GAMEFICATIONSYS solution architectural recommendations

Short summary

Below you can find suggested GAMEFICATIONSYS architecture and recommendation for implementation.
GAMEFICATIONSYS is gamification platform for B2B and C2C market.

Recommendations for “to be” architecture

See scheme:

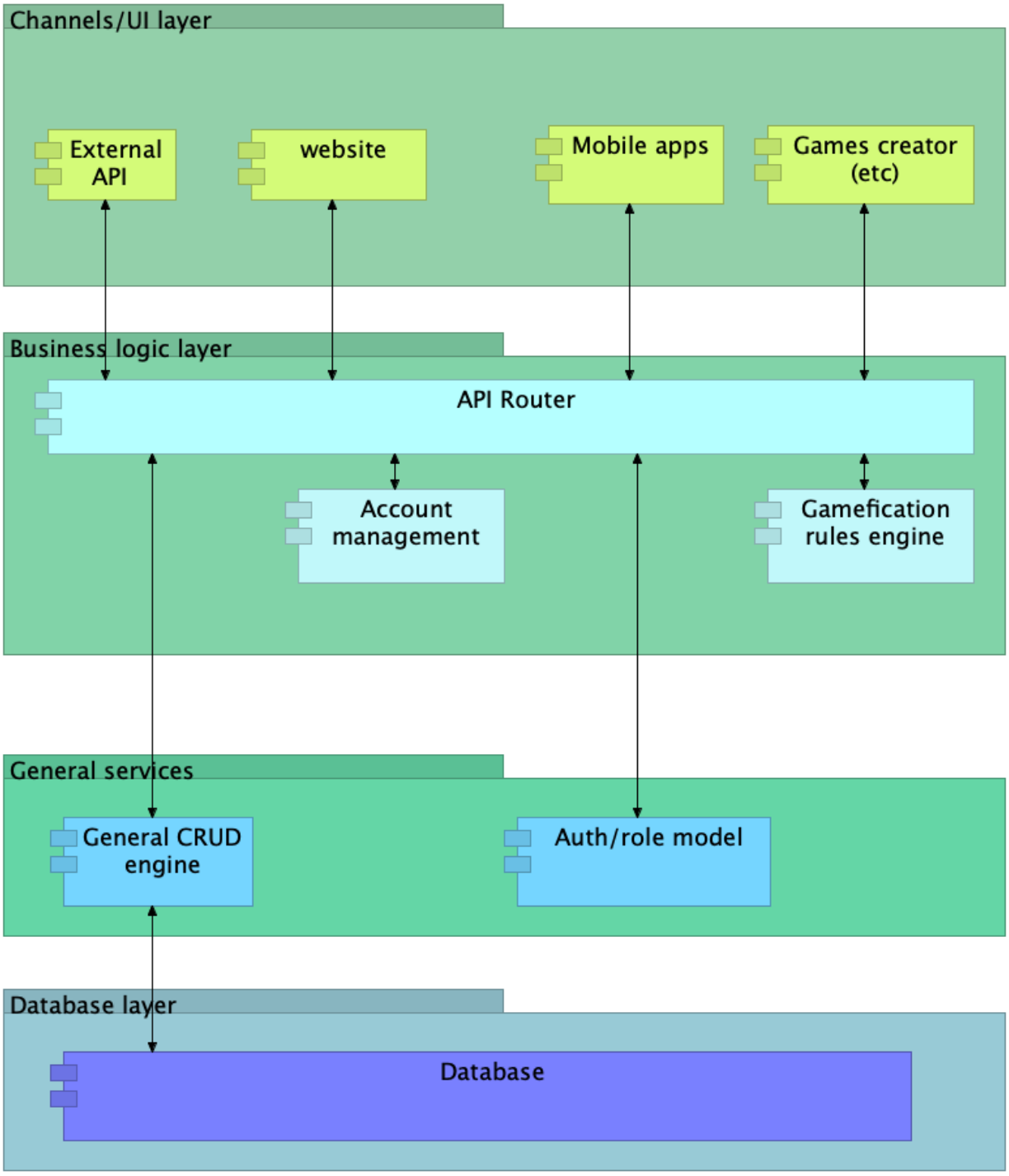

Figure 1. Recommended "to be" high-level architecture

GAMEFICATIONSYS solution architectural recommendations

It is suggested to make layered architecture approach more distinctive.
You can find “to be” architecture above. Below I have listed key points.

1. On the top layer we have UI (User Interface) and API (Application Program Interface) for external systems. UI – it could be website, mobile applications, in future Game creator app etc. API is introduced mainly to take into account needs of corporate clients.
   API should allow all possible functions of the system which is done by UI but on API level.
   For instance, add new user, or extract report, etc. It will be used mainly for integration with company IT lansdscape and **API will be source of additional income for Platform owner (development of integrations)**. For instance, integration with SAP SuccessFactors or internal learning system etc.)
2. On the next layer key component to review is API Router. API router accepts requests from UI and API from top level, optionally makes some additional processing and passes request to different microservices.
   Also it “routes” requests between microservices.
   Examples (just for illustration, not implying specifics of implementation).

   Example 1) request from UI - “/getlistoflocation” will go to router and then directly to General Services layer and return list of available locations from database.
   Example 2) request from UI “/[user_id]/[object_id]/get_access_rights” will be routed by API router to Auth/role model microservice.
   Example 3) the same request ““/[user_id]/[object_id]/get_access_rights” from General CRUD Engine to will go to router and routed to Auth/role model microservice again.

   Important – on scheme above all microservices can talk to each other (like Auth with CRUD Engine, or Gamefication rules engine with all of them, but they connect not directly, but through API router.

   Exact scenarios and exact list of microservices can be suggested by developers, architectural scheme describes principle.

   Importantly, API Router abstracts end-points of microservices, due to it in complex upgrade scenario you can have to end-points for Accounts (old version and new version) and by configuring API Router you can change how routing will go. You need to account costs of future system development now, right architecture will lower those costs.

3. General CRUD Engine it is microservice that abstracts database operations. But it is not ORM, it is web-service. Unfortunately, ORM approach according to modern practice suggest making a lot of decisions for developer which sometimes is suboptimal.
   (see https://blog.logrocket.com/why-you-should-avoid-orms-with-examples-in-node-js-e0baab73fa5/ , for instance)

   It is suggested to use simple ‘broker’ approach to communicate with database layer.
   See my old implementation, code and ideas can be freely used
   https://github.com/sergey-chekriy/node-sqlbroker

Approach to corporate clients rollout – architectural solution

It is suggested to implement “programmable infrastructure” solution.

GAMEFICATIONSYS solution architectural recommendations

Let's review the following scenario, when Best Corp decided to buy software, paid for corporate account and responsible person setup system for the 1st time:

1. New corporate customer from Best Corp signs up on landing site for corporate clients, confirms payment (or pays directly by card), selects that his portal page will be bestcorp. platformdomain.com.
2. After customer confirm choices, script in cloud created new container on existing server (optionally created a server). Almost all clouds have API to create servers, container on server will be created by custom script (need to be developed). Container configuration parameter is selected subdomain name (bestcorp. platformdomain.com).
3. Containers starts and customer get access to bestcorp. platformdomain.com.
   Note: financially this valid. Yearly hosting of 16 GB RAM server is around $1K (based on Linode prices), it can host up to 3 containers with App, if we base on pricing model by Atlassian, price for customer will be at least $1000/per application, and in average $2500. And could be up to $150K/year.
   Which means cash-out $1K and cash-in from $3K (min) to $7,5K per server (average).

   Note: SSL certificate for *. platformdomain.com need to be acquired (domain incl subdomains) to ensure https access to subdomains.

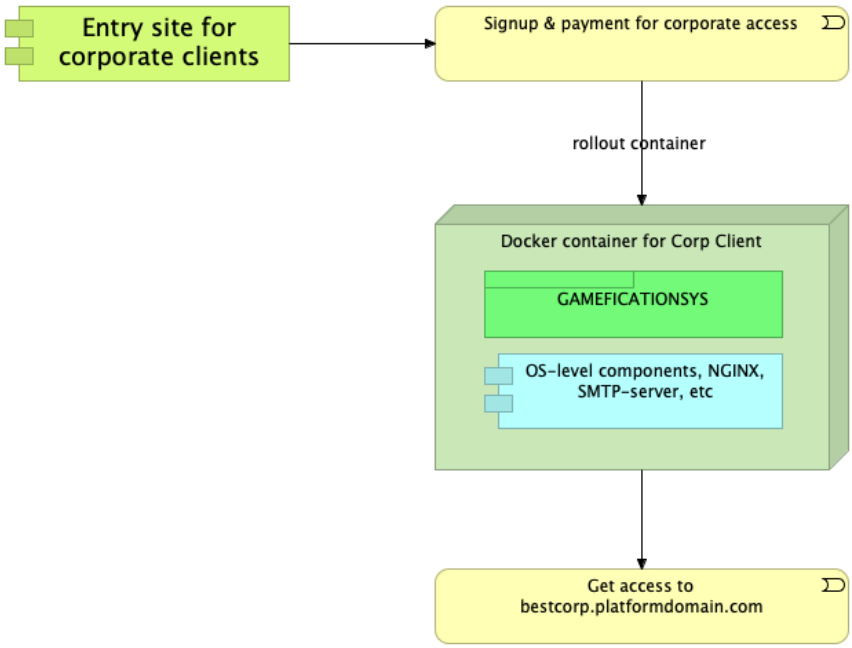

Figure 2. Approach to rollout for corporate clients

In this scenario, we have corporate clients like IBM, CISCO, Coca-Cola, all of them using application in dedicated containers: ibm.platformdomain.com, cisco. platformdomain.com, cocacola. platformdomain.com.
Number of simultaneous users even in biggest companies can't exceed 500-1000, and based on suggested architecture with some vertical scaling of server (i.e. increase memory) should be enough to process such number.
This architecture also highly marketable. Corporate clients will value that their data/system is in dedicated container.

Approach to "community" version scaling – architectural solution

From other hand we also have "community version" of application (C2C), where number of simultaneous users optimistically can grow above 10000. Which is at least 1 million of registered users.
By using the same pattern we can allocate users to different containers and scale horizontally. But to ensure single data source we will need to slightly adjust architecture, when it will come to this load. See Fig.3 below.

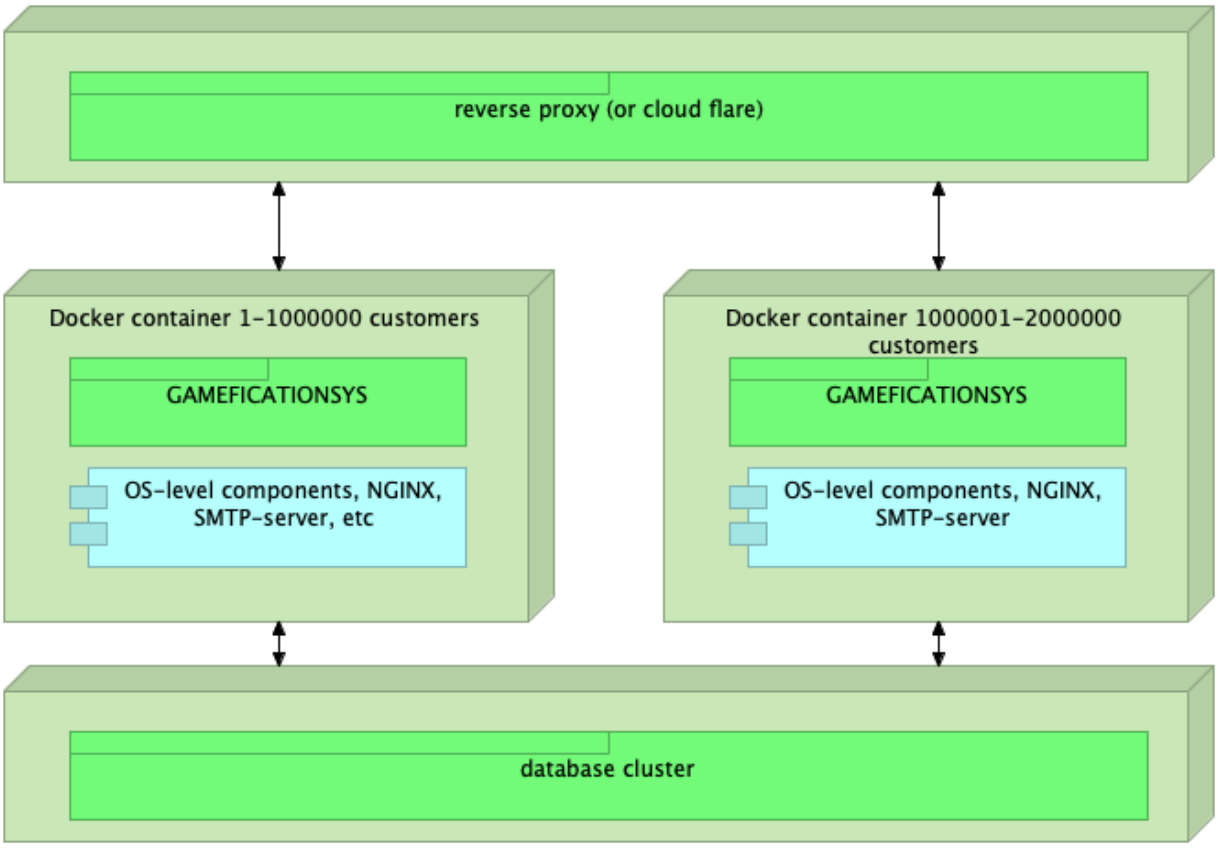

Figure 3. Possible scaling approach for community version

What will change:

1) Database layer will be moved out of containers and will be replaced by cluster database solution;
2) "General CRUD engine" microservice will be replaced possibly (depends on implementation)

This approach is not considered in details, as it is hardly possible to face scalability issues of community version in foreseeable future, and our task here is to ensure that proposed architecture (Fig.1) enables scalability of "community" version in general.

Recommendations summary

1. Take into account layered architecture approach in development, as per description above.
2. Implement system delivery in docker container
3. Implement scripts of containers auto-rollout ("programmable infrastructure") for corporate scenario and auto-scalability of "community version"