multi-assignment (microservices)

Introduction

Hello all, thank you for the opportunity for me to develop, architect and explain my decisions for this study case.

All deliverables (and examples, starting with "example-"") will be inside the src/ folder.

src/application - contains all code and classes. src/infrastructure - contains all scripts and K8S manifests.

You can find the repository here.

Business Context

Property Technology Solutions B.V. is pioneering digital transformations within the real estate industry. Our flagship mobile application, serving over 500k users, is emblematic of our dedication and prowess. This app, harnessing event campaigns, notifications, and personalized offers, thrives on .NET Core 6.

Our API, at present, supports 10+ distinct shopping center mobile applications. Despite each having its unique data set, they uniformly benefit from our robust API architecture.

1. Microservices Architecture & Integration:

> Detail your approach to structuring microservices and their inter-communication.

My best approach would be for these microservice applications to be orchestrated in a container orchestration technology like **Kubernetes**.

Some of the benefits of having them living in containers and kubernetes itself are the following:

- **Containerization** (applications can be deployed all in the same environment)
- **Security** (network and "operating system" are isolated in the pods level, and on the K8S cluster level we can have certain port-forwarding / load balancing rules through K8S Services)
- **Scaling** (we can define the number of pods running parallel based on metrics)
- **Health-checks** (health checks are regularly done to verify status of the pod, if the pod dies one is recreated with the same configuration)

For their inter-communication it is very simple. Kubernetes provides us with a resource called **Service**. A key aim of Services in Kubernetes is to abstract the exposure of groups of pods over the network.

Example: We have three pods running an image uploading service. The frontend doesn't need to connect to a specific one, it just needs to find one of the three, and it will most likely be the most available one in terms of worker node resource usage. With this service we can abstract this to one single endpoint on the frontend side.

We can apply the example to this context, but the frontend here would be microservice B and the image uploading service microservice A. They would be able to intercommunicate with eachother and be scalable.

For example ms-a.mycompany.com (with an ingress rule for this specific endpoint type, or the service's clusterIP which is constant and doesn't change) could live as the connection address in the ms-b application configuration and ms-b would be able to communicate with ms-a, all internally.

> Explain strategies for seamless integration of microservices and handling potential issues.

For multiple replicas of one microservice the answer is in the previous question's answer and can be resolved using a K8S Service. The generated address will forward the request to one of the available microservice pods (load balancing).

For different microservices intercommunication there are several factors we must handle or take account of:

- **Network Isolation**
- **Availability**
  - Load balancing
  - Automatic Scaling
- **Resilience**
  - Self-healing
- **Observability**
  - Logging
  - Metrics
  - Tracing
  - Alerting

If the microservices need to communicate with eachother and live in the same network they should be **internal**. If they need to communicate with external clients then we can either **whitelist** an IP address/range OR create a **public endpoint**.

If there is a lot of demand on one pod we should **increase the amount of replicas** in order to split the demand and not overwork one pod. This can also be said of the cluster's node itself, we can have pods in different worker nodes so we can split the work also between different servers. We can also **load balance** users between all active pods.

If a microservice crashes it must **revive itself and be active**, we need to atleast have one pod running.

For any other potential issues like application exceptions, pod logs and cluster errors we should have good structured **logging** according to the type of log. We should treat errors and exceptions with higher priority and we need observability or alerting of this.

Metrics are also very important in order to **measure resource consumption**, vitals or executed actions during the application's timeframe.

Tracing is useful but necessary in order to **measure an application's performance** on certain actions.

2. Data Model and API Design:

What is an event campaign?

An event campaign is a an individual or set of events organized for a period of time. It could be a discount/sale phase like black friday, or maybe a new website launch phase, etc.

The three main attributes would be the **one-to-many relationship** between an event campaign and it's events, the **start and end dates** of events as they might not be permanent events and a **name/id** so we can distinct events.

Database Schema

Apart from the columns there are also a few considerations, we need to create **indexes** for the most often accessed columns, like the Id, Name, StartDate and EndDate column so we can optimize reads. We can use a clustered or non-clustered index, in this case we'll use a clustered as it's faster.

The Id columns will be **Primary Keys** with an automatic identity incrementation of 1 starting at 0, so each row inserted the Id will be incremented.

I have created a Schema so we can **separate** these tables logically. The Schema could have been called Events but I decided to name it Marketing.

You can find the sample code in src/2-data-model-and-api-design/model folder.

API Endpoints

Following **REST design** we'll create endpoints for fetching/creating event campaigns and events.

We'll have the following endpoint groups:

- /campaign-events/
- /events/

From these two groups we can then have logical separation of concerns.

Apart from that we also need to decide on the endpoint methods, following rest design I used the following:

GET - to fetch simple information with a safe parameter
POST - to create from a JSON object on the request's body
PUT - we could have used PUT too but we would need to specify the id in the request, this should be used if we plan to update the resource on the server (not implemented)
PATCH - to do a partial update (not implemented)
DELETE - to delete a resource

Campaign Events API Documentation

Campaign Events GET Endpoints

The {id} is a parameter as part of the endpoint address path, it could also be part from a querystring like /campaign-events?id={id} but it's not the best practice.

We could also have a GET endpoint to get all campaign events but I did not implement it.

- GET /campaign-events/{id} (gets campaign event by id)
- GET /campaign-events/{id}/events (gets a campaign event's sub events)
- GET /campaign-events/live (gets all live events)

Campaign Events POST Endpoints

The POST request must have a body of a JSON object with the following format:

```
{
  "name": "xxxxx",
  "startDate": "yyyy/MM/dd hh:mm:ss",
  "endDate": "yyyy/MM/dd hh:mm:ss"
}
```

- POST /campaign-events/

Campaign Events DELETE Endpoints

The DELETE request will use a path parameter of {id} to delete that resource

- DELETE /campaign-events/{id}

Events

Events GET Endpoints

- GET /events/{id} (gets campaign event by id)
- GET /events/live (gets all live events)

Events POST Endpoints

The POST request must have a body of a JSON object with the following format:

```
{
  "name": "xxxxx",
  "campaignEventId": "xxxx"
  "startDate": "yyyy/MM/dd hh:mm:ss",
  "endDate": "yyyy/MM/dd hh:mm:ss"
}
```

- POST /vents/

Events DELETE Endpoints

The DELETE request will use a path parameter of {id} to delete that resource

- DELETE /events/{id}

3. Performance Considerations:

There are a few approaches for optimizing data **retrieval** with and without **caching**, let's start with data retrieval without caching.

Without Caching

Database queries must be **optimized** and tables should have **indexes** for fast data retrieval. A query filter should be applied directly on index columns for fast scans.

Common Table Expressions should also be used for large datasets as these are cached in memory and are good for reusability.

Database **load** is very important, if there are a lot of **write and read** operations **locks** can happen and data retrieval can be slow, to improve this we can do two things.

Database **sharding** if we have a lot of records, want load balancing and reliability. Sharding **distributes** data across various servers and if one node fails, the traffic is redirected to the alive nodes.

Database **partioning** splits a large table within a server and is useful for large table queries.

Considering our context we can expect a few thousands of records per table (campaign events and events) after many years, therefore we do not need sharding or partioning.

With Caching

Data in caches is often data that does not frequently change, and I believe our events tables are a good application of caching.

Data in caches should have a TTL (time to live) or a process that updates a key value or the entire cache by action or by time.

We can implement two types of caching. Request caching or distributed caching.

Request/response caching is exactly what it is, it caches requests' responses and returns them if they're not expirated/invalid. This caching is per application and often requires a small TTL, not the best approach for our use case but it's still useful.

Distributed caching means having a cache engine hosted somewhere where we can access it often and can be used by multiple services at the same time, and as it's memory cached it's very fast but volatile.

This would probably be the best approach to scale our systems as we'll have hundreds of thousands of requests. Each microservice application can have a memory cache read of the distributed cache for even faster reads.

This is all dependent on the infrequent changes of the data, if a change happens the distributed cache must be updated and the application cache must also be updated.

For this update we can use a message queue system to update the distributed cache and to then update the application cache.

The distributed cache and application cache act as a message consumer, and when a message is received it queries the database and refreshes the cache.

A few example of **technologies** used for this:

- Redis and Redis Clusters (distributed caching, clusters)
- Memcached (application cache)
- RabbitMQ, Azure Service Bus, AWS SQS

4. Security Measures:

There are two types of data that we have to take account of when saving and/or displaying. One of those are personal data and the other sensitive personal data.

Personal Data

Personal data means any information relating to an **identified or identifiable natural** person ('data subject').

In other words, any information that is clearly about a particular person. In certain circumstances, this could include anything from someone’s name to their physical appearance.

Sensitive Personal Data

In its most basic definition, sensitive data is a specific set of “special categories” that must be treated with extra security. These categories are:

Racial or ethnic origin;
Political opinions;
Religious or philosophical beliefs;
Genetic data;
Biometric data (where processed to uniquely identify someone).

Security Measures

Some cloud providers do not have **GDPR compliance** as they do not own and/or process their data storage. So each provider must be properly **selected**.

Data separation is also a key factor, personal data and sensitive personal data should **not be stored together**.

Both types of data must either be **encrypted or pseudonymised**.

For our case, personal data should be **hashed, encrypted and then stored**. Encryption keys should be **secret and intransmutable**. For decryption we'll only compare the hashed values.

For displaying personal data in logs, metrics, etc, we should redact parts of the sensitive data. For example: joao00paixao@gmail.com would show up as **********@gmail.com for example.

Apart from encryption measures we should also have **backup measures** to not lose information. This can be obtained through cloud backup services or on-demand backup services.

5. Data Management:

We can manage ACID (Atomic, Consistent, Isolated and Durable) through our database system. To keep things consistent we should use transactions. If a piece of code fails we should have retry policies or a unit of work design pattern.

Data consistency also consists of having a consistent distributed cache data, if a change occurs in the database, it must be replicated onto the cache.

To avoid locks, we could also use read replicas for read only operations. When a write happens, the change is replicated to the replicas. This would help solve some concurrency issues. The best approach would be sharding to avoid concurrency and high demand issues.

6. Problem-Solving Scenarios:

I think this is a very abstract question/problem, here are some of the ways I would tackle an issue.

If it's an issue that requires maintenace, fix or hotfix I would first start by where the issue was **located** and if there was any **logs** associated with the exception. If there's no exception and it's just a logical error I would try and find if the issue is associated with a request, if it's not an request and some UI logic then I would have a pretty good idea where the issue is.

If it's an issue in a request I would need to know the **request path** and the **parameters**, for that I would need to simulate a request on the UI/client. After having a concrete request path what I could do is debug and find the issue.

If the issue is dependent on production data the complexity increases a bit more as I would need to either **restore** a new local/dev database from a production database and if that's not possible I would need to **simulate the data** accordingly.

If it's an issue on a new feature while developing I would try to **divide it into smaller problems** and tackle one by one, create unit tests and create an efficient local testing environment.

If it's a performance issue I would contemplate **tracing, profiling and simulating the environment** as close as possible and see which methods are taking most memory or cpu. From there I would try **optimizing / refactoring** following the 80-20 rule which I will mention below.

The Pareto principle, also known as the 80/20 rule, states that 80% of the effects come from 20% of the causes. In other words, you can achieve **80% of the benefits** of refactoring by focusing on 20% of the code. To apply this principle, you need to prioritize the code that has the most value, impact, or usage in your system.

7. Team Collaboration:

I have created an API integration guide for all types of clients under src/1-microservices-architecture-and-integration which would help frontend developers **integrate** the API in their applications.

I think the best synchronization tool is **good documentation/guidance** on how to integrate systems in other types of applications. Apart from that there can be a **process** for frontend developers to request certain API endpoints with their desired data.

Also a good approach would be to have an *examples/samples folder in the repository with examples of integrations* that can be shared within the team for knowledge sharing.

8. Coding Guidelines

Everyone has preferences when it comes to writing code. I am a person that mostly focuses on readability with a good mix of performance too. I try writing clean code and following clean architecture like Domain Driven Design and design patterns.

I also write code that is testable, applying DRY and SOLID principals.

What I would avoid is: inconsistente variable naming, repeating code, having small and concise methods, instantiating dependencies inside of methods and not classes which causes the class to not be testable, acronyms or one to three letter names on variables, incorrect indentation and so on.

9. Testing Protocols

Unit Testing only becomes a possibility if the codebase is well written. If methods follow DRY and SOLID principals and there are abstractions it becomes easier to unit test the logic inside of the methods as long as you're able to mock the external dependencies like database, storage, environment etc.

With abstractions we're able to abstract the implementation details and create a "mock" object on which we can configure the final result/return so we can unit test a class/method.

Apart from that there're certain rules that we can follow like the AAA principle of Arrange, Act, Assert where we first setup the input data and dependencies, we act on it and we assert the results.

Test Driven Development can also be a good approach when first starting a new feature/class. The main focus is in starting writing the tests and the results before writing the actual class, this is a good approach as it will decrease a lot of possible bugs.