FHV
Vorarlberg University
of Applied Sciences

Contemporary Software Development

Andrea Janes
FHV Vorarlberg University of Applied Sciences
andrea.janes@fhv.at

FHV
Vorarlberg University
of Applied Sciences

Project

Andrea Janes
FHV Vorarlberg University of Applied Sciences
andrea.janes@fhv.at

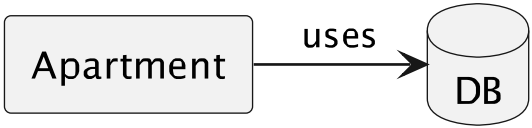
Important videos to re-watch

- Martin Fowler: Microservices, GOTO Conferences (GOTO 2014), https://www.youtube.com/watch?v=wgdBVIX9ifA&t=1078s
- Martin Fowler: Introduction to NoSQL, GOTO Conferences (GOTO 2012), https://www.youtube.com/watch?v=qI_g07C_Q5I
- Martin Fowler: The many meanings of event-driven architectures, GOTO Conferences (GOTO 2017), https://www.youtube.com/watch?v=STKCRSUsyP0

Idea

- We want to try out many technologies we have seen during the lecture:
  - Microservices
  - Docker
  - CQRS
  - Message Queues
  - Event-driven architectures
  - Actor model
  - Event sourcing
- I suggest an order of implementation, but you are free to change it if you want.

Step 1: Implement Apartment



- Implement the following API for `Apartment`:
  - `/add?name=(string)&address=(string)&noiselevel=(number)&floor=(num`
  - `/remove?id=(apartement id)`
  - `/list` (lists all apartments)
- Use UUIDs as IDs in the various tables.
- As DB I suggest SQLite but you can use anything you want.
- The following `docker-compose.yml`, `app.py`, `dockerfile`, and `requirements.txt` represent a possible starting point for a microservice.

docker-compose.yml

```
version: "3.9"
services:
  apartments:
    build: ./apartments
    ports:
      - "5001:5000"
```

apartments/app.py

```
from flask import Flask
import requests

app = Flask(__name__)

@app.route('/')
def hello():
    return "This is the apartment microservice!"
```

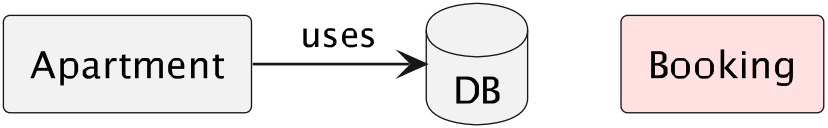
apartments/dockerfile

```
FROM python:3.9-alpine
WORKDIR /home
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

apartments/requirements.txt

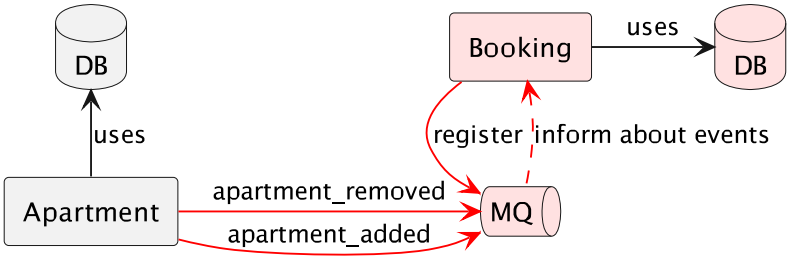
```
flask
requests
```

Step 2: Implement Booking



- Implement the following API for `Booking`:
  - `/add?apartment=(apartment id)&from=(yyyymmdd)&to=(yyyymmdd)&who=(string)`
  - `/cancel?id=(booking id)`
  - `/change?id=(booking id)&from=(yyyymmdd)&to=(yyyymmdd)`
  - `/list` (lists all bookings)
  - Booking should not directly call Apartment to find out if an apartment exists. This is why you need to add a message queue (see next slide).
  - Changes should only work if the apartment is available during the new time frame.

Step 2: Implement a message queue

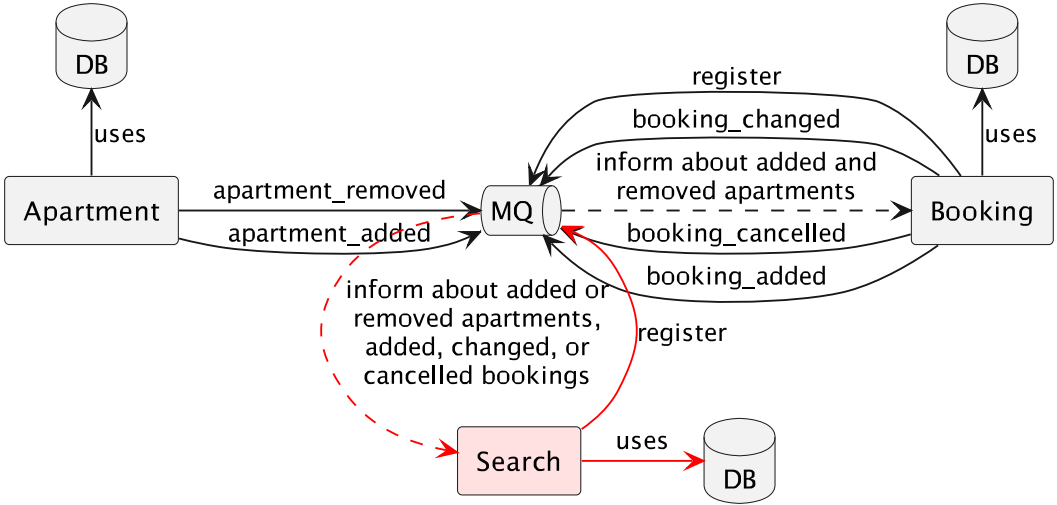


- Whenever an appartment is added or removed, the event is posted to the queue and `Booking` gets these events, too (because it registers for apartment events).
- `Booking` keeps a copy of the relevent data about apartments (it needs only the list of existing apartments) locally.
- When a booking occurs, the service does not need to contact `Apartment`.

Step 3: Implement Search

- `Search` needs to know which apartments exists and which ones are available.
- This is why also booking needs to post events about bookings and cancellations to the message queue so that search can keep a copy of the data it needs to perform a search without contacting the other services.
- Implement the following API for `Search`:
  - `/search?from=(yyyymmdd)&to=(yyyymmdd)`
  - Search should not directly call the other services to find out if an apartment exists and if it is available.

Step 3: Implement Search

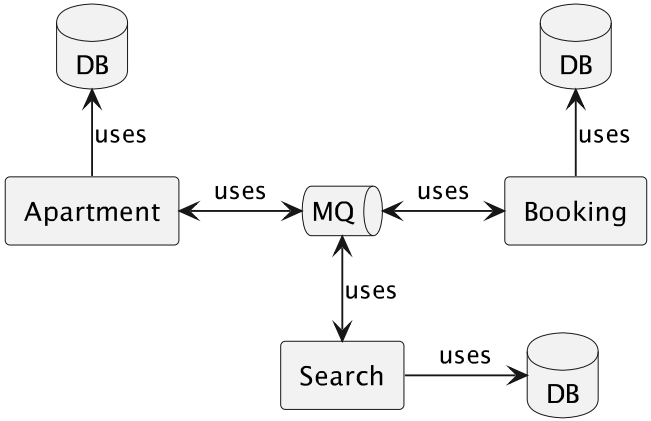


Step 4: Implement Initialization

- If a new service is deployed, it cannot know the past events.
- To handle such cases, implement
  - a direct call from `Booking` to `Apartment`, to initialize its list of apartments, in case this table is empty.
  - a direct call from `Search` to `Apartment`, to initialize its list of apartments, in case this table is empty.
  - a direct call from `Search` to `Booking`, to initialize its list of bookings, in case this table is empty.
- To implement the direct calls, you can use the `/list` endpoints already implemented.

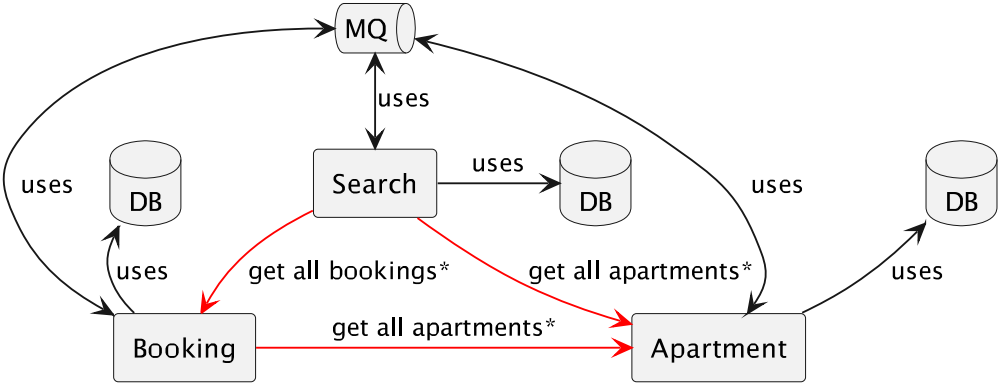
Step 3: Implement Initialization

- To make the diagram less complex, I replace all interactions with the message queue as “uses”



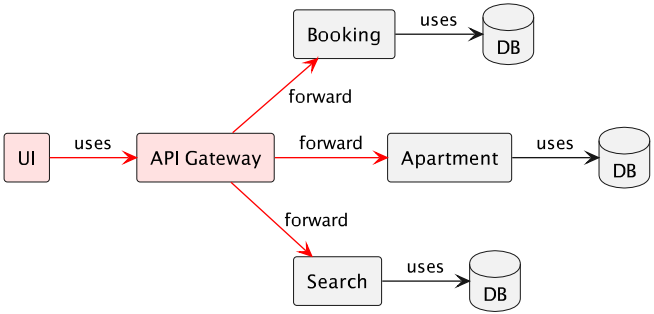
Step 3: Implement Initialization

- In this diagram I show the initialization calls.



Step 4: Implement an API gateway

- UI developers do not want to directly access backend services.
- Develop an API gateway that simply forwards all calls to the right services.



Step 4: Implement an API gateway

- Implement the API for `API Gateway` so that:
  - Calls starting with `/apartment` are forwarded to the `Apartment` microservice.
  - Calls starting with `/booking` are forwarded to the `Booking` microservice.
  - Calls starting with `/search` are forwarded to the `Search` microservice.
- For example, adding an apartment would than be:
  `/apartment/add?name=...&address=...&noiselevel=...&floor=...`

Optional step 5: Implement Event Sourcing

- Change the `Booking` service so that bookings are stored using event sourcing, i.e., added, changed, and cancelled bookings are stored using the event sourcing technique.
- You might use https://eventsourcing.readthedocs.io/en/stable/.
- Add the following API for `Booking`:
  - `/rollback?to=(booking id)`, which deletes all events until the bookingid (which is not deleted) and restores the state of booking at that time.
- Add the necessary events so that the other services can update their local data stores accordingly.

Test cases

- Here are some URLs that describe how the interaction with such system should work:
  - /apartment/add?name=A&address=Bolzano&noiselevel=4&floor=0
  - /apartment/add?name=B&address=Merano&noiselevel=0&floor=2
  - /apartment/add?name=C&address=Trento&noiselevel=1&floor=1
  - /booking/add?apartment=(id of apartment A)&from=20240101&to=20240201&who=Matteo
  - /booking/add?apartment=(id of apartment B)&from=20240301&to=20240307&who=Paola
  - /booking/change?id=(id of previous booking)&from=20240301&to=20240308
  - /booking/cancel?id=(id of previous booking)