbunden ist. Weitere technische Details zu diesen beiden Komponenten finden sich in der Beschreibung der Microservices.

Die Process Engine Component führt die Modelle aus den Process Model Storage Data aus und die Ergebnisse werden über die Eventlogger Component in den Workflow Execution Data gespeichert.

**Process Store and Process Model Storage**

Diese beiden Komponenten managen die Data Objects, wo die Prozesse gespeichert, bewertet und gekauft werden:

* Process Store Data
* Process Model Storage

Die Data Models sind in der Lage Daten untereinander zu laden und auszutauschen.

**Configuration Service**

Wie bereits im Kapitel der Microservices beschrieben verteilt der ConfigurationService die Configuration-Data an die Mircoservices, sobald diese starten. Es kann ggf. auch zur Laufzeit die Konfiguration der Microservices geändert werden, indem an diese ein POST-Request durch das Gateway gesendet wird und diese sich dann die neuen Konfigurationen abholen.

Die Konfigurationen werden im Configuration Data Object gespeichert. Dieses Objekt ist dann für das Gateway erreichbar.

**Application Layer Summary**

|  |  |  |
| --- | --- | --- |
| **Component (microservices)** | **Application Function** | **Data Object / Dataschema** |
| Process Engine | Execute Process | Process Model Storage  Service Discovery Data |
| Event Logger | Log Events | Workflow Execution Data |
| Modelling Platform | Process Modelling | Process Model Storage  Service Discovery Data |
| External Communicator | Receive/Get external Data | External Data  Process Model Storage  Service Discovery Data |
| Configuration Service | Distribute Configuration Data | Configuration Data  Service Discovery Data |
| Service Discovery | Save Service Discovery Data | Service Discovery Data |
| Administrative (Gateway) | Authentification Service | Administration Data  Service Discovery Data |
| Process Store | Rate, Buy, Sell, Approve Processes | Process Store Data  Service Discovery |
| ProcessModelStorage | Save Process | Process Model Storage  Service Discovery Data |
| GUI |  | Service Discovery Data |

## Technology- und Physical-Layer

Es existiert eine zentrale Maschine, in der Abbildung als PC beschrieben. Auf dieser läuft ein Application- und ein Datenbank-Server. Der Application-Server wiederrum stellt den Technology-Service „Application Hosting“ zur Verfügung. Dieser steht dann in Verbindung mit den verschiedenen Microservices wie beispielsweise dem Gateway oder der Service Discovery.

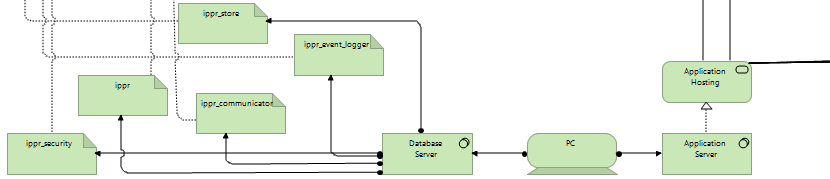


Abbildung 11: ArchiMate – Technology- und Physical-Layer

Der Database-Server verwaltet die Datenbank mit ihren fünf Schemata:

* Ippr\_security: Persistieren der User + Berechtigungen
* Ippr: Persistieren der Prozessmodelle
* Ippr\_store: Persistieren der Daten aus dem Store wie Rating und Kommentare
* Ippr\_event\_logger: Persistieren der Daten des EventLoggers, vor allem Daten der ProcessEngine wie z.B. Ausführungszeiten
* Ippr\_communicator: Persistieren von Daten, die für den ExternalCommunicator relevant sind

Auf diese Daten können kann dann mittels der Daten Objekte des Application Layer entsprechend zugegriffen werden, bzw. werden die Daten dann wieder in die entsprechenden Datenbank-Schemata presistiert.

# Workflow Management Coalition

Die 1993 gegründete Workflow Management Coalition (WfMC) ist eine globale Organisation von Anwendern, Entwicklern, Beratern, Analysten sowie Universitäts- und Forschungsgruppen, die sich mit Workflows und BPM befassen. WfMC erstellt und trägt zu prozessbezogenen Normen bei, informiert den Markt über verwandte Themen und ist die einzige Normungsorganisation, die sich ausschließlich auf den Prozess konzentriert.

## EasyBiz vs WfMC

Basierend auf den WfMC Interoperabilitäts-Standards wurden sechs Haupt-Applikationsfunktionen eines Workflowmanagementsystems identifiziert und gruppiert:

* Workflow Enactment Service / Workflowentscheidungs Service
* Process Definition / Prozessdefinitonen mit Tools
* Client Anwendung (GUI)
* Aufrufen und Interagieren von externen Applikationen
* Interface für andere Workflow Engines
* Administration und Monitoring

Die nachfolgenden Unterkapitel beschreiben zusammenfassend diese Funktionen und vergleichen diese mit der EasyBIZ Applikation. Für detaillierte Informationen über WfMC und der zugehörigen Architektur sollte die umfangreiche WfMC Dokumentation herangezogen werden. Die Kapitel der Mircoservices und ArchiMate Architektur erläutern kurz die technische Implementierung und die Interaktion der Microservices untereinander.

### Workflow Enactment Service

Der Workflow Enactment Service besteht aus einer oder mehreren Workflow Engines um spezielle Workflow Instances zu erstellen, zu managen oder auszuführen. Eine Workflow Engine ist ein Software-Service oder “-Engine“, die eine Ausführungsumgebung für Prozessinstanzen zur Verfügung stellt.

EasyBiz hat eine operationales Prozess-Engine welche durch die GUI bzw. das Applikationsinterface angesprochen werden. Dies Prozess-Engine hat auch die Fähigkeit weitere Applikationen aufzurufen. Aber es hat keine Interfaces, um die Arbeitslast mit anderen Engines zu teilen. Mit Hilfe des EventLoggers können die einzelnen Prozessinstanzen geloggt werden.

### Process Definition

Mit dem Process Definition Tool wird die Prozessbeschreibung in einer für den Computer verarbeitbaren Form erstellt. Diese Prozessdefinition sollte alle notwendigen Informationen über den Prozess enthalten.

EasyBiz ermöglicht durch ihre User-Interfaces die Prozessmodellierung und -definition. Diese Modelle könne als OWL (Web Ontology Language) Files exportiert werden. Andere Modelle können ebenfalls als OWL File importiert werden. Diese Prozessdefinitionen enthalten alle notwendigen organisatorischen sowie Prozessausführungsdaten, die von der Process Engine benötigt werden.

### Client Anwendung (GUI)

Eine Workflow-Clientanwendung ist eine Anwendung, die mit einer Workflow-Engine interagiert und Einrichtungen und Dienste von der Engine abruft.

Durch die userfreundliche EasyBIZ GUI, Können die verschiedenen Funktionen durch den User aufgerufen werden.

### Invoking and interacting with External Application

Es gibt eine Vielzahl an Software, die die Aufgabenbeschreibungen, die Dokumentation und die Übertragung, sowie deren Daten, die außerhalb des Workflow Management Service existieren, verwalten und ermöglichen. Daher sollte ein WfMC-freundliches System in der Lage sein, solche externen Anwendungen aufzurufen und Schnittstellen zur Kommunikation mit ihnen bereitzustellen.

Der External Communicator ist der Microservice, der es der Process Engine ermöglicht externe Applikationen aufzurufen und mit diesen zu interagieren. Der Microservice benutzt dabei sein eigenes Datenschema.

### Interface für andere Workflow Engines

Workflow-Interoperabilität ist definiert als die Fähigkeit von zwei oder mehr Workflow-Engines, zu kommunizieren und zu interagieren, um Workflow-Prozessinstanzen zwischen diesen Engines zu koordinieren und auszuführen.

Die EasyBIZ Process Engine kommunziert nicht mit anderen Engines. Alle Prozessausführungen werden von den Microservices angestoßen und gemanaged. Dennoch existiert eine Komponente innerhalb des External Communicators, die konfiguriert werden kann, um eine Kommunikation mit anderen Engines zu ermöglichen.

### Administration and Monitoring

Innerhalb eines Workflow-Systems gibt es eine Reihe von Überwachungsfunktionen (supervisory functions), die zur Verfügung stehen. Diese basieren typischerweise auf der Grundlage von Supervisionsrechten für einen bestimmten Arbeitsplatz oder Benutzer.

Die GUI ermöglicht den System- und Organisations-Administratoren den Zugang anderer User zu validieren und kontrollieren. Das EasyBIZ Rollen-Schema ermöglicht ein individuelles Anlegen und Verwalten von Rollen und Berechtigungen.

Die Interaktionen und Prozessinstanziierungen werden beide, in den entsprechenden Daten Objekten gelogged, was eine weitere Monitoring-Funktionalität bietet.

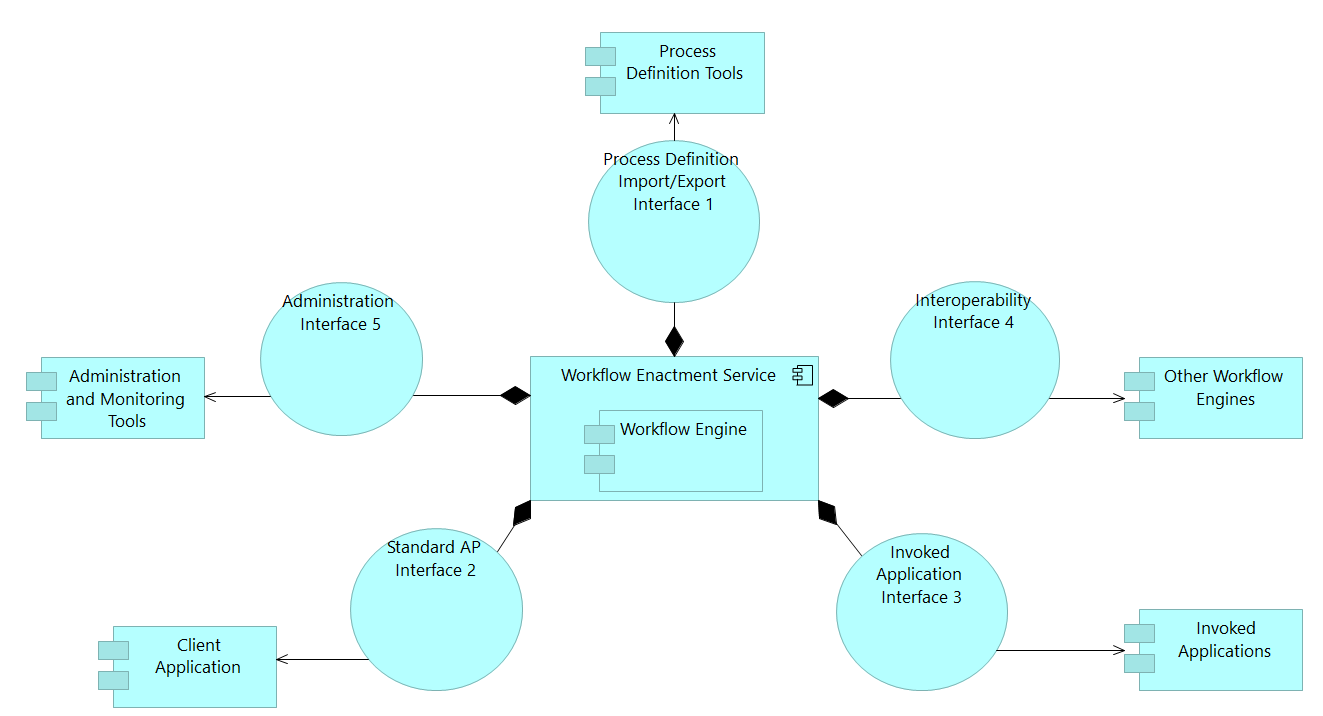


Abbildung 12: Workflow Enactment Service

# Twelve-Factor App Vergleich

Dieser Abschnitt beschäftigt sich mit dem Vergleich der bestehenden Microservices in Bezug auf die sogenannte „Twelve-Factor App“ Methode. Anfangs werden zuerst die Basics dieser Methode erläutert und anschließend wird ein Vergleich zwischen diesen zwölf Prinzipien und der bestehenden Architektur aufgestellt.

### Twelve-Factor App Grundlagen

Die Twelve-Factor App Methode bietet eine Möglichkeit, um Software-As-A-Service Apps zu bauen die:

* deklarative Formate benutzen für die Automatisierung der Konfiguration, um Zeit und Kosten für neue Entwickler im Projekt zu minimieren;
* einen sauberen Vertrag mit dem zugrundeliegenden Betriebssystem haben, maximale Portierbarkeit zwischenda Ausführungsumgebungen bieten;
* sich für das Deployment auf modernen Cloud-Plattformen eignen, die Notwendigkeit von Servern und Serveradministration vermeiden;
* die Abweichung minimieren zwischen Entwicklung und Produktion, um Continuous Deployment für maximale Agilität ermöglichen;
* und skalieren können ohne wesentliche Änderungen im Tooling, in der Architektur oder in den Entwicklungsverfahren.

Die Zwölf-Faktoren-Methode kann auf Apps angewendet werden, die in einer beliebigen Programmiersprache geschrieben sind, und die eine beliebige Kombination von unterstützenden Diensten benutzen (Datenbank, Queue, Cache, …)

Wie der Name bereits vermuten lässt, besteht diese Methode aus den folgenden zwölf Faktoren:

|  |  |
| --- | --- |
| **I. Codebase** | Eine im Versionsmanagementsystem verwaltete Codebase, viele Deployments |
| **II. Abhängigkeiten** | Abhängigkeiten explizit deklarieren und isolieren |
| **III. Konfiguration** | Die Konfiguration in Umgebungsvariablen ablegen |
| **IV. Unterstützende Dienste** | Unterstützende Dienste als angehängte Ressourcen behandeln |
| **V. Build, release, run** | Build- und Run-Phase strikt trennen |
| **VI. Prozesse** | Die App als einen oder mehrere Prozesse ausführen |
| **VII. Bindung an Ports** | Dienste durch das Binden von Ports exportieren |
| **VIII. Nebenläufigkeit** | Mit dem Prozess-Modell skalieren |
| **IX. Einweggebrauch** | Robuster mit schnellem Start und problemlosen Stopp |
| **X. Dev-Prod-Vergleichbarkeit** | Entwicklung, Staging und Produktion so ähnlich wie möglich halten |
| **XI. Logs** | Logs als Strom von Ereignissen behandeln |
| **XII. Admin-Prozesse** | Admin/Management-Aufgaben als einmalige Vorgänge behandeln |

Weitere Informationen zu diesen Faktoren befinden sich unter: <https://12factor.net>.

Anhand dieser zwölf Faktoren lässt sich einfach feststellen, wie zukunftssicher eine Software aufgestellt ist. In diesem Zusammenhang folgt nun eine Gegenüberstellung der zwölf Faktoren mit der bestehenden Software EasyBiz, inklusive einer genaueren Erklärung der einzelnen Punkte.

### Vergleich mit bestehender Software

**I. Codebase:**

Eine Zwölf-Faktor-App wird immer in einem Versionsmanagementsystem verwaltet, wie zum Beispiel Git, Mercurial oder Subversion. Eine Kopie der Versionsdatenbank wird Code Repository genannt, kurz Code Repo oder auch nur Repo. Eine Codebase ist jedes einzelne Repo (in einem zentralisierten Versionsmanagementsystem wie Subversion) oder jede Menge von Repos, die einen ursprünglichen Commit teilen (in dezentralisierten Versionsmanagementsystemen wie git). Es gibt nur eine Codebase pro App aber viele Deploys der App. Ein Deploy ist eine laufende Instanz der App. Meist gibt es ein Produktionssystem und eines oder mehrere Staging-Systeme.

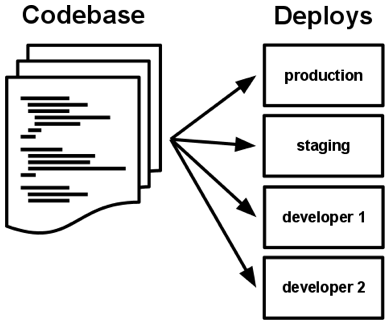


Abbildung 13: 12 Factor - Codebase

Der Code der bestehenden EasyBiz Plattform wird zurzeit auf mehreren GitHub Repositories gehostet. Das aktuelle Repository befindet sich unter: <https://github.com/I2PM/Cloudbased-S-BPM-WfMS>. Zurzeit gibt es eine Master Branch als Deploy und noch weitere Branches für das Development. Zusätzlich haben die Entwickler lokale Kopien des Codes, was auch als einzelner Deploy pro Entwickler zählt.

**II. Abhängigkeiten**

Die meisten Programmiersprachen bieten ein System an, um unterstützende Bibliotheken zu verbreiten, wie CPAN für Perl oder Rubygems für Ruby. Aus einem Paketsystem stammende Bibliotheken können systemweit installiert werden (auch “Site Packages” genannt) oder in ein Verzeichnis der App beschränkt werden (genannt “vendoring” oder “bundling” - deutsch auch “mitliefern”). Eine Zwölf-Faktor-App verlässt sich nie auf die Existenz von systemweiten Paketen. Sie deklariert alle Abhängigkeiten vollständig und korrekt über eine Abhängigkeitsdeklaration. Weiter benutzt sie zur Laufzeit ein Werkzeug zur Isolation von Abhängigkeiten um sicherzustellen, dass keine impliziten Abhängigkeiten aus dem umgebenden System “hereinsickern”. Die vollständige und explizite Spezifikation der Abhängigkeiten wird gleichermaßen in Produktion und Entwicklung angewandt.

Grundsätzlich werden die Abhängigkeiten von EasyBiz über Gradle verwaltet. Sprich die Abhängigkeiten befinden sich nicht selbst in der Codebase. Weiters unterstützt Spring Boot Dependency Injection, das bedeutet, dass die Abhängigkeiten noch weiter isoliert und verwaltet werden.

**III. Konfiguration**

Die Konfiguration einer App ist alles, was sich wahrscheinlich zwischen den Deploys ändern wird (Staging, Produktion, Entwicklungsumgebungen, usw.). Dies umfasst:

* Resource-Handles für Datenbanken, Memcached und andere unterstützende Dienste
* Credentials für externe Dienste wie Amazon S3 oder Twitter
* Direkt vom Deploy abhängige Werte wie der kanonische Hostname für den Deploy

Die Zwölf-Faktor-App speichert die Konfiguration in Umgebungsvariablen (kurz auch env vars oder env). Umgebungsvariablen von Deploy zu Deploy zu ändern ist einfach; im Gegensatz zu Konfigurationsdateien ist es unwahrscheinlich, dass sie versehentlich ins Code Repository eingecheckt werden und im Gegensatz zu speziellen Konfigurationsdateien oder anderen Konfigurationsmechanismen wie den Java Properties sind sie Sprach- und Betriebssystemunabhängig.

Die Software EasyBiz beinhaltet einen Configuration Service, welcher die Konfiguration an die entsprechenden Microservices weiterverteilt. Hier sind Ports und Hostnamen der Microservices und der Datenbank hinterlegt.

EasyBiz ist noch nicht so dynamisch konfiguriert, dass die Konfiguration über externe Variablen beeinflusst werden kann. Spring Boot würde jedoch grundsätzlich die Möglichkeit über POST Requests bieten die Konfiguration von außen zu beeinflussen, dies müsste aber noch genauer recherchiert und implementiert werden.

**IV. Unterstützende Dienste**

Ein unterstützender Dienst ist jeder Dienst, den die App über das Netzwerk im Rahmen ihrer normalen Arbeit konsumiert. Beispiele sind Datenspeicher (wie MySQL oder CouchDB), Messaging/Queueing-Systeme (wie RabbitMQ oder Beanstalkd), SMTP-Dienste für das Senden von Mail (wie Postfix), und Cache-Systeme (wie Memcached).

Der Code einer Zwölf-Faktor-App macht keinen Unterschied zwischen lokalen Diensten und solchen von Dritten. Für die App sind sie beide unterstützende Dienste, zugreifbar über eine URL oder andere Lokatoren/Credentials, die in der Konfiguration gespeichert sind. Ein Deploy einer Zwölf-Faktoren-App könnte eine lokale MySQL-Datenbank, durch eine von Dritten zur Verfügung gestellten, ersetzen (wie Amazon RDS).

Jeder einzelne unterstützende Dienst ist eine Ressource. So ist zum Beispiel eine MySQL-Datenbank eine Ressource; zwei MySQL-Datenbanken (die für ein Sharding auf Applikationsebene verwendet werden) stellen zwei Ressourcen dar. Dass die Zwölf-Faktor-App diese Datenbanken als angehängte Ressourcen behandelt, zeigt ihre lose Kopplung, zu dem Deploy an dem sie hängen.

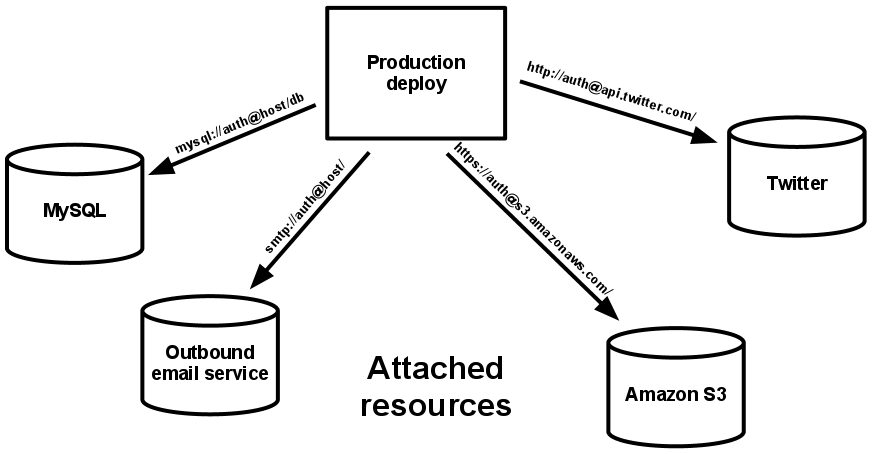


Abbildung 14: 12 Factor - Unterstützende Dienste

In EasyBiz ist es über den Configuration Server einfach entsprechende Verbindungen zu unterstützenden Diensten anzugeben. So ist es mühelos möglich, eine lokale MySQL Instanz mit einer externen MySQL Instanz auszutauschen. Die Verbindung zu diesen externen Diensten erfolgt dann über den zuvor erwähnten External Communicator. Dieser erlaubt die Kommunikation mit externen Subjekten, jedoch wird diese Funktion zurzeit nicht in der Anwendung verwendet.

Um jedoch eine Änderung der unterstützenden Dienste ohne Code-Änderung durchführen zu können, müssen noch Anpassungen an der Konfiguration und an dem Configuration Server getätigt werden.

**V. Build, release, run**

Eine Codebase wird durch drei Phasen in einen (Nicht-Entwicklungs)-Deploy transformiert:

* Die Build-Phase ist eine Transformation, die ein Code-Repository in ein ausführbarers Code-Bündel übersetzt, das man auch Build nennt. Ausgehend von einer Code-Version eines Commits, der im Deployment-Prozess festgelegt wurde, holt sie Abhängigkeiten, verpackt sie zum Mitliefern, und kompiliert Binaries und Assets.
* Die Release-Phase übernimmt den Build von der Build-Phase und kombiniert ihn mit der zum Deploy passenden Konfiguration. Der so erzeugte Release enthält sowohl den Build, als auch die Konfiguration und kann direkt in einer Ausführungsumgebung ausgeführt werden.
* Die Run-Phase (auch “Laufzeit” genannt) führt die App in einer Ausführungsumgebung aus, indem sie eine Menge der Prozesse der App gegen einen ausgewählten Release ausführt.

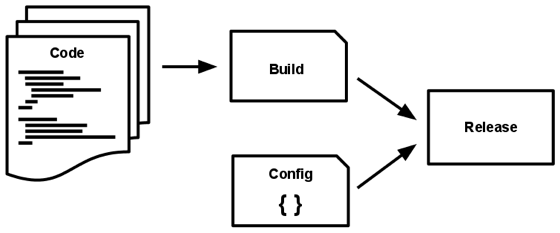


Abbildung 15: 12 Factor - Build, release, run

Die Zwölf-Faktor-App trennt strikt zwischen Build-, Release- und Run-Phase. Es ist nicht möglich, Code-Änderungen zur Laufzeit zu machen, weil es keinen Weg gibt, diese Änderungen zurück in die Build-Phase zu schicken.

Für EasyBiz ist der erste Schritt der Builderstellung auf jeden Fall gegeben, jedoch gibt es noch Unklarheiten beim Erstellen von Releases und deren Verwaltung. Grundsätzlich gibt es die Möglichkeit die Release Verwaltung über GitHub durchzuführen, jedoch wird dies noch nicht so gehandhabt. Hier würde es noch Verbesserungspotential im Rahmen der 12-Faktor Methode geben. Ausführen lässt sich die Software direkt über die Build Files (jar) beziehungsweise, soll die Software in Zukunft auch containerisiert werden.

**VI. Prozesse**

Die App wird grundsätzlich als ein oder mehrere Prozesse ausgeführt. Im einfachsten Fall ist der Code ein Stand-alone-Skript, die Ausführungsumgebung ist der lokale Laptop eines Entwicklers mit einer installierten Laufzeitumgebung einer Sprache, und der Prozess wird von der Kommandozeile gestartet. Am anderen Ende des Spektrums kann eine hochentwickelte App viele Prozesstypen benutzen, die in keinen oder mehreren Prozessen laufen.

Zwölf-Faktor-Apps sind zustandslos und Shared Nothing. Alle Daten werden in unterstützenden Diensten gespeichert, normalerweise einer Datenbank. Der RAM oder das Dateisystem des Prozesses kann als kurzfristiger Cache für die Dauer einer Transaktion verwendet werden. Zum Beispiel kann ein Prozess eine Datei herunterladen, sie verarbeiten und die Ergebnisse in einer Datenbank speichern. Die Zwölf-Faktor-App geht nie davon aus, dass irgendetwas aus dem RAM oder im Dateisystem zwischengespeichertes für einen künftigen Request oder Job verfügbar sein wird.

In EasyBiz werden alle Daten in einer MySQL Datenbank, also in einem unterstützenden Dienst gespeichert. Es werden keine Daten in Stick Sessions, im Cache oder ähnlichen Diensten zwischengespeichert.

Weiters kann jeder einzelne Mircoservice als eigener Prozess angesehen werden, welcher unabhängig von den anderen instanziiert werden kann. Hierzu müsste jedoch noch ein entsprechendes Load-Balancing eingebaut werden.

**VII. Bindung an Ports**

Web-Apps laufen manchmal in einem Webserver als Container. Zum Beispiel laufen PHP-Apps als Modul in Apache HTTPD, oder Java-Apps laufen manchmal in Tomcat (EasyBiz verwendet Tomcat).

Die Zwölf-Faktor-App ist vollständig eigenständig und verlässt sich nicht darauf, dass ein externer Webserver zur Laufzeit injiziert wird, um dem Web einen Dienst zur Verfügung zu stellen. Die Web-App exportiert HTTP als Dienst, indem sie sich an einen Port bindet und wartet an diesem Port auf Requests.

Grundsätzlich erfüllt EasyBiz diese Herausforderungen, jedoch muss der Configuration Server zuvor erreichbar sein um einen erfolgreichen Start der Microservices gewährleisten zu können. Durch die Verwendung von Docker wird mittlerweile auch für jeden Service ein statischer Port benötigt.

**VIII. Nebenläufigkeit**

Jedes Computerprogramm wird, wenn es läuft, repräsentiert durch einen oder mehrere Prozesse. Webapps nutzen verschiedenste Arten der Prozess-Ausführung. Zum Beispiel laufen PHP-Prozesse als Kind-Prozesse von Apache und werden nach Bedarf gestartet, wenn Requests kommen. Java-Prozesse gehen anders vor: die JVM stellt einen massiven Über-Prozess zur Verfügung der große Mengen an Systemressourcen (Speicher und CPU) reserviert und die Nebenläufigkeit wird intern über Threads verwaltet. In beiden Fällen sind die laufenden Prozesse für die Entwickler der App nur minimal zu sehen.

Die Skalierung wird dargestellt als laufende Prozesse, die Diversität der Workload wird dargestellt als Prozesstypen. In der Zwölf-Faktor-App sind Prozesse First Class Citizens. Die Prozesse der Zwölf-Faktor-App orientieren sich am Unix-Prozess-Modell für Service Daemons. Mit diesem Model können Entwickler ihre App für die Bearbeitung verschiedenster Aufgaben konzipieren in dem sie jeder Aufgabe einen Prozesstyp zuweisen. Zum Beispiel können HTTP-Requests durch einen Web-Prozess bedient werden und langlaufende Hintergrundarbeiten durch einen Worker-Prozess.

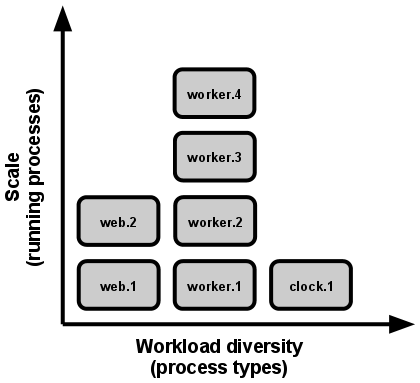


Abbildung 16: 12 Factor - Nebenläufigkeit

Das Prozess-Modell glänzt besonders, wenn es um Skalierung geht. Die Shared-Nothing, horizontal teilbare Art und Weise der Prozesse der Zwölf-Faktor-App hat zur Folge, dass weitere Nebenläufigkeit einfach und zuverlässig hinzugefügt werden kann. Das Bündel von Prozesstypen und die Zahl der Prozesse wird auch Prozess-Formation genannt. Die Prozesse einer Zwölf-Faktor-App sollten nie als Daemons laufen oder PID-Dateien schreiben. Stattdessen sollen sie sich auf den Prozessmanager des Betriebssystems verlassen.

EasyBiz verwendet keine Daemons, sondern klassische Java Prozesse wie zuvor erwähnt. Jeder einzelne Microservice / Prozess wird also als eigener, unabhängiger Java Prozess gestartet.

**IX. Einweggebrauch**

Die Prozesse einer Zwölf-Faktor-App können weggeworfen werden, sie können also schnell gestartet und gestoppt werden. Dies erleichtert schnelles elastisches Skalieren, schnelles Deployment von Code oder Konfigurationsänderungen und macht Produktionsdeployments robuster.

Prozesse sollten möglichst geringe Startup-Zeiten haben. Idealerweise braucht ein Prozess wenige Sekunden vom Startkommando bis der Prozess läuft und Requests oder Jobs entgegennehmen kann. Eine kurze Startup-Zeit gibt dem Release-Prozess und der Skalierung mehr Agilität; sie hilft der Robustheit, weil ein Prozessmanager bei Bedarf einfacher Prozesse auf neue physikalische Maschinen verschieben kann. Wie auch immer ist eine Zwölf-Faktor-App darauf ausgelegt mit unerwarteten, irregulären Stopps umgehen zu können.

Für EasyBiz gelten ähnliche Vorrausetzungen. Die einzelnen Services sind recht robust und können theoretisch gesehen „weggeworfen“ werden. Das bedeutet, dass die Microservices schnell wieder gestartet und gestoppt werden können. Dies trifft insbesondere dann zu, wenn die Microservices komplett containerisiert worden sind. Wie sich ein Stoppen der einzelnen Microservices jedoch auf den laufenden Betrieb auswirkt, ist abhängig von der Art der Services. Wenn z.B. das Gateway gestoppt wird, könnte es zu Problemen mit der gesamten Anwendung kommen. Hier würde sich Load-Balancing erneut anbieten. Das bedeutet, dass grundsätzlich die Infrastruktur recht robust ist, die einzelnen Services aber trotzdem Probleme verursachen könnten.

**X. Dev-Prod-Vergleichbarkeit**

Historisch gibt es große Lücken zwischen Entwicklung (wo ein Entwickler live an einem lokalen Deploy der App Änderungen macht) und Produktion (ein laufender Deploy der App, auf den Endbenutzer zugreifen). Diese Lücken zeigen sich auf drei Gebieten:

* **Die Zeit-Lücke** Ein Entwickler arbeitet an Code der Tage, Wochen oder sogar Monate braucht um in Produktion zu gehen.
* **Die Personal-Lücke**: Entwickler schreiben Code, Operatoren deployen ihn.
* **Die Werkzeug-Lücke**: Entwickler nutzen vielleicht einen Stack wie Nginx, SQLite und OS X - die Produktion nutzt Apache, MySQL und Linux.

Die Zwölf-Faktor-App ist ausgelegt auf Continuous Deployment indem sie die Lücke zwischen Entwicklung und Produktion klein hält. Mit Blick auf die oben beschriebenen drei Lücken:

* Die Zeit-Lücke klein halten: Ein Entwickler kann Code schreiben, der Stunden oder sogar Minuten später deployed wird.
* Die Personal-Lücke klein halten: Entwickler die Code schreiben sind intensiv am Deployment und der Überwachung des Verhaltens auf Produktion beteiligt.
* Die Werkzeug-Lücke klein halten: Entwicklung und Produktion so ähnlich wie möglich halten.

Der Zwölf-Faktor-Entwickler widersteht dem Drang, verschiedene unterstützende Dienste in Entwicklung und Produktion zu verwenden, selbst wenn Adapter über alle Unterschiede hinweg abstrahieren. Unterschiede zwischen den unterstützenden Diensten verursachen kleinste Inkompatibilitäten, und Code, der in Entwicklung oder Staging funktioniert und Tests besteht, scheitert in Produktion.

Da EasyBiz noch nicht im Produktivbetrieb ist, ist dieser Faktor recht schwer zu beurteilen. Von der Werkzeug-Seite her lässt sich jedoch sagen, dass grundsätzlich nur eine MySQL Datenbank benötigt wird. Hier sollte es also keine Probleme zwischen Dev und Prod geben. Bezüglich Personal- und Zeit-Lücke lassen sich zurzeit keine Angaben machen.

**XI. Logs**

Logs machen das Verhalten einer laufenden App sichtbar. In Server-basierten Umgebungen werden sie üblicherweise in eine Datei auf der Platte geschrieben (eine Logdatei) - das ist aber nur ein Ausgabeformat. Logs sind der Stream von aggregierten, nach Zeit sortierten Ereignissen und zusammengefasst aus den Output Streams aller laufenden Prozesse und unterstützenden Dienste. Logs in ihrer rohen Form sind üblicherweise ein Textformat mit einem Ereignis pro Zeile (obwohl Backtraces von Exceptions über mehrere Zeilen gehen können). Eine Zwölf-Faktor-App kümmert sich nie um das Routing oder die Speicherung ihres Output Streams. Sie sollte nicht versuchen, in Logdateien zu schreiben oder diese zu verwalten. Grundsätzlich verfügt EasyBiz über einen EventLogger welcher nur für Ausführung der Prozesse entsprechende Logs anlegt. Die restlichen MicroServices verwenden individuelles Logging, so verfügt der Process Store z.B. über seine eigenen Logs. Zurzeit ist das Log Level noch auf „Debug“ konfiguriert, jedoch können hier auch noch andere Levels ausgewählt werden.

Die einzelnen Logs der Microservices werden jedoch nirgends gesammelt und deswegen auch nicht ausgewertet. Hier bietet sich also noch großes Verbesserungspotential im Rahmen der zwölf Faktor Methode.

**XII. Admin-Prozesse**

Einmalige Administrationsprozesse sollten in einer Umgebung laufen, die identisch ist zu der Umgebung der üblichen langlaufenden Prozesse. Sie laufen gegen einen Release und benutzen dieselbe Codebase und Konfiguration wie jeder Prozess, der gegen einen Release läuft. Administrationscode wird mit dem App-Code ausgeliefert um Synchronisationsprobleme zu vermeiden.

Solche einmaligen administrativen Prozesse wurden für EasyBiz noch nicht festgelegt. Weiters wurde zurzeit auch noch kein Konzept für diese Prozesse erarbeitet. Es müssten entsprechende Szenarien definiert werden um anschließend die nötigen Tools (etc. REPL Shell) zu eruieren.

### Zusammenfassung & Verbesserungen

Die folgende Zusammenfassung beinhaltet alle Verbesserungsvorschläge für einen zukünftigen Stand der Software.

|  |  |
| --- | --- |
| **I. Codebase** | Die Voraussetzungen sind bereits erfüllt. |
| **II. Abhängigkeiten** | Die Voraussetzungen sind bereits erfüllt. |
| **III. Konfiguration** | Hier bietet sich noch die Möglichkeit, die Konfiguration im laufenden Betrieb zu ändern. Dies muss noch entsprechend recherchiert und umgesetzt werden. Grundsätzlich sollte dies über POST Requests durchgeführt werden können. |
| **IV. Unterstützende Dienste** | Die Voraussetzungen sind bereits erfüllt. |
| **V. Build, release, run** | Dieser Faktor ist nur teilweise erfüllt. An dieser Stelle müsste noch ein passendes Build/Release Management umgesetzt werden. |
| **VI. Prozesse** | Die Voraussetzungen sind bereits erfüllt. |
| **VII. Bindung an Ports** | Die Voraussetzungen sind bereits erfüllt. (Durch die Verwendung von statischen Ports.) |
| **VIII. Nebenläufigkeit** | Die Voraussetzungen sind bereits erfüllt. |
| **IX. Einweggebrauch** | Grundsätzlich ja, jedoch kann es bei den einzelnen Services zu Problemen kommen falls ein Service ausfällt der relevant für die gesamte Funktionalität der Anwendung ist. |
| **X. Dev-Prod-Vergleichbarkeit** | Dieser Faktor ist zurzeit noch nicht relevant für diese Anwendung, da es noch keine Produktivlösung gibt. |
| **XI. Logs** | Jeder einzelne Service loggt selbst, jedoch gibt es noch keinen Eventstream der alle Logs sammelt und somit eine entsprechende Auswertung ermöglichen würde. |
| **XII. Admin-Prozesse** | Es gibt zurzeit noch keine Admin-Prozesse, jedoch könnten z.B. Prozesse erstellt werden wie das erneute Abholen der Konfiguration beim Configuration Service mittels POST Request. |

# Cloud Design Patterns

Im folgenden Abschnitt werden die bereits umgesetzten Designpatterns von EasyBiz beschrieben. Außerdem wird auch auf jene eingegangen, wo eine Umsetzung zum aktuellen Stand ebenfalls sinnvoll wäre. Weitere Designpatterns, die zwar sinnvoll wären, aber einen zu hohen Implementierungsaufwand darstellen, werden im Weiteren nicht erwähnt.

Abbildung 17: Cloud Design Patterns

## Availability

Die Availability ist die Zeit, in der ein System funktional und lauffähig ist. Normalerweise wird sie als Prozentsatz der Uptime gemessen. Da bei Cloud Applikationen SLAs gelten, ist es wichtig die Availability so hoch wie möglich zu halten.

**Health Endpoint Monitoring**

Für ein Health Endpoint Monitoring werden Checks implementiert, die die Microservices in regelmäßigen Intervallen prüfen. Hierbei kann geprüft werden ob die virtuelle Maschine lauft oder auch ob der spezielle Service im Container läuft und auf z.B. HTTP Anfragen antwortet.

Zurzeit ist noch kein Monitoring implementiert, dies kann man jedoch über das eingebaute Monitoring Feature in Azure, über third-party Framework wie Microsoft System Center Operations Manager oder über ein benutzerdefiniertes Dienstprogramm oder Service auf einem Server, gelöst werden.

## Data Management

Data Management ist der wichtigste Teil einer Cloud Anwendung. Daten werden normalerweise an verschieden Orten und über verschiedene Services gehostet um beispielsweise die Performance oder die Verfügbarkeit zu verbessern. Wichtig ist hierbei, dass die Konsistenz der Daten gewährleistet wird.

**Index Table Pattern**

Hierfür werden zusätzliche Indexe für die Felder einer Datenbank erstellt. Diese Indexe beziehen sich auf Fehler die oft für Queries benötigt werden. In relationalen Datenbanken sind secondary indexes normalerweise möglich, bei NoSQL Datenbanken kann man diese Funktionalität über eigene Index Tables implementieren. Hierfür kann entweder die ganze Tabelle, mit anderem Index, dupliziert werden. Dies ist jedoch nur sinnvoll, wenn die darin enthaltenen Daten statisch sind, da ansonsten der Verarbeitungsaufwand für die Verwaltung der Indextabellen zu groß wird. Weiters brauchen die zusätzlichen Tabellen viel Platz. Um das zu vermeiden gibt es noch eine weitere Möglichkeit. Hierfür wird eine Fact Table angelegt. Die jeweiligen Index Tables referenzieren die Daten über den Primary Key der Fact Table.

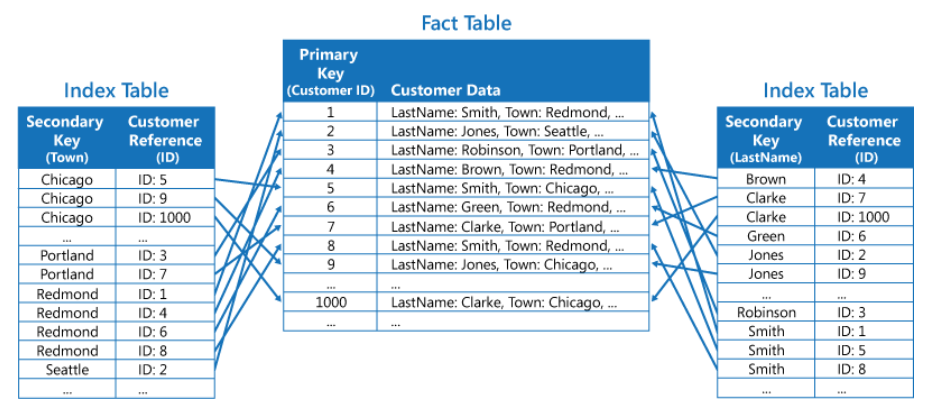


Abbildung 18: Index Table Pattern 1

Bei der dritten Methode werden die Index Tables mit unterschiedlichen Keys organisiert und duplizieren oft benötigte Felder aus der Fact Table.

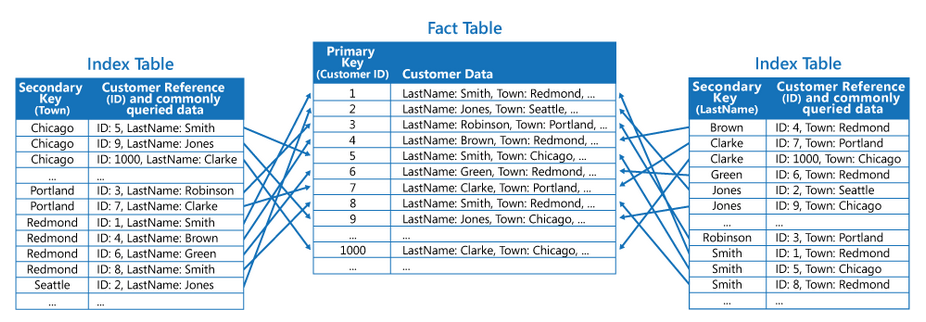


Abbildung 19: Index Table Pattern 2

Eazy Biz arbeitet mit relationalen Datenbanken, daher ist es einfach möglich secondary indexes zu verwenden.

**Static Content Hosting Pattern**

Statische Inhalte können in einem cloudbasierten Speicherdienst bereitgestellt werden. Dieser kann die Inhalte direkt an den Client übermitteln. Diese statischen Ressourcen können HTML-Seiten, Bilder oder Dokumente.

Bei Eazy Biz wäre es möglich die BLOB Files vom Process Store in einen eigenen Speicherdienst abzulegen und diese statisch zur Verfügung zu stellen.

**Valet Key Pattern**

Der Valet Key Pattern verlangt nach einem Token, der Clients eingeschränkten Zugriff auf bestimmte Ressourcen bietet, um die Datenübertragung aus der Anwendung zu verschieben. Dies ist besonders nützlich in Anwendungen, die Cloud-gehostete Speichersysteme oder Queues verwenden, und kann die Kosten minimieren und die Skalierbarkeit und Leistung maximieren.

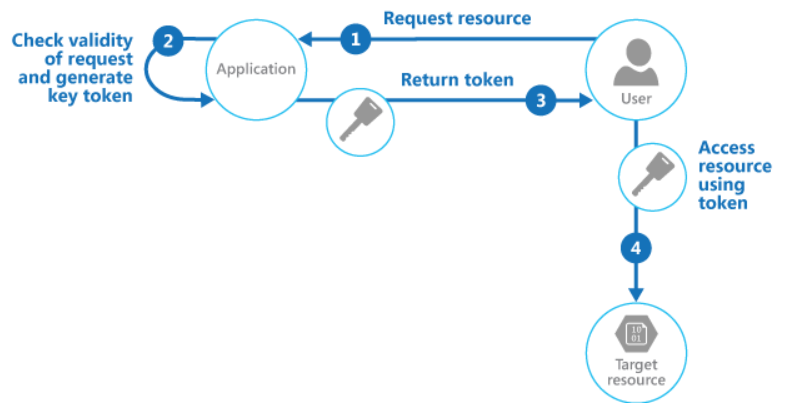


Abbildung 20: Valet Key Pattern

Der aktuelle IST Zustand bei EasyBiz ist, dass der Valet Key Pattern noch nicht vollständig implementiert wurden. Die User bekommen bereits einen Token mit Berechtigungen. Time Limits, damit die Session abläuft, existieren noch nicht und müssten noch implementiert werden.

## Design and Implementation

Design- und Implementierungsentscheidungen haben großen Einfluss auf die Qualität und die TCO (Total Cost of Ownership) für in der Cloud gehostete Services. Dazu gehören Themen wie Konsistenz und Kohärenz während dem Design und der Entwicklung von Komponenten, die Vereinfachung von Verwaltung und Entwicklung sowie die Wiederverwendbarkeit, damit Komponenten und Subsysteme in anderen Anwendungen oder Szenarien genutzt werden können.

**External Configuration Store pattern**

Die Konfigurationsinformationen werden aus dem Anwendungsentwicklungspaket verschoben in einen eigenen, zentralen Service. Das ermöglicht ein einfacheres Management und Kontrolle über die Konfigurationsdaten, sowie das Teilen der Konfigurationsdaten in der Anwendung.

Bei EasyBiz gibt es einen eigenen Microservice der die Konfigurationen der anderen verwaltet und diese beim Start oder bei Änderungen pushed. Mehr Informationen zum sogenannten ConfigurationService findet man im Kapitel Microservices.

**Gateway Aggregation pattern**

Um mehrere Requests zusammenzufassen, wird ein Gateway verwendet. Dieses Muster ist nützlich, wenn ein Client mehrere Aufrufe an verschiedene Back-End-Systeme ausführen muss, um eine Operation auszuführen.

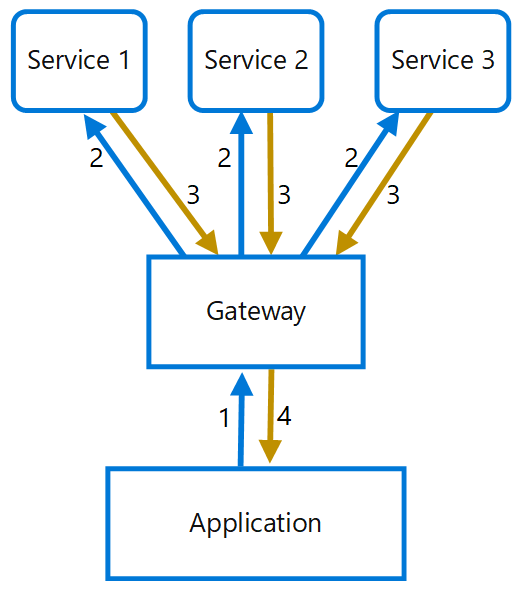


Abbildung 21: Gateway Aggregation Pattern

Durch die Microservice Architektur von EasyBiz kommen alle Requests über die GUI zum Gateway und werden von dort and die jeweiligen Services weitergeleitet. Weitere Informationen zum Gateway finden sich im Kapitel Microservices.

**Gateway Routing Pattern**

Der Gateway Routing Pattern verlangt nach einer Weiterleitung von Anforderungen mithilfe eines einzelnen Endpunktes (Gateway). Dies ist vor allem nützlich wenn mehrere Dienste auf einen einzelnen Endpunkt verfügbar sein sollen. Dieser leitet die Requests dann an die zuständigen Services weiter.

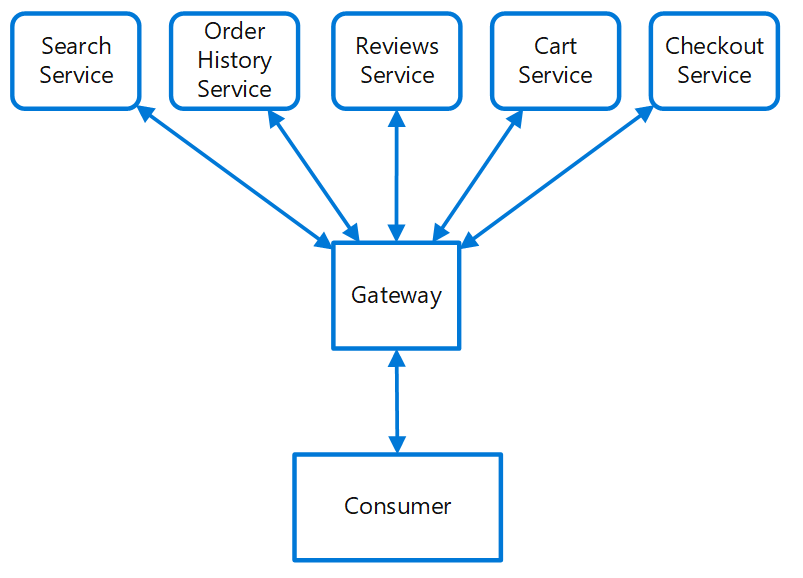


Abbildung 22: Gateway Routing Pattern

Dieses Pattern wurde, wie der Gateway Aggregation Pattern, über die Microservice Architektur gelöst. Der Gateway Service leitet alle Anforderungen an die benötigten Services weiter.

**Sidecar pattern**

Um Isolation zu gewährleisten werden die Komponenten einer Anwendung in separate Prozesse oder Container unterteilt. Der Name Sidecar stammt daher, dass die Services wie ein Sidecar zu einem Motorrad agieren. Das Sidecar ist an der Parent-Anwendung angebracht und stellt unterstützende Funktionen bereit. Das Sidecar hat denselben Lebenszyklus wie die übergeordnete Anwendung und wird zusammen mit der übergeordneten Anwendung erstellt und neu erstellt.

Durch die Microservice Architektur werden automatisch die Komponenten in unterschiedliche isolierte Prozesse oder Container unterteilt.

**Static Content Hosting**

**Strangler Pattern**

Der Strangler Pattern umfasst die inkrementelle Migration eines Altsystems. Das Altsystem wird nach und nach durch neue Anwendungen und Dienste ersetzt bis alle Dienste migriert wurden. Danach können die alten Dienste aus dem System genommen werden.

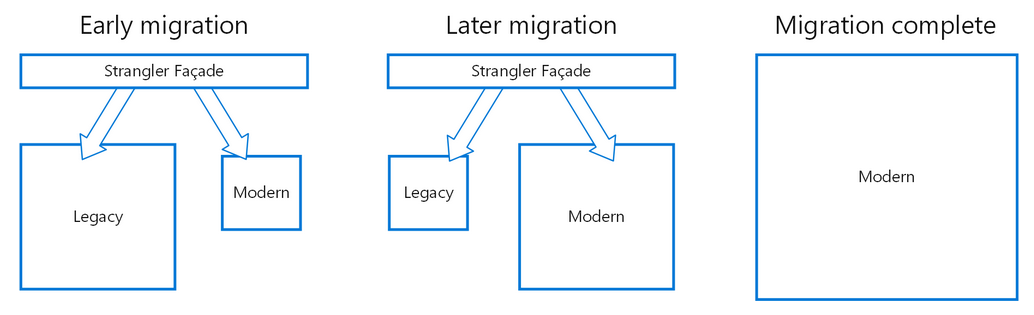


Abbildung 23: Strangler Pattern

Der Strangler Pattern würde bei EasyBiz genutzt werden sobald es zu Migrationen der Services kommt.

## Messaging Patterns

Der verteilte Aufbau einer Cloud Infrastruktur benötigt eine Nachrichten Infrastruktur, um Komponenten mit Services zu verbinden. Oft werden asynchrone Nachrichten verwendet.

**Publisher Subscriber Pattern**

In cloudbasierten und verteilten Anwendungen ist es oft notwendig, dass Informationen von einer Komponente des Systems zu einer anderen Komponente weitergegeben werden muss. Asynchrones Messaging ist eine effektive Methode, um Absender von Verbrauchern zu entkoppeln und zu vermeiden, dass der Absender blockiert wird, während er auf Antwort wartet. Wichtig ist zu beachten, dass die Nachrichten nur an berechtigte Services weitergeleitet werden dürfen.

Bei EasyBiz verwendet der Service Discovery Microservice so eine Art der Kommunikation mit den anderen Services. Weitere Informationen zum Service Discovery befinden sich im Kapitel Microservices.

## Management and Monitoring Patterns

Da Cloud Anwendungen nicht in einem lokalen Datencenter ausgeführt werden, besitzt man nicht die volle Kontrolle über die Infrastruktur. Das führt zu einem schwierigeren Management und auch Monitoring der Anwendungen. Die Anwendung muss Informationen zur Laufzeitumgebung freigeben, damit Administratoren die Anwendung managen und monitoren können.

Alle Cloud Design Patterns, welche dieser Kategorie zugewiesen werden, wurden bereits zuvor beschrieben, deswegen wird hier nicht nochmals auf diese eingegangen. Die bereits umgesetzten oder leicht umsetzbaren Patterns sind die folgenden:

* Gateway Aggregation Pattern
* Gateway Routing Pattern
* Health Endpoint Monitoring Pattern
* Sidecar Pattern
* Strangler Pattern

## Performance and Scalability Patterns

Die Performance eines Systems zeigt ob ein System in einen bestimmten Zeitintervall Aufgaben ausführen kann. Scalability steht für die Fähigkeit eines Systems einen Lastzuwachs zu bewältigen, ohne die Performance zu beeinflussen oder die verfügbaren Ressourcen automatisch zu erhöhen. In einer Cloudumgebung sollte es möglich sein, dass Anwendungen in einem Limit automatisch skalieren, um Leistungspeaks on demand abhandeln zu können und sich danach wieder reduzieren. Das umfasst nicht nur Rechnerinstanzen, sondern auch Datenspeicher, Nachrichteninfrastruktur und mehr.

Alle Cloud Design Patterns, welche dieser Kategorie zugewiesen werden, wurden bereits zuvor beschrieben, deswegen wird hier nicht nochmals auf diese eingegangen. Die bereits umgesetzten oder leicht umsetzbaren Patterns sind die folgenden:

* Index Table Pattern
* Sharding Pattern
* Static Content Hosting Pattern

## Resilliency Patterns

Resiliency ist die Fähigkeit eines Systems, Fehler ordnungsgemäß zu behandeln und zu beheben. Um die Ausfallsicherheit aufrechtzuerhalten, müssen Fehler erkannt und schnell und effizient behoben werden.

**Bulkhead pattern**

Damit bei einem Ausfall nicht das ganze System betroffen ist, werden Teile der Anwendung isoliert, somit beeinflusst ein Ausfall dieser Teile nicht das ganze System.

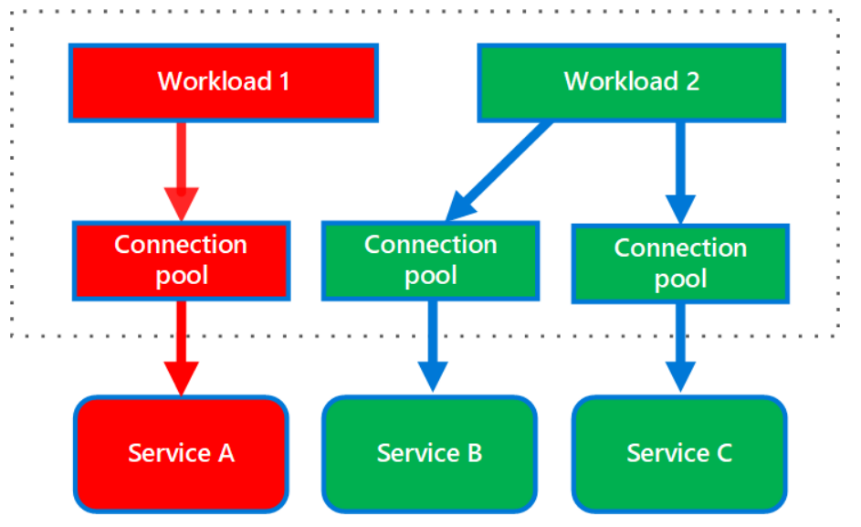


Abbildung 24: Bulkhead pattern

Teile der Architektur von EasyBiz sind so aufgebaut, dass es egal für die restliche Anwendung ist, wenn diese wegfallen. Durch die Microservice Architektur sind die Services prinzipiell schon von einander getrennt. Wenn beispielsweise der ConfigurationService oder der Eventlogger wegfallen hat es keine Auswirkungen auf den Rest.

**Health Endpoint Monitoring**

## Security Patterns

Security Patterns dienen dazu, das System vor böswilligen oder versehentlichen Handlungen außerhalb der vorgesehenen Verwendung zu beschützen. Außerdem muss der Verlust oder die nicht erlaubte Weitergabe von Daten vermieden werden. Cloud Anwendungen sind über das Internet abrufbar, daher müssen Anwendungen so designed werden werden, dass böswillige Attacken vermieden und verhindert werden können. Auch Berechtigungen sind ein wichtiges Thema in der Cloud, daher dürfen User nur auf bestimmte Ressourcen zugreifen. Dieser Zugriff muss administrativ geregelt werden (RBAC).

Alle Cloud Design Patterns, welche dieser Kategorie zugewiesen werden, wurden bereits zuvor beschrieben, deswegen wird hier nicht nochmals auf diese eingegangen. Die bereits umgesetzten oder leicht umsetzbaren Patterns sind die folgenden:

* Valet Key Pattern0

# SOLL-Szenarien Cloud Migration

## Managed Service

In diesem Szenario soll der Managed Service Azure App Service genutzt werden um die Applikation EasyBiz in die Cloud zu migrieren und dort produktiv laufen zu lassen. Hierbei hat man wiederum zwei Möglichkeiten. Die eine wäre einen App Service zu erstellen mit Docker Compose Support und alle Microservices auszuführen. Die zweite Möglichkeit wäre für jedes Microservice, abgesehen der Datenbank, einen eigenen App Service zu erstellen. Die Migration der Datenbank gestaltet sich mit einem eigenen Service in Azure am besten. Hierfür würde eine Azure Database for MySQL aufgesetzt werden.

Die größte Schwierigkeit, die dieses Szenario mit sich bringt, ist die Komplexität der Kommunikation zwischen den Microservices und der Datenbank die als eigenes Service ausgeführt wird. Man muss, durch Öffnen der richtigen Ports bei den jeweiligen Services, dafür sorgen, dass alle Microservices von extern erreichbar sind und miteinander kommunizieren können. Zusätzlich müssen alle Microservices mit der Datenbank kommunizieren können.

## Container Instances

Eine weitere Möglichkeit die Applikation EasyBiz nach Azure zu migrieren wäre durch die Verwendung des Azure Container Instances Service. In diesem Fall würden alle Microservices in isolierte Container gehostet werden und in einer Azure Container Group zusammengefasst sein, die dann über einen zentralen FQDN von extern erreichbar wäre. Nachteil dieser Variante ist jedoch, dass Azure Container Instances kein natives Service Discovery zwischen den bereitgestellten Containern ermöglicht. Zusätzlich wird auch kein Docker Compose unterstützt. Hier müsste man sich also ein Konzept aufbauen die die Kommunikation zwischen den Microservices ermöglicht.

# IST-Szenarien Cloud Migration

In diesem Kapitel werden die zwei umgesetzten Szenarien für eine Cloud Migration der Applikation EasyBiz beschrieben. Im Rahmen des Unterrichts von AIM18 wurde in EBUSA ebenfalls eine Entscheidung bezüglich der passendsten Cloud-Computing Plattform gefällt. Diese Entscheidung fiel auf die Plattform Microsoft Azure (genauere Infos dazu gibt es im entsprechenden Dokument). Diese Szenarien beinhalten eine klassische Lift & Shift Migration, eine Variante die Docker Compose mithilfe einer VM in Azure nutzt.

## Lift & Shift Migration

Dieses Szenario ist von der Implementierung her, das simpelste und am wenigsten aufwändige Szenario. Hier werden einfach die bestehenden Microservices in einer Virtuellen Maschine in der Cloud ausgeführt. Es werden keine Änderungen an der Architektur oder an den Services vorgenommen. In Microsoft Azure muss hierzu einfach eine passende VM angelegt werden. Diese VM sollte zumindest über 16GB RAM und 4 vCPU verfügen.

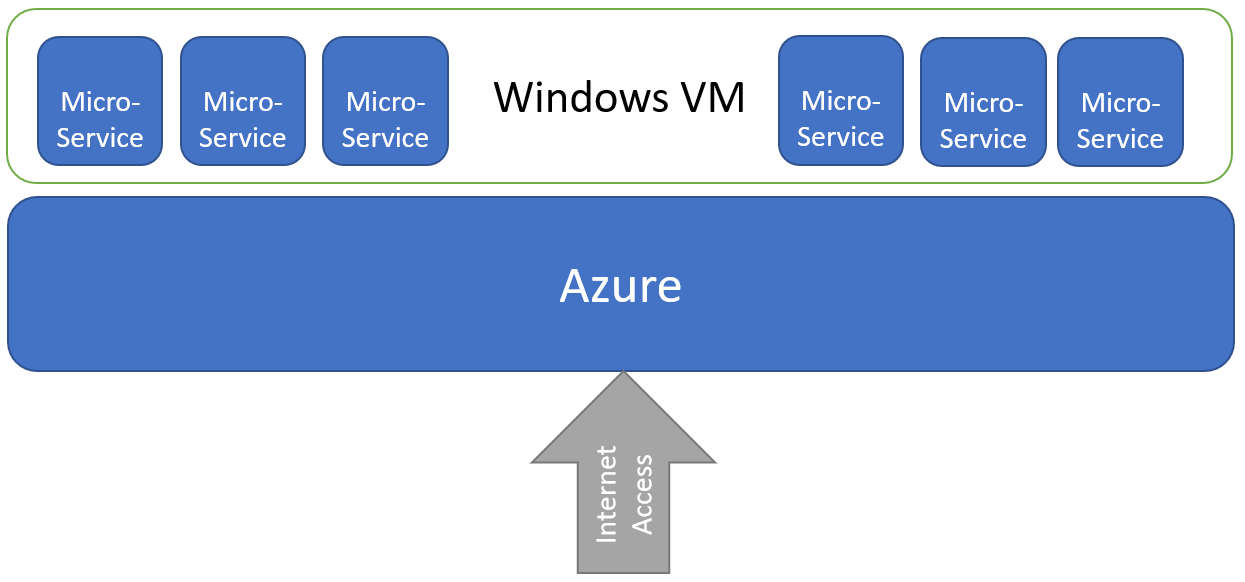


Abbildung 25 - Lift & Shift Migration

In diesem Szenario wird die VM in Azure als Host für die Applikation EasyBiz verwendet und gleich eingesetzt wie ein lokaler Client auf dem die Applikation, ohne Docker ausgeführt wird. Es wird der Source Code von GitHub gepullt und alle notwendigen Programme (Git, NodeJS, Java, Gradle) in der VM installiert. Danach wird die Applikation in der VM ausgeführt ohne diese in einem Docker Container laufen zu lassen, ganz im Stile einer „Just another application in the Cloud“ Lösung.

## Azure for Docker Migration

In diesem Szenario werden die Microservices der Applikation EasyBiz auch in einer Azure VM gehostet, jedoch werden diese in Docker provisioniert, wie in der Abbildung 26 gezeigt.

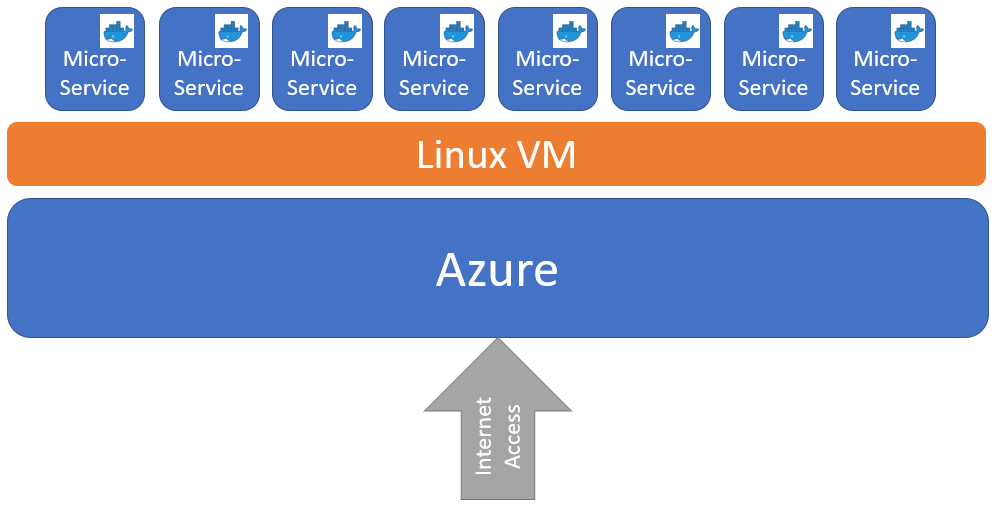


Abbildung 26 - Docker in Azure VM

In diesem Fall würde es grundsätzlich keine Rolle spielen, ob es sich hierbei um eine Windows oder um eine Linux VM handelt. Jedoch unterstützt Windows Server 2019 Docker Container nur nach bestimmten Konfigurationen und nur mit einer Windows 10 Pro Version. Für den Produktivbetrieb ist jedenfalls eine Linux VM empfohlen, da hier keine Lizenzkosten für die VM anfallen und Docker bereits nativ unterstützt wird.

Um dieses Szenario umzusetzen müssen alle JAR Files vorerst gebuildet und dabei die Ordnerstruktur beibehalten werden. Anschließend wird die Linux VM in Azure erstellt und alle benötigten Daten in die VM kopiert. Auf der Maschine selbst muss dann nur mehr Docker Compose ausgeführt werden, um die Applikation EasyBiz auszuführen.

Wichtig hierbei ist, dass sichergestellt wird, dass die einzelnen Services miteinander kommunizieren können. Das wird grundsätzlich über Docker Compose geregelt. Wenn die GUI von der VM aus erreicht werden kann und alle Funktionen erfolgreich getestet werden können, kann die externe Erreichbarkeit der Anwendung überprüft werden. Über die Public IP kann die VM auch von außen erreicht werden, jedoch muss sichergestellt werden, dass die entsprechenden Ports auch geöffnet sind. Dies kann bei dem jeweiligen Netzwerk Interface der VM in Azure konfiguriert werden.

Falls die Anwendung nun auch von extern erreicht werden kann, wurde die Applikation erfolgreich in die Cloud migriert. Nun können optional noch passende DNS Records gesetzt werden, um die Anwendung auch direkt über einen Namen erreichbar zu machen.