Abschlussprüfung Sommer 2021

Fachinformatik für Anwendungsentwicklung

Dokumentation zur betrieblichen Projektarbeit

Cloud Speicheroptimierung

Entlastungen und Kosteneinsparung des bestehenden Speichers durch Anpassungen der Infrastruktur

Prüfungsbewerber:
Christine Dizon
Abt-Rudolf Str. 29,
73479 Ellwangen
29.07.1994

Ausbildungsbetrieb:
PlanB. GmbH
Kocherstr. 15
73460 Hüttlingen

Inhalt

Einleitung

1.1 Projektziel

Das Abschlussprojekt wird in dem Kundenprojekt „ESB – Enterprise Service Bus“ für einen namenhaften internationalen Technologiekonzern erarbeitet. Hierfür soll eine Azure Funktion und ein „Application Programming Interface“ (API ist eine technische Schnittstelle) so angepasst werden, dass die Speicherung von Kundendaten und internen Daten in den korrekten Blob Storage gespeichert werden.

Der bestehende Speicher ist jedoch sehr teuer. Um die Kosten zu senken, kam die Idee einen weiteren Blob Storage zu implementieren, mit anderen Zugriffsoptionen. Diese Zugriffsoptionen können festgelegt werden und immer wieder verändert werden. Die Daten des bestehenden Speichers müssen nicht alle mit derselben Redundanz gelagert werden.

1.2 Projektumfeld

Der IT-Konzern PlanB. GmbH, deren Hauptstandort in der Kocherstraße 15, 73460 Hüttlingen liegt, ist Spezialist in microsoftbasierten Technologien. Die PlanB. GmbH bietet ihre Dienstleistungen an viele internationale und regionale Unternehmen an und unterstützt diese bei technischen Lösungen und Anwendungen. Die Firma PlanB. hat über 100 Mitarbeiter. Die meisten IT-Spezialisten arbeiten am Firmensitz in Hüttlingen. Weitere Büros befinden sich in Frankfurt und in Bulgarien.

Für den großen Auftraggeber entwickelt die PlanB unter anderem im Projekt ESB („Enterprise Service Bus“) für das Schnittstellenmanagement seit Anfang 2017. Das Schnittstellenmanagement ist für die Kommunikation verschiedener Geschäftsbeziehungen verantwortlich. Zum einen gibt es die B2C - Beziehung (Business to Customer) und die B2B – Beziehung (Business to Business).

1.3 Projektbegründung

Das ESB Projekt entstand aufgrund des immensen Datenflusses. Der Kunde wollte eine Lösung, um Daten schnell, sicher und einfach zu speichern und weiterzuleiten, ohne manuell sich darum kümmern zu müssen. Die ESB Thematik gibt es schon seit längerer Zeit. Das Projekt befindet sich also zwischen Datenankunft und Datenweiterleitung. Dadurch ist das System flexibler, mobiler und ist mit einem High-Level Protokoll (ermöglicht Steuerung und Sicherung von Punkt-Punkt-Verbindungen auf dedizierten Netzabschnitten) zur Kommunikation zwischen Applikationen ausgestattet.

1.4 Prozessschnittstellen

Die Komponenten des ESB-Projekts werden in der Cloud gehostet und auch dort entwickelt. Die Projektmitglieder nutzen die Microsoft Cloud namens „Azure“. Für das ESB benötigt das Team spezielle Lösungen, bzw. Services. Zum einen „Azure Blob Storage“. Dieser Service ist eine Dokumenten Datenbank und der andere Service der benutzt wird lautet „Azure Functions“ (containerbasierte Konsolenanwendungen, zu deutsch Funktion). Azure Functions werden im Visual Studio entwickelt. Die Projektmitglieder arbeiten mit einer „virtuellen Maschine“ (Software, die vorgibt, ein eigenständiger Rechner zu sein), die in das interne virtuelle Netzwerk des Kunden eingebunden ist und die Azure Anwendungsoberfläche zur REST API Konfiguration und der zugehörigen Richtlinien verwendet. Die Coding Richtlinien des Kunden befinden sich im sogenannten Projektwiki, welches sich in Azure DevOps befindet. Das Azure DevOps ist ein weiterer Service von Azure, der das Backlog (Aufgaben für die Entwickler), die Repositories und Testpläne enthält. Die Testpläne werden vom Entwickler erstellt und auch automatisch gestartet werden. Diese Pläne werden auch „Integration Tests“ genannt, welche in der Azure Function implementiert werden.

1.5 Kundenwünsche

Der Wunsch des Kunden ist die korrekte Speicherung der Daten. Als Entwickler muss man sicherstellen, dass beim Absenden eines Requests durch die API die Daten nicht im falschen Container gespeichert werden. Dies waren die einzigen Kundenwünsche zu diesem Problem, die auch als Musskriterium ins Pflichtenheft aufgenommen wurden.

1.6 Ansprechpartner

Betreuer für meine Abschlussarbeit ist mein Projektleiter Markus Meyer. Er ist der Technology Advisor vom ESB Projekt. Eine Benutzerdokumentation gibt es nicht, jedoch eine technische Dokumentation, da der Product Owner selbst im Projekt mitarbeitet und sich mit den verwendeten Technologien bestens auskennt.

2. Projektplanung

2.1 Ablaufplan

Anbei die Projektphasen inklusive Zeitaufwand für jeden Arbeitsschritt:

Arbeitsschritt	**Zeitaufwand**
Analyse der Anforderungen, Finden der technischen Lösung, Projektplan und Pflichtenheft	15
Anlegen des Azure Speichers	5
Frontend der Api	15
Qualitätssicherung und Testen	15
Datenmigration	5

Erstellen der Dokumentation	10
Übergabe von Entwicklung und Dokumentation	5
Insgesamt	**70**

2.2 Ressourcenplanung

Die hier dargestellte Planung der Ressourcen besteht aus den benutzen Arbeitsmitteln, aus den Azure Kosten, den Personalkosten und der Zeitplanung.

Arbeitsmittel

Arbeitsmittel	**Anzahl**
Notebook	1
Azure Lizenz	1
Visual Studio	1
Arbeitsplatz und Internet	1
AT&T Global Network Client	1

Azure Kosten

Zusätzliche Kosten entstehen keine, da es bereits bestehende Kosten für die Durchführung der Applikation gibt. Implementationskosten kommen jedoch noch hinzu. Die bestehenden Ressourcen für die Problembehebung können ohne weitere Kosten benutzt werden.

Personalkosten

Name	**Position**	**Kosten/h**	**Zeitaufwand**	**Kosten**
Christine Dizon	Junior Entwickler	47,50 €	70	3.325 €
Markus Meyer	Projektleiter/Betreuer	95€	6	570 €
				= 3.895 €

Terminplanung

Im Projekt findet jeden Tag ein Meeting statt, das wir „Daily“ nennen. An dem Daily nehmen alle Projektmitglieder teil, sowie die Entwickler des Kunden. Im ESB folgen wir dem Projektmanagement „Kanban“. Durch das Kanban wird die Anzahl der parallelen Arbeiten begrenzt, um kürzere Durchlaufzeiten zu erreichen.

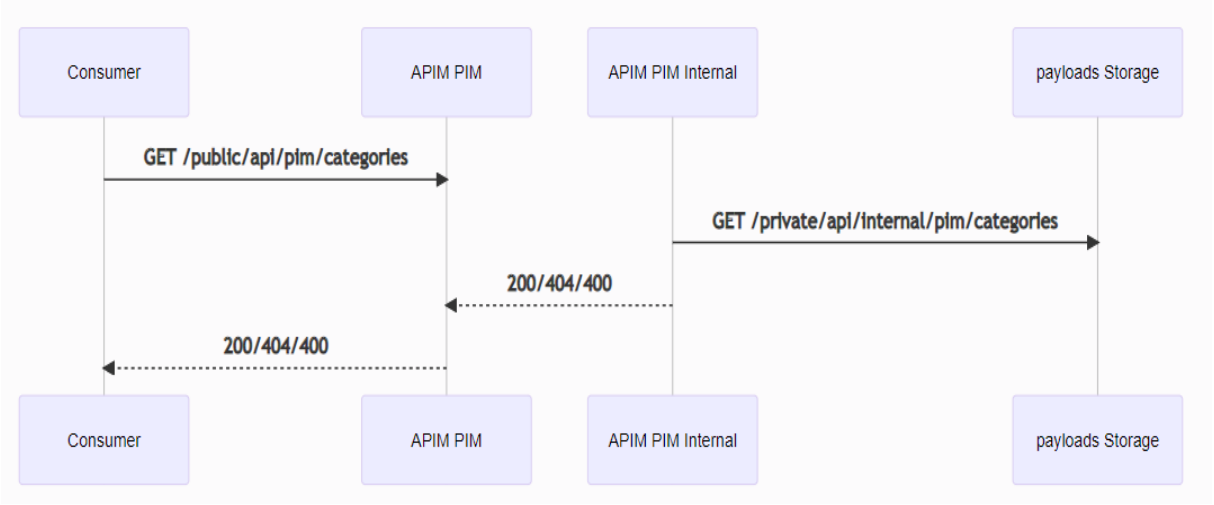
Des Weiteren gibt es eine Besprechung bezüglich des Vorgehens und der Infrastruktur beim Start der Projektdurchführung, das auch zum Kanban gehört. Am Ende meines Projektes wird es dann eine weitere Besprechung geben mit dem Kunden inklusive Übergabe der Entwicklung bzw. Anpassung. Zu dieser Besprechung kommt auch mein Projektleiter hinzu, der mich in dieser Thematik betreut.

2.3 Prozessablauf

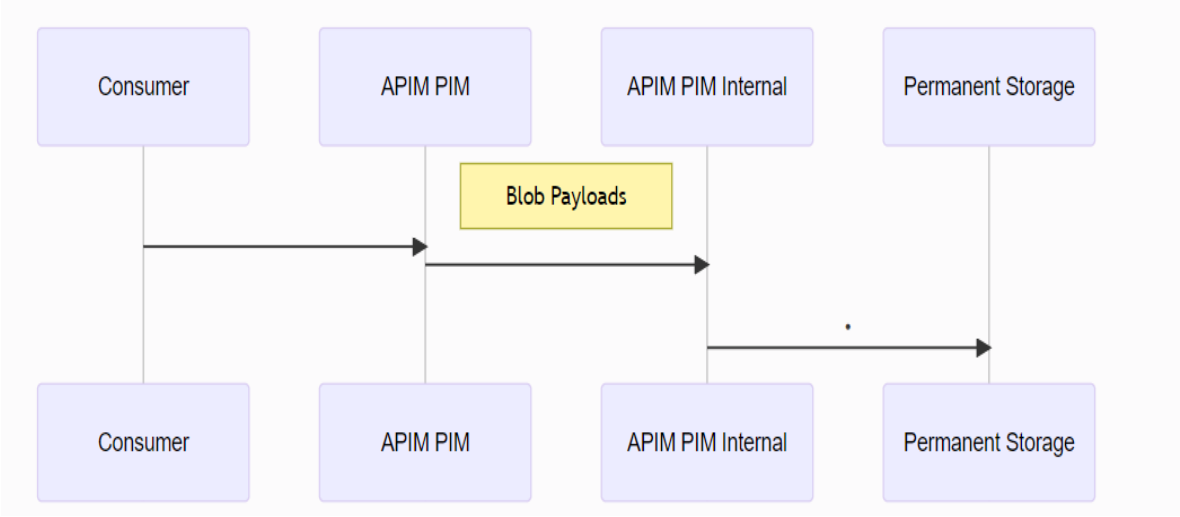
GET Methode

Eine GET Methode wird benutzt, um Daten eines bestimmten Speichers zu erhalten und zu lesen.

IST Zustand:



SOLL Zustand:

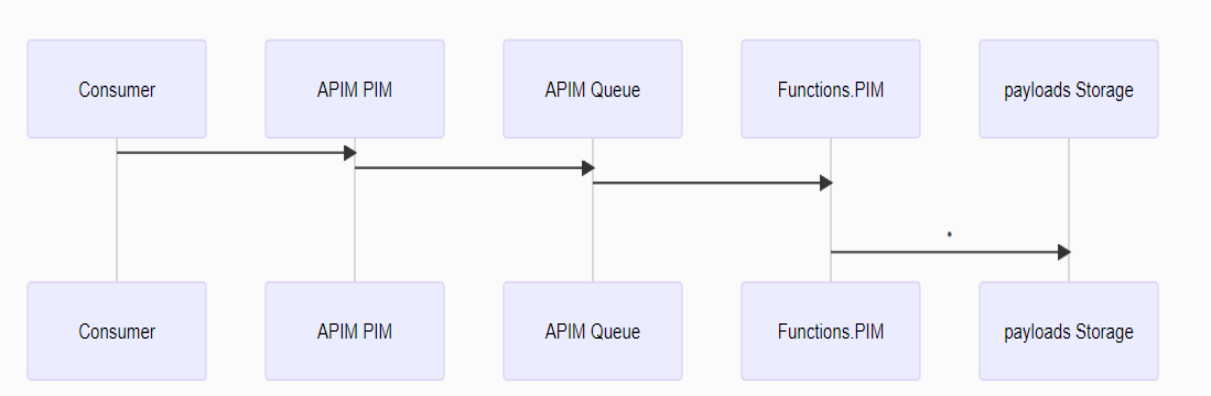


Diese Grafiken werden in dem Projekt für jede API erstellt, damit Kunde und Entwickler genau wissen, wie eine API korrekt läuft. Somit kann bei Anpassungen das zuständige Projektmitglied den Datenfluss komplett verstehen. Dies ist sehr wichtig, um die Änderungen korrekt durchzuführen, ohne dass dabei auf der Ebene Production etwas nicht korrekt funktioniert. Dies würde viel Geld Kosten, da sich der Entwickler nochmals kümmern muss um dieses Problem.

POST Methode

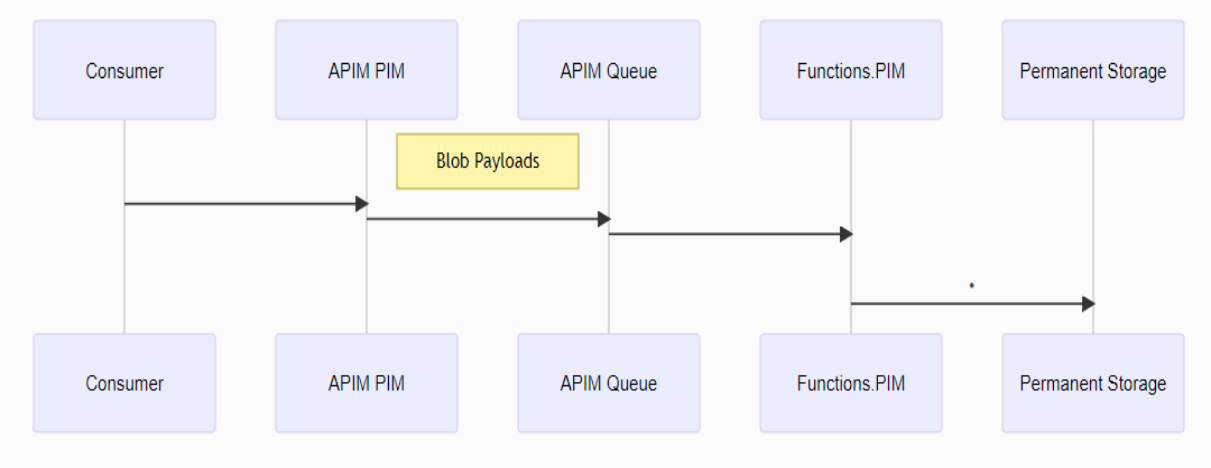
Eine POST Methode wird benutzt, um Daten zu erstellen oder zu ändern. Früher wurde diese Methode nur für das Ändern und Anlegen von Daten in den payload Blob verwendet.

IST Zustand:



SOLL Zustand:

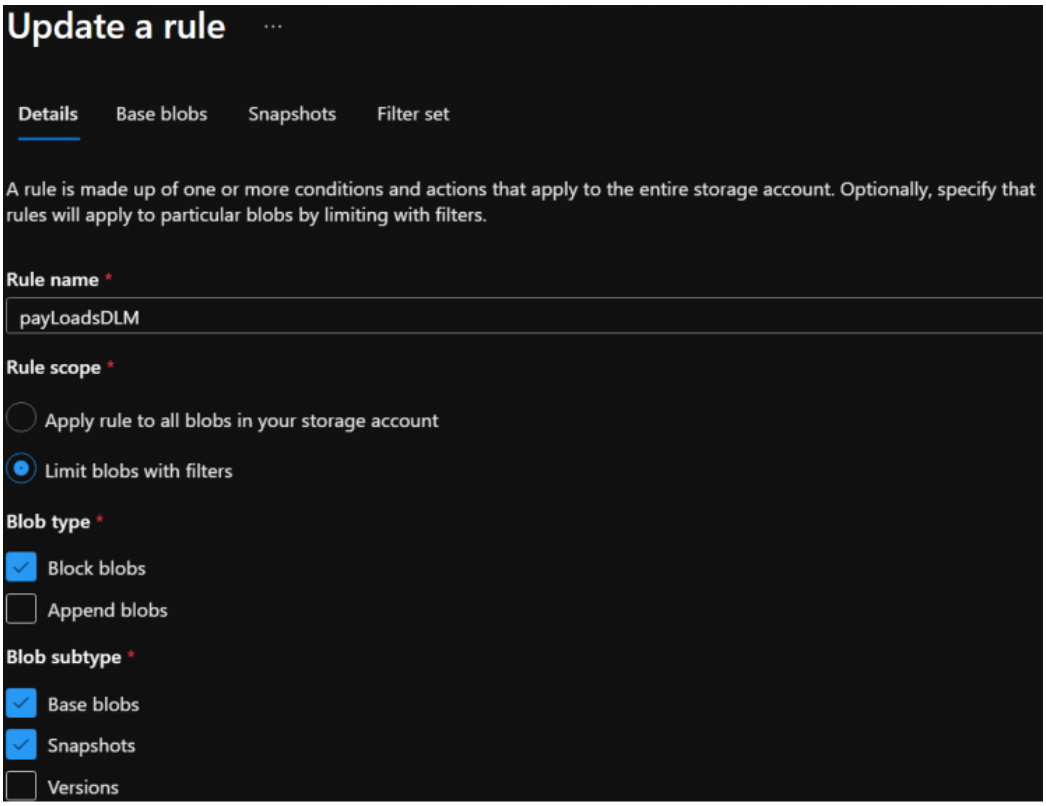
Nun werden die Daten auch im „permanentStorage“ gespeichert.



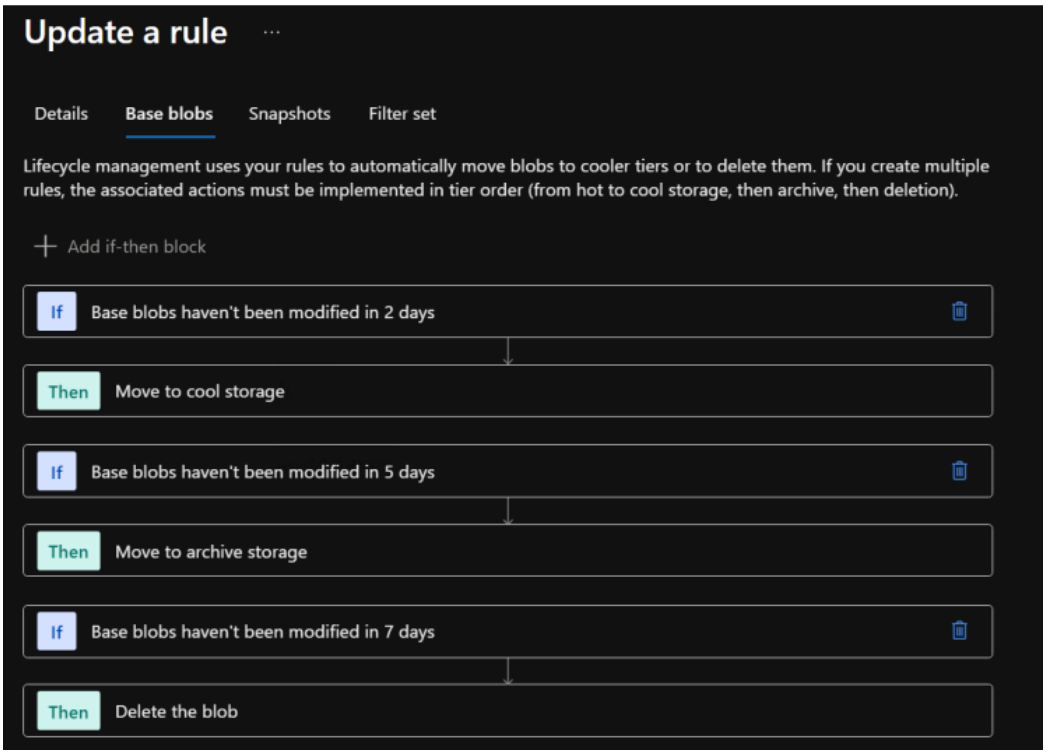
In dieser Abbildung wird demnach auch klar, dass die Daten erstmal in den Blob Storage Payloads geleitet werden und am Ende dann die relevanten Daten in den neu angelegten Speicher „permanentStorage“ landen.

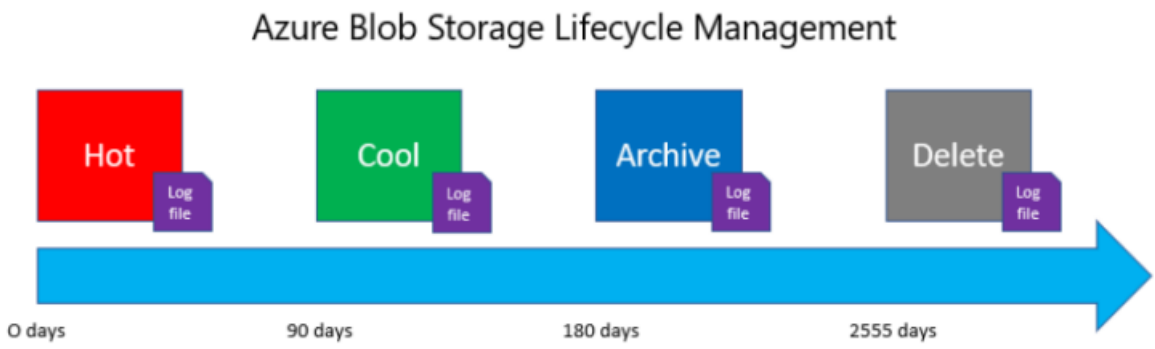
2.4 Lifecycle Management

Das Lifecylce Management ist eine Funktionalität von Blob Storage. Dieses Management zeigt an, welche Regeln für die einzelnen Blobs gelten. Man kann die Optionen je nach Bedarf ändern. Pro Blob können 100 Regeln festgelegt werden, welche automatisch durchgeführt werden und daher sehr hilfreich beim Verwalten von Blob Storages ist. Das Lifecycle Management ist ohne weitere Kosten zu nutzen. Anbei die Regeln des payload Storage.



Ausschnitt aus dem Azure Portal Lifecycle Management. Hier sieht man die Details des Speichers.





Quelle: https://medium.com/@raduvunvulea/azure-blob-storage-life-cycle-management-3ca54acd7977

3. Durchführung

3.1 Analysephase

In dieser Phase wurden die Kundenanforderungen und Wünsche an die Firma PlanB. analysiert und geprüft, ob diese Anforderungen umsetzbar sind. Nach der Überprüfung wurde die dazugehörige API analysiert und auf ihre Implementierung untersucht, um eine technische Lösung zu erstellen und mit dem Projektleiter diese Lösung zu besprechen. Während der Analysephase wurde ebenfalls ein Pflichtenheft erstellt, welches unter Punkt 3.2 zu sehen ist.

3.1.1 Beschreibung von Datenredundanz

Azure Storage bietet verschiedene Datenredundanzen, die sich preislich und auch an Speicherkapazität unterscheiden. Datenredundanzen sind wichtig, da man dadurch die Daten vor ungeplanten Ereignissen schützen kann, z.B. Ausfall des Netzwerkes durch Stromausfall.

3.1.2 Unterscheidung der verschiedenen SKUs

SKU steht für „Stock Keeping Unit“, was bedeutet, dass es ein virtuelles Lager gibt, in dem alle Daten auf verschiedene Arten gespeichert werden. Diese Informationen findet man auf der Internetseite von Microsoft Docs https://docs.microsoft.com/de-de/azure/storage/common/storage-redundancy?toc=/azure/storage/blobs/toc.json .

Unter den SKUs gibt es vier unterschiedliche Replikationen (Bildung von Datenkopien):

LRS = Locally redundant storage

Daten werden repliziert und speichert drei Kopien in sogenannte Fehlerdomänen in das Rechenzentrum einer Region.

LRS ist am kostengünstigsten, jedoch hat dies auch ein geringeres Maß an Dauerhaftigkeit der Daten, d.h. wenn das Rechenzentrum ausfällt aufgrund eines Stromausfalles, dann gehen alle Replikate verloren und können nicht mehr wiederhergestellt werden. LRS ist nur dann eine gute Wahl, wenn Daten leicht manuell wiederherzustellen sind.

ZRS = Zone redundant storage

Daten werden synchron, also gleichzeitig, über drei Azure-Verfügbarkeitszonen (getrennter physischer Standort, mit unabhängiger Energieversorgung und Netzwerk) in der primären Region repliziert. Ein Jahr werden die Daten im ZRS Modus gespeichert und bietet eine Dauerhaftigkeit von mindestens 99,99999%.

Auf zonenredundanten Daten kann man Lese- und Schreibvorgänge durchführen, auch wenn eine Azure – Zone nicht mehr verfügbar ist, da sie ja noch in den anderen beiden Zonen verfügbar sind.

Wenn man eine Datei im Speicher ändern möchte, dann erfolgt dies synchron, d.h. die Schreibanforderungen werden erst dann zurückgegeben, wenn die Daten in allen drei Verfügbarkeitszonen vorhanden sind.

Funktionsweise bei Ausfall einer Zone:

Azure Netzwerkupdates werden durchgeführt, z.b. Festlegung neuer DNS-Ziele (DNS= Domain Name System: verknüpft URLs mit kryptischen IP-Adressen), welche sich auf die Anwendung auswirken können, wenn man auf den Speicher zugreifen will, obwohl das Update noch nicht komplett durchgelaufen ist.

Deswegen ist es auch sinnvoll, dass der Entwickler die Wiederholungsrichtlinien implementiert, bevor man den ZRS Speicher benutzt, damit man zusätzlich ein Backup (Kopien) hat.

GRS = Geo redundant storage / RA-GRS = Read-access geo redundant storage

Bei georedundanten Speichern werden Daten gleichzeitig dreimal in einem Standort in Region A mit dem Prinzip von LRS kopiert und danach werden die Daten asynchron (nicht gleichzeitig) in die Region B kopiert. Sobald der Nutzer schreiben möchte, wird zuerst die Änderung am Speicherort von Region A gespeichert und dann mit LRS repliziert. Anschließend werden die Änderungen asynchron in die Region B geschrieben, welche dann wieder mit LRS repliziert werden. Zusammengefasst gibt es sechs Kopien in Region A und B.

Funktionsweise bei Ausfall einer Zone:

Die Azure Storages werden in Region A und B automatisch miteinander verbunden und Region B wird bei Ausfall von Region A primär.

GZRS = Geo zone redundant storage / RA-GZRS = Read-access geo redundant storage

Diese Redundanzstufe ist eine Kombination aus ZRS und GRS. Daten sind auch dann geschützt, wenn eine Region komplett ausfällt.

Funktionsweise bei Ausfall:

Daten werden nur dann lesbar, wenn der Nutzer ein Failover von Region A zu Region B initiiert. Bei einem Failover wird der sekundäre Endpunkt so aktualisiert, dass dieser zum primären Endpunkt für das Speicherkonto wird.

3.1.3 Kosten des Azure Blob Storages

Datenübertragungskosten für die Georeplikation

Diese Kosten gibt es nur für Speicherkonten mit konfigurierter Georeplikation und wird pro Gigabyte verbucht.

Datenzugriffskosten

Die Preise für einen Datenzugriff ist höher, bei einer „cool“ Ebene. Dies bedeutet, dass die Daten dieser Ebene nicht immer benutzt werden und mindestens 30 Tage archiviert werden.

Frühzeitiges Löschen von Kalt- und Archivspeicher

Wenn man den Blob Storage frühzeitig löscht, also weniger als 180 Tage nutzt, dann fällt eine extra Gebühr an. Aber wenn der Blob Storage mindestens 180 Tage genutzt wird, dann werden natürlich dem Kunden keine Gebühren gestellt für das Löschen.

Beispiel

Blob wird nach 45 Tagen von der Archivebene gelöscht oder verschoben, dann wird für die 135 Tage die Gebühr berechnet

=> **180 – 45 = 135 Tage.**

3.2 Pflichtenheft

Musskriterien

Einbindung der App in die ESB Infrastruktur:

Die Richtlinien müssen mit dem „Product Owner“ besprochen werden.

Ressourcen müssen auf der ESB Umgebung verwaltet werden.

Anpassen der dazugehörigen API:

Prüfen, ob die Daten in den korrekten Storage Account gelangen.

Anpassen der dazugehörigen Azure Function:
Umbenennung des connection strings von „payloadStorage“ zu „permanentStorage“ = neu hinzugefügter Speicher.

Anpassen des Service Bus Triggers (ausgelöst, sobald eine Nachricht von der „Service Bus Queue“ ankommt)

Wunschkriterien

Wunschkriterien wurden nicht erfasst, da der Kunde keine hatte.

Abgrenzung

Ein Benutzerhandbuch gibt es nicht, nur eine technische Dokumentation. Die Anpassung der Blob Storages wurde nach Fertigstellung sofort übernommen, d.h alle Verbesserungen sind sofort auf der Bereitstellungsumgebung Production gewesen. Production bedeutet, dass die Entwicklung in vollem Umfang vom Kunden genutzt wird. In dieser Phase dürfen keine Fehler mehr aufkommen und der Kunde hat die Verantwortung, dass die Applikation auch nur dann genehmigt wird, wenn diese auch wirklich funktioniert. Aus dem Grund muss der PO erstmal die Applikation für sich selbst testen. Wenn er damit einverstanden ist, wird die Applikation auf Production Ebene genehmigt.

3.3 Bereitstellungsumgebungen innerhalb des Projektes

Für das strukturierte Arbeiten im Projekt sind Bereitstellungsumgebungen implementiert, welche sich im Azure DevOps befinden.

DEV ist die Entwicklungsumgebung (englisch development environment). Die Softwareanforderungen werden in dieser Umgebung umgesetzt.

TEST ist die Testumgebung, in der automatisierte Softwaretests ausgeführt werden.

STAGE stellt die Entwicklung dem PO zur Verfügung für die Abnahme. Diese Umgebung muss möglichst der Produktivumgebung entsprechen.

Production ist die Umgebung für den eigentlichen Einsatzzweck. Die Applikation wird in vollem Umsatz vom Kunden verwendet, welche mittels Logging, Monitoring und Auditing stets überwacht wird.

Ausschnitt der Bereitstellungsumgebungen, die jede Entwicklung durchläuft. NL steht für das Gebiet Niederlande und IE für Irland.

3.4 Recherche

Zuerst begann die Recherche, wie man die Kosteneinsparung der Blob Storages umsetzen kann. Es gibt einen sehr interessanten online Artikel von Microsoft. In diesem Artikel geht es um die Kostenzusammensetzung von verschiedenen Datenredundanzen innerhalb von Blob Storages. Wenn Daten so verwaltet werden, dass es zum bestehenden Speicher auch einen anderen noch gibt, der auf einer anderen Redundanz arbeitet, dann wird dies die Kosten enorm senken. Vor der Implementierung wurde nämlich viel mehr Geld ausgegeben als nötig.

3.5 Implementierung

In der Implementationsphase wurde die Anpassung der Azure Function, sowie die API im Azure API Management angepasst. Es musste sichergestellt werden, dass die API mit der Azure Function und den beiden Blob Storages (payload und permanent) problemlos interagiert.

API Management

Das API Management von Azure ist da, um APIs zu kreieren. Es hilft Organisationen ihre Schnittstellen nach außen an ihre Kunden bereitzustellen oder aber auch für die Organisation innerhalb. Jede API enthält eine oder mehrere Operationen, welche immer wieder erweitert werden können, z.B. eine weitere Operation hinzufügen.

Einige Szenarios für das Benutzen vom API Management sind hier beschrieben, welche auch unter der Internetseite https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts zu lesen sind:

- Sichere mobile Infrastruktur

- Internes API Programm
  Innerhalb der Firma wird über eine zentrale Lage die Kommunikation durchgeführt, speziell für die neuesten Änderungen der API. Dadurch kann die API an jedem Standort aufgerufen werden ohne Einschränkungen.

- Einfaches Aufnehmen von Entwicklungspartner
  APIM bietet ein einfaches Entkoppeln von internen Implementierungen anderer Firmen. So kann eine Firma ganz leicht zusammen mit einer anderen Firma, die schon das APIM besitzt, arbeiten.

Anbei noch die dazugehörigen Komponenten des APIMs:

API gateway	Akzeptiert alle API Anfragen und leitet diese weiter an die zuständigen Endpunkte
	Verifiziert API Schlüssel, Zertifikate und andere Zugangsarten
	Protokollieren von Metadaten for analytische Zwecke im App-Insights
Azure Portal	Definiert oder importiert API Schemas
	Erstellen von Richtlinien für die API
	Verwalten der Nutzer
Developer Portal	Lesen von API Dokumentationen
	Testen einer API über die interaktive Konsole
	Erstellen eines Kontos und Abos um API Schlüssel zu erhalten
	Zugriff auf Analytik, zum Überprüfen des Datenverbrauchs

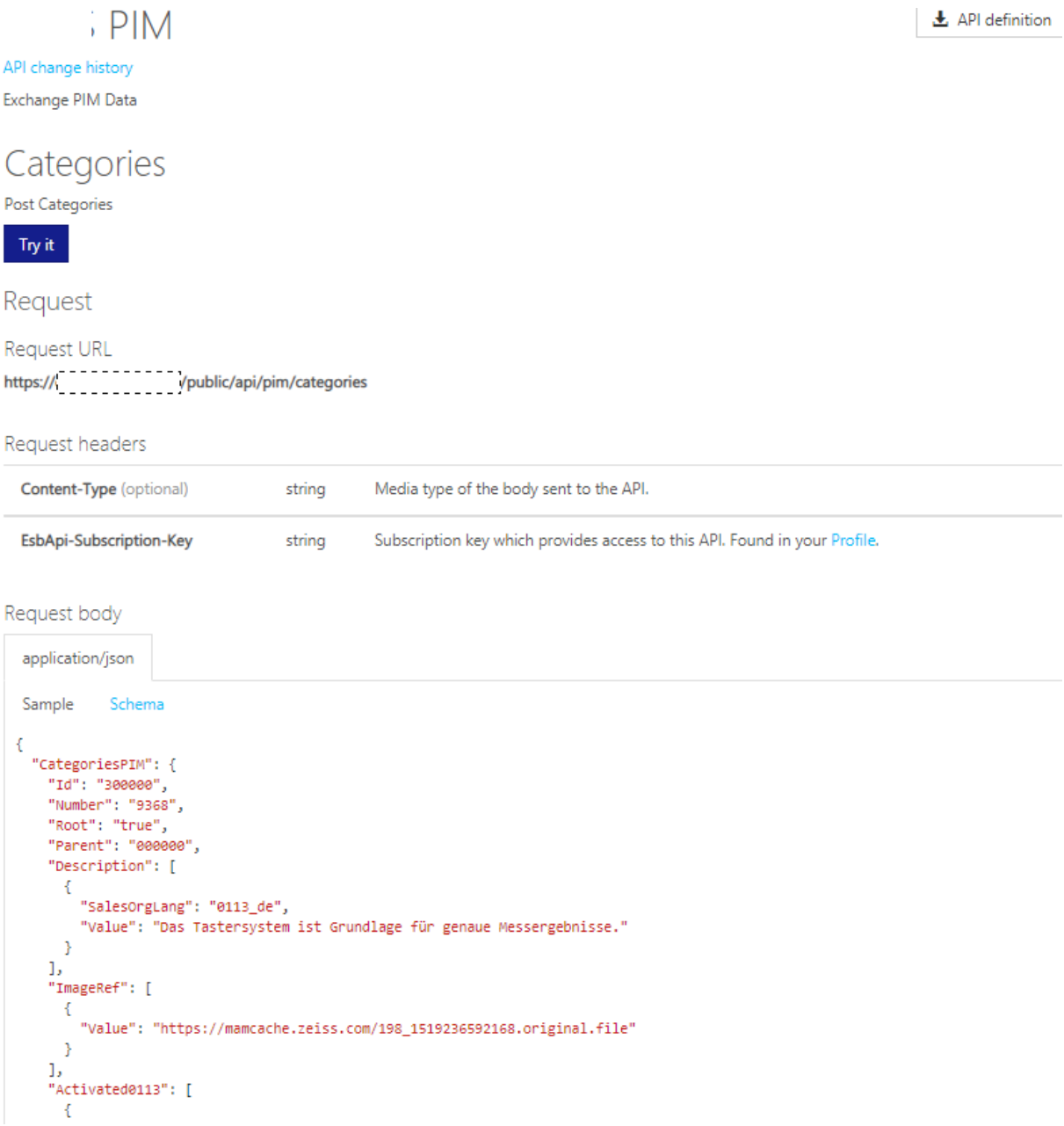
Im API Management werden Richtlinien festgelegt, wie die API zu funktionieren hat. Speziell in der Abteilung „API Management Policies“ (Regeln). Die verschiedenen Funktionen sind die Datenspeicherung in die Blob Storages und die Prüfung der Daten. Diese Regeln wurden mit XML und C# festgelegt. In der spezifischen API Policy wurde auch die „queue“ vermerkt. Die Queue ist eine Leitung zwischen einem Anfangsort (von wo aus die Daten gesendet werden) zu einem Endziel

(Speicherung der Daten) für eine API.

```
<rewrite-uri template="{{ESB_APIM_QUEUING_INTERNAL_OUTBOUND_PATH}}apim/category?queue=q.global.pim.category.storage" />
<set-body>@{
    return context.Request.Body.As<string>();
}</set-body>
```

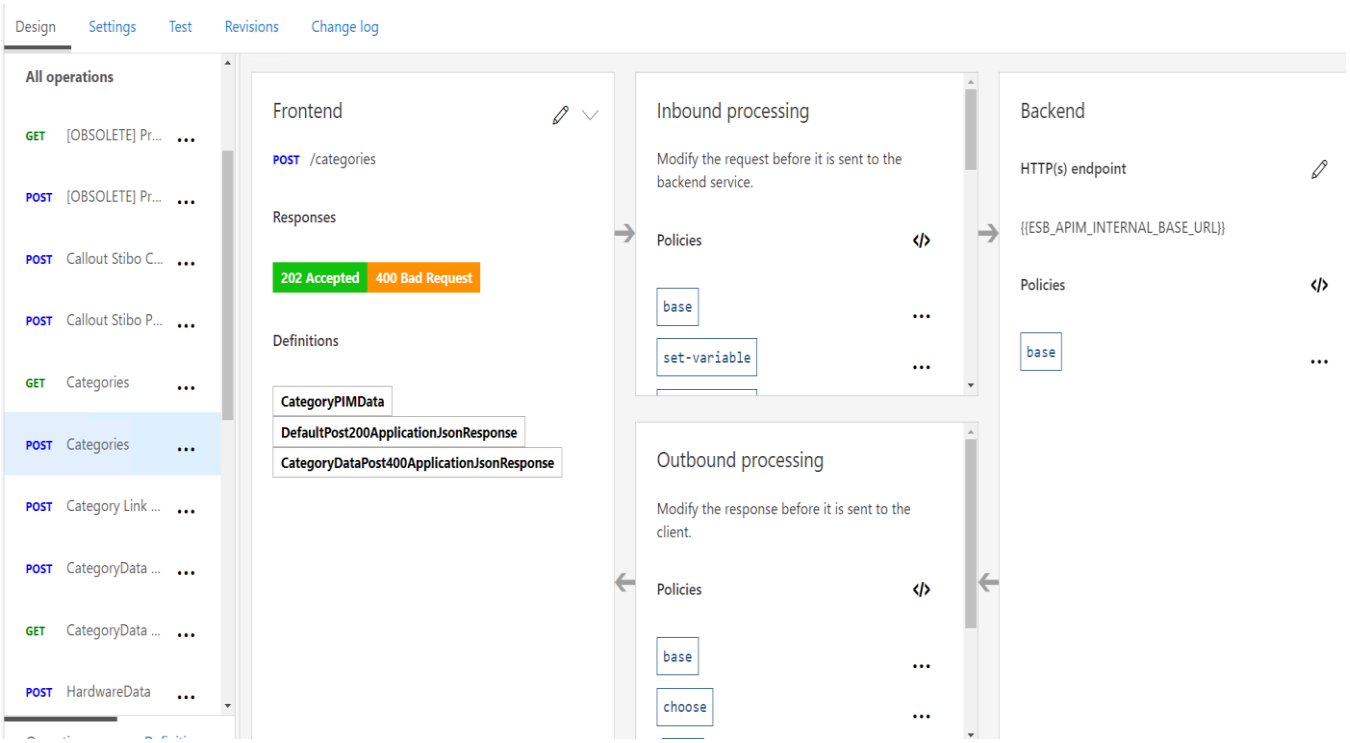
Ausschnitt aus dem API Management unter Richtlininen mit der queue.

Im Developer Portal kann man eine API und Funktion testen, indem man einen „Request Body“ im JSON Format absendet. Dazu ist es auch wichtig eine GET – oder POST Methode zu selektieren, je nach Absicht. Dieses Portal wird von Azure automatisch zur Verfügung gestellt, beim Buchen des APIM.



Ausschnitt des Developer Portals. Hier wird mit der POST Categories Methode getestet.

Anbei noch ein Ausschnitt des API Managements:



Ganz links sind alle Operationen innerhalb der PIM API verwaltet.

Azure Function

Die Azure Function ist eine weitere Technologie, die für die Implementation der Thematik genutzt wurde. Diese sendet ständig Updates, um immer auf dem neuesten Stand der Infrastruktur und Ressourcen zu sein, um problemlos Applikationen auszuführen.

Die Technologie ist eine containerbasierte Konsolenanwendung, die nach festgelegten Ereignissen einen Durchlauf startet und den programmierten Code durchführt. Die Azure Function wurde in C# programmiert und ist mit den Blob Storages verbunden, sowie mit der API im API Management, die auf Daten zugreifen kann und auch diese Daten speichert.

Die Anwendungsphase meiner Projektarbeit ist demnach auch der aufwendigste Teil.

Am Anfang wurde die Umbenennung des neuen Blob Storages gekümmert innerhalb der Azure Funktion vorgenommen, indem man den originalen Stand vom zuständigen Repository (Verwahrungsort) im Azure DevOps kopiert habe und lokal so bearbeiten konnte. Die Funktion war an sich schon fertig, jedoch musste die Anpassung separat erfolgen, damit es keine Probleme gibt beim Ausführen der Applikation der Endnutzer. Der zusätzliche Blob Storage wurde „permanentStorage“ genannt. Der bestehende Blob Storage „payloadStorage“ bleibt weiterhin

bestehen. Diese Namen sind einfach nur Variablen, die auf die Blob Storages verweisen. Die Umbenennung der Namen erfolgte in jeder .cs Klasse in der Funktion, denn jede Operation hat eine Klasse, z.b. Category, HardwareData oder MachineCatalogData. Das heißt innerhalb der Funktion müssen Anpassungen für jede Operation stattfinden. Insgesamt sind es zehn Operationen, die bearbeitet werden müssen.

```
1 reference
public static async Task PushCategory(
[ServiceBusTrigger("q.global.pim.category.storage", Connection = "esbreceive")] Microsoft.Azure.ServiceBus.Message messageHeader,
[Blob("payloads/category/{MessageId}", FileAccess.Read, Connection = "payloadStorage")] string payload,
[Blob("pim", Connection = "permanentStorageAccount")] CloudBlobContainer cloudBlobContainer,
ILogger log)
```

Ausschnitt der Azure Function. Benutzen Blob „permanentStorageAccount“ und von CloudBlobContainer.

3.5.1 ARM Vorlage

Die ARM Vorlage (Azure Ressource Manager Vorlage) wurde für das Erstellen des neuen Azure Blob Storages verwendet.

Diese Vorlage hilft Entwicklern eine neue Infrastruktur in Form von Code in Azure zu implementieren. Das Ausführen der ARM Vorlage kann immer wieder gestartet werden, wenn man die Infrastruktur auf eine andere Umgebung innerhalb Azure bringen möchte.

Zuerst wird der Container angelegt, welcher sich in dem Storage Account befindet. Ein Container ist ein Unterordner innerhalb eines Speichers. Dazu muss man den Datentyp angeben, sowie den Name des Containers. Danach wird der „permanentStorageAccount“ erstellt innerhalb der ARM Vorlage.

```
"containerName": {
  "type": "string",
  "defaultValue": "pim"
},
"permanentStorageAccount": {
  "type": "string",
  "defaultValue": "[redacted]"
}
```

Ausschnitt aus dem ARM template, innerhalb der Azure Function.

Innerhalb des Ressource Abschnittes wird die SKU angelegt (wie in 3.1.2 „Unterscheidung der verschiedenen SKUs“). Außerdem wird in diesem Abschnitt die apiVersion, type (Typ des Speicher, hier storageAccount), name (Name des Speichers) und die location (wo sich der Speicher in Azure befindet).

```
, "resources": [
 {
  "apiVersion": "[providers('Microsoft.Storage','storageAccounts').apiVersions[0]]",
  "type": "Microsoft.Storage/storageAccounts",
  "name": "[parameters('storageName')]",
  "location": "[resourceGroup().location]",
  "sku": {
   "name": "Standard_RAGRS"
  },
```

Ausschnitt aus der ARM Vorlage, Teil Resources, wie oben beschrieben.

Um den Azure Blob Storage in die richtige Azure Umgebung zu bringen, muss der Endpunkt (DefaultEndpoint) in der ARM Vorlage festgelegt werden. Dieser Endpoint bekommt der Entwickler direkt vom Blob Storage Abteil.

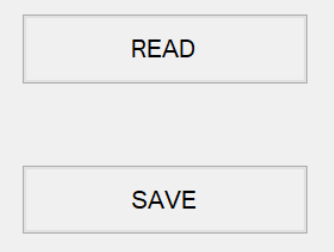
```
"StorageAccount": {
 "type": "string",
 "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',parameters('storageName'),';AccountK
},
```

Ausschnitt aus der ARM Vorlage, mit dem Angeben des Endpunktes unter Punkt „value“.

3.5.2 Frontend der API erstellen

Das Erstellen des Frontends für die API habe wurde mit Windows Forms .NET Framework umgesetzt. Windows Forms .NET Framework ist ein GUI-Toolkit (Werkzeugkasten für grafische Benutzeroberflächen) innerhalb Visual Studio, die man beim Erstellen der Applikation auswählt.

Die verwendeten Steuerelemente sind zum einen Buttons, welche ich für das Auswählen der Operation verwendete. Insgesamt habe ich dafür zwei Buttons eingesetzt. Einmal für die GET Operation (Daten holen) und einmal für die POST Operation (Daten bearbeiten).



Ausschnitt aus der Windows Forms Applikation mit den Buttons READ und SAVE.

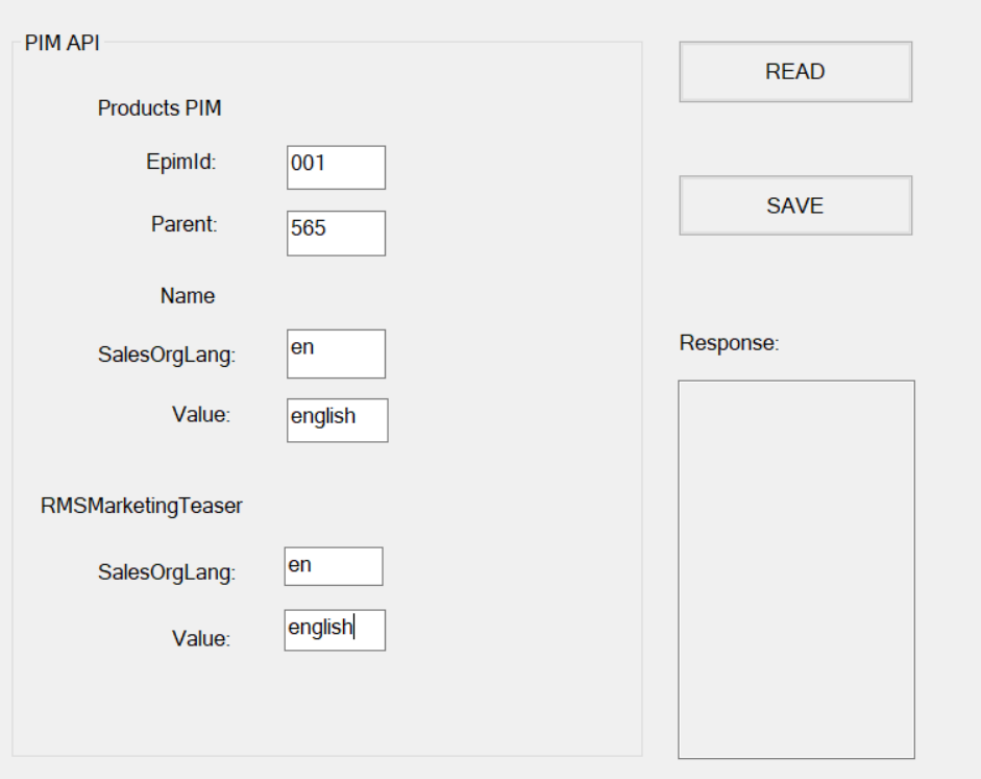
Die verwendeten Daten sind praxisnahe Daten aus dem Azure Blob Storage, welcher Kundeninformationen enthält.

Das Aufrufen der API wurde mit der Programmiersprache C# umgesetzt.

Die Logik besteht darin, Daten aus dem Azure Blob Storage zu erhalten, sowie Daten zu verändern, mit den vorgegebenen Parametern.

Bei diesem Ausschnitt werden zwei Arrays gebildet mit dem Namen „Name“ und dem Name „RMSMarketingTeaserShort“. Die dazugehörigen Parameter lauten „SalesOrgLang“ und „Value“ bei beiden Arrays. In den Textboxen kann der Nutzer Werte als Datentyp Zeichenkette (string) eingeben, welche dann in den Blob Storage gesendet werden.

Anbei das komplette Frontend:



3.5.3 Datenmigration

Die Datenmigration wurde mit „azCopy“ durchgeführt, welches eine Konsolenanwendung ist. Dieses kann von einer Konsole gestartet werden. Laut online Doku gibt es für die unterschiedlichen Betriebssysteme auch unterschiedliche Versionen dieser Konsolenanwendung. Es ist mit vielen Konsolen möglich, dies auszuführen.

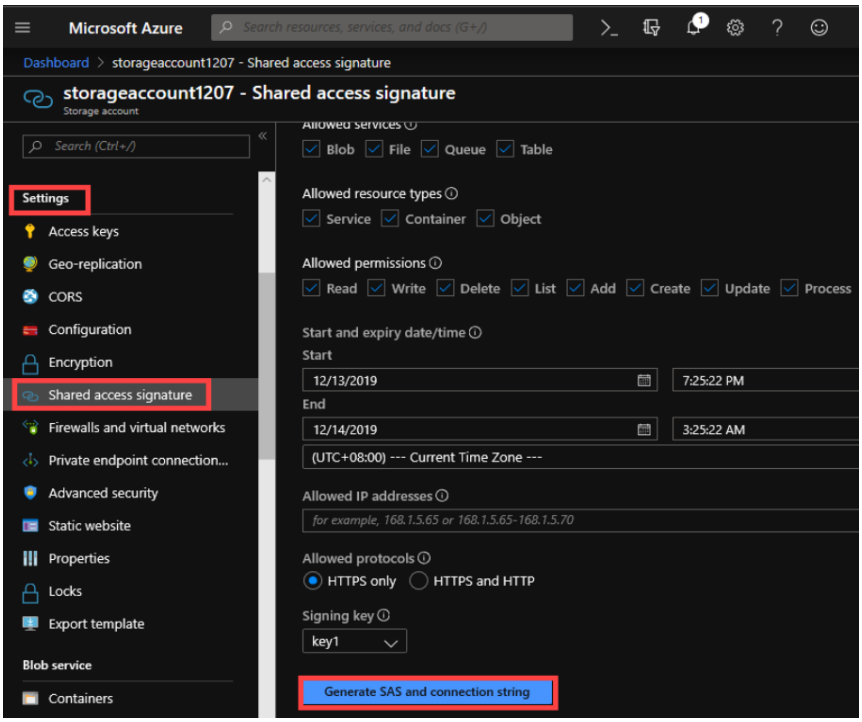
Ich habe mich für „azCopy“ entschieden, da dies der sicherste und einfachste Weg ist, Daten zu migrieren. Manuelles Migrieren wäre eine sehr schlechte Idee, da man tausende von Dateien kopieren und einfügen muss, welches unglaublich viel Zeit benötigen wird und man als Entwickler somit auch den Überblick verliert, welche Daten schon kopiert wurden und welche noch nicht.

Diese muss als Administrator ausgeführt werden, sonst wird man nicht genug Berechtigungen erhalten, um „azCopy“ zu benutzen.

Anschließend muss man sich über die PowerShell bei Azure anmelden. Dabei gibt man seine Anmeldedaten ein, sowie die Tenant ID (ID einer Organisation, zu dem der Speicher gehört).

Was man auch noch benötigt ist der SAS-Token (Shared Access Signature, zu deutsch Signatur für gemeinsamen Zugriff). Der SAS-Token ist eine Zeichenfolge, die auf der Clientseite generiert wird in Azure. Dieser kann aus Sicherheitsgründen nicht verfolgt werden und es können eine unbegrenzte Anzahl an SAS-Token erstellt werden.

So sieht die Seite für das Erstellen des SAS Token aus in Azure:



Nach Ausführen des Befehls wurde kontrolliert, ob die Daten im neuen Blob Storage „permanentStorage“ angekommen sind. Daran erkenne ich, dass die Datenmigration erfolgreich war. Es wird also nur die korrekte Syntax benötigt und trägt die korrekten Werte in die Syntax ein. Innerhalb von Sekunden sind dann alle Dateien migriert.

Die Syntax für den kompletten Befehl sieht so aus:

```
azcopy copy 'https://<source-storage-account-name>.blob.core.windows.net/<container-name>/<blob-path><SAS-token>' 'https://<destination-storage-account-name>.blob.core.windows.net/<container-name>/<blob-path>'
```

3.6 Technologien und SDKs

Die Entwicklungen des ESB Projekts werden immer in einer virtuellen Maschine durchgeführt. Eine virtuelle Maschine hat dieselben Funktionen wie physische Computer und führen Anwendungen und

ein Betriebssystem aus. Das Team benutzt den Network Client von AT&T, welcher ein VPN Client ist (Virtuelles Privates Netzwerk). Ein VPN ist ein Netzwerk, das eine Telekommunikationsstruktur wie das Internet verwendet.

Über das Azure DevTestLab wird die virtuelle Maschine gestartet. Jedes Projektmitglied hat seinen eigenen Zugang, mit dem er arbeitet. Dies hat den Vorteil, dass man sich in einer isolierten Umgebung befindet, in der man an Anwendungen oder Prozessen arbeiten kann, ohne das eigentliche Host-System zu schädigen. Dadurch wir außerdem gewährleistet, dass bei einem Absturz der virtuellen Maschine das Host-System nicht mit betroffen ist.

Mit dem Azure Service „API Management“ wurde in der Cloud gearbeitet. API Management bietet eine Lösung zum Erstellen und Konfigurieren von APIs mit hoher Sicherheit vor Datenmissbrauch. Diese APIs werden mit der Entwicklersprache C# und XML bearbeitet.

Es wurde auch noch eine Azure Function verwendet, die in Verbindung mit der API und den dazugehörigen Blob Storages steht. Die Azure Function wird im Visual Studio programmiert und ist eine containerbasierte Konsolenanwendung. Diese Technologie benutzt einige SDKs, die mit „using“ deklariert werden. Englisch bedeutet SDK „Software Development Kit“, was zu deutsch „Software Entwickler Kasten“ heißt.

Anbei die genutzten SDKs für die Projektarbeit:

Microsoft.Azure.ServiceBus

Cloud Speicheroptimierung

Microsoft.Azure.WebJobs

Newtonsoft.Json

Newtonsoft.Json.Linq

Functions.PIM.Extensions

Microsoft.WindowsAzure.Storage.Blob

Microsoft.VisualStudio.TestTools.UnitTesting

Moq

3.7 Testphase

Mit Hilfe von Integration Tests und Unit Test wurden Fehler in der Applikation ermittelt. Sobald man einen Fehler findet, wird dieser sofort angepasst und durch Speichern gesichert.

Unit Tests

Diese Tests überprüfen, ob die Komponenten so arbeiten, wie es sein soll. Es werden die Unit Test immer in der Azure Function angelegt. Für die Übersichtlichkeit ist es am besten eine eigene Klasse für diese Tests zu erstellen.

Integration Tests

Können Probleme mit den Schnittstellen zwischen Programmkomponenten aufdecken. (näher beschrieben bei Punkt 3.9 Qualitätssicherung) zuständig.

3.8 Dokumentation

Die Dokumentation beinhaltet eine Übersicht der Entwicklung, sowie durchgeführte Anpassungen. Dies wird alles auf technischer Ebene dokumentiert, da der Kunde selbst im Projekt mitarbeitet und sich mit den Technologien bestens auskennt.

3.9 Vorgehensweise

Das Projekt wurde mit dem Kanban Projektmanagement durchgeführt und die Aufgaben nach Wichtigkeit verteilt. Besonders wichtig sind „Bugs“, die Fehler in der entwickelten Applikation sind und diese müssen sofort behoben werden, damit der Fehler nicht das ganze angehörige System lahm legt. Das kann dazu führen, dass der Kunde seine Arbeit nicht mehr fortsetzen kann und Geld verliert.

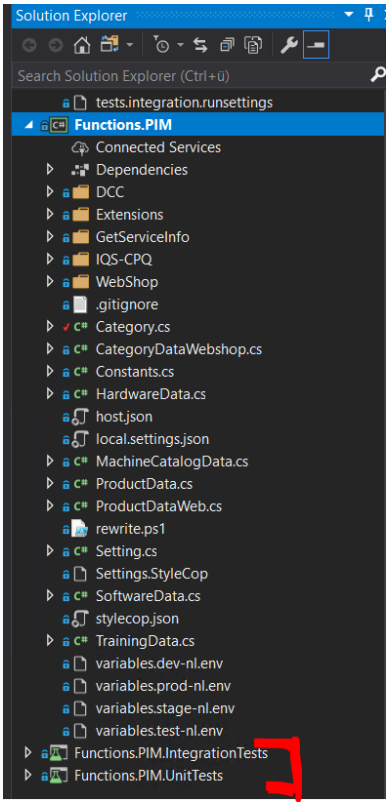
3.10 Qualitätssicherung

Für das Sicherstellen der Qualität nach der Implementation und dem Testen, wurde eine technische Dokumentation verfasst. Der Grund für die technische Dokumentation ist, dass der Product Owner

diese bei Problemen sich anschauen kann, um nachvollziehen zu können, an was es liegen könnte, dass die Applikation nicht funktioniert. Die Dokumentationen werden im Projekt immer im Wiki des Azure DevOps verwahrt.

Man muss auch wissen, was der Entwickler da implementiert hat, um den Zweck der Entwicklung zu erfassen.

Eine weitere Art der Qualitätssicherung war die Verwendung von Unit- und Integration Tests. Diese sollen sicherstellen, dass der Code ohne Fehler funktioniert. Die Tests werden alle in der Azure Function vermerkt und sind abgegrenzt von dem restlichen Quellcode, d.h. eine extra Datei für die Unit Tests und eine extra Datei für die Integration Tests.



Ausschnitt aus dem VS Solution Explorer.

Rot markiert die Projekte für die Unit Tests und Integration Tests, die hier Functions.PIM.IntegrationsTest und Functions.PIM.UnitTests heißen. In den beiden Projekten gibt es mehrere Dateien als Unit- und Integration Tests.

Nach diesen beiden Testarten wurde dann manuell über den Azure Dienst namens „Developer Portal“ getestet. Näher beschrieben unter 3.5 Implemetierung APIM.

Die Request URL ist schon automatisch vorhanden, d.h. beim Testen muss dieser nicht extra hinzugefügt werden. Beim Klick auf den Sende Knopf erhält der Nutzer eine Antwort, welche vom Azure Portal kommt. Entweder bekommt man einen erfolgreichen Code (200, 201, 202) zurück oder einen fehlerhaften Code (400, 401, 500).

3.11 Anpassungen und Entscheidungen

Die Anpassungen wurden zum einen im Azure API Management vorgenommen. Die Policy der bestehenden API und die Named Values (Schnittstellenvariablen, die im Azure Portal angelegt werden müssen).

Und zum anderen wurde die schon bestehende Azure Funktion angepasst, damit die Daten in den neu angelegten Blob Storage gespeichert werden. Dazu wurde ein eigener Branch erstellt, damit man die bestehende Funktion nicht kaputt macht, die von einem anderen Entwickler erstellt wurde. Ein Branch ist ein Ast, der dazu da ist, eine bestehende Funktion zu duplizieren, ohne dass der produktive Code geändert wird.

Man hat dann die Originalversion auf dem „Master Branch“ oder „Main Branch“ für alle verfügbar und die duplizierte Form ist die Version zum Anpassen.

4. Projektergebnis

4.1 Ist – Soll – Analyse

Anforderung	Entwicklung	Akzeptanz
Named Values (Variablen) hinzufügen	Im Azure Portal angelegt und im API Management unter Policies eingefügt	Akzeptiert
Richtlinien beachten	Entwickelt nach ESB-Richtlinien	Akzeptiert
Anlegen des zweiten Blob Storages	Blob erfolgreich im Azure Portal angelegt	Akzeptiert
Testen der Applikation	Mit Unit- und Integration Tests in Azure Function die Applikation geprüft	Akzeptiert
Einzigartige ID generieren für jede Anfrage	Correlation ID implementiert in der Azure Function und im API Management	Akzeptiert
Anlegen der Ressourcen	Erstellt mit der ARM Vorlage	Akzeptiert
Technische Dokumentation	Eigenhändig erstellt	Akzeptiert

Die Entwicklung des Projektes warf keine Probleme auf. Nach dem Testen der Applikation konnte sichergestellt werden, dass alles einwandfrei läuft. Die Qualität ist damit gesichert und auch gewährleistet, dass der Kunde problemlos die Applikation nutzen kann. Die technische Dokumentation hilft außerdem auch den anderen Entwicklern, falls wieder etwas angepasst werden soll.

4.2 Wirtschaftlichkeitsanalyse

Ein Mitarbeiter des ESB Teams kostet pro Stunde ca. 93 €. Nach der Anpassung der Azure Infrastruktur, sowie Entlastungen des bestehenden Blob Storages waren die Kosten für die Speicherbelegung um einiges geringer, da die Daten sich auf zwei verschiedenen Redundanzebenen befinden. Redundanzebenen unterscheiden sich preislich, wie bei Punkt 3.1.3 beschrieben.

Rechenbeispiel Blob Storages:

Blob Storage A vor der Implementierung kostete pro Monat 0,261353 €

RA GRS Hot

500 MB Speichernutzung für 20 Tage	0,020665 €
20.000 Schreibvorgänge	0,23614 €

Cloud Speicheroptimierung

15.000 Lesevorgänge	0,00473 €

Blob Storage B nach der Implementierung kostet pro Monat 0,228 €

RA GZRS Hot

500 MB Speichernutzung für 20 Tage	0,02425 €
20.000 Schreibvorgänge	0,1982 €
15.000 Lesevorgänge	0,00555 €

Ersparnis Blobs: 0,033353€ / Monat

Rechenbeispiel Instandhaltung:

Entwickler kostet pro Stunde ca. 93 €

Aufwand vor der Implementierung pro Monat:	8 h * 93 € = 744€
Aufwand nach der Implementierung pro Monat:	3 h * 93 € = 279 €

Ersparnis: 465 € / Monat

⇨ **Ersparnis Insgesamt / Monat: 465,03€**

Die Kosten für die Instandhaltung reduzieren sich, da die Daten auf zwei Blob Storages verteilt sind, was eine übersichtlichere Lösung darstellt. Somit werden mit der Instandhaltung die meisten Kosten gespart.

4.3 Fazit

Die Speicheroptimierung spart dem Kunden eine Menge an Geld. Die Umsetzung hat sich somit nicht nur wirtschaftlich gelohnt, denn es wurde somit eine übersichtlichere Idee implementiert, die es den Entwicklern erleichtert die Speicher zu verwalten. Außerdem hat die PlanB. zusätzliches Vertrauen gewonnen, da der Kunde weiß, dass wir nur das Beste für ihn im Sinne haben, anstatt ihn einfach zahlen zu lassen. Ein Projekt sollte immer so umgesetzt werden, dass man als Entwickler auch dem Kunden Vorschläge macht, um ein besseres Ergebnis zu erhalten.

4.4 Abweichungen und Anpassungen

Abweichungen oder Anpassungen gab es während der Durchführung keine. Die Projektarbeit wurde so umgesetzt, wie dies am Anfang festgelegt und auch vom PO genehmigt wurde.

5. Anhang

5.1 Abkürzungsverzeichnis

Abkürzung	Bedeutung
PO	**Product Owner** Vertreter des Auftraggebers. Verantwortlich für das zu entwickelte Produkt.
API	**Application Interface** Schnittstelle zwischen zwei Programmen.
ESB	**Enterprise Service Bus** Projekt, das viele Schnittstellen innerhalb der Firma instand hält
HTML	**HyperText Markup Language** Eine textbasierte Sprache zum Strukturieren von Webseiten.
B2B	**Business to Business** Beziehung zwischen mehreren Unternehmen.
B2C	**Business to Customer** Beziehung zwischen Konsument und Unternehmen.
HTTP	**HyperText Transfer Protocol** Definiert, wie eine Webseite aufgebaut wird und regelt, wie diese Seite vom Server zum Client übertragen wird.
XML	**Extensible Markup Language** Textbasiertes Datenformat. Daten können in einem Texteditor bearbeitet werden und Computer können dieses Format lesen und beabreiten.
JSON	**JavaScript Object Notation** Datenformat für den Datenaustausch zwischen Apps.
Azure Funktionen; engl. Azure Functions	**Containerbasierte Konsolenanwendung** Arbeiten in der Azure Cloud und werden bei einem bestimmten Trigger gestartet.
REST API	**Representational State Transfer Application Programming Interface** Eine Schnittstelle, die Kommunikation im Internet beschreibt.
Production	**Produktionsumgebung** In dieser Umgebung ist die Entwicklung in vollem Einsatz. Keine Fehler dürfen auftreten.
ARM	**Azure Ressource Manager** Tool zum Bereitstellen von Services auf der Azure Cloud Plattform. Basiert auf dem JSON Format.
SAS-Token	**Shared Access Signature, zu deutsch Signatur für gemeinsamen Zugriff** Gibt an, wie der Client auf die Ressourcen zugreifen kann. (nur Lesezugriff, Lese – und Schreibzugriff)
Tenant ID	**Eindeutige ID** Definiert, unter welcher Azure Active Directory Instanz sich die Applikation befindet

Azure Active Directory	**Microsoft Cloud Dienst** Plattform zum Verwalten und Absichern von Identitäten
Repository	**Verwaltungsort für Code**

5.2 Detaillierte Zeitplanung

Aktion	Arbeitsschritt	Aufwand
1. Analyse der Anforderungen	1.1 Analyse der Anforderungen	2
	1.2 Analyse des jetzigen Standes	2
	1.3 Technische Lösungsfindung	7
	1.4 Erstellen von Pflichtenheft und Projektplan	4
2. Anlegen des Azure Speichers	2.1 Konfiguration von Azure Funktion	2,5
	2.2 Erstellen eines Azure Blob Storages	2,5
3. Erstellen des Frontends + Logik	3.1 Mit Windows Forms .NET Framework	15
4. Qualitätssicherung und Testen	4.1 Erstellen von Testplänen für die Integration Tests	5
	4.2 Erstellen von Unit Tests	5
	4.3 Ausführen von API und Azure Funktion	5
5. Datenmigration	5.1 Daten des bestehenden Blob Storage in den neu angelegten Blob Storage migrieren	5
6. Dokumentation	6.1 Erstellung der Dokumentation	
	6.1.1 Erstellen des Layouts	1
	6.1.2 Sammeln der Daten für die Doku	1
	6.1.3 Verfassen der Dokumentation	5
	6.2 Erstellung der technischen Dokumentation	3
7. Übergabe der Entwicklung und Dokumentation	7.1 Übergabe an Kunde	
	7.1.2 Besprechung mit dem Teamleiter/Betreuer	2
	7.1.3 Besprechung mit dem PO	3
Insgesamt		**70**

5.3 Auszug des Quellcodes der Azure Funktion

```
[FunctionName(nameof(PushCategory))]
1 reference
public static async Task PushCategory(
[ServiceBusTrigger("q.global.pim.category.storage", Connection = "esbreceive")] Microsoft.Azure.ServiceBus.Message messageHeader,
[Blob("payloads/category/{MessageId}", FileAccess.Read, Connection = "payloadStorage")] string payload,
[Blob("pim", Connection = "permanentStorageAccount")] CloudBlobContainer cloudBlobContainer,
ILogger log)
{
    if (messageHeader == null)
    {
        throw new ArgumentNullException(nameof(messageHeader));
    }

    if (cloudBlobContainer == null)
    {
        throw new ArgumentNullException(nameof(cloudBlobContainer));
    }

    var trace = new Dictionary<string, string>();
    var esbCorrelationId = Guid.Parse(messageHeader.CorrelationId ?? Guid.NewGuid().ToString());
    var methodName = MethodBase.GetCurrentMethod().Name;
    var eventId = new EventId(esbCorrelationId.GetHashCode(), Constants.EsbCorrelationTraceName);

    using (log.BeginScope("Method:{methodName} CorrelationId:{CorrelationId}", methodName, esbCorrelationId))
    {
        trace.Add(Constants.EsbCorrelationTraceName, esbCorrelationId.ToString());

        try
        {
            if (payload == null)
            {
                trace.Add("Skipped processing", "no Payload found for Category.");
                throw new ArgumentNullException(nameof(payload));
            }

            trace.Add("Category Payload", payload);

            var payloadObject = JObject.Parse(payload);

            var id = payloadObject["CategoriesPIM"][0]["Id"].Value<string>();
```

5.4 Auszug API Management Policies

```
PIM > Categories > Policies
<policies>
    <inbound>
        <base />
        <set-backend-service base-url="{{ESB_APIM_INTERNAL_BASE_URL}}" />
        <set-header name="EsbApi-Subscription-Key" exists-action="override">
            <value>{[redacted]}</value>
        </set-header>
        <set-variable name="categoryIdValue" value="@{
                string categoryid =  (string)context.Request.Url.Query.GetValueOrDefault("categoryid");
                if(string.IsNullOrWhiteSpace(categoryid))
                {
                    categoryid = "categories"; // as default
                }

                return categoryid;
                }" />
        <set-header name="Storage" exists-action="override">
            <value>permanentStorage</value>
        </set-header>
        <rewrite-uri template="@("{{[redacted]}}" + $"blob?resource=[redacted]}")" copy-unmatched-params="
    </inbound>
    <backend>
        <base />
    </backend>
    <outbound>
        <base />
        <choose>
            <when condition="@( ((IResponse)context.Response).StatusCode == 404 )">
                <set-body />
            </when>
        </choose>
    </outbound>
    <on-error>
        <base />
    </on-error>
</policies>
```

Ausschnitt API Management GET Methode. In blau markiert der Unterschied zum neuen Blob permanent Storage.

[redacted] PIM > Categories > Policies

```
<policies>
    <inbound>
        <base />
        <set-variable name="isValid" value="@(true)" />
        <set-variable name="isJsonBody" value="@{
                    try {
                        string body = context.Request.Body.As<string>(true);
                        JObject.Parse(body);
                        JObject bodyObj = context.Request.Body.As<JObject>(true);
                        return true;
                    }
                    catch {
                        return false;
                    }
        }" />
        <choose>
            <when condition="@(context.Variables.GetValueOrDefault<bool>("isJsonBody") == false)">
                <set-variable name="message" value="@("Body is not a valid JSON-Document.")" />
                <set-variable name="isValid" value="@(false)" />
            </when>
            <otherwise>
                <set-variable name="body" value="@(context.Request.Body.AsJObject(true, new JsonSerializerSettings() { DateParseHandling = DateParseHandling.None }))" />
            </otherwise>
        </choose>
        <choose>
            <when condition="@(!String.IsNullOrEmpty(context.Variables.GetValueOrDefault<string>("message")))">
                <return-response>
                    <set-status code="400" reason="Bad Request" />
                    <set-header name="Esb-Correlation-Id" exists-action="override">
                        <value>@( context.Variables.GetValueOrDefault<string>("esb-correlation-id") )</value>
                    </set-header>
                    <set-header name="Content-Type" exists-action="override">
                        <value>application/json</value>
                    </set-header>
                    <set-body>@{
        return "{\"error\": \"" + context.Variables.GetValueOrDefault<string>("message") + "\", \"esbCorrelationId\": \"" + context.Var
                    }</set-body>
                </return-response>
            </when>
        </choose>
        <set-backend-service base-url="{{ESB_APIM_INTERNAL_BASE_URL}}" />
        <set-header name="EsbApi-Subscription-Key" exists-action="override">
            <value>{{[redacted]}}</value>
        </set-header>
        <set-header name="Esb-Correlation-Id" exists-action="override">
            <value>@( context.Request.Headers["Esb-Correlation-Id"][0] )</value>
        </set-header>
        <rewrite-uri template="{{[redacted]}}" />
        <set-body>@{
                return context.Request.Body.As<string>();
            }</set-body>
    </inbound>
    <backend>
        <base />
    </backend>
    <outbound>
        <base />
        <choose>
            <when condition="@( ((IResponse)context.Response).StatusCode == 201 )">
                <return-response>
                    <set-status code="202" reason="Accepted" />
                    <set-header name="Esb-Correlation-Id" exists-action="override">
                        <value>@( context.Variables.GetValueOrDefault<string>("esb-correlation-id") )</value>
                    </set-header>
                    <set-header name="Content-Type" exists-action="override">
                        <value>application/json</value>
                    </set-header>
                    <set-body>@("{\"status\": \"" + "Accepted by ServiceBus" + "\"}")</set-body>
                </return-response>
```

Ausschnitt API Management POST Methode

Die Policies werden immer mit der Programmiersprache XML festgelegt. Die API wird dann ausgelöst, wenn ein bestimmtes Ereignis statt findet. Die Implementierung sieht man gleich in der ersten Zeile

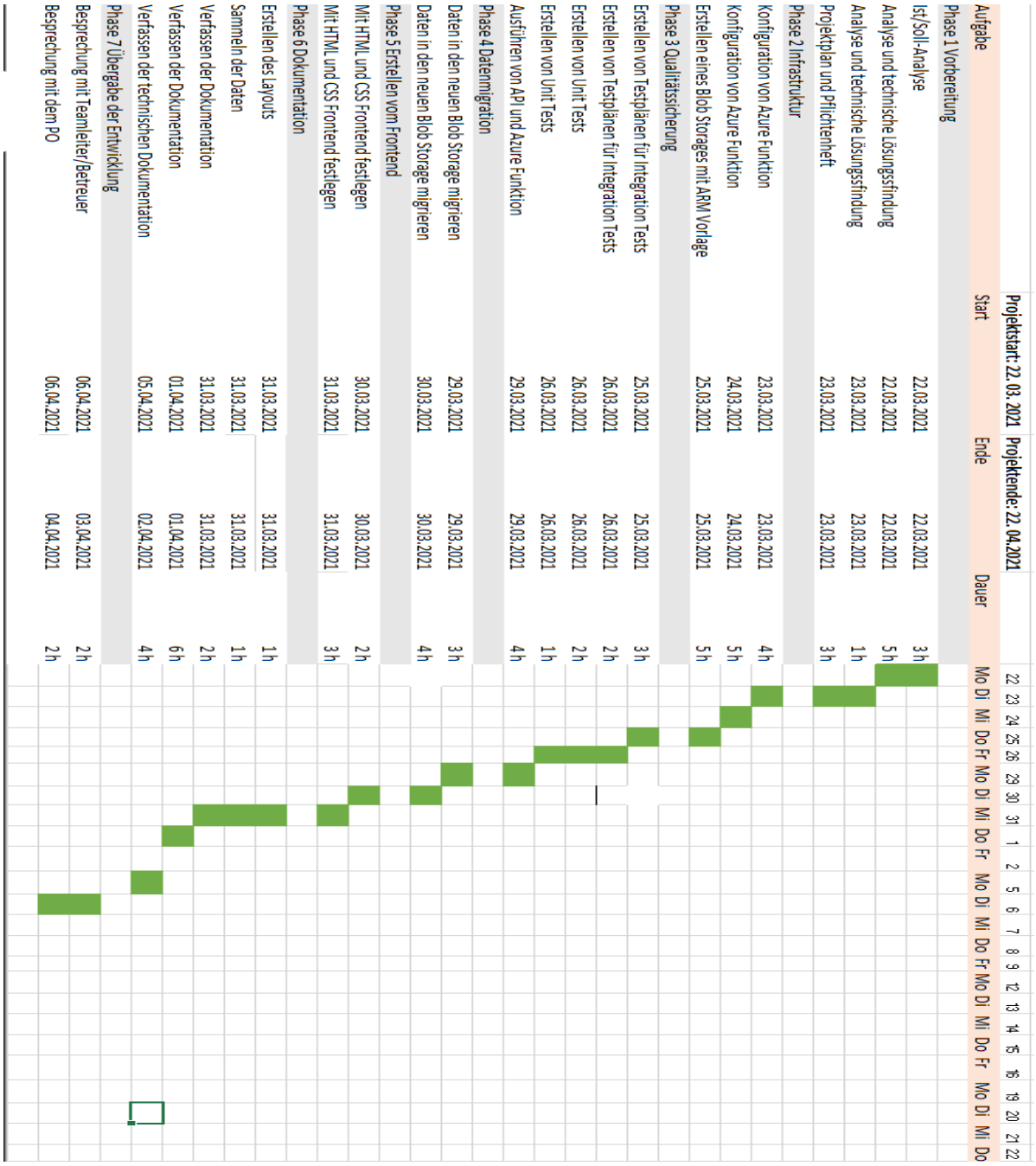
des Auszuges. In den Policies werden die möglichen Status Codes festgelegt. Die „Esb-Correlation-Id“ ist für das Identifizieren eines Requests da (beliebig automatisch erstellte Transaktionsnummer).

Bei dem Punkt *<value> appliation/json </value>* wurde das Format angelegt, welches als Dokument ankommen soll beim Nutzer. Jede API Operation hat außerdem noch einen eindeutigen Wert (value), der im Azure Portal vorhanden ist und in den Policies vermerkt werden muss für das Ausführen der API.

Zum Schluss wird dann noch eine Antwort an den Nutzer gesendet, mit folgendem Code:

<set-body>@{ return context.Request.Body.As<string>(); } </set-body>

5.5 Gantt-Diagramm



Technische Dokumentation

Cloud Speicheroptimierung

Inhalt

Zweck der technischen Dokumentation

Mit der technischen Dokumentation hat der Kunde eine Darstellung der technischen Implementierung der Cloud Speicheroptimierung. Außerdem dient sie zur richtigen Nutzung, sowie die generelle Beschreibung der API und Azure Funktion. Für diese Dokumentation benötigt man Grundkenntnisse in Azure, C#, HTTP Requests und JSON. Die API und die dazugehörige Funktion wurden nacheinander aufgebaut, welche auf der Originalversion basiert.

1. Benutzte Technologien und SDKs

1.1 Technologien

Aufgelistet sind die benutzen Technologien dieses Projektes:

- XML -> API Management Policies
- HTTP -> POST und GET Requests
- .NET Core -> Azure Functions
- .NET Framework -> API Management Policy

1.2 SDKs

Es gibt eine Menge an SDKs, die für dieses Projekt genutzt wurden. Für die Azure Funktion wurden folgende SDKs genutzt:

- Microsoft.Azure.WebJobs
- Microsoft.Azure.WebJobs.Extensions.Http
- Microsoft.AspNetCore.Http
- Microsoft.Extensions.Logging
- Newtonsoft.Json
- Newtonsoft.Json.Linq
- Microsoft.WindowsAzure.Storage
- Microsoft.WindowsAzure.Storage.Blob

2. Zweck und Nutzen der Azure Funktion

2.1 Nutzung vor der Implementierung

Vor der Anpassung der Azure Funktion war diese zuständig Daten in den „payloadStorage“ zu senden. Dieser Speicher ist auch ein Azure Blob Storage. Außerdem benutzten die Entwickler den Parameter „IBinder“, welcher im Nachhinein zu „cloudBlobContainer“ umbenannt wurde.

2.2 Nutzung nach der Implementierung

Die Azure Funktion besitzt nun einen anderen Connection String, welcher für die Speicherung in den neuen Storage verantwortlich ist. Dieser neue Speicher heißt „permanentStorage“. Sowie der payloadStorage, ist der „permanentStorage“ ein Azure Blob Storage, nur mit anderen Redundanzen. Den Parameter namens „IBinder“ gibt es nun nicht mehr. Dieser wurde zu „cloudBlobContainer“.

3. Zweck und Nutzen der API.

3.1 Nutzung vor der Implementierung

Vor der Implementierung wurden die Daten in nur einem Blob Storage gespeichert, der „payloadStorage“ heißt. Dies wurde mit der POST Methode durchgeführt. Der JSON Body wurde zur API weitergeleitet und im „payloadStorage“ gespeichert. Sobald man die GET Methode aufruft, wird der benötigte Inhalt von diesem Speicher geholt.

3.2 Nutzung nach der Implementierung

Nach der Implementierung ändert sich der Prozess der Datenspeicherung in den jeweiligen Speicher, da es zwei verschiedene Speicher gibt. Zum einen den „payloadStorage“ und zum anderen den „permanentStorage“, welcher neu hinzugefügt wurde während der Projektarbeit. Ein JSON-Dokument wird an eine Azure Funktion gesendet und speichert diese im Blob Storage.

Der Response-Body (Antwort mit Inhalt im JSON Format) beinhaltet weiterhin die einmalige Correlation-ID (wird in unserem Projekt so genannt). Diese Correlation-ID wird für das Logging genutzt, denn mit dieser ID kann man im Logging nach der bestimmten Antwort suchen, jedoch nur mit Zugriffberechtigungen. Außerdem prüft die POST-Methode, ob die abgesendete JSON Datei ein korrektes Format besitzt. Sobald dies zutrifft, werden die Daten weitergeleitet.

Und so soll eine JSON Datei aussehen:

Links ist ein Screenshot als Beispiel für ein JSON Dokument, der für andere Methoden und APIs aber anders aussehen würde. Wichtig ist jedoch, dass der Datentyp angegeben wird, keine doppelten Anführungszeichen fehlen, keine Klammern fehlen, sowie die Kommas komplett eingefügt sind. Wenn diese Regeln nicht eingehalten werden, dann kann das JSON Dokument nicht gespeichert werden und der Nutzer bekommt einen Fehlercode als Antwort.

In den doppelten Anführungszeichen setzt man dann einen Wert ein, der als Datentyp string weitergeleitet wird.

4.Erfolgreiches Durchführen

Bei einem erfolgreichen Durchlauf werden die abgesendeten Daten im korrekten Blob Storage gespeichert. Außerdem sollten die JSON Bodies komplett am Zielspeicherort ankommen, ohne dass ein Teil verloren geht. Sobald ein Request (Absenden eines JSON Dokuments) erfolgreich ist, wird ein Statuscode „200“ = OK an den Nutzer zurückgegeben bei einem GET Request und Statuscode „201“ = Created bei einem POST Request.

5.Fehlerhaftes Durchführen

Bei einem fehlerhaften Durchlauf werden die abgesendeten Daten nicht im korrekten Blob Storage gespeichert. Die Antwort an den Nutzer lautet „400“ = Bad Request bei einem POST Request und bei einem GET Request erhält der Nutzer den Code „404“ = Not Found. Es kann aber auch vorkommen, dass man die Antwort „Internal Server Error“ (Code 500) erhält. Übersicht der möglichen HTTP Statuscodes

6. Mögliche Statuscodes

Code	Erklärung
200	**OK** Erfolgreiche Anfrage. Das Ergebnis wird in der Antwort vermerkt.
201	**Created** Erfolreiche Anfrage. Das Dokument wurde erstellt, bzw. angepasst.
202	**Accepted** Erfolgreiche Anfrage, die zu einem späteren Zeitpunkt ausgeführt wird.
400	**Bad Request** Fehlerhafte Anfrage.
401	**Unauthorized** Fehlerhafte Anfrage. Der Absender hat keine Berechtigung.
404	**Not Found** Fehlerhafte Anfrage. Die API findet das angeforderte Dokument nicht.
500	**Internal Server Error** Unerwarteter Serverfehler.