SpringBoot Workshop: Verteilte Anwendungen

Dieser Workshop beschäftigt sich mit der Implementierung eines Online-Shops. Die Architektur ist eine verteilte polyglotte Microservice Architektur. Es wird bewusst auf eine sogenannte Choreographie verzichtet, da diese im Vergleich, zu der hier verwendeten Orchestration, den zeitlichen Rahmen überschreitet. Die von euch entwickelten Microservices können untereinander genutzt werden. Hierfür stellen wir eine Service Discovery bereit, die auf dem Netflix Open Source Stack basiert.

Der Workshop bietet einen Schnelleinstieg in die Entwicklung von Applikationen mit dem Java-Framework SpringBoot. Es werden verschiedene In-Memory Datenbanken wie H2 (relationale DB) und MongoDB (Dokumenten-DB) verwendet. Außer der Konfiguration werden die Unterschiede der Datenbanken durch Spring abstrahiert und bieten so ein einheitliches Programmiermodell. Das Einfügen und Manipulieren von Daten wird über REST-Schnittstellen realisiert werden. Diese Schnittstellen können dann über einen bereitgestellten Client genutzt werden.

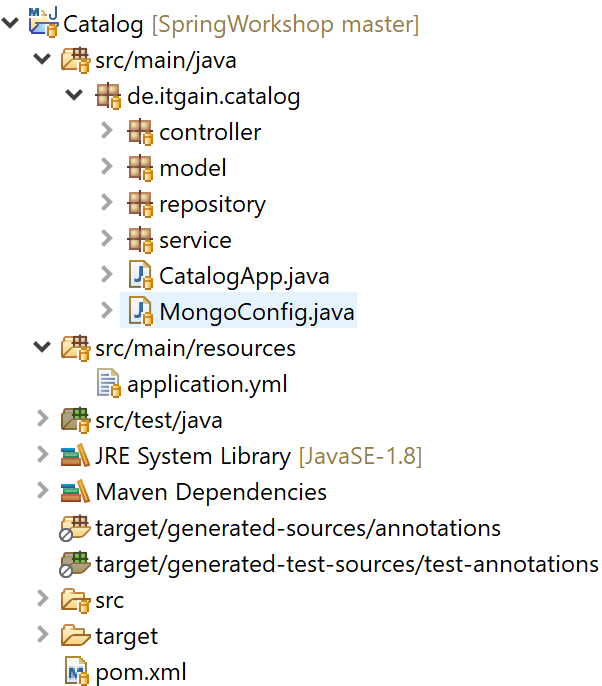
# Allgemeine Konfigurationen

Die Service Discovery wird über einen Eureka Server geregelt. Daher ist es für alle Anwendungen notwendig die *application.yml* zu konfigurieren.

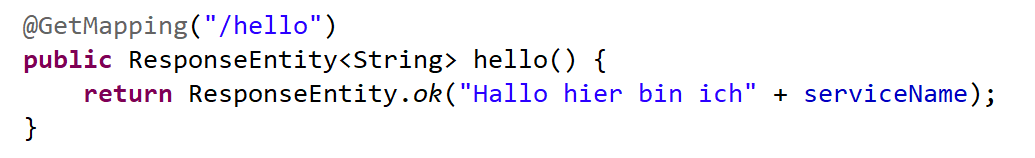
|  |
| --- |
| spring:  application:  name: <ServiceName>    eureka:  client:  serviceUrl:  defaultZone: <EurekaServerURL>  instance:  metadataMap:  instanceId: applicationname:${spring.application.instance\_id:${random.value}}  preferIpAddress: true  statusPageUrlPath: /hello/    server:  port: 9891  servlet:  context-path: /<ServiceName> |

Einige Anwendungen benötigen zusätzliche Konfiguration, z.B. für die Anbindung an eine Datenbank. Diese Informationen residieren entweder in entsprechenden Configuration-Klassen oder werden in der *application.yml* des Projektes gepflegt*.*

Pro Service wurde ein Maven-Projekt angelegt. In der pom.xml sind die Abhängigkeiten bereits gelistet (z.B. SpringBoot, Junit, H2, MongoDB…) und es müssen keinen weiteren Abhängigkeiten ergänzt werden. Allgemein sieht die Struktur der Projekte in Eclipse so aus:



Um sicherzustellen, dass jeder Service via Service Discovery auffindbar ist, wird die Verbindung zum Eureka-Server getestet. Daher wird zunächst in jeder Anwendung ein REST-Endpunkt “/hello” erstellt. Dieser soll bei einer Anfrage die Zeichenkette “Hallo hier bin ich <Servicename>” als Antwort zurückgeben.

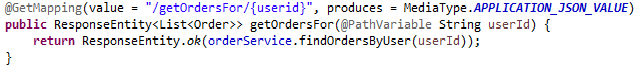


## @PostMapping

Dieser Controller Mapped automatisch den Request in ein JSON Objekt, damit der Service die Daten weiterverarbeiten kann. Ebenso wird der HTTP Response und der Payload in das JSON Format gemapped. ResponseEntity ermöglicht direkt den Status OK zu übermitteln. Im Fehlerfall sollte man einen Bad Request oder ähnliches zurückliefern.

## @GetMapping

Der GetController ist ähnlich wie der Post Controller. Hier können Daten direkt über die URL mitgesendet werden und diese werden dann anhand der **@PathVariable** Annotation gemapped.



# Datenhaltung

Alle Services, bis auf der OrchestrationService, sollen eine In-Memory-Datenbank verwenden. In diesem Workshop soll nicht eine Datenbank verwendet werden, sondern verschiedene. Es soll abhängig vom Anwendungsfall die Datenbank genutzt werden, die sich am besten eignet (polyglotte Persistenz). Wenn eine Service-Beschreibung im folgenden Abschnitt eine Sektion “Datenhaltung” hat, sind folgende Schritte notwendig.

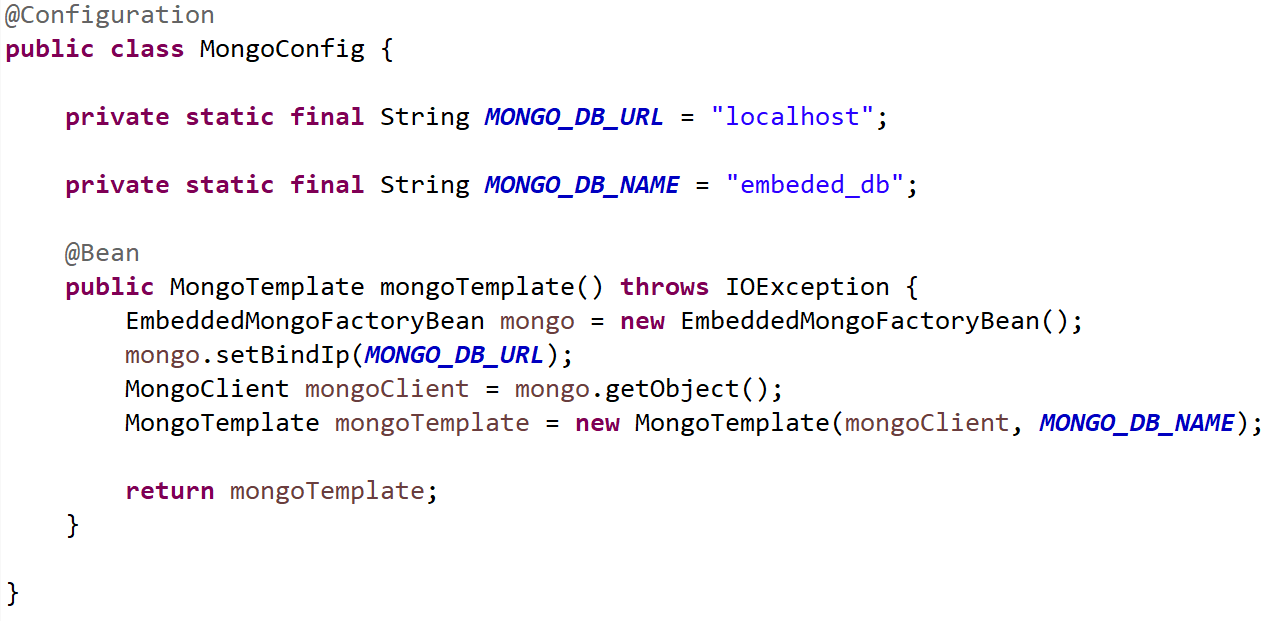
## Konfiguration der Datenbank

Spring ermöglicht es Datenbanken beim Start der Anwendung In-Memory zu erzeugen. Die Konfiguration ist in der application.yml eines Projektes enthalten. Datenbank-spezifische Einstellungen sind in der Beschreibung des jeweiligen Service zu finden.

Bei Start der Anwendung ist die In-Memory-Datenbank mit Testdaten zu befüllen. Am besten eignet sich die Annotation ***@PostConstruct.*** Hierdurch wird die Methode ausgeführt, sobald Spring alle

### MongoDB Konfiguration

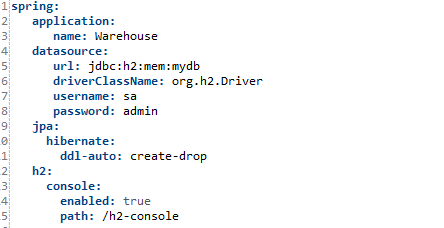
Folgende Konfigurationen sind nötig damit die In-Memory-Datenbank korrekt konfiguriert ist. Die Klasse, welche die Konfiguration enthält, muss mit ***@Configuration*** annotiert werden und soll in dem Package ***de.itgain.catalog*** liegen.



Weitere Informationen unter <https://docs.spring.io/spring-data/mongodb/docs/1.3.3.RELEASE/reference/html/mongo.repositories.html>

### H2-Datenbank Konfiguration

Folgende Konfigurationen sind in der **application.yml** nötig



## Interaktion mit der Datenbank

Um auf Daten zuzugreifen wird hier SpringJPA verwendet. Durch die Annotation ***@Repository*** an einem Java-Interface wird automatisch eine Tabelle, Entität (je nach DB) mit dem Namen der Klassen angelegt. Darüber hinaus sind CRUD- (Create, Read, Update, Delete) und einige weitere Operationen automatisch verfügbar. Welches Repository benötigt wird steht ausführlich bei den Beschreibungen der Services. Die Repositories soll in dem Package ***de.itgain.catalog.repository*** liegen und ein Repository injecten.

Die Definition eines Repositories sieht formal so aus:

|  |
| --- |
| public interface MyRepository<T, ID> extends Repository<T, ID> |

T und ID sind müssen hier bei Implementierung mit den entsprechenden Typen befüllt werden. Bei T handelt es sich um den Typ des Objektes, der gespeichert werden soll. ID ist der eindeutige Identifier des Objektes in der Datenbank (z.B. Long, String, …).

Eine einfache Read-Operation eines Objektes sieht dann so aus und ist bereits von Spring implementiert:

|  |
| --- |
| public Optional<T> findById(ID id) |

Soll beispielsweise ein Objekt vom Typ Product mit einer ID vom Typ String persistiert werden, würde die Definition des Repositories so aussehen:

|  |
| --- |
| public interface ProductRepository<Produkt, String> extends Repository<Produkt, String> |

Die Signatur für die Operation, um ein Produkt mit einer bestimmten ID zu finden, hätte dann folgende Form:

|  |
| --- |
| public Optional<Product> findById(String id) |

Ebenfalls gibt es Repositories für Paging und Sortierung:

|  |
| --- |
| PagingAndSortingRepository<Entity, String> |

# Bereitgestellte Komponenten

Es werden sowohl ein Client als auch die Service Discovery von uns bereitgestellt.

## Client

Der dient Client zur Interaktion mit den Services durch eine benutzerfreundliche Schnittstelle. Hier handelt es sich um eine Webanwendung, die ebenfalls im Git-Repository enthalten ist. Hierfür muss lediglich die index.html-Datei des Client-Projektes in einem Browser geöffnet werden. Der Server, welcher durch den Client angesprochen werden soll, ist bereits vorkonfiguriert.

## EurekaServer

Diese beiden Komponenten werden vorab gestellt. Der Client soll nicht Ziel des Workshops sein. Er bietet eine vorab definierte Schnittstelle und kommuniziert mit dem OrchestrationService. Der Eureka Server ist in der Basis Konfiguration sehr trivial und wird daher von uns gestellt :)

# Services

In diesem Abschnitt stellen wir euch die verschiedenen Services mit ihren Operationen vor.

## OrchestrationService

Dieser Service aggregiert die Anfragen von dem Client. Eine andere Möglichkeit wäre eine Choreographie, diese werden wir aufgrund der Komplexität und zeitlichen Begrenzung hier nicht thematisieren.

### Vorabkonfiguration

OrchestrationService.zip

1. Application Property Datei anpassen:
   1. Name: OrchestrationService
   2. Port: freien auf dem Endgerät wählen

### Endpoints

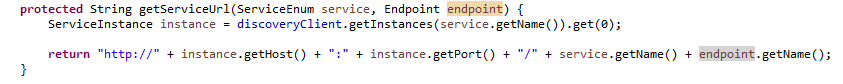
Der Service hat folgende Schnittstellen Root node ist /api :

* Zeige nur X Produkte (Paging)
  + GET catalog/getProductPage/{page}
  + Liste von Produkten im JSON Format
* Zeige nur X Produkte an mit Filter auf Suchbegriff
  + GET catalog/filterProductPage /{Query}/{page}
  + Liste von Produkten im JSON Format
* Zeige ein spezielles Produkt an
  + GET catalog/getProduct/{id}
  + Produkt im JSON Format
* Erstelle ein Produkt und setze Warenbestand (Hinweis mehrere Service Calls)
  + POST catalog/createProduct/
  + Produkt im JSON Format
* Registrierung eines Kunden
  + POST Register/{Name}{Password}
  + User im JSON Format
* Login eines Kunden
  + POST login/{Name}{Password}
  + User im JSON Format
* Logout eines Kunden
  + POST logout/{Name}{Password}
  + User im JSON Format
* Zeige die Bestellungen eines Kunden
  + GET order/getOrdersFor/{userId}
  + Liste von Orders im JSON Format
* Füge Produkt dem Warenkorb hinzu
  + POST /shoppingcart/addProduct/{userId}{productId}{amount}
  + ShoppingCart JSON
* Führe Bestellung aus (Hinweis mehrere Service Calls)
  + POST order/createOrder/{Order}
  + Order im JSON Format

Der Client kennt nur den OrchestrationService, dieser wiederrum koordiniert die einzelnen Aufrufe und aggregiert die Ergebnisse verschiedener Services zu einer Antwort für den Client.

Zu Beachten ist der Umgang mit den JSON Strukturen die die Microservices zurückgeben. Diese sind unter der Sektion Modell zu finden. Der OrchestrationService muss teilweise aus Elemente zugreifen und diese weiterverarbeiten, dahe rist eine einheitliche JSON Struktur wichtig.

Allgemein muss für jede Operation eine Serviceinstanz vom Eureka Server angefordert werden. Nachdem die Service Instanz bekannt ist soll auf dieser die Operation ausgeführt werden. Um die ServiceDiscovery von Eureka zu verwenden wird ein **DiscoveryClient** eingebunden. Mit .get(Name) bekommt man von dem DIscoveryClient alle Services die sich mit dem Namen registriert haben. Nun haben können die konkrete URL und Port verwendet werden um einen Serviceaufruf zu starten.

Es empfiehlt sich Enums für Services und URLs pro Services bereitzustellen. Dies erleichtert das Zusammenbauen der URLs.

### Orchestrierende Services

In diesem Abschnitt haben wir zum allgemeinen Verständniss die Usecases hinter den Schnittstellen des Orchestrationgateways zusammen getragen. DIe Bilder geben wir euch nochmal separat, hier sind sie leider unscharf:

#### User Operationen:

#### 

Register Request:

<http://localhost:9894/api/user/register/>

Post

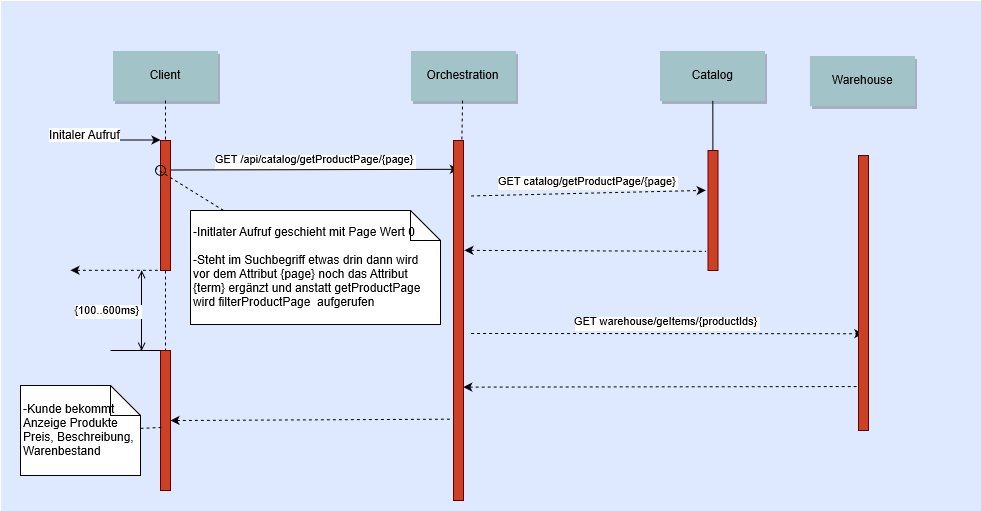
* {name: "hans", password: "www"}
  + name: "hans"
  + password: "www"

Response

* {id: 1, name: "hans", password: "www"}
  + id: 1
  + name: "hans"
  + password: "www"

#### Items anlegen:

#### Catalog Paging:



Paging Request:

<http://localhost:9894/api/catalog/getProductPage/0>

GET

Response:

* content: [,…]
  + 0: {id: "5dd720e148f4c237ac78686b", name: "Fleisch", description: "Super leckers Fleisch", price: 5.99}
  + 1: {id: "5dd720e248f4c237ac78686c", name: "Kaese", description: "Herzhafter Kaese", price: 3.59}
  + 2: {id: "5dd720e248f4c237ac78686d", name: "Nudel", description: "Kaum keimbelastete Nudeln", price: 2.95}
* empty: false
* first: true
* last: false
* number: 0
* numberOfElements: 3
* pageable: {sort: {sorted: false, unsorted: true, empty: true}, offset: 0, pageNumber: 0, pageSize: 3,…}
  + offset: 0
  + pageNumber: 0
  + pageSize: 3
  + paged: true
  + sort: {sorted: false, unsorted: true, empty: true}
    - empty: true
    - sorted: false
    - unsorted: true
  + unpaged: false
* size: 3
* sort: {sorted: false, unsorted: true, empty: true}
  + empty: true
  + sorted: false
  + unsorted: true
* totalElements: 6
* totalPages: 2

Produkte suchen nach Namen,

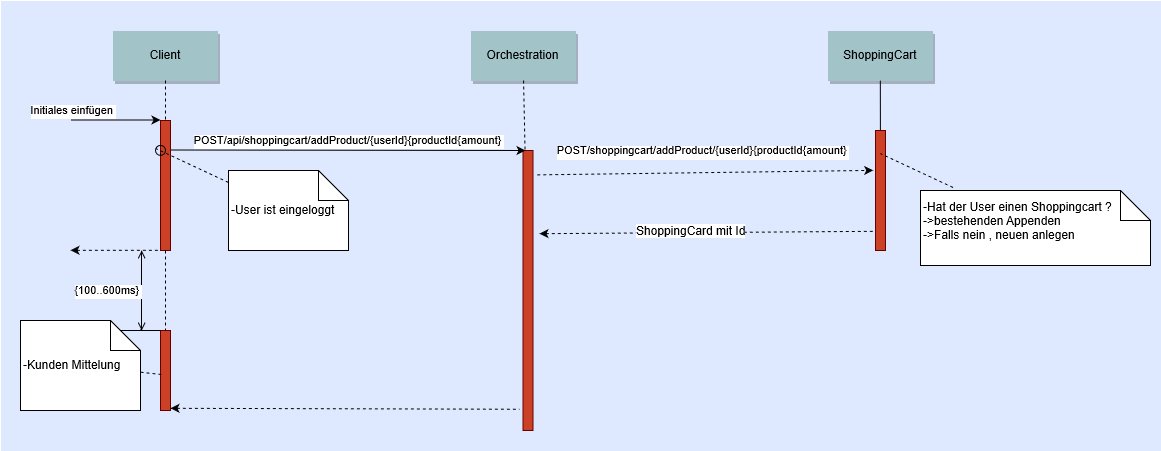
<http://localhost:9894/api/catalog/filterProducts/Nudel/0>

GET

Response

* content: [,…]
  + 0: {id: "5dd720e248f4c237ac78686d", name: "Nudel", description: "Kaum keimbelastete Nudeln", price: 2.95}
  + 1: {id: "5dd720e248f4c237ac78686e", name: "Nudelteig",…}
* empty: false
* first: true
* last: true
* number: 0
* numberOfElements: 2
* pageable: {sort: {sorted: false, unsorted: true, empty: true}, offset: 0, pageNumber: 0, pageSize: 3,…}
  + offset: 0
  + pageNumber: 0
  + pageSize: 3
  + paged: true
  + sort: {sorted: false, unsorted: true, empty: true}
  + unpaged: false
* size: 3
* sort: {sorted: false, unsorted: true, empty: true}
  + empty: true
  + sorted: false
  + unsorted: true
* totalElements: 2
* totalPages: 1

#### Produkt dem Einkaufswagen hinzufügen:

Add Item Request:

<http://localhost:9894/api/shoppingcart/addProduct/>

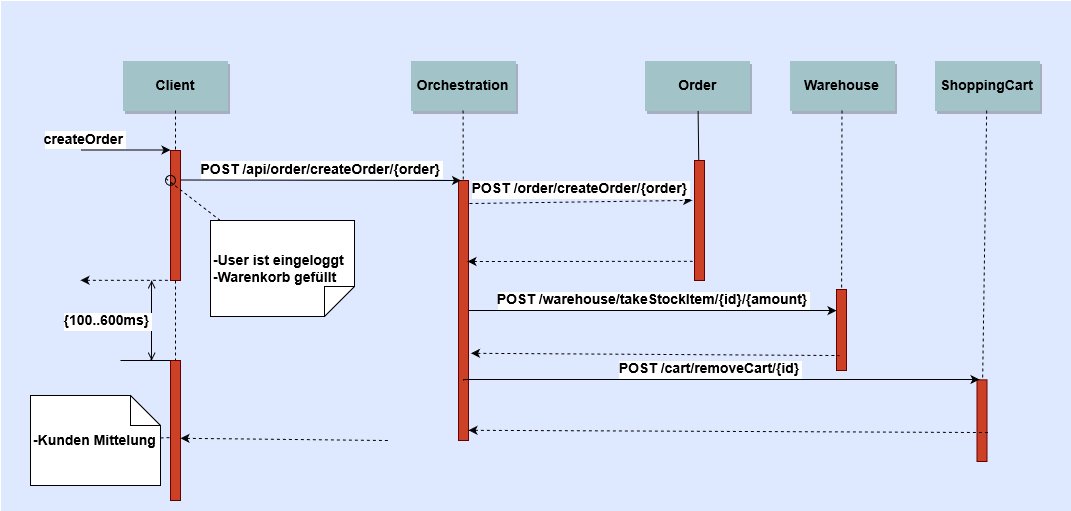
Post

* item: {id: "5dd720e148f4c237ac78686b", name: "Fleisch", description: "Super leckers Fleisch", price: 5.99,…}
  + amount: 1
  + description: "Super leckers Fleisch"
  + id: "5dd720e148f4c237ac78686b"
  + name: "Fleisch"
  + price: 5.99
* userId: 1

Response

* id: "1b3fda87-7d61-434c-b922-46e27a9c7ab8"
* items: [{id: "5dd720e148f4c237ac78686b", name: "Fleisch", price: 5.99, amount: 1}]
  + 0: {id: "5dd720e148f4c237ac78686b", name: "Fleisch", price: 5.99, amount: 1}
* updated: "2019-11-22T01:10:19.4233755"
* userId: 1

#### Order erstellen:

Create Order Request

http://localhost:9894/api/order/createOrder/

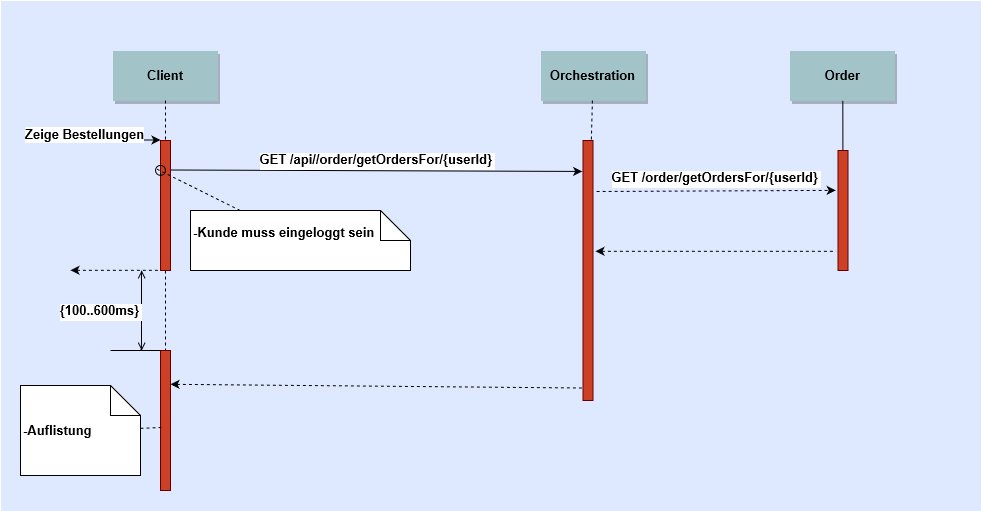
Post

* cart: {id: "1b3fda87-7d61-434c-b922-46e27a9c7ab8", userId: 1, updated: "2019-11-22T01:10:19.4233755",…}
  + id: "1b3fda87-7d61-434c-b922-46e27a9c7ab8"
  + items: [{id: "5dd720e148f4c237ac78686b", name: "Fleisch", price: 5.99, amount: 1}]
  + updated: "2019-11-22T01:10:19.4233755"
  + userId: 1
* userId: 1

Response

* created: "2019-11-22T00:12:20.365+0000"
* id: "5dd727e448f4c22c8c35d671"
* products: [{productId: "5dd720e148f4c237ac78686b", amount: 1}]
  + 0: {productId: "5dd720e148f4c237ac78686b", amount: 1}
    - amount: 1
    - productId: "5dd720e148f4c237ac78686b"
* status: "INIT"
* userId: "1"

#### Zeige Bestellungen:



Zeige Bestellung Request

http://localhost:9894/api/order/getOrdersFor/1

Get

Response

* created: "2019-11-22T00:12:20.365+0000"
* id: "5dd727e448f4c22c8c35d671"
* products: [{productId: "5dd720e148f4c237ac78686b", amount: 1}]
  + 0: {productId: "5dd720e148f4c237ac78686b", amount: 1}
* status: "INIT"
* userId: "1"

## UserService

Der Nutzerverwaltung ermöglicht eine Registrierung in dem Shop und hält die Daten der Nutzer vor. Es werden drei Operationen für die Benutzerverwaltung angeboten.

Ein Benutzer besitzt eine eindeutige ID, einen Vor- sowie Nachname, ein Passwort. Auf weitere Daten wird an dieser Stelle, aus Gründen der Einfachheit, verzichtet. Die ID wird vom System vergeben und muss eindeutig sein. Alle anderen Daten sind während des Registrierungsprozesses vom Anwender anzugeben. Hat ein Benutzer sich erfolgreich registriert ist er automatisch eingeloggt. Ist ein Benutzer registriert und nicht eingeloggt, kann sich dieser mit seinem Namen und Passwort einloggen. Im eingeloggten Zustand ist ein Logout möglich. Hierdurch wird die Session beendet.

### Vorabkonfiguration

UserService.zip

1. Application Property Datei anpassen:
   1. Name: UserService
   2. Port: freien auf dem Endgerät wählen

### Endpoints

Die Endpoints sind unter dem Root Node user:

* Registrierung eines Kunden
  + POST Register/{Name}{Password}
  + User im JSON Format
* Login eines Kunden
  + POST login/{Name}{Password}
  + User im JSON Format
* Logout eines Kunden
  + POST logout/{Name}{Password}
  + User im JSON Format

CatalogService

Der CatalogService ist eine Implementierung eines Produktkatalog-Dienst. Ein Produktkatalog ermöglicht es dem Anwender, Produkte abzufragen. Dabei soll es auch möglich sein nach Produkten zu filtern, die einen bestimmten Begriff beinhalten. Da es bei einer hohen Anzahl von Produkten im Sortiment nicht praktikabel ist alle Produkte abzufragen, soll immer nur eine bestimmte Anzahl an Produkten gesendet werden. Der Client kann auf diese Weise immer eine fest definierte Menge an Produkten darstellen und bietet eine Möglichkeit zu Pagination.

### Vorabkonfiguration

CatalogService.zip

1. Application Property Datei anpassen:
   1. Name: Catalog
   2. Port: freien auf dem Endgerät wählen

## Endpoints

DIe Endpoints sind unter dem **Root Node catalog**:

* Zeige nur X Produkte (Paging)
  + GET /getProductPage/{page}/{size}
  + Liste von Produkten im JSON Format
* Zeige nur X Produkte an mit Filter auf Suchbegriff
  + GET / filterProductPage /{Query}/{page}
  + Liste von Produkten im JSON Format
* Erstelle ein Produkt
  + POST /createProduct/
  + Produkt im JSON Format

### Modell

Ein Produkt besteht aus einem eindeutigen Identifier, einem Namen einer Beschreibung und einem Preis. Die Wahl der Datentypen sollte sinnvoll sein aber bleibt dem Entwickler überlassen.

Erwartete JSON Struktur:

|  |
| --- |
| [  {  "id": "5dc550b692b65238042b0b4e",  "name": "Fleisch",  "description": "Super leckers Fleisch",  "price": 5.99  },  {  "id": "5dc550b692b65238042b0b4f",  "name": "Kaese",  "description": "Herzhafter Kaese",  "price": 3.59  },  {  "id": "5dc550b692b65238042b0b50",  "name": "Nudel",  "description": "Kaum keimbelastete Nudeln",  "price": 2.95  },  {  "id": "5dc550b692b65238042b0b51",  "name": "Nudelteig",  "description": "Noch weniger keimbelastete Nudeln sind hiermit moeglich",  "price": 3.49  }  ] |

### Datenhaltung

Der CatalogService wird eine MongoDB verwenden. Siehe Abschnitt MongoDB Konfiguration

## WarehouseService

Der WarehouseService verwaltet den Warenbestand.

### Vorabkonfiguration

WarehouseService.zip

1. Application Property Datei anpassen:
   1. Name: Warehouse
   2. Port: freien auf dem Endgerät wählen

## Endpoints

Die Endpoints sind alle unter dem Endpoint warehouse:

* Frage den Warenbestand von X Produkten ab
  + GET / getItems /{ProductIds}
  + Liste mit den Warenbeständen
* Erhöhe den Warenbestand von einem Produkt um eine Menge
  + GET /addStockItem/{id}/{amount}
  + Aktuellen Warenbestand
* Verringere den Warenbestand von einem Produkt um eine Menge
  + GET /takeStockItem/{id}/{amount}
  + Aktuellen Warenbestand

Datenhaltung

## OrderService

Dieser Service legt Bestellungen für einen Kunden an und bietet Verwaltungsoperationen.

### Vorabkonfiguration

OrderService.zip

1. Application Property Datei anpassen:
   1. Name: OrderService
   2. Port: freien auf dem Endgerät wählen

### Endpoints

Die Services sind unter **dem Endpunkt /order**

* Zeige die Bestellungen eines Kunden
  + GET /getOrdersFor/{userid}
  + Liste von Orders als JSON
* Zeige eine Bestellungen eines Kunden
  + GET /getOrdersFor/{userid}/[orderid]
  + Liste von Orders als JSON
* Lege Order an
  + POST /createOrder
  + Payload JSON Order

## Datenhaltung

Der Orderservice speichert die Daten in einer MongoDB die Konfiguration in dem Bereich Datenhaltung MongoDB nachzulesen.

Modell

|  |
| --- |
| [  {  "id": "5dc550b692b65238042b0b4f",  "products": [  { "productId":"6ac550b692b65238042b0b4f", "amount":4 },  { "productId":"7ac550b692b65238042b0b4f", "amount":2 }  ],  "userId": "abc234"  }  ] |

## ShoppingCart

Dieser Service ist legt für den Kunden einen Warenkorb an. Diese ist für eine Bestellung nötig.Vorabkonfiguration

OrderService.zip

1. Application Property Datei anpassen:
   1. Name: ShoppingCart
   2. Port: freien auf dem Endgerät wählen

### Endpoints

Die Services sind unter **dem Endpunkt /order**

* Füge einen Artikel dem ShoppingCart hinzu
  + POST /additem/{ShoppingCart<0
  + ShoppingCart JSON
* Lege Order an
  + POST /removeCart/{cartId}
  + Payload JSON Order

## Datenhaltung

Der ShoppingCart speichert die Daten in einer MongoDB die Konfiguration in dem Bereich Datenhaltung MongoDB nachzulesen.

Modell

ShoppingCart als JSON

|  |
| --- |
| {  "id": "b2423e91-4fcf-4657-8ddc-f5fbe374d153",  "userId": 123,  "updated": "2019-11-16T16:16:47.3903258",  "items": [  {  "id": "5dd00b6b48f4c23700698b92",  "name": "Fleisch",  "price": 5.99,  "amount": 1  }  ]  } |