Multi-tenant configuration for API

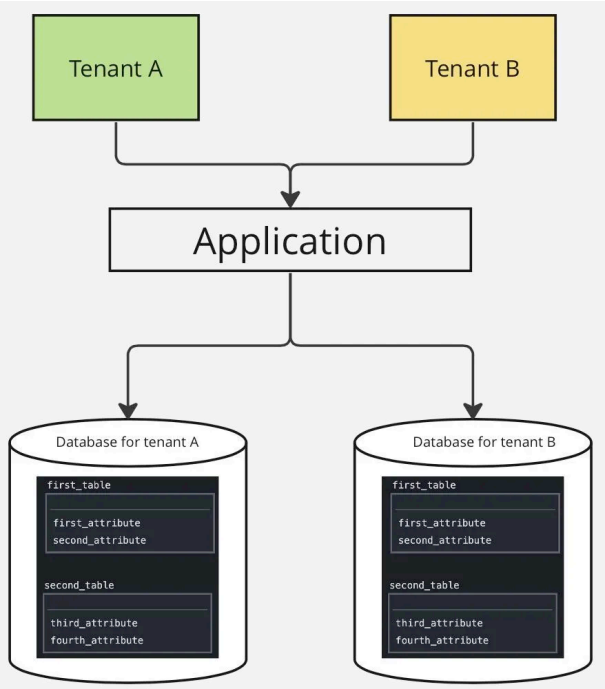
What is multi-tenancy

Multi-tenancy refers to an architecture in which a single instance of a software application serves multiple tenants or customers.

It enables the required degree of **isolation** between tenants so that the data and resources used by tenants are separated from the others.

There are three main **approaches** or models to multi-tenancy:

- Separate Database
- Shared Database and Separate Schema
- Shared Database and Shared Schema



Introducing Pcaas UI

UI is an interface for Pcaas.

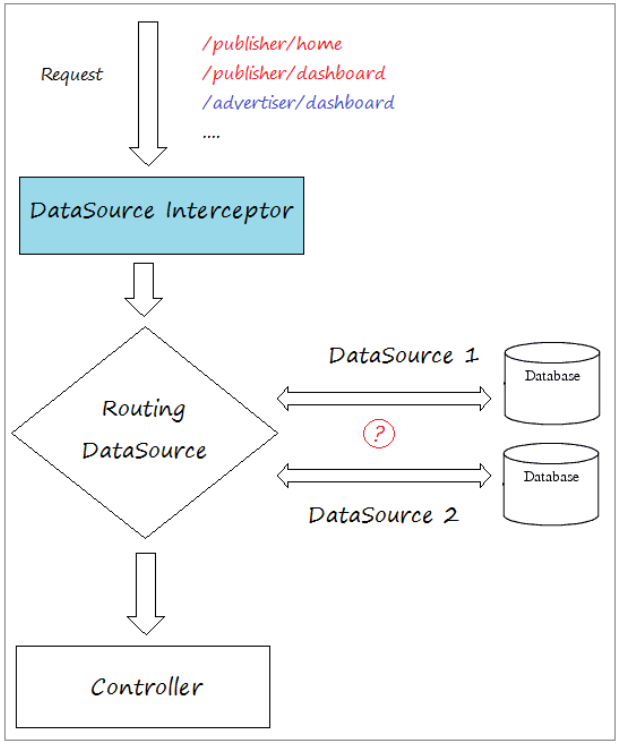
The requirement. Having API configured with two different data sources that have the same tables (and identical entities), with the API deciding **dynamically** which DB should be updated or queried based on a parameter sent in a request header.

The chosen multi-tenancy approach was **Database per Tenant**, in which each tenant's data is kept in a separate database.

How multitenancy works in Spring

Before doing a transaction like saving a new Employee we will need to know the **tenant ID.**
For this we create a **TenantFilter** (or interceptor) that gets the tenant ID from the header of the request before hitting controller endpoints.

We extend the **AbstractRoutingDataSource** and create the **TenantSelector** that will route **getConnection()** calls to one of the various **target DataSources** that are selected in the MultitenantConfig based on a lookup key that is found in the application.properties. The **TenantContext** class is used for storing the current tenant in each request.



First POC

- Created a new SpringBoot application outside Pcaas API.
- Used **Docker compose to run two different Postgres data sources** both with an identical employees table.
- Implemented multi-tenancy with Spring Data JPA by **routing data sources at runtime based on the current tenant identifier.**
- Set **targetDatasources** based on application.properties.

```
defaultTenant=mg-postgres-a
datasource.tenants.names=mg-postgres-a,mg-postgres-b

datasource.tenant.mg-postgres-a.url=jdbc:postgresql://localhost:5433/
datasource.tenant.mg-postgres-a.username=a-user
datasource.tenant.mg-postgres-a.password=a-password

datasource.tenant.mg-postgres-b.url=jdbc:postgresql://localhost:5434/
datasource.tenant.mg-postgres-b.username=b-user
datasource.tenant.mg-postgres-b.password=b-password
```

```yaml
services:
  mg-postgres-a:
    image: postgres
    ports:
      - "5433:5432"
    networks:
      - poc_multi_tenancy
    environment:
      POSTGRES_USER: a-user
      POSTGRES_PASSWORD: a-password
    volumes:
      - a_postgresql:/var/lib/postgresql
      - a_postgresql_data:/var/lib/postgresql/data
    restart: unless-stopped
  mg-postgres-b:
    image: postgres
    ports:
      - "5434:5432"
    networks:
      - poc_multi_tenancy
    environment:
      POSTGRES_USER: b-user
      POSTGRES_PASSWORD: b-password
    volumes:
      - b_postgresql:/var/lib/postgresql
      - b_postgresql_data:/var/lib/postgresql/data
    restart: unless-stopped

networks:
  poc_multi_tenancy:
    driver: bridge

volumes:
  a_postgresql:
  a_postgresql_data:
  b_postgresql:
  b_postgresql_data:
```

Second POC

The Challenge: integrate into PCaaS api module, and make this work together with the existing configuration for databases found in DbConfig class.

We tried **reusing the Bean for Hikari config**, and inject it into the MultitenantConfiguration class so that we could **avoid code duplication** and be able to call the getHikariConfig method, but it was designed to **support only one DB**, and would mix the properties for both, resulting in trying to use invalid credentials to connect and fail.

For this reason we tried to override the bean, but the solution at that time was to use the **@ConditionalOnProperty** annotation, to not initialize the bean if the property api.enabled was set to true… and this worked.

… But we were still not satisfied with this approach because we didn't want to have to alter pre-existing DBConfig and were looking for a more abstract approach.

Final Implementation in API

- Make the configuration more abstract and generic
- Use one configuration to load DB properties for all the databases without having to use the **DbConfig** Hikari bean and not being affected by it's initialization,
- Do not make any change in the original DbConfig class using @ConditionalOnProperty.

To make it more scalable and be able to add more Data Sources in the future, the **targetDataSources** bean was changed to get the databases names from the application.properties enabling the possibility of having one, two or more databases without having to change code.

For each db, in the properties file a new HikariConfig will be created, this replaced the HikariConfig Bean.

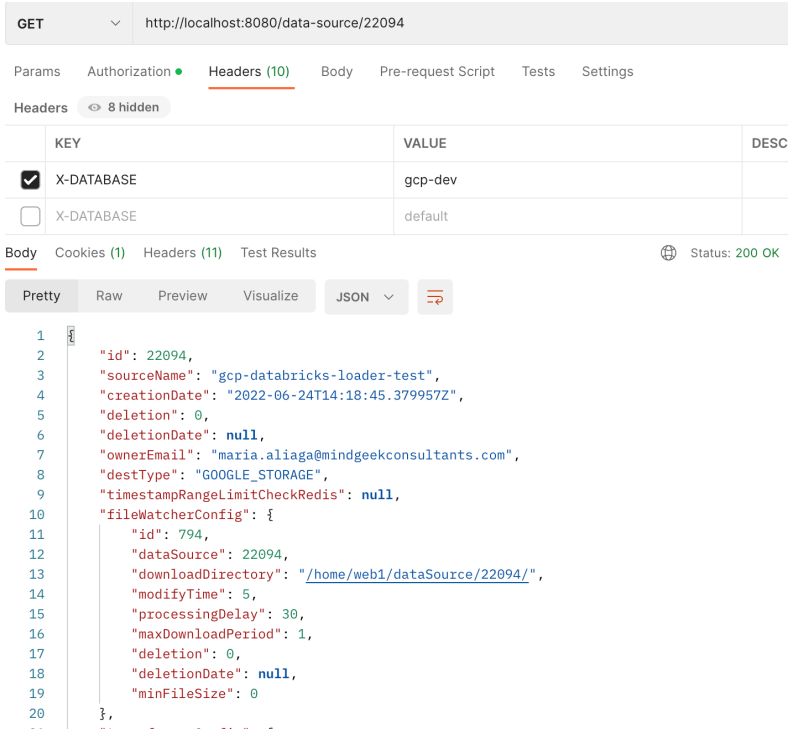
```
/**
 * Setting targetDataSources configurations.
 */
@Bean("targetDataSources")
Map<Object, Object> targetDataSources(AdditionalDatabases databases,
                                      @Qualifier("dataSource") DataSource defaultDataSource) {
    final var dataSources = new HashMap<>();

    dataSources.put("default", defaultDataSource);

    for (final var databaseName : databases.getDatabaseNames()) {
        var databaseProperties = databases.getDatabases().get(databaseName);
        dataSources.put(databaseName, new HikariDataSource(getHikariConfigMap(databaseProperties)));
    }
    return dataSources;
}
```

Final Implementation in API

Finally the X-Tenant-ID header in the **DatabaseSelectorFilter** was replaced by X-DATABASE as we found it better reflected our use case.



Links and Sources

Links

- Postman collection for requests https://www.getpostman.com/collections/ddb6862bb926034ea3cc
- Sources:
  - https://www.bytefish.de/blog/spring_boot_multitenancy_configuration.html
  - https://www.baeldung.com/multitenancy-with-spring-data-jpa