

HTTP

Dienstag, 6. September 2022 10:03

URL:

http:// www.google.com /test/abc ? parameter=1 & pl=test . -

Protovoll

IP-Adr.

Ordner

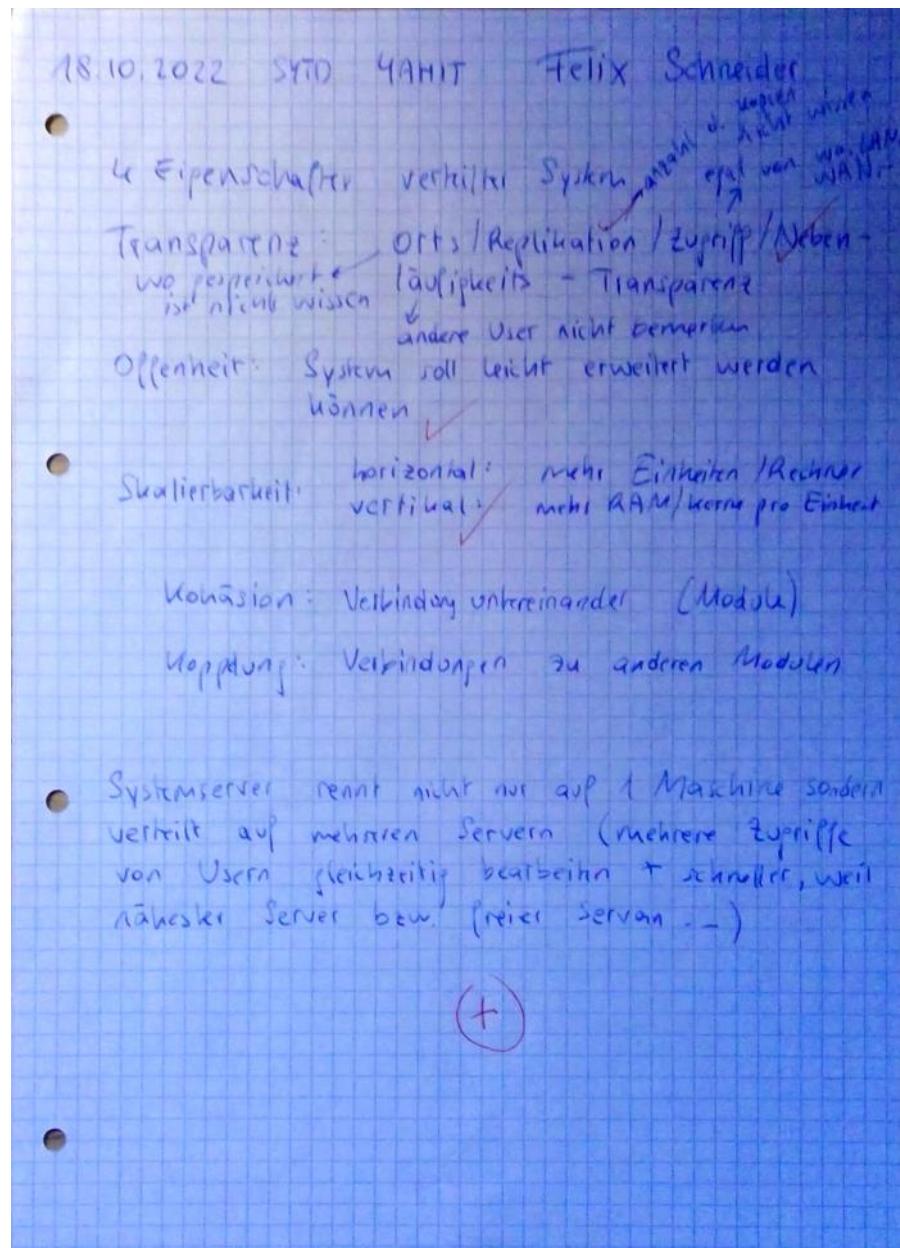
Parameter

GET Informationen

→ unsichtbare POST Informationen

SMÜ, am 18.10.2022

Dienstag, 6. September 2022 09:56



Test, am 08.11.2022

Dienstag, 29. November 2022 09:46

Name: Felix Schneider	SYTD Test	8.11.2022
<p>1. Erklären Sie den Begriff „Service-oriented Architecture“. (1 Punkte) (= z.B.: Microservice-Archi.) Eine Architektur, wo es mehrere Module gibt, die eine spezifische Aufgabe haben (→ Service-orientiert). Jeder Service / Module kann mit Schnittstellen mit anderen Services interagieren.</p> <p>↳ besteht aus mehreren voneinander unabhängigen Komponenten, die miteinander kommunizieren</p> <p>2. Erklären Sie, warum die Microservice-Architektur gegenüber der monolithischen eine höhere Wartbarkeit besitzt. (2 Punkte) veraltete Services (ältere Programmiersprache, ...) können unabhängig vom gesamten System in anderen Programmiersprachen programmiert werden. Außerdem können einzelne Services ausgetauscht werden, ohne, dass das gesamte System läuft.</p> <p>3. Beschreiben Sie den Begriff „Koppelung“. (1 Punkt) Die Verbindungen zwischen den Modulen. Je weniger Koppelung, desto besser, weil dann nur 1 definierte Schnittstelle verwendet wird.</p> <p>4. Erläutern Sie drei Arten von Transparenz in verteilten Systemen. (6 Punkte)</p> <p>Nebenläufigkeitstransparenz: Ein Nutzer weiß nicht, wenn viele andere Nutzer auf den Service zugreifen.</p> <p>Ortstransparenz: Die Nutzer wissen nicht, wo ihre Daten physisch gespeichert sind.</p> <p>Replikationstransparenz: Der Nutzer weiß nicht, wie viele Kopien seiner Daten vorhanden sind. Alle Kopien müssen allerdings gleichen Namen haben. (setzt Ortstransparenz voraus)</p> <p>+ Zugriffstransparenz: Nutzer kann von überall drauf zugreifen. (LAN, WAN, ...)</p>		

Felix Schneider

5. Beschreiben Sie den Begriff „Kohäsion“. (1 Punkt) Ich beschreibe lieber Kohäsion:

Jedes Modul hat eine spezifische Aufgabe.
Diese Aufgaben sind untereinander verbunden.

Mehr Kohäsion und lose Koppelung sind optimal.

6. Nennen und beschreiben Sie drei Vorteile der Microservice-Architektur gegenüber der monolithischen. (6 Punkte)

Offenheit: bei 7. beschreiben

Skalierbarkeit: einzelne Services können nach Last / Zugriff skaliert werden

Agiles Projektmanagement: gleichzeitiges Arbeiten
(Iteration Prozess)

hohe Wartbarkeit: bei 2. beschreiben

Robustheit: Wenn 1 Service ausfällt, fällt nicht alles aus

7. Was bedeutet Offenheit im Bezug auf verteilte Systeme? (2 Punkte)
einzelne Services können in unterschiedlichen Programmiersprachen geschrieben werden, solange die vordefinierten Schnittstellen einheitlich und unverändert bleiben.

Die Services können auch unabhängig voneinander erweitert werden.

8. Ist keine Koppelung anstrebenswert? Begründen Sie Ihre Antwort. (1 Punkt)

Auf keinen Fall, denn dann wäre theoretisch geschen jeder Dienst ein Monolith, da dieser keine anderen Services weiterverwenden kann (keine Koppelung).
Jeder Dienst müsste alle Funktionalitäten selbst **Sehr Gut** aus implementieren. (golden Mitte: lose Koppelung)

1: 19-20
2: 17-18
3: 13-16
4: 10-12
5: 0-9

Ergebnis: 20 / 20

Überblick

Montag, 28. August 2023 10:07

Cloud Computing

- viele Services in Cloud umsetzen (programmieren)
- Kopplung / Kohäsion
- Deployment / Monitoring

IaaS PaaS SaaS

Freitag, 8. September 2023 16:07

Arbeit geringer / aber teurer



as a Service

verschiedene
Cloudanbieter

The diagram illustrates three cloud service models arranged horizontally. The first model, 'alles selber', is represented by the text 'alles selber' with an upward arrow pointing to it from below. The second model, 'VM, VPS', is represented by the text 'VM, VPS' with an upward arrow pointing to it from below. The third model, 'Webserver', is represented by the text 'Webserver' with an upward arrow pointing to it from below.

z.B.: Ubuntu deployn z.B.: php-Server kaufen,

Linux mit .NET

Runtime

mit

Webserver

(meistens)

Azure)

Pertige Software

z.B.: Office 365

Cloud Computing

Dienstag, 12. September 2023 15:02

Vorteile:

- billiger für private Unternehmen
 - leichter skalierbar (rauf und runter)
 - das zahlen, was man braucht (Kostentransparenz)
 - niedriger Einstiegspreis
 - hohe Erreichbarkeit / Verfügbarkeit
 - ↳ SLA: Service Level Agreement
- < 99.95% uptime → 10% discount
< 99% uptime → 25% discount
< 95% uptime → 100% discount

Big Player

AWS ~ 32%

Azure ~ 22%

Google ~ 11%

Compute resources

Compute resources

Services, welche etwas berechnen

Costs: ab Moment, wo Ressource existiert

Examples:

WebServer, Serverless on demand computing,
Virtual Machine, Docker images, Docker containers

Storage resources

Services, welche Daten speichern

Cost: for time and storage

Examples:

Storage Account, Cosmos DB, Data Lake Storage

Mixed Resources

Services, berechnen und speichern

Cost:

Examples:

SQL Server (virtuelle Kerne, Speicher)

REDUNDANCY

LOCALLY REDUNDANT STORAGE

3x am gleichen physischen Standort
(synchron; gleiche Daten) \hookrightarrow billig

(synchron; gleiche Daten) \hookrightarrow billig

ZONE-REDUNDANT STORAGE

3x in gleicher Zone
(synchron) \hookrightarrow high availability

GEO-REDUNDANT STORAGE

3x am gleichen physischen Standort
1x asynchron in anderer Zone
(hundreds of miles away)

99.999999999999% 16 9's

Reliable Messaging

Dienstag, 19. September 2023 14:54

- Eventually all messages should be processed by receiving service
- high guarantee

Possible problems

- Message gets lost
- read, but not deleted
- deleted, but not processed

Solution

- wait for success response
- store message permanently

Examples Message Brokers

- RabbitMQ
- Apache Kafka
- Azure Service Bus

Protocols

- AMQP: Advanced Message Queuing Protocol
- MQTT

Topic: alle Subscriber bekommen die Nachricht

Queue: 1 bekommt Nachricht

Dead Letter Queue

→ messages that cannot be delivered end up in

- messages that cannot be delivered end up in dead letter queue
- message broker tries to redeliver few times before
- DLQ: can check for errors

Why not HTTP?

- couple services together

Advantages message brokers

- services loosely coupled
- messages act like buffer
- task distribution (producer/consumer pattern)

Advanced Features

- Structur (give namespaces)
- message sessions (guarantee delivery by using request-response access)
- routing
- scheduled delivery
- filters
- duplication detection
- secure access (roles / permissions)
- geo-disaster recovery

Comparison

Service Bus comes in Basic, standard, and premium tiers. Here's how they compare:			
Feature	Basic	Standard	Premium
Queues	✓	✓	✓
Scheduled messages	✓	✓	✓
Topics		✓	✓
Transactions		✓	✓
De-duplication		✓	✓
Sessions		✓	✓
ForwardTo/SendVia		✓	✓
Message Size	256 KB	256 KB	100 MB
Resource isolation			✓
Geo-Disaster Recovery (Geo-DR)			✓ <small>*Requires additional Service Bus Premium namespaces in another region.</small>
Java Messaging Service (JMS) 2.0 Support			✓
Availability Zones (AZ) support			✓

Serverless Computing

Dienstag, 19. September 2023 15:04

Serverless Computing

- Ausführung in der Cloud
- Server Management hidden from Developer
- allocate resources on demand

Azure Functions → Azure Serverless

↳ innerhalb einer Function App

Example (uses cron expression / Acrontab)

```
[Function("RecurringEvent")]
public static string Run(
    [TimerTrigger("*/10 * * * *")] MyInfo myTimer,
    FunctionContext context)
{
    var logger = context.GetLogger("RecurringEvent");
    logger.LogInformation(
        $"Function executed at: {DateTime.Now}");
}
```

Structor

Azure Functions = Methode / Function

AF braucht Trigger / Input Binding

AF kann Output Binding haben

Platforms for AF

.NET

Node.js

Python

Java

Powershell Core

Triggers

Time, HTTP, Queue, Topic, Blob,
CosmosDB, Event Grid, Event Hub, ...

Hosting

- Consumption (Serverless)
- Functions Premium
 - ↳ running continuously (not Serverless)
- App Service Plan
 - ↳ AF Ausführung beschränkt auf 10min
 - ↳ Methode muss innerhalb ~2h fertig werden

.NET -execution models

- Worker Process
 - ↳ Function & Host laufen mit gleicher .NET v
 - ↳ only LTS versions
 - ↳ obsolete in future

↳ obsolete in future

· Isolated

↳ isolated process

[↳ version independent]

↳ flexible

↳ some bindings not supported

IDE

Rider / VS Code

Function Outputs

→ JSON Object

→ HTTP Trigger

→ otherwise NuGet-Packages

Output Bindings

→ with Attributes

→ or Parameter

→ or ... (just look at docs)

Configurations

↳ UseDevelopmentStorage = true

↳ ... Connection String für Local Development

→ Use Development String

↪ Alias für Connection String für Local Development

Durable Functions

Dienstag, 26. September 2023 15:18

→ Erweiterung Azure Functions

→ AF: stricter stateless Aufbau
(Ausführungen unabhängig)

→ AF limitiert: 10min Zeitig müssen → no complex workflows
(long-running operations)

Execution time limitations → exec. of single AF

	Default	Max
Consumption	5min	10min
Premium	30min	Unlimited (60min guarantee)
Dedicated	30min	Unlimited (60min guarantee)

→ no AF - interaction

→ SOLUTION: DURABLE
FUNCTIONS

→ can interact with each other

→ stateful execution

→ waiting for human inputs

Orchestrator

↳ one function which orchestrates
/call other functions (with return)

→ long running

→ lifespan: milliseconds to never-ending

→ Always start from the beginning

Activity

→ called by orchestrator

→ actual work / logic

External Event

→ wait for event from external client

→ bindings / HTTP call

Function Patterns

↳ Function Chaining

→ results from one activity are inputs for next activity

→ easier to overlook than normal AF trigger queues
(big picture is described by orchestrator)

- ↳ Fan out / Fan in
 - parallel execution and waiting for all functions to finish
- ↳ Async HTTP APIs
- ↳ Monitoring
- ↳ Human interactions
- ↳ Aggregators

Test, am 12.01.2024

Montag, 28. August 2023 10:07



Test 2 aus SYTD

12.01.2024

SAHIT

Name: Felix Schneider 26 P

(1)

Ausgangssituation:

Sie sind Teil eines Softwareentwicklungs-Teams, das ein Ticketverkaufsservice entwickelt und betreibt. Kunden können über Ihr Service Tickets für verschiedene Events (Konzerte, Theater, Festivals, ...) kaufen.

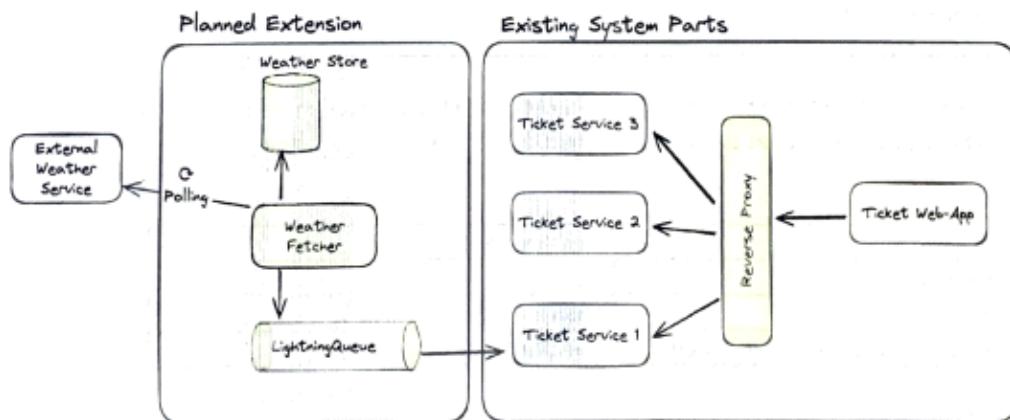
Ihr Team beschließt ein neues Feature zu implementieren. Für Open-Air Events, die unter freiem Himmel stattfinden, soll eine Wettervorhersage integriert werden, das den Kunden auf der Verkaufsseite anzeigen, wie wahrscheinlich mit gutem Wetter zu rechnen ist, oder ob es besser ist eine Regenjacke einzupacken.

Sie haben auch schon einen Anbieter von Wetterdaten ausgewählt, der besonders zuverlässige Daten liefert. Leider bietet dieser Anbieter seine Wetterdaten nur über eine REST-API an. Sie würden aber gerne ein regelmäßiges Abfragen dieses Services vermeiden, da Performanz in Ihrem Bereich besonders wichtig ist, besonders dann, wenn viele Kunden gleichzeitig begehrte Tickets kaufen möchten. Stattdessen würden sie einen Ansatz bevorzugen, bei dem Ihre Wetterdaten nur dann aktualisiert werden, wenn sich etwas an der Wettervorhersage, bei relevanten Standorten, an denen Events stattfinden, ändert.

Sie entscheiden sich für folgende Lösung:

Sie entwickeln ein Service (Weather Fetcher) das regelmäßig die Wetterdaten von dem ausgewählten externen Wetter-Service abfragt. Dann vergleichen sie die Daten mit bereits vorhanden Daten in einem Weather Store, und falls Änderungen vorhanden sind, leiten sie die neuen Wetterdaten an Ihr System weiter, indem Sie es in eine Message Queue stellen (sie wollen dafür ein Service namens Lightning Queue verwenden), von wo aus es für Ihr Ticket Service bezogen und angezeigt werden kann.

Glücklicherweise verwenden Sie bereits eine Micro-Service Architektur mit mehreren Docker-Container, die diese Erweiterung einfach möglich macht. Die Architektur der geplanten Erweiterung ist in diesem Diagramm dargestellt.



Gehen Sie der Reihe nach vor, um die neue Funktionalität zu entwickeln.

Seite 1 von 5

Test 2 aus SYTD

12.01.2024

SAHIT

1) Erweitern des Docker-Compose Files

Sie nutzen derzeit folgendes docker-compose.yaml File um Ihr Gesamtsystem lokal zu starten und weiterzuentwickeln.

6 Punkte

3

```
services:  
  tickerservice1:  
    build:  
      context: ./ticketservice1  
      dockerfile: ./Dockerfile  
    ports:  
      - "8080:8080"  
  tickerservice2:  
    build:  
      context: ./ticketservice2  
      dockerfile: ./Dockerfile  
    ports:  
      - "8081:8080"  
  tickerservice3:  
    build:  
      context: ./ticketservice3  
      dockerfile: ./Dockerfile  
    ports:  
      - "8082:8080"  
  reverseproxy:  
    image: "awesome-proxy:4.2.1337"  
    ports:  
      - "8083:8080"  
    depends_on:  
      - tickerservice1  
      - tickerservice2  
      - tickerservice3  
  webapp:  
    build:  
      context: ./webapp  
      dockerfile: ./Dockerfile  
    ports:  
      - "80:8080"  
    depends_on:  
      - reverseproxy
```

Sowohl für den Weather Store als auch für die Message Queue, greifen Sie auf bereits bestehende Docker Images zurück, die auf hub.docker.com verfügbar sind.

Weather Store:

Image Name:	weatherstore
Tag Name:	3.12

Weather Store:

Image Name:	weatherstore
Tag Name:	3.12
Default Port:	5698
Pfad für persistierte Wetterdaten:	/weatherdata

Message Queue:

Image Name:	lightning-queue
Tag Name:	12.7
Default Port:	5698
Pfad für persistierte Nachrichten:	/messages

Geben Sie an, welche Erweiterungen am docker-compose File nötig sind, um diese beiden Services in Ihr System aufzunehmen. Für jegliche Daten, die persistent gespeichert werden, sollen Named Volumes verwendet werden.

Seite 2 von 5

Test 2 aus SYTD

12.01.2024

Felix Schneider

SAHIT

weather-store:
image: weatherstore:3.12
container_name: weatherstore
ports:
- "8083:5698"
volumes:
- weatherdata:/weatherdata

→ Seite 6 →

2) Erstellen des Dockerfile für das Weather Fetcher Service

7 Punkte

Sie erstellen eine C# Konsolenapplikation, um das Weather Fetcher zu implementieren.
Erstellen Sie dafür eine Dockerfile mit dem Sie in weiterer Folge ein Image erstellen können.

7

Sie erinnern sich, dass folgende Schritte nötig sind:

1. Basieren Sie Ihr Image auf folgendem Basis-Image:

mcr.microsoft.com/dotnet/sdk:8.0

2. Arbeiten Sie im Image-Dateisystem unter dem Pfad /src um die Anwendung zu kompilieren.

3. Kopieren Sie alle Dateien aus dem aktuellen Verzeichnis des Host Systems in das Arbeitsverzeichnis des Image-Dateisystems.

4. Verwenden Sie das Kommando `dotnet restore WeatherFetcher.csproj` um die NuGet Packages für das Projekt zu laden.

5. Verwenden Sie das Kommando `dotnet publish WeatherFetcher.csproj -o /app` um die Anwendung zu kompilieren.

6. Setzen Sie das Arbeitsverzeichnis nun auf das Verzeichnis /app in dem die kompilierte App vorhanden ist.

weil nicht multistaging optional

5. Verwenden Sie das Kommando `dotnet build` um die Anwendung zu komplizieren.
 6. Setzen Sie das Arbeitsverzeichnis nun auf das Verzeichnis `/app` in dem die kompilierte App vorhanden ist.
 7. Geben Sie als Einsprungpunkt für das Image das Kommando `dotnet WeatherFetcher.dll` an.

weil ~~multistaging~~
~~optional~~

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS base
WORKDIR /src
COPY . .
RUN dotnet restore WeatherFetcher.csproj
RUN dotnet publish WeatherFetcher.csproj -o /app
WORKDIR /app
CMD ["dotnet", "WeatherFetcher.dll"]
ENTRYPOINT
```

Seite 3 von 5

Test 2 aus SYTD

12.01.2024

SAHIT

3) Implementierung der WeatherFetcher Konsolenanwendung

6 Punkte

Schreiben Sie eine C# Konsolenanwendung, die regelmäßig, alle 30 Sekunden, Daten vom externen Service abfragt, überprüft, ob diese neuer sind, als die aktuell im WeatherStore vorhandenen und falls ja, die Daten im WeatherStore aktualisiert und auch an die Message Queue weiterleitet. Sie können folgende bereits vorhandene Methoden und Klassen verwenden:

b

```
async Task<WeatherReport> FetchWeatherDataFromExternalServiceAsync()
```

```
async Task<DateTime> GetLatestTimestampFromWeatherStoreAsync(
    string host, string port)
```

```
async Task InsertToWeatherStoreAsync(
    string host, string port, WeatherReport report)
```

```
async Task InsertToLightningQueueAsync(
    string host, string port, WeatherReport report)
```

```
class WeatherReport {
    public DateTime GeneratedAt { get; set; }
    // Additional Weather Data
    // ...
}
```

while (true) {

var report = FetchWeatherDataFromExternalServiceAsync();

var generatedAtNew = GetLatestTimestampFromWeatherStoreAsync();

if ("weatherstore", 8084);

5698

IP ("weatherstore", "5698")

```
if (report.GeneratedAt != generatedAtNew) {
```

InsertToWeatherStoreAsync()

"weatherstore", ~~8084~~, "5698", report);

Insert To Lightning Queue Async (

"lightning-queue", ~~808T~~, report);
"5698"

3

~~Three Thread. Sleep (30 000);~~

3

Seite 4 von 5

Test 2 aus SYTD

12.01.2024

SAHIT

Felix Schaeide

卷二十一 三

4) Testen Sie Ihr WeatherFetcher Service
Sie möchten Ihr entwickeltes Service testen, ohne dabei das Gesamtsystem über docker-compose zu starten. Schreiben Sie die Docker CLI Befehle, zum Erstellen eines Images unter der Verwendung des Dockerfiles aus Schritt 2, sowie zum Starten eines Containers, der auf dem Image basiert.

Der Container soll beim Starten im Hintergrund (**detached**) laufen und nach dem Stoppen automatisch schließen.

Die Container für WeatherStore und Lightning-Queue laufen bereits und wurden zu einem Docker Netzwerk mit dem Namen `weatherfetchnetwork` hinzugefügt.

```
docker build -t weatherfetcher
docker run -d --rm --network weatherfetchnetwork
weatherfetcher
```

5) Erweiterung des Docker Compose File

Erweitern Sie nun das docker-compose.yaml File und fügen Sie Ihr neu erstelltes WeatherFetcher Service hinzu. Verweisen Sie nicht auf das bereits erstellte Image, sondern auf das Dockerfile aus dem das Image erstellt werden soll.

weather pecker

build: ~~short~~: /weather fetcher/

weatherfetcher:

build:

context: ./weatherfetcher

Dockerfile: ./Dockerfile

name: weatherfetcher

Kdependson

(im Netzwerk sind sie automatisch, weil docker compose file)

6) Starten Sie das fertige Gesamtsystem

Verwenden Sie docker-compose um das Gesamtsystem zu starten. Geben Sie den Befehl dafür an. Die Services sollen im Hintergrund (detached) gestartet werden.

2 Punkte

2

docker compose up -d

Für rebuild von ersten 5 Images:
(docker-compose up -d --build)

Seite 5 von 5



1) lightning-queue:

image: lightning-queue: 12.7

container_name: lightning-queue

ports:

- "8085": 5698

volumes:

- lightning-queue: /messages

- lightning-queue : /messages

ff volumes