T-302-HONN: Project 2 Microservices

Þórður Friðriksson
thordurf@ru.is

Háskólinn í Reykjavík — Reykjavík University

Inngangur

Takið eftir: Vinsamlegast lesið yfir alla eftirfarandi punkta áður en þið byrjið á verkefninu

1. **Hópar**
   - Þetta verkefni getur verið skilað í 1-4 manna hópum
   - Þið þurfið eingöngu að skila verkefninu einu sinni per hóp
   - Munið að að merkja verkefnið á forsíðu með nafni og netfangi á öllum hópmeðlimum
2. **Reglur um svindl**
   - Lausnin verður að vera ykkar eigið verk, ef upp kemst um svindl fá allir tengdir aðilar 0 fyrir verkefnið ef þetta er fyrsta brot, ef þetta er endurtekið brot má búast við harðari refsingu. Sjá reglur skólans um verkefnavinnu: ru.is/namid/reglur/reglur-um-verkefnavinnu
3. **Verkefnaskil**
   - Verkefnum skal vera skilað á Canvas bæði sem zip skrá fyrir kóðann
   - zip skráin skal heita: `{student1@ru.is}-{student2@ru.is}-project2.zip`
   - **Ef ekki er fylgt fyrirmælum um verkefnaskil má búast við niðurlækkun á einkun**
4. **Sein skil**
   - Sjá late submission policy

1 Yfirlit

Í þessu verkefni á að smíða vörukerfi með *event-driven microservice architecture*. Kerifð á að geta stofnað og gjaldfært kaup sem og að senda út email fyrir tiltekin skref í verkflæðinu.

Kerfið mun samanstanda af nokkrum litlum microservices þar sem hvert service er einangrað og decoupled frá hinum service-unum og munu þau eingögnu hafa samskipti við hvort annað í gegnum network calls. Sum network calls munu vera event-driven en önnur munu vera request/response based.

Markmiðið með verkefninu er að fá tilfinnignu fyrir því hvernig við getum þróað highly scalable, decoupled og fault tolerant hugbúnað með event-um og distributed service-um.

Við verkefnið munum þið nota REST(framework af ykkar vali), RabbitMQ, Python, Docker og gagnagrunn af ykkar vali.

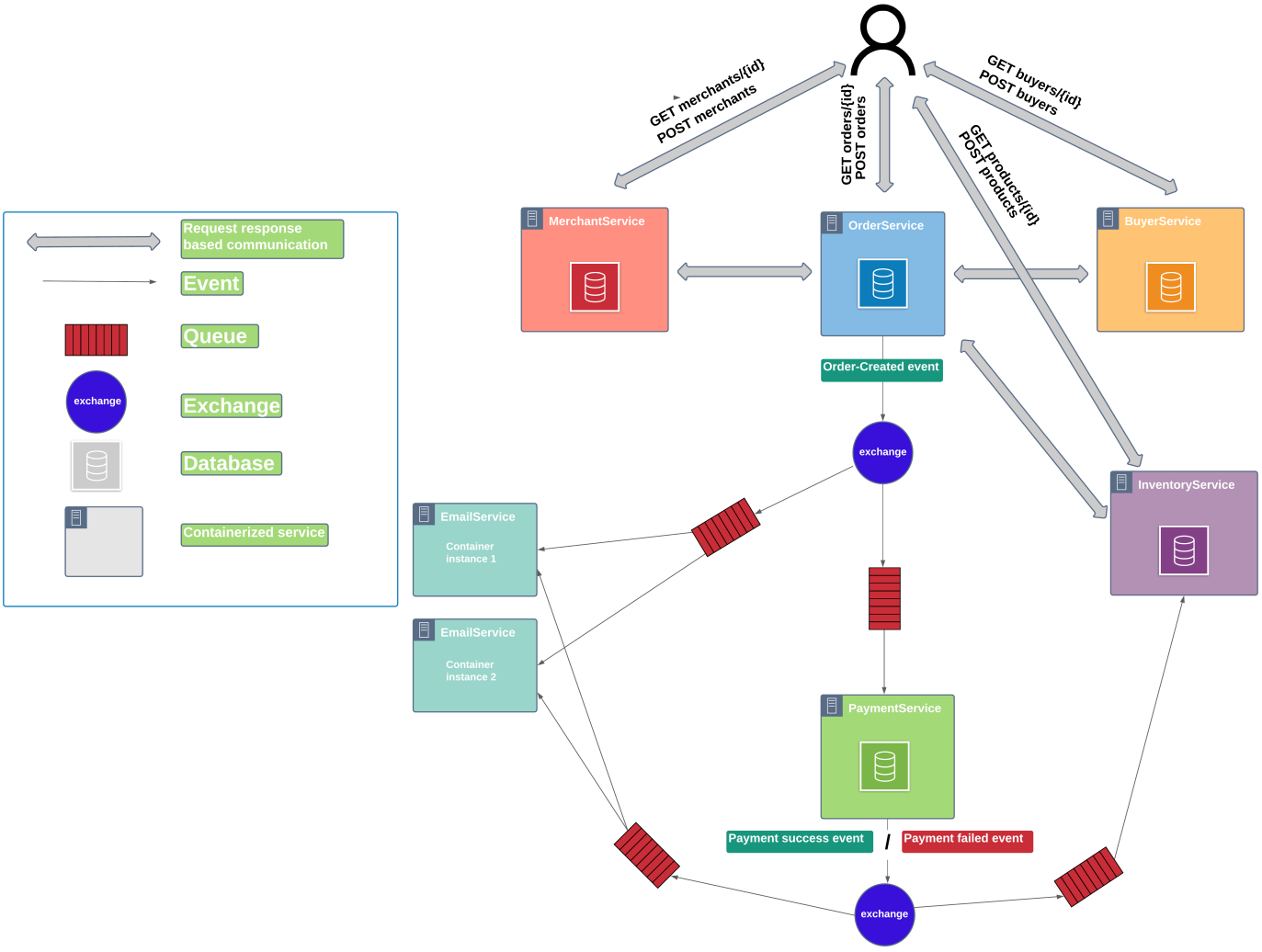
Ath Hafið í huga að þetta kerfi sem þið munuð hanna er oversimplified og fylgir ekki öllum best-practices við þróun á event-driven microservice kerfi. Það vantar security, almennilegt error handling, monitoring, logging, betra verkflæði, container orchestration og microservice-in sjálf eru einföld og bare bones.

Verkefnið er aðallega ætlað til að gefa ykkur skyndisýn af því hvernig það er að vinna með microservices, event-driven architecture, distributed kerfi og Docker.

2 Yfirlitsmynd yfir flæðið í kerfinu

Kerfið þarf að geta meðhöndlað beiðni um að stofna kaup, ferlið sem felst í því að stofna kaup í kerfinu er margþætt:

1. Fyrst sendum við/notandinn POST request-u á `OrderService` um að stofna kaup, þessi POST request-a inniheldur t.d. `buyer_id`, `merchant_id`, `product_id`, `credit_card` og fleiri upplýsingar um kaupin.
2. Næst, þegar `OrderService` fær beiðnina um að stofna kaup, þá talar `OrderService` við `MerchantService` og `BuyerService` með Request/Response based communication (t.d. REST) og athugar hvort að það er til Kaupandi og Seljandi fyrir tilsvarandi `buyer_id` og `merchant_id`. Ef svo þá talar OrderService næst við `InventoryService` með request/response based communication og tekur frá vöru með `product_id`. Ef það tókst síðan að taka frá þessa vöru(varan var ekki uppseld / hún var til) þá vistar `OrderService` kaupin í gagnagrunn og sendir út event um að kaup hafa verið stofnuð.
3. Þetta event um að kaup hafa verið stofnuð er pikkað upp af bæði `EmailService` og `PaymentService`. `EmailService` sér um að senda út email að ákveðin kaup hafa verið stofnuð og `PaymentService` sér um að gjaldfæra kaupin. `PaymentService` sendir síðan út event um niðurstöðu gjaldfærslunnar.
4. Þetta event um niðurstöðu gjaldfærslunnar er síðan pikkað upp að `EmailService` og `InventoryService`. `EmailService` sendir út email um niðurstöðuna, `InventoryService` Skráir kaupin sem hafa verið tekin frá sem annaðhvort seld ef gjaldfærslan tókst en annars ef gjaldfærslan tókst ekki þá er varan ekki lengur tekin frá.

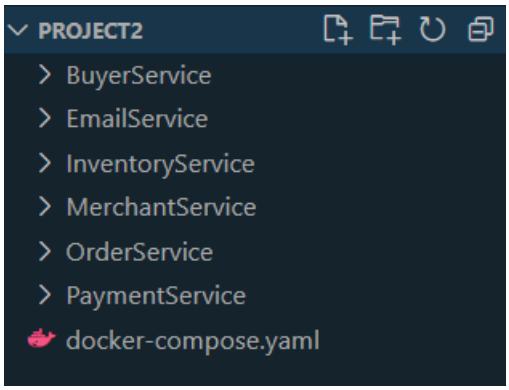


3 Kröfur

Ath Ég mæli með að þið lesið yfir alla verkefnalýsinguna áður en þið byrjið á útfærslunni.

3.1 Project Structure

- Ættuð að hafa sér folder fyrir hvert microservice þar sem hvert service folder heitir sama nafni og service-ið sjálft, t.d. ætti folder-inn sem hýsir `OrderService` að heita OrderService etc.
- Á sama folder level-i og microservice folder-arnir ætti að vera `docker-compose.yaml` file sem start-ar upp öllu kerfinu.



3.2 Databases

Allir gagnagrunnar sem eru notaðir í microservice-unum mega vera á hvaða formi sem er, þetta gæti verið SQL gagnagrunnur eins og `Postgres` eða NOSQL gagnagrunnur eins og `MongoDB` eða þetta gæti einfaldlega verið textaskrá sem er lesið úr og skrifað í.

Eina krafan er að gögn eru **persistent** á milli container tilvika.

3.3 Service APIs

`OrderService`, `BuyerService`, `MerchantService` og `InventoryService` hafa öll einhvern "External" Api sem notandi kerfisins getur kallað á til þessa að stofna eða sækja ákveðin kaup, seljanda, kaupanda eða vöru.

External API-arnir fyrir microservice-in eiga að vera gerð með REST samskiptum. Framework-ið sem er notað til að útfæra þessa External API-a er í ykkar höndum og má vera hvað sem er t.d. Django, FastApi, Flask

Það ætti að vera hægt að kalla á þessa external API-a locally þegar kerfið er keyrt um með `docker-compose.yaml`

Internal request-based samskipti (þ.e.a.s service-to-service samskipti) má vera útfært hvernig sem er, það má vera gert með REST(t.d. getur service notað external api-in á öðru service-i til að tala við það) eða það getur verið gert með pseudo-synchronous communication með RabbitMQ eða þetta getur verið gert með gRPC, GraphQL eða hverju sem þið viljið.

Ath Hafið í huga að oft myndum við ekki vilja að notandi kerfisins gæti kallað beint í microservice-in, við myndum mögulega frekar vilja hjúpa service-in okkar með einhvers skonar API-gateway. Með hlutum eins og kubernetes er einnig hægt að ráða nákvæmlega hvaða service API eru expose-uð út á við Kubernetes Ingress

Við gerum þetta eingöngu svona núna fyrir einfaldleika verkefnisins.

3.4 RabbitMQ

Sum samskipti á milli microservice-a munu vera gerð með events, til þess munum við nota RabbitMQ message broker-inn.

3.5 Docker

Allt kerfið á að vera containerized með Docker.

- Hvert service ætti að hafa `Dockerfile` og `.dockerignore` file
- Allir gagnagrunnar ættu að vera containerized og persistent
- RabbitMQ server-innn ætti að vera containerized
- Allt kerfið ætti að vera hægt að start-a upp með `docker-compose.yaml` file

Ath Í raun heiminum er ekki endilega normið að start-a öllu kerfinu upp með docker-compose þó það er oft algegnt. Aðrar leiðir eru að keyra bara upp service-ið sem þú ert að vinna í og tala við service keyrandi í skýinu í development umhverfi (t.d. með kubernetes er hægt að tengjast locally running service inn í cluster-inn með Bridge to kubernetes eða með Telepresence). Alternative við compose er t.d. hlutir eins og Tye

3.6 OrderService

3.6.1 API

OrderService þarf að hafa external API sem notandi kerfisins getur notað til þess að stofna og sækja kaup.

1. `POST /orders`
    - Requets-ann á að sjá um að stofna kaup
    - Dæmi um request body-ið er:

```
{
  "productId": 123,
  "merchantId": 123,
  "buyerId": 123,
  "creditCard": {
    "cardNumber": "12341234123412341234",
    "expirationMonth": 8,
    "expirationYear": 2025,
    "cvc": 123
  },
  "discount": 0.2
}
```

    - OrderService ætti að skila **400 HTTP Status Code** með villuskilaboðunum "Merchant does not exist" ef það er ekki til seljandi með tiltekið `merchantId`.
    - OrderService ætti að skila **400 HTTP Status Code** með villuskilaboðunum "Buyer does not exist" ef það er ekki til kaupandi með tiltekið `buyerId`
    - OrderService ætti að skila **400 HTTP Status Code** með villuskilaboðunum "Product does not exist" Ef vara er ekki til með tilekið `productId`
    - OrderService ætti að skila **400 HTTP Status Code** með villuskilaboðunum "Product is sold out" Ef vara með tiltekið `productId` er uppseld
    - OrderService ætti að skila **400 HTTP Status Code** með villuskilaboðunum "Product does not belong to merchant" ef seljandi með `merchantId` á ekki vöru með `productId`.
    - OrderService ætti að skila **400 HTTP Status Code** með villuskilaboðunum "Merchant does not allow discount" ef seljandi með `merchantId` leyfir ekki afslátt og tilgreindur `discount` er eitthvað annað en null eða 0.
    - Ef allt validation gengur upp þá ætti OrderService að taka frá vöruna, vista kaup-in í gagnagrunn, senda event um að kaup hafa verið stofnuð og skila **201 HTTP Status Code** með order id-i sem response message.

2. `GET /orders/{id}`
    - Ætti að skila **200 HTTP Status Code** með response body sem kaup fyrir tilsvarandi `id`. Kaupin ættu að vera skiluð út á eftirfarandi json form-i:

```
{
  "productId": 123,
  "merchantId": 123,
  "buyerId": 123,
  "cardNumber": "************1234"
  "totalPrice": 123.0
}
```

      `cardNumber` er af sömu lengd og upprunalega kortanúmerið en sýnir eingöngu seinustu fjórar tölurnar og `totalPrice` er verðið á tiltekinni vöru með tilsvarandi `productId` með afslættinum sem er skilgreindur þegar kaup-in voru stofnuð, t.d. ef verð á tiltekinni vöru er 100 og `discount`-ið sem er gefið er 0.2 þá væri `totalPrice` = price * (1 - 0.2)

    - OrderService ætti að skila **404 HTTP Status Code** með villuskilaboðunum "Order does not exist" ef kaup með tilsvarandi `id` er ekki til.

3.6.2 Database

`OrderService` ætti að hafa persistent gagnagrunn til að geyma kaupin

3.6.3 RabbitMQ

Við stofnun kaupa ætti OrderSerivce að senda út event um að kaup hafa verið stofnuð, event-ið ætti að innihalda allar þær upplýsingar um stofnuðu kaupin sem event processor-ar þurfa til að vinna úr event-inu.

3.6.4 Docker

- OrderService ætti að vera containerized
- Order Gagnagrunnurinn ætti að vera containerized
- RabbitMQ server-inn ætti að vera containerized
- Það ætti að vera hægt að kalla á `OrderService` API-inn locally á port-i 8000. T.d. ætti að vera hægt að senda http requestun-a `GET http://localhost:8000/orders/5` til að sækja kaup með id 5
- `OrderService`, Order Gagnagrunnurinn og `RabbitMQ` ættu að vera partur af sameiginlegri `docker-compose.yaml` skrá til að start-a upp öllu kerfinu.

3.7 MerchantService

3.7.1 API

1. `POST /merchants`
   - Requets-ann á að sjá um að stofna seljanda
   - Dæmi um request body-ið er:

```
{
  "name": "John Johnson",
  "ssn": "4000000000",
  "email": "email@email.com",
  "phoneNumber": "1234567",
  "allowsDiscount": true
}
```

   - MerchantService ætti að vista seljandann í gagnagrunn.
   - Ætti að skila **201 HTTP Status Code** með merchant id-i sem response message.

2. `GET /merchants/{id}`
   - Ætti að skila **200 HTTP Status Code** með response body sem seljandi fyrir tilsvarandi `id`. Seljandinn ættu að vera skilaður út á eftirfarandi json form-i:

```
{
    "name": "John Johnson",
    "ssn": "4000000000",
    "email": "email@email.com",
    "phoneNumber": "1234567",
    "allowsDiscount": true
}
```

   - MerchantService ætti að skila **404 HTTP Status Code** ef seljandi með tilsvarandi `id` er ekki til.

3. Leyfilegt er að skilgreina fleiri endapunkta fyrir internal network köll (t.d. OrderService að kalla á MerchantService) ef það er vilji til þess en önnur microservice mega einnig kalla á External endapunktana sem eru skilgreindir að ofan. Þessir auka endapunktar mega vera gerðir meða hvaða network communication sem er (REST, GraphQL, gRPC, RabbitMQ etc).

3.7.2 Database

`MerchantService` ætti að hafa persistent gagnagrunn til að geyma seljendurna.

3.7.3 Docker

- MerchantService ætti að vera containerized
- Merchant Gagnagrunnurinn ætti að vera containerized
- Það ætti að vera hægt að kalla á `MerchantService` API-inn locally á port-i 8001. T.d. ætti að vera hægt að senda http requestun-a `GET http://localhost:8000/merchants/5` til að sækja seljanda með id 5
- `MerchantService` og Merchant Gagnagrunnurinn ættu að vera partur af sameiginlegri `docker-compose.yaml` skrá til að start-a upp öllu kerfinu.

3.8 BuyerService

3.8.1 API

1. `POST /buyers`
   - Requets-ann á að sjá um að stofna kaupanda
   - Dæmi um request body-ið er:

```
{
  "name": "John Johnson",
  "ssn": "4000000000",
  "email": "email@email.com",
  "phoneNumber": "1234567",
}
```

   - BuyerService ætti að vista kaupandann í gagnagrunn.
   - Ætti að skila **201 HTTP Status Code** með buyer id-i sem response message.

2. `GET /buyers/{id}`
   - Ætti að skila **200 HTTP Status Code** með response body sem kaupandi fyrir tilsvarandi `id`. Kaupandinn ættu að vera skilaður út á eftirfarandi json form-i:

```
{
  "name": "John Johnson",
  "ssn": "4000000000",
  "email": "email@email.com",
  "phoneNumber": "1234567",
}
```

   - BuyerService ætti að skila **404 HTTP Status Code** ef kaupandi með tilsvarandi `id` er ekki til.

3. Leyfilegt er að skilgreina fleiri endapunkta fyrir internal network köll (t.d. OrderService að kalla á BuyerSercice) ef það er vilji til þess en önnur microservice mega einnig kalla á External endapunktana sem eru skilgreindir að ofan. Þessir auka endapunktar mega vera gerðir meða hvaða network communication sem er (REST, GraphQL, gRPC, RabbitMQ etc).

3.8.2 Database

`BuyerService` ætti að hafa persistent gagnagrunn til að geyma kaupendurna.

3.8.3 Docker

- BuyerService ætti að vera containerized
- Buyer Gagnagrunnurinn ætti að vera containerized
- Það ætti að vera hægt að kalla á `BuyerService` API-inn locally á port-i 8002. T.d. ætti að vera hægt að senda http requestun-a `GET http://localhost:8002/buyers/5` til að sækja kaupanda með id 5
- `BuyerService` og Buyer Gagnagrunnurinn ættu að vera partur af sameiginlegri `docker-compose.yaml` skrá til að start-a upp öllu kerfinu.

3.9 PaymentService

3.9.1 Payment virkni

PaymentService-ið mun auðvitað ekki hafa neina alvöru gjaldfærslu logic, í staðinn munum við eingöngu validate-a credit card upplýsingarnar og ef það eru í lagi þá sendum við út payment success event en annars sendum við út payment failure event

- Notum luhn algorithm-ann til að validate-a kortanúmerið
- Month expiration gild-ið ætti að vera tala á bilinu 1 til 12
- Year expiration gild-ið ætti að vera fjögurra stafa tala
- CVC gild-ið ætti að vera þriggja stafa tala.

3.9.2 Database

`PaymentService` ætti að hafa persistent gagnagrunn til að geyma payment niðurstöðurnar. Það sem þarf að geyma er order id-ið og payment niðurstaðan (þ.e.a.s success eða failed)

3.9.3 RabbitMQ

PaymentService-ið pikkar upp Order-Created event og validate-ar kortaupplýsingarnar fyrir kaupin, ef kortaupplýsingarnar eru í lagi þá á PaymentService að senda út Payment-Successful event, annars ætti PaymentService að senda út Payment-Failure event.

3.9.4 Docker

- PaymentService ætti að vera containerized
- PaymentService gagnagrunnurinn ætti að vera containerized
- RabbitMQ server-inn ætti að vera containerized
- PaymentService ætti að vera partur af sameiginlegri `docker-compose.yaml` skrá til að start-a upp öllu kerfinu.

3.10 EmailService

3.10.1 Email virkni

EmailService á að geta sent út email um að ákveðin event hafi gerst til bæði kaupandans og seljandans.

- Til að senda email munum við nota Sendgrid
- Búið til Sendgrid aðgang
- Búið til Sender í Sendgrid (Þetta er email-ið sem þið munuð senda frá, þetta gæti verið RU netfang hjá ykkur eða t.d. nýtt temporary gmail netfang)
- Búið til API lykil með því að fara undir settings -> API Keys -> Create API Key
- Geymið þennan API lykil ásamt sender email-inu í .env skrá í rót EmailService foldersins og load-ið því í python kóðanum til að byrja að nota Sendgrid. Í .env skránni ætti API lykill-inn að vera undir nafninu `SENDGRID_API_KEY` og email-ið ætti að vera undir nafninu `SENDGRID_SENDER_EMAIL`
- Þið getið síðan t.d. fylgt þessum kóða til að senda email með Sendgrid í python -> Sendgrid python example

Ath Ekki skila kóðanum ykkar eða commit-a honum á git með þessum breytum í .env skránni þar sem þetta eru viðkvæmar uppýsingar. .gitignore-ið frekar .env.

3.10.2 RabbitMQ

1. `Order-Created event`
    - EmailService á að geta pikk-að upp Order-Created event-ið sem OrderService sendir frá sér þegar kaup eru stofnuð.
    - Þegar EmailService pikk-ar þetta event upp þá er sent email á bæði kaupandann og seljandann.
    - Email-ið ætti að hafa Subject-ið sem "Order has been created"
    - Email body-ið ætti að innihalda id-ið á kaupunum, nafn-ið á vörunni og verðið á kaupunum (verð á vöru með afslátt).
2. `Payment-Success event`
    - EmailService á að geta pikk-að upp Payment-Success event sem PaymentService sendir frá sér þegar greiðsla tekst.
    - Þegar EmailService pikkar þetta event upp þá er sent email á bæði kaupandann og seljandann.
    - Email-ið ætti að hafa subject-ið sem "Order has been purchased"
    - Email body-ið ætti að segja "Order {order id} has been successfully purchased" (skipta {order id} fyrir actual order id-ið)
3. `Payment-Failure event`
    - EmailService á að geta pikk-að upp Payment-Failure event sem PaymentService sendir frá sér þegar greiðsla tekst ekki.
    - Þegar EmailService pikkar þetta event upp þá er sent email á bæði kaupandann og seljandann.
    - Email-ið ætti að hafa subject-ið sem "Order purchase failed"
    - Email body-ið ætti að segja "Order {order id} purchase has failed" (skipta {order id} fyrir actual order id-ið)
4. EmailService ætti að virka þannig að þegar það eru fleiri en eitt tilvik af EmailService container í gangi þá ættu EmailService-in að skiptast á að taka events af queue-inu, þ.e.a.s að tvö EmailService tilvik process-a aldrei sama event-ið því við viljum ekki að tvö email sendist út um sömu kaupin.

3.10.3 Docker

- EmailService ætti að vera containerized
- RabbitMQ server-inn ætti að vera containerizeds
- EmailService ætti að vera partur af sameiginlegri `docker-compose.yaml` skrá til að start-a upp öllu kerfinu.
- Kerfið ætti að vera start-að upp með docker-compose up –scale EmailService=2 til að kerfið keyri með tvö tilvik af EmailService í gangi.

3.11 InventoryService

3.11.1 API

InventoryService þarf að hafa external API sem notandi kerfisins getur notað til þess að stofna og sækja vörur.

1. `POST /products`
   - Requets-ann á að sjá um að stofna vöru
   - Dæmi um request body-ið er:

```
{
  "merchantId": 123,
  "productName": "some product name",
  "price": 123.0
  "quantity": 20
}
```

   - InventoryService ætti að vista vöruna í gagnagrunn.
   - Ætti að skila **201 HTTP Status Code** með product id-i sem response message.
2. `GET /products/{id}`
   - Ætti að skila **200 HTTP Status Code** með response body sem vöru fyrir tilsvarandi `id`. Varan ætti að vera skiluð út á eftirfarandi json form-i:

```
{
  "merchantId": 123,
  "productName": "some product name",
  "price": 123.0
  "quantity": 20,
  "reserved": 5
}
```

     `quantity` er fjöldi vara sem eru til, `reserved` er fjöldi vara sem er búið að taka frá.
   - InventoryService ætti að skila **404 HTTP Status Code** með villuskilaboðunum "Product does not exist" ef vara með tilsvarandi `id` er ekki til.
3. Leyfilegt er að skilgreina fleiri endapunkta fyrir internal network köll (t.d. OrderService að kalla á InventoryService) ef það er vilji til þess en önnur microservice mega einnig kalla á External endapunktana sem eru skilgreindir að ofan. Þessir auka endapunktar mega vera gerðir meða hvaða network communication sem er (REST, GraphQL, gRPC, RabbitMQ etc).
   - Til dæmis þarf að vera einhver virkni til að OrderService geti tekið frá vöru.

3.11.2 Database

`InventoryService` ætti að hafa persistent gagnagrunn til að geyma vörurnar, fjöldi vara in stock og fjöldi vara sem eru frátekin.

3.11.3 RabbitMQ

1. `Payment-Success event`
    - Þegar InventoryService pikkar upp payment-Success event þá ætti hún hætta að taka frá tiltekna vöru og minnka fjöldi vara in stock.
2. `Payment-Failure event`
    - Þegar InventoryService pikkar upp Payment-Failure event þá ætti hún að hætta að taka frá tiltekna vöru.

3.11.4 Docker

- InventoryService ætti að vera containerized
- Inventory Gagnagrunnurinn ætti að vera containerized
- RabbitMQ server-inn ætti að vera containerized
- Það ætti að vera hægt að kalla á `InventoryService` API-inn locally á port-i 8003. T.d. ætti að vera hægt að senda http request-una `GET http://localhost:8003/products/5` til að sækja vöru með id 5
- `InventoryService`, Inventory Gagnagrunnurinn og `RabbitMQ` ættu að vera partur af sameiginlegri `docker-compose.yaml` skrá til að start-a upp öllu kerfinu.

3.12 Bónus API-Layer

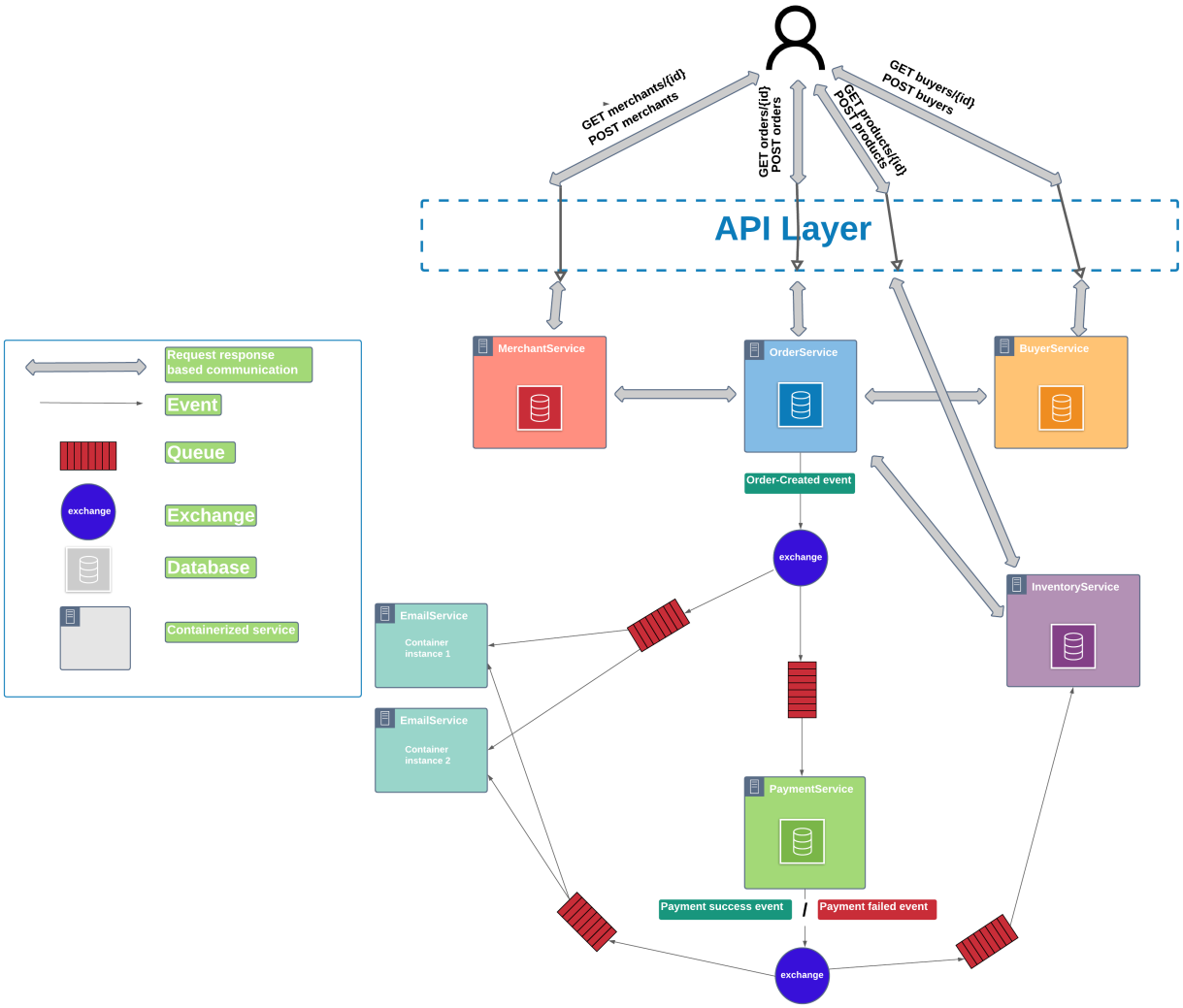
Eins og er þá er notandi kerfisins að kalla beint á microservice-in, bætið við API-layer á milli microservice kerfisins og notandans.

API layer-ið á að vera skilgreind með REST endapunktum, hvaða framework er notað er valkvætt. Layer-ið ætti að hafa eftirfarandi endapunkta:

- `GET api/orders/id`
- `POST api/orders`
- `GET api/merchants/id`
- `POST api/merchants`
- `GET api/buyers/id`
- `POST api/buyers`
- `GET api/products/id`
- `POST api/products`

Endapunktarnir ættu að skila og taka við sömu gögnum og tilsvarnadi endapunktar á microservice-unum sjálfum.

API layer-ið ætti að vera containerized og vera partur af docker-compose.yaml skránni. Það ætti eingöngu að vera hægt að kalla á API layer-ið locally á port-i 8000. T.d. `GET http://localhost:8000/api/orders/5` skilar okkur kaupum með id 5. Við ættum ekki lengur að geta kallað á microservice-in locally.



4 Einkunnagjöf

1. 20 stig fyrir rétta útfærslu á OrderService
2. 20 stig fyrir rétta útfærslu á InventoryService
3. 20 stig fyrir rétta útfærslu á PaymentService
4. 20 stig fyrir rétta útfærslu á EmailService
5. 10 stig fyrir rétta útfærslu á MerchantService
6. 10 stig fyrir rétta útfærslu á BuyerService
7. 10 bónus stig fyrir rétta útfærslu á API Layer-inu