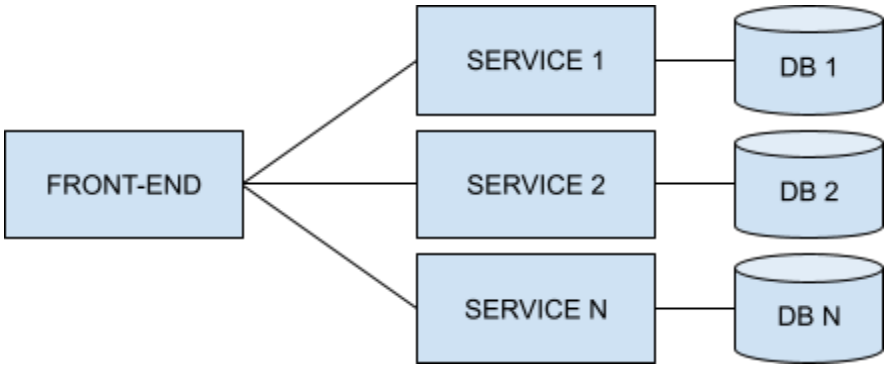
AstroPay - Proposed Solution

Current Scenario

From problem description:

- “we now have a lot of services, each one of them with a somewhat clear scope and responsibilities”, and
- “emerging challenge of aggregating and organizing user activities from different sources”

One can can imagine a scenario more or less like that:



Expected outcome

“to create an endpoint or set of endpoints that can be plugged to a frontend allowing a user to browse his activities and potentially support search, sort and filter in an efficient manner”

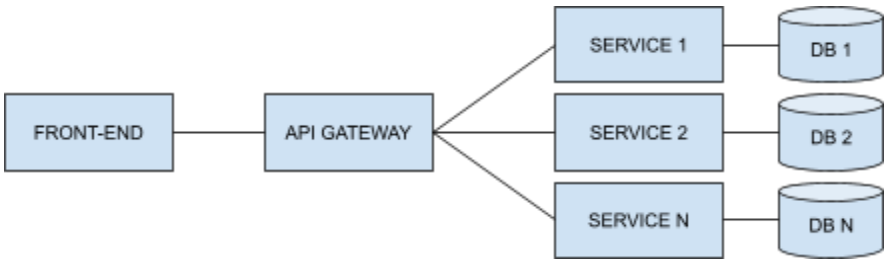
Proposed solution

Part 1 - Hide backend complexity from frontend

As an organization grows and becomes more complex, the same happens to their systems. In a distributed microservices architecture, it is hard to effectively manage a growing number of services, especially when those services are probably created and maintained by different teams.

A frontend application making use of those services will be hard to develop and maintain, since it will be required to access many different APIs, keep up-to-date with their modifications, route requests to each particular service and manage/aggregate returning data.

To address that we can use an API Gateway has follows:



An API gateway serves as a central entry point for managing and orchestrating access to multiple microservices with many advantages:

- Centralized management
- Security
- Load balancing
- Monitoring and analytics
- Rate limiting and throttling
- Transformation and aggregation
- Caching
- Logging

The API Gateway will simplify development of clients while enabling us to freely change/improve backend service APIs or data structures without impacting the frontend.

For our scenario, the API Gateway:

- can break down a single request into multiple requests to different services.
- can transform responses from microservices to meet the specific needs of clients, such as converting data formats or filtering unnecessary information.
- can aggregate data from multiple services into a single response, reducing the number of requests needed from clients.
- allows for versioning of APIs, ensuring that clients can continue to operate with a specific version of an API even as updates are made.

Part 2 - Backend

Once we have implemented Part 1, we are able to start monitoring API calls to identify bottlenecks and prioritize working on improving those services with more impact.

The information provided in the Technical Test is not enough to enable me to suggest how to change microservices APIs or even the database architecture in order to improve performance. Every decision to make a change on a service or data structure should be based on data collected while monitoring user interactions in production.

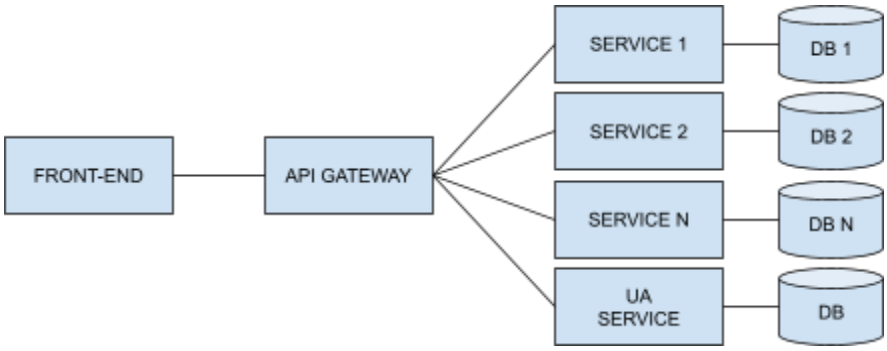
For this Technical Test I will make **the following assumptions**:

- microservices (card payment, deposit, p2p transfer etc.) are working fine in terms of correctness;
- microservices have good performance for transactional requests;
- microservices don't perform well for searching, sorting and querying functionalities (expected outcome);

So, basically, I will focus on how we can improve the browsing/reporting functionalities on the backend.

A new dedicated microservice

My suggestion is to create a new microservice dedicated to serve requests related to browsing user activities. Let's call it "User Activities (UA) Microservice". This microservice will have a copy of data related to user activities.



Other microservices will replicate their data to the UA microservice whose database architecture can be SQL or NoSQL depending on specific details of user queries we want to serve, the number of concurrent users, volume of data stored etc.

When we decouple and separate responsibilities in that way, the other services are free to evolve and change without having to worry about breaking browsing functionalities or causing performance degradation. And the team working on the User Activities service is free to decide on the best architectural solution to address user queries and reports and how to scale.

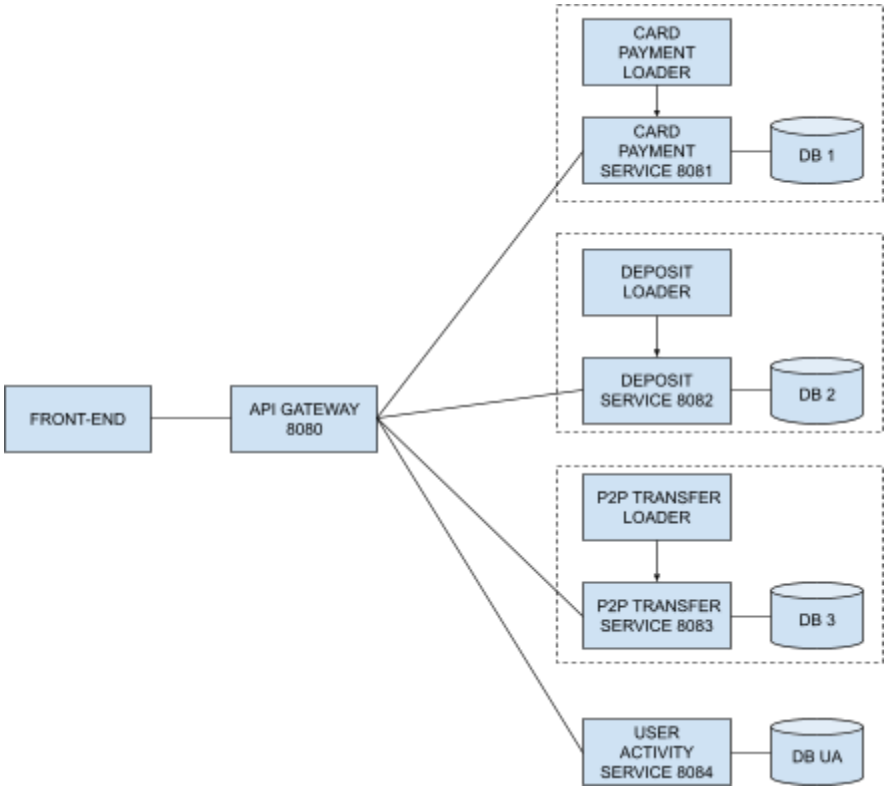
As I mentioned before, the selection of a suitable database solution hinges on acquiring more comprehensive details about the underlying problem. The ultimate choice should also factor in the team's familiarity with available options and their comfort level in assuming potential risks.

For data replication in a production environment, a number of solutions could be used. For example, current services could publish event notifications when data is created or modified. The User Activities service would then subscribe to those notifications and save data to its local database. We could use a dedicated messaging system like Kafka or AWS SQS/SNS. For simplicity and to avoid dependency on additional infrastructure, in our sample application current microservices will send notifications directly to the User Activities service using a REST API.

Technical Implementation

For this technical test I created a POC structured as follows:

- Spring Boot Data REST applications to represent current microservices: card payment, deposit and p2p transfers. Spring Data REST was chosen so I could easily implement CRUD and search/browsing/reporting functionalities.
- The User Activities microservice has also been coded using Spring Boot Data REST.
- A simple API Gateway implemented using Spring Cloud Gateway.
- All databases are H2 in-memory embedded in corresponding microservices.
- Data Loaders for card-payment, deposit and p2p transfers. They are used to simulate real transactions happening on production and data being replicated to UA service.



The new database for user activities transactions is a merge of all 3 other databases, as follows:

CARD PAYMENT	**DEPOSIT**	**P2P TRANSFER**	**USER ACTIVITY**
payment_id	deposit_id	transfer_id	activity_id
			activity_type
card_id			card_id
user_id	user_id	sender_id	user_id
payment_amount	deposit_amount	transfer_amount	amount
payment_currency	deposit_currency	transfer_currency	currency
status	status	status	status
created_at	created_at	created_at	created_at
	expires_at		expires_at
merchant_name			merchant_name
merchant_id			merchant_id

mcc_code			mcc_code
	payment_method_code		payment_method_code
		recipient_id	recipient_id
		comment	comment

Language and packages versions:

- Java 17
- Maven 3.6.3
- Spring Boot 3.2.1

Building

The source code for this project is in this GitHub address:

https://github.com/mmantovani/astropay

The following instructions assume you are building the software in a Linux machine.

```
git clone https://github.com/mmantovani/astropay
cd astropay
mvn clean package
```

Expected result:

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] cardpayment-api 0.0.1-SNAPSHOT ..................... SUCCESS [  7.271 s]
[INFO] cardpayment-loader 0.0.1-SNAPSHOT .................. SUCCESS [  0.263 s]
[INFO] card-payment 0.0.1 ................................. SUCCESS [  0.022 s]
[INFO] deposit-api 0.0.1-SNAPSHOT ......................... SUCCESS [  4.994 s]
[INFO] deposit-loader 0.0.1-SNAPSHOT ...................... SUCCESS [  0.192 s]
[INFO] deposit 0.0.1 ...................................... SUCCESS [  0.001 s]
[INFO] p2p-transfer-api 0.0.1-SNAPSHOT .................... SUCCESS [  4.746 s]
[INFO] p2p-transfer-loader 0.0.1-SNAPSHOT ................. SUCCESS [  0.169 s]
[INFO] p2p-transfer 0.0.1 ................................. SUCCESS [  0.002 s]
[INFO] user-activity-api 0.0.1-SNAPSHOT ................... SUCCESS [  4.749 s]
[INFO] user-activity 0.0.1 ................................ SUCCESS [  0.002 s]
[INFO] api-gateway 0.0.1-SNAPSHOT ......................... SUCCESS [  4.213 s]
[INFO] all-modules 0.0.1 .................................. SUCCESS [  0.001 s]
[INFO] ------------------------------------------------------------------------
```

Running and testing

Execute each back-end microservice:

```
# Starts Card Payments API on port 8081
java -jar card-payment/api/target/cardpayment-api-0.0.1-SNAPSHOT.jar &

# Starts Deposits API on port 8082
java -jar deposit/api/target/deposit-api-0.0.1-SNAPSHOT.jar &

# Starts P2P Transfers API on port 8083
java -jar p2p-transfer/api/target/p2p-transfer-api-0.0.1-SNAPSHOT.jar &

# Starts User Activities API on port 8084
java -jar user-activity/api/target/user-activity-api-0.0.1-SNAPSHOT.jar &
```

Now let's simulate the processing of a new card payment transaction. For that we will use the card payment loader. This command line utility will post a new card payment to the corresponding API:

```
# Creates a new card payment transaction
java -jar card-payment/loader/target/cardpayment-loader-0.0.1-SNAPSHOT.jar
```

We can verify that it has been created successfully using `curl` to query the API:

```
curl localhost:8081/card-payments

{
  "_embedded" : {
    "card-payments" : [ {
      "card_id" : 4759,
      "user_id" : 761,
      "payment_amount" : 194.00,
      "payment_currency" : "USD",
      "status" : "PENDING",
      "created_at" : "2023-12-31 11:16:55",
      "merchant_name" : "MERCH1",
      "merchant_id" : 2024290190,
      "mcc_code" : 190,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8081/card-payments/d9fabab5-764e-4b91-ab32-3a3dfce1da0b"
        },
        "card_payment" : {
          "href" : "http://localhost:8081/card-payments/d9fabab5-764e-4b91-ab32-3a3dfce1da0b"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8081/card-payments?page=0&size=20"
    },
    "profile" : {
      "href" : "http://localhost:8081/profile/card-payments"
```

```
    },
    "search" : {
      "href" : "http://localhost:8081/card-payments/search"
    }
  },
  "page" : {
    "size" : 20,
    "total_elements" : 1,
    "total_pages" : 1,
    "number" : 0
  }
}
```

All APIs expose a search end-point so we can browse transactions simulating a front-end application. However, for this POC, only a few properties of each entity can be filtered. The APIs also provide paging and sorting. Paging would be very important in production to improve performance, avoiding long data transfers between front and back applications.

```
curl localhost:8081/card-payments/search

{
  "_links" : {
    "find_by_user_id_and_status_and_from_and_to" : {
      "href" : "http://localhost:8081/card-payments/search/q{?user_id,status,from,to,sort}",
      "templated" : true
    },
    "self" : {
      "href" : "http://localhost:8081/card-payments/search"
    }
  }
}
```

Now, lets simulate one deposit and one transfer transaction:

```
# Creates a new deposit transaction
java -jar deposit/loader/target/deposit-loader-0.0.1-SNAPSHOT.jar

# Creates a new p2p transfer transaction
java -jar p2p-transfer/loader/target/p2p-transfer-loader-0.0.1-SNAPSHOT.jar
```

For each new transaction created, original backend services will replicate data to the new user activities microservice. So we should expect to find 3 objects in its repository:

```
curl --silent localhost:8084/user-activities | jq '.page'

{
  "size": 20,
  "total_elements": 3,
  "total_pages": 1,
  "number": 0
}
```

In the command above we used `jq` to filter the results. If this command is not available/installed in your system, just omit it.

In our POC, for replication to work, the User Activity Microservice should be up and running. In a real production scenario, this is not acceptable. We should use some kind of solution that holds data while the destination is down, like Apache Kafka or AWS SQS/SNS.

For simplification, we are using an in-memory H2 database, so all data will be lost when backend microservices are restarted.

Now let's create a few more transactions:

```
java -jar card-payment/loader/target/cardpayment-loader-0.0.1-SNAPSHOT.jar 99
java -jar deposit/loader/target/deposit-loader-0.0.1-SNAPSHOT.jar 99
java -jar p2p-transfer/loader/target/p2p-transfer-loader-0.0.1-SNAPSHOT.jar 99
```

The argument "99" is the number of transactions to create, so at this point we should have 300 user activities in total:

```
curl --silent localhost:8084/user-activities | jq '.page'

{
  "size": 20,
  "total_elements": 300,
  "total_pages": 15,
  "number": 0
}
```

We can now start our API Gateway and simulate a frontend application consuming the APIs:

```
# Starts API Gateway on port 8080
java -jar api-gateway/target/api-gateway-0.0.1-SNAPSHOT.jar &
```

Let's suppose that the frontend application wants to get all card payment transactions with status "PENDING".

It could call the original backend service on port 8081:

```
# Test 1
curl --silent localhost:8081/card-payments/search/q?status=PENDING | jq '.page'

{
  "size": 20,
  "total_elements": 33,
  "total_pages": 2,
  "number": 0
}
```

Or it could call the API Gateway on port 8080 passing the activity type and status as parameters:

```
# Test 2
curl --silent
'localhost:8080/user-activities/search/q?status=PENDING&activity_type=CARD_PAYMENT' |
jq '.page'

{
  "size": 20,
  "total_elements": 33,
  "total_pages": 2,
  "number": 0
}
```

Notice: the results are the same but may be different than 33 in your case as loaders generate random data.

However, more interestingly, it could make the same request as test 1, just changing the port number to 8080:

```
# Test 3
curl --silent localhost:8080/card-payments/search/q?status=PENDING | jq '.page'

{
  "size": 20,
  "total_elements": 33,
  "total_pages": 2,
  "number": 0
}
```

This will work because we configured our API Gateway to automatically add query parameter `activity_type` to all request made to end-point `card-payments/search`:

```
# Excerpt from file application.yaml of API Gateway
spring:
    cloud:
      gateway:
          routes:
          - id: card_payments_search_route
            uri: http://localhost:8084
            predicates:
            - Path=/card-payments/search/**
            filters:
            - AddRequestParameter=activity_type, CARD_PAYMENT
            - RewritePath=/card-payments/(?<segment>.*), /user-activities/$\{segment}
```

In this last case, we just needed a very small change to our frontend application, that is, the destination port number. The URL of the request was the same.

This is just a sample of the power and flexibility we have using an API Gateway as explained in more details previously in “Proposed Solution”.

As another example, let’s now suppose the frontend application needs to show all pending transactions for all user activity types.

Originally, the frontend application should make 3 separate calls to 3 different backend services, wait for all services to return data and finally aggregate results.

Now, it can consume a single end-point:

```
# Getting all pending transactions
curl --silent localhost:8080/user-activities/search/q?status=PENDING | jq '.page'

{
  "size": 20,
  "total_elements": 98,
  "total_pages": 5,
  "number": 0
}
```

Using the API Gateway we can change one backend service at a time. We can inspect which requests have the most impact in user experience and prioritize that.

The API Gateway can also implement a **Reactive approach** over our original microservices and enhance the overall system's ability to handle a large number of concurrent connections, respond to events in real-time, and gracefully manage potential spikes in traffic.

Limitations

In addition to the limitations and simplifications already mentioned in this document, there are more due to lack of time:

- Declaration of entities in backend microservices has been simplified. There is no worry about data restrictions, column sizes, indexes etc.
- If data can’t be replicated to the User Activities service, it will be lost. A solution that guarantees transactions hasn’t been implemented due to lack of time and to avoid too much complexity.
- The implementation of search filters in backend services reference properties names as strings, that is, I haven’t configured projects to generate Querydsl entities.
- To avoid dependency on external tools and frameworks, I haven’t used things that would be present in real cloud architecture like docker, kubernetes, terraform etc.