Microservices and Docker Compose

Alessandro Bocci

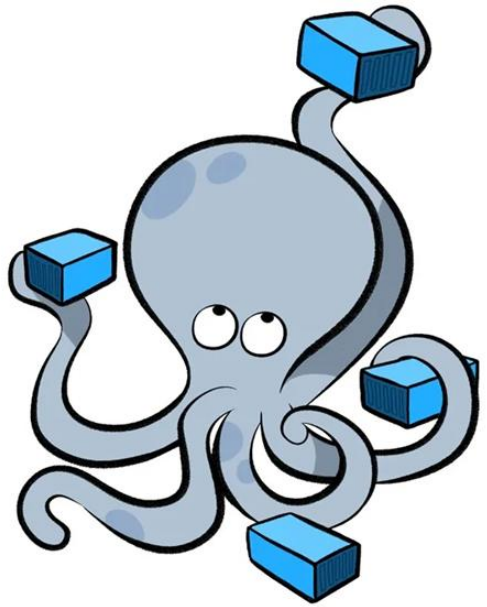
name.surname@unipi.it

Advanced Software Engineering (Lab)

24/10/2025

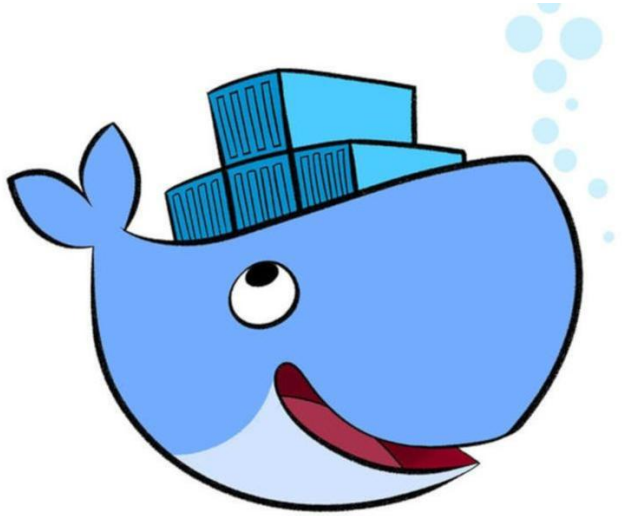
What will you do?

- Split the monolith of previous labs into a microservice architecture.
- Use Docker Compose to run the microservice architecture.



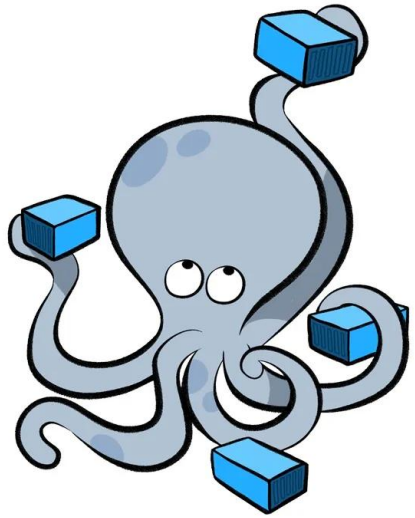
Software Prerequisites

- Docker Engine
- Docker image `python:3.12-slim`
- (Bonus stage) Docker image `redis:7-alpine`



Docker compose

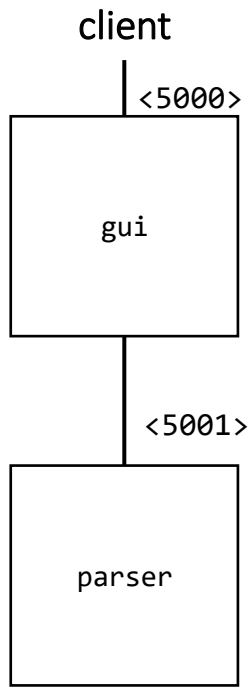
A tool for running multi-container applications on Docker defined using the Compose file format.
Suitable for single host deployment.



Docker compose

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
       - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```

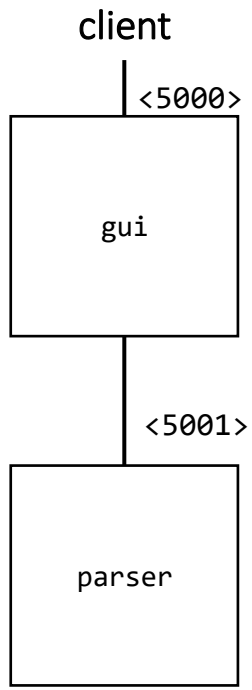


Docker compose

Declaration of services

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



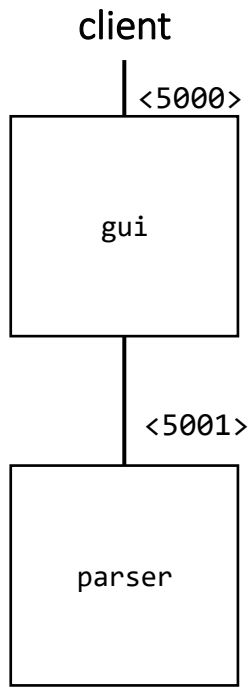
Docker compose

Two services:

- Parser
- Gui

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```

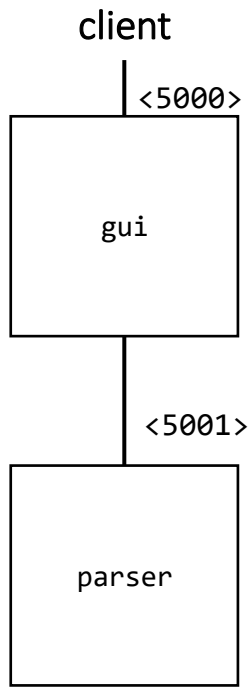


Docker compose

For a service I can declare:

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
       - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```

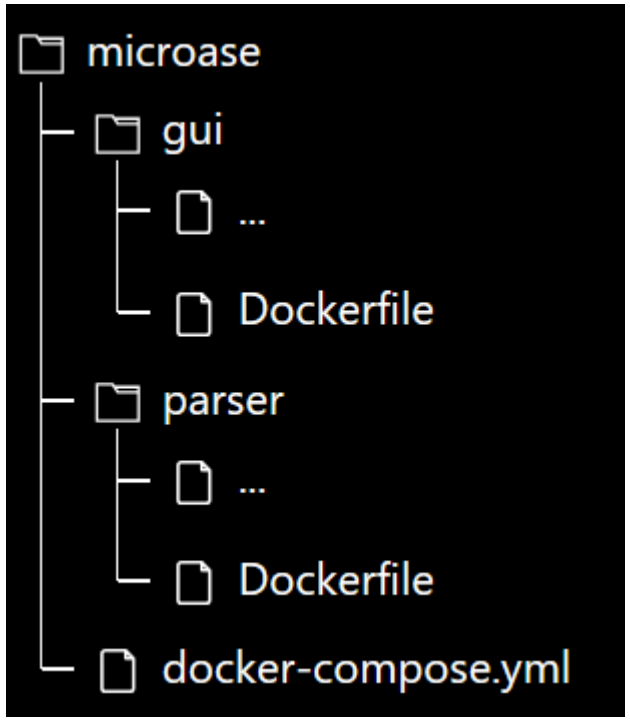


Docker compose

For a service I can declare:

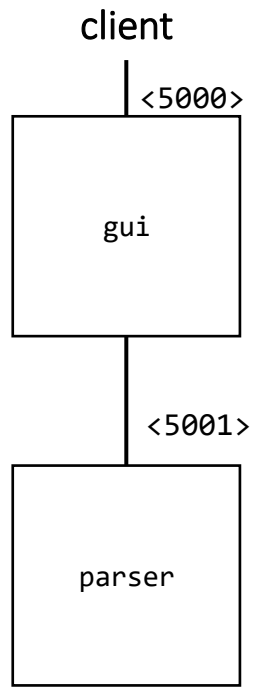
- Folder of the Dockerfile

  a.k.a. the building context



YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



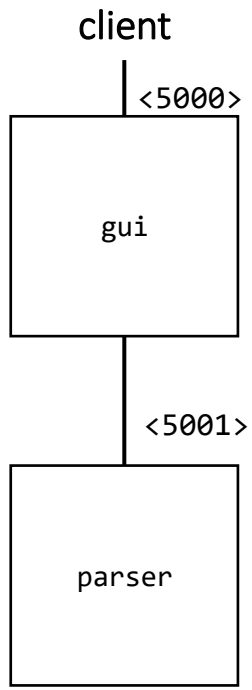
Docker compose

For a service I can declare:

- Folder of the Dockerfile
- Container Name

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



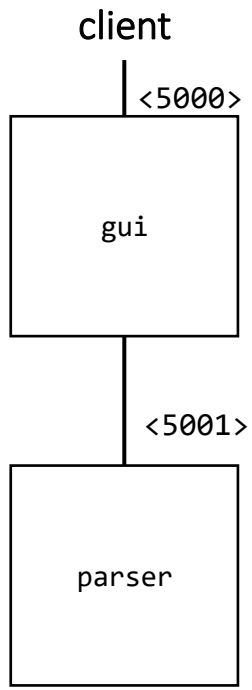
Docker compose

For a service I can declare:

- Folder of the Dockerfile
- Container Name
- Policy on failure

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



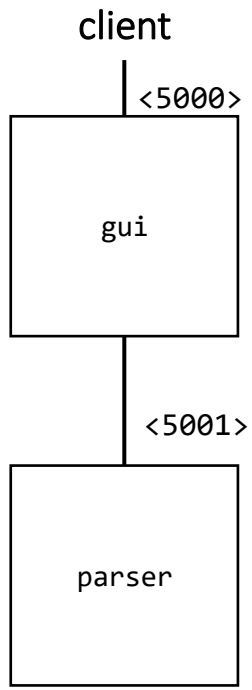
Docker compose

For a service I can declare:

- Folder of the Dockerfile
- Container Name
- Policy on failure
- Volumes mount

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



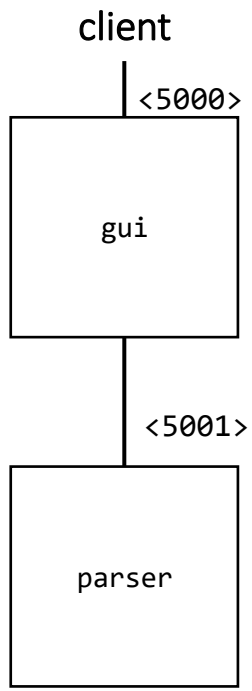
Docker compose

For a service I can declare:

- Folder of the Dockerfile
- Container Name
- Policy on failure
- Volumes mount
- Port bindings

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



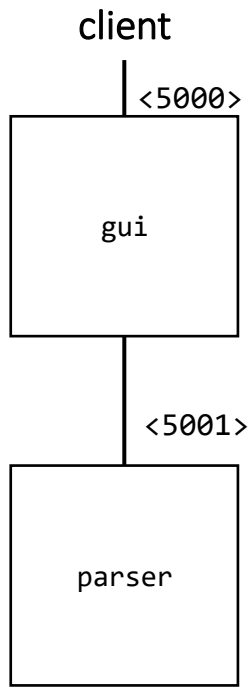
Docker compose

For a service I can declare:

- Folder of the Dockerfile
- Container Name
- Policy on failure
- Volumes mount
- Port bindings
- Service dependencies

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



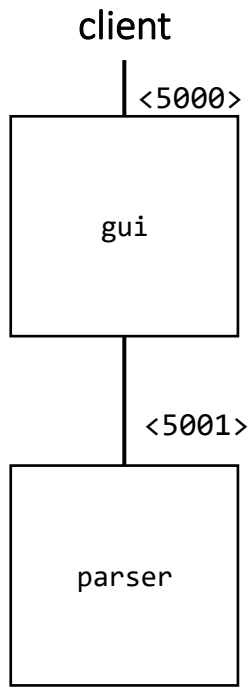
Docker compose

For a service I can declare:

- Folder of the Dockerfile
- Container Name
- Policy on failure
- Volumes mount
- Port bindings
- Service dependencies
- … and many other things

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



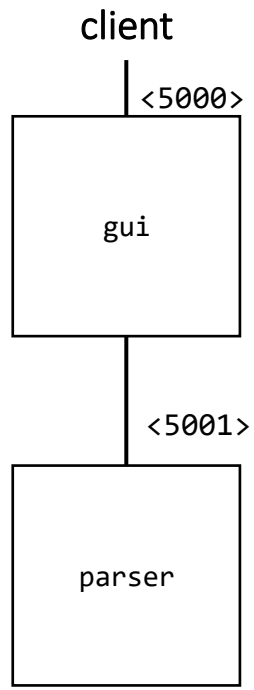
https://docs.docker.com/compose/

Docker compose

This is a ‘named volume’, it is fully managed by compose to persist data

YAML File structure

```
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```

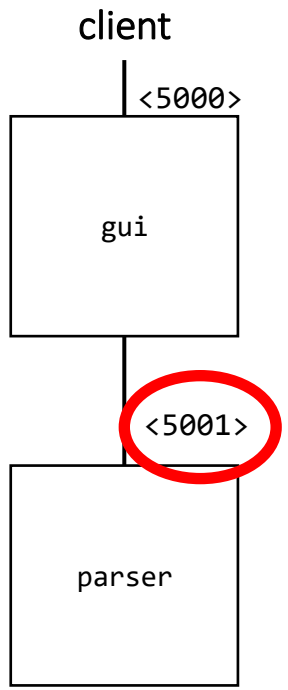


Docker compose

What about the port of the `parser` service?

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
       - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



Docker compose

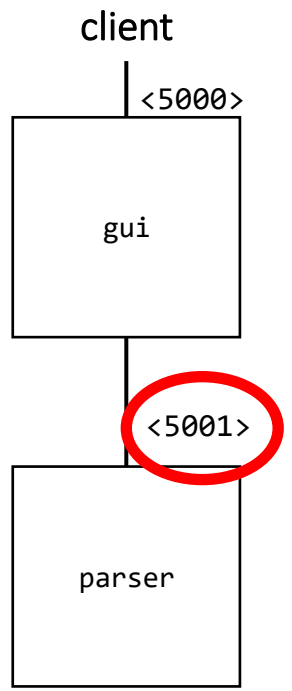
Docker compose creates a default docker network which links all the containers' exposed ports.

In the file, you can define your networks to group the containers of the application in different networks.

The **parser** service exposes port 5001 in its Dockerfile.

YAML File structure

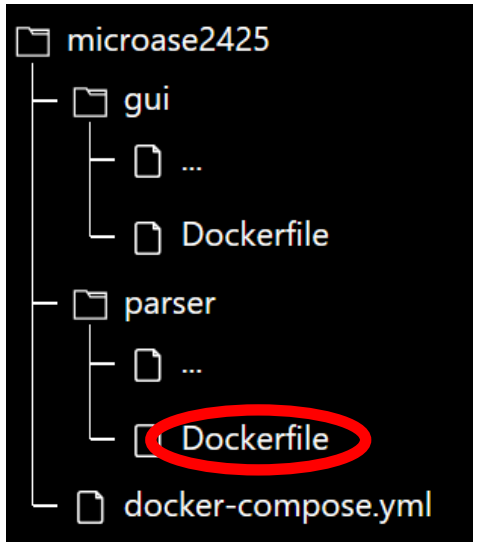
```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```



Docker compose

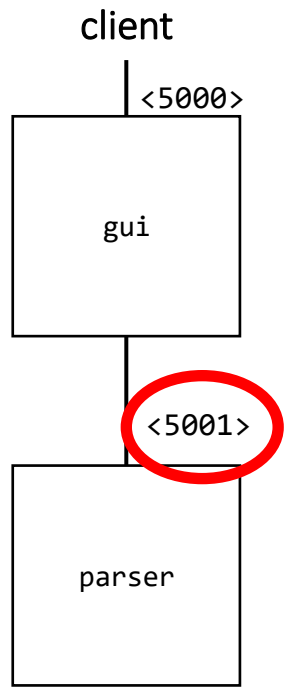
Moreover, you can reach other services using their name.
For instance, the **gui** service can send an HTTP request to the **parser** API to

http://parser:5001



YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
      - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```

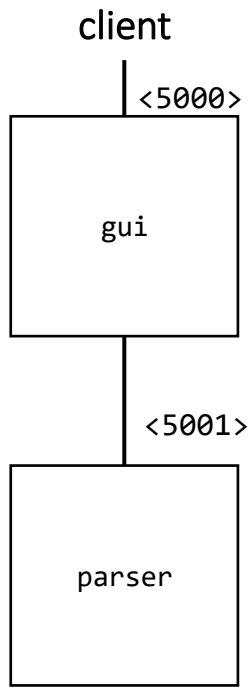


Docker compose

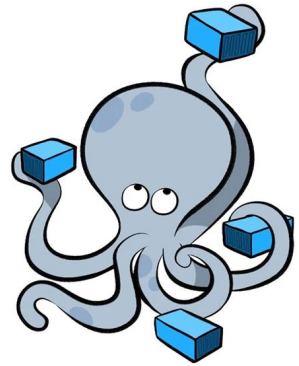
Launching the services is as easy as
`docker compose up`
which will build the images, create volumes, networks and run the containers.

YAML File structure

```yaml
services:
  parser:
    build: ./parser
    container_name: parser
    restart: always
    volumes:
       - myvolume:/app:rw

  gui:
    build: ./gui
    container_name: gui
    ports:
      - 5000:5000
    depends_on:
      - parser
volumes:
    myvolume:
```

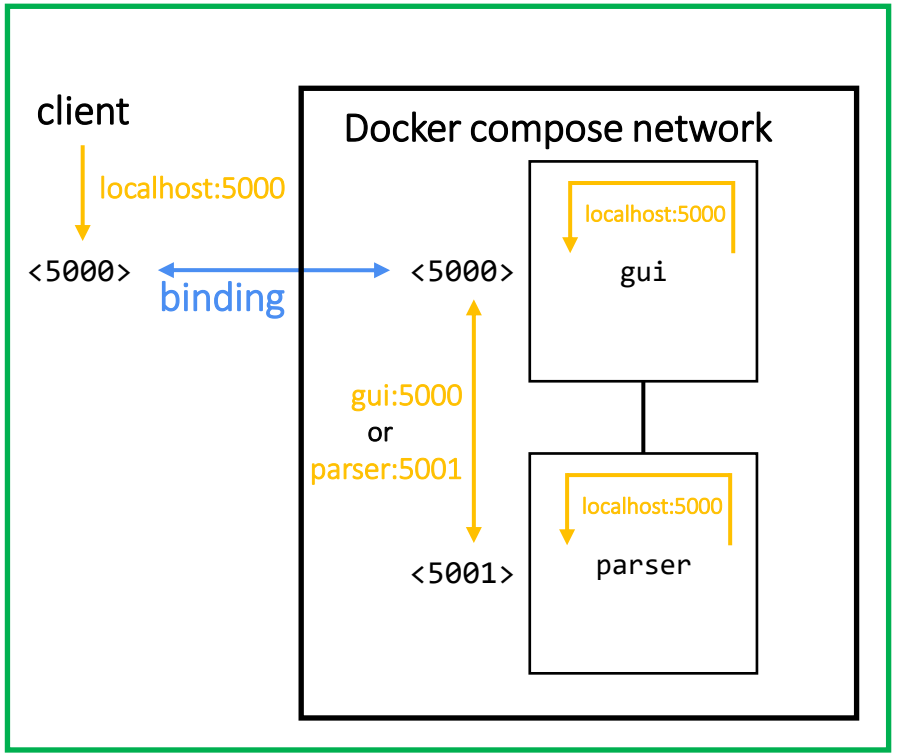


About Docker compose network



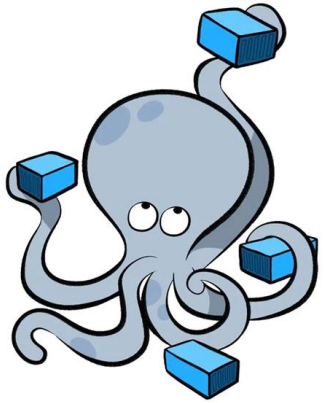
- **`parser`** and **`gui`** can communicate using their exposed port (5000 and 5001) and their name, e.g. http://parser:5001
- **`parser's`** port is bound with the 5000 port of the machine, using **`ports`** with **`5000:5000`**. It is equivalent to running a container with **`-p 5000:5000`**.
- From outside the docker compose network, ports 5000 and 5001 of the microservices are not reachable directly. A client can send requests only to port 5000 of the machine, and the request will be forwarded to the 5000 of **`gui`** because the ports are bound.
- From the code of **`gui`** and **`parser`**, using **`127.0.0.1`** (localhost) means the localhost of the container, not the localhost of the local machine.
- Note that the ports are unique per machine or per container. This means that the **`parser`**'s exposed port can be 5000 without problems, and the machine port can be a different one as well. What you cannot do is bind the same port of your local machine multiple times.

Local machine network



`gui` Dockerfile exposes 5000
`parser` Dockerfile exposes 5001

About Docker compose network



In Dockerfiles and docker compose files:

- Expose the ports you want to reach from within the docker compose network.
- Use port bindings for microservices you want to reach from outside the docker compose network.

In the code (inside the docker compose network):

- Make a service listen to an exposed port.
- To make requests toward other services use their name and the exposed ports.
- The use of localhost refers to the container not the hosting machine.
- If you want to reach a service outside the docker compose network, it is possible in different ways. You can check online how to do it (some solutions are platform dependent).

DOCKER COMPOSE CHEAT SHEET

File

structure

```
services:
   container1:
      properties: values

   container2:
      properties: values

networks:
   network:

volumes:
   volume:
```

Types

value

```
key: value
```

array

```
key:
   - value
   - value
```

dictionary

```
master:
   key: value
   key: value
```

Properties

build

build image from dockerfile in specified directory

```
container:
   build: ./path
   image: image-name
```

image

use specified image

```
image: image-name
```

container_name

define container name to access it later

```
container_name: name
```

volumes

define container volumes to persist data

```
volumes:
   - /path:/path
```

command

override start command for the container

```
command: execute
```

environment

define env variables for the container

```
environment:
   KEY: VALUE
---
environment:
   - KEY=VALUE
```

env_file

define a env file for the container to set and override env variables

```
env_file: .env
---
env_file:
 - .env
```

restart

define restart rule (no, always, on-failure, unless-stopped)

```
expose:
    - "9999"
```

networks

define all networks for the container

```
networks:
   - network-name
```

ports

define ports to expose to other containers and host

```
ports:
    - "9999:9999"
```

expose

define ports to expose only to other containers

```
expose:
    - "9999"
```

network_mode

define network driver (bridge, host, none, etc.)

```
network_mode: host
```

depends_on

define build, start and stop order of container

```
depends_on:
    - container-name
```

Other

idle container

send container to idle state > container will not stop

```
command: tail -f /dev/null
```

named volumes

create volumes that can be used in the volumes property

```
services:
  container:
    image: image-name
    volumes:
      - data-
volume:/path/to/dir

volumes:
  data-volume:
```

networks

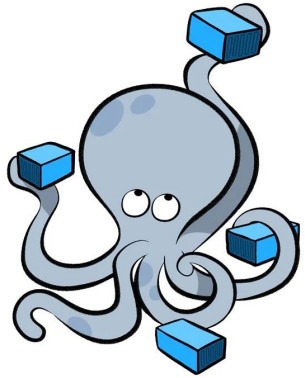
create networks that can be used in the networks property

```
networks:
   frontend:
      driver: bridge
```

https://devopscycle.com/blog/the-ultimate-docker-compose-cheat-sheet/

Docker compose commands

```
Commands:
  build       Build or rebuild services
  config      Parse, resolve and render compose file in canonical format
  cp          Copy files/folders between a service container and the local filesystem
  create      Creates containers for a service.
  down        Stop and remove containers, networks
  events      Receive real time events from containers.
  exec        Execute a command in a running container.
  images      List images used by the created containers
  kill        Force stop service containers.
  logs        View output from containers
  ls          List running compose projects
  pause       Pause services
  port        Print the public port for a port binding.
  ps          List containers
  pull        Pull service images
  push        Push service images
  restart     Restart service containers
  rm          Removes stopped service containers
  run         Run a one-off command on a service.
  scale       Scale services
  start       Start services
  stop        Stop services
  top         Display the running processes
  unpause     Unpause services
  up          Create and start containers
  version     Show the Docker Compose version information
  wait        Block until the first service container stops
  watch       Watch build context for service and rebuild/refresh containers when files are updated
```



Docker compose commands

Be careful with `docker compose up`!
It builds the images only the first time is used.
If you change the code of a service you have to use

```
docker compose build
```

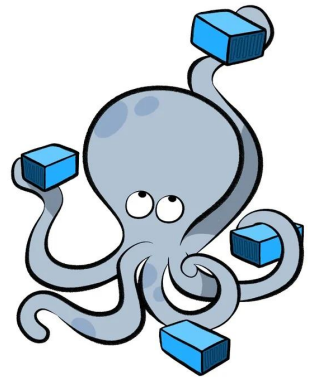
followed by

```
docker compose start
```

Otherwise:

```
docker compose up --build
```

to force the build of the images.



Previously...

Previously...

app.py

add
sub
mul
div
mod
random
reduce
upper
lower
concat
crash
last

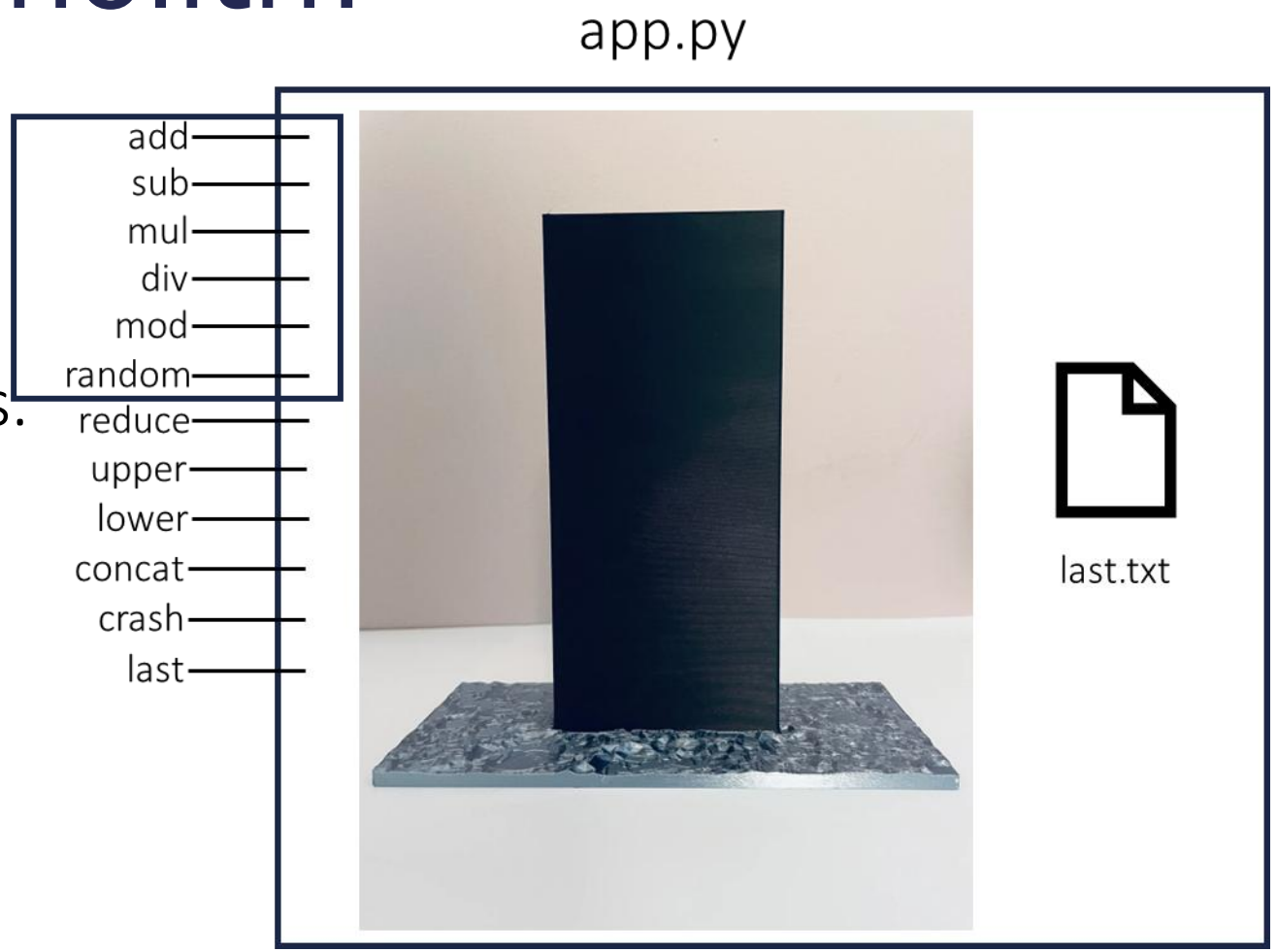
last.txt

Let’s split the monolith!

There is no DB!
Let’s find the seams based on bounded context

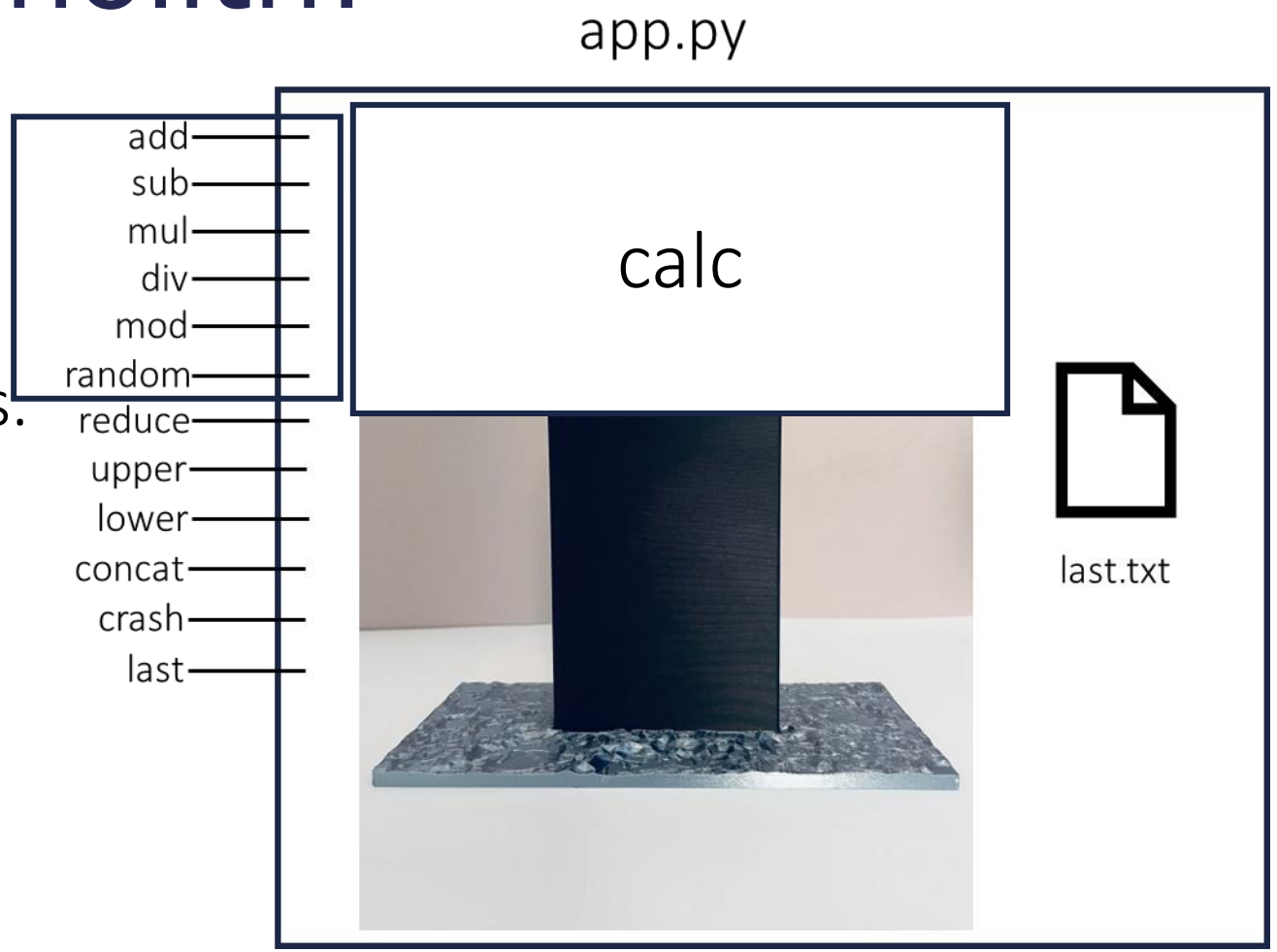
Let’s split the monolith!

Calculator bounded context endpoints.
Let’s extract them in a microservice.

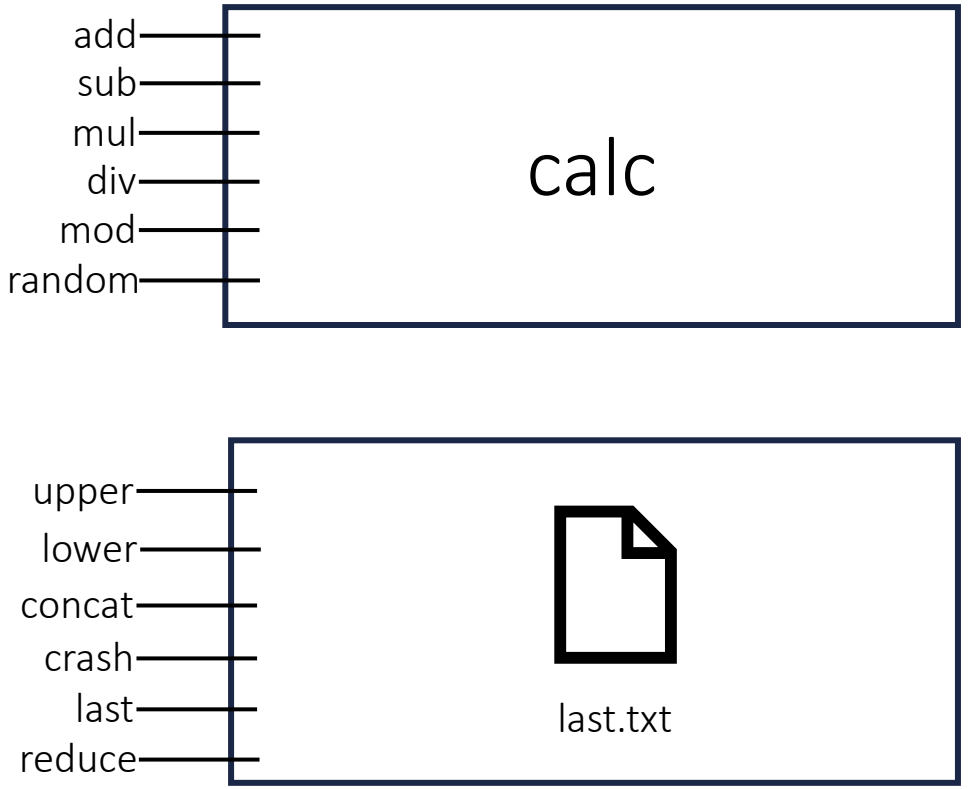


Let’s split the monolith!

Calculator bounded context endpoints.
Let’s extract them in a microservice.

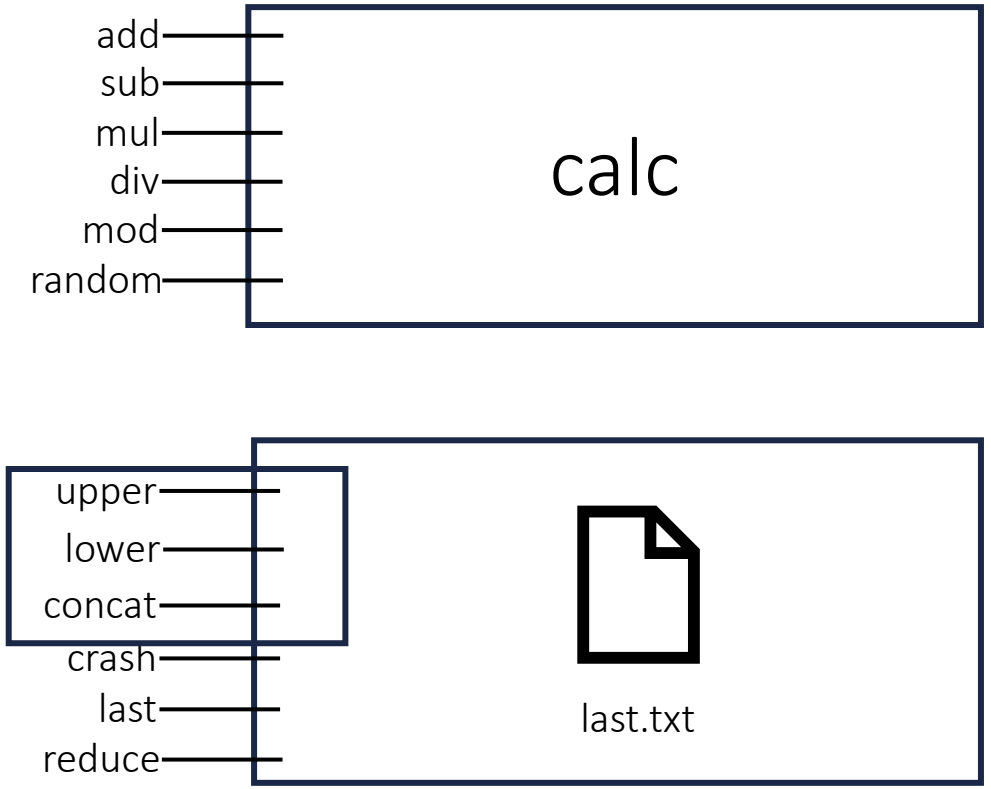


Let’s split the monolith!



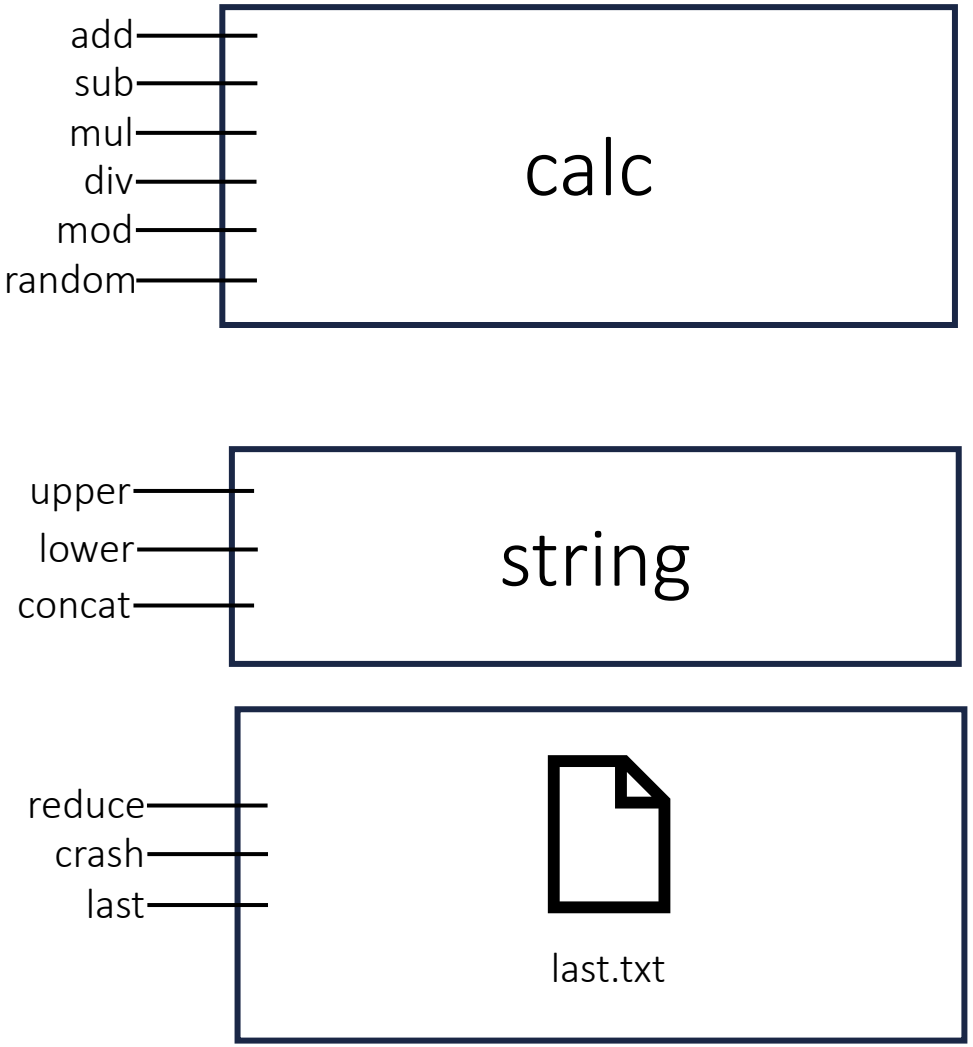
Let’s split the monolith!

String operations bounded context endpoints.
Let’s extract them in a microservice.



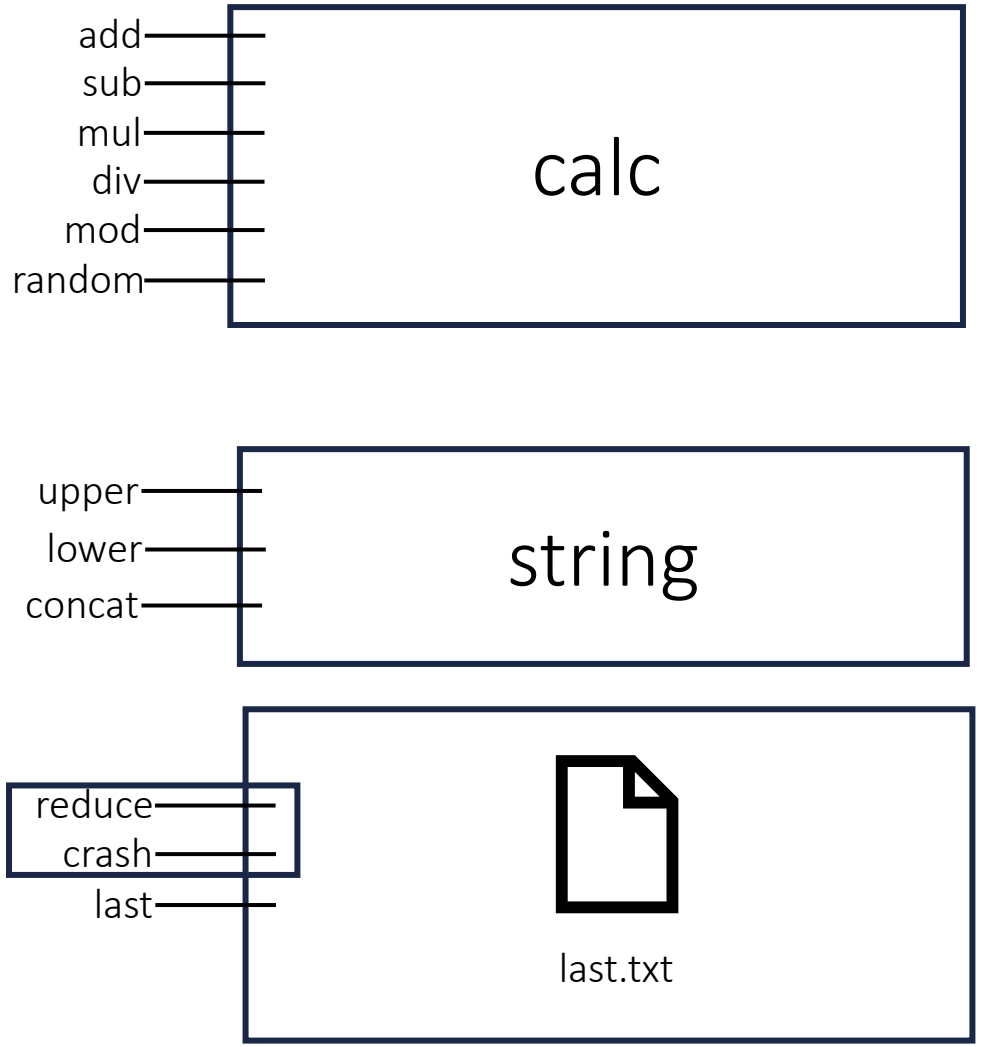
Let’s split the monolith!

String operations bounded context endpoints.
Let’s extract them in a microservice.

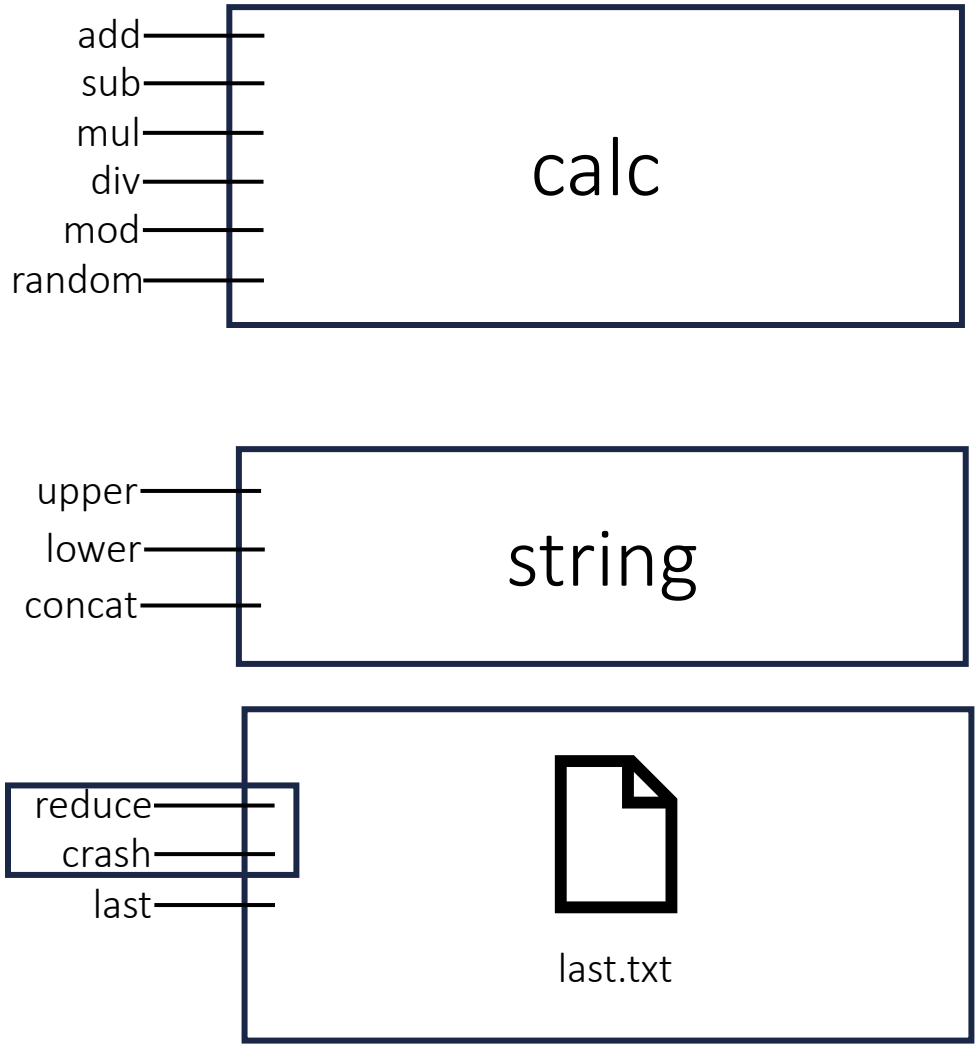


Let’s split the monolith!

What about them?

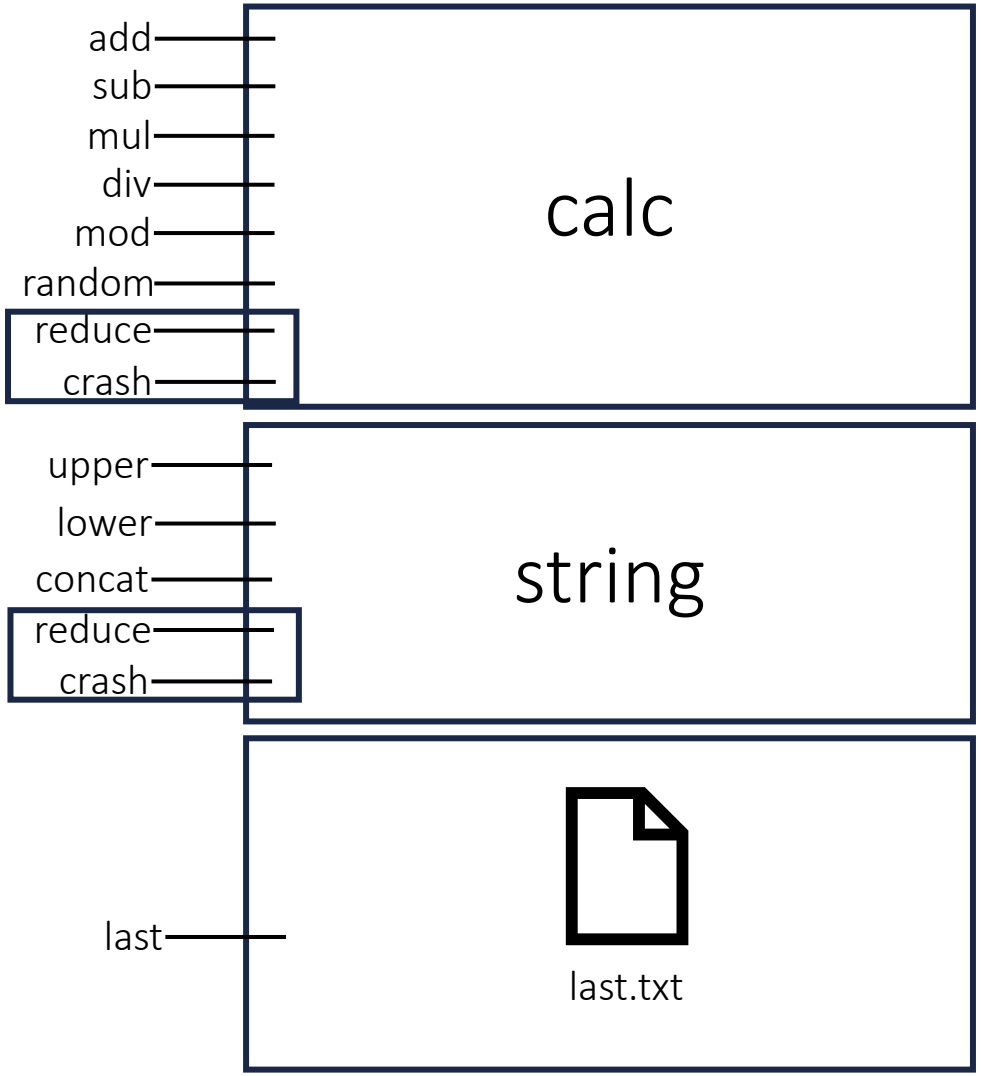


Let’s split the monolith!

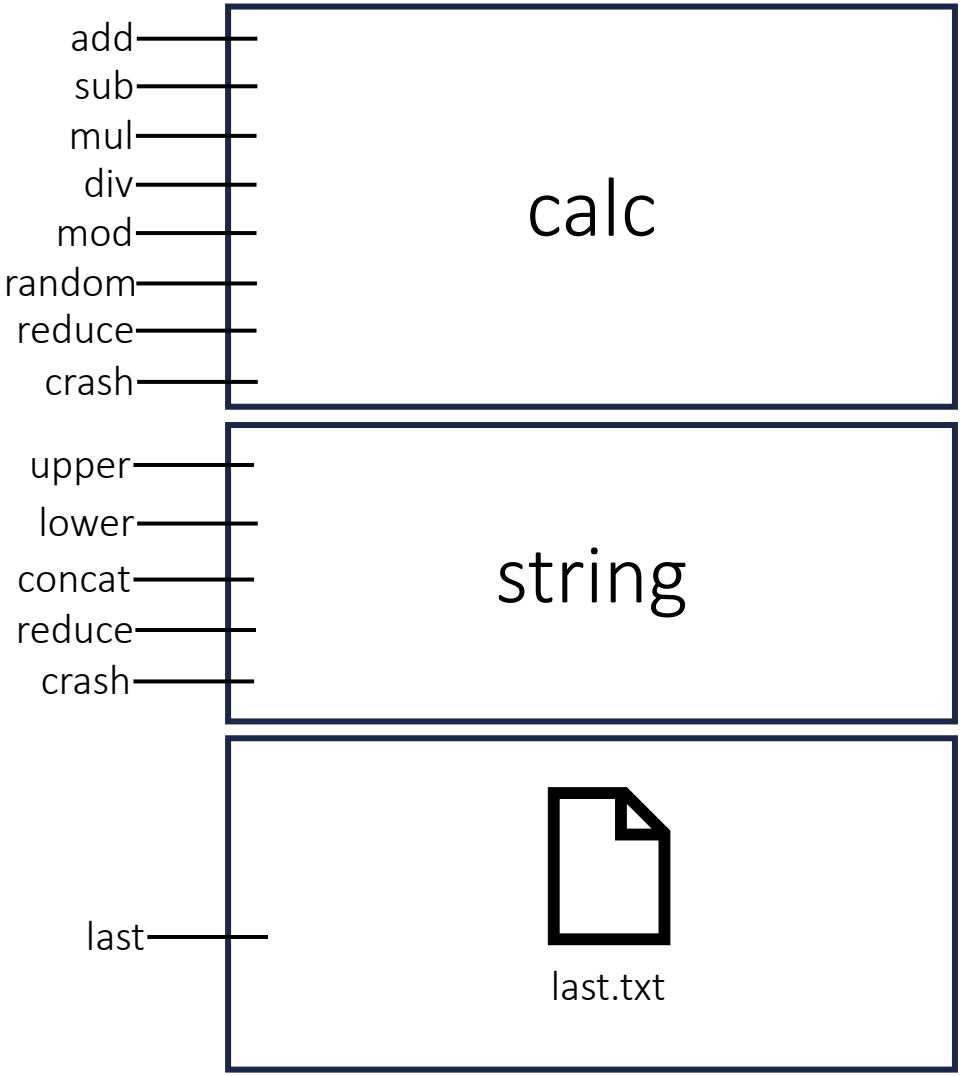


They are common functionalities, let’s split them

Let’s split the monolith!

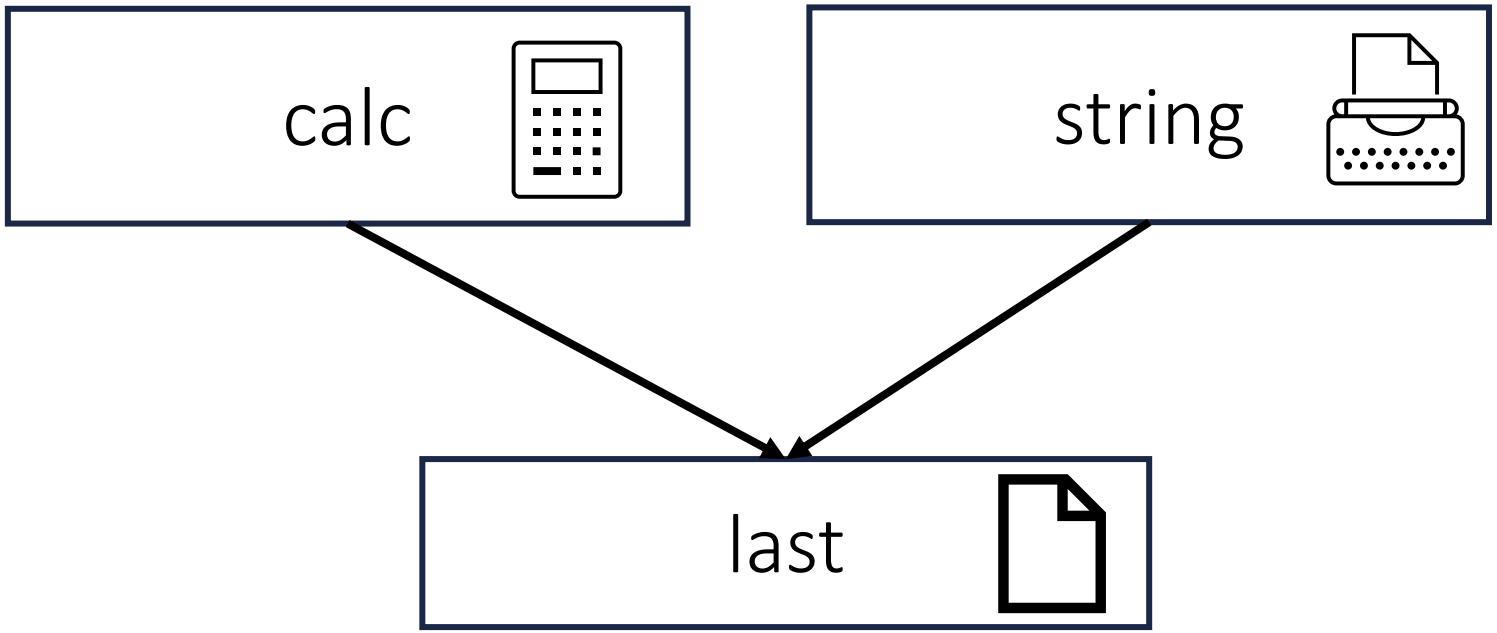


Let’s split the monolith!

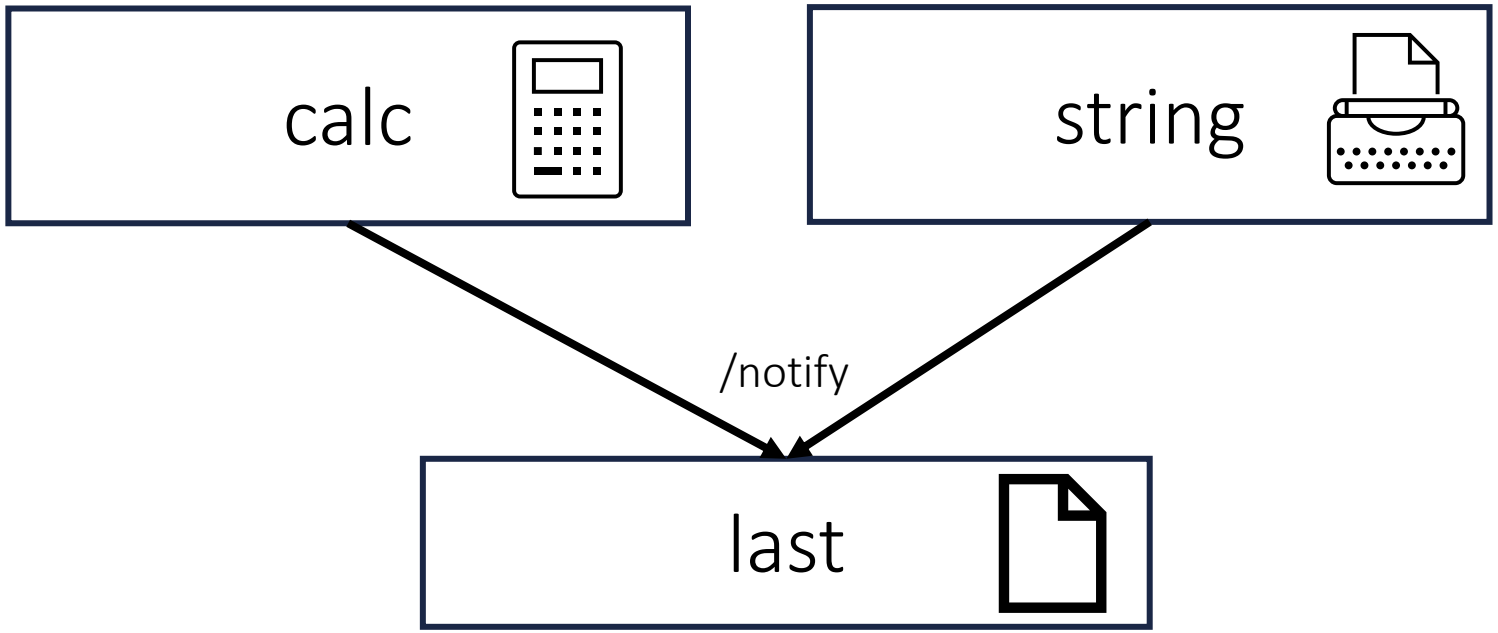


There is no database to split, `last` is to simple as data storage

This is our microservice architecture `microase`

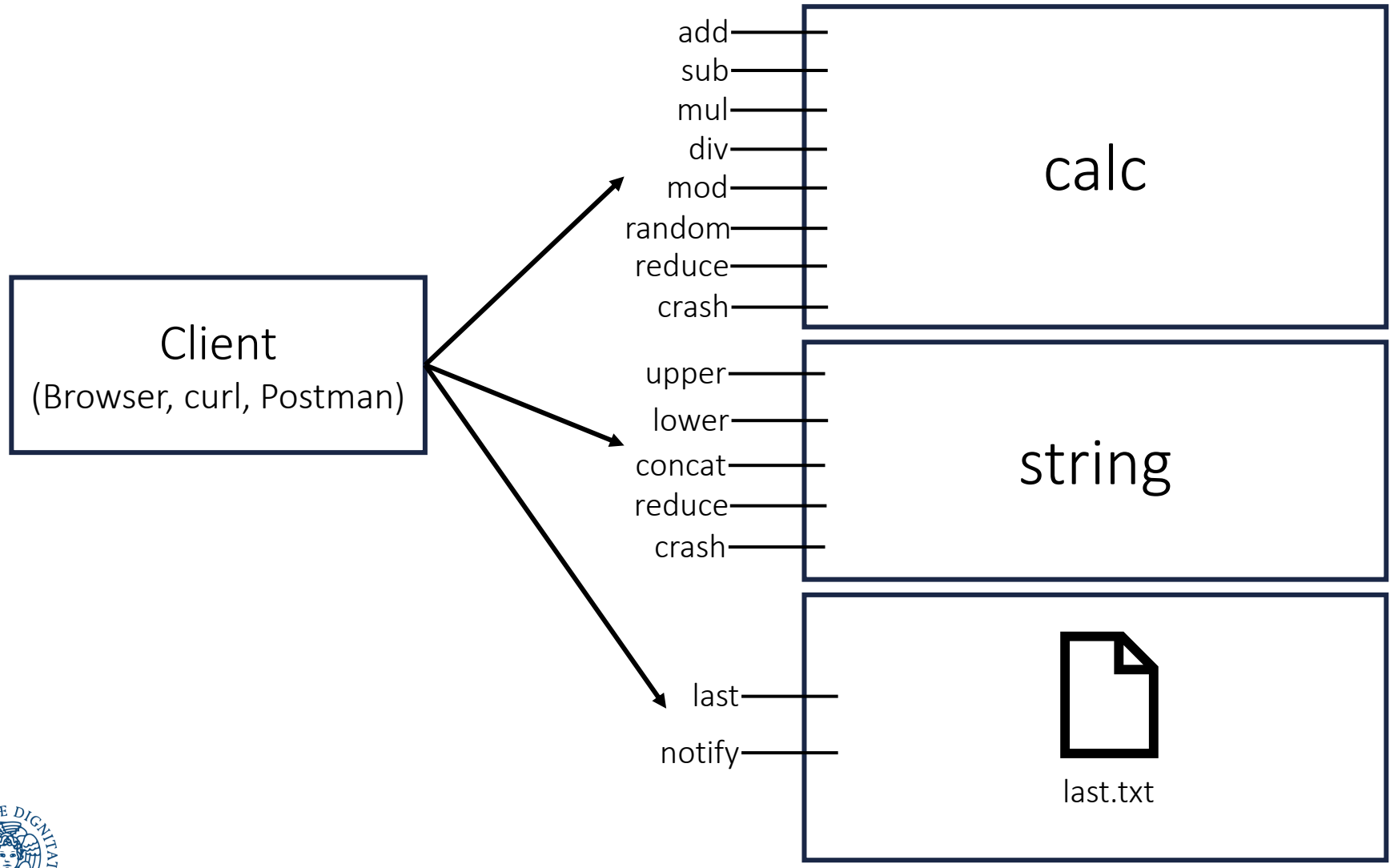


Rember that each operation should be saved

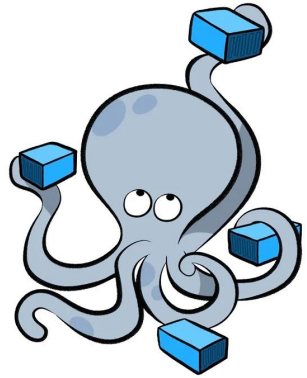


`last` should expose and endpoint to be notified of an operation.
`calc` and `string` should call that endpoint after each operation.
Previoulsy it was a function call inside the monolith, now it is and endpoint invocation.

Final view


Client
(Browser, curl, Postman)
add
sub
mul
div
mod
random
reduce
crash
calc
upper
lower
concat
reduce
crash
string
last
notify
last.txt

It’s your turn!

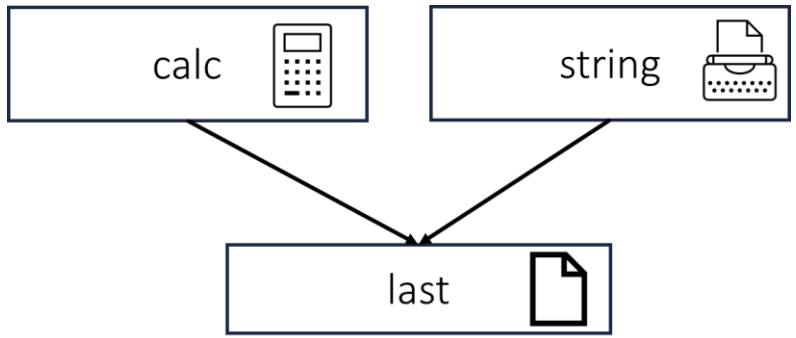


Use the code of Lab 4 (or download it from Moodle).

In three separate folders, create the three Flask microservices: calc, string, and **`last`**.

Those microservices should all be reachable from a client (browser, curl, etc.) using localhost and a port you bind.

1. Most of the code of **`calc`** and **`string`** should be a refactoring of the old code.
2. The **`last`** microservice should expose the old **`/last`** endpoint and a **`/notify`** endpoint, which from a HTTP POST receives a JSON in the format `{''timestamp'': ts, ''op'': op_string}` and save it in a file.
3. Change the **`save_last`** function of **`calc`** and **`string`** from a file **`write`** to an HTTP request (https://requests.readthedocs.io) sending to **`/notify`** the JSON file.
4. Write the Dockerfile of each microservice.
5. In the root folder, write the a **`docker-compose.yml`** file to run the microservice architecture.
6. Try all the microservices invoking their REST API.



About HTTP Requests

You should add the `requests` Python module in the requirements files and import it in the code.

- HTTP GET

```
x = requests.get(URL)
x.raise_for_status()
json_res = x.json()
```

- HTTP POST with JSON payload

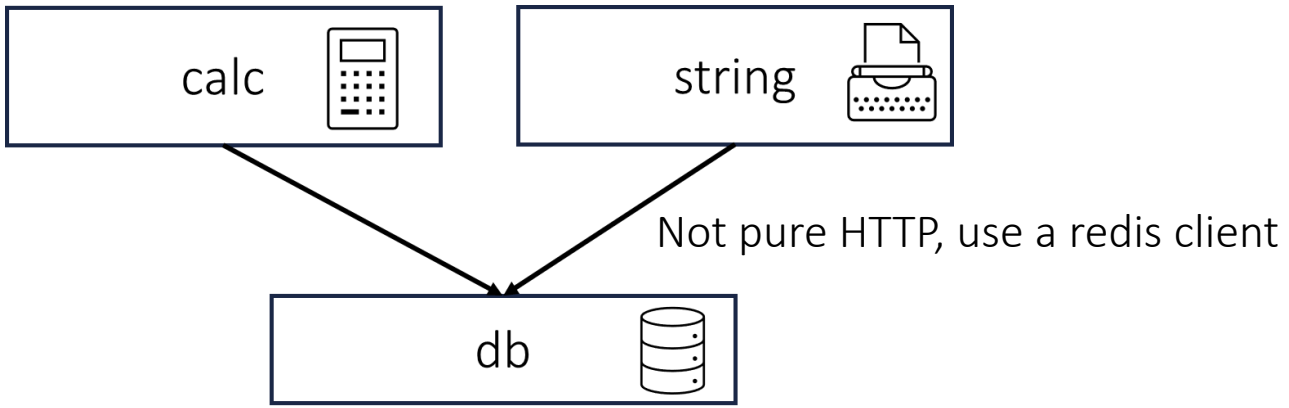
```
x = requests.post(URL,json={'timestamp':str(datetime.now()), 'op':s})
x.raise_for_status()
```

`raise_for_status()` throws an exception if response is not 200 OK

Bonus part!

Change the `last` service to a `db` service using redis.

- Substitute the `last` microservice in the docker compose file with `redis`, specifying the `redis:7-alpine` docker image, `db` as host name and `always` as the restart policy.
- Modify the code of the `calc` and `string` microservices from HTTP requests to the key-values pairs (`timestamp`, `op_dict`), where `op_dict` is a string representing the dict `{"op"=op,"args"=list_args,"res"=res}`.
- Use docker compose to run the microservice application.
- Use the browser to test the endpoints and check the database content.



Redis details

Redis is an in-memory, key-value database.

Those are the Python instructions to set it up and store the values (put them in `calc` and `string` services):

```python
#Redis client import
import redis

#DB connection setup (decode_response to have str and not bytes)
r = redis.Redis(host='db', port=6379, decode_responses=True)
#To store a (key, value) pair, value must be bytes, string, int or float
r.set(key, value)
```

https://redis.io/docs/latest/develop/connect/clients/python/redis-py/

This is a possible command to launch redis with docker compose:

```
command: redis-server --save 20 1 --loglevel warning
```

Redis check db content

To check the content of the database you have to use its client.

Retrieve the name of the db container with `docker ps`.

```
alebocci@LaptopUnipiBoc:~$ docker ps
CONTAINER ID   IMAGE            COMMAND                  CREATED          STATUS          PORTS                    NAMES
5bbd8e2d7a30   bonus-string     "/bin/sh -c 'flask r…"   11 minutes ago   Up 11 minutes   0.0.0.0:5002->5000/tcp   bonus-string-1
58601b7ab119   bonus-calc       "/bin/sh -c 'flask r…"   11 minutes ago   Up 11 minutes   0.0.0.0:5001->5000/tcp   bonus-calc-1
dcca1843c69e   redis:7-alpine   "docker-entrypoint.s…"   11 minutes ago   Up 11 minutes   0.0.0.0:6379->6379/tcp   bonus-db-1
```

To access the redis client run the command

`docker exec -it bonus-db-1 redis-cli`

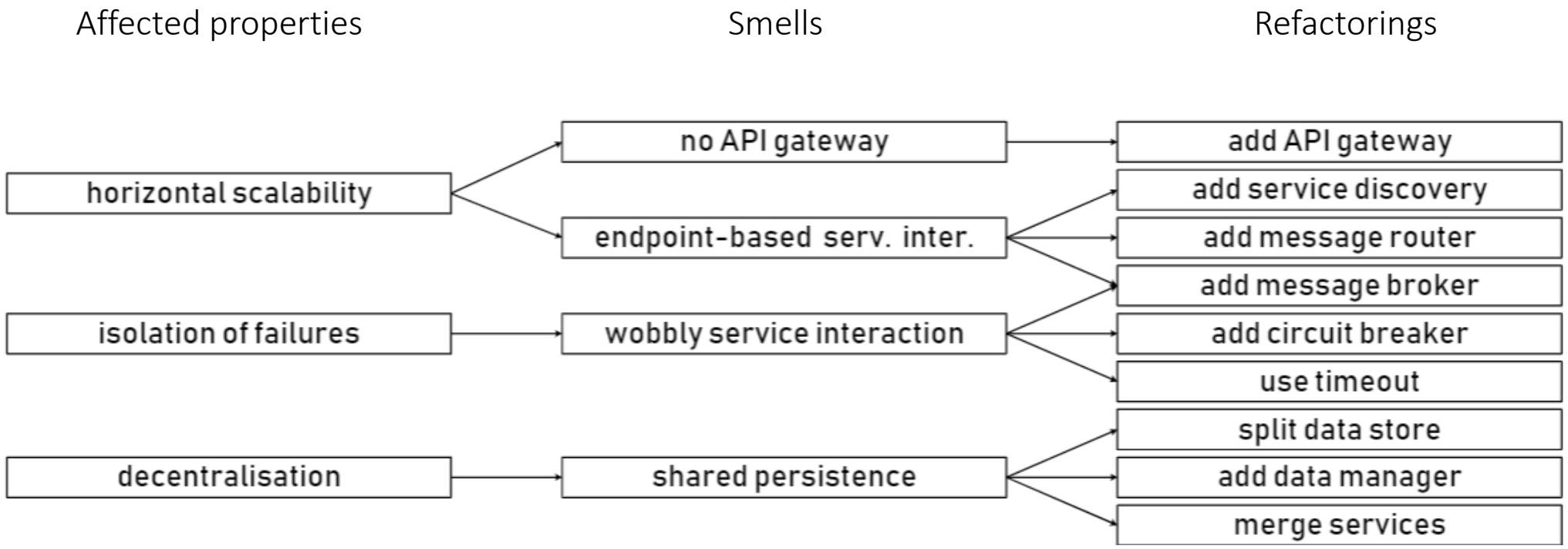
with your container name instead of **bonus-db-1**.

In the client you can type KEYS * to obtain all the keys stored and GET + a key to obtain a value.

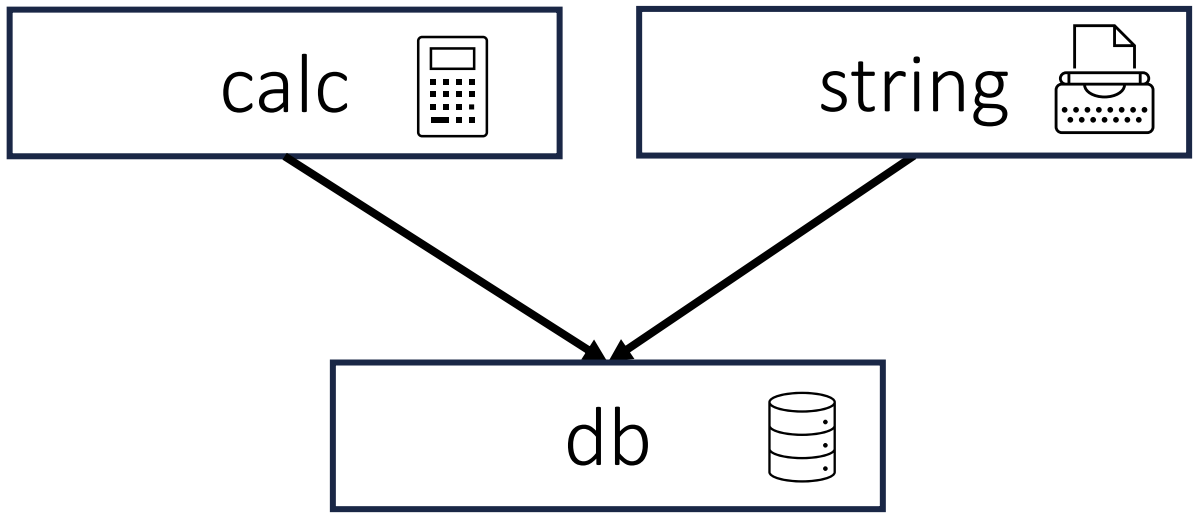
Redis check db content


```
alebocci@LaptopUnipiBoc:~$ docker exec -it bonus-db-1 redis-cli
127.0.0.1:6379> KEYS *
1) "1760970411.3051293"
2) "1760970409.138571"
3) "1760970134.8896759"
127.0.0.1:6379> GET "1760970134.8896759"
"{'op': 'add', 'args': (2.0, 9.0), 'res': 11.0}"
127.0.0.1:6379> quit
```

What about architectural smells?

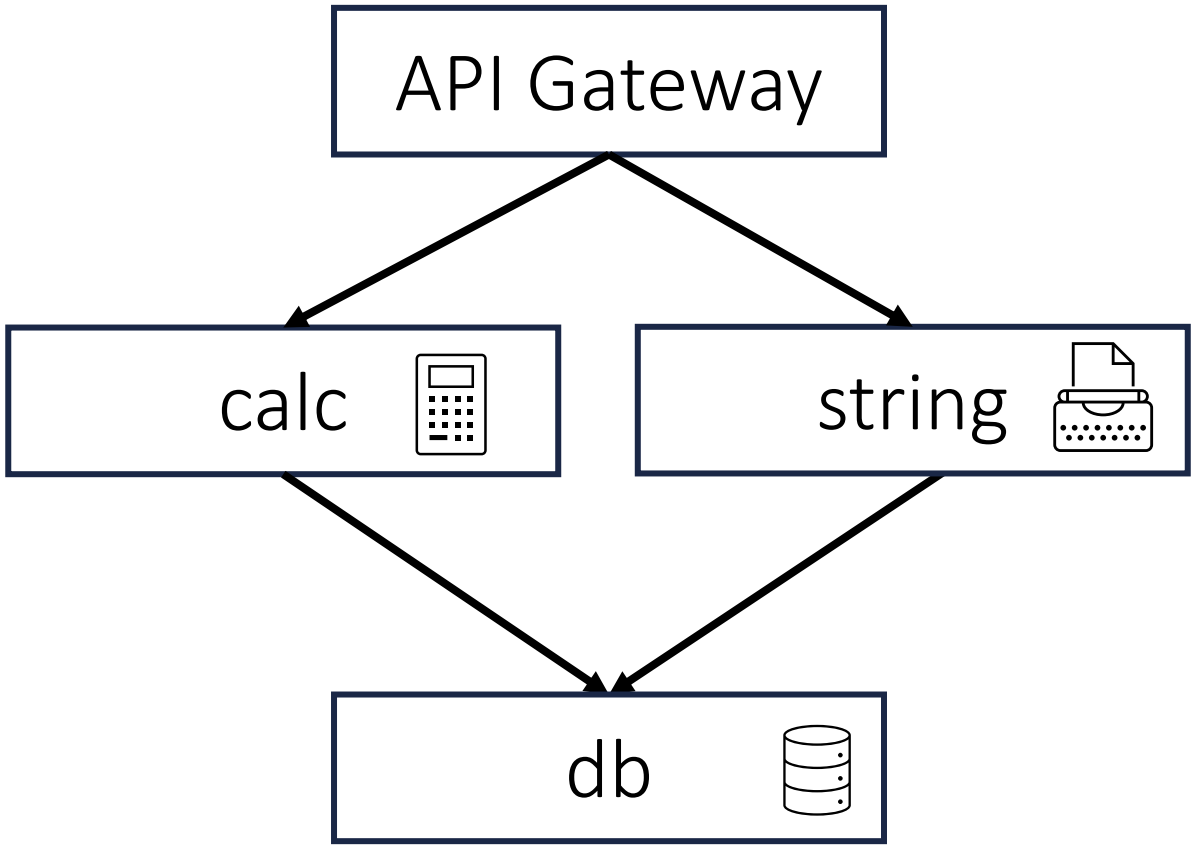


What about architectural smells?



- No API Gateway
- Shared Persistance
- Wobbly Service Interaction
- Endpoint-based Service Interaction

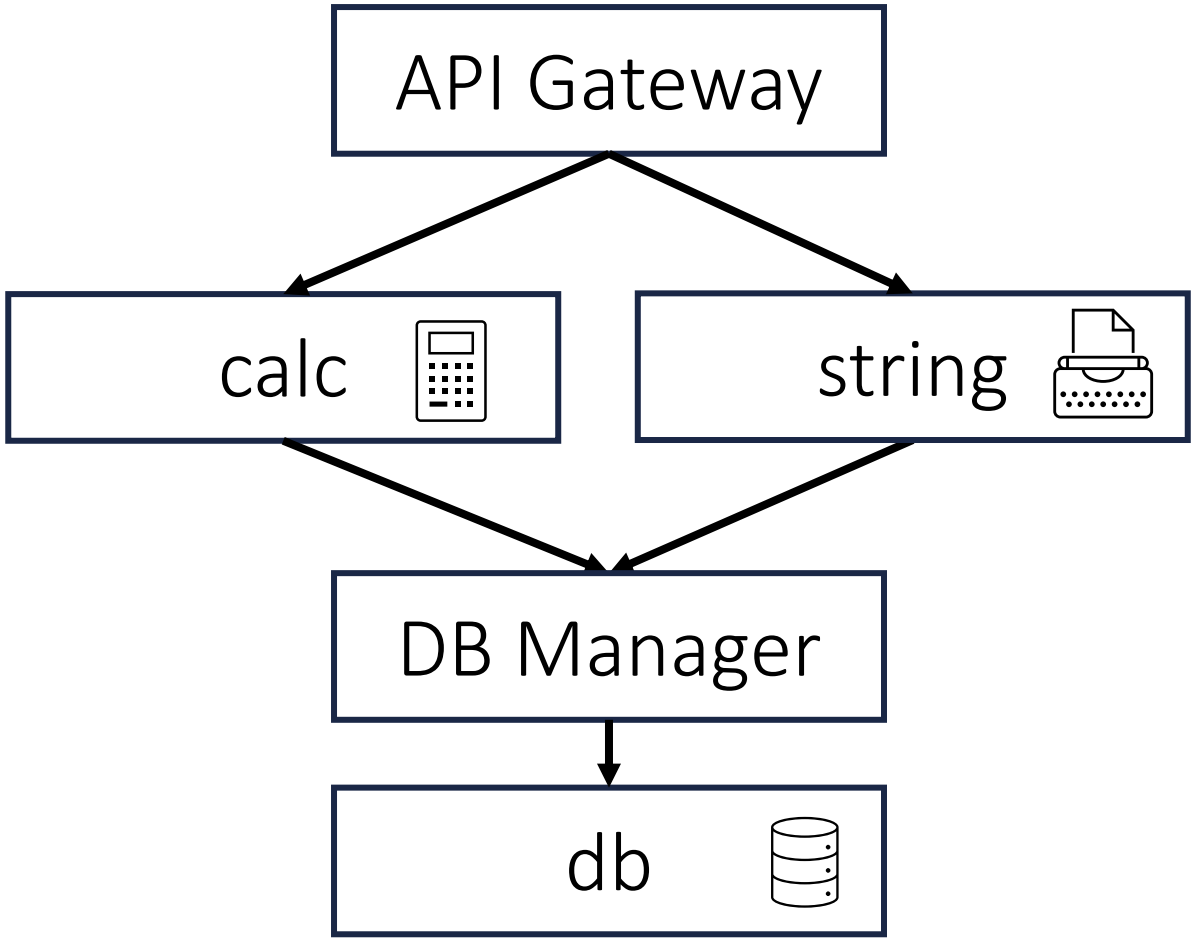
What about architectural smells?



- ~~No API Gateway~~
- Shared Persistance
- Wobbly Service Interaction
- Endpoint-based Service Interaction

Added an API Gateway

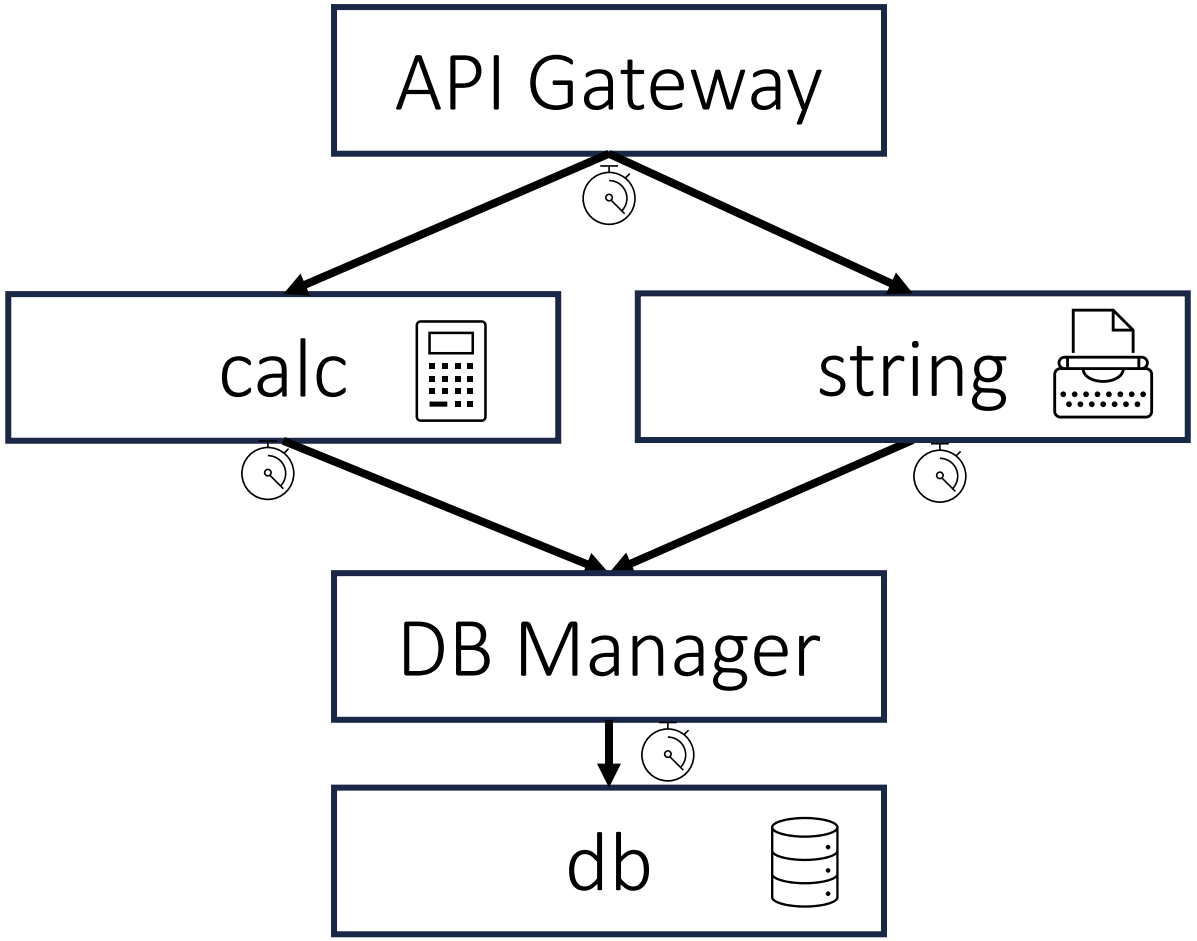
What about architectural smells?



- ~~No API Gateway~~
- ~~Shared Persistance~~
- Wobbly Service Interaction
- Endpoint-based Service Interaction

Added a DB Manager

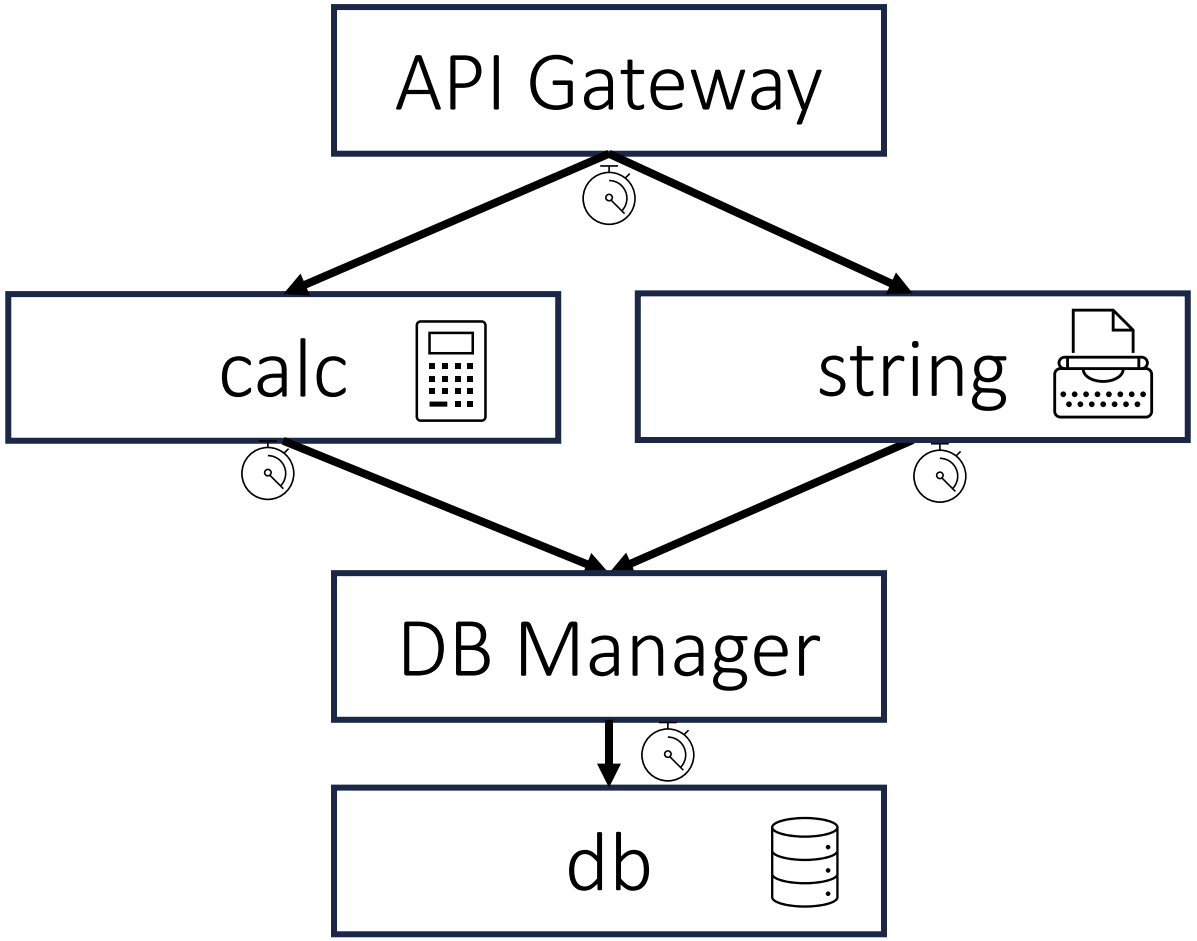
What about architectural smells?



- ~~No API Gateway~~
- ~~Shared Persistance~~
- ~~Wobbly Service Interaction~~
- Endpoint-based Service Interaction

Added timeouts

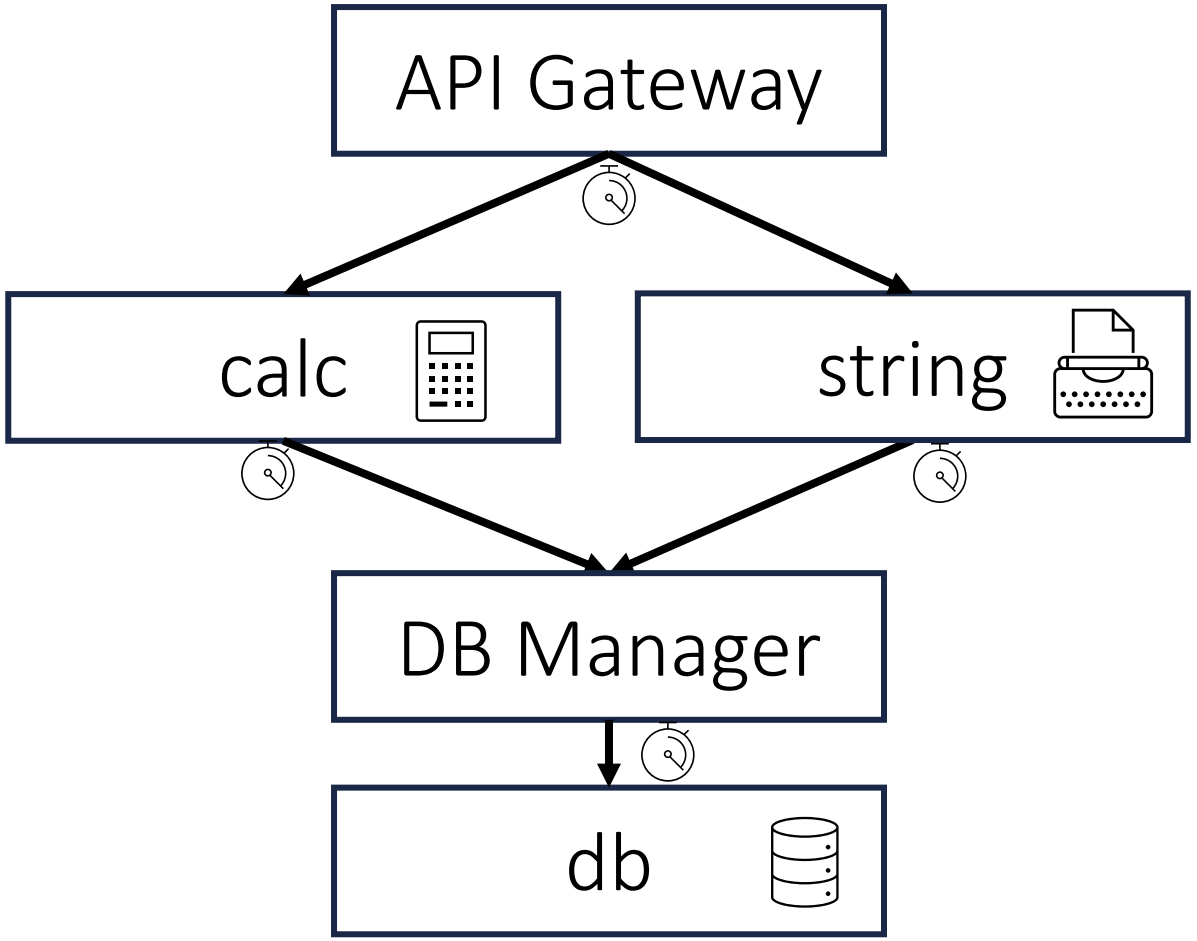
What about architectural smells?



- ~~No API Gateway~~
- ~~Shared Persistance~~
- ~~Wobbly Service Interaction~~
- Endpoint-based Service Interaction

We should add an element to manage messages

What about architectural smells?



- ~~No API Gateway~~
- ~~Shared Persistance~~
- ~~Wobbly Service Interaction~~
- ~~Endpoint-based Service Interaction~~

To simplify things, we ignore this smell in our lab

What about architectural smells?

The refactorings applied to the architecture should be reflected in the software:

- The API Gateway is a microservice which interacts with the clients.
- The DB manager is a microservice which abstracts the DB interface.
- Timeouts and circuit breakers are mechanism to prevent the propagation of failures.
- Message brokers/routers are microservices which manage the architecture communications.

What about architectural smells?

In the project, you have to avoid the architectural smells:

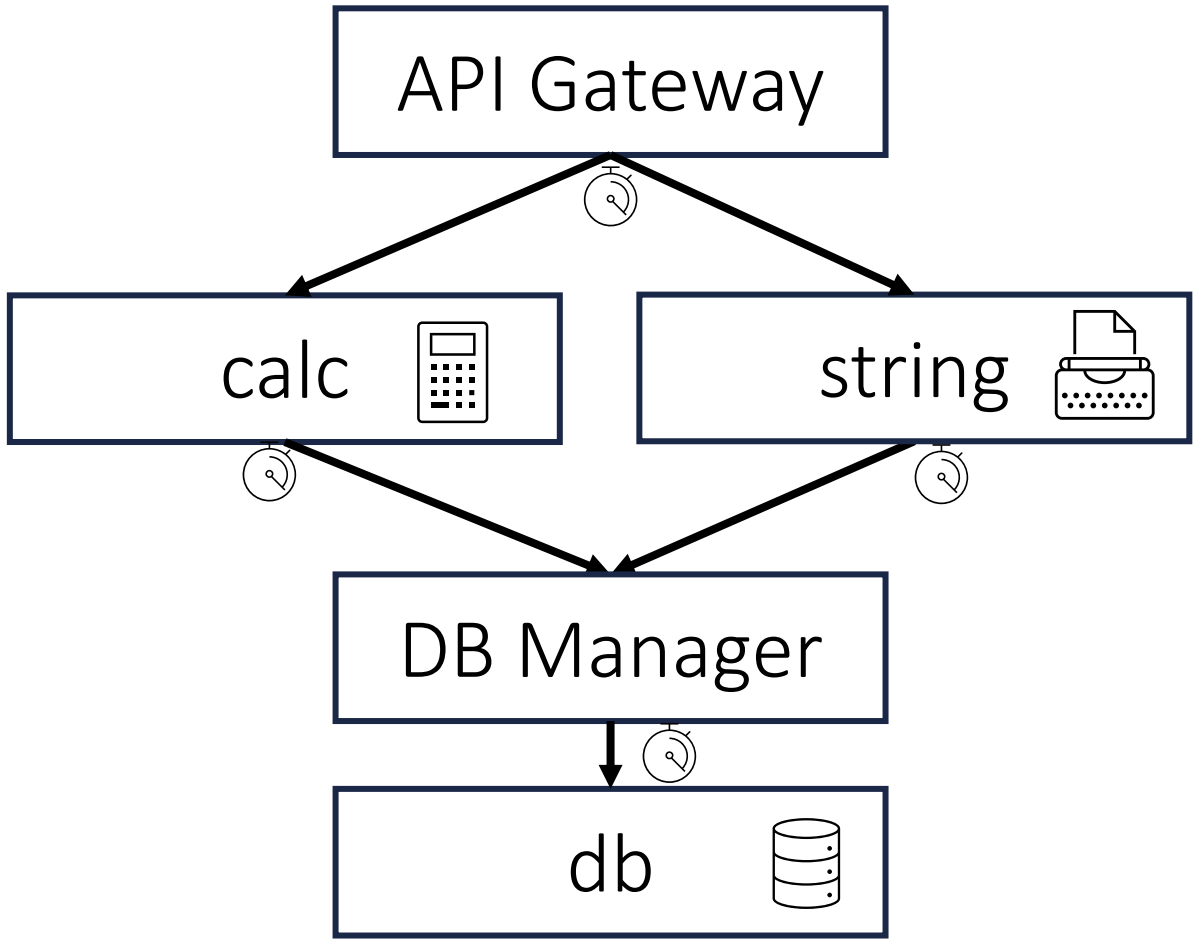
- No API Gateway
- Shared Persistence
- Wobbly Service Interaction

What about architectural smells?

Next week I will give you the full architecture for the testing lab.

The DB will be a MongoDB instance with a volume to give you a different example of DB.

If you want to use your code for the lab, you have to update today's version.



Lab take away

- ❑ Write a docker compose file
- ❑ Break a simple monolithic application
- ❑ Deploy and run a multiservice application with Docker Compose

Project take away

- ❑ The project must be a microservice architecture.
- ❑ The delivered architecture should be split in microservices.
- ❑ You must use docker compose to manage the architecture.
- ❑ You can use whatever database technology you want.
- ❑ Your architecture must not have smells except the Endpoint-based one.