

# **BACHELORARBEIT**

zur Erlangung des akademischen Grades  
„Bachelor of Science in Engineering“  
im Studiengang Informatik/Computer Science

## **Entwicklung einer skalierbaren Applikation basierend auf Cloud- und Serverless- Computing**

Ausgeführt von: Islam Elmoghazi  
Personenkennzeichen: 2010257188

Begutachter: Mag. Edin Pezerovic BSc MSc MSc

Wien, 30. April 2022

## Eidesstattliche Erklärung

„Ich, als Autor und Urheber der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idgF sowie Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idgF).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

---

Ort, Datum

---

Unterschrift

# Kurzfassung

Parallel zur monolithischen Architektur habe sich modulare Ansätze, z. B. Microservices und Funktionen etabliert. Aktuell werden aufgrund der Tatsache, dass Microservices und Funktionen im Trend sind, dieses stets empfohlen. Meistens werden nur die Vorteile erläutert und ein Muster als generische Lösung für alle Probleme verkauft. [1]

Dafür werden die unterschiedlichen Ansätze analysiert und in weiterer Folge auf eine hypothetische Applikation, ein e-Shop, angewandt. Parallel dazu werden unterschiedliche Alternativen, Kombinationen und unterstützende Technologien, auf Basis einer verwalteten Plattform z. B. Cloud, vorgestellt, um den Entwickler\*Innen- und Betriebsaufwand zu minimieren und zeitgleich die Skalierbarkeit und Leistungsfähigkeit des Systems zu erhöhen.

Das Ziel: Ein klares Bild dessen vermitteln, was existiert und was möglich ist. Wie von Cloud und Serverless profitiert werden kann und wie Applikationen effizient, in Bezug auf Kosten und Komplexität, skaliert werden können. Zusätzlich werden auch Edge-Cases z. B. Skalierung von verbindungsorientierten Services behandelt. Außerdem wird eine hauptsächlich lesende Applikation mit einer identen, auf Microsoft Orleans basierenden Applikation, in Bezug auf Leistung und Entwicklungsaufwand, verglichen.

**Schlagwörter:** Cloud, Architektur, Skalierbarkeit, Monolith, modularer Monolith, Microservices, Funktionen, Orleans, Azure

# Abstract

Modular concepts, e.g. microservices and functions, have established themselves parallel to monolithic architecture. Currently, due to the fact that microservices and functions are in trend, these are always recommended. Mostly, only the advantages are presented and a pattern is sold as a generic solution for all problems. [1]

For this purpose, the different concepts are analysed and subsequently applied to a hypothetical application, an e-shop. In parallel, different alternatives, combinations and supporting technologies, based on a managed platform e.g. cloud, are presented in order to minimise the developer and operational effort while increasing the scalability and performance of the system.

The goal: give a clear picture of what exists and what is possible. How to benefit from serverless and how applications can be scaled efficiently, in terms of cost and complexity. Additionally, edge cases, e.g. scaling of connection-oriented services, will also be covered. Furthermore, a read-heavy application is compared, in terms of performance and development effort, with an identical application based on Microsoft Orleans.

**Keywords:** cloud, architecture, scalability, monolith, modular monolith, microservices, functions, orleans, azure

# Inhaltsverzeichnis

1	Einleitung .....	7
2	Methodik .....	8
3	Definition .....	9
3.1	Monolith .....	10
3.1.1	Modularer Monolith .....	10
3.2	Microservices .....	11
3.3	Funktionen .....	12
4	Technologiestand .....	13
5	Konzept .....	14
5.1	Ausgangslage .....	14
5.2	Skalierbarkeit .....	15
5.2.1	Applikationsorientierte Skalierung .....	15
5.2.2	Serviceorientierte Skalierung .....	22
5.2.3	Funktionsorientierte Skalierung .....	25
5.2.4	Hybride Skalierung .....	27
5.2.5	Skalierung der Abhängigkeiten .....	29
5.3	Anwendungsbereiche .....	29
5.3.1	Grenzen .....	29
6	Umsetzung .....	30
7	Conclusio .....	38
	Literaturverzeichnis .....	40
	Abbildungsverzeichnis .....	44
	Tabellenverzeichnis .....	45
	Diagrammverzeichnis .....	46
	Abkürzungsverzeichnis .....	47
	Anhang A: ConnectionManager .....	48

Anhang B: TestApplikation .....	72
---------------------------------	----

# 1 Einleitung

Diese Arbeit soll bei der Entwicklung und Betrieb von komplexen Applikationen unterstützen, indem die wesentlichen Vor- und Nachteile der gängigsten Architekturen erläutert werden.

Ausgangslage der Arbeit ist eine hypothetische monolithische Applikation, dessen Konzept im Laufe der Arbeit angepasst und optimiert wird. Anfangs wird ein Lösungskonzept anhand von Microservices entwickelt. Progressiv werden Brennpunkte in Funktionen extrahiert.

Das Microservice-Funktionen-Hybrid Konzept wird wiederum gegen eine modularisierte monolithische Applikation, die auch auf Funktionen setzt, verglichen. Vergleichskriterien sind Entwicklungsaufwand, Komplexität beim Bereitstellen und die Komplexität der Verwaltung bzw. laufenden Wartung und des Betriebs.

In dieser Arbeit werden Konzepte, die unterstützenden Technologien und darauf basierend diverse Services von Cloud-Providern vorgestellt. Dafür entwickelte Codefragmente werden in C# verfasst, sofern nicht anders annotiert ist. Vorgestellte Konzepte und Ansätze werden nicht vollständig entwickelt, jedoch werden essenzielle Teilauszüge in Form von Proof of Concepts inkludiert.

Zielgruppe dieser Arbeit sind Personen und Entitäten, welche mithilfe von Cloud- und Serverless-Computing diverse Kriterien wie Skalierbarkeit, Leistung, (Saisonale-)Verfügbarkeit und auch SLAs verbessern wollen, ohne großen Entwickler\*Innen- und Betriebsaufwand.

## 2 Methodik

Serverless-Computing ist als Konzept relativ neu und wurde dementsprechend nicht in großem Umfang wissenschaftlich abgedeckt. Wissenschaftliche Arbeiten zur grundlegenden Funktionsweise existieren zwar, beschreiben jedoch mehr die Details als die tatsächliche Anwendung. Dementsprechend werden primär Artikel und Blogeinträge zu tatsächlichen Anwendungen als Quellen herangezogen. Zusätzlich werden auch Whitepaper und Beschreibungen von unterschiedlichen Service- und Cloud-Anbieter referenziert.

Zusätzlich haben sich einige der in weiterer Folge vorgestellten Konzepte, in Kombination mit den genutzten Technologien, im Rahmen diverser Tätigkeiten ergeben.



### 3 Definition

*„Serverless-Computing ist eine Methode zur Bereitstellung von Back-End-Diensten auf der Grundlage der tatsächlichen Nutzung. Mit einem Serverless-Provider können Benutzer Code schreiben und bereitstellen, ohne sich um die zugrunde liegende Infrastruktur kümmern zu müssen. Einem Unternehmen, das Backend-Dienste von einem Serverless-Anbieter erhält, wird die verwendete Rechenleistung berechnet. Es muss keine feste Bandbreite oder Anzahl von Servern reservieren und bezahlen, da der Service automatisch skaliert.“ [2]*

*„Serverless architecture is a way of building applications without needing to think about the underlying infrastructure that supports it. In a serverless model, instead of provisioning servers upfront to meet your needs, all you need to do is write code and push it to a serverless platform in the cloud. You don't have to think about traditional server management (or virtual machines, containers, or another traditional unit of infrastructure).“ [3]*

Cloud- und Serverless-Computing ist, aus organisatorischer Sicht, lediglich die Art der Bereitstellung und dadurch resultierende Abrechnung. Technisch lässt sich dieses Konzept in Form von Monolithen, Microservices und Funktionen umsetzen.

Für Entwickler\*Innen bedeutet das, dass die Applikation, nach der Bereitstellung, automatisch skaliert, ohne dass aktiv eingegriffen werden muss. Die Plattform stellt automatisch Ressourcen bereit bzw. baut diese auch automatisch ab. Jedoch muss berücksichtigt werden, dass nicht jede Applikation einfach skalieren kann. Es existieren durchaus viele Use-Cases, wo die Skalierung selbst in der Applikation berücksichtigt werden muss, z. B. Connection-basierte Applikationen wie Chats oder Gameserver, aber auch Applikationen, die ihren Zustand teilweise im Arbeitsspeicher vorberechnet.

Serverless betrifft aber nicht nur die Applikation, sondern auch die Datenbank. Größere Cloud-Provider wie z. B. Microsoft Azure und Amazon AWS sowie Datenbankanbieter wie z. B. MongoDB und CockroachDB bieten bereits serverlose Datenbanken an, wodurch Kunden nur für tatsächliche Nutzung, nach Anzahl der Transaktionen, bezahlen. Diese Dienste sind dazu ausgelegt, dynamisch und transparent zu skalieren. [4, 5, 6, 7]

## 3.1 Monolith

Ein Monolith ist eine vollwertige Applikation mit allen bzw. einem Großteil der Use-Cases und der gesamten Businesslogik. Das ist grundsätzlich immer der primäre Ansatz, wenn eine neue Applikation entwickelt wird und zeichnet sich durch folgende Kriterien aus:

- Einfache Entwicklung, da alle Services und Datenendpunkte in vollem Umfang verfügbar sind
- Einfache Inbetriebnahme, denn es wird lediglich eine Applikation bereitgestellt. Das Update muss im Vergleich zu Microservices nicht mit den anderen Services koordiniert werden
- Einfaches Testen, da alle Dimensionen in einem Prozess verfügbar sind
- Keine Abhängigkeiten zwischen den Services selbst
- Anfragen werden, aufgrund vom geringen Overhead, wesentlich schneller verarbeitet.
- Zusätzlich sind Neuzugänge im Team, aufgrund von der gebündelten Architektur, schneller produktiv

Monolithen eignen sich sehr gut für Applikationen in frühen Phasen und allgemein für Applikationen, die nur in Grenzen skalieren bzw. keine Unmengen an Anfragen verarbeiten sollten, denn die unterschiedlichen Services in Paket haben jeweils andere Anforderungen bzw. eigenen Performancemerkmale und skalieren dementsprechend unproportional. Üblicherweise muss die komplette Applikation skalieren, nur weil ein bestimmtes Service ausgelastet ist. [8]

Zusätzlich tendieren Entwickler\*Innen dazu den kürzesten Weg in der Entwicklung zu nehmen, auch wenn das in Technical-Debt resultiert. Dementsprechend gehören stark verknüpfte Services zur allgemeinen Landschaft von monolithischer Applikation, worunter in weiterer Folge die Wartbarkeit stark leidet. Zusätzlich fällt es bestehenden Entwickler\*Innen und Kenner\*Innen des Systems, sowie Neuzugängen schwer sich zu navigieren und Änderungen vorzunehmen, ohne unerwartete Komplikationen hervorzurufen. Die Entwicklung verlangsamt sich und die Effizienz sinkt. [9]

### 3.1.1 Modularer Monolith

Viel unterscheidet sich der modulare Monolith vom Konventionellen nicht. Technologisch gesehen stellen beide dasselbe dar, lediglich die Architektur nach Innen ist modularer und strukturierter. Im Endeffekt verfolgt das Entwicklungsteam diverse Clean-Code-Prinzipien und schafft klare Grenzen zwischen den jeweiligen Services. [10]

Jedes Service an sich ist ein eigenes Projekt und hat dementsprechend auch eigene Abhängigkeiten. Die Kommunikation zwischen den Services muss durch vordefinierte Interfaces gehen, wie sonst auch üblich bei Microservices. Die Services starten jedoch gemeinsam in einem Prozess und werden von außen als Einheit angesehen. [11]

Zusätzlich können Services mit unterschiedlichen Ansprüchen entkoppelt und separat bereitgestellt werden. Im Code selbst ändert sich dabei nur die Referenz auf das öffentliche Interface des jeweiligen Service. [11]

## 3.2 Microservices

Dieses Konzept beschreibt die Gliederung eines Gesamtsystem in individuelle Services die jeweils als Einheit und dementsprechend eigenständiges System angesehen werden. Diese Services verfügen üblicherweise über eigene Abhängigkeiten z. B. zu einer oder mehreren Datenbanken oder zu anderen Services. [12, 13]

Jedes Service übernimmt die Use-Cases einer bestimmten Gruppe. Beispielsweise existiert ein eigenes Service für die Userverwaltung. Dieses Service beinhaltet sämtliche Daten und Logik, die einen User betreffen würde, z. B. Anmeldung, Autorisierung, Passwort zurücksetzen usw.

Parallel dazu existiert, z. B. in einem e-Shop ein Service für alle angebotenen Artikel. Dieses Service besitzt die Datenhoheit und stellt die Zentrale, für sämtlichen Abfragen betreffend Artikel, bereit. Möchte ein User beispielsweise alle Artikel anzeigen lassen, so bekommt er diese vom entsprechenden Service.

Dieser Ansatz ist zwar in der Entwicklung komplizierter, die dadurch resultierende Applikation jedoch viel flexibler. So können unterschiedliche Sprachen, Technologien und Prinzipien parallel und dennoch unabhängig voneinander angewandt werden, um so, das Beste aus jedem Service zu holen. [1]

Zusätzlich kann jedes Service als Einheit, inklusive Abhängigkeiten, unabhängig vom Gesamtsystem, skaliert werden, sodass in Falle einer Überbeanspruchung eines bestimmten Service nur dieses skalieren muss. [1]

Wie allgemein auch bewährt hat jedes Konzept seine Nachteile. In diesem Falle sind folgende wesentlich:

- Die Koordination des Gesamtsystem gestaltet sich, durch die starke Fragmentierung, sehr kompliziert und riskant
- Laufende Änderungen machen die Koordination noch um einiges mühsamer

- Sehr hoher operativer Aufwand, da die Services pro Instanz ungefähr denselben Overhead mitbringen wie eine monolithische Applikation, die alle Services unterbringt.
- Ein Request braucht durchschnittlich länger, da die Kommunikation zwischen den Services einer Netzwerkverzögerung unterliegt. [14]

### 3.3 Funktionen

Funktionen sind die nächstkleinere Einheit von Microservices. Anstatt das Service als Einheit darzustellen, werden hier die jeweiligen Kommandos bzw. Funktionen eigenständig definiert und skaliert. [15]

Im Vergleich zu den anderen Ansätzen laufen Funktionen nicht in der typischen Runtime, sondern auf einer speziellen Plattform, dementsprechend kommt es auch schnell zum Vendor lock-in. Im Zuge dieser Plattform und der endlosen Möglichkeiten, in Bezug auf Skalierung, hat sich der Begriff „Serverless“ etabliert, denn Cloud- und Service-Provider bieten diese Funktionen im „Pay-as-you-go“-Modell an, wodurch der Endnutzer nur für die tatsächliche Anzahl an Exekutionen bezahlt. Dementsprechend sind der Einstieg und die allgemeine Nutzung in kleinem Umfang sehr kostengünstig und effizient.

Diverse Anbieter bieten parallel zum Verbrauchsbasierendem Tarif auch dedizierte Pläne, welche nach Ressourcennutzung nicht nach Anzahl der Transaktionen verrechnet. Solche Pläne sind für größere Vorhaben langfristig wesentlich kostengünstiger und kalkulierbarer. [16]

Funktionen kommen im Vergleich zu anderen Konzepten mit dem größten Overhead, da jeder einzelne Aufruf einer Netzwerkverzögerung unterliegt. Dementsprechend müssen Funktionen mit Bedacht angeboten werden. Einfache Operationen die als Funktionen ausgelagert wurde, würden das Gesamtsystem fragmentieren und dementsprechend hemmen.

Beispielsweise kann ein e-Shop seine PDF-Rechnungsgenerierung und E-Mail-Versand in Funktionen ausgliedern, denn diese Services sind im Vergleich zu den restlichen Use-Cases sehr rechenintensiv und können allein das Service so weit beanspruchen, dass Skalierung unumgänglich wird. Zusätzlich sind diese Funktionen in einer Common-Domäne für alle Services verfügbar.

Ausgelagerte Logik kann in weiterer Folge auch einfach angepasst, erweitert oder optimiert werden, ohne dass tatsächliche Services bereitgestellt werden müssen. Die abstrahierte Logik kann beispielsweise, transparent, durch eine komplett andere ersetzt werden. [15]

## 4 Technologiestand

Monolithen stellen den Grundstein der Software-Entwicklung dar, denn alle anschließenden Konzepte wurde mit dem Gedankenzug entwickelt den Monolithen zu ersetzen. [17]

Microservices und Funktionen sind zwar relativ neue, aber sehr ausgereifte Konzepte, die von sehr vielen einflussreichen Firmen und Organisationen produktiv und missionskritisch eingesetzt werden. [3]

Alle großen Cloudanbieter stellen die notwendige Infrastruktur zur Verfügung, um skalierbare Applikationen entwickeln und betreiben zu können. So bietet beispielsweise Microsoft in Azure sowohl App-Services als auch Funktionen an. Amazon bietet in AWS und Google in GCP ähnliche Services an. [16, 18, 19, 20, 21, 22]

Alle dieser Anbieter unterstützen die gängigsten Programmiersprachen z. B. JavaScript, Python, Java und C#. Parallel dazu werden auch unterstützende Technologien und Services z. B. Redis, Message Queue und Lastverteilung angeboten. Diese Services werden vollständig von der Cloud verwaltet und skalieren automatisch dem Anspruch entsprechend. [23, 24, 25, 26, 27]

Zusätzlich können Personen und Organisationen diese Infrastruktur selbst bereitstellen, indem beispielsweise Kubernetes benutzt wird. Basierend darauf existieren diverse Open-Source Projekte um Applikationen, Funktionen und alle unterstützenden Services zu betreiben und verwalten. Kubernetes selbst kann auf eigene Hardware oder in der Cloud, auf verwaltete Hardware, installiert und betrieben werden. [28]

## 5 Konzept

Unter Serverless Computing versteht man eine Architektur, in der die jeweiligen Services, auch gruppiert, eigenständige Komponenten darstellen und unabhängig voneinander arbeiten und skalieren.

Außerdem impliziert dieses Konzept nicht, dass eine Komponente immer eine eigene Maschine (virtuell oder physisch) darstellt. Ressourcen werden geteilt z. B. auf einer Maschine könnte Datenbank, Cache, Services und Funktionen parallel laufen.

**Serverless bedeutet im Endeffekt das die Services nicht an einen Server gebunden sind. Services verschieben und skalieren sich laufend, über jede verfügbare Ressource.**

Wichtig ist, dass, jedes Service seine Endpunkte ausreichend absichert und sämtlichen Traffic im Ein- und Ausgang validiert und verifiziert. Die Aufrufhistorie soll im Optimalfall zentral, extern und damit unabhängig vom Gesamtsystem abgelegt werden. Zusätzlich können diese mit Performance-Indikatoren bereichert werden, wodurch sowohl Traffic als auch Performance des Service nachvollzogen werden dann.

Aktuell etablieren sich diverse Standards zur Formatierung und Übermittlung von Metriken und Logs, namentlich „OpenTelemetry“. Dieser Standard löst das Problem mit unterschiedlichen Programmiersprachen, denn zuvor hatte jede Sprache eigene Best-Practices die sich nur schwer kollektiv verhalten haben. [29]

### 5.1 Ausgangslage

Ausgangslage ist ein einfacher e-Shop bestehend auf folgenden Services:

- Userservice
- Artikelservice
- Bestellungsservice
- Warenkorbsservice

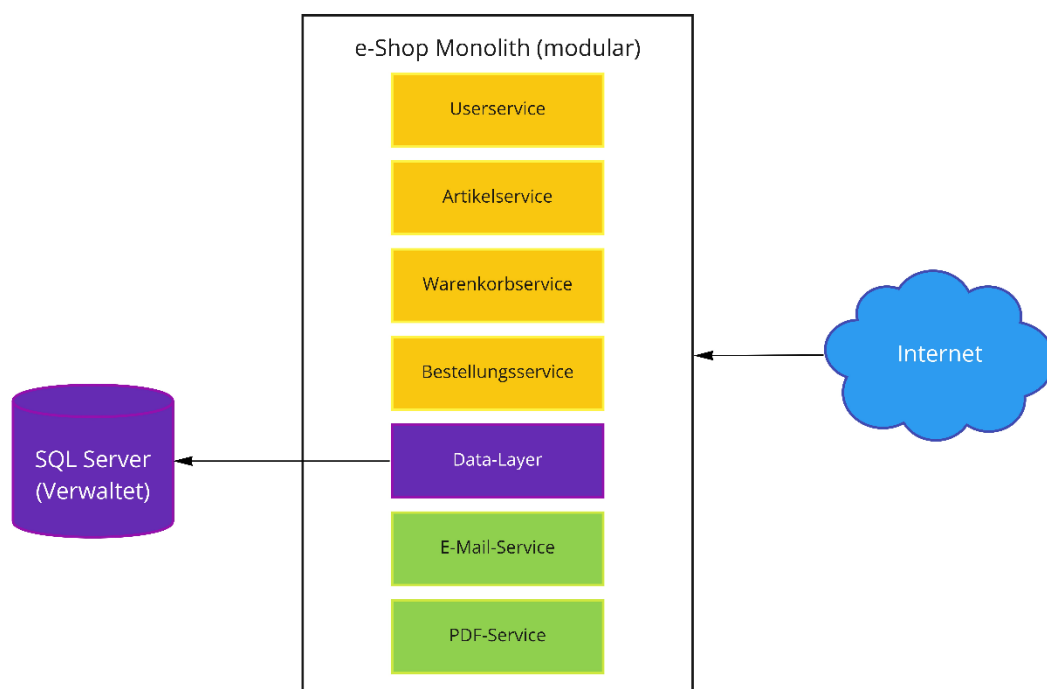
Zusätzlich zu diesen Services wird eine Datenbank betrieben, die alle Informationen beinhaltet. Außerdem werden im Bestellungsservice diverse PDFs, beispielsweise Rechnungen, Lieferscheine, Retourscheine und Mahnungen, generiert.

Parallel dazu senden alle Services E-Mails aus. Das Userservice verifiziert die E-Mail-Adresse bzw. veranlasst Änderungen am Userkonto erst nach Aufruf eines Bestätigungslinks in der zugesandten E-Mail. Artikelservice benachrichtigt Kunden und potenzielle Kunden

über aktuelle Sales und Trends. Im Bestellservice werden E-Mails zu Bestellungen, Rechnungen und Lieferungen verschickt. Der Warenkorb erinnert User\*Innen nach einer Weile, dass noch Artikel in seinem Warenkorb sind.

## 5.2 Skalierbarkeit

In der einfachsten Form wird der e-Shop als modularer Monolith entwickelt und beantwortet selbständig alle Anfragen. Zusätzlich ist die Nutzeroberfläche, in diesem Fall eine React-Anwendung, eingebunden und wird direkt ausgeliefert.



**Abbildung 1:** Architektur des Monolithen inklusive Abhängigkeiten (Gelb: öffentliche Services, Lila: Datenbankverknüpfung, Grün: private Services)

Die Architektur ist nach Clean-Code-Prinzipien entwickelt worden und Services sind strikt voneinander getrennt. Zusätzlich wurden alle externen Abhängigkeiten abstrahiert angebunden.

### 5.2.1 Applikationsorientierte Skalierung

Soll diese Applikation nur skaliert werden, so kann einfach eine zweite Instanz gestartet werden. Damit der Traffic jedoch zwischen den Instanzen aufgeteilt wird, wird ein Lastverteiler benötigt. Dieser wird von allen Cloud-Anbietern zur Verfügung gestellt. Alternativ bietet, beispielsweise in ihren Webapps, automatische Skalierung an, die Applikation skaliert beliebig und Azure schaltet automatisch einen Lastverteiler davor.

Choose how to scale your resource

**Manuelle Skalierung**

Feste Anzahl von Instanzen beibehalten

**Benutzerdefinierte Autoskalierung**

Hiermit wird basierend auf beliebigen Metriken eine Skalierung nach einem Zeitplan durchgeführt.

Benutzerdefinierte Autoskalierung

Name der Einstellung für die Autoskalierung \*

Ressourcengruppe

Standard \* Auto created scale condition

Warnung löschen

Die letzte Wiederholungsregel oder die Standardregel für die Wiederholung kann nicht gelöscht werden. Sie können stattdessen die Autoskalierung deaktivieren.

Skalierungsmodus

Basierend auf einer Metrik skalieren

Auf eine bestimmte Anzahl von Instanzen skalieren

Regeln

Es wird empfohlen, mindestens eine Regel für das Abskalieren zu konfigurieren. Klicken Sie zum Erstellen neuer Regeln auf [Regel hinzufügen](#).

Aufskali...

Zeitpunkt (Mittelwert) CpuPercentage > 70 Anzahl um 1 erhöhen

+ Regel hinzufügen

Instanzgrenzwerte

Minimum 1

Maximum 10 ✓

Standard 1

Zeitplan

Diese Skalierungsbedingung wird ausgeführt, wenn keine der anderen Skalierungsbedingungen zutrifft.

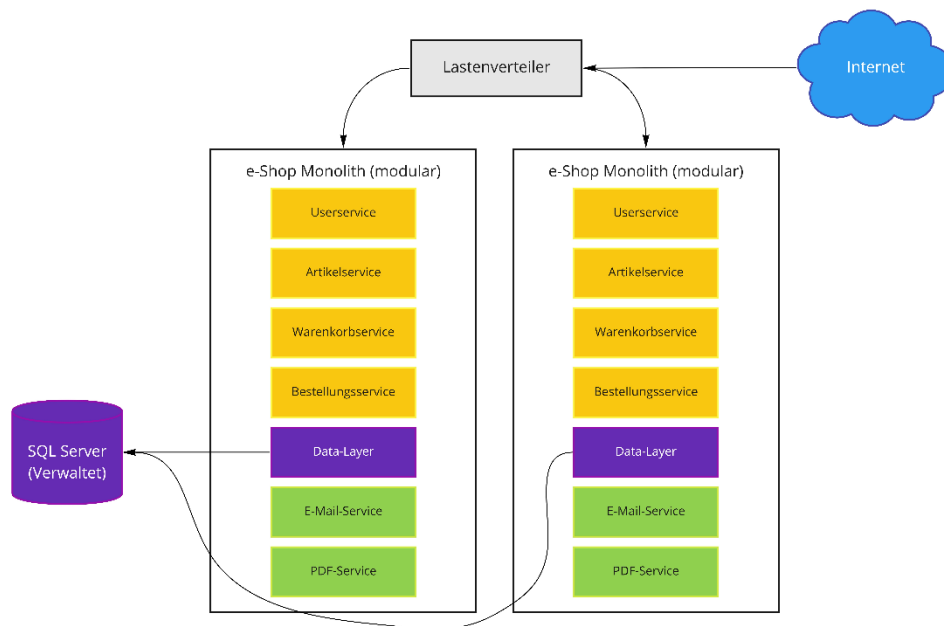
+ Skalierungsbedingung hinzufügen

Abbildung 2: Auszug aus der automatischen Skalierung in Azure Web Apps

In der Abbildung ist ein Auszug aus Azure zu sehen. Die aktuelle Applikation skaliert automatisch, sobald die durchschnittliche Prozessorauslastung 70% überschreitet. Die Prozessorauslastung wird Instanz-übergreifend kalkuliert.

16





**Abbildung 3:** Monolith mit eigenem Lastverteiler

Im Falle das der eigene Cloud-Anbieter keine vollständig verwalteten Applikationen anbietet kann ein Lastverteiler die Anfragen aus dem Internet verteilen.

Grundsätzlich gibt es mehrere Verfahren wie ein Lastverteiler arbeitet. Primär in Anwendung kommen das Rundlauf-Verfahren, auch „Round-Robin“ genannt oder Zufällige Zuweisung. Außerdem existieren viele Verfahren, die je nach Use-Cases sinnvoller sind. Beispielsweise kann bei einer Mandantenfähigen Applikation zuerst nach Mandanten gruppiert werden dann ein anderes Verfahren angewandt werden, dies versichert, dass ein bestimmtes Mandat nur von einer relativ fixen Teilmenge aller Server verarbeitet wird und Daten nicht über alle Server hinweg verstreut werden.

In diesem Fall kann der e-Shop problemlos skalieren, da der Warenkorb nur angemeldeten Usern zur Verfügung steht. Das ist jedoch kein üblicher Ansatz, denn anderen Shops bieten den Warenkorb auch für anonyme Nutzer\*Innen an. Dieser wird dann mittels Sitzungskennung verknüpft. Dieses Identifikationsmittel, ausgestellt von einer Instanz, steht in einer direkten Beziehung zum Browser und ist den anderen Instanzen nicht bekannt. Das kann gelöst werden, indem der Warenkorb inklusive Sitzungskennung in der Datenbank persistiert wird. Üblicherweise werden für solche Use-Cases ein verteilter Cache, z. B. Redis, eingesetzt.

Zusätzlich kommen auch kompliziertere Use-Cases vor, die beispielsweise Verbindungsbasierend arbeiten wie z. B. Gameserver, Chatserver oder allgemein Echtzeitbenachrichtigungen. Solche Anwendungen benötigen immer zusätzliche Planung. Je nach Anforderungen existieren mehrere Möglichkeiten, die Verbindungen zu verteilen.

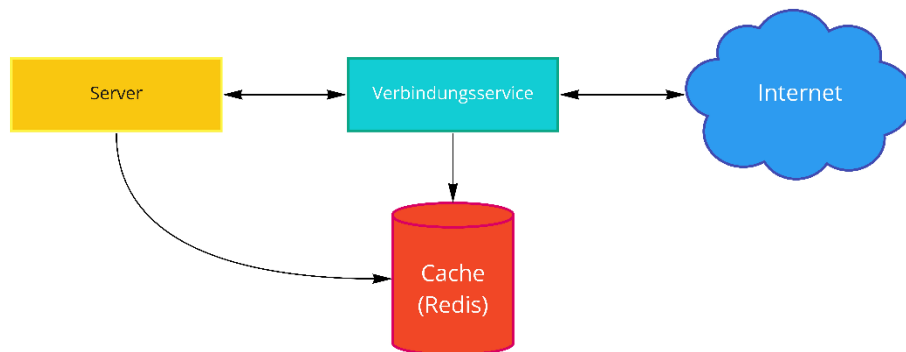
## Nachrichtenbasierte Verbindungen skalieren

Beispielsweise verbindet sich ein Klient mit dem Chatserver via SignalR, mit WebSockets als unterliegende Technologie. In diesem Fall kann ein verwaltetes SignalR-Service benutzt werden, welche die Verbindungen verwaltet und anschließend mittels Messaging mit dem Server kommuniziert. Der Server arbeitet dementsprechend nicht Verbindungsorientiert, sondern reagiert auf Anfragen und hat parallel eine Liste mit verbundenen Klienten, die gegebenenfalls benachrichtigt werden können. [30]

Dementsprechend besteht keine direkte Abhängigkeit zwischen Client und Server. In vielen Fällen bedeutet das, jeder Server kann jede Anfrage bearbeiten. Zusätzlich können Anfragen einer Verbindung von mehreren Servern verarbeitet werden.

## Kontinuierliche Verbindungen skalieren

Dennoch existieren Use-Cases, beispielsweise ein Gameserver, der stets Anfragen von einer bestimmten Gruppen an Klienten beantworten muss, weil diese innerhalb einer Region sind. In diesem Fall muss das Verbindungsservice die Anfragen immer an den richtigen Server propagieren.



**Abbildung 4:** Grobe Architektur wie eine Verbindung immer an einen bestimmten Server weitergeleitet wird

Das Verbindungsservice trägt die neue Verbindung im Cache ein und verbindet diesen mit einem beliebigen Server. Im Cache ist dementsprechendes, beispielsweise, folgendes hinterlegt:

Region	Server
A	127.0.0.1:63000
B	127.0.0.1:63001
C	127.0.0.1:63001

**Tabelle 1:** Liste mit den Regionen und dem entsprechenden Server

ConnectionId	State	ClientId	Region
f4c2f0b6	initial	null	null

**Tabelle 2:** Eintrag unmittelbar nach Verbindungsaufbau

Nun wird die Verbindung an einen zufälligen Server weitergeleitet. Dieser entnimmt dem Cache den Status entgegen und beginnt den Client zu authentifizieren. Sobald dies erfolgt ist, trägt der Server die „ClientId“ ein. Die „ClientId“ ist eine applikationsspezifische Userkennung. Weiters ermittelt der Server, ob ein anderer Server diese\*n Nutzer\*In übernehmen kann, beispielsweise weil diese\*r Nutzer\*In in eine Region eines anderen Servers fällt. Sollte es keinen passenden Server geben, so übernimmt der aktuelle Server diese Region.

ConnectionId	State	ClientId	Region
f4c2f0b6	ready	359a05c95abd	A

**Tabelle 3:** Eintrag nach der Authentifizierung

Kann der aktuelle Server den Client bedienen, so bleibt die Verbindung aufrecht und die tatsächliche Kommunikation beginnt. Ist der behandelnde Server ein anderer, so wird die Verbindung zwischen Server und Verbindungsservice geschlossen. Das Verbindungsservice entnimmt dem Cache den aktuellen Zustand und erkennt, dass der Server einen anderen vorschlägt. Eine Verbindung wird mit dem Server der zugewiesenen Region initiiert.

Der neue Server, der diese Region tatsächlich verwaltet, nimmt die Verbindung an, passt den Datensatz im Cache an und beginnt die tatsächliche Kommunikation.

ConnectionId	State	ClientId	Region
f4c2f0b6	connected	359a05c95abd	A

**Tabelle 4:** Eintrag, wenn die tatsächliche Kommunikation begonnen hat

Sollte die Authentifizierung nicht erfolgreich ablaufen, oder der Server die echte Verbindung schließen wollen, so wird als Status, auch „State“, „closed“ hinterlegt. Das Verbindungsservice schließt die entsprechende Verbindung und löscht im Anschluss den Datensatz vom Cache.

Regionen sollten entsprechend definiert werden, sodass ein Server diese komplett übernehmen kann. Falls notwendig, kann ein Server auch mehrere Regionen übernehmen. Sollte ein Server ausfallen, so merkt das Verbindungsservice das und vergibt die Regionen an andere Server. Alle Verbindungen der beeinträchtigten Regionen werden an die neuen Server weitergeleitet, ohne dass die tatsächlichen Verbindungen abbrechen. Im Optimalfall merken die Klienten nichts von diesem Manöver, dafür muss der Zustand der Region

persistiert oder zumindest im Cache abgelegt werden, sodass der neue Server diesen einfach übernehmen kann.

Dieses Konzept wurde in Form eines Proof of Concepts implementiert (Siehe Anhang A). Beachtenswert ist jedoch, dass nicht der komplette Funktionsumfang implementiert wurde. Einfachheitshalber wurde angenommen, dass jede\*r Nutzer\*In eine eigene Region repräsentiert. Nutzer\*Innen wurden bei Serverabsturz migriert, ohne, dass der Client etwas merkt.

```
C:\Users\babaelmo\source\repos\ConnectionManager\ConnectionManager.Server\bin\Debug\net6.0\ConnectionManager.exe
Client 00000000-0000-0000-0000-000000000000 data received!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd connection id received and activated!
dbug: Program[0]
ConnectionHandler disposed
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data received!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd processing line: ping!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data received!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd processing line: status!

C:\Users\babaelmo\source\repos\ConnectionManager\ConnectionManager.Server\bin\Debug\net6.0\ConnectionManager.exe
info: Program[0]
SocketListener started listening on 0.0.0.0:54910

C:\Users\babaelmo\source\repos\ConnectionManager\ConnectionManager\bin\Debug\net6.0\ConnectionManager.exe
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd server activated the connection!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data being redirected to client!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd sent data! Server accepting messages: True
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data being redirected to client!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd sent data! Server accepting messages: True
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data being redirected to client!

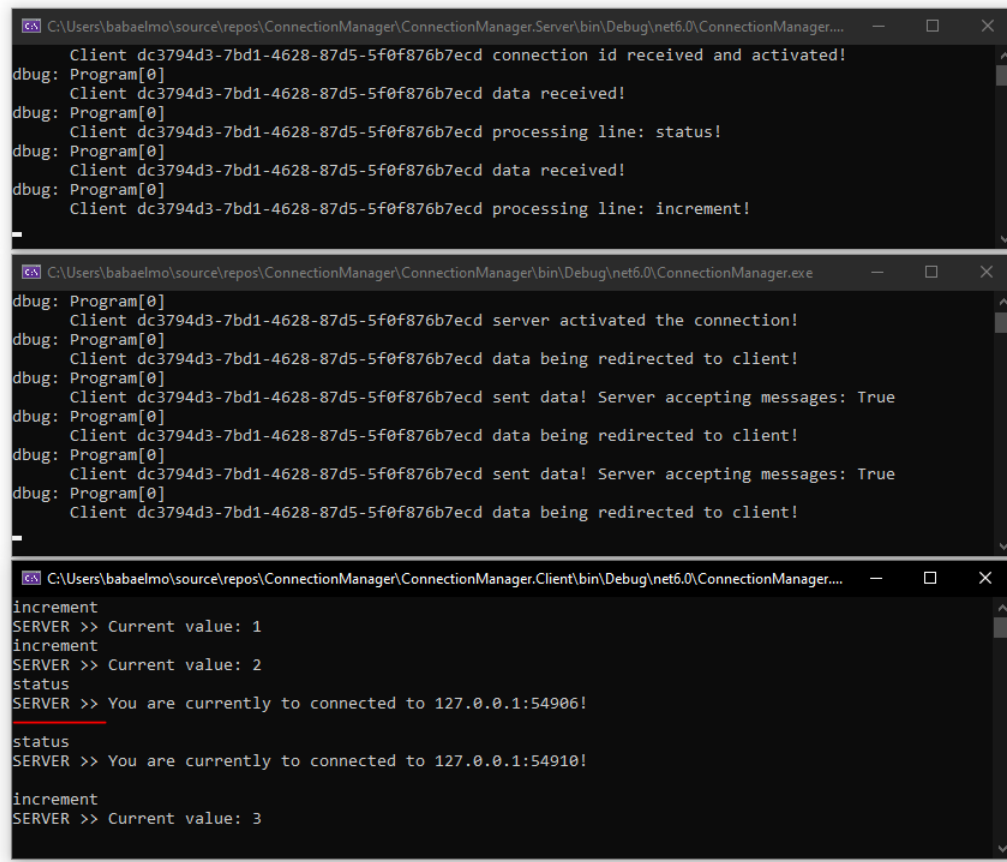
C:\Users\babaelmo\source\repos\ConnectionManager\ConnectionManager.Client\bin\Debug\net6.0\ConnectionManager.exe
SERVER >> authentication required
authenticate test
ping
SERVER >> pong 11.05.2022 16:24:34
status
SERVER >> You are currently to connected to 127.0.0.1:54906!
```

**Abbildung 5:** Ausführung vom Proof of Concept: Ein Lastverteiler, ein Klient und zwei Server

Wie der Abbildung entnommen werden kann, läuft der Lastverteiler und leitet jegliche Last an zwei verfügbare Server. Der Lastverteiler kennt die Server eigentlich nicht und versucht auch nicht grundlos eine Verbindung aufzubauen, dies kann man der zweiten Konsole entnehmen, dieser Server wurde noch nicht angesprochen, obwohl er verfügbar wäre.

Die Server tragen sich selbständig im Cache ein und sind auch dafür verantwortlich sich wieder auszutragen. Sollte der Server einfach abstürzen und diesen Schritt überspringen, so kommt es zu Weiterleitungen an nicht verfügbare Server und der Lastverteiler lehnt die Verbindung vom Klienten ab. Zusätzlich identifizieren sich die Server über ihren Endpunkt. Beide Entscheidungen sind fatal und würde das Skalieren bzw. zuverlässige Betreiben hindern. Dementsprechend muss folgendes angepasst werden:

- Der Server muss sich mittels Endpunkts und Zeitpunkt des Starts identifizieren, dementsprechend kann, wenn sich ein neuer Server mit demselben Endpunkt, aber späteren Zeitpunkt meldet, davon ausgegangen werden, dass der alte Server nicht mehr verfügbar ist.
- Der Lastverteiler muss beim Verbindungsaufbau zu einem Server bereits einen zweiten Server bereit haben, der im Problemfall die Verbindung bekommt. Beispielsweise wird aktuell zufällig von den fünf am wenigsten beanspruchten Servern ausgewählt. In weiterer Folge, sollen zwei oder drei Server ausgewählt werden und bei Verbindungsschwierigkeiten übernehmen.
- Der Lastverteiler muss Verbindungsprobleme aufzeichnen und über ein bestimmtes Zeitfenster verfolgen. Stellt sich ein Server als nicht zuverlässig, so wird dieser progressiv abgebaut bzw. die Gewichtung verringert. Im Optimalfall kann der Lastverteiler mit der bereitstellenden Plattform kommunizieren und automatisch den Server austauschen lassen.



```
C:\Users\babaelmo\source\repos\ConnectionManager\ConnectionManager.Server\bin\Debug\net6.0\ConnectionManager.exe
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd connection id received and activated!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data received!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd processing line: status!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data received!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd processing line: increment!

C:\Users\babaelmo\source\repos\ConnectionManager\ConnectionManager\bin\Debug\net6.0\ConnectionManager.exe
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd server activated the connection!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data being redirected to client!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd sent data! Server accepting messages: True
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data being redirected to client!
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd sent data! Server accepting messages: True
dbug: Program[0]
Client dc3794d3-7bd1-4628-87d5-5f0f876b7ecd data being redirected to client!

C:\Users\babaelmo\source\repos\ConnectionManager\ConnectionManager.Client\bin\Debug\net6.0\ConnectionManager.exe
increment
SERVER >> Current value: 1
increment
SERVER >> Current value: 2
status
SERVER >> You are currently to connected to 127.0.0.1:54906!
status
SERVER >> You are currently to connected to 127.0.0.1:54910!
increment
SERVER >> Current value: 3
```

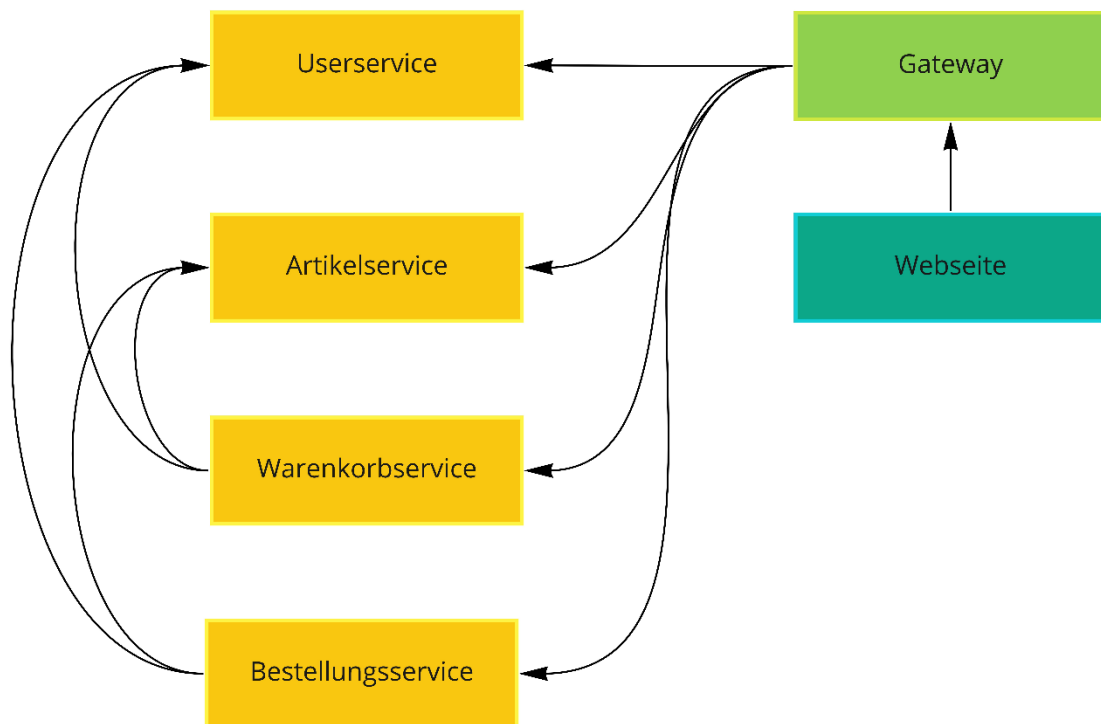
**Abbildung 6:** Der aktive Server ist abgestürzt, die Applikation kommuniziert nun mit einem anderen Server, ohne dass die Verbindung abbricht

Der Klient kommuniziert problemlos mit einem Server. Dieser ist abgestürzt und nicht verfügbar, der Lastverteiler erkennt das und stellt auf einen anderen Server um. In der **Abbildung 6** passiert dieses Szenario an der roten Linie. Der Klient und seine Daten sind dennoch vorhanden. Der Klient merkt von der Migration nichts und führt im Anschluss eine Operation aus, dessen Ergebnis in die Sequenz passt.

### 5.2.2 Serviceorientierte Skalierung

Die monolithische Applikation inklusive Lastverteiler sollte den e-Shop in eine gute Form bringen, dennoch soll die Applikation weiter skalieren. Das Problem könnte nun sein, dass das Bestellservice und Warenkorbservice im Verhältnis zu den anderen Services übermäßige Auslastungen ausweisen und die komplette Applikation nur aus diesem Grund skalieren muss. Zusätzlich sollen diverse Daten zu Bestellungen in ein separates Datenbanksystem migriert werden.

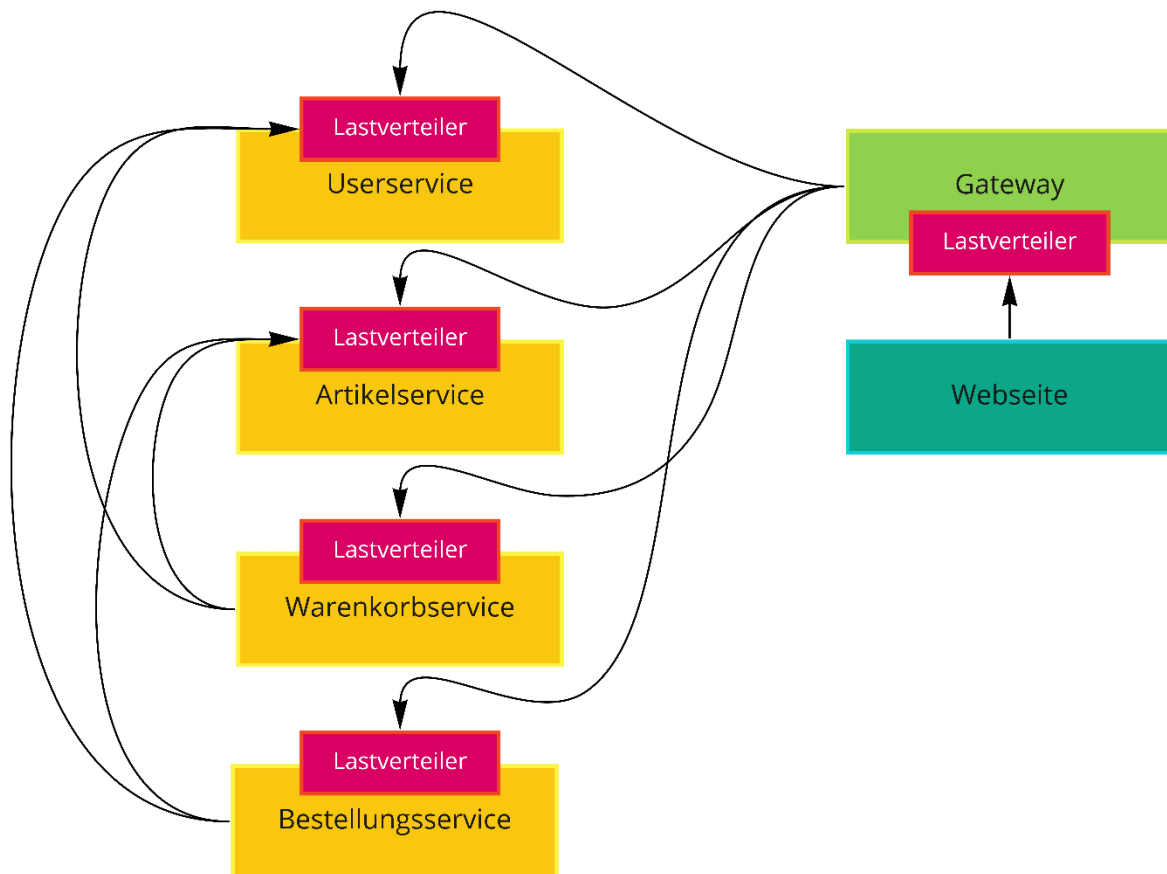
Annahme: Das neue Datenbanksystem bietet ein Dashboard für statistische Auswertungen.



**Abbildung 7:** Primitive Architektur des Shops

Der erste Ansatz den Monolithen aufzuteilen wäre, jedes Service als eigene Applikation bereitzustellen und die Applikationen mittels Verlinkung, beispielsweise REST über HTTP, verbinden. Eine solche Applikationsgruppe würde produktiv zwar funktionieren, könnte aber nicht horizontal skalieren. Problematisch ist primär die statische Bindung der Services aneinander. Es wäre dennoch möglich vertikal zu skalieren, beispielsweise könnte für das Bestellservice eine stärkere Maschine benutzt werden.

Diese Hürde könnte gelöst werden, indem sich jedes Service hinter einen Lastverteiler versteckt. Dadurch müssen die Services nur die Adresse des jeweiligen Lastverteiler Balancers kennen. Die Skalierung der Services geschieht nun transparent.



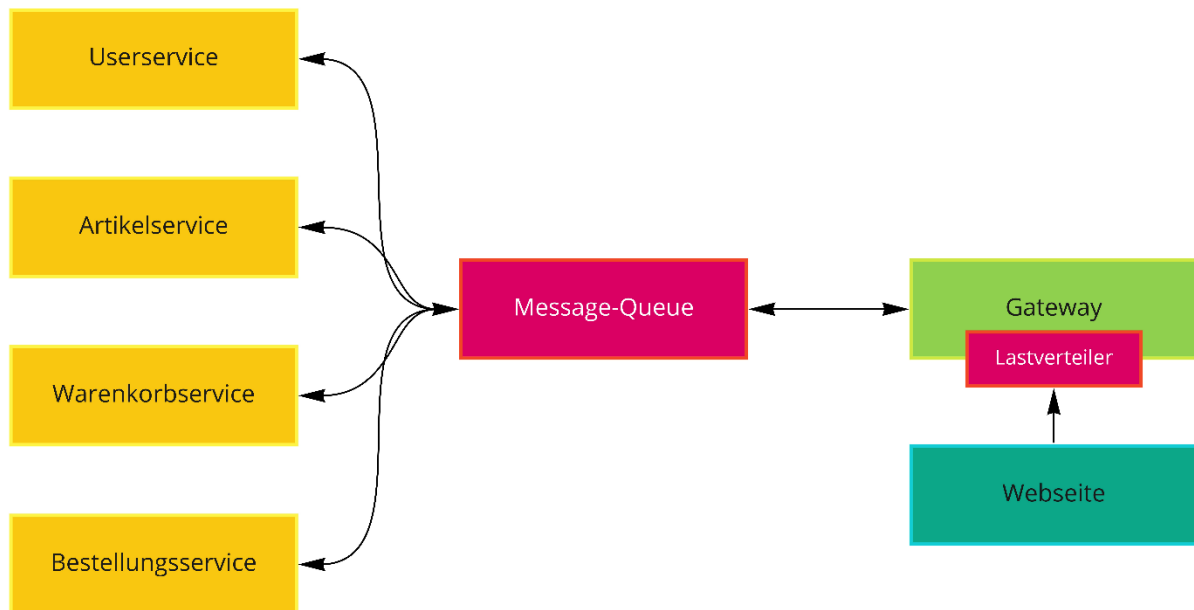
**Abbildung 8:** Architektur des Load-Balancing

Diese Architektur kommt produktiv sehr häufig vor, hat aber trotzdem einen wesentlichen Nachteil. Die Anbindung ist nach wie vor statisch. Beispielsweise verweist das User-Service auf ein User-Session-Service, welches die jeweiligen Sessions verwaltet (z. B. Kryptografie und Logging). Möchte ein System diese Logs einsehen, so müsste dieses durch das Gateway über das Userservice in das User-Session-Service. Dieser Weg impliziert jedoch, dass ein Service zu viel beansprucht wurden.

Zusätzlich können die vielen Lastverteiler, auf manche Plattformen, selbst zu einem Kostenfaktor werden, denn es wird parallel zu jeder Servicegruppe ein eigener Lastverteiler bereitgestellt. Dieses Konzept benötigt für den e-Shop mindestens elf Services inklusive Datenbank, um zu funktionieren, was für eine relativ weniger besuchte Plattform ein ziemlicher Overkill ist.

Eine Lösung wäre dazu die statische Bindung durch eine Message-Queue zu ersetzen. Zusätzlich zur Entkopplung würde diese Lösung die Möglichkeit anbieten, dass die Empfänger den Durchsatz nach Kapazitäten definieren. Wird die Queue immer länger, so kann automatisiert skaliert werden, ohne dass davor Anfragen zu lange verweilen und ohne das gar die Latenzen aller Services verfolgt werden.





**Abbildung 9:** Architektur mit einer Message-Queue

Diese Architektur ist wesentlich einfacher und praktischer als die in **Abbildung 8**. Man beachte jedoch, wie das Konzept, wie Services miteinander kommunizieren, sich fundamental verändert. Ein Service wartet nicht mehr, bis eine Anfrage kommt, sondern fragt die Queue aktiv nach Aufgaben. Dadurch kann ein Service wesentlich einfacher Herauf- und Herunterskalieren. Wenn Elemente in der Queue länger als durchschnittlich oder auch länger als erwünscht, so werden neue Instanzen hochgefahren. Verarbeitet eine Instanz innerhalb einer gewissen Zeitspanne keine Elemente, wird diese wieder abgebaut. Instanzen müssen sich nun auch nicht bei einem Lastverteiler melden und stets einen Status übermitteln, sondern Verbinden sich selbständig mit der Queue und nehmen sich passende Elemente heraus, die dann verarbeitet werden. [31]

Die Message-Queue ist zusätzlich von der Cloud verwaltet, d.h. die Queue-Instanzen werden von der Cloud reguliert und skaliert. Für die Anwendung bedeutet das, die Queue ist eine Ressource, die immer verfügbar ist. [27, 32]

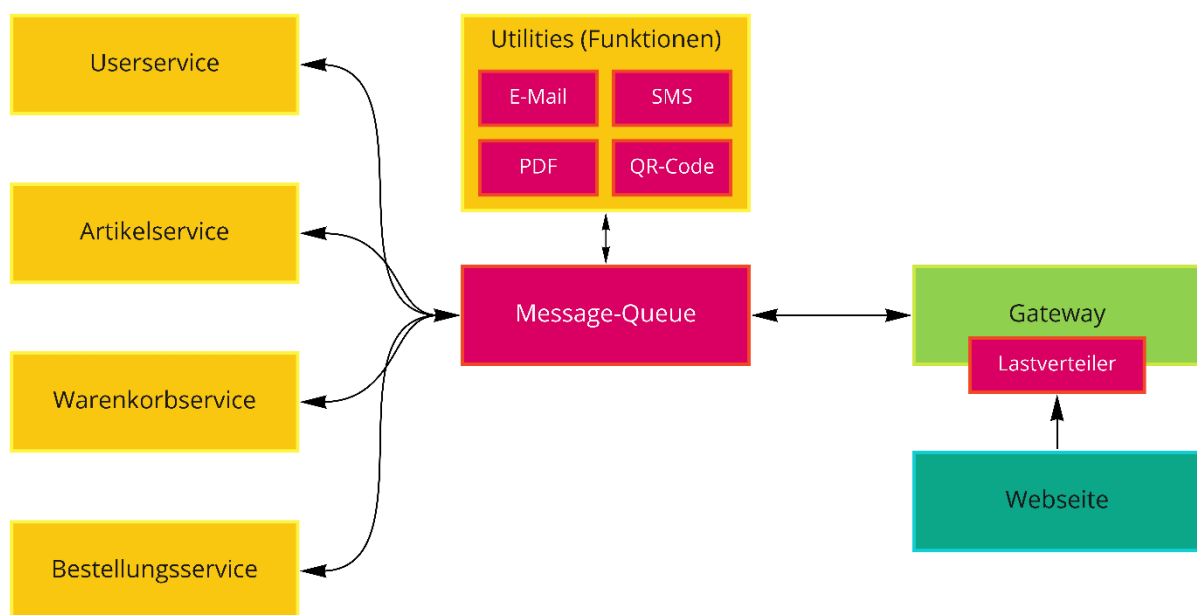
### 5.2.3 Funktionsorientierte Skalierung

Im vorigen Abschnitt wurde die Skalierung der jeweiligen Services unter die Lupe genommen.

Annahme: Das Bestellservice generiert beim Eingehen einer Bestellung eine Rechnung und verschickt diese via E-Mail. (Siehe Funktionen)

Das Generieren einer PDF ist im Vergleich zu den anderen Operationen sehr rechenintensiv und kann im Optimalfall ausgelagert werden. Da dies für ein ganzes Service nicht ausreicht, wird schlichtweg die Funktion skaliert.

Dieses Vorhaben kann jedoch nicht in einem skalierten Service untergebracht werden, sondern wird in ein eigenes Schein-Service ausgelagert. Dieses Service wird "Utilities" genannt. Dieses neue Service beinhaltet nun skalierbare Funktionen, die generisch für alle Services sind, z. B. Versenden von E-Mails, Versenden von SMS, Generierung von PDFs, Generierung von QR- und Barcodes.



**Abbildung 10:** Architektur mit Utilities als Funktionen extrahiert

Zusätzlich können nun Kernfunktionalitäten, welche als Funktionen ausgegliedert wurde, mehr Logik beinhalten als zunächst erwünscht. Die E-Mail-Funktion kann nun zwischen zahlreichen Service-Anbieter wechseln. Dies kann beispielsweise dann sinnvoll sein, wenn ein Anbieter kostengünstiger, aber dafür länger für die Zustellung braucht. Dann werden Newsletter und Coupons verbilligt zugestellt und kritische Benachrichtigungen, wie Rechnungen oder Mahnungen zu Normalpreisen zeitnah übermittelt. Diese Logik ist für den Aufrufer nicht ersichtlich, wodurch sich die Flexibilität und das Potential des Gesamtsystems immens erhöht.

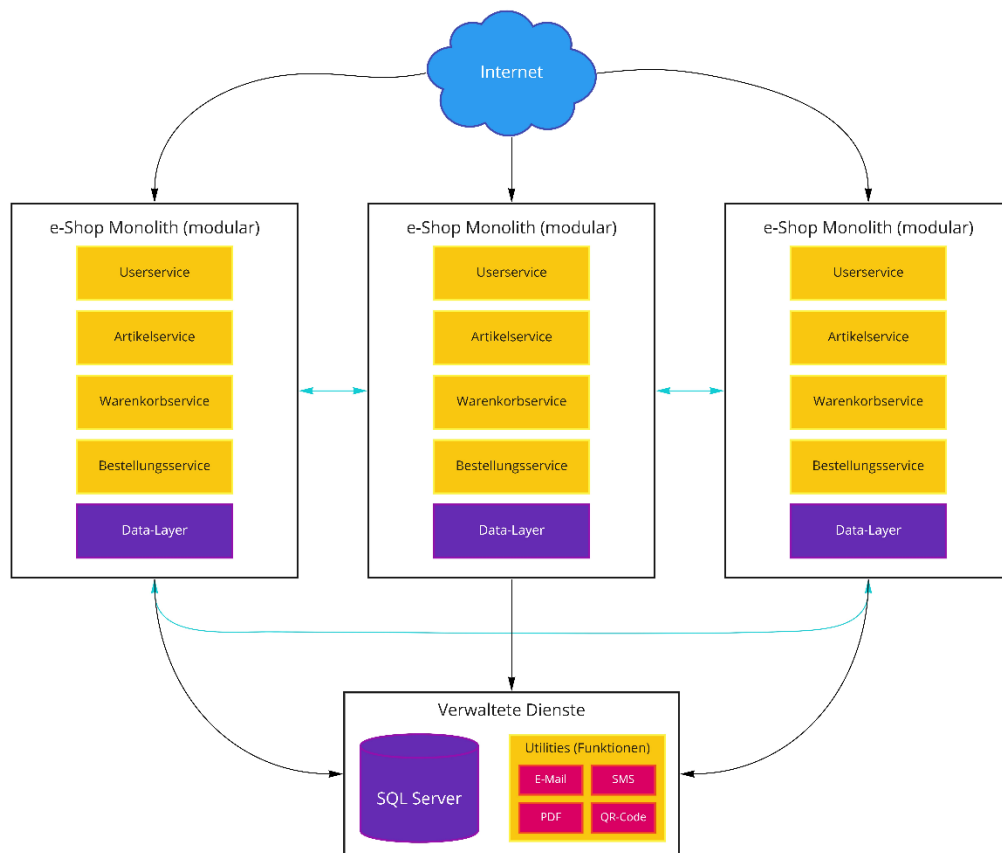
## 5.2.4 Hybride Skalierung

In letzter Zeit wurden diverse ältere Paradigmen neu entdeckt, eines dieser Paradigmen ist das Actor-Model. Eine Ableitung davon, namentlich „Virtual-Actor-Model“, wurde im Zuge des Microsoft-Projekt „Orleans“ erstellt. [33]

Dieses .NET Framework erlaubt es, verteilte Applikationen zu erstellen, ohne Lastenverteiler und Message-Queue. Zusätzlich agiert die Applikation indirekt als Cache, was wiederum die Leistung steigert. [34]

Konkret werden Akteure für sämtliche Objekte erstellt. Diese enthalten, sowohl Code, um das Objekt zu konstruieren, z. B. aus der Datenbank auslesen, und die Businesslogik. Akteure sind in der Servergruppen einmalig, das heißt, gebe es eine „Bestellung 1“, wird dieser, wenn als Akteur vertreten, nur einmalig im gesamten Cluster instanziiert. Jegliche Anfragen gegen „Bestellung 1“ gehen immer an denselben synchronen Akteur. Zusätzlich kann ein Akteur so implementiert werden, dass der referenzierte Datensatz im RAM geladen wird. Dementsprechend fungiert der Akteur auch als Cache.

Trotzdem arbeitet jede Instanz im Cluster so, als wäre der Akteur lokal vorhanden. Die tatsächlichen Aufrufe werden abstrahiert und greifen entweder tatsächlich auf ein lokales Element oder werden serialisiert und an die tatsächliche Instanz weitergeleitet. [34]



**Abbildung 11:** Modularer Monolith mit virtuellen Akteuren und verwaltete Dienste

Alle rechenintensiven Funktionen wurden extrahiert und in ein vollständig verwaltetes, sich automatisch skalierendes, Service gruppiert. Die Datenbank ist genauso vollständig verwaltet und kann dynamisch vergrößert und verkleinert werden. Die Applikationsinstanzen sind untereinander Verbunden und basieren auf virtuelle Akteure.

Zusätzlich werden Datensätze bei aktiver Nutzung im RAM gehalten, wodurch sich die Leistung extrem erhöht. Möchte man skalieren, so muss lediglich eine neue Applikationsinstanz gestartet werden (Siehe **Abbildung 2**). Aufgrund der Tatsache, dass alle rechenintensiven Funktionen extrahiert werden, kann davon ausgegangen werden, dass sämtliche übrigen Use-Cases in der Applikation gleichmäßig auslasten. Dementsprechend ist es durchaus sinnvoll den Overhead von Microservices zu ersparen und die Services so nah wie möglich aneinander zu halten.

Trotzdem gibt es Use-Cases die im Vergleich zu den anderen Use-Cases sehr viel Kapazität beanspruchen, dafür hat Orleans auch eine Lösung: „Heterogene Instanzen“, diese bieten ihre Ressourcen nur für eine definierte Gruppe an Akteure. Dementsprechend kann, obwohl die Applikation als Ganzes skaliert, nur ein oder mehrere Services skaliert werden. [35]

### 5.2.5 Skalierung der Abhängigkeiten

Oft liegt der Engpass nicht an der Applikation selbst sondern an den Abhängigkeiten. Ein sehr gängiges Beispiel dafür ist die Datenbank, auch wenn das meistens ein Resultat von nicht optimierten Abfragen und Modellen ist.

Die Skalierbarkeit ist abhängig vom Datenbanksystem selbst und von den Use-Cases. Beispielsweise unterstützt MongoDB wenn die Datenbank horizontal skaliert keine eindeutigen Felder mehr. [36] Parallel existieren auch Datenbanken die explizit dafür entwickelt wurden zu skalieren z. B. CockroachDB und YugabyteDB.

Parallel dazu werden auch sehr dynamische Serverless-Datenbanken angeboten. Die skalieren automatisch, abhängig von der Last. Verrechnet werden nur die tatsächlich genutzten Ressourcen. [4, 7, 6]

## 5.3 Anwendungsbereiche

Prinzipiell lassen sich alle Applikationen und Use-Cases skalieren. Applikationen mit speziellen Bedürfnissen z. B. Gameserver oder Transaktionen, benötigen erhöhte Aufmerksamkeit und in den meisten Fällen auch eigene Strategien (Siehe Kontinuierliche Verbindungen skalieren)

Mit Serverless Computing, kann die Applikation unabhängig von der Plattform skaliert werden. Zusätzlich können verwaltete Dienste beansprucht werden, welche vereinfacht oder gar automatisch skalieren. Das Entwicklungsteam muss sich primär nicht mit den Servern selbst befassen und kann sich auf das Produkt konzentrieren.

### 5.3.1 Grenzen

Applikationen, die sehr eng mit Transaktionen arbeiten, sind im Vergleich komplizierter zu skalieren. Das liegt aber meistens eher daran, dass die Datenbank selbst generisch transaktional ist, d.h. die Datenbank kennt die Use-Cases nicht, bietet dennoch Transaktionen an.

Eine Lösung wäre dazu Transaktionen bereits in einer höheren Ebene, in der Applikation, abzuwickeln, beispielsweise unterstützt Orleans Transaktionen. Dazu werden im Akteur die jeweiligen Funktionen als transaktional markiert. Vorausgesetzt das Datenbanksystem unterstützt Transaktionen z. B. PostgreSQL, MySQL, MSSQL aber auch Azure Table und MongoDB (Replica-Set). Trotzdem ist das nur ein Lösungsvorschlag und muss nicht für alle Probleme funktionieren. Es existieren auch spezielle Datenbanklösungen, z. B. Nuodb für Banken, die für bestimmte Operationen optimiert wurden. [37]

## 6 Umsetzung

Als kleines Beispiel von der Wirkung des Cache wurde folgendes Szenario angenommen. Zwei Services die jeweils auf die gleiche Datenbank, ein Key-Value Store, zugreifen. Die Datenbank ist grundsätzlich stark belastet und reagiert durchschnittlich nach 50ms. Dieser Verzug wurde künstlich hinzugefügt.

```
1      public async Task<UserView> Get(int id) {
2          await Task.Delay(50);
3          var value = await _database.StringGetAsync(id.ToString());
4          if (!value.HasValue || value.IsNull) {
5              return null;
6          }
7
8          return JsonConvert.DeserializeObject<UserView>(value);
9      }
10
11     public async Task<bool> Set(int id, UserView data) {
12         await Task.Delay(50);
13         return await _database.StringSetAsync(id.ToString(), JsonConvert.SerializeObject(data));
14     }
```

**Abbildung 12:** Source-Code vom Zugriff auf die Datenbank

Wie der **Abbildung 12** entnommen werden kann, wird in Zeile 2 und Zeile 12 künstlich ein Verzug von 50ms hinzugefügt. Dieses Datenbankmodul wird einem Userservice untergeordnet welches, ohne jeglicher Logik, die Daten abruft.

```
1      public Task<UserView> GetUserById(int id) {
2          return _repository.Get(id);
3      }
4
5     public Task<bool> UpdateUserById(int id, UserView user) {
6         return _repository.Set(id, user);
7     }
```

**Abbildung 13:** Source-Code vom Userservice

Die erste Applikation greift direkt auf das erstellte Service zu und gibt den Datensatz stets in der aktuellsten Form von der Datenbank bereit.

```
1      [HttpGet("/users/{id:int}")]
2      public async Task<IActionResult> Get([FromRoute] int id, [FromServices] IUserService userService) {
3          var result = await userService.GetUserById(id);
4          if (result == null) {
5              return NotFound();
6          }
7
8          return Ok(result);
9      }
10
11     [HttpPatch("/users/{id:int}")]
12     public async Task<IActionResult> Update([FromRoute] int id, [FromBody] UserView user,
13                                             [FromServices] IUserService userService) {
14         await userService.UpdateUserById(id, user);
15         return Ok();
16     }
```

**Abbildung 14:** Source-Code von der REST-Schnittstelle, welche direkt auf das Userservice basiert

Die zweite Applikation hingegen hat Orleans eingebunden und definiert dadurch eine weitere Schicht, welche das Userservice grundsätzlich ersetzt.

```
1      public interface IUserGrain : IGrainWithIntegerKey {
2
3          [ReadOnly, AlwaysInterleave]
4          Task<UserView> GetUser();
5
6          Task UpdateUser(UserView user);
7
8      }
9
10     ...
11
12     public async Task<UserView> GetUser() {
13         if (!_loaded) {
14             _loaded = true;
15             _userView = await _repository.GetUserById((int)this.GetPrimaryKeyLong());
16         }
17
18         return _userView;
19     }
20
21     public async Task UpdateUser(UserView user) {
22         if (await _repository.UpdateUserById((int)this.GetPrimaryKeyLong(), user)) {
23             _userView = user;
24         }
25     }
```

**Abbildung 15:** Source-Code von der „Grain“-Definition und Ausschnitt aus der Implementierung

In diesem Fall wurde direkt auf das Repository zugegriffen. Prinzipiell setzt das „Grain“ zwischen Datenbank und Service an. Trotzdem beinhaltet das „Grain“ grundlegende Funktionen und Mechanismen, die eventuell in Bezug mit der Datenbank oder mit anderen „Grains“ stehen. Die eigentliche Anwendungslogik befindet sich dennoch im Service.

Einfachheitshalber wurde auf das Service verzichtet, da keine Applikationslogik vorhanden ist. Die REST-Schnittstelle sieht dementsprechend wie folgt aus:

```
1      [HttpGet("/orleans_users/{id:int}")]
2      public async Task<IActionResult> GetFromOrleans([FromRoute] int id,
3
4          [FromServices] IClusterClient clusterClient) {
5          var result = await clusterClient.GetGrain<IUserGrain>(id).GetUser();
6          if (result == null) {
7              return NotFound();
8          }
9
10         return Ok(result);
11     }
12
13     [HttpPatch("/orleans_users/{id:int}")]
14     public async Task<IActionResult> UpdateInOrleans([FromRoute] int id, [FromBody] UserView user,
15
16         [FromServices] IClusterClient clusterClient) {
17         await clusterClient.GetGrain<IUserGrain>(id).UpdateUser(user);
18         return Ok();
19     }
```

**Abbildung 16:** Source-Code von der REST-Schnittstelle, welche direkt auf das „UserGrain“ basiert



Grundsätzlich sieht der Source-Code beider Implementierungen gleich aus. Abgesehen davon, dass das Service als „Grain“ definiert wurde und der Zugriff über einen Client erfolgt, hat sich nichts verändert. Fundamental unterscheidet sich das „Grain“ vom „Service“ in der Art des Zugriffes, denn das Service bekommt bei jedem Abruf die „Id“ des Zielobjekts mit, während das „Grain“ diese bei der Initialisierung im Konstruktor bekommt.

In weiterer Folge muss sich das Entwicklungsteam um den State keine Gedanken machen, denn im „Grain“ können langlebige Felder angelegt werden, welche nicht persistiert werden.

```
1      public class UserGrain : Grain, IUserGrain {
2
3          #region FIELDS
4          private bool _loaded;
5          private UIView _userView;
6          #endregion
7
8          ...
9
10     }
```

**Abbildung 17:** Source-Code von den langlebigen Felder des „UserGrains“

In der **Abbildung 17** wird der eigentliche Datenbankeintrag abgelegt. Dementsprechend liegt der Wert direkt im Arbeitsspeicher der verarbeitenden Maschine und muss im Optimalfall nicht erneut von der Datenbank geladen werden. Zusätzlich werden diese „Grains“ einmalig und verteilt im Cluster angelegt und referenziert (Siehe Orleans in Hybride Skalierung). Dementsprechend kann gewährleistet werden, dass sämtliche Anfragen von derselben Instanz synchron verarbeitet werden.

In weiterer Folge wurden beide Applikationen gebündelt auf zwei virtuelle Maschinen bereitgestellt. Jede virtuelle Maschine besitzt 2 virtuelle Kerne und 4GB Arbeitsspeicher. Zusätzlich sind die Server mit einer 5Gbit Leitung direkt verbunden. Ein Lastverteiler wurde beiden Servern vorgestellt und verteilt die Last proportional.

Außerdem wurden zwei virtuelle Maschinen erstellt, mit jeweils 2 virtuellen Kernen und 4GB Arbeitsspeicher, die als Client die Server zeitgleich für eine Minute mit Last getestet haben. Die Ressourcen wurden fast alle in Nürnberg, Deutschland, beim Anbieter „Hetzner“ bereitgestellt. Einer dieser Clients steht jedoch in Helsinki, Finnland, um den Faktor Netzwerkanbindung hervorheben zu können.

The image shows two terminal windows side-by-side, both running Siege 4.0.4 tests. The left window is titled 'root@ubuntu-2gb-hell-1:~#' and the right is 'root@ubuntu-2gb-nbg1-1:~#'. Both tests target 'http://142.132.240.229/users/6 -t 1M'. The left test results are: Transactions: 108610 hits, Availability: 100.00 %, Elapsed time: 59.99 secs, Data transferred: 3.94 MB, Response time: 0.14 secs, Transaction rate: 1810.47 trans/sec, Throughput: 0.07 MB/sec, Concurrency: 252.83, Successful transactions: 108610, Failed transactions: 0, Longest transaction: 1.14, Shortest transaction: 0.10. The right test results are: Transactions: 219726 hits, Availability: 100.00 %, Elapsed time: 59.07 secs, Data transferred: 7.96 MB, Response time: 0.07 secs, Transaction rate: 3719.76 trans/sec, Throughput: 0.13 MB/sec, Concurrency: 254.54, Successful transactions: 219726, Failed transactions: 0, Longest transaction: 0.27, Shortest transaction: 0.05.

**Abbildung 18:** Zwei Clients mit dem Testergebnissen nach jeweils zwei Tests mit jeweils 255 virtuellen Nutzern

Wie der **Abbildung 18** entnommen werden kann, wurden jeweils zwei Tests an den Lastverteiler gerichtet. Dieser hat die Last gleichmäßig auf beide Server verteilt. Getestet wurde die REST-Schnittstelle des Userservice. Beide Clients kommen zusammen auf ca. 5.500 Anfragen pro Sekunde.

The image shows two terminal windows side-by-side, both running Siege 4.0.4 tests. The left window is titled 'root@ubuntu-2gb-hell-1:~#' and the right is 'root@ubuntu-2gb-nbg1-1:~#'. Both tests target 'http://142.132.240.229/orleans\_users/6 -t 1M'. The left test results are: Transactions: 138831 hits, Availability: 100.00 %, Elapsed time: 59.60 secs, Data transferred: 5.03 MB, Response time: 0.11 secs, Transaction rate: 2329.38 trans/sec, Throughput: 0.08 MB/sec, Concurrency: 251.53, Successful transactions: 138831, Failed transactions: 0, Longest transaction: 1.18, Shortest transaction: 0.04. The right test results are: Transactions: 362137 hits, Availability: 100.00 %, Elapsed time: 59.55 secs, Data transferred: 13.12 MB, Response time: 0.04 secs, Transaction rate: 6081.23 trans/sec, Throughput: 0.22 MB/sec, Concurrency: 254.50, Successful transactions: 362137, Failed transactions: 0, Longest transaction: 0.12, Shortest transaction: 0.00.

**Abbildung 19:** Zwei Client mit den Testergebnissen nach jeweils zwei Tests mit jeweils 255 virtuellen Nutzern

In der **Abbildung 19** wurde der Test von **Abbildung 18** wiederholt, mit dem Unterschied, dass diesmal die REST-Schnittstelle zum „UserGrain“ bzw. die Orleans-Implementierung getestet wurde. Obwohl die Implementierung sich nicht maßgeblich unterscheidet, kommen beide Clients, in diesem Test, auf ca. 8.000 Anfragen pro Sekunde, eine Steigerung von ca. 45%!

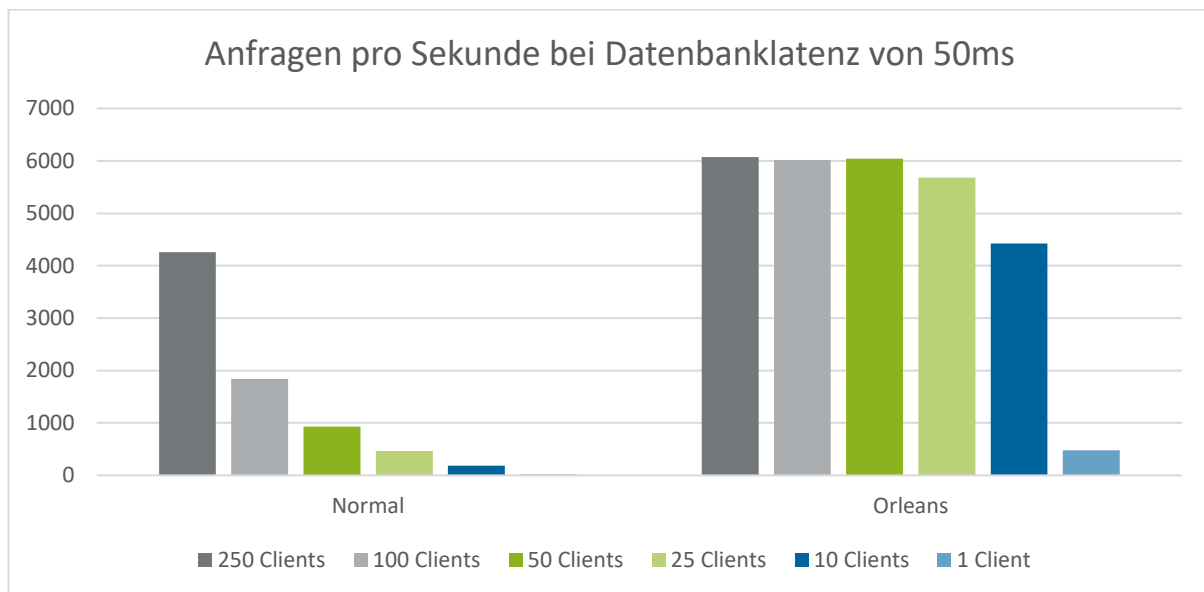
```
root@ubuntu-2gb-nbgl-1: ~
** SIEGE 4.0.4
** Preparing 100 concurrent users for battle.
The server is now under siege...
Lifting the server siege...
Transactions:      355662 hits
Availability:      100.00 %
Elapsed time:      59.08 secs
Data transferred:  12.89 MB
Response time:     0.02 secs
Transaction rate:  6020.01 trans/sec
Throughput:        0.22 MB/sec
Concurrency:       99.69
Successful transactions: 355662
Failed transactions: 0
Longest transaction: 0.07
Shortest transaction: 0.00

root@ubuntu-2gb-nbgl-1:~# siege -c 250 -t 1M -f urls.txt
** SIEGE 4.0.4
** Preparing 250 concurrent users for battle.
The server is now under siege...
Lifting the server siege...
Transactions:      363644 hits
Availability:      100.00 %
Elapsed time:      59.86 secs
Data transferred:  13.18 MB
Response time:     0.04 secs
Transaction rate:  6074.91 trans/sec
Throughput:        0.22 MB/sec
Concurrency:       249.52
Successful transactions: 363644
Failed transactions: 0
Longest transaction: 0.12
Shortest transaction: 0.00

root@ubuntu-2gb-nbgl-1:~#
```

Abbildung 20: Tests mit einer Maschine als Client

In weiterer Folge wurden die Tests wiederholt. Diesmal wurde jeweils mit einem, zehn, 25, 50, 100 und 250 parallelen virtuellen Besuchern getestet. Getestet wurden dann jedes der zwei Konzepte. Die Ergebnisse wurden letztendlich visualisiert.



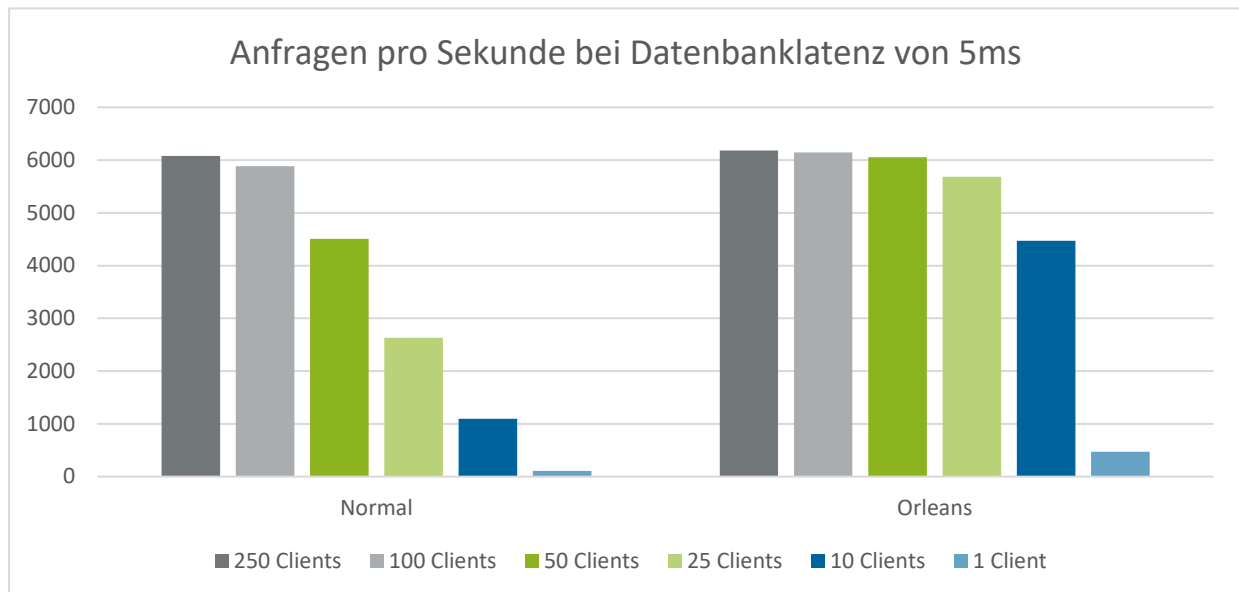
**Diagramm 1:** Ergebnisse der Tests bei einer künstlichen Datenbanklatenz von 50ms

Die Grafik zeigt deutlich, wie Orleans, aufgrund vom Caching schon bei einer geringen Anzahl an Parallelbesuchern das Maximum aus dem Server rausholen kann. Während die normale Implementierung mit einem Nutzer lediglich ca. 18 Anfragen pro Sekunden verarbeiten konnte, wurden von Orleans ca. 475 Anfragen verarbeitet.

Bemerkenswert ist außerdem, dass die normale Implementierung durchschnittlich ca. 55ms für eine Anfrage benötigt hat, was sich aus 50ms Datenbanklatenz und 5ms allgemeine Latenz zusammenstellt. Dieses Schema kann auch bei zehn Parallelbesuchern erkannt werden. Hier werden in der normalen Implementierung ca. 185 Anfragen pro Sekunde verarbeitet, wodurch sich pro Benutzer wieder ca. 18 Anfragen pro Sekunde ergeben. Wenn Orleans bei ca. 6.000 Anfragen pro Sekunde, unabhängig von der Nutzeranzahl erreicht, dann bräuchte die normale Implementierung hochgerechnet ca. 333 parallele Besucher. In der **Abbildung 18** wurden dieser Test parallel von zwei Maschinen mit je 255 virtuellen Nutzern ausgeführt, in Summe 510 virtuelle Nutzer. Jedoch konnten nur ca. 5.000 Anfragen pro Sekunde verarbeitet werden.

Die Grafik zeigt hervorragend, wie effektiv das Caching von Datensätzen ist. Orleans ist im Grunde nicht von der Latenz der Datenbank betroffen. Sämtliche Operationen werden gespiegelt, in der Applikation und in der Datenbank, ausgeführt. Orleans bietet dabei die Option die aktuelle Version zu behalten und sie bei zukünftigen Anfragen wiederzuverwenden. Dabei kann Orleans im Median jede Anfrage ca. 50ms schneller beantworten und leichter skalieren.

In weiterer Folge wurde der Test mit einer verringerten Zugriffszeit von 5ms auf die Datenbank auf 5ms wiederholt.



**Diagramm 2:** Ergebnisse der Tests bei einer künstlichen Datenbanklatenz von 5ms

Im Einklang zu **Diagramm 1** steigt die Leistungsfähigkeit der normalen Implementierung bis zur von Orleans erreichte Grenze von ca. 6.100 Anfragen pro Sekunde bei 250 virtuellen Besuchern weiterhin linear an. Leistungstechnisch hat sich bei Orleans, bei Vergleich zum Vortest, nichts verändert. Die idente Leistung unterstützt die Behauptung, dass Orleans grundsätzlich unabhängig von der Datenbanklatenz bzw. allgemein der Latenzen von Abhängigkeiten arbeiten kann. Natürlich nur dann, wenn der Cache tatsächlich genutzt wird, was bei einer hauptsächlich lesenden Applikation eher der Fall ist.

Allgemein kann Orleans im Vergleich zu einer konventionellen Applikation wesentlich schneller und entsprechend effizienter verarbeiten. Das liegt, einerseits am Cache, wodurch die Datenbanklatenz ausgehebelt wird, andererseits an der verteilten Architektur wodurch Objekte direkt mit dem jeweiligen Datensatz verknüpft im Cluster verteilt angelegt werden.

## 7 Conclusio

Konzepte wie Monolithen, Microservices oder auch Funktionen sind Gang und gebe und in jeder Applikation vertreten. Dennoch setzen die meisten Architekt\*innen immer nur auf einer dieser Technologien, statt diese zu kombinieren.

Mittels Cloud ist es nun wesentlich einfacher verwaltete Dienste zu benutzen, die vereinfacht oder auch automatisch skalieren. Dadurch kann ein flexibles und resilientes Gesamtsystem gestaltet werden.

Beispielsweise können, wie in Hybride Skalierung beschrieben wurde, verteilte Applikationen aus Basis eines Monolithen entwickelt werden und ressourcenintensivere Operationen als Funktionen extrahiert werden. Dadurch kann die Applikation wesentlich einfacher, kostengünstiger und performanter betrieben werden. (Siehe Umsetzung) Muss die Applikation stark skalieren oder wollen unterschiedliche Technologien eingesetzt werden, dann sind auch Microservices sinnvoll.

Parallel dazu können verwaltete Komponenten, z. B. Datenbank und Protokollierung hinzugezogen werden und unabhängig betrieben werden.

Im Optimalfall wird die Applikation als modularer Monolith implementiert. Bemerkenswert ist, dass sich der Monolith zum modularen Monolith nur in der sauberen Trennung der Services bzw. Subsysteme unterscheidet. Wird stets strikt getrennt, dann ist die Rede immer von einem modularen Monolith. Der modulare Monolith erlaubt auch die unkomplizierte Trennung der Services, falls Microservices erwünscht sind.

In weiterer Folge werden sämtliche rechenintensive Funktionen extrahiert und separat betrieben. Sollte die Applikation dennoch ihre Grenzen erreichen, so können die unterschiedlichen Modulare paarweise bzw. falls notwendig eigenständig, als Microservices, betrieben werden.

Empfehlenswert ist immer ein Cache, um sich wiederholende Anfragen schneller und ressourcenschonend zu verarbeiten. Sollte die Applikation in C# entwickelt werden, so wäre Orleans eine gute Alternative zum dedizierten Cache.

Sollte Orleans nicht eingesetzt werden, so empfiehlt es sich, eine Message-Queue zu benutzen, um die Kommunikation zwischen Lastverteiler und Applikationen bzw. Services zu standardisieren und dementsprechend vereinfachen. Zusätzlich kann die Message-Queue ein Überlasten der jeweiligen Instanzen verhindern und bietet zusätzlich essenzielle Metriken, um effizient skalieren zu können.

Auch wenn Framework wie z. B. Orleans die Entwicklung von verteilten Applikationen vereinfachen, muss die Leistungsfähigkeit des Systems stets verfolgt werden. Denn selbst diese haben ihre Grenzen, beispielsweise ist ein Aufruf einer lokalen Methode, welche lediglich einen fixen Wert liefert, im Vergleich zu einer identen Akteur-Methode, bei lokaler Ausführung, ca. 1.250x schneller.

```
1      public interface ITestGrain : IGrainWithIntegerKey {
2          [ReadOnly, AlwaysInterleave]
3          Task<int> Read();
4      }
5
6      public class TestGrain : Grain, ITestGrain { // Orleans
7          public Task<int> Read() { // ca. 191.000 op/s
8              return Task.FromResult(1);
9          }
10     }
11
12     public class TestGrainWithoutAny { // Kein Orleans
13         public Task<int> Read() { // ca. 240.000.000 op/s
14             return Task.FromResult(1);
15         }
16     }
```

**Abbildung 21:** Vergleich Akteur-Methode und lokale Methode

Dementsprechend muss, beim Planen einer Architektur, der Nutzen im Verhältnis zum Mehraufwand verglichen werden und parallel die operativen Kosten berücksichtigt werden. Zusätzlich muss stets berücksichtigt werden, dass Serverless-Dienste einen sehr kostengünstigen Einstieg ermöglichen, aber aufgrund der nutzungsbasierten Abrechnungen schnell teurer werden können. In diesen Fällen sollte auch evaluiert werden, ob das eigenständige Bereitstellen dieses Dienstes möglich und sich rentieren würde.

# Literaturverzeichnis

- [1] LogicMonitor, „What Are Microservices and Why Use Them?“, [Online]. Available: <https://www.logicmonitor.com/blog/what-are-microservices>. [Zugriff am 01 05 2022].
- [2] Cloudflare, „Was ist Serverless-Computing? | Was bedeutet Serverless“, [Online]. Available: <https://www.cloudflare.com/de-de/learning/serverless/what-is-serverless/>. [Zugriff am 12 12 2021].
- [3] MongoDB, „What is Serverless Architecture?“, [Online]. Available: <https://www.mongodb.com/serverless>. [Zugriff am 01 05 2022].
- [4] Microsoft, „Azure SQL Database serverless“, [Online]. Available: <https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview?view=azuresql>. [Zugriff am 04 05 2022].
- [5] Amazon AWS, „Amazon Aurora Serverless“, [Online]. Available: <https://aws.amazon.com/de/rds/aurora/serverless/>. [Zugriff am 04 05 2022].
- [6] Cockroach Labs, „CockroachDB Pricing“, [Online]. Available: <https://www.cockroachlabs.com/get-started-cockroachdb/>. [Zugriff am 04 05 2022].
- [7] MongoDB, „Serverless Database“, [Online]. Available: <https://www.mongodb.com/cloud/atlas/serverless>. [Zugriff am 4 5 2022].
- [8] C. Richardson, „Pattern: Monolithic Architecture“, [Online]. Available: <https://microservices.io/patterns/monolithic.html>. [Zugriff am 4 5 2022].
- [9] S. Watts, „Technical Debt: The Ultimate Guide“, [Online]. Available: <https://www.bmc.com/blogs/technical-debt-explained-the-complete-guide-to-understanding-and-dealing-with-technical-debt/>. [Zugriff am 04 05 2022].
- [10] JRebel, „What is a Modular Monolith“, [Online]. Available: <https://www.jrebel.com/blog/what-is-a-modular-monolith>. [Zugriff am 04 05 2022].
- [11] P. Gupta, „Understanding the modular monolith and its ideal use cases“, [Online]. Available: <https://www.techtarget.com/searchapparchitecture/tip/Understanding-the-modular-monolith-and-its-ideal-use-cases>. [Zugriff am 04 05 2022].
- [12] F. Hakamine, „Was sind Mikroservices?“, [Online]. Available: <https://www.okta.com/de/blog/2021/02/microservices/>. [Zugriff am 12 12 2021].



- [13] J. L. u. M. Fowler, „Microservices: a definition of this new architectural term,“ [Online]. Available: <https://martinfowler.com/articles/microservices.html#Microservices>. [Zugriff am 12 12 2021].
- [14] JRebel, „Common Performance Problems With Microservices,“ [Online]. Available: <https://www.jrebel.com/blog/performance-problems-with-microservices>. [Zugriff am 04 05 2022].
- [15] M. Roberts, „Serverless Architectures,“ [Online]. Available: <https://martinfowler.com/articles/serverless.html>. [Zugriff am 12 12 2021].
- [16] Microsoft, „Azure Functions,“ [Online]. Available: <https://azure.microsoft.com/de-de/services/functions>. [Zugriff am 12 12 2021].
- [17] A. und Ć. PRZEMYSŁAW, „Microservices instead of the monolith. Why you should move away from a monolithic application (and how to do it),“ [Online]. Available: <https://kissdigital.com/blog/microservices-instead-of-the-monolithic-application>. [Zugriff am 04 05 2022].
- [18] Microsoft, „Azure App Service,“ [Online]. Available: <https://azure.microsoft.com/de-de/services/app-service/#overview>. [Zugriff am 05 05 2022].
- [19] Google, „Cloud Functions,“ [Online]. Available: <https://cloud.google.com/functions>. [Zugriff am 12 12 2021].
- [20] Azure, „Lambda,“ [Online]. Available: <https://aws.amazon.com/de/lambda/>. [Zugriff am 12 12 2021].
- [21] Amazon, „Amazon EC2,“ [Online]. Available: <https://aws.amazon.com/de/ec2/?p=ft&c=wa&z=2>. [Zugriff am 05 05 2022].
- [22] Google, „Cloud App Engine,“ [Online]. Available: <https://cloud.google.com/appengine?hl=de>. [Zugriff am 05 05 2022].
- [23] Amazon, „Amazon ElastiCache,“ [Online]. Available: <https://aws.amazon.com/de/elasticache>. [Zugriff am 05 05 2022].
- [24] Amazon, „Amazon Simple Queue Service,“ [Online]. Available: <https://aws.amazon.com/de/sqs/>. [Zugriff am 05 05 2022].
- [25] Google, „Memorystore,“ [Online]. Available: <https://cloud.google.com/memorystore?hl=de>. [Zugriff am 05 05 2022].

- [26] Microsoft, „Azure Cache for Redis,“ [Online]. Available: <https://azure.microsoft.com/de-de/services/cache/#overview>. [Zugriff am 05 05 2022].
- [27] Microsoft, „Queue Storage,“ [Online]. Available: <https://azure.microsoft.com/de-de/services/storage/queues/#overview>. [Zugriff am 05 05 2022].
- [28] Microsoft, „Azure Kubernetes Service (AKS),“ [Online]. Available: <https://azure.microsoft.com/en-us/services/kubernetes-service/#overview>. [Zugriff am 05 05 2022].
- [29] Cloud Native, „High-quality, ubiquitous, and portable telemetry to enable effective observability,“ [Online]. Available: <https://opentelemetry.io/>. [Zugriff am 05 05 2022].
- [30] Microsoft, „Azure SignalR Service,“ [Online]. Available: <https://azure.microsoft.com/de-de/services/signalr-service/#overview>. [Zugriff am 05 05 2022].
- [31] M. Osminer, „WHAT IS THE BEST MESSAGE QUEUE FOR YOUR APPLICATION?,“ [Online]. Available: <https://www.cardinalpeak.com/blog/what-is-the-best-message-queue-for-your-application>. [Zugriff am 11 05 2022].
- [32] Microsoft, „Service Bus,“ [Online]. Available: <https://azure.microsoft.com/en-us/services/service-bus/#features>. [Zugriff am 11 05 2022].
- [33] IEvangelist und bradygaster, „Microsoft Orleans,“ [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/orleans/overview>. [Zugriff am 12 05 2022].
- [34] IEvangelist, „Location transparency,“ [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/orleans/benefits#location-transparency>. [Zugriff am 12 05 2022].
- [35] IEvangelist, „Heterogeneous silos overview,“ [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/orleans/host/heterogeneous-silos>. [Zugriff am 12 05 2022].
- [36] MongoDB, „Unique Indexes in Sharded Collections,“ [Online]. Available: <https://www.mongodb.com/docs/manual/core/sharded-cluster-requirements/#unique-indexes-in-sharded-collections>. [Zugriff am 12 05 2022].
- [37] Nuodb, „Next-Gen Banking with Nuodb,“ [Online]. Available: <https://nuodb.com/banking>. [Zugriff am 12 05 2022].
- [38] D. J. Jung, „Serverless Computing hat sich im letzten Jahr verdreifacht,“ [Online]. Available: <https://www.zdnet.de/88395040/serverless-computing-hat-sich-im-letzten-jahr->

verdreifacht/. [Zugriff am 12 12 2021].

# Abbildungsverzeichnis

<b>Abbildung 1:</b> Architektur des Monolithen inklusive Abhängigkeiten (Gelb: öffentliche Services, Lila: Datenbankverknüpfung, Grün: private Services) .....	15
<b>Abbildung 2:</b> Auszug aus der automatischen Skalierung in Azure Web Apps .....	16
<b>Abbildung 3:</b> Monolith mit eigenem Lastverteiler.....	17
<b>Abbildung 4:</b> Grobe Architektur wie eine Verbindung immer an einen bestimmten Server weitergeleitet wird.....	18
<b>Abbildung 5:</b> Ausführung vom Proof of Concept: Ein Lastverteiler, ein Klient und zwei Server .....	20
<b>Abbildung 6:</b> Der aktive Server ist abgestürzt, die Applikation kommuniziert nun mit einem anderen Server, ohne dass die Verbindung abbricht .....	22
<b>Abbildung 7:</b> Primitive Architektur des Shops .....	23
<b>Abbildung 8:</b> Architektur des Load-Balancing .....	24
<b>Abbildung 9:</b> Architektur mit einer Message-Queue.....	25
<b>Abbildung 10:</b> Architektur mit Utilities als Funktionen extrahiert.....	26
<b>Abbildung 11:</b> Modularer Monolith mit virtuellen Akteuren und verwaltete Dienste .....	28
<b>Abbildung 12:</b> Source-Code vom Zugriff auf die Datenbank .....	30
<b>Abbildung 13:</b> Source-Code vom „UserService“ .....	30
<b>Abbildung 14:</b> Source-Code von der REST-Schnittstelle, welche direkt auf das „UserService“ basiert.....	31
<b>Abbildung 15:</b> Source-Code von der „Grain“-Definition und Ausschnitt aus der Implementierung.....	32
<b>Abbildung 16:</b> Source-Code von der REST-Schnittstelle, welche direkt auf das „UserGrain“ basiert .....	32
<b>Abbildung 17:</b> Source-Code von den langlebigen Felder des „UserGrains“ .....	33
<b>Abbildung 18:</b> Zwei Clients mit dem Testergebnissen nach jeweils zwei Tests mit jeweils 255 virtuellen Nutzern .....	34
<b>Abbildung 19:</b> Zwei Client mit den Testergebnissen nach jeweils zwei Tests mit jeweils 255 virtuellen Nutzern .....	34
<b>Abbildung 20:</b> Tests mit einer Maschine als Client.....	35
<b>Abbildung 21:</b> Vergleich Akteur-Methode und lokale Methode.....	39

## Tabellenverzeichnis

<b>Tabelle 1:</b> Liste mit den Regionen und dem entsprechenden Server .....	18
<b>Tabelle 2:</b> Eintrag unmittelbar nach Verbindungsaufbau .....	19
<b>Tabelle 3:</b> Eintrag nach der Authentifizierung .....	19
<b>Tabelle 4:</b> Eintrag, wenn die tatsächliche Kommunikation begonnen hat .....	19

## Diagrammverzeichnis

**Diagramm 1:** Ergebnisse der Tests bei einer künstlichen Datenbanklatenz von 50ms .....36

**Diagramm 2:** Ergebnisse der Tests bei einer künstlichen Datenbanklatenz von 5ms .....37

# Abkürzungsverzeichnis

HTTP	HyperText Transfer Protocol
REST	Representational State Transfer
RAM	Random-Access Memory
Cluster	Eine Gruppe von Servern

## Anhang A: ConnectionManager

ConnectionManager.Shared/\_Abstracts/DisposableObjectBase.cs

```
using Microsoft.Extensions.Logging;
using ConnectionManager.Shared._Extensions;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared._Abstracts {

    public abstract class DisposableObjectBase : IDisposable {

        #region PROPERTIES
        protected ILogger _logger { get; private set; }
        protected bool Disposed => _disposed > 0;
        protected bool Disposing => _disposed == 1;
        #endregion

        #region FIELDS
        private List<IDisposable> _disposables;
        private int _disposed; // 0 = not disposed, 1 = disposing, 2 = disposed
        #endregion

        #region CONSTRUCTORS
        public DisposableObjectBase(ILogger logger) {
            _logger = logger;
            _disposables = new List<IDisposable>();
        }
        #endregion

        #region HELPERS
        private void ActualDispose(Action callback) {
            void Try(Action action) {
                try {
                    action?.Invoke();
                } catch (Exception e) {
                    _logger.LogError(e, "Error while calling dispose handler");
                }
            }

            if (Interlocked.CompareExchange(ref _disposed, 1, 0) == 1) return;

            Try(DisposeHandler); // dispose self
            Try(() => { // dispose children
                foreach (var disposable in _disposables) {
                    try {
                        disposable?.Dispose();
                    } catch { }
                }
            });

            Try(callback);

            _disposed++;
        }
    }
}
```



```

#endregion

#region ABSTRACTS
protected abstract void DisposeHandler();
#endregion

#region FUNCTIONS
/// <summary>
/// Check if the current instance is disposed
///
/// If the caller should also run while disposing, then setting <paramref
name="allowWhileDisposing"/> to true is required
/// </summary>
/// <param name="allowWhileDisposing">if it should pass while
disposing</param>
protected void ThrowIfDisposed(bool allowWhileDisposing = false) {
    if (IsDisposed(allowWhileDisposing)) {
        throw _logger.LogError(new
ObjectDisposedException(GetType().Name), "object is disposed [this exception is
intended]");
    }
}

/// <summary>
/// Check if the current instance is disposed
///
/// If the caller should also run while disposing, then setting <paramref
name="allowWhileDisposing"/> to true is required
/// </summary>
/// <param name="allowWhileDisposing">if it should pass while
disposing</param>
protected bool IsDisposed(bool allowWhileDisposing = false) {
    return _disposed == 2 || (allowWhileDisposing && _disposed == 1);
}

/// <summary>
/// Bind a disposable to parents dispose.
///
/// Meaning the provided disposable will be automatically disposed with
the current instance.
/// </summary>
/// <param name="disposable"></param>
protected T Using<T>(T disposable) where T : IDisposable {
    if (disposable == null) return disposable;

    lock (_disposables) {
        _disposables.Add(disposable);
    }
    return disposable;
}
#endregion

#region DISPOSE
public void Dispose() {
    ActualDispose(() => {
        _logger.LogDebug(() => $"{GetType().Name} disposed");
        GC.SuppressFinalize(this);
    });
}

```

```

        ~DisposableObjectBase() {
            ActualDispose(() => {
                _logger.LogDebug(() => $"{GetType().Name} disposed via
destructor");
            });
        }
        #endregion
    }
}

```

ConnectionManager.Shared/\_Extensions/LoggerExtension.cs

```

using Microsoft.Extensions.Logging;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared._Extensions {
    public static class LoggerExtension {

        public static Exception LogError(this ILogger logger, Exception
exception, string error = "") {
            logger.LogError(exception, error, null);
            return exception;
        }

        public static void LogDebug(this ILogger logger, Func<string> message) {
            if (logger.IsEnabled(LogLevel.Debug)) {
                logger.LogDebug(message());
            }
        }
    }
}

```

ConnectionManager.Shared/Network/\_Abstracts/SocketHandlerBase.cs

```

using ConnectionManager.Shared._Abstracts;

using Microsoft.Extensions.Logging;
using ConnectionManager.Shared._Extensions;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Network._Abstracts {
    public abstract class SocketHandlerBase : DisposableObjectBase {

        #region FIELDS

```

```

private Socket _socket;
private byte[] _buffer;
protected EndPoint _endpoint;
protected bool _listen = true;
#endregion

#region CONSTRUCTORS
public SocketHandlerBase(Socket socket, int bufferSize, ILogger logger) :
base(logger) {
    _socket = socket ?? throw _logger.LogError(new
ArgumentNullException(nameof(socket)), "Error while constructing socket
handler!");
    _endpoint = _socket.RemoteEndPoint;
    _buffer = new byte[bufferSize];
}
#endregion

#region HELPERS
protected void BeginReceive() {
    if (Disposing) return;

    try {
        if (_socket != null) {
            _socket.BeginReceive(_buffer, 0, _buffer.Length,
SocketFlags.None, new AsyncCallback(OnReceiveData), null);
        }
    } catch (Exception e) {
        _logger.LogError(e, "Error while starting to receive!");
        Dispose();
    }
}

private void OnReceiveData(IAsyncResult result) {
    if (Disposing) return;
    int bytesCount = 0;

    try {
        if (_socket != null) {
            bytesCount = _socket.EndReceive(result);
        }
    } catch { }

    if (bytesCount < 1 || bytesCount > _buffer.Length) {
        Dispose();
        return;
    }

    Receive(_buffer.AsMemory(0, bytesCount));

    if (_listen) {
        BeginReceive();
    }
}

private void OnDataSend(IAsyncResult result) {
    if (Disposing) return;

    try {
        if (_socket != null) {
            _socket.EndSend(result);

```

```

        }
        } catch (Exception e) {
            _logger.LogError(e, "Error while sending!");
            Dispose();
        }
    }
}
#endregion

#region FUNCTIONS
public abstract void Receive(Memory<byte> data);
public void Send(byte[] data) {
    if (Disposing || data == null || data.Length == 0) return;

    try {
        if (_socket != null && _socket.Connected) {
            _socket.BeginSend(data, 0, data.Length, SocketFlags.None, new
AsyncCallback(OnDataSend), null);
        }

        } catch (Exception e) {
            _logger.LogError(e, "Error while starting to send!");
            Dispose();
        }
    }
}
#endregion

#region DISPOSE
protected override void DisposeHandler() {
    _listen = false;

    try {
        _logger.LogInformation($"Client [{_endpoint}] disconnected!");
        _socket.Close(100);
    } catch (Exception e) {
        _logger.LogError(e, "Error while closing connection");
    }
}
#endregion

}
}

```

ConnectionManager.Shared/Network/\_Abstracts/SocketListenerBase.cs

```

using ConnectionManager.Shared._Abstracts;

using Microsoft.Extensions.Logging;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Network._Abstracts {
    public abstract class SocketListenerBase : DisposableObjectBase {

```

```

        #region PROPERTIES
        public bool IsRunning { get => _isRunning; internal set => _isRunning =
value; }
        #endregion

        #region FIELDS
        protected IPEndPoint _endpoint;
        protected Socket _socket;
        private bool _isRunning;
        #endregion

        #region CONSTRUCTORS
        protected SocketListenerBase(IPEndPoint localEndpoint, int backlog,
ILogger logger) : base(logger) {
            _endpoint = localEndpoint;
            _socket = new Socket(localEndpoint.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
            _socket.Bind(localEndpoint);
            _socket.Listen(backlog);
            // _socket.Blocking = false;

            _isRunning = true;
            _logger.LogInformation($"SocketListener started listening on
{_socket.LocalEndPoint}");

            BeginAccept();
        }
        #endregion

        #region HELPERS
        private void BeginAccept() {
            if (!_isRunning) {
                return;
            }

            try {
                _socket.BeginAccept(HandleConnection, null);
            } catch (Exception e) {
                _logger.LogError(e, "Error while initializing accepting
connection!");
            }
        }

        private void HandleConnection(IAsyncResult result) {
            if (!_isRunning) {
                return;
            }

            try {

                Socket client = _socket.EndAccept(result);

                // client.NoDelay = true;
                // client.Blocking = false;

                if (_logger.IsEnabled(LogLevel.Debug)) {
                    _logger.LogDebug($"Connection from [{client.RemoteEndPoint}]
accepted!");
                }
            }
        }
    }

```

```

        Accept(client);

    } catch (Exception e) {
        _logger.LogError(e, "Error while accepting connection!");
    }

    BeginAccept();
}
#endregion

#region FUNCTIONS
public void Stop() {
    _isRunning = false;
    _logger.LogWarning($"SocketListener stopped listening on
{_endpoint}");
}

protected abstract void Accept(Socket socket);
#endregion

#region DISPOSE
protected override void DisposeHandler() {
    Stop();
    if (_socket != null) {
        _socket.Dispose();
        _socket = null;
    }
}
#endregion
}
}

```

ConnectionManager.Shared/Services/Cache/\_Interfaces/ICacheService.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Services.Cache._Interfaces {
    public interface ICacheService {

        Task<T?> Get<T>(string key, T? defaultValue = default);
        Task Set<T>(string key, T value);
        Task Delete(string key);

        Task<string[]> GetIdentifiersFromCounter(string key, bool ascending, int
count);
        Task<long> GetIdentifiersCountFromCounter(string key);
        Task IncrementCounter(string key, string identifier);
        Task DecrementCounter(string key, string identifier);
        Task ResetCounter(string key, string identifier, bool delete = false);
        Task DeleteCounter(string key);

    }
}

```

ConnectionManager.Shared/Services/Cache/RedisCacheService.cs

```
using ConnectionManager.Shared.Services.Cache._Interfaces;

using Newtonsoft.Json;

using StackExchange.Redis;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Services.Cache {
    public class RedisCacheService : ICacheService {

        #region FIELDS
        private IConnectionMultiplexer _redis;
        private IDatabase _database => _redis.GetDatabase(10);
        #endregion

        #region CONSTRUCTORS
        public RedisCacheService(IConnectionMultiplexer redis) {
            _redis = redis;
        }
        #endregion

        #region KEY-VALUE
        public async Task Delete(string key) {
            await _database.KeyDeleteAsync(key);
        }

        public async Task<T?> Get<T>(string key, T? defaultValue = default) {
            var value = await _database.StringGetAsync(key);

            if (!value.HasValue || value.IsNull) {
                if (!EqualityComparer<T?>.Default.Equals(defaultValue, default))
                {
                    await _database.StringSetAsync(key,
                        JsonConvert.SerializeObject(defaultValue), when: When.NotExists);
                }

                return defaultValue;
            }

            return JsonConvert.DeserializeObject<T>(value);
        }

        public async Task Set<T>(string key, T value) {
            await _database.StringSetAsync(key,
                JsonConvert.SerializeObject(value));
        }
        #endregion

        #region COUNTERS
        public async Task<long> GetIdentifiersCountFromCounter(string key) {
            return await _database.SortedSetLengthAsync(key);
        }
    }
}
```

```

        public async Task<string[]> GetIdentifiersFromCounter(string key, bool
ascending, int maxCount) {
            var result = await _database.SortedSetRangeByScoreAsync(key, take:
maxCount, order: ascending ? Order.Ascending : Order.Descending);
            return result.Select(x => x.ToString()).ToArray();
        }

        public async Task IncrementCounter(string key, string identifier) {
            await _database.SortedSetIncrementAsync(key, identifier, 1);
        }

        public async Task DecrementCounter(string key, string identifier) {
            if (await _database.SortedSetDecrementAsync(key, identifier, 1) < 0)
            {
                await IncrementCounter(key, identifier);
            }
        }

        public async Task ResetCounter(string key, string identifier, bool delete
= false) {
            if (delete) {
                await _database.SortedSetRemoveAsync(key, identifier);
            } else {
                await _database.SortedSetAddAsync(key, identifier, 0);
            }
        }

        public async Task DeleteCounter(string key) {
            await _database.SortedSetRemoveRangeByValueAsync(key,
double.NegativeInfinity, double.PositiveInfinity);
        }
    }
}
#endregion
}
}

```

ConnectionManager.Shared/Services/Connection/\_Interfaces/IConnectionClientService  
 .CS

```

using ConnectionManager.Shared.Services.Connection._Models;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Services.Connection._Interfaces {
    public interface IConnectionClientService {

        Task<ConnectionClientState> Get(Guid connectionId);
        Task Set(Guid connectionId, ConnectionClientState clientState);
        Task Delete(Guid connectionId);

    }
}

```



ConnectionManager.Shared/Services/Connection/\_Interfaces/IConnectionHandlersService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Services.Connection._Interfaces {
    public interface IConnectionHandlersService {

        Task Register(string identifier);
        Task Unregister(string identifier);
        Task IncrementConnections(string identifier);
        Task DecrementConnection(string identifier);
        Task<string> GetIdleServer();

        Task<string> GetRegionHandler(string region, string defaultIdentifier);
        Task AssignRegionHandler(string region, string identifier);
        Task DeleteRegionHandler(string region);

    }
}
```

ConnectionManager.Shared/Services/Connection/\_Models/\_Enumerables/ConnectionStates.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Services.Connection._Models._Enumerables {
    public enum ConnectionState {

        INITIAL,
        READY,
        CONNECTED,
        CLOSED

    }
}
```

ConnectionManager.Shared/Services/Connection/\_Models/ConnectionClientState.cs

```
using ConnectionManager.Shared.Services.Connection._Models._Enumerables;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Services.Connection._Models {
    public class ConnectionClientState {
```

```

        public ConnectionState State { get; set; }
        public string ClientId { get; set; }
        public string Region { get; set; }
    }
}

```

ConnectionManager.Shared/Services/Connection/ConnectionClientService.cs

```

using ConnectionManager.Shared.Services.Cache._Interfaces;
using ConnectionManager.Shared.Services.Connection._Interfaces;
using ConnectionManager.Shared.Services.Connection._Models;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Services.Connection {
    public class ConnectionClientService : IConnectionClientService {

        #region CONSTANTS
        private const string CONNECTION = "connection_";
        #endregion

        #region FIELDS
        private ICacheService _cache;
        #endregion

        #region CONSTRUCTORS
        public ConnectionClientService(ICacheService cache) {
            _cache = cache;
        }
        #endregion

        #region HELPERS
        private string GetKey(Guid connectionId) {
            return CONNECTION + connectionId.ToString();
        }
        #endregion

        #region FUNCTIONS
        public async Task Delete(Guid connectionId) {
            await _cache.Delete(GetKey(connectionId));
        }

        public async Task<ConnectionClientState> Get(Guid connectionId) {
            return await _cache.Get<ConnectionClientState>(GetKey(connectionId),
null);
        }

        public async Task Set(Guid connectionId, ConnectionClientState
clientState) {
            await _cache.Set(GetKey(connectionId), clientState);
        }
        #endregion
    }
}

```

```

    }
}

```

ConnectionManager.Shared/Services/Connection/ConnectionHandlersService.cs

```

using ConnectionManager.Shared.Services.Cache._Interfaces;
using ConnectionManager.Shared.Services.Connection._Interfaces;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Shared.Services.Connection {
    public class ConnectionHandlersService : IConnectionHandlersService {

        #region CONSTANTS
        private const string HANDLERS = "handlers_counter";
        private const string REGIONS = "region_";
        #endregion

        #region FIELDS
        private ICacheService _cache;

        private ThreadLocal<Random> _random;
        private int _seed;
        #endregion

        #region CONSTRUCTORS
        public ConnectionHandlersService(ICacheService cache) {
            _cache = cache;

            _seed = Environment.TickCount;
            _random = new ThreadLocal<Random>(() => new
Random(Interlocked.Increment(ref _seed)));
        }
        #endregion

        #region FUNCTIONS
        public async Task<string> GetIdleServer() {
            var result = await _cache.GetIdentifiersFromCounter(HANDLERS, true,
5);

            if (result == null || result.Length == 0) {
                return null;
            }

            if (result.Length == 1) {
                return result[0];
            }

            return result[_random.Value.Next(0, result.Length)]; // get any
random
        }

        public async Task Register(string identifier) {
            await _cache.ResetCounter(HANDLERS, identifier); // reset without
delete = set to 0
        }
    }
}

```

```

        public async Task Unregister(string identifier) {
            await _cache.ResetCounter(HANDLERS, identifier, true); // reset with
delete = delete
        }

        public async Task IncrementConnections(string identifier) {
            await _cache.IncrementCounter(HANDLERS, identifier);
        }

        public async Task DecrementConnection(string identifier) {
            await _cache.DecrementCounter(HANDLERS, identifier);
        }

        public async Task<string> GetRegionHandler(string region, string
defaultIdentifier) {
            return await _cache.Get(REGIONS + region, defaultIdentifier);
        }

        public async Task AssignRegionHandler(string region, string identifier) {
            await _cache.Set(REGIONS + region, identifier);
        }

        public async Task DeleteRegionHandler(string region) {
            await _cache.Delete(REGIONS + region);
        }
        #endregion
    }
}

```

ConnectionManager/Network/ConnectionHandler.cs

```
using ConnectionManager.Shared.Network._Abstracts;
using ConnectionManager.Shared.Services.Connection._Interfaces;
using ConnectionManager.Shared.Services.Connection._Models;
using ConnectionManager.Shared.Services.Connection._Models._Enumerables;

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

using Nito.AsyncEx;

using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Network {
    internal class ConnectionHandler : SocketHandlerBase {

        #region PROPERTIES
        public Guid ConnectionId { get; } = Guid.NewGuid();
        #endregion

        #region FIELDS
        private IConnectionClientService _connectionClient;
        private IConnectionHandlersService _connectionHandlers;
        private ConnectionHandlerClient _client;
        #endregion

        #region INTERNAL
        internal async Task Connect(string region = null) {
            string handler = await _connectionHandlers.GetIdleServer();

            _logger.LogDebug($"Idle server: {handler}");

            if (region != null) {
                handler = await _connectionHandlers.GetRegionHandler(region,
handler);
            }

            _logger.LogDebug($"Actual server: {handler} for Region: {region}");
            _client = new ConnectionHandlerClient(this, handler, _logger);
        }

        internal async Task HandlerDisconnected() {
            _client = null;
            var state = await _connectionClient.Get(ConnectionId);

            _logger.LogDebug($"Client {ConnectionId} returned to manager! State:
{state.State} and Region: {state.Region}");

            switch (state.State) {
                case ConnectionState.CLOSED:
                    Dispose();
                    break;
                case ConnectionState.READY:
```

```

        case ConnectionState.INITIAL: // reconnect no big deal
            await Connect(state.Region);
            break;
        case ConnectionState.CONNECTED: // server crashed
            // ... for now we simply revoke
the servers ability for current region and assign to new server
            // usually you would want to
check if the server is really down before migrating a whole region

            // currently we simply reset the region in the cache and
order a reconnect
            // this is possible because the handler assigns region =
connectionId meaning each region has only this connection
            await _connectionHandlers.Unregister(await
_connectionHandlers.GetRegionHandler(state.Region, ""));
            await _connectionHandlers.DeleteRegionHandler(state.Region);
            await Connect(state.Region);
            break;
    }
}

internal async Task Initialize() {
    await _connectionClient.Set(ConnectionId, new() {
        State = ConnectionState.INITIAL
    });

    await Connect();
}
#endregion

#region FUNCTIONS
public ConnectionHandler(Socket socket, IConnectionClientService
connectionClientService, IConnectionHandlersService connectionHandlersService,
ILogger logger) : base(socket, 1024, logger) {
    _connectionClient = connectionClientService;
    _connectionHandlers = connectionHandlersService;

    AsyncContext.Run(Initialize); // Extract Initialize to be called by
initializer in an async scope
    BeginReceive();
}

public override void Receive(Memory<byte> data) {
    // TODO retain data while connection is pending (instead of dropping)
    _logger.LogDebug($"Client {ConnectionId} sent data! Server accepting
messages: {_client.IsReady}");
    if (_client != null && _client.IsReady) {
        _client.Send(data.ToArray());
    }
}

protected override void DisposeHandler() {
    AsyncContext.Run(() => _connectionClient.Delete(ConnectionId)); //
not optimal, opt to use IAsyncDispose
    _client?.Dispose();

    base.DisposeHandler();
}
#endregion

```

```

    }
}

```

ConnectionManager/Network/ConnectionHandlerClient.cs

```

using ConnectionManager.Shared._Abstracts;
using ConnectionManager.Shared.Network._Abstracts;

using Microsoft.Extensions.Logging;

using Nito.AsyncEx;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Network {
    internal class ConnectionHandlerClient : SocketHandlerBase {

        #region STATIC
        private static Socket CreateClient(string endpoint) {
            var socket = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp) {
                // Blocking = false
            };

            socket.Connect(IPEndPoint.Parse(endpoint));
            return socket;
        }
        #endregion

        #region PROPERTIES
        public bool IsReady { get; private set; }
        #endregion

        #region FIELDS
        private ConnectionHandler _handler;
        #endregion

        #region CONSTRUCTORS
        public ConnectionHandlerClient(ConnectionHandler handler, string
endpoint, ILogger logger) : base(CreateClient(endpoint), 1024, logger) {
            _handler = handler;

            BeginReceive();

            _logger.LogDebug($"Client {handler.ConnectionId} sent id to
server!");
            Send(handler.ConnectionId.ToByteArray());
        }
        #endregion

        #region FUNCTIONS
        public override void Receive(Memory<byte> data) {
            if (!IsReady) {
                if (data.Length >= 2) {

```

```

        _logger.LogDebug($"Client {_handler.ConnectionId} data
received from server!");

        var readyFlag = data.Slice(0, 2).ToArray();
        if (readyFlag[0] == 123 && readyFlag[1] == 195) {
            IsReady = true;
            data = data.Slice(2); // remove the first part

            _logger.LogDebug($"Client {_handler.ConnectionId} server
activated the connection!");
        }
    }

    if (!IsReady) {
        _logger.LogCritical("Received invalid ready flag!");
        Dispose();
        return;
    }

    if (_handler != null) {
        _logger.LogDebug($"Client {_handler.ConnectionId} data being
redirected to client!");
        _handler.Send(data.ToArray());
    }
}
#endregion

#region DISPOSE
protected override void DisposeHandler() {
    AsyncContext.Run(_handler.HandlerDisconnected); // IAsyncDispose
}
#endregion
}
}

```

ConnectionManager/Network/ConnectionListener.cs

```

using ConnectionManager.Shared.Network._Abstracts;
using ConnectionManager.Shared.Services.Connection._Interfaces;

using Microsoft.Extensions.Logging;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Network {
    public class ConnectionListener : SocketListenerBase {

        #region FIELDS
        private IConnectionClientService _connectionClient;
        private IConnectionHandlersService _connectionHandlers;
        #endregion
    }
}

```



```

        #region CONSTRUCTORS
        public ConnectionListener(IPEndPoint localEndpoint,
            IConnectionClientService connectionClientService, IConnectionHandlersService
            connectionHandlersService, ILogger logger) : base(localEndpoint, 100, logger) {
            _connectionClient = connectionClientService;
            _connectionHandlers = connectionHandlersService;
        }
        #endregion

        #region HELPERS
        protected override void Accept(Socket socket) {
            new ConnectionHandler(socket, _connectionClient, _connectionHandlers,
            _logger);
        }
        #endregion
    }
}

```

ConnectionManager/Program.cs

```

// See https://aka.ms/new-console-template for more information
using ConnectionManager.Network;
using ConnectionManager.Shared.Services.Cache;
using ConnectionManager.Shared.Services.Cache._Interfaces;
using ConnectionManager.Shared.Services.Connection;
using ConnectionManager.Shared.Services.Connection._Interfaces;

using Microsoft.Extensions.DependencyInjection;
using System.Net;

using StackExchange.Redis;
using Microsoft.Extensions.Logging;

IServiceProvider serviceProvider = new ServiceCollection()
    .AddLogging(opt => opt.AddFilter((_, _, _) => true).AddConsole())
    .AddSingleton<IConnectionMultiplexer>(_ =>
    ConnectionMultiplexer.Connect("127.0.0.1:6379"))
    .AddSingleton<ICacheService, RedisCacheService>()
    .AddSingleton<IConnectionHandlersService, ConnectionHandlersService>()
    .AddSingleton<IConnectionClientService, ConnectionClientService>()
    .AddSingleton<ConnectionListener>(serviceProvider => new(
        new IPEndPoint(IPAddress.Any, 9500),
        serviceProvider.GetService<IConnectionClientService>(),
        serviceProvider.GetService<IConnectionHandlersService>(),
        serviceProvider.GetService<ILogger<Program>>())) // not a really clean
logging impl but meh
    .BuildServiceProvider();

var server = serviceProvider.GetService<ConnectionListener>();

Console.ReadLine();
server.Stop();

```

ConnectionManager.Server/Network/ConnectionHandler.cs

```
using ConnectionManager.Shared.Network._Abstracts;
using ConnectionManager.Shared.Services.Cache._Interfaces;
using ConnectionManager.Shared.Services.Connection._Interfaces;
using ConnectionManager.Shared.Services.Connection._Models;
using ConnectionManager.Shared.Services.Connection._Models._Enumerables;

using Microsoft.Extensions.Logging;

using Nito.AsyncEx;

using System.Net.Sockets;
using System.Text;

namespace ConnectionManager.Server.Network {
    internal class ConnectionHandler : SocketHandlerBase {

        #region PROPERTIES
        public Guid ConnectionId { get; private set; }
        #endregion

        #region FIELDS
        private ConnectionListener _listener;
        private IConnectionClientService _connectionClient;
        private IConnectionHandlersService _connectionHandlers;
        private ICacheService _cache;
        private bool _isReady;

        private Memory<byte> _data;
        private ConnectionState _connectionState;
        #endregion

        #region INTERNAL
        private async Task Initialize() {
            _connectionState = await _connectionClient.Get(ConnectionId);
            if (_connectionState.State == ConnectionState.INITIAL) {
                _logger.LogDebug($"Client {ConnectionId} auth required sent!");
                Send("authentication required\n");
            } else if (_connectionState.State == ConnectionState.READY) {
                _connectionState.State = ConnectionState.CONNECTED;
                await _connectionClient.Set(ConnectionId, _connectionState);
            }
        }

        private bool TryReadLine(out string line) {
            line = default;

            var indexOfNewLine = _data.Span.IndexOf((byte)'\n');
            if (indexOfNewLine > -1) {
                line = Encoding.UTF8.GetString(_data.Slice(0,
indexOfNewLine).Span).Trim();
                _data = _data.Slice(indexOfNewLine + 1);
                return true;
            }

            return false;
        }

        private void Process() {
```

```

        if (_data.Length <= 0) return;
        if (!TryReadLine(out string line)) {
            return;
        }

        _logger.LogDebug($"Client {ConnectionId} processing line: {line}!");
        var parts = line.Split(' ');
        if (parts.Length > 0) {

            if (parts[0].ToLower() == "authenticate") {
                _logger.LogDebug($"Client {ConnectionId} authenticated as:
{parts[1]}!");

                if (parts[1] == "nope") {
                    _connectionState.State = ConnectionState.CLOSED;
                } else {
                    _connectionState.ClientId = parts[1];
                    _connectionState.Region = _connectionState.ClientId;
                    _connectionState.State = ConnectionState.READY;
                }

                AsyncContext.Run(() => _connectionClient.Set(ConnectionId,
_connectionState)); // save to cache and close for connectionManager to handle
                Dispose();
            }

            if (_connectionState.State == ConnectionState.CONNECTED) {
                switch (parts[0].ToLower()) {
                    case "ping":
                        Send($"pong {DateTime.UtcNow}\\n");
                        break;
                    case "status":
                        Send($"You are currently to connected to
{_listener.ServerId}\\n");
                        break;
                    case "increment":
                        var current = AsyncContext.Run(() =>
_cache.Get(_connectionState.ClientId, 0));
                        AsyncContext.Run(() =>
_cache.Set(_connectionState.ClientId, ++current));

                        Send($"Current value: {current}");
                        break;
                }
            }
        }

        Process(); // repeat because there may be more data to process
    }
#endregion

#region FUNCTIONS
    public ConnectionHandler(Socket socket, ConnectionListener listener,
ICacheService cacheService, IConnectionClientService connectionClientService,
IConnectionHandlersService connectionHandlersService, ILogger logger) :
base(socket, 1024, logger) {
        _listener = listener;
        _cache = cacheService;
        _connectionClient = connectionClientService;
        _connectionHandlers = connectionHandlersService;
    }

```

```

        AsyncContext.Run(() =>
            _connectionHandlers.IncrementConnections(listener.ServerId));
        BeginReceive();
    }

    public override void Receive(Memory<byte> data) {
        _logger.LogDebug($"Client {ConnectionId} data received!");

        // TODO retain data while connection is pending (instead of dropping)
        if (!_isReady) {
            if (data.Length >= 16) {
                ConnectionId = new Guid(data.Slice(0, 16).ToArray());
                data = data.Slice(16);

                _isReady = true;

                _logger.LogDebug($"Client {ConnectionId} connection id
received and activated!");
                Send(new byte[] { 123, 195 }); // activate message-passing
            }

            if (!_isReady) {
                _logger.LogCritical("Failed to read connection id!");
                Dispose();
                return;
            }

            AsyncContext.Run(Initialize); // again consider moving all to
async
        }

        if (data.Length > 0) {
            if (_data.Length == 0) {
                _data = data;
            } else {
                Memory<byte> buffer = new Memory<byte>(new byte[_data.Length
+ data.Length]);
                _data.CopyTo(buffer);
                data.CopyTo(buffer.Slice(_data.Length));
                _data = buffer;
            }
        }

        Process();
    }

    public void Send(string message) {
        Send(Encoding.UTF8.GetBytes(message));
    }
#endregion

#region DISPOSE
    protected override void DisposeHandler() {
        AsyncContext.Run(() =>
            _connectionHandlers.DecrementConnection(_listener.ServerId));
        base.DisposeHandler();
    }
#endregion

```

```

    }
}

```

ConnectionManager.Server/Network/ConnectionListener.cs

```

using ConnectionManager.Shared.Network._Abstracts;
using ConnectionManager.Shared.Services.Cache._Interfaces;
using ConnectionManager.Shared.Services.Connection._Interfaces;

using Microsoft.Extensions.Logging;

using Nito.AsyncEx;

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace ConnectionManager.Server.Network {
    public class ConnectionListener : SocketListenerBase {

        #region PROPERTIES
        public string ServerId { get; private set; }
        #endregion

        #region FIELDS
        private ICacheService _cache;
        private IConnectionClientService _connectionClient;
        private IConnectionHandlersService _connectionHandlers;
        #endregion

        #region CONSTRUCTORS
        public ConnectionListener(IPEndPoint localEndpoint, ICacheService
cacheService, IConnectionClientService connectionClientService,
IConnectionHandlersService connectionHandlersService, ILogger logger) :
base(localEndpoint, 100, logger) {
            _cache = cacheService;
            _connectionClient = connectionClientService;
            _connectionHandlers = connectionHandlersService;

            Initialize();
        }
        #endregion

        #region HELPERS
        protected void Initialize() {
            ServerId = "127.0.0.1:" + (_socket.LocalEndPoint as IPEndPoint).Port;
            AsyncContext.Run(() => _connectionHandlers.Register(ServerId));
        }

        protected override void Accept(Socket socket) {
            new ConnectionHandler(socket, this, _cache, _connectionClient,
_connectionHandlers, _logger);
        }
        #endregion

        #region DISPOSE

```

```

        protected override void DisposeHandler() {
            AsyncContext.Run(() => _connectionHandlers.Unregister(ServerId));
            base.DisposeHandler();
        }
    #endregion
}
}

```

ConnectionManager.Server/Program.cs

```

using ConnectionManager.Shared.Services.Cache;
using ConnectionManager.Shared.Services.Cache._Interfaces;
using ConnectionManager.Shared.Services.Connection;
using ConnectionManager.Shared.Services.Connection._Interfaces;

using Microsoft.Extensions.DependencyInjection;
using System.Net;

using StackExchange.Redis;
using Microsoft.Extensions.Logging;
using ConnectionManager.Server.Network;

IServiceProvider serviceProvider = new ServiceCollection()
    .AddLogging(opt => opt.AddFilter( (_, _, _) => true).AddConsole())
    .AddSingleton<IConnectionMultiplexer>(_ =>
ConnectionMultiplexer.Connect("127.0.0.1:6379"))
    .AddSingleton<ICacheService, RedisCacheService>()
    .AddSingleton<IConnectionHandlersService, ConnectionHandlersService>()
    .AddSingleton<IConnectionClientService, ConnectionClientService>()
    .AddSingleton<ConnectionListener>(serviceProvider => new(
        new IPEndPoint(IPAddress.Any, 0),
        serviceProvider.GetService<ICacheService>(),
        serviceProvider.GetService<IConnectionClientService>(),
        serviceProvider.GetService<IConnectionHandlersService>(),
        serviceProvider.GetService<ILogger<Program>>())) // not a really clean
logging impl but meh
    .BuildServiceProvider();

var server = serviceProvider.GetService<ConnectionListener>();

Console.ReadLine();
server.Stop();

```

ConnectionManager.Client/Program.cs

```
using System.Net;
using System.Net.Sockets;
using System.Text;

TcpClient client = new TcpClient("127.0.0.1", 9500);
NetworkStream stream = client.GetStream();

Task.Run(async () => {
    byte[] buffer = new byte[1024];
    while (true) {
        int length = stream.Read(buffer, 0, buffer.Length);
        if (length == 0) {
            Console.WriteLine("Connection closed!");
            return;
        }

        Console.WriteLine("SERVER >> {0}", Encoding.UTF8.GetString(buffer, 0,
length));
    }
});

while (true) {
    string line = Console.ReadLine();
    if (line == "exit") {
        client.Close();
        return;
    } else {
        var data = Encoding.UTF8.GetBytes(line + "\n");
        stream.Write(data, 0, data.Length);
    }
}
```

## Anhang B: TestApplikation

TestApplikation/Repositories/\_Interfaces/IRepository.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestApplikation.Repositories._Interfaces {
    public interface IRepository<T> {

        Task<T> Get(int id);
        Task<bool> Set(int id, T data);

    }
}
```

TestApplikation/Repositories/UserInMemoryRepository.cs

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using TestApplikation.Repositories._Interfaces;
using TestApplikation.ViewModels.User;

namespace TestApplikation.Repositories {
    public class UserInMemoryRepository : IRepository<UserView> {

        #region FIELDS
        private ConcurrentDictionary<int, UserView> _users;
        #endregion

        #region CONSTRUCTORS
        public UserInMemoryRepository() {
            _users = new ConcurrentDictionary<int, UserView>();
        }
        #endregion

        #region FUNCTIONS
        public async Task<UserView> Get(int id) {
            await Task.Delay(50); // artificial delay
            if (_users.TryGetValue(id, out var user)) {
                return user;
            }
            return null;
        }

        public async Task<bool> Set(int id, UserView data) {
            await Task.Delay(50); // artificial delay
            _users.AddOrUpdate(id, data, (_, _) => data);
            return true;
        }
    }
    #endregion
}
```



```
    }  
}
```

TestApplikation/Repositories/UserRedisRepository.cs

```
using Newtonsoft.Json;  
  
using StackExchange.Redis;  
  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
using TestApplikation.Repositories._Interfaces;  
using TestApplikation.ViewModels.User;  
  
namespace TestApplikation.Repositories {  
    public class UserRedisRepository : IRepository<UserView> {  
        #region FIELDS  
        private IConnectionMultiplexer _redis;  
        private IDatabase _database => _redis.GetDatabase(10);  
        #endregion  
  
        #region CONSTRUCTORS  
        public UserRedisRepository(IConnectionMultiplexer redis) {  
            _redis = redis;  
        }  
        #endregion  
  
        #region FUNCTIONS  
        public async Task<UserView> Get(int id) {  
            await Task.Delay(5);  
            var value = await _database.StringGetAsync(id.ToString());  
            if (!value.HasValue || value.IsNull) {  
                return null;  
            }  
  
            return JsonConvert.DeserializeObject<UserView>(value);  
        }  
  
        public async Task<bool> Set(int id, UserView data) {  
            await Task.Delay(5);  
            return await _database.StringSetAsync(id.ToString(),  
JsonConvert.SerializeObject(data));  
        }  
        #endregion  
    }  
}
```

TestApplikation/Services/User/\_Interfaces/IUserService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using TestApplikation.ViewModels.User;

namespace TestApplikation.Services.User._Interfaces {
    public interface IUserService {

        Task<UserView> GetUserById(int id);
        Task<bool> UpdateUserById(int id, UserView user);

    }
}
```

TestApplikation/Services/User/UserService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using TestApplikation.Repositories._Interfaces;
using TestApplikation.Services.User._Interfaces;
using TestApplikation.ViewModels.User;

namespace TestApplikation.Services.User {
    public class UserService : IUserService {

        #region FIELDS
        private IRepository<UserView> _repository;
        #endregion

        #region CONSTRUCTORS
        public UserService(IRepository<UserView> repository) {
            _repository = repository;
        }
        #endregion

        #region FUNCTIONS
        public Task<UserView> GetUserById(int id) {
            return _repository.Get(id);
        }

        public Task<bool> UpdateUserById(int id, UserView user) {
            return _repository.Set(id, user);
        }
        #endregion

    }
}
```

TestApplikation/ViewModels/User/UserView.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestApplikation.ViewModels.User {
    public class UserView {

        public string Firstname { get; set; }
        public string Lastname { get; set; }
    }
}
```

TestApplikation.OrleansApi/Controllers/UsersController.cs

```
using Microsoft.AspNetCore.Mvc;

using Orleans;
using TestApplikation.OrleansApi.Grains._Interfaces;

using TestApplikation.Services.User._Interfaces;
using TestApplikation.ViewModels.User;

namespace TestApplikation.OrleansApi.Controllers {

    [ApiController]
    public class UsersController : ControllerBase {

        [HttpGet("/users/{id:int}")]
        public async Task<IActionResult> Get([FromRoute] int id, [FromServices]
IUserService userService) {
            var result = await userService.GetUserById(id);
            if (result == null) {
                return NotFound();
            }

            return Ok(result);
        }

        [HttpPatch("/users/{id:int}")]
        public async Task<IActionResult> Update([FromRoute] int id, [FromBody]
UIView user, [FromServices] IUserService userService) {
            await userService.UpdateUserById(id, user);
            return Ok();
        }

        [HttpGet("/orleans_users/{id:int}")]
        public async Task<IActionResult> GetFromOrleans([FromRoute] int id,
[FromServices] IClusterClient clusterClient) {
            var result = await clusterClient.GetGrain<IUserGrain>(id).GetUser();
            if (result == null) {
                return NotFound();
            }

            return Ok(result);
        }

        [HttpPatch("/orleans_users/{id:int}")]
        public async Task<IActionResult> UpdateInOrleans([FromRoute] int id,
[FromBody] UIView user, [FromServices] IClusterClient clusterClient) {
            await clusterClient.GetGrain<IUserGrain>(id).UpdateUser(user);
            return Ok();
        }

        public const string HelloWorld = "Hello!";

        [HttpGet("/test")]
        public async Task<IActionResult> Test() {
            return Ok(HelloWord);
        }
    }
}
```

```
}
```

TestApplikation.OrleansApi/Grains/\_Interfaces/IUserGrain.cs

```
using Orleans;
using Orleans.Concurrency;

using TestApplikation.ViewModels.User;

namespace TestApplication.OrleansApi.Grains._Interfaces {

    public interface IUserGrain : IGrainWithIntegerKey {

        [ReadOnly, AlwaysInterleave]
        Task<UserView> GetUser();

        Task UpdateUser(UserView user);

    }

}
```

TestApplikation.OrleansApi/Grains/UserGrain.cs

```
using Orleans;

using TestApplication.OrleansApi.Grains._Interfaces;

using TestApplikation.Services.User._Interfaces;
using TestApplikation.ViewModels.User;

namespace TestApplication.OrleansApi.Grains {

    public class UserGrain : Grain, IUserGrain {

        #region FIELDS
        private bool _loaded;
        private UserView _userView;
        private IUserService _userService;
        #endregion

        #region CONSTRUCTORS
        public UserGrain(IUserService userService) {
            _userService = userService;
        }
        #endregion

        #region FUNCTIONS
        public async Task<UserView> GetUser() {
            if (!_loaded) {
                _loaded = true;
                _userView = await
                _userService.GetUserById((int)this.GetPrimaryKeyLong());
            }

            return _userView;
        }

    }

}
```

```

        public async Task UpdateUser(UserView user) {
            if (await _userService.UpdateUserById((int)this.GetPrimaryKeyLong(),
user)) {
                _userView = user;
            }
        }
    }
    #endregion
}
}

```

TestApplikation.OrleansApi/Program.cs

```

using Orleans;
using Orleans.Configuration;
using Orleans.Hosting;

using StackExchange.Redis;

using System.Net;

using TestApplication.OrleansApi.Grains;
using TestApplication.OrleansApi.Grains._Interfaces;

using TestApplikation.Repositories;
using TestApplikation.Repositories._Interfaces;
using TestApplikation.Services.User;
using TestApplikation.Services.User._Interfaces;
using TestApplikation.ViewModels.User;

static System.Net.IPAddress GetPublicIp(string serviceUrl =
"https://ipinfo.io/ip") {
    return System.Net.IPAddress.Parse(new
System.Net.WebClient().DownloadString(serviceUrl));
}

var builder = WebApplication.CreateBuilder(args);

builder.Host.UseOrleans(opt => {
    opt.UseRedisClustering(opt => {
        opt.ConnectionString = "127.0.0.1:6379,password=hallo1234567890"; // This
is the default
        opt.Database = 0;
    })
    .Configure<EndpointOptions>(opt => {
        opt.SiloPort = 8080;
        opt.GatewayPort = 0;
        opt.AdvertisedIPAddress = GetPublicIp();
    })
    .Configure<ClusterOptions>(options => {
        options.ClusterId = "c-1.0.0";
        options.ServiceId = "testApplication";
    })
    .ConfigureApplicationParts(builder =>
builder.AddApplicationPart(typeof(IUserGrain).Assembly));
});

// Add services to the container.

builder.Services.AddControllers();

```

```

// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddSingleton<IConnectionMultiplexer>(_ =>
    ConnectionMultiplexer.Connect("127.0.0.1:6379,password=hallo1234567890"));
builder.Services.AddSingleton<IRepository<UserView>, UserRedisRepository>();
builder.Services.AddSingleton<IUserService, UserService>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment()) {
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

```