hochschule mannheim

Globale Bereitstellung einer Web App in der Public Cloud unter Berücksichtigung der Sicherheit und des Datenschutzes

Nicolas Boskamp und Lara Elvira Gómez

Projektlabor

Industrial Internet of Things

im Studiengang Informationstechnik der Fakultät Informationstechnik

Vorgelegt von	Nicolas Boskamp
	Lara Elvira Gómez
am	19. Februar 2022
Professor	Prof. Dr. Eckhart Körner
Korreferent	Robert Schmidt

Schriftliche Versicherung laut Studien- und Prüfungsordnung

Hiermit erklären wir, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Mannheim, 19. Februar 2022

Nicolas Boskamp und Lara Elvira Gómez

Zusammenfassung

Der vorliegende Bericht beschäftigt sich mit der prototypischen globalen Bereitstellung einer Web Anwendung in der Cloud. Die Anwendung wird in den Regionen Europa und Nordamerika zur Verfügung gestellt.
Für die Umsetzung wird der Cloud-Service Microsoft Azure verwendet. Es wird eine ASP.NET Core Anwendung konzipiert und implementiert, welche Fitnessdaten von Nutzern sammelt, verarbeitet und visualisiert. Die Daten der Nutzer werden in SQL-Datenbanken gehalten, wobei entsprechend der Datenschutzbestimmungen in jeder Region eine eigene Datenbank erzeugt wird. Für die Authentifizierung der Nutzer wird ein Azure Active Directory verwendet. Dieses wird aufgrund der DSGVO in Europa erstellt. Um die Sicherheit der Datenübertragung zu gewähren, werden die Resourcen in virtuelle Netze integriert und der Zugriff über das öffentliche Internet wird blockiert. Damit aus dem Ausland auf Nutzerdaten zugegriffen werden kann, wird VNET-Peering zwischen den Regionen eingerichtet.
Die Anwendung soll horizontal skalierbar sein, um bei hoher Last oder dem Ausfall einzelner Resourcen weiterhin zur Verfügung zu stehen. Für eine Umsetztung der horizontalen Skalierbarkeit werden zwei Ansätze getestet. Im ersten Ansatz wird die Anwendung in Kubernetes Clustern und im zweiten über Azure App Services bereitgestellt.
Da bei der Umsetzung mit Kubernetes Clustern Probleme mit der Kompabilität mit dem Azure Active Directory aufgetreten sind, werden für die endgültige Architektur Azure App Services verwendet. Die Bereitstellung der Anwendung mit den erforderlichen Komponenten für die sichere Authentifizierung und Datenverwaltung der Nutzer ist erfolgt. Dabei werden die Anforderungen an die Hochverfügbarkeit und die Einhaltung der Datenschutzbestimmungen eingehalten.

Abstract

This report deals with the prototype global deployment of a web application in the cloud. The application is made available in the regions of Europe and North America. The Microsoft Azure cloud service is used for the implementation. An ASP.NET Core application will be designed and implemented that collects, processes and visualizes fitness data from users. The users' data is held in SQL databases, with a separate database created in each region in accordance with data protection regulations. An Azure Active Directory is used to authenticate the users. This is created in Europe due to the GDPR. To ensure the security of data transmission, the resources are integrated into virtual networks and access via the public Internet is blocked. To ensure that user data can be accessed from abroad, VNET peering is set up between the regions. The application is to be horizontally scalable in order to remain available in the event of high load or the failure of individual resources. Two approaches are being tested to implement horizontal scalability. In the first approach, the application is deployed in Kubernetes clusters and in the second via Azure App Services.
Since the implementation using Kubernetes clusters encountered issues with compatibility with Azure Active Directory, Azure App Services is used for the final architecture. Deployment of the application with the necessary components for secure authentication and data management of users has been done. High availability and privacy compliance requirements are met.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AAD	Azure Active Directory
AKS	Azure Kubernetes Service
TTL	Time to Live
JWT	JSON Web Token
ACR	Azure Container Registry
DSGVO	Datenschutz-Grundverordnung
VNET	Virtual Network
FQDN	Fully Qualified Domain Name

Kapitel 1

Einleitung

1.1 Motivation

Früher wurden Anwendungen auf physischen Servern ausgeführt, die von Organisationen verwaltet wurden. Mit den Cloud Diensten die zur heutigen Zeit zur Verfügung gestellt werden, ist es möglich Ressourcen bedarfsabhängig zu nutzen und nur die in Anspruch genommenen Leistungen zu zahlen.[1] Mit Cloud Anbietern wie Azure kann die globale Bereitstellung von Ressourcen erzielt werden.
Auch die sichere Authentifizierung von Nutzern muss gewährleistet werden, dafür bietet Azure ein Identitätsverwaltungsdienst namens Active Directory an. Ein weiterer Sicherheitsaspekt ist der Azure Key Vault, der Zertifikate, Schlüssel und Geheimnisse sichern kann. Dadurch muss nicht offen mit Anwendungsgeheimnissen im Projektcode gearbeitet werden.
Eine globale Anwendung muss die Gesetzte der betroffenen Regionen, in diesem Fall Europa und USA, berücksichtigen. Die einfachste Möglichkeit ist die Bestimmungen der strengsten Datenschutzverordnungen, in diesem Fall Europa, geltend zu machen. Daten aus Europa müssen nach europäischen Bestimmungen in Europa gespeichert werden. Da die Datenschutzbestimungen der Vereinigen Staaten das nicht verlangen, können Daten aus den USA in Europa gehalten werden.
Neben der Gesetzgebung ist bei der Bereitstellung einer globalen Anwendung die horizontale Skalierbarkeit wichtig. Damit kann eine Verfügbarkeit trotz Ausfällen einzelner Instanzen oder bei hohen Lasten gewährleistet werden.

1.2 Zielsetzung

In dem vorliegenden Bericht soll eine prototypische Fitness Anwendung entwickelt werden, die weltweit in der public Cloud bereitgestellt wird. In ihrer ersten Version soll die Anwendung Nutzern in den Vereinigten Staaten und in Europa zur Verfügung gestellt werden. Die Resourcen einer Region sollen in virtuelle Netze integriert und der Zugriff über das öffentliche Internet geblockt werden. Der Datenaustausch soll zwischen den unterschiedlichen Regionen über VNET-Peering realisiert werden. Das soll dazu beitragen die Sicherheit der Datenübertragungen und den Datenschutz(DSGVO) einzuhalten. Nutzer sollen sich über das Azure Active Directory anmelden und registrieren können. Um ein Multi-Cloud Szenario vorzubereiten, soll die Applikation exemplarisch mit je einem Kubernetes Cluster und einem zugehörigen Load Balancer pro Region bereitgestellt werden. Der Zugriff auf die Anwendung soll über den geographisch nächsten Zugriffspunkt erfolgen.

1.3 Aufbau der Arbeit

Zunächst wird in Kapitel 2 der aktuelle Stand der Technik, der in dieser Arbeit verwendeten Azure Komponenten und der JSON Web Token erläutert. Im darauf folgenden Kapitel 3 werden die Anforderungen an das zu entwickelnde System gestellt.
Im Anschluss folgt der Hauptteil dieser Arbeit. Zunächst werden in Kapitel 4 die Konzeption und Realisierung der prototypischen Fitness-Web-Anwendung beschrieben. In Kapitel 5 werden die Konzeption und Realisierung der Architektur mit Kubernetes Clustern vorgestellt. Darauf aufbauend wird in Kapitel 6 eine neue Konzeption und Realisierung einer Architektur unter Verwendung von Azure App Services beschrieben. Abschließend werden in Kapitel 7 die Ergebnisse der Arbeit vorgestellt und evaluiert.

Kapitel 2

Stand der Technik

Ziel dieses Kapitels ist die Vermittlung von grundsätzlichem Wissen zum Verständnis der folgenden Kapitel über relevante Technologien von Azure und Json Web Token. Diese Kapitel hat nicht den Anspruch sämtliche Informationen zu enthalten, sondern soll einen groben Überblick über die Themen geben. Für weitere Informationen werden die aufgeführten Quellen der offizielen Microsoft-Dokumentation [2], [3], [4], [5] und [6] empfohlen.

2.1 Azure Active Directory B2C

Microsoft Azure stellt zur Authetifizierung das sogenannte Azure Active Directory (AAD) bereit. Dabei wird zwischen dem normalen AAD und dem AAD B2C (Business to Customers) unterschieden. Beide basieren auf derselben Technologie, dienen aber verschiedenen Zwecken. Das AAD B2C zielt auf Unternehmen oder Einzelpersonen, die kundenorientierte Mobil- oder Web-Anwendungen entwickeln, worüber sich Endnutzer authentifizieren können. [2]
Bei dem Azure Active Directory B2C handelt es sich um eine CIAM-Lösung (Customer Identity Access Management). Diese ermöglichen es Millionen von Nutzern und Milliarden von Authentifizierungen pro Tag zu unterstützen. [2]
Azure B2C garantiert die Sicherheit durch Bedrohungen wie Denial-of-Service-, Kennwort-Spray- oder Brute-Force Angriffen. [2] Im folgenden Verlauf der Arbeit wird das AAD B2C der Einfachheit halber AAD abgekürzt.

2.2 Azure App Service

Der Azure App Service ist ein *PaaS*-Dienst, welchen Azure zum Erstellen und Bereitstellen von Webanwendungen anbietet. Von Azure werden die Laufzeit, die Dienstschicht, das Betriebssystem, die Visualisierung, der Server, sowie Speicher und Netzwerk verwaltet. Entwickler müssen sich nur um die Anwendung und die Datenverwaltung kümmern. [3]
Ein App Service ist ein *HTTP*-basierter Dienst zum Hosten von Webanwendungen, API-Anwendungen und mobilen Applikationen. Es sind eine Vielzahl von Programmiersprachen nutzbar, wie .NET, .NET Core, Java oder Python. Applikationen können sowohl in Windows oder Linux Umgebungen laufen und zusätzlich skaliert werden.[3]
Da ein App Service immer in einem App Service-Plan ausgeführt wird, muss bei einer Skalierung der Plan mit Resourcen erweitert werden. Dabei wird beispielsweise mehr Arbeitsspeicher hinzugefügt oder die Anzahl der Instanzen erhöht. [7]

2.3 Azure Kubernetes Service

Orchestrierungstools wie Kubernetes sind nützlich, um Container bereitzustellen, zu verwalten und zu skalieren. Es werden unterschiedliche Container-Tools wie Docker unterstützt. Dazu bietet Azure des Azure Kubernetes Service (AKS) an. Container sind im Vergleich zu virtuellen Maschinen leichtgewichtiger, da sie sich das Host-Betriebssystem teilen. Sie besitzen eigene Dateisysteme und sind über Betriebssystem-Distributionen portabel. [8]
Ein AKS-Cluster besteht aus einem Master und mehreren Nodes, wie in Abbildung 2.1 dargestellt ist. In dem Master gibt es einen Kube-API-Server, welcher die Schnittstelle zum Master verfügbar macht. Mit der CLI kann Zugriff darauf erhalten werden. Etc ist eine weitere Komponente im Master, dabei handelt es sich um ein Key-Value Speicher, welcher die Konfigurationen des Clusters hält. Der scheduler weist neue Pods einem Node zu und berücksichtigt dabei beispielsweise die Ressourcenanforderungen. Der Controller Manager überwacht die Nodes und reagiert zum Beispiel bei einem Ausfall. [9]
Ein Node besteht aus einem kubelet. Dieser stellt einen Agenten dar, welcher die Pods überwacht und sicherstellt, dass sie ausgeführt werden. Der kube-proxy stellt einen Dienst dar, auf den zugegriffen werden kann. So können einzelne Anfragen auf die einzelnen Pods weitergereicht werden. Die letzte Komponente eines Nodes ist die Containerlaufzeit. Diese führt die einzelnen Container aus. Auf den einzelnen Nodes laufen Container in Pods, welche die Web Anwendung ausführen.[9]

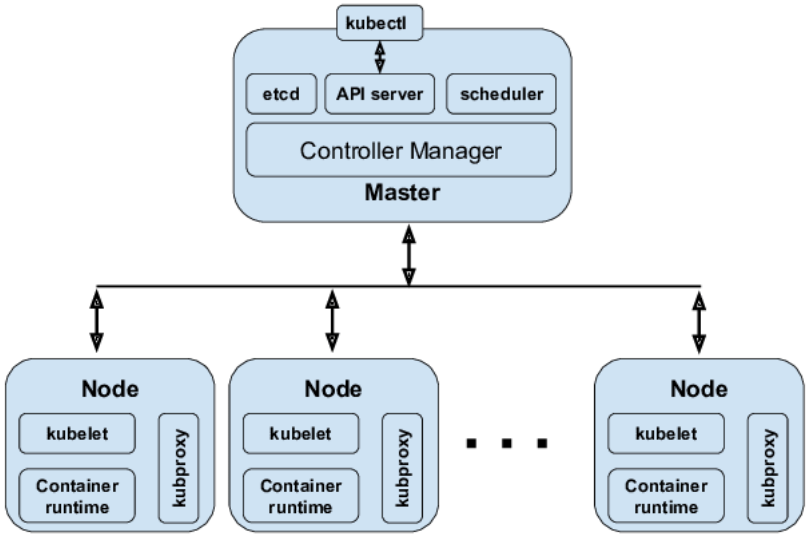


Abbildung 2.1: Architektur Kubernetes [10]

2.4 Azure Key Vault

Azure Key Vault ist ein Dienst von Microsoft, welcher erlaubt Geheimnisse zu speicheren und den Zugriff auf diese zu administrieren. Ein Geheimnis kann beispielsweise ein Kennwort, Zertifikate oder eine Verbindungzeichenfolge einer Datenbank sein. Beim Key Vault wird zwischen zwei Arten von Containern unterschieden. Zum einen Tresore und zum anderen *Hardware Security Module Pools (HSM-Pools)*. Mit einem Tresor-Container können Schlüssel. Geheimnisse und Zertifikate verwaltet werden. In einem HSM-Pool-Container können nur HSM-geschütze Schlüssel verwaltet werden. [6]

2.5 JSON-Web-Token

Bei einer Übertragung von Claims zwischen zwei Parteien kann ein JSON Web Token (JWT) verwendet werden. Dieser bietet ein sicheres Mittel zur Darstellung von Claims. Ein Claim ist dabei ein Name/Werte Paar. Der Name wird durch einen String repräsentiert und der Wert kann jeden JSON-Wert annehmen. Ein Beispiel hierfür ist “Nutzername“: “John Doe“.

Ein JWT besteht aus drei Teilen. Einem Header, einer Payload und einer Signatur. In dem Header wird der Typ, der verwendete Algorithmus für die Signaur und der “key identifier (kid) “festgehalten. Der “kid “beschreibt ein Schlüsselidentifizierer, welcher genutzt wird. Dies ist besonders von Bedeutung, wenn mehrere Schlüssel vorhanden sind, um zu wissen welcher genutzt wurde, um das Token zu signieren.

Der zweite Teil des JWT ist die *Payload*, welche die wesentlichen Daten des Tokens enthält. Dabei wird beispielsweise bei der Anmeldung im AAD die Domäne, die zum Login führt im Issuer-Claim, als *https://<primary domain>* gehalten. Außerdem können personenbezogene Daten, wie das Herkunfsland, die Anschrift, die Email-Adresse, der Nutzername, sowie Vor- und Nachname enthalten sein. Ein weiterer wichtiger Punkt ist der tfp-Eintrag, welcher den Typ des Benutzerflows im AAD enthält. Dieser beschreibt, ob es sich um eine Anmeldung, Registrierung oder um eine Passwortänderung handelt. In Listing 2.1 werden die Daten eines decodierten JWTs, bei Verwendung von AAD, angezeigt. [11] [12]

```
\\HEADER{
  "typ": "JWT",
  "alg": "<Algorithm>",
  "kid": "<key identifier>"
}
\\PAYLOAD{
  "exp": <Expiration-Time-Claim>,
  "nbf": <Not-Before-Claim>,
  "ver": "<version-number>",
  "iss": "<Issuer-Claim>",
  "sub": "<Object-ID-User>",
  "aud": "<AAD-Client-ID>",
  "nonce":"<nonce>",
  "iat": <Issued-At-Claim >,
  "auth_time": <Authorisation-Time>,
  "oid": "<Object-ID-User>",
  \\User profile data
  "city": "<city>",
  "country": "<country>",
  "name": "<user-name>",
  "given\_name": "<given-name>",
  "postalCode": "<postal-code>",
  "state": "<state>",
  "streetAddress": "<street-address>",
  "family\_name": "<family-name>",
  "emails": ["<e-mail adress>"],
  "tfp": "<Userflow-Name>"
}
```

Listing 2.1: Dekodierter JWT [11] [12]

Den letzten Teil des JWTs bildet die Signatur, mit einem privaten und öffentlichen Schlüssel. In Listing 2.2 wird die Kodierung in Base64 des Headers und des Payloads, mit der anschließender Hashbildung dargestellt [13].

Der Header, der Payload und die Signatur werden durch einen Punkt getrennt und bei dem Aufruf einer Seite mitgegeben.

```
1 var encoded = RSASHA256( base64UrlEncode(header) + "." +
     base64UrlEncode(payload);
2 var hash = HMACSHA256(encoded, secret);
3
4 var jwt = base64UrlEncode(header) + "." + base64UrlEncode(
     payload) + "." + hash
```

Listing 2.2: Signatur eines JWT [13]

Kapitel 3

Anforderungsanalyse

In diesem Kapitel werden die Software-Anforderungen beschrieben. Diese sind unterteilt in drei Bereiche. Der Erste beschreibt die Anforderungen an die zu entwickelnde Web Applikation. Der zweite Teil beschreibt die Anforderungen an die globale Bereitstellung. Im dritten und letzten Teil werden die Anforderungen an die Sicherheit und Datenschutzbestimmungen erläutert.

3.1 Web-Anwendung

In diesem Unterkapitel werden die Anforderungen an die zu entwickelnde Anwendung beschrieben. Daraus geht hervor, dass eine prototypische Applikation mit dem Web-Framework ASP.NET Core entwickelt werden soll. Für die Cloudbereitstellung soll Microsoft Azure verwendet werden. Die Daten sollen in Azure SQL-Datenbanken gespeichert werden.

ID	Beschreibung
SWR_ 001	Es soll eine prototypische Fitness-Web-Anwendung entwickelt werden.
SWR_ 002	Die Web-Anwendung soll mit dem Web-Framework ASP.NET Core implementiert werden.
SWR_ 003	Die Daten sollen in Azure SQL-Datenbanken gehalten werden.

Tabelle 3.1: Anforderungen an die Web-Anwendung

3.2 Globale Bereitstellung

In diesem Unterkapitel werden die Anforderungen, die die globale Bereitstellung betreffen, erläutert. Daraus geht hervor, dass die entwickelte Fitness-Anwendung mit dem Cloud-Dienst Microsoft Azure weltweit bereitgestellt werden soll. Prototypisch soll dies für die beiden Regionen Europa und USA realisiert werden. Um ein Multi-Cloud-Scenario vorzubereiten, soll der AKS verwendet werden. Pro Region soll dazu ein AKS-Cluster mit zugehörigem Loadbalancer bereitgestellt werden. Außerdem soll ein Traffic Manager von Azure eingerichtet werden, welcher die Nutzer auf den geografisch nächsten Zugriffspunkt der Anwendung weiterleitet. Der Datentransfer zwischen den Regionen soll über VNET Peering realisiert werden.

ID	Beschreibung
SWR_ 004	Die Entwicklung der Cloud-Architektur soll in Microsoft Azure umgesetzt werden.
SWR_ 005	Die Fitness Anwendung soll weltweit bereitgestellt werden. Das soll prototypisch für die beiden Regionen Europa und Vereinigte Staaten erfolgen.
SWR_ 006	Um ein Multi-Cloud-Szenario vorzubereiten, soll der AKS verwendet werden.
SWR_ 007	In jeder Region soll ein AKS-Cluster mit vorgeschaltetem Loadbalancer implementiert sein.
SWR_ 008	Es soll ein Azure Traffic Manager eingesetzt werden, der die Nutzer auf den geografisch nächsten Zugriffspunkt der Anwendung weiterleitet.
SWR_ 009	Der Datentransfer zwischen den beiden prototypischen Regionen soll über VNET Peering stattfinden.

Tabelle 3.2: Anforderungen an die globale Bereitstellung

3.3 Sicherheit und Datenschutzbestimmungen

In diesem Unterkapitel werden die Anforderungen bezüglich der Sicherheit und des Datenschutzes nach DSGVO-Norm beschrieben. Daraus geht hervor, dass Nutzer sich vor dem Zugriff auf die Applikation über ein AAD unter der Verwendung von JWT authentifizieren sollen. Um eine Entscheidung zu treffen, auf welche Datenbank zugegriffen wird, soll die Herkunft des Nutzers aus dem JWT hervorgehen. Die Anmeldeinformationen des AAD sollen DSGVO-Konform für europäische Nutzer nur in Europa gespeichert werden. In der Datenbank soll zwischen personen-und nicht-personenbezogenen Daten unterschieden werden. Der Abruf von personenbezogenen Daten soll unter Einhaltung der DSGVO auch aus einer anderen Region möglich sein. Um nicht mit Anwendungsgeheimnissen offen arbeiten zu müssen, soll ein Key-Vault verwendet werden.

ID	Beschreibung
SWR_ 010	Bevor Nutzer Zugriff auf die Anwendung erlangen, soll es eine Authentifizierung der Nutzer geben.
SWR_ 011	Die Authentifizierung soll mit AAD unter Verwendung von JWT realisiert werden.
SWR_ 012	Aus den JWT soll die Herkunft der Nutzer hervorgehen.
SWR_ 013	Je nach Herkunft eines Nutzers soll die Entscheidung des Zugriffs auf die entsprechende Datenbank (Europa/USA) nach erfolgreicher Authentifizierung getroffen werden.
SWR_ 014	Die Anmeldeinformationen europäischer Nutzer des AAD müssen in Europa gespeichert sein (DSGVO).
SWR_ 015	In den Datenbanken soll zwischen Personen und nicht-personen bezogenen Daten unterschieden werden.
SWR_ 016	Die personenbezogenen Daten von Nutzern in Europa sollen DSGVO-Konform nur in Europa gehalten werden. Es soll keine Replikationen in Nordamerika geben.
SWR_ 017	Unter Einhaltung des Datenschutzes soll der Abruf personenbezogener Daten aus einer anderen Region möglich sein.
SWR_ 018	Die Verbindungszeichenfolgen der Datenbanken sollen in einem Azure Key-Vault gespeichert werden.

Tabelle 3.3: Anforderungen an die Sicherheit und Datenschutzbestimmungen

Kapitel 4

Konzeption und Realisierung der prototypischen Fitness-Anwendung

Im folgenden Kapitel wird die prototypische Web Anwendung beschrieben. Dabei handelt es sich um die Anwendung, welche im späteren Verlauf über die Cloud bereitgestellt wird. Im ersten Unterkapitel wird die Konzeption beschrieben. Abschließend wird die Realisierung der Applikation vorgestellt.

4.1 Konzeption der Fitness-Anwendung

Es soll eine Anwendung konzepiert werden, welche Fitnessdaten von Nutzern erfasst, diese speichert und visualisiert. Über einen *Sign in*-Button am oberen Reiter sollen sich Anwender über AAD bei der Web-Applikation authentifizieren können. Angemeldete Nutzer sollen über einen Reiter auf eine *Workout-Seite*, eine *BMI-Seite* und eine *Profil-Seite* gelangen.
Auf der *Workout-Seite* sollen Nutzer über Textfelder ihre Workout Übungen und Zeiten beziehungsweise Wiederholungen erfassen können und diese mit einem Button bestätigen. Ein Beispiel hierfür wäre ein Eintrag für die Übung „Liegestützen“ mit der Angabe 20 Wiederholungen. Diese Daten sollen in einer Datenbank gespeichert und von dieser zur Visualisierung abgerufen werden können. Die erfassten Übungen sollen für die einzelnen Tage in einem Balkendiagramm dargestellt werden. So soll eingesehen werden können, welche Übungen an welchem Tag mit welcher Intensität gemacht werden. In Abbildung 4.1 ist das geplante User Interface zu sehen.

Abbildung 4.1: UI-Konzeption der Workout-Seite

Auf der *BMI-Seite* soll der Nutzer sein Gewicht und seine Körpergröße in Textfelder eintragen und über einen Button bestätigen können. Diese Daten sollen, wie bei der *Workout-Seite* in einer Datenbank gespeichert und von dieser zur Visualisierung abgerufen werden können. Unter den Eingabefeldern soll ein Liniendiagramm zu sehen sein, welches die Körpergröße und das Gewicht über die Zeit darstellt. Aus diesen Angaben soll ein Body Mass Index (BMI) berechnet und im selben Diagramm visualisiert werden. Das konzipierte User Interface ist in Abbildung 4.2 dargestellt.

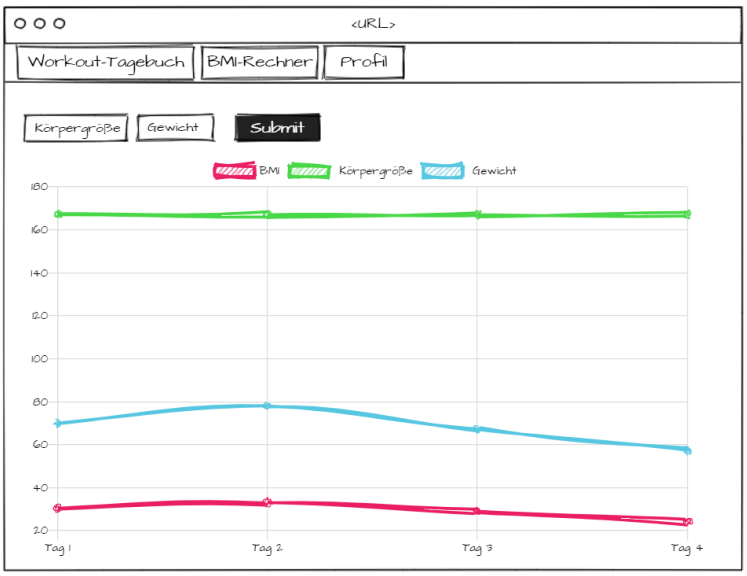


Abbildung 4.2: UI-Konzeption der BMI-Seite

Zusätzlich soll eine Profil-Seite konzipiert werden, welche die Daten aus dem JWT darstellt. Diese Seite soll nur zur prototypischen Darstellung der JWT-Inhalte dienen und veranschaulichen, wie auf die einzelnen Claims zugegriffen werden kann und wie diese genutzt werden.

4.2 Realisierung der Fitness-Anwendung

Zur Realisierung der Web-Anwendung wird das Framework ASP.NET 5.0 und das Core-UI Dashboard Template verwendet, welches ein Bootstrap Template für HTML und CSS mit UI-Vorlagen enthält. Mit dessen Hilfe werden UI-Elemente in der Anwendung gestaltet.
Für die Realisierung der *Workout-Seite* werden zwei Textfelder und ein Bestätigungs-Button implementiert. Darüber werden die Daten mittels eines dafür erstellten Modells in die Datenbank geschrieben. Das Modell besteht aus einem Zeitstempel und der Übung in Form von Strings und der Anzahl in Form eines Integer-Wertes.
Diese Daten werden in einem Balkendiagramm visualisiert. Dazu wird ein Chart-Plugin von *Chart.js* verwendet. Es werden eine Reihe von Funktionen implementiert. Zunächst werden die Workout-Daten zur Visualisierung nach Tagen sortiert. Gibt ein Nutzer eine Übung ein, die zum ersten Mal durchgeführt wird, wird diese zur Legende hinzugefügt und ein Balken erscheint an dem entsprechenden Tag im Balkendiagramm. Sollte ein Nutzer an einem Tag eine Übung mehr als einmal ausführen, werden diese Eingaben zu der bereits getätigten Übung an dem entsprechenden Tag aufaddiert.
Für die *BMI-Seite* werden wie auch für die *Workout-Seite* zwei Textfelder und ein Bestätigungs-Button implementiert. Ebenfalls wird ein Modell erzeugt, welches Körpergröße und Gewicht in Form von Integers und einen Zeitstempel in Form eines Strings enthält. Die erfassten Daten der Nutzer werden in einem Liniendiagramm dargestellt. Dieses wird ebenfalls mit dem Chart-Plugin *Chart.js* erzeugt. Dabei werden die Einträge mit einem Zeitstempel, welcher Tag und Uhrzeit enthält, im Diagramm dargestellt. Bei einer Eingabe wird das Diagramm aktualisiert.
Zuletzt werden auf der Profil-Seite die Claims des JWT, wie in der Konzeption beschrieben, in einer Tabelle aufgelistet. Damit lässt sich prototypisch zeigen, wie mit den Attributen des JWT gearbeitet werden kann. Für den späteren Verlauf wird der Eintrag der Herkunft eines Anwenders dazu genutzt, um zu entscheiden, auf welche Datenbank der Zugriff erfolgt.
Die Realisierung des Zugriffs auf die Datenbank und den Key Vault aus dem Projektcode heraus werden in den jeweiligen Kapiteln beschrieben. Das gilt auch für das Einbinden des AAD. Die Ergebnisse zur Realisierung der Web Anwendung werden in Kapitel 7.1 vorgestellt.

Kapitel 5

Konzeption und Realisierung der Kubernetes-Architektur

Dieses Kapitel befasst sich mit der Konzeption und Realisierung der Architektur unter Verwendung von AKS. Dazu wird zunächst die Konzeption der Gesamtarchitektur vorgestellt.
Im Anschluss wird die Realisierung der Architektur erläutert. Danach wird die Realisierung der Komponenten AAD, SQL-Datenbank und Azure Key Vault vorgestellt. Das Zusammenspiel dieser drei Komponenten wird anschließend mithilfe eines App Services getestet. Daraufhin wird mithilfe von Docker aus der Web-Anwendung ein Container erzeugt, welcher in eine Azure Container Registry (ACR) deployed wird. In einem letzten Schritt wird ein Kubernetes Cluster erzeugt, in welchem die containerisierte Anwendung zur Ausführung gebracht wird.

5.1 Konzeption der Kubernetes-Architektur

In diesem Kapitel wird die Gesamtarchitektur mit den Kubernetes Clustern beschrieben. In Abbildung 5.1 ist diese dargestellt. Die Anwendung soll für die zwei Regionen Europa und Vereinigte Staaten realisiert werden. Wie in den Anforderungen beschrieben, soll sich in jeder Region ein Kubernetes Cluster in einem virtuellen Netz befinden, welches die Container Instanz aus einem ACR bezieht und in die einzelnen Pods repliziert. Dem Kubernetes Cluster soll jeweils ein Load Balancer zum Lastenausgleich zwischen den Pods vorgeschaltet werden. Mithilfe eines Traffic Managers soll die Anfrage der Nutzer auf die Anwendung über den geographisch nächsten Zugriffspunkt geleitet werden.
Es soll ein AAD zur Authentifizierung der Nutzer zur Verfügung gestellt werden. Dieses soll aufgrund der DSGVO in Europa stationiert werden, um zu garantieren

dass europäische Nutzerdaten in Europa gehalten werden. In jeder Region soll es eine Datenbank und ein Key Vault geben. In den Datenbanken sollen die Nutzerdaten der jeweiligen Region gespeichert werden. Der Key Vault dient dem Schutz der Verbindungszeichenfolge der Datenbank. Der Zugriff auf die Datenbank und den Key Vault soll in der eigenen Region über das virtuelle Netz erfolgen. Befindet sich zum Beispiel ein Nutzer aus Europa in den USA soll er über Peering auf die Datenbank und den Key Vault in Europa zugreifen können.

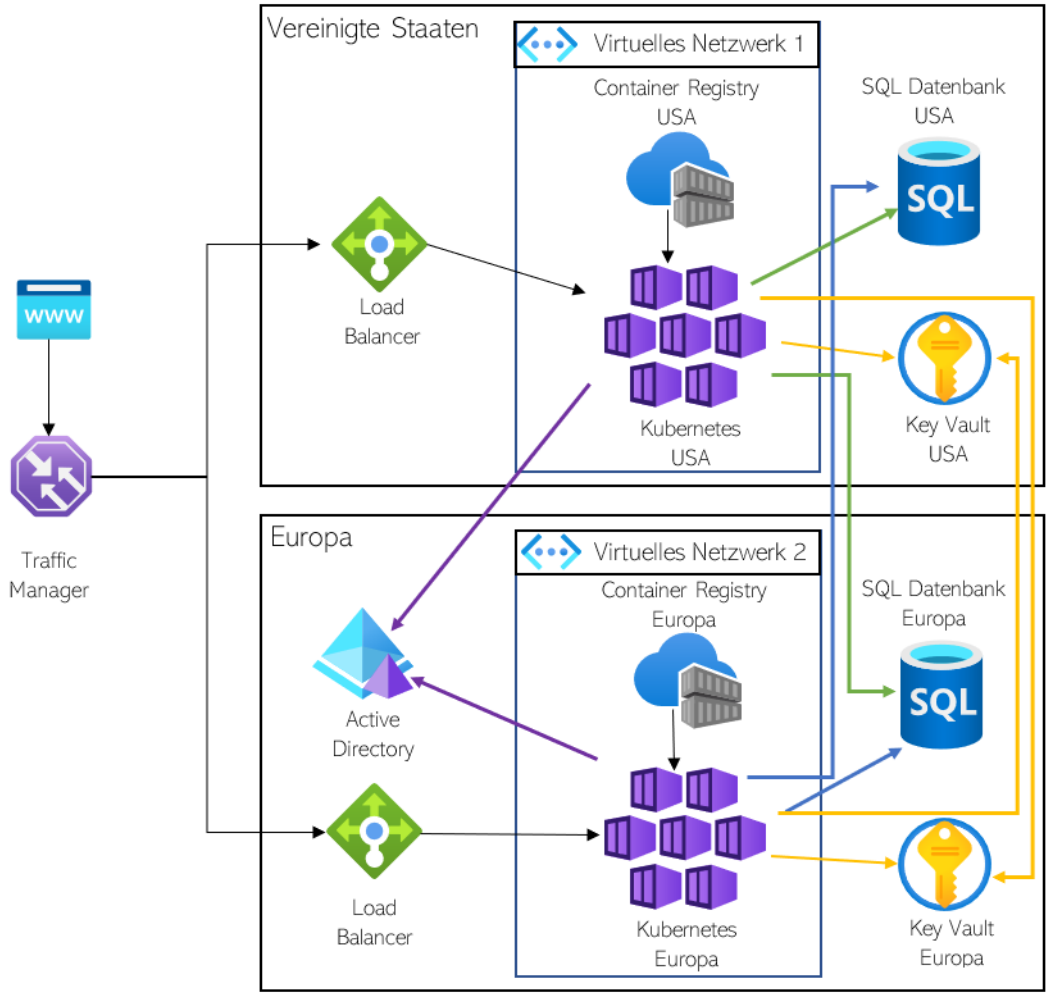


Abbildung 5.1: Konzeption der Gesamtarchitektur mit Kubernetes

Bei der Umsetztung soll wie folgt vorgegangen werden. In einem ersten Schritt soll die Architektur für eine Region realisiert werden. Dazu sollen zunächst die Komponenten AAD, Key Vault und Datenbank aufgestellt werden. Um das Zusammenspiel der drei Komponenten zu testen, wird ein kostenfreier App Service erzeugt. Diese Testarchitektur wird in Abbildung 5.2 dargestellt. Nachdem die Verknüpfung die-

ser Komponenten erfolgt ist, soll der App Service schließlich durch das Kubernetes Cluster, einen Load Balancer und die ACR ersetzt werden. Anschließend soll das Kubernetes Cluster in ein virtuelles Netz integriert werden.
Bei erfolgreicher Umsetztung soll bei einem nächsten Schritt diese Architektur für eine zweite Region dupliziert werden. Zum Schluss sollen das Peering zwischen den beiden Regionen und der Traffic Manager eingerichtet werden.

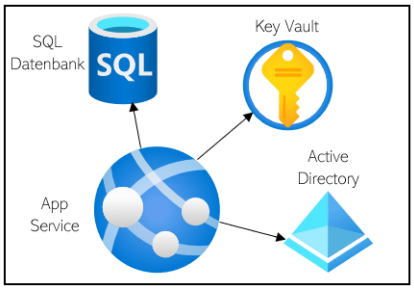


Abbildung 5.2: Test-Architektur der Komponenten AAD, SQL-DB und Key Vault

5.2 Azure Active Directory

In diesem Kapitel wird die Vorgehensweise erläutert, nach welcher das AAD erzeugt, konfiguriert und mit einer Web-Anwendung verbunden wird. Dabei wird Azure-seitig das Erstellen eines AAD B2C-Mandanten, die Appregistrierung beim AAD mit zugehöriger Konfiguration derselben und das Einrichten von Benutzerflows beschrieben. Dies wird zunächst, wie in Kapitel 5.1 beschrieben, mit einem Azure App Service Plan umgesetzt. Anschließend wird die Konfiguration zur Verbindung der ASP.NET Web-Anwendung mit dem AAD in der *appsetting.json*-File des Projekts erläutert.

5.2.1 Erstellung des AAD im Azure Portal

Um ein AAD B2C anzulegen, wird im Azure Portal unter „Resourcen erstellen“ das „Azure Active Directory“ ausgewählt und mit dem Mandantentyp B2C erstellt. Die Parameter, welche dabei festgelegt werden, sind ein Organisationsname, der Name der Anfangsdomäne, die Resourcen-Gruppe, sowie die Region, in welcher der Mandant erzeugt wird. Für die Region wird Europa gewählt, um die Datenschutzbestimmungen einzuhalten, nach welchen die Daten aus der USA in Europa gespeichert werden dürfen, aber nicht umgekehrt. Der AAD B2C-Dienst ist unabhängig von der konfigurierten Region weltweit verfügbar. Bei dem Erstellen des AAD B2C-Dienstes, wird ein neues Verzeichnis erstellt. Der Name des Verzeichnisses ist der

vorher gewählte Organisationsname. Nachdem der Mandant erstellt ist, wird dieser konfiguriert. Dazu wird im Azure Portal zum Mandanten-Verzeichnis gewechselt.

5.2.2 App-Registrierung im AAD

Im AAD-Verzeichnis kann eine Applikation registriert werden. Bei der Registrierung werden neben dem Namen die unterstützten Kontotypen, die Umleitungs-URI und Berechtigungen konfiguriert. Für die unterstützten Kontotypen wird die Einstellung *„Konten in einem beliebigen Identitätsanbieter oder Organisationsverzeichnis (zum Authentifizieren von Benutzern mit Benutzerflows)“* gewählt.
Dabei handelt es sich um die gängigste Option für Anwendungen, die für ihre Kunden verfügbar gemacht werden sollen. Nur mit dieser Option können AAD B2C-Benutzerflows erzeugt werden, welche im späteren Verlauf konfiguriert werden. Außerdem können mit dieser Option eine große Auswahl von Benutzeridentitäten verwendet werden, wie zum Beispiel Microsoft, Facebook oder Google und es können eigene OpenID Connect-Anbieter konfiguriert werden.
Die Umleitungs-URI gibt an, zu welcher Website das AAD einen Benutzer nach erfolgreicher Anmeldung weiterleitet. Dafür wird zum einen ein localhost gewählt, welcher zum lokalen Debuggen verwendet werden kann. Außerdem wird die URI des Azure App Service, welcher zum Testen der Komponenten verwendet wird, mit der Endung *<App-Service-URL>/signin-oidc* eingetragen. Die Umleitungs-URIs können jederzeit geändert werden. Für die Front-Channel-Abmeldung wird die URL *<App-Service-URL>/signout-oidc* festgelegt. Für die Berechtigung wird „Administratoreinwilligung für openid- und offline_access-Berechtigungen erteilen“ gewählt.

5.2.3 Authentifizierung der registrierten Applikation

Nach dem Abschließen der Anwendungs-Registrierung wird die Authentifizierung der Applikation konfiguriert. Dazu wird zunächst unter Platformkonfiguration eine Web-Platform hinzugefügt, wodurch eine Webserver-Anwendung erstellt, gehostet und bereitgestellt wird. Neben der Umleitungs-URI, welche an dieser Stelle verändert werden kann, werden die ID-Token aktiviert. Diese sind für die Authentifizierung zuständig.

5.2.4 Benutzerflows

Benutzerflows werden beim AAD verwendet um Nutzern bestimmte Aktionen zu ermöglichen. Darunter fallen die Registrierung und Anmeldung, sowie die Profilbearbeitung und die Anforderung eines neuen Passworts, falls dieses vergessen wurde. Im Folgenden werden drei Benutzerflows konfiguriert, welche diese Aufgaben erfüllen. Für die Registrierung und Anmeldung wird ein gemeinsamer Betzerflow erstellt. Dabei wird zunächst ein Name und ein Identitätsanbieter festgelegt. Für letzteren wird *Email signin* konfiguriert.
Um die Sicherheit zu erhöhen, wird eine Multi-Faktor-Authentifizierung erzwungen. Zuletzt werden die „Benutzerattribute und Tokenansprüche“ festgelegt. Damit wird konfiguriert, was alles bei der Registrierung angegeben werden muss (wie zum Beispiel Name, Geburtsdatum, Wohnort...). Aus diesen Daten werden dann die Authentifizierungstoken (JWT) erzeugt. Für das Ziel der Wahrung des Datenschutzes ist an dieser Stelle vor Allem die Region, aus der ein Nutzer kommt von Bedeutung. Die Benutzerflows für die Profilbearbeitung und die Passwortzurücksetzung werden nach demselben Schema konfiguriert.

5.2.5 Verbindung der ASP.NET-Anwendung mit dem AAD

Um die Web-Anwendung mit dem AAD zu verbinden, muss im ASP.NET-Projekt das *appsettings.json*-File angepasst werden. Dazu wird der Code-Block in Listing 5.1 in dem File konfiguriert.

```
1  "AzureAdB2C": {
2     "Instance": "https://<AAD-Name>.b2clogin.com/tfp/",
3     "Domain": "<AAD primary domain>",
4     "TenantId": "<AAD tenant-id>",
5     "ClientId": "<AAD client-id>",
6     "CallbackPath": "/signin-oidc",
7     "SignUpSignInPolicyId":"<UserflowName-LoginRegistration>",
8     "ResetPasswordPolicyId":"<UserflowName-ResetPassword>",
9     "EditProfilePolicyId": "<UserflowName-EditProfile>",
10    "SignoutCallbackUrl": "/signout/<UserflowName-
         LoginRegistration>"
11   }
```

Listing 5.1: Appsetting-JSON-File-Konfiguration für die Verbindung mit dem AAD

Der Name des AAD wird in Form von *<AAD-Name>.onmicrosoft.com* für den Eintrag *Domain* und für *Instance* benötigt. Das *„/tfp/“* hinter der Domäne ist ein Platzhalter für den aktuellen Benutzerflow. Bei *TenantId* wird die Mandanten-ID des AAD eingetragen. Für die *ClientId* wird die Anwendungs-ID der registrierten Applikation im AAD verwendet. Der Callback-Pfad */signin-oidc* wird gemäß bei der Umleitungs-URI des AAD konfiguriert. In Zeile 7 bis 9 werden die Benutzerflows für die Anmeldung und Registrierung, die Passwortrücksetzung und Profilbearbeitung zugewiesen. Durch die Konfiguration von *SignoutCallbackUrl* mit */signout/<Benutzerflow AnmeldenRegistrieren>* wird ein Nutzer beim Abmelden wieder zur Anmeldung umgeleitet.

5.3 Azure SQL-Datenbank

Es werden für Europa und USA je eine SQL-Datenbank erstellt, um die Anforderung zu erfüllen, die Nutzerdaten in den Regionen seperat zu speichern. Bei der Erstellung wird ein DTU-basierter Basic Plan mit 0,5 GB Speicher und lokal redundantem Sicherheitsspeicher gewählt. Es wird ein Server erstellt. Hierbei wird ein Serveradministratorname und ein Passwort gesetzt. Damit können beispielsweise im Azure Portal über den Abfrageeditor die Daten und Tabellen manipuliert werden. Desweiteren sind diese ebenfalls in der Verbindungszeichenfolge enthalten.
In dem Azure Portal wird die Verbindungszeichenfolge kopiert, um diese für eine Verbindung zu der Datenbank nutzen zu können. Um die Verbindungszeichenfolge im Code nicht offen verwenden zu müssen, wird in einem späteren Schritt ein Key Vault eingerichtet. In Kapitel 5.4 wird darauf näher eingegangen.

Um auf die Datenbank in der ASP.NET Anwendung zugreifen zu können, wird zuerst eine Verbindung mit *SqlConnection()* aufgebaut, wie in Listing 5.2 zu sehen ist. Dafür wird als Parameter das Geheimnis aus dem Key Vault, welches die Verbindungszeichenfolge enthält, übergeben. Beim Laden einer Seite der Web-Anwendung wird zunächst überprüft, ob die Verbindung zur Datenbank bereits besteht. Sollte dies der Fall sein, wird die Verbindung geschlossen und neu geöffnet, um fehlerhafte Verbindungen zu vermeiden.

```
1 SqlConnection connection = new SqlConnection(secret);
```

Listing 5.2: Verbindung zur SQL-Datenbank

Nach erfolgreicher Verbindung, werden alle vorhandenen Daten eines Nutzers aus der Datenbank geladen und nach Daten sortiert. Dafür wird ein Modell zur Erzeugung und Speicherung der Daten erstellt. Die Daten werden in das Diagramm geladen und der Ladungsprozess der Seite ist beendet.

Es gibt zwei wesentliche Komponenten in der Datenbank. Die Nutzertabelle und die einzelnen Tabellen der Nutzer, mit den gesammelten Daten. Bei der Registrierung im AAD bekommt jeder Nutzer eine Identifikationsnummer zugewiesen. In der Nutzertabelle werden alle Nutzer-IDs und deren Regionen gehalten. Außerdem werden für jeden Nutzer zwei weitere Tabellen erzeugt, wie in Abbildung 5.3 dargestellt ist. In der Web-Anwendung gibt es zwei Seiten, auf denen Nutzer Daten eintragen können, wie in Kapitel 4 erläutert. Zum einen können sie ihren BMI berechnen. Dafür wird eine Tabelle erzeugt die mit *<Identifikationsnummer>_Fitness_Data* benannt wird. In dieser Tabelle werden ein Zeitstempel, das Gewicht und die Körpergröße gespeichert.
Auf der zweiten Seite können Daten zu einzelnen Workouts erfasst werden. Die zugehörige Tabelle wird *<Identifikationsnummer>_Workout_Data* benannt. Bei der Workout-Seite besteht ein Eintrag in der Tabelle aus dem Zeitstempel, dem Übungsnamen (z.B Joggen) und der Intensität, welche angibt wie lange oder wie häufig eine Aktivität ausgeführt wurde (z.B 30 Minuten oder 20 Wiederholungen).

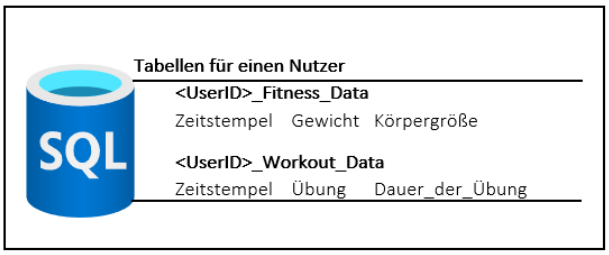


Abbildung 5.3: Tabellen in der SQL-Datenbank für einzelne Nutzer

Wenn ein Nutzer erfolgreich authentifiziert wurde, können seine Daten in die jeweilige Datenbank-Tabelle geschrieben werden. Wenn der Nutzer zum ersten Mal eine Eingabe in die Datenbank tätigt, wird eine neue Tabelle für ihn angelegt und seine Nutzer Identifikationsnummer in die Nutzertabelle geschrieben. Zusätzlich werden seine persönlichen Tabellen erzeugt. Wenn ein Anwender bereits Daten in seiner Tabelle stehen hat, können die vorangegangenen Schritte übersprungen werden und seine Angaben werden in die Tabelle geschrieben.

5.4 Azure Key Vault

Um eine offene Verwendung der Verbindungszeichenfolge der Datenbank im Projektcode zu vermeiden, wird pro Region ein Azure Key Vault angelegt, dies ist in Abbildung 5.4 dargestellt. In den Key Vaults werden die Verbindungszeichenfolgen der jeweiligen Datenbanken in der Region gespeichert. Bei der Erstellung des Key Vaults wird ein Administrator ausgewählt, dem der Zugriff auf den Tresor und des-

sen Geheimnisse gewährt wird. Für die Erstellung eines Geheimnisses werden ein Name und das zugehörige Geheimnis in Form von Strings festgelegt.

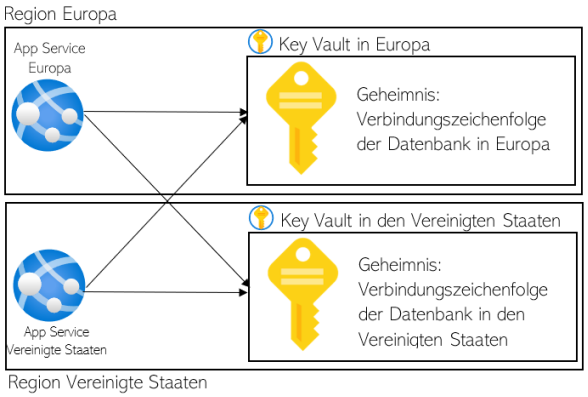


Abbildung 5.4: Gesamtarchitektur des Key Vaults

Die Verbindungszeichenfolgen werden zur Laufzeit aus dem Key Vault geladen. Um die Komponenten mithilfe eines App Services testen zu können, muss dem App Service die Zugriffsberechtigung auf die Geheimnisse des Key Vaults erteilt werden. Dies ist in Abbildung 5.5 am Beispiel einer Region dargestellt. In Kapitel 5.5 wird beschrieben, was von Seiten des App Services dafür konfiguriert werden muss.

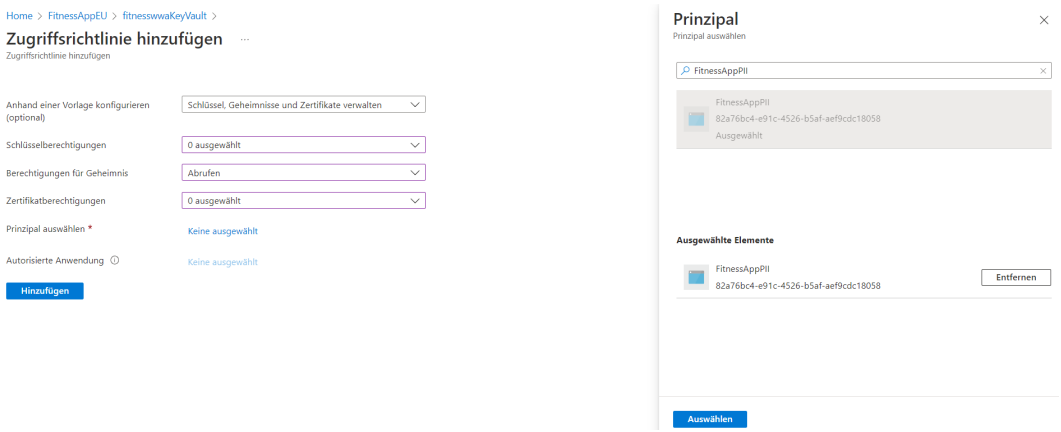


Abbildung 5.5: Erteilung des Zugriffs auf den Key Vault

Der vorgeschlagene Weg der offiziellen Microsoft Dokumentation zum Zugriff auf den Key Vault aus dem Projektcode heraus war nicht umsetzbar. Deshalb wird die Verbindung mit dem Key Vault mit der Funktion *GetConnectionSQL()*, wie in Listing 5.3 realisiert. Dafür wird zunächst eine Instanz des *AzureServiceTokenProvider* erzeugt, mit welcher wiederum eine Instanz des *keyVaultClient* erstellt wird. Die Tresor-URI wird als String gespeichert. Mithilfe dieser und dem Namen des Geheimnisses kann darauf hin das Geheimnis als Bundle aus dem Key Vault aufgerufen werden. Der

Rückgabewert enthält den Wert der Verbindungszeichenfolge.

```
string GetConnectionSQL(){

AzureServiceTokenProvider azureServiceTokenProvider = new
    AzureServiceTokenProvider();

KeyVaultClient keyVaultClient = new KeyVaultClient(new
    KeyVaultClient.AuthenticationCallback(
    azureServiceTokenProvider.KeyVaultTokenCallback));
string keyvault="https://<keyvault-name>.vault.azure.net/";

SecretBundle bundle= keyVaultClient.GetSecretAsync(keyvault,
    "<secret-name>").Result;
secret=bundle.Value;

return secret;
}
```

Listing 5.3: Code zum abfragen der Verbinungszeichenfolge aus dem Key Vault

Um eine Entscheidung treffen zu können auf welche Datenbank, bzw. Key Vault ein Nutzer zugreifen kann, gibt es eine Abfrage der Herkunft über den JWT. Ist ein Nutzer aus den Vereinigten Staaten wird seine Anfrage in den USA weitergeleitet. Um allen Ländern der Europäischen Union den Zugriff auf den Key Vault und somit auf die Datenbank zu geben, wird eine Liste an europäischen Ländern gehalten. Wenn ein angemeldeter Nutzer aus einem dieser Länder stammt, landet seine Anfrage bei dem europäischen Key Vault.

5.5 Testen der Grundkomponenten

Wie bereits in Kapitel 5.1 beschrieben sollen zunächst die Komponenten AAD, SQL-Datenbank und Key Vault zusammen gestestet werden. Das soll Fehlerquellen ausschließen, bevor ein Kubernetes Cluster mit ACR und die VNET Integration implementiert werden. Dafür wird ein Free App Service Plan erzeugt und die ASP.Net Anwendung in diesem deployed.
Um beim Key Vault die Berechtigung des App Service zum Abrufen von Geheimnissen gewähren zu können, muss im App Service die Identität aktiviert werden. Dadurch wird die Ressource beim AAD registriert und der Zugriff durch andere Dienste wie beispielsweise dem Key Vault erlaubt. Die URL des App Services wird

wie in Kapitel 5.2 beschrieben, als Umleitungs-URI bei der App-Registrierung im AAD eingetragen.
Um die Komponenten zu testen, wird die Anwendung über die URL des App Service aufgerufen. Über den Sign in Button wird ein neues Benutzerkonto registriert. Anschließend werden über die Passwort-Vergessen und Profil-Bearbeitungs Buttons die restlichen Benutzerflows des AAD getestet. Nach erfolgreicher Registrierung und anschließender Anmeldung, ist der Test der Funktionalität des AADs abgeschlossen. Die Umleitung, sowie die Benutzerflows wurden erfolgreich getestet.
Anschließend wird auf der Profil-Seite die Ausgabe der Claims des JWT überprüft, welche in einer Tabelle aufgelistet sind. Daraus können die Benutzer-ID und das Herkunfsland, welche für die Datenbank benötigt werden, eingesehen werden.
Abschließend wird eine Eingabe in die Textfelder der Webseite getätigt. Damit wird getestet, ob die Verbindungszeichenfolge aus dem Key Vault erfolgreich abgerufen werden kann und ob die Tabellen in der Datenbank für den soeben registrierten Nutzer erzeugt und mit den eingegebenen Daten gefüllt werden. Nach dem erfolgreichen Abschluss dieser Tests wird im den folgenden Kapiteln der App Service durch ein Kubernetes Cluster ersetzt.

5.6 Azure Container Registry und Docker

In diesem Kapitel wird beschrieben, wie die vorhandene ASP.NET Anwendung containerisiert und in eine ACR gestaged wird. Die ACR hält eine Container Image Instanz, die auf den replizierten Pods ausgeführt werden kann.
Dafür wird aus dem Code der ASP.NET Anwendung und den zugehörigen Libraries mit Docker ein Image gebaut. Zu diesem Zweck muss im Projektordner ein Dockerfile erzeugt werden, welches als Template für das Image genutzt wird. Mit dem Befehl in Listing 5.4 wird das Docker Image *fitnesswebapp* lokal erzeugt.

```
1 docker build -t fitnesswebapp -f Dockerfile .
```

Listing 5.4: docker build

Im nächsten Schritt wird im Azure Portal oder über die CLI eine ACR erstellt. Mit dem Befehl in Listing 5.5 wird das erstellte Image in das ACR gestaged.

```
1 az acr build --registry <ACR-Name> --image fitnesswebapp:
    latest .
```

Listing 5.5: Stagen eines Images in die ACR

5.7 Azure Kubernetes Service

Laut den Anforderungen soll sich in jeder Region ein AKS-Cluster befinden, welchem ein Load Balancer vorgeschaltet ist. Dabei wird die Architektur vorerst für eine Region implementiert und getestet. In den folgenden Unterkapiteln wird die Herangehensweise bei der Realisierung in zwei Ansätzen beschrieben.

5.7.1 Erster Ansatz

In einem ersten Ansatz wird versucht eine schematische Architektur wie in Abbildung 5.6 zu realisieren. Dabei soll prototypisch eine Web-Applikation in einem Kubernetes-Cluster mit der Anmeldemöglichkeit mit dem AAD bereitgestellt werden. Die Komponenten die dazu benötigt werden sind die bereits getesteten Komponenten aus Kapitel 5.5, sowie die in Kapitel 5.6 erstellte ACR.

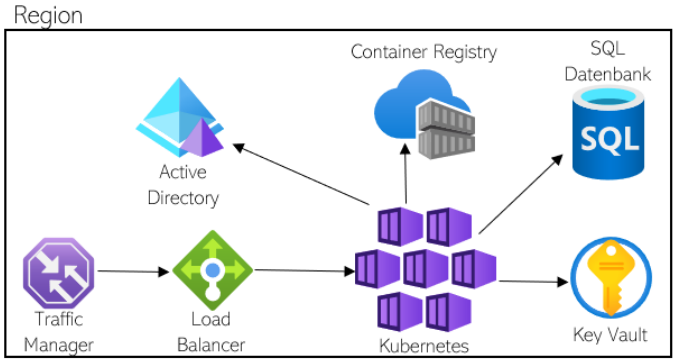


Abbildung 5.6: Schematische Architektur mit Kubernetes

Im ersten Schritt wird dafür ein Kubernetes Cluster erstellt. Das Docker Image liegt aus Kapitel 5.6 im ACR vor. Es wird ein Deployment und ein Service des Typs Loadbalancer eingerichtet. Hierfür werden zwei YAML-Files erstellt. In der *deployment.yaml* wird auf das Docker Image im Containerregistry zugegriffen und durch den Befehl *kubectl apply -f deployment.yaml* die Konfiguration in Kubernetes übernommen. Hier werden auch die Anzahl der Replikationen angegeben. Für Testzwecke wird eine Replikationsanzahl von zwei gewählt.
Ebenso wird die Loadbalancer.yaml für den Zugriff auf den Cluster angewendet. Diese zwei YAML-Files können im Anhang eingesehen werden. Zur Überprüfung der Pods und Services kann über die Befehle *kubectl get service* und *kubectl get pods* geprüft werden, ob diese korrekt ausgeführt werden. Bei den Services wird die externe IP-Adresse des Load Balancers angezeigt. Dem Load Balancer kann im Azure Portal

unter Konfigurationen ein DNS Name zugewiesen werden.
Nach der Konfiguration ist der Loadbalancer unter der Adresse *http://<Web-App-Name>.<Region>.cloudapp.azure.com* erreichbar. Beim Zugriff auf die IP-Adresse oder den DNS besteht das Problem, dass ein Anwender sich nicht mit dem AAD in die Web-Anwendung authentifizieren kann. Die Umleitungs URL im AAD erwartet von Entwicklern eine *HTTPS*-URI. Da der Loadbalancer nur eine *HTTP* zur Verfügung stellt, führt das zu einem Konflikt. Der Nutzer wird nicht authentifiziert und vom AAD nicht auf die Web-Anwendung zurückgeleitet. In der Umleitungs-URI der Adresszeile, durch eine manuelle Änderung von *HTTP* zu *HTTPS* kann eine Weiterleitung erzwungen werden. Dies stellt jedoch keine praktikable Lösung für das Problem dar.
Um das Problem zu umgehen, wird im Folgenden ein neuer Ansatz, unter Verwendung eines nginx Ingress und eines Zertifikats entwickelt und vorgestellt.

5.7.2 Zweiter Ansatz: Teil 1

Da es im ersten Ansatz Probleme mit der Verknüpfung des AAD und des Kubernetes Clusters gibt, wird im Folgenden die Intergration eines Nginx Ingress erläutert. Durch das Hinzufügen eines nginx Ingress und eines Zertifikates, soll die Umleitungs-URI das Unterstütze Protokoll verwenden und eine sichere TLS Verbindung aufbauen. Die schematische Architektur ist in Abbildung 5.7 zu sehen.

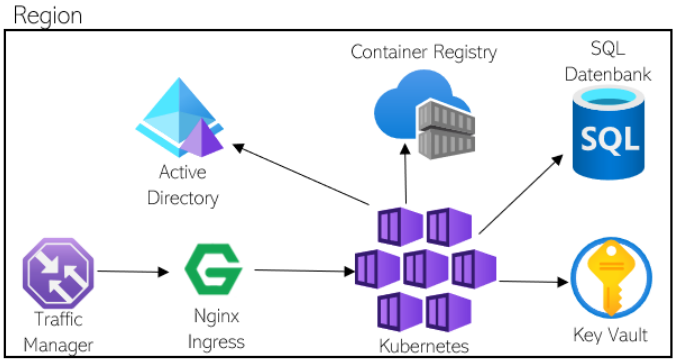


Abbildung 5.7: Schematische Architektur mit Kubernetes und einem Nginx Ingress

Ein Ingress verwaltet den Zugriff von außen auf den Service innerhalb eines Kubernetes-Clusters. Dabei können Funktionalitäten wie das Load Balancing und SSL genutzt werden [14]. Der Nginx Ingress ist ein Ingress Kontroller für Kubernetes, welcher Nginx als reverse Proxy und LoadBalancer nutzt [15].
Wie beim ersten Ansatz wird ein Kubernetes Cluster und die ACR erstellt. Durch den Befehl *az network public-ip create* wird eine öffentliche IP Adresse mit den Namen *nginx-LoadBalancer-IP* erzeugt.

Für die Nutzung von Nginx wird die Domain *fitnesswebappwwafitness.de* angemietet und ein Ressource Record eingerichtet. Dafür wird ein A-Record zur Namensauflösung der IPv4-Adresse, mit der erzeugten *nginx-LoadBalancer-IP* angelegt. Die gewählte TimeToLive (TTL) Zeitspanne beträgt 3600 Sekunden (1 Stunde).
Mithilfe von *Helm* wird das *ingress-nginx Repository* im Namensraum *ingress-basic* hinzugefügt und installiert. Bei der Installation wird die zuvor erzeugte *nginx-LoadBalancer-IP* als Load Balancer IP übergeben. Nun läuft die Anwendung durch das A-Record auf der Domain.
Um eine sichere Verbindung mit *HTTPS* zu gewährleisten, wird noch ein selbst signiertes Zertifikat mit openssl erstellt. Mit dem Befehl in Listing 5.6 können ein x509 Zertifikat und ein privater Schlüssel erstellt werden.

```
1 openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout <
    private key name> -out <certificate name> -subj "/CN=<
    common name>/O=<Organization Name>"
```

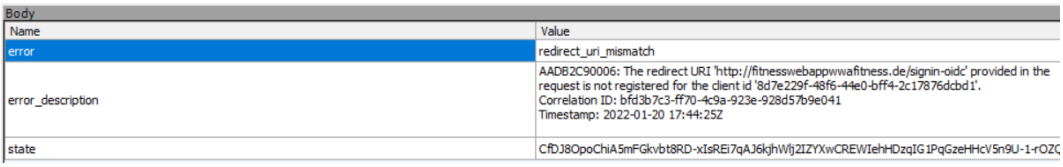
Listing 5.6: Erzeugung eines Zertifikates und eines privaten Schlüssels

Mit dem Befehl in Listing 5.7 wird ein Geheimnis im Kubernetes Cluster angelegt.

```
1 kubectl create secret tls <secret name> --key <private-key-
    name> --cert <certificate-names>
```

Listing 5.7: Erzeugung eines Geheimnisses im Kubernetes Cluster

In einem neuen Namespace wird mitthilfe von *Helm* das *cert-manager Repository* hinzugefügt und installiert. In diesem Ansatz werden zwei YAML-Files angelegt. Eine *deployment.yaml* wie in Ansatz eins und zuzätzlich eine *deployment-ingress.yaml* vom Typ Ingress.
In der *deployment-ingress.yaml* werden der TLS Host (FQDN) und das erzeugte Geheimnis im Cluster angegeben. Damit wird das Zertifikat der Domain zugewiesen und die Webseite gilt als sicher. Diese zwei YAMLS sind im Anhang zu finden. Mit dem *Debugging-Proxy-Server-Tool* Fiddler kann beobachtet werden, dass aus der Domain trotz dem zur verfügunggestellten Zertifikat, bei der Umleitungs-URL aus dem *HTTPS* ein *HTTP* macht. Dies ist in Abbildung 5.8 angezeigt.


Body

Name	Value
error	redirect_uri_mismatch
error_description	AADB2C90006: The redirect URI 'http://fitnesswebappwwafitness.de/signin-oidc' provided in the request is not registered for the client id '8d7e229f-48f6-44e0-bff4-2c17876dcbd1'. Correlation ID: bfd3b7c3-ff70-4c9a-923e-928d57b9e041 Timestamp: 2022-01-20 17:44:25Z
state	CfDJ8OpoChiA5mFGkvbt8RD-xIsREi7qAJ6kjhWlj2IZYXwCREWIehHDzqIG1PqGzeHHcV5n9U-1-rOZ(

Abbildung 5.8: Auszug aus dem Fiddler-Tool

Um eine Weiterleitung vom AAD zurück auf die Webseite in dem Pod zu ermöglichen, wird ein zuätzlicher Schritt getestet. Bei diesem Ansatz wird ein reverse Proxy zur Authentifizierung in den Pods integriert.

5.7.3 Zweiter Ansatz: Teil 2

Zur Authentifizierung der Nutzer in den Pods, wird ein OAuth-Proxy in die Architektur integriert. Dieser Ansatz stellt einen Erweiterung des zweiten Ansatzes dar. In Abbildung 5.9 ist die schematische Architektur dieses Ansatzes dargestellt.

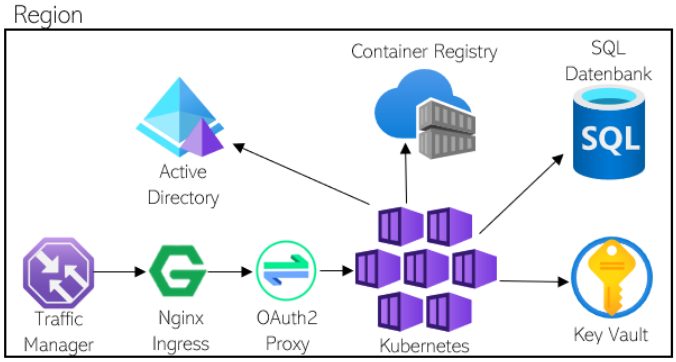


Abbildung 5.9: Schematische Architektur mit Kubernetes, Nginx Ingress und OAuth2 Proxy

Zusätzlich zu dem Nginx-Ingress und den Zertifikaten, wird für den Zugriff in die Pods ein OAuth2-Proxy in dem AKS-Cluster installiert, damit eine Authentifizierung mit dem AAD möglich wird.
Als Umleitungs-URL wird dafür *https://<FQDN>/oauth2/callback* im AAD gesetzt und eine *oauth2_proxy.yaml* konfiguriert und auf das Kubernetes-Cluster angewendet. Diese YAML wird zusätzlich zu denen in Kapitel 5.7.2 erstellt. In der *Ingress YAML* wird zusätzlich zu den Angaben aus Kapitel 5.7.2 eine Authentifizierungs-URL *(auth-url)* und eine Anmelde-URL *(auth-signin)* angegeben. Die YAML-Files können im Anhang eingesehen werden.
Beim Zugriff auf die Seite wird dem Nutzer ein Sign in Button angezeigt. Jedoch scheitert die Verbindung zu dem Server und ein *Internal Server Error 500* wird angezeigt.
Da die Verknüpfung von Kubernetes und dem AAD sich als problematisch erwiesen hat, wird ein neues Konzept entwickelt und realisiert, welches im nächsten Kapitel vorgestellt wird. Deshalb wird das Peering und die Integration in das VNET nicht in Kombination mit Kubernetes, sondern in den nächsten Kapiteln mit dem App Service realisiert.

Kapitel 6

Konzeption und Realisierung der App Service-Architektur

Da bei der Realisierung der Architektur mit AKS Probleme mit der AAD-Integration aufgetreten sind, wird eine neue Architektur konzipiert. Dazu werden die AKS-Cluster aus der vorherigen Konzeption durch skalierbare App Services ersetzt. In den folgenden Kapiteln werden die Konzeption und die Realisierung der neuen Architektur beschrieben. Die Komponenten Key Vault, SQL-DB und AAD werden mit leichten Modifikationen weiterverwendet. Diese Modifikationen werden in den folgenden Kapiteln beschrieben.

6.1 Konzeption der App Service-Architektur

In diesem Kapitel wird die Konzeption für die neue Architektur unter Verwendung von Azure App Service Plänen vorgestellt. Da die Komponenten AAD, SQL-DB, Key Vault und App Service in Kapitel 5.5 bereits zusammen getestet worden sind, wird das weitere Vorgehen wie folgt geplant.
Zuerst werden zwei VNETs für die beiden prototypischen Regionen erstellt und konfiguriert. Im Anschluss wird die Realisierung zur Integration der Komponenten in das VNET und für das Peering vorgestellt. Dazu wird pro Region ein App Service Plan konfiguriert, welcher in die virtuellen Netze integriert wird. Zusätzlich werden für die VNET-Integration nötige Konfigurationen an Key Vault und SQL-DB vorgenommen. Dabei werden die Endpunkte eingerichtet, welche nötig sind, um die Komponenten zu verbinden und einen Datenaustausch im VNET und das Peering zwischen den Regionen zu ermöglichen. Zum Abschluss wird ein Traffic Manager konfiguriert, welcher die Nutzer auf den geographisch nächsten Zugriffspunkt der Anwendung routet. In Abbildung 6.1 ist die neue Architektur zu sehen.

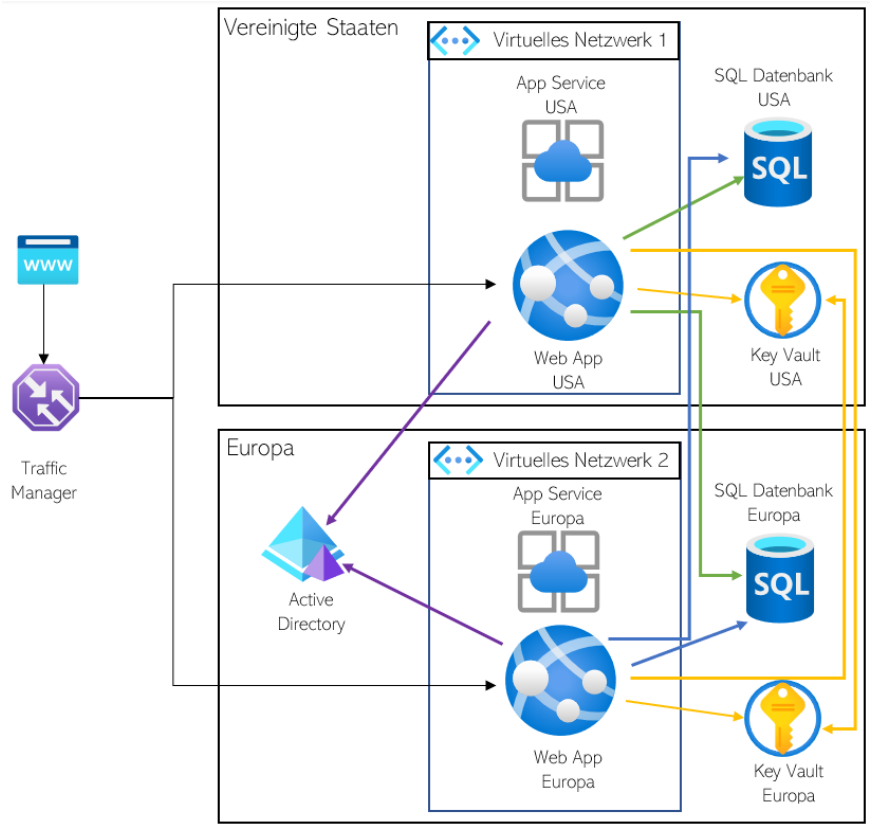


Abbildung 6.1: Konzeption der Gesamtarchitektur mit Azure App Service Plan

6.2 Azure virtuelles Netzwerk und Peering

Die Resourcen für die Regionen USA und Europa werden in virtuellen Netzwerken gekapselt. Dazu wird im Azure Portal pro Region ein virtuelles Netzwerk erstellt. Zu beachten ist dabei, dass alle Resourcen, die in das VNET integriert werden sollen in derselben Region angelegt sind, wie das VNet selbst. Außerdem ist es wichtig zu beachten, dass die beiden Adressräume der VNETS sich nicht überschneiden, da es sonst zu Problemen beim Peering kommt, wenn die beiden VNETS verbunden werden. Pro VNET werden zwei Subnetze eingerichtet. Das erste Subnetz dient der App Service-Integration. Das zweite Subnetz wird für die Endpunkte eingerichtet, welche in den nachfolgenden Kapiteln beschrieben werden. In Abbildung 6.2 sind die beiden VNETs mit ihren Subnetzen dargestellt.

Um später das Peering zwischen den beiden konfigurierten VNETs zu ermöglichen, wird bei einem der VNETs im Azure Portal ein Peering hinzugefügt. Neben zwei Namen für die Peering-Links wird das andere VNET ausgewählt, mit welchem es verbunden werden soll. Damit wird das Peering automatisch für die VNETs in beide Richtungen konfiguriert. Somit ist es möglich, dass beispielsweise von einer

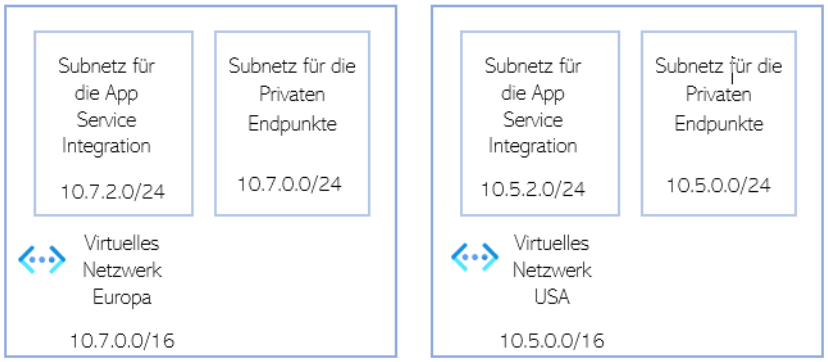


Abbildung 6.2: Aufbau der vituellen Netze und Subnetze

Anwendungsinstanz aus dem USA-VNET auf die Datenbank in Europa über private Endpunkte zugegriffen werden kann. Das ist notwendig aufgrund der Anforderung, dass ein Nutzer auf den geographisch nächsten Zugriffspunkt der Anwendung geroutet werden soll, die Daten europäischer Nutzer allerdings nach DSGVO-Norm nicht in den USA gespeichert werden dürfen. Die Konfiguration der Endpunkte und des geographischen Routings wird in den folgenden Kapiteln erläutert.

6.3 App Service mit VNET-Integration

Für die Bereitstellung der Web-Anwendung wird im Azure Portal ein App Service erstellt. Zuerst wird neben dem Namen die Region festgelegt. Dazu wird, wie zuvor erwähnt, dieselbe gewählt, in welcher auch das VNET liegt. Als Betriebssystem wird Linux gewählt und für die Runtime .NET 5. Außerdem wird die Zonenredundanz aktiviert, um die Anforderung der Hochverfügbarkeit zu erfüllen. Um eine Integration des App Services in das virtuelle Netzwerk zu ermöglichen, muss ein Premium v2 Service Plan erstellt werden. Niedrigeren Service Plänen unterstützen die Integration in ein virtuelles Netzwerk nicht. Außerdem bietet dieser Appservice-Plan die Möglichkeit einen Traffic-Manager zu verwenden, welcher nötig ist, um den Nutzer auf die geographisch nächste Instanz weiterzuleiten und er beinhaltet eine automatische Horizontale Skalierung mit bis zu 20 Instanzen, was der Erfüllung der Anforderung der Hochverfügbarkeit dient.
Nach der Erstellung wird der App Service unter Netzwerk in das VNET integriert. Dabei wird das zuvor erstellte VNET der entsprechenden Region und das für die App Service Integration vorgesehene Subnetz konfiguriert. Somit ist der App Service in das Virtuelle Netzwerk integriert. Damit der App Service über das AAD erreichbar ist, wird im Anschluss pro App Service noch eine Anwendung registriert und die Umleitungs-URI angepasst, wie in Kapitel 5.2 beschrieben.

6.4 Key Vault mit VNET-Integration

Der Key Vault wird grundsätzlich, wie in Kapitel 5.4 beschrieben konfiguriert. Um die Datenschutzbestimmungen einzuhalten, wird pro Region ein Key-Vault erstellt, welcher in derselben Region, wie das entsprechende VNET liegt. Um den Key Vault in das virtuelle Netz zu integrieren, werden private Endpunkte sowie die Firewall- und VNET -Einstellungen konfiguriert.
Bei den Firewall- und VNET -Einstellungen wird festgelegt, dass der Zugriff nur über private Endpunkte und ausgewählte Netzwerke zugelassen wird. Somit wird der Key Vault vom globalen Internet gekapselt und ist nur über Endpunkte oder speziell konfigurierte IP-Adressen erreichbar. Daher muss, um weiterhin Geheimnisse im Key Vault verwalten und erstellen zu können, der Zugriff über die IP des Entwicklers in der Firewall zugelassen werden.
Im Anschluss werden die privaten Endpunkte konfiguriert. Es werden pro Key Vault je zwei Endpunkte benötigt. Der erste wird für die Verbindung innerhalb der eigenen Region erstellt. Der zweite dient dem Peering zwischen den Regionen und ist dafür da, dass ein Nutzer aus Deutschland, wenn er in den USA ist, auf den Key Vault aus seiner eigenen Region zugreifen kann. Somit wird für jeden Key Vault in jeder Region ein Endpunkt erzeugt.
Bei der Erstellung wird neben dem Namen die entsprechende Region, in welcher der Key Vault integriert werden soll, konfiguriert. Für den ersten Endpunkt wird die eigene Region und für den zweiten die andere Region konfiguriert. Als Resourcentyp wird “Microsoft.KeyVault/vaults“und als Resource der entsprechende Key Vault gewählt.
Im Anschluss wird das VNET und das Subnetz festgelegt, in welchem der private Endpunkt erzeugt wird. Für das VNET wird im ersten Fall das VNET der eigenen Region und im zweiten Fall das VNET der anderen Region gewählt. Für das Subnetz wird jeweils das Subnetz für die Endpunkte in den entsprechenden VNETs, wie in Kapitel 6.2 beschrieben, verwendet. Somit gibt es in jeder Region einen Endpunkt, welcher auf den Key Vault zugreifen kann.
Anschließend wird der Private Endpunkt noch in die DNS-Zone integriert. Diese Einträge sind nötig, dass eine Verbindung mit den privaten Endpunkten hergestellt werden kann. Dasselbe wird für den zweiten Key Vault gemacht. Somit gibt für jede Region eine DNS-Zone für die Key Vaults, in welcher die Adressen der Endpunkte im VNET für die entsprechenden beiden Key Vaults hinterlegt sind.
Insgesamt gibt es nach der Erstellung aller nötigen Endpunkte für die beiden Key Vaults vier Endpunkte. Die erzeugten Endpunkte sind in Tabelle 6.1 zu sehen. Darin sind die Regionen, in welcher der Key Vault und der Privaten Endpunkt liegen, sowie ein Name und eine Beschreibung des Endpunktes angegeben.

Privater Endpunkt Name	Key Vault Region	Privater Endpunkt in VNET Region	Beschreibung
KV_DE	Germany West Central	Germany West Central	Endpunkt für Zugriff auf Key Vault DE innerhalb DE-VNET
KV_USA_to_DE	Germany West Central	East US	Endpunkt für Zugriff auf Key Vault DE aus USA-VNET
KV_USA	East US	East US	Endpunkt für Zugriff auf Key Vault USA innerhalb VNET USA
KV_DE_to_USA	East US	Germany West Central	Endpunkt für Zugriff auf Key Vault USA aus VNET DE

Tabelle 6.1: Endpunkte für den KeyVault

6.5 SQL-Datenbank mit VNET-Integration

Die Integration der SQL-Datenbanken in das VNET wird analog zu der Integration der Key Vaults im vorherigen Kapitel 6.4 durchgeführt. Zunächst wird im Azure Portal beim SQL-Server unter Firewalls und virtuelle Netzwerke der Zugriff auf öffentliche Netzwerke verweigert. Somit sind die Datenbanken nur noch über private Endpunkte erreichbar und vom Internet gekapselt. Außerdem werden wie für die Key Vaults ebenfalls vier Endpunkte erzeugt, welche in Tabelle 6.2 zu sehen sind. Diese erfüllen den selben Zweck wie die der Key Vaults und das Vorgehen ist dabei das selbe, wie in Kapitel 6.4 beschrieben. Der einzige Unterschied ist, dass bei Resourcentyp “Microsoft.Sql/servers“konfiguriert und im Anschluss die entsprechende Datenbank ausgewählt wird. Schließlich werden die Endpunkte der Datenbanken wieder in DNS-Zonen integriert.

Privater Endpunkt Name	SQL DB Region	Privater Endpunkt in VNET Region	Beschreibung
DB_DE	Germany West Central	Germany West Central	Endpunkt für Zugriff auf SQL DB DE innerhalb DE-VNET
DB_USA_to_DE	Germany West Central	East US	Endpunkt für Zugriff auf SQL DB DE aus USA-VNET
DB_USA	East US	East US	Endpunkt für Zugriff auf SQL DB USA innerhalb VNET USA
DB_DE_to_USA	East US	Germany West Central	Endpunkt für Zugriff auf SQL DB USA aus VNET DE

Tabelle 6.2: Endpunkte für die SQL-Datenbank

6.6 Traffic Manager

Aus den Anforderungen geht hervor, dass Nutzer unabhängig von ihrem Herkunftsland auf die geographisch nächste Instanz der Web Anwendung weitergeleitet werden sollen. Das bedeutet, dass ein deutscher Nutzer, wenn er in den USA ist, zu einer Instanz der Applikation in den Vereinigten Staaten geroutet werden soll. Zu diesem Zweck wird ein Traffic Manager konfiguriert. Dieser wird im Portal erstellt. Dabei wird ein Name festgelegt. Daraus ergibt sich später die URL:<*Name*>.*trafficmanager.net*, unter der er erreichbar ist. Als Routingmethode wird der Anforderung entsprechend „geographisch“ gewählt.
Nachdem der Traffic Manager erstellt ist, werden zwei Endpunkte konfiguriert. Als Endpunkt-Typ wird Azure-Endpunkt und als Zielresourcentyp App Service ausgewählt. Danach wird als Zielresource einer der App Services eingestellt und die gewünschte Region, in welcher die Anwendung erreichbar ist. Nach diesem Schema werden zwei Endpunkte für die beiden Regionen erstellt. Die Regionen die verwendet werden sind Europa und Vereinigte Staaten.

Kapitel 7

Evaluierung

In diesem Kapitel werden die Ergebnisse der Arbeit vorgestellt. Zunächst wird die prototypische Web Anwendung vorgestellt. Im zweiten Teil der Evaluierung wird die entgültige Gesamtarchitektur vorgestellt.

7.1 Evaluierung der prototypischen ASP.NET Anwendung

Es wurde erfolgreich eine prototypische Fitness-Web-Anwendung konzipiert und realisiert. Diese wurde mit dem Framework ASP.NET 5.0 implementiert. Sie beinhaltet drei Seiten, auf welche Nutzer über einen Reiter im oberen Bereich oder eine aus- und einklappbare Navigationsleite auf der linken Seite gelangen können. Diese sind *Workout-Tagebuch*, *BMI-Rechner* und *Profil*.
Das *Workout-Tagebuch*-Userinterface ist in Abbildung 7.1 zu sehen. Auf dieser Seite wird die Möglichkeit geboten, eine Art Fitness-Tagebuch zu führen. Dazu können über Textfelder Übungen mit dazugehöriger Intensität eingetragen werden, wie zum Beispiel Liegestützen und 30 Wiederholungen. Unter den Eingabefeldern befindet sich ein Balkendiagramm, welches die Übungen mit Daten versehen über die Zeit darstellt.
Die *BMI-Rechner*-Seite ist in Abbildung 7.2 dargestellt. Dort können Nutzer ihr Gewicht und ihre Körpergröße in Textfelder eingeben. Unter den Eingabefeldern wird ein Linien-Diagramm dargestellt, worin die beiden Größen und ein daraus errechneter BMI-Wert über die Zeit dargestellt werden.

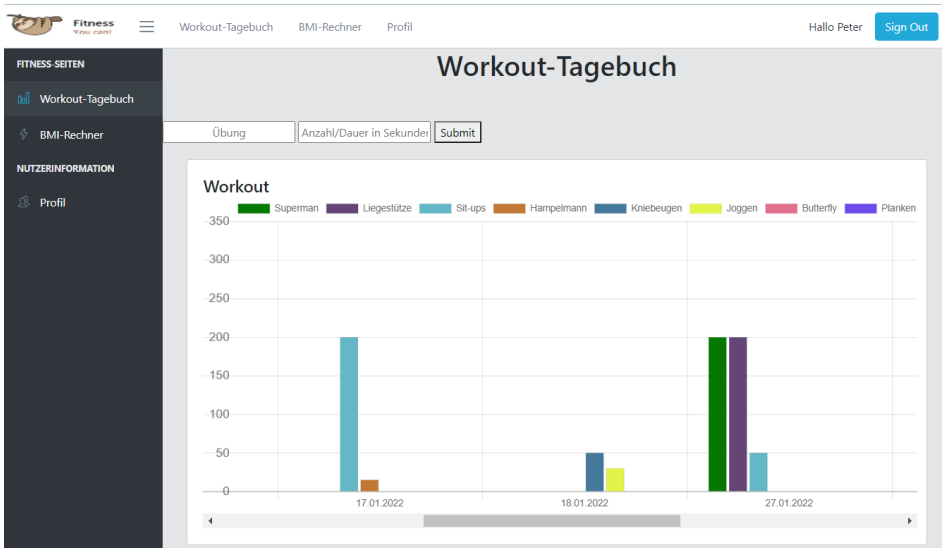


Abbildung 7.1: UI-Ergebnis der Workout-Seite

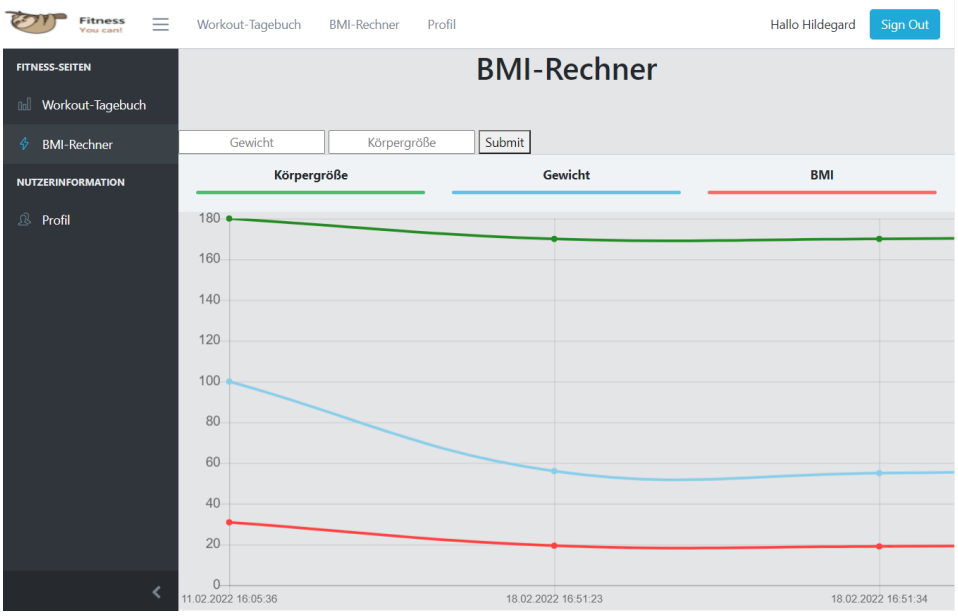


Abbildung 7.2: UI-Ergebnis der BMI-Seite

Da es sich bei dieser Arbeit um eine prototypische Anwendung handelt, werden auf der *Profi*-Seite die Claims aus dem JWT in einer Tabelle aufgelistet. Diese Informationen könnten beispielsweise für eine typische Profil-Seite genutzt werden. Für diese Arbeit soll damit veranschaulicht werden, welche Informationen in den JWTs enthalten sind und wie sie genutzt werden können. Mit der Information aus dem JWT darüber, woher ein Nutzer kommt, wird darüber entschieden, auf welche Resourcen (SQL-DB und Key Vault) er Zugriff erhält.

7.2 Evaluierung der Gesamtarchitektur

In diesem Kapitel wird die endgültige Architektur mit einer detaillierten Beschreibung vorgestellt. Diese ist in Abbildung 7.3 zu sehen.

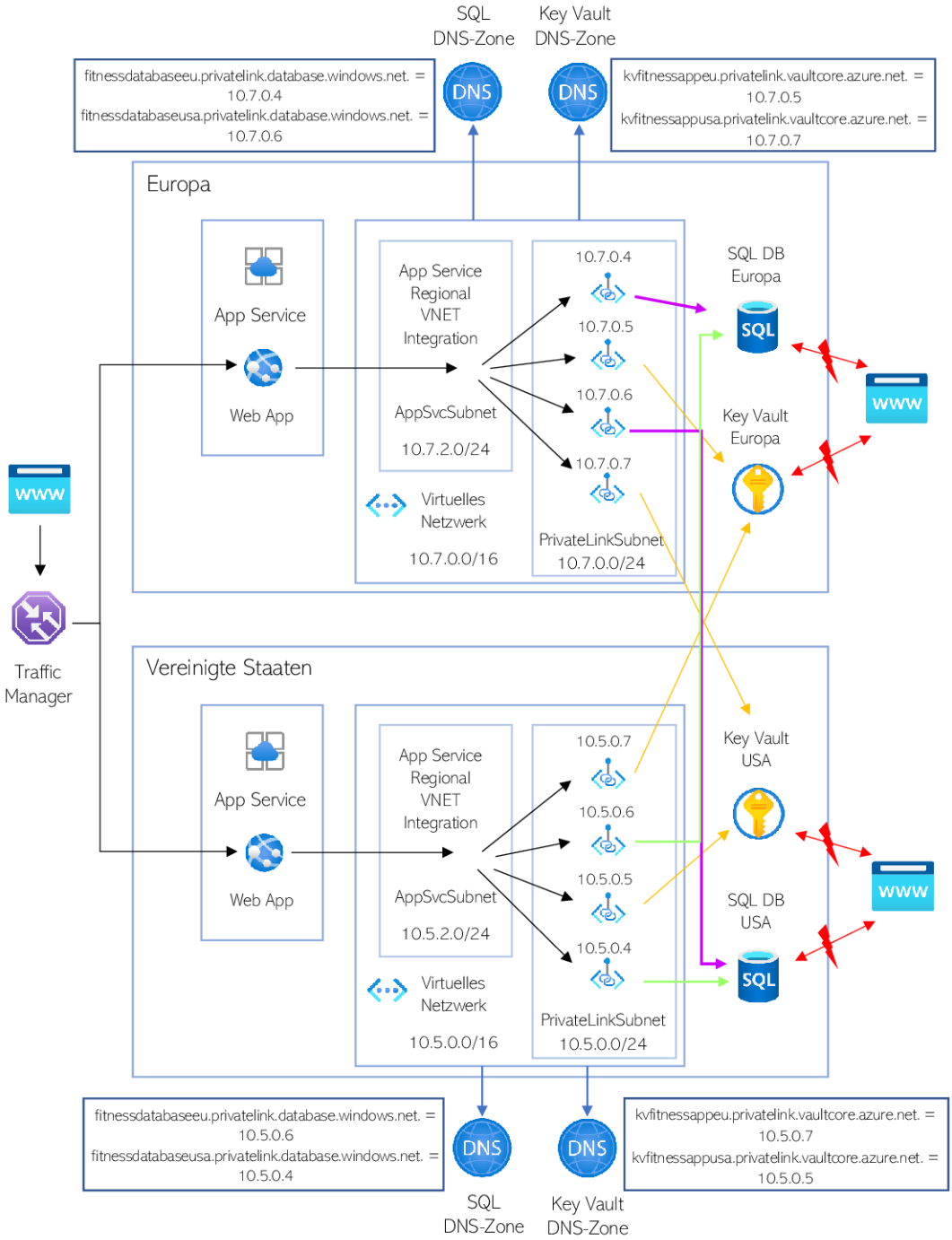


Abbildung 7.3: Ergebnis der Gesamtarchitektur

Die Entwicklung der Cloud-Architektur ist den Anforderungen gemäß mit Microsoft Azure umgesetzt und prototypisch für die Regionen Europa und Vereinigte Staaten bereitgestellt worden. Die Nutzer werden wie in den Anforderungen beschrieben über einen Traffic Manager auf die geographisch nächste Instanz der Anwendung geroutet.
Statt der anfänglich geplanten Umsetzung der Architektur mit AKS werden nun Azure App Services verwendet. Die Änderung der Architektur folgt aus der Anforderung, dass ein AAD zur Authentifizierung verwendet werden soll. Die Anwendung konnte in das Kubernetes Cluster integriert und ausgeführt werden. Allerdings gab es dabei Komptabilitätsprobleme mit dem AAD, wie in Kapitel 5.7 beschrieben. Eine Möglichkeit das Problem zu umgehen, wäre eine eigene Implementierung eines Logins. Durch die Wahl des App Service Plans können die Anforderungen an die Hochverfügbarkeit und horizontale Skalierbarkeit trotzdem erfüllt werden. Auch das Loadbalancing ist in dem App Service automatisch inbegriffen.
Die Authentifizierung wurde erfolgreich mittels JWT Token über das AAD realisiert. Dabei wurden Benutzerflows angelegt, die es Nutzern ermöglichen, sich zu registrieren und zu authentifizieren. Außerdem können das Passwort zurückgesetzt und das Profil bearbeitet werden. Das AAD wurde in Europa implementiert, um den Datenschutzbestimmungen nachzukommen, nach welchen europäische Daten nur in Europa gehalten werden dürfen. Da die Gesetze in den Vereinigten Staaten das nicht verlangen, können diese auch im selben AAD registriert werden.
Es wurden für jede Region ein eigener Key Vault und eine SQL-Datenbank konfiguriert. In den Key Vaults der Regionen sind die jeweiligen Verbindungszeichenfolgen für die Datenbanken gespeichert. Durch die Duplizierung der Komponenten kann die DSGVO eingehaltenen werden, nach der europäische Daten nur in Europa gespeichert werden dürfen. Mit der Information über die Herkunft eines Nutzers aus den JWT wird entschieden, auf welche Datenbank und Key Vault er zugreifen darf.
Die VNETs wurden wie geplant konfiguriert und über Peering verbunden. Die Adressbereiche der VNETs und Subnetze sind in Abbildung 7.3 zu sehen. Dabei wurde darauf geachtet, dass sie sich nicht überschneiden. Jedes VNET hat zwei Subnetze. Eins für die App Service Integration und eins für die Endpunkte. Die App Service Integration in das VNET ist erfolgreich umgesetzt worden.
In dem Subnetz für die Endpunkte sind pro Region vier Endpunkte erzeugt worden, wovon je zwei für die Key Vaults und zwei für die Datenbanken sind. Die Adressierung der Endpunkte innerhalb des VNETs ist ebenfalls Abbildung 7.3 zu entnehmen. Bei der Erzeugung der Endpunkte sind diese in DNS-Zonen integriert worden. Somit sind die Routing-Informationen bekannt, welche nötig sind, um auf die Datenbanken und Key Vaults zuzugreifen. Für jede Region gibt je eine DNS-Zone für die Key Vaults und die Datenbanken. Die Einträge in den DNS-Zonen sind in der Abbildung 7.3 zu sehen.

Somit kann die Anforderung erfüllt werden, wonach Nutzer über das VNET auf die Datenbank und den Key Vault in der Region, in der sie sich aufhalten oder mittels Peering auf die Datenbank und den Key Vault der anderen Region zugreifen können. Die Verbindung in das öffentliche Internet ist für die Key Vaults und Datenbanken blockiert worden. Der Zugriff ist nur über die privaten Endpunkte der VNETs möglich. Damit werden die Anforderungen an die Sicherheit der Datenübertragung erfüllt.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Literaturverzeichnis

[1] *Cloud Server versus Inhouse Server*. URL: https://www.axxiv.ch/cloud-server-versus-inhouse-server/.

[2] *Was ist Azure Active Directory B2C?* URL: https://docs.microsoft.com/de-de/azure/active-directory-b2c/overview.

[3] *App Service: Übersicht*. URL: https://docs.microsoft.com/de-de/azure/app-service/overview.

[4] *Azure Kubernetes Service*. URL: https://docs.microsoft.com/de-de/azure/aks/intro-kubernetes.

[5] *Was ist Azure SQL?* URL: https://docs.microsoft.com/de-de/azure/azure-sql/azure-sql-iaas-vs-paas-what-is-overview.

[6] *Grundlegende Konzepte von Azure Key Vault*. URL: https://docs.microsoft.com/de-de/azure/key-vault/general/basic-concepts.

[7] *Azure App Service-Plan – Übersicht*. URL: https://docs.microsoft.com/de-de/azure/app-service/overview-hosting-plans.

[8] *What is Kubernetes?* URL: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/.

[9] *Kubernetes components*. URL: https://kubernetes.io/de/docs/concepts/overview/components/.

[10] *Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms*. URL: https://www.researchgate.net/publication/336889240_Enabling_HPC_Workloads_on_Cloud_Infrastructure_Using_Kubernetes_Container_Orchestration_Mechanisms.

[11] *JSON Web Token*. URL: https://jwt.io/.

[12] *JWT Documentation from IETF*. URL: https://datatracker.ietf.org/doc/html/rfc7519.

[13] *Die Bestandteile eines JSON Web Tokens*. URL: https://de.wikipedia.org/wiki/JSON_Web_Token.

[14] *Ingress*. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/.

[15] *Ingress NGINX Controller*. URL: https://github.com/kubernetes/ingress-nginx.

Anhang A

YAML-Files: Erster Ansatz

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fitnesswebapp
spec:
  selector:
     matchLabels:
       app: fitnesswebapp
  replicas: 2
  template:
     metadata:
       labels:
          app: fitnesswebapp
     spec:
       containers:
       - name: fitnesswebapp
         image: acrfitnessappeu.azurecr.io/fitnesswebapp:
            latest
         imagePullPolicy: Always
         readinessProbe:
           httpGet:
             port: 44367
             path: /
         livenessProbe:
           httpGet:
             port: 44367
             path: /
```

```
27          resources:
28            requests:
29              memory: "128Mi"
30              cpu: "100m"
31            limits:
32              memory: "256Mi"
33              cpu: "500m"
```

Listing A.1: Erster Ansatz: deployment.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: fitnesswebapp-loadbalancer
5  spec:
6    externalTrafficPolicy: Cluster
7    type: LoadBalancer
8    selector:
9      app: fitnesswebapp
10   ports:
11   - protocol: TCP
12     port: 80
13     targetPort: 44367
```

Listing A.2: Erster Ansatz: loadbalancer.yaml

Anhang B

YAML-Files: Zweiter Ansatz - Teil 1

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: development-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/use-regex: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /1
    cert-manager.io/cluster-issuer: "prod"
spec:
  tls:
  - hosts:
    - fitnesswebappwwafitness.de
    secretName: mycert
  rules:
  - host: fitnesswebappwwafitness.de
    http:
      paths:
      - backend:
          service:
            name: fitnesswebapp
            port:
              number: 44367
        path: /(.*)
        pathType: Prefix
  defaultBackend:
```

```
    service:
      name: fitnesswebapp
      port:
        number: 44367
```

Listing B.1: Zweiter Ansatz: Teil 1: deployment-ingress.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fitnesswebapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fitnesswebapp
  template:
    metadata:
      labels:
        app: fitnesswebapp
    spec:
       containers:
       - name: fitnesswebapp
         image: acrfitnessappeu.azurecr.io/fitnesswebapp:
            latest
         imagePullPolicy: Always
         readinessProbe:
           httpGet:
             port: 44367
             path: /
         livenessProbe:
           httpGet:
             port: 44367
             path: /
         resources:
           requests:
             memory: "128Mi"
             cpu: "100m"
           limits:
             memory: "256Mi"
             cpu: "500m"
```

```
34 ---
35 apiVersion: v1
36 kind: Service
37 metadata:
38   name: fitnesswebapp
39 spec:
40   type: ClusterIP
41   ports:
42   - port: 80
43     targetPort: 44367
44   selector:
45     app: fitnesswebapp
```

Listing B.2: Zweiter Ansatz: Teil 1: deployment.yaml

Anhang C

YAML-Files Zweiter Ansatz - Teil 2

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: development-ingress
  namespace: development
  annotations:
    kubernetes.io/ingress.class: "nginx"
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/auth-url: "https://
       fitnesswebappwwafitness.de/"
    nginx.ingress.kubernetes.io/auth-signin: "https://
       fitnesswebappwwafitness.de/oauth2/callback"
spec:
  rules:
    - host: fitnesswebappwwafitness.de
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                 name: fitnesswebapp
                 port:
                   number: 44367
  tls:
  - hosts:
    - fitnesswebappwwafitness.de
```

```
26     secretName: mycert
27 ---
28 apiVersion: networking.k8s.io/v1
29 kind: Ingress
30 metadata:
31   name: oauth2-proxy
32   namespace: development
33   annotations:
34     kubernetes.io/ingress.class: "nginx"
35 spec:
36   rules:
37     - host: fitnesswebappwwafitness.de
38       http:
39         paths:
40           - path: /oauth2
41             pathType: Prefix
42             backend:
43               service:
44                 name: oauth2-proxy
45                 port:
46                   number: 4180
47   tls:
48   - hosts:
49     - fitnesswebappwwafitness.de
50       secretName: mycert
```

Listing C.1: Zweiter Ansatz: Teil 2: deployment-ingress.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: oauth2-proxy
5    namespace: development
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10       app: oauth2-proxy
11   template:
12     metadata:
13       labels:
```

```
14        app: oauth2-proxy
15    spec:
16      containers:
17      - env:
18          - name: OAUTH2_PROXY_PROVIDER
19            value: azure
20          - name: OAUTH2_PROXY_AZURE_TENANT
21            value: 773e219b-5e3d-4c53-aef9-02abe90ed3af
22          - name: OAUTH2_PROXY_CLIENT_ID
23            value: 191e8fa9-7ad5-4d05-98bb-f485ea77fb44
24          - name: OAUTH2_PROXY_CLIENT_SECRET
25            value: uvn7Q~v5oUHiLMLsaDS6xfNbx95X74r3PxEjM
26          - name: OAUTH2_PROXY_COOKIE_SECRET
27            value: S2nqcotHdfMjNUQlsML7Ww==
28          - name: OAUTH2_PROXY_HTTP_ADDRESS
29            value: "0.0.0.0:4180"
30          - name: OAUTH2_PROXY_UPSTREAM
31            value: "http://fitnesswebappwwafitness.de"
32        image: machinedata/oauth2_proxy:latest
33        imagePullPolicy: IfNotPresent
34        name: oauth2-proxy
35        ports:
36        - containerPort: 4180
37          protocol: TCP
38  ---
39  apiVersion: v1
40  kind: Service
41  metadata:
42    labels:
43      k8s-app: oauth2-proxy
44    name: oauth2-proxy
45    namespace: development
46  spec:
47    ports:
48    - name: http
49      port: 4180
50      protocol: TCP
51      targetPort: 4180
52    selector: app: oauth2-proxy
```

Listing C.2: Zweiter Ansatz: Teil 2: oauth2-proxy.yaml