Westfälische Hochschule

Programmierung eines Onlineshops mit Microservice Architektur

Philipp Honsel, Lennart Kampshoff

Version vom 24. Juli 2022

Cloud Grundlagen und Programmierung (SS 2022)

Prof. Dr. Raphael Herding

Inhaltsverzeichnis

Glossar

DTO	Data Transfer Object, Datenmodell zum Übertragen von Daten
JWT	Json Web Token, eine Methode Daten für die Authentifizierung bereit zu stellen
REST	REpresentational State Transfer, ein Standard zur Übermittlung von Objekten
DSP	Desired State Principal

1 Einleitung

In der heutigen Zeit wird das Einkaufen im Internet immer relevanter. Gerade durch die Corona-Pandemie hat sich der Umsatz von *E-Commerce Shops* stark nach oben entwickelt. [1]

Vor allem bei einem Onlineshop ist es sehr wichtig, dass dieser jeder Zeit verfügbar ist. Jeder Ausfall des Shops kann nicht nur unmittelbar den Wegfall des Umsatzes bedingen, da nichts bestellt werden kann, sondern auch die Kundenbindung nachhaltig schwächen. Hat ein Kunde einmal schlechte Erfahrung mit einem Geschäft gemacht, ist die Wahrscheinlichkeit, dass er erneut den Onlineshop besucht, geringer. Zudem hat Greg Linden bereits im Jahr 2006 ein Experiment durchgeführt, bei dem die Ladezeiten vom Amazon künstlich in Inkrementen von 100 Millisekunden erhöht wurden. Dabei hat Amazon jedes Inkrement circa 1% an Umsatz gekostet. [2]

Aus diesen Gründen ist das Ziel dieser Projektarbeit einen hoch verfügbaren Onlineshop zu entwickeln und diesen im Nachgang auch mithilfe moderner Lösungen möglichst ausfallsicher bereitzustellen.

Zum Erreichen dieser Ziele wurden zeitgemäße Technologien genutzt. Darunter *ASP.NET Core* für die Services im Backend, *Angular* für das Frontend, *Docker* als Containerisierungslösung und *Kubernetes* für die Container-Orchestrierung.

2 Anforderungen

Im folgenden werden kurz die wichtigsten Anforderungen an das Shopsystem zusammengefasst. [3]

2.1 Funktionale Anforderungen

Der Onlineshop soll in drei große Hauptbereiche aufgeteilt werden: die Startseite, den Warenkorb und die Bestellbestätigungsseite.

Auf der Startseite werden alle Produkte aufgelistet. Diese können per Knopfdruck in den Warenkorb gelegt werden. Produkte, die bereits in dem Warenkorb liegen, werden auf der Startseite nicht mehr angezeigt, da von jedem Produkt nur ein Exemplar auf einmal bestellt werden kann.

Im Warenkorb werden nicht nur alle gewählten Gegenstände angezeigt, sondern auch die Versandkosten für die aktuelle Bestellung. Dazu gibt es ein Formular für die Kontakt- und Zahlungsinformationen.

Nach der Bestellung gelangt der Benutzer auf die Bestellbestätigungsseite. Auf dieser werden die gekauften Produkte angezeigt. Zudem eine Zusammenfassung seiner Informationen und die Lieferungsdaten.

2.2 Nicht-funktionale Anforderungen

Die Kommunikation zwischen den einzelnen Services muss über *REST* geschehen. Die URL's der Services dürfen nicht hart im Quelltext stehen, sondern müssen in Kubernetes konfigurierbar sein. Alle Endpunkte müssen dokumentiert werden.

Das System muss auf Knopfdruck gestartet werden können. Eine automatische *Horizontale Skalierung* muss implementiert und getestet werden. Dazu soll *Kubernetes* und *Docker* genutzt werden.

3 Backend Architektur

Vor und während der Entwicklung des Shopsystems wurden einige Architekturentscheidungen getroffen. Eine Auswahl dieser werden im folgenden erläutert.

3.1 Komponentenarchitektur

3.1.1 Speicherung und Zuordnung des Warenkorbs

Jeder Warenkorb muss einem Benutzer zugeordnet werden können. Dazu muss entweder bei jeder Abfrage ein *Sessiontoken* mit an das Backend gesendet werden, oder das Cart im Frontend gespeichert werden. In diesem Onlineshop wird allerdings eine Mischung der beiden Ansätze genutzt.

Im Frontend wird die Id des Warenkorbs gespeichert, während im Backend die zugehörigen Produkte gespeichert werden. Die Vorteile von diesem Ansatz sind zum einen, dass es keine Konflikte mit dem *Sessiontoken* in der Datenbank gibt. Zum anderen müssen nicht bei jeder Änderung des Warenkorb alle Informationen an das Backend geschickt werden, sondern nur solche die sich geändert haben.

Nachdem der Benutzer seinen Warenkorb bestellt hat, wird dieser nicht gelöscht, sondern deaktiviert. Deaktivierte Warenkörbe können weder bearbeitet noch erneut bestellt werden. So muss der Checkout Service nicht nachhalten, welche Produkte in einer Bestellung bestellt wurden. Diese Informationen liegen weiterhin im Cart Service.

3.1.2 Generisches Antwort DTO

Jeder Service antwortet im gleichen Schema. Dieses wurde darauf ausgelegt, dass andere Backend Services und das Frontend zuverlässig erkennen können, ob die Anfrage erfolgreich war. Dazu wird der Inhalt einer jeden Antwort in das *ResponseModel DTO* (siehe Code-Ausschnitt 1) verpackt.

```
1 {
2     "success": boolean,
```

```
3     "data": object,
4     "error": string[]
5 }
```

Code-Ausschnitt 1: ResponseModel DTO

Neben dem HTTP Statuscode kann so der Empfänger dieser Antwort direkt am *success*-Feld erkennen, ob die Operation erfolgreich war. Wenn dieses Feld *true* ist, ist das *data*-Feld gefüllt. Sollte die Antwort *false* sein, ist das *error*-Feld mit einer Liste an Fehlern gefüllt.

3.1.3 Behandlung von Fehlern

Sollte während der Bearbeitung einer Anfrage ein Fehler auftreten, muss nicht nur der Benutzer informiert werden, sondern im besten Fall auch die Transaktionssicherheit gewahrt werden.

Dazu wird in jedem Service für den aktuellen *Scope*[1] ein *NotificationHandler* angelegt. Dieser hält eine Liste von Fehlermeldungen vor. Sollte während der Abarbeitung der Anfrage ein Fehler auftreten, wird eine Fehlernachricht an diese Liste angehangen und die Methode abgebrochen. Daraufhin wird automatisch das *ResponseModel* so angepasst, dass dieser Umstand an den aufrufenden Service weitergegeben wird.

In aufrufende Richtung wird somit zumindest die Atomarität der Transaktion gewahrt, da es nicht vorkommen kann, dass der aufrufende Service weiterarbeitet, wenn in dem aufgerufenem Service ein Fehler aufgetreten ist.

3.1.4 Validierung von Eingaben

Die Eingaben der Benutzers werden im Backend auf Vollständigkeit und semantische Korrektheit geprüft. Dazu werden *Validatoren* mithilfe des *FluentValidation Pakets* entwickelt. Dabei werden Regeln festgelegt und welcher Fehler bei einem Verstoß geworfen werden soll.

```
1 RuleFor(c => c.Email)
2     .NotEmpty()
3     .WithMessage("E-Mail darf nicht leer sein")
4     .EmailAddress()
```

[1] Ein Scope ist der Aktionsbereich eines Http Contexts, und damit einer Http Anfrage.

```
5       .WithMessage("E-Mail muss im E-Mail Format sein");
```

Code-Ausschnitt 2: Beispiel von FluentValidation anhand einer Email

In Code-Ausschnitt 2 ist ein Beispiel für die Überprüfung von E-Mail Adressen zu sehen. In Zeile 1 wird festgelegt für welches Feld des *DTO* die Regeln festgelegt werden. Darauf folgen die Aufrufe `.NotEmpty()` und `.EmailAddress()`. Dies sind von dem Paket implementierte Methoden und validieren, wie es der Name vermuten lässt, dass das Feld weder leer, noch in einem fehlerhaften E-Mail Format ist. Sollte die vorhergegangene Methode nicht erfolgreich sein, wird der folgende Fehler `.WithMessage("...")` gespeichert und die Validierung als ganze als fehlgeschlagen markiert. Dann wird auch die aufrufende Methode abgebrochen und ein Fehler auf den *NotificationHandler* (siehe Unterunterabschnitt 3.1.3) gelegt.

3.1.5 Sicherheit

Um einzelne Funktionen oder ganze Endpunkte abzusichern, authentifizieren sich die Backend Services untereinander mit *Json Web Token (JWT)*.

Ein *JWT* ist im Prinzip drei in *Base64* kodierte *JSON* Objekte, welche durch einen Punkt getrennt sind. Dabei gibt das erste Objekt, der *Header*, allgemeine Informationen zu diesem Token an, wie den Algorithmus, der zum verschlüsseln genutzt wurde. Das zweite Objekt, die *Payload*, bietet sogenannte *Claims*. In diesen stehen beispielsweise *User Ids*, *Ablaufdatum des Tokens* oder in unserem Fall der Systemname des Aufrufers. Ziel des Tokens ist es nicht diese Informationen zu verstecken, sondern die Integrität dieser Daten zu beweisen. Dazu wird der letzte String genutzt, die *Signatur*. In der *Signatur* werden die Daten aus dem *Header* und der *Payload* zusammengefasst und verschlüsselt. Solange der Empfänger Zugriff auf das *Secret* hat, welches zum verschlüsseln genutzt wurde, kann er die Integrität der Daten prüfen. [4]

Und genau das machen die Services, wenn der Zugriff geschützt werden muss. Bei dem Starten eines Backend Services wird einmalig ein Token generiert, welcher der einfachheithalber kein Ablaufdatum hat. Zudem wird eine Liste an Services hinterlegt. Nur diese haben dann die Rechte auf gesicherte Methoden zuzugreifen. Bei einem Aufruf einer gesicherten Methode wird der *Claim forService* entnommen und geprüft, ob dieser in der Liste der validen Systeme steht. Ist das nicht der Fall, wird die Bearbeitung der Anfrage abgebrochen.

3.2 Verfügbare Endpunkte

Im folgenden werden die Endpunkte aller Services kurz inhaltlich beschrieben. Eine ausführliche und technische Dokumentation ist mittels *Swagger* und *Swashbuckle* erfolgt.

Ein *:text* in der Url steht für Variablen (in diesem Fall namens *text*), welche in der Url übergeben werden.

3.2.1 Catalog Service

GET /	Lädt alle verfügbaren Produkte aus dem Katalog auf einmal
GET /list	Lädt alle Produkte, welche in dem *productIds* Queryparameter übergeben wurden
GET /:id	Lädt genau ein Produkt, welches durch *id* identifiziert wird

Tabelle 1: Tablle aller Endpunkte im Catalog Service

3.2.2 Cart Service

GET /:id	Lädt die Details eines Warenkorbs, der durch *id* identifiziert wird
GET /:id/count	Lädt die Anzahl der Produkte in einem Warenkorbs, der durch *id* identifiziert wird
POST /	Erstellt einen neuen Warenkorb und gibt diesen zurück
PATCH /:cartId/:productId	Fügt ein Produkt mit *productId* zum Warenkorb (*cartId*) hinzu

DELETE /:cartId/:productId	Entfernt ein Produkt mit *productId* aus dem Warenkorb (*cartId*)
PATCH /:cartId	Aktualisiert den angegebenen Warenkorb (cartId), kann mehrere Produkte auf einmal hinzufügen oder löschen

Tabelle 2: Tablle aller Endpunkte im Cart Service

3.2.3 Shipping Service

GET /:id	Lädt die Details zu einer Lieferung, die mit *id* identifiziert wird
POST /	Erstellt eine neue Lieferung und gibt die Details zu dieser zurück
GET /estimate/:price	Gibt einen Lieferpreis für einen Warenkorbpreis (price) zurück, ohne eine Lieferung zu erstellen

Tabelle 3: Tablle aller Endpunkte im Shipping Service

3.2.4 Checkout Service

GET /:id	Lädt die Details zu einer Bestellung
POST /	Erstellt eine neue Bestellung anhand eines Warenkorbs
GET /shipping/:cartId	Gibt einen Lieferpreis für einen Warenkorb zurück

Tabelle 4: Tablle aller Endpunkte im Checkout Service

4 Frontend

Das Frontend wurde mithilfe von Angular 14 und dem *Eva Design System*[2] entwickelt. Mit *Nebular*[3] gibt es eine fertige Implementation von diesem, von dem nur noch die Komponenten genutzt werden. Die Struktur des Onlineshops ist fest vorgegeben worden. (siehe Unterabschnitt 2.1)

4.1 State Management

Um den aktuellen State der Angular Applikation nach zuhalten wurde *ngxs*[4] genutzt. Dies ist ein *State Management Paket*, welches dem *CQRS* Design Pattern (Command Query Responsibility Segregation) folgt. Bei *CQRS* geht es darum, die Zuständigkeiten von Methoden in Befehle (Commands) und Abfragen (Queries) aufzuteilen.

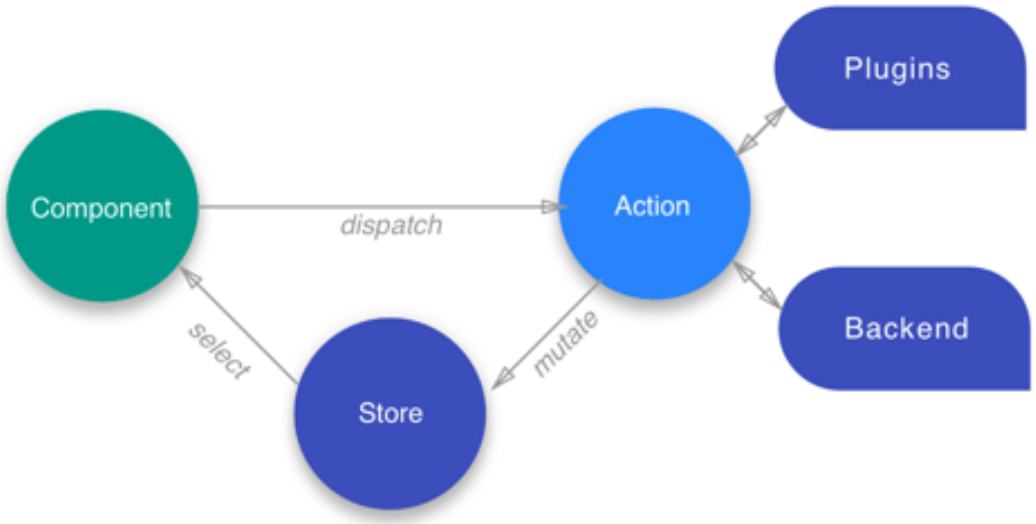

Abbildung 1: Konzept von NGXS [5]

In Abbildung 1 ist das Konzept von NGXS zu sehen. Dabei sendet in NGXS eine Angular Komponente eine Action (in *CQRS*: Command), welche dann von einem *Store* behandelt wird. Dieser kommuniziert dann mit dem Backend oder stellt eigene Berechnungen an. Mit dem

[2]https://eva.design/
[3]https://akveo.github.io/nebular/
[4]https://www.ngxs.io/

Ergebnis aus der Kommunikation kann dann eine Action als einziges Element in diesem Konstrukt den aktuellen *State* der Applikation verändern. Auf die Änderung dieses States hört wiederrum die Angular Komponente (in *CQRS*: Query), welche dann neue Dinge anzeigen kann.

In dem entwickelten Onlineshop wird nur der State des Warenkorbs nachgehalten. In diesem ist die Id der aktuellen Cart und eine Liste von Ids der ausgewählten Produkte. Die Details zu diesen Produkten werden nachgeladen, sobald diese benötigt werden.

4.2 Laufzeitvariablen für Deployment

Um das Frontend in unterschiedlichen *Staging Umgebungen* bereitstellen zu können, müssen die Url's zu den Backend Services konfigurierbar sein. Dies soll zur Laufzeit des Containers geschehen, um das Prinzip *Build Once, Deploy Many* nutzen zu können. Angular unterstützt standardmäßig keine Environment Konfiguration zur Laufzeit.

Allerdings gibt es dennoch eine Methode die Laufzeitvariablen anpassen zu können. Dabei wird in der *index.html* eine JavaScript Datei geladen, welche globale Variablen definiert. Die Variablen sind im Format von Linux Shell Parametern (siehe Code-Ausschnitt 3). Mit dem Tool *envsubst* werden dann beim Starten des Containers alle Parameter mit der angegebenen Environment Variable ersetzt.

```
1 (function (window) {
2     window["env"] = window["env"] || {};
3     window["env"].CATALOG_URL = '${ENV_CATALOG_URL}';
4     // ...
5 })(this)
```

Code-Ausschnitt 3: Laufzeit Environment

In der *environment.prod.ts* werden die Variablen abschließend nicht mehr über *hardcoded* Werte initialisiert, sondern über `window["env"].PARAMETER_NAME`. [6]

5 Kommunikation zwischen einzelnen Services

5.1 HTTP-REST Kommunikation zwischen den Microservices

Die Kommunikation der einzelnen Microservices mit dem Frontend und untereinader läuft mit HTTP-Anfragen ab. Zusätzlich kommt bei der Systemarchitektur der REST-Standard zum Einsatz. REST ist ein verbreiteter Standard zur konsistenten Übertragung von Objekten. Dieser arbeitet sehr eng mit HTTP zusammen, indem eine Mischung von Methoden, Pfaden, Parametern und Request-Payloads genutzt wird, um ein Objekt in einem Microservice zu verwalten Das Ergebnis der REST-Operation wird über HTTP-Statuscodes kommuniziert [7, 8, 9]. Mögliche Aktionen, welche auf ein Objekt angewendet werden können, sind in Tabelle 5 aufgelistet.

Aktion	**Methode**	**Pfad**	**Payload**
Abrufen	GET	/:id	Nein
Erstellen	POST	/	Ja
Aktualisieren (teilweise)	PATCH	/:id	Ja
Aktualisieren (komplett)	PUT	/:id	Ja
Löschen	DELETE	/:id	Optional

Tabelle 5: Auflistung aller REST-Aktionen

5.2 Sequenzdiagramme

5.2.1 Warenkorb befüllen

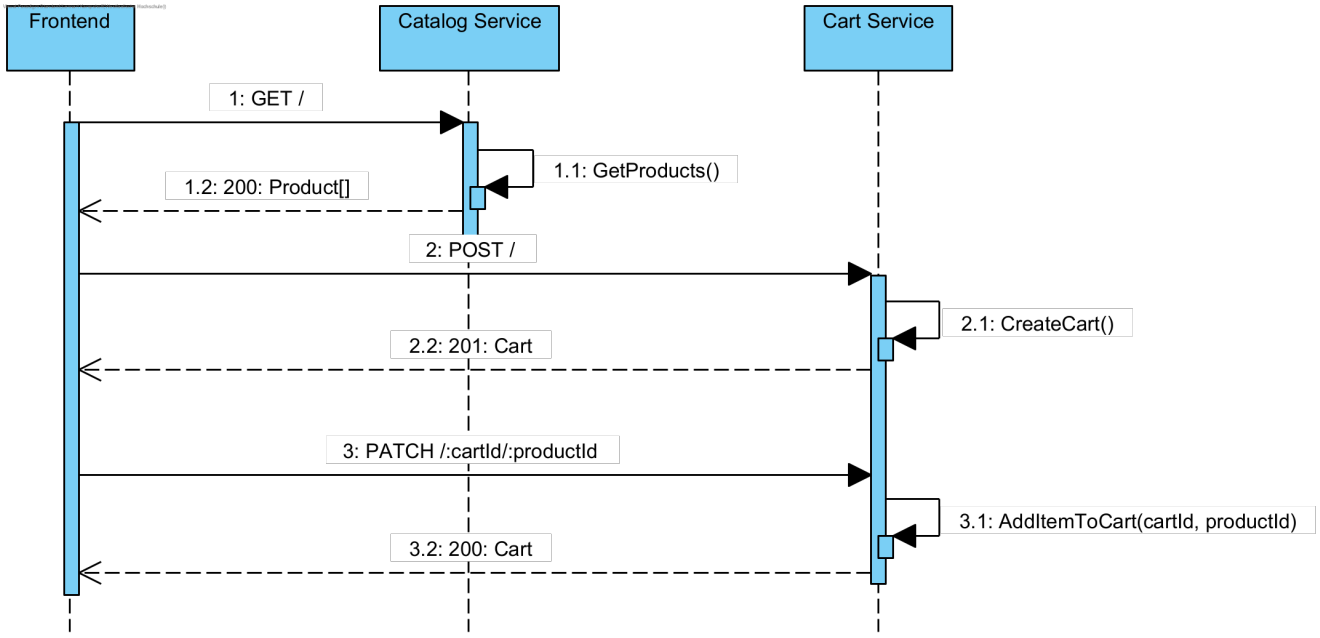

Abbildung 2: Sequenzdiagramm der Platzierung eines Produktes in den Warenkorb [eigene Darstellung]

In Abbildung 2 ist das Sequenzdiagram zu sehen, welches den HTTP-Anfragen-Fluss für die Einfügung eines Produktes in den Warenkorb darstellt. Um den Warenkorb zu befüllen, muss zuerst der komplette Produkt-Katalog per *HTTP GET /* vom *Catalog Service* geladen werden. Anschließend muss mit *HTTP POST /* beim *Cart Service* ein Warenkorb erstellt werden. Der Warenkorb kann per *HTTP PATCH /:cartId/:productId* gefüllt werden. Dabei entspricht der Parameter *cartId* der ID des zuvor erstellten Warenkorbs und der Parameter *productId* einer Produkt-ID des zuvor geladenen Produkt-Katalogs.

5.2.2 Bestellung

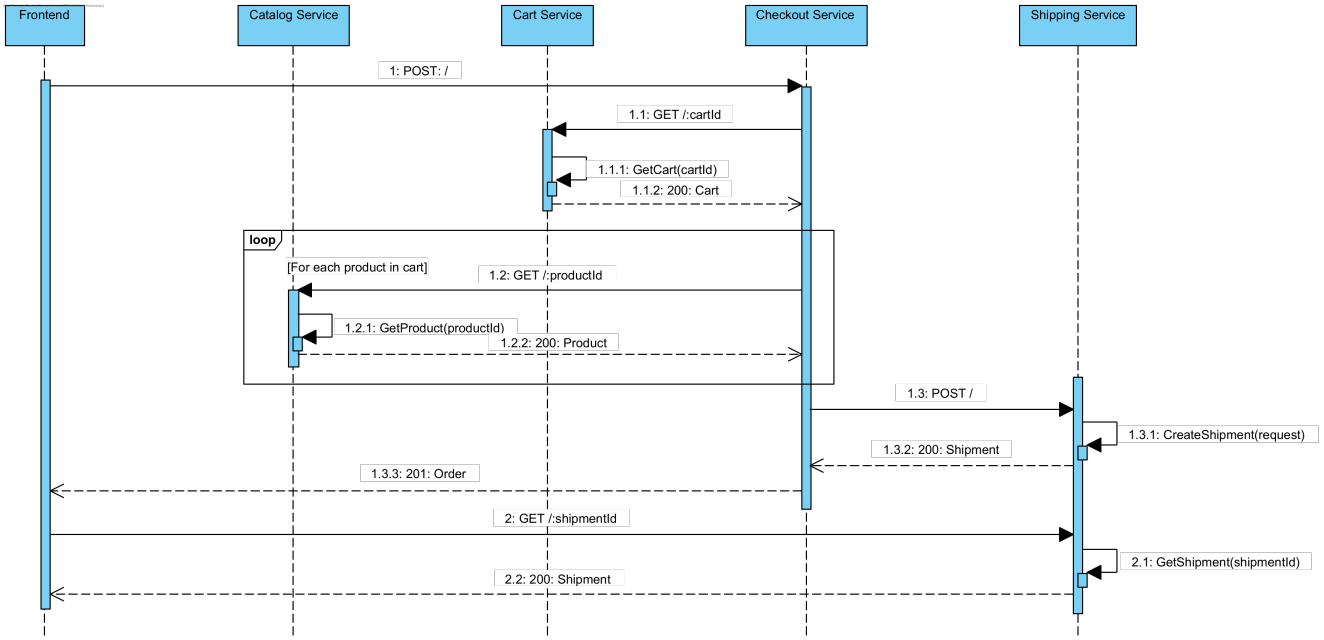

Abbildung 3: Sequenzdiagramm einer Bestellung [eigene Darstellung]

In Abbildung 3 ist das Sequenzdiagramm für eine erfolgreiche Bestellung zu sehen. Mit einem *HTTP POST /* an den *Checkout Service* kann eine Bestellung aufgegeben werden. Dabei werden die ID des Warenkorbs sowie alle Kundeninformationen im Body der HTTP-Anfrage übermittelt. Der *Checkout Service* erhält dann per *HTTP GET /:cartId* an den *Cart Service* alle Produkte im Warenkorb. Danach können die Produkinformationen per *HTTP GET /:productId* für jedes Produkt beim *Catalog Service* abgerufen werden. Nach Erhalt aller Informationen wird ein *HTTP POST /* beim *Shipping Service* aufgerufen, um eine Lieferung zu erstellen und die Versandkosten zu berechnen. Dabei werden die ID der Bestellung und der Gesamtpreis des Warenkorbs im Body der Anfrage übermittelt. Anschließend wird die gesamte Bestellung an das Frontend zurückgegeben.

6 Systemarchitektur

6.1 Microservices

Im Rahmen des Projektes sollten vier Microservices für den Onlineshop implementiert werden:

- Catalog Service, stellt einen Produktkatalog bereit
- Cart Service, verwaltet die Warenkörbe
- Checkout Service, verwaltet die Bestellungen
- Shipping Service, verwaltet die Lieferungen.

Dabei darf nicht jeder Service auf alle anderen Services zugreifen. Nur der Checkout Service und das Frontend können mit den anderen Services kommunizieren. Catalog, Cart und Shipping Service sind Sackgassen.

6.2 Bereitstellung der Dienste in Docker

Docker bietet die Containerisierung von Anwendungen. Das bedeutet, dass Docker eine Möglichkeit darstellt, eine konsistente Laufzeitumgebung für Anwendung bereitzustellen. So sorgt Docker zum Beispiel dafür, dass alle benötigten Abhängigkeiten einer Anwendung immer in der richtigen Version zusammen mit der Anwendung ausgeliefert werden können. Dabei ist es nicht notwendig, die Maschine, auf der die Anwendung läuft, zusätzlich zu konfigurieren. Zudem bietet Docker eine Kapselung von Containern, welche gleichzeitig auf einer Maschine laufen. Eine Anwendung in einem Container weiß nicht, dass sie in einem Container läuft. Dadurch sind die Anwendungen voneinander abgekapselt. Die Containerisierung ist also eine Abstraktion des Betriebssystems. Ein anderer Weg dieser Kapselung ist die Auslieferung der Anwendung in einer virtuellen Maschine. Dies hat jedoch den Nachteil, dass eine virtuelle Maschine pro Anwendung einen großen Overhead hat, da ein echtes Betriebssystem betrieben werden muss, anstatt nur einer Abstraktion dessen [10, 11].

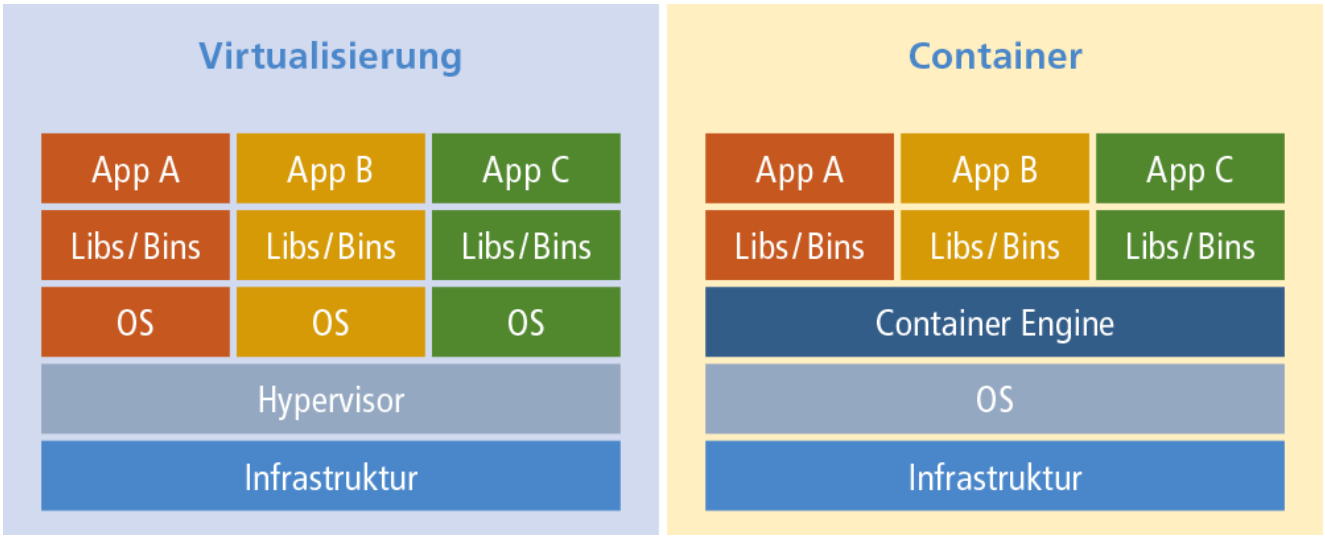

Abbildung 4: Gegenüberstellung von Docker und virtuellen Maschinen [11]

In Abbildung 4 werden die Unterschiede in der Architektur von Docker und einer VM dargestellt. Es ist zu sehen, dass bei einer oder mehrerer VMs ein Hypervisor sowie mehrere Betriebssysteme Systemresourcen verbrauchen. Bei Docker wird nur ein Betriebssystem und eine Container-Engine benötigt, was dazu führt, dass für einen zusätzlichen Docker-Container weniger Resourcen benötigt werden, als für eine neue virtuelle Maschine.

6.3 Hosting in Kubernetes

6.3.1 Kubernetes

Die Verwaltung der Docker-Container erfordert jedoch einen großen Aufwand. So muss jeder Container manuell gestartet oder gestoppt werden und im Fehlerfall muss der abgestürzte Container zuerst ermittelt und untersucht werden. Wenn das ganze System auf nur einer Maschine betrieben wird, ist der ganze Verwaltungsaufwand eventuell noch überschaubar. Sobald die Container für eine Anwendung auf mehrere Rechner verteilt werden sollen, wächst der Aufwand extrem schnell an. Für diesen Anwendungsfall wurde Kubernetes entwickelt. Kubernetes ist eine sogenannte Container-Orchestrierungs Software. Diese verwaltet die Container einer Anwendung. Zudem unterstützt Kubernetes auch eine Installation über mehrere Server hinweg, sodass das komplette Cluster von nur einer Master-Einheit aus gesteuert werden kann.

6.3.2 API-Objekte und Desired State Principle

Mit dem *Desired State Principal* (DSP), stellt Kubernetes sicher, dass immer der gewünschte Status der Software vorliegt. Bei der Konfiguration des Kubernetes Clusters wird vom Entwickler angegeben, wie das System aussehen soll. Dieses wird so von Kubernetes erstellt. Dazu können Entwickler auf eine Reihe von API-Objekten zurückgreifen, aus denen das komplette Cluster besteht. Die API-Objekte, die bei diesem Projekt zum Einsatz kommen sind:

- Deployment,
- Service,
- ConfigMap,
- Secret,
- Horizontal Pod Autoscaler.

Ein Deployment ist eine Definition eines oder mehrerer Docker-Container. Das Deployment definiert ein Template, nach welchem ein Pod (quasi ein Docker-Container) gestartet werden soll. In der Template Definition befinden sich alle Informationen, welche der Docker-Container benötigt. Dazu zählen Umgebungsvariablen, Port-Mappings und Resourcenlimits. Zusätzlich kann ein Deployment eine Anzahl an gewünschten Replicas des Pods festlegen. Eine Replica ist eine Instanz eines Docker-Containers. So können durch ein Deployment mehrere Pods mit der selben Konfiguration parallel betrieben werden. Durch das *DSP* sorgt Kubernetes dafür, dass immer die gewünschte Anzahl an Pods des Deployments gestartet sind.

Ein Service ermöglicht die Erreichbarkeit eines Deployments inner- und außerhalb des Kubernetes Clusters. Durch den Service werden die Ports des Deployments, die nur innerhalb des Pods erreichbar sind, dem Cluster bekannt gemacht.

ConfigMaps und Secrets dienen im Prinzip dem gleichen Zweck. Mit diesen API-Objekten können Daten zentral gespeichert und in gewünschte Deployments, etwa durch Umgebungsvariablen, eingefügt werden. Ein Secret ist jedoch für geheime Daten, wie zum Beispiel Passwörter oder Access Tokens, vorgesehen. ConfigMaps kommen bei allgemein bekannten Daten zum Einsatz.

Der Horizontal Pod Autoscaler bietet eine automatische Skalierung der Deployments. Durch eine Überwachung der Pods kann der Autoscaler eine bei einer Überschreitung der definierten Resourcen-Limits, etwa durch eine erhöhte Anzahl an Aufrufen auf einen Microservice, weitere Replicas eines Deployments bereitstellen.

6.3.3 Horizontal Pod Autoscaler

Um den *Horizontal Pod Autoscaler* nutzen zu können, muss zuerst die Erweiterung *Metrics Server* für Kubernetes installiert werden. Mit *Metrics Server* können die Metriken von einzelnen Pods ausgelesen werden. Anhand dieser Informationen rechnet der *Horizontal Pod Autoscaler* dann mithilfe der Formel `desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]` die benötigte Anzahl an Pods aus. [12]

Dafür werden in dem *API-Objekt* die Werte *minReplicas*, *maxReplicas* und *targetCPUUtilizationPercentage* konfiguriert. Letzteres gibt an wie hoch die Auslastung eines Pods maximal sein sollte, bevor ein weiterer Pod gestartet wird. Um Kosten zu sparen ist der Standardwert für die Anzahl an Pods pro Deployment nur ein Container. Diese können auf bis zu 5 Pods hoch skaliert werden. Die *Target CPU Percentage* beträgt 75%, sodass unerwartet komplizierte Aufgaben auch noch bearbeitet werden können.

Um Last auf das System zu simulieren wurde eine *Postman Collection* angelegt, welche wiederholt die Produkte abfragt, eine Cart erstellt, ein Item in diese legt und den Warenkorb abschließend bestellt. Vor dem Start des Tests waren alle Pods bei 0% CPU Auslastung. Zudem kann Abbildung 5 entnommen werden, dass von jedem Deployment nur ein Pod hochgefahren ist.

```
NAME                REFERENCE                          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
cart-service        Deployment/cart-service            0%/75%    1         5         1          82s
catalog-service     Deployment/catalog-service         0%/75%    1         5         1          82s
checkout-service    Deployment/checkout-service        1%/75%    1         5         1          82s
shipping-service    Deployment/shipping-service        1%/75%    1         5         1          82s
```

Abbildung 5: Horizontal Pod Autoscaler vor Lasttest

In Abbildung 6 sind die *Horizontal Pod Autoscaler* während der Last zu sehen. Hierbei fällt auf, dass der Cart Service mit 76% CPU Auslastung bereits einen zweiten Pod zugeteilt bekommen

hat, während der Checkout Service bei 73% noch immer nur einen Pod hat.

```
NAME               REFERENCE                      TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
cart-service       Deployment/cart-service        76%/75%   1         5         2          9m45s
catalog-service    Deployment/catalog-service     57%/75%   1         5         1          9m45s
checkout-service   Deployment/checkout-service    73%/75%   1         5         1          9m45s
shipping-service   Deployment/shipping-service    69%/75%   1         5         1          9m45s
```

Abbildung 6: Horizontal Pod Autoscaler während Lasttest

Nachdem die Lastspitze vorrüber ist, werden überschüssige Container nach 5 Minuten wieder heruntergefahren.

6.3.4 Microservice-Definition im Cluster

In diesem Projekt wird jeder Microservice mit einem Deployment und einem Service bereitgestellt. Die Verteilung von internen Service-URLs erfolgt über Config-Maps. Die Anmeldeinformationen für die Datenbanken werden über Kubernetes Secrets konfiguriert. Diese Informationen werden in den Deployments jeweils als Umgebungsvariablen bereitgestellt.

6.3.5 Beispielhaftes Deployment

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: seq-logging
5  spec:
6    selector:
7      matchLabels:
8        app: seq-logging
9    template:
10     metadata:
11       labels:
12         app: seq-logging
13     spec:
14       containers:
15       - name: seq-logging
```

```
16        image: datalust/seq
17        resources:
18          limits:
19            cpu: "100m"
20            memory: "128Mi"
21        env:
22          - name: ACCEPT_EULA
23            value: 'Y'
24        ports:
25        - containerPort: 5341
26        - containerPort: 80
```

Code-Ausschnitt 4: Beispielhaftes Deployment

In Code-Ausschnitt 4 ist das beispielhafte Deployment des Seq-Logging Dienstes zu sehen. Zu Beginn wird mit den Parametern `apiVersion` und `kind` die Version und der Typ des API-Objektes angegeben. Anschließend folgen unter `metadata` einige Informationen zum Objekt. Unter dem Punkt `spec` wird nun das Deployment spezifiziert. Zuerst werden `matchLabels` vergeben, mit denen das Deployment im späteren Verlauf von anderen API-Objekten (z.B. Services) referenziert werden kann. Danach folgt die Spezifizierung der Pods, die von dem Deployment erstellt werden sollen. In dieser Spezifizierung werden Informationen zum zu verwendenden Docker-Image, Ressourcen-Limits, Umgebungsvariablen und das gewünschte Port-Mapping angegeben.

6.4 Helm Deployment

Helm ist ein Paket-Manager für komplette Kubernetes Cluster. In einem Helm Chart werden die Definitionen aller API-Objekte gespeichert, die in einem Kubernetes Cluster verwendet werden. So müssen nicht alle Cluster-Definitionen einzeln ausgeliefert werden, sondern können als gemeinsames Paket installiert werden. Zudem werden in einem Helm Deployment die Struktur und die Werte der API-Objekte getrennt. Dies geschieht über sogenannte Templates. Es können beliebig viele Templates definiert werden, die später mit Werten aus einer *values.yaml* Datei gefüllt werden. Diese Templates erlauben auch Scripting. Das Scripting der Templates bietet die Möglichkeit, die Definition repetitiver API-Objekte zu vereinfachen. In

diesem Projekt kommt diese Vereinfachung sehr gelegen. Da es vier Microservices gibt, die im Grunde genommen die gleiche Konfiguration benötigen, werden die Kubernetes Deployment und Services der Microservices zusammengefasst. Diese Zusammenfassung spart zum einen Verwaltungsaufwand, da nicht für jeden Microservice eigene Deploy-Skripte erstellt werden müssen, zum anderen vermeidet die Zusammenfassung Flüchtigkeitsfehler, die auftreten könnten, wenn an jedem Deployment der Microservices etwas verändert werden müsste.

6.5 Externe Dienste

6.5.1 Datenbank - PostgreSQL

Da jeder Microservice Daten verwalten muss, wird eine Datenbank zur Speicherung der Daten verwendet. Zur Speicherung der Daten kommt in diesem Projekt das DBMS PostgreSQL zum Einsatz. Das DBMS wird zusammen mit den Microservices im Kubernetes Cluster betrieben.

6.5.2 Logging - Seq

Um eine zentrale Überwachung der geloggten Informationen aller Microservices zu ermöglichen, wurde das Logging-Tool Seq in dieses Projekt integriert. Seq bietet eine Übersicht über die Logs aller laufenden Microservices. Der Seq-Dienst wird im Kubernetes Cluster gehostet.

7 Service Mesh

Ein Service Mesh in einem Kubernetes Cluster ermöglicht die Steuerung und Überwachung des Netzwerkverkehrs im Cluster. In einem Cluster ohne Service Mesh kann jeder Pod mit jedem Service uneingeschränkt kommunizieren. Zudem ist auch keine Überwachung der Netzwerkkommunikation möglich, etwa um Bottlenecks im Cluster aufzudecken. Ein Service Mesh wird durch eine zusätzliche Netzwerkschicht im Kubernetes Cluster bereitgestellt. Dabei erhält jeder Kubernetes Service einen eigenen Proxy, durch den der komplette Datenverkehr dieses Services geleitet wird. So kann jeder Aufruf an einen Service überwacht und gegebenenfalls eingeschränkt werden. Ein Service Mesh erhöht dadurch die Sicherheit im Cluster, da festgelegt werden kann, welche Services untereinander Daten austauschen können.

Ein Service Mesh könnte in diesem Projekt auch zum Einsatz kommen. Aus der Anforderung wird klar, dass die Kommunikation zwischen einzelnen, ausgewählten Microservices nicht gewünscht ist. So darf darf nur der Checkout-Service andere Microservices im Cluster erreichen. Alle anderen Microservices sind für ausgehende Anfragen an Microservices im Cluster nicht freigegeben. Um diese Anforderung umzusetzen, bietet sich der Einbau eines Service Mesh an.

8 Fazit

Alle vier Backend Systeme und das Frontend wurden im Laufe der Arbeitsphase erfolgreich umgesetzt. Dabei wurde sich an das Anforderungsdokument gehalten. Dennoch ist der Shop aktuell in seiner Funktionalität sehr primitiv.

Wichtige Features wie eine Liste aller getätigten Bestellungen, Bestellbestätigungen per E-Mail oder die Funktion mehrere Gegenstände des selben Typs zu bestellen fehlen allerdings. Aber durch die von Anfang an groß gedachte Komponenten- und Systemarchitektur und saubere Entwicklung der Services stellen solche Anpassungen auch in Zukunft kein Problem dar.

Die durchgehende Verfügbarkeit des Systems ist durch Kubernetes selber schon sichergestellt. Mit der Einrichtung des *Horizontal Pod Autoscalers* sind zudem immer Pods vorhanden, welche noch Kapazitäten zur Verfügung haben.

Codeverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] *E-Commerce - Entwicklung des Umsatzes 2022*. Statista. URL: https://de.statista.com/statistik/daten/studie/3979/umfrage/e-commerce-umsatz-in-deutschland-seit-1999/ (besucht am 10. 07. 2022).

[2] Greg Linden. *Geeking with Greg: Slides from my talk at Stanford*. Geeking with Greg. 4. Dez. 2006. URL: http://glinden.blogspot.com/2006/12/slides-from-my-talk-at-stanford.html (besucht am 10. 07. 2022).

[3] Raphael Herding. „Cloud Grundlagen und Programmierung: Abschlussprojekt“. In: (). URL: https://moodle.w-hs.de/pluginfile.php/493212/mod_resource/content/2/Abschlussprojekt_v2.pdf.

[4] auth0.com. *JWT.IO - JSON Web Tokens Introduction*. URL: http://jwt.io/ (besucht am 12. 07. 2022).

[5] *Concepts*. URL: https://www.ngxs.io/concepts/intro (besucht am 13. 07. 2022).

[6] Gaetano Piazzolla. *Update Angular/React environment dynamically reading variables from Kubernetes ConfigMaps*. CodeX. 21. Juli 2021. URL: https://medium.com/codex/update-angular-react-environment-dynamically-reading-variables-from-kubernetes-configmaps-ae32b8d4021c (besucht am 14. 07. 2022).

[7] *What is REST*. REST API Tutorial. URL: https://restfulapi.net/ (besucht am 19. 07. 2022).

[8] Lokesh Gupta. *HTTP Methods*. REST API Tutorial. 24. Mai 2018. URL: https://restfulapi.net/http-methods/ (besucht am 19. 07. 2022).

[9] Lokesh Gupta. *HTTP Status Codes*. REST API Tutorial. 30. Mai 2018. URL: https://restfulapi.net/http-status-codes/ (besucht am 20. 07. 2022).

[10] *Containerplattform: Lego für DevOps*. Informatik Aktuell. URL: https://www.informatik-aktuell.de/entwicklung/methoden/containerplattform-lego-fuer-devops.html,%20https://www.informatik-aktuell.de/entwicklung/methoden/containerplattform-lego-fuer-devops.html (besucht am 20. 07. 2022).

[11] *Der App-Store für Server? Wofür Sie Container brauchen - SHD*. URL: https://www.shd-online.de/fachartikel/der-app-store-fuer-server-wofuer-sie-container-brauchen/ (besucht am 20. 07. 2022).

[12] *Horizontal Pod Autoscaling*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (besucht am 23. 07. 2022).