Proposal for the development of the gPlatform product to ensure its operation as a SaaS

1. How can we design the system in a way that every Company will be able to serve games on their gaming site from their domain?

First, we need to determine the database architecture of the future SaaS. Currently, there are three main types of SaaS database architecture:

1. Shared schema
2. Shared database
3. Separated databases

Each of these methods has its own advantages and disadvantages. In the following sections, we will discuss the implementation of one of them in more detail.

We need to configure the application architecture so that the application can establish the tenant associated with each request to the application. This can be implemented at the following levels:

1. Gateway level
2. Application level
    - Middleware level
    - Business logic level

The choice of implementation method depends on the software's technical requirements. Examples of questions that should guide the decision on how to determine the request tenant include:

- Can the user interface be unique for each tenant?
- Is geographical data isolation planned for the future?

In the subsequent sections, we will analyze one of the methods for determining the tenant at the request level.

For this task, we will not consider all methods, as this requires a more thorough analysis of the product, potential risks, and expected load on the future product.

2. What modification should be done to the users table at gPlatform to support this change?

For this test task, we will choose the simplest implementation of multi-tenant software - shared schema.

What needs to be changed in the database structure to support multi-tenancy:

1. Add a `tenants` table to store information about each software tenant. This table must include the following fields:
    - id
    - domain
2. Add a `tenant_id` column to each table whose data should be unique for each tenant (`users`, `auth_tokens`, etc.). Link this column to the tenants table by id.
3. In each table with the new `tenant_id` column, replace all unique constraints except `id` with composite unique constraints and include `tenant_id` in the composite key.

3. Considering we have 1 backend cluster that serves all companies, how can we validate a user login on one gaming domain in such a way that it does not give access to a different gaming domain? (i.e. authenticating on site A, grants access to site A only)

To precisely delineate tenant access rights to data, the following application components need to be added:

1. Determine the tenant associated with the request. By default, each browser request transmits the request domain and cookies associated with that domain. This is sufficient to determine from which domain the request is made and to obtain the user's token from the cookies to identify the user making the request. Next, we need to find the `tenant_id` to use in database queries. To do this, we find the record in the `tenants` table with the corresponding `domain = 'tenant1.example.com'` where `tenant1.example.com` is the domain from which the current request was made.
2. Row Level Security Policies - necessary to restrict data access at the tenant level. RLS can be implemented at the database or application level. Both options have their pros and cons, the analysis of which is beyond the scope of this test task. For simplicity, we will consider application-level implementation. Application-level implementation involves modifying all queries by adding additional conditions like `WHERE tenant_id = '...'` for read and modify operations and adding the `tenant_id` field when inserting a record.