SUGARSPACE

Gaming platform that provides support for game data management and server side logic

Content

GENERAL

What is the platform itself

SugarSpace is a gaming platform that provides support for managing game data and implementing server-side game logic. In fact, this is a three-component game support system: code generation on the client, custom logic on the server, and a game content management dashboard.

Platform capabilities

Initially, the platform was conceived as a remote game configuration management service, but over time, the platform has been expanded and now, in addition to working with game content, it supports game accounts, provides a flexible system of game events, supports authorization with tokens and automatic state synchronization of game models between server and client.
In general, the platform has everything you need to implement meta-part of the game.

Tech stack

Backend + Dashboard

The backend for the system uses MERN web technology stack: MongoDB as a base for storing content and game accounts, Express.js as a means of client and server interaction through requests and events, React.js as a library for building UI for a dashboard, and Node.js as execution environment for authoritative code. There is javascript in use.

Client

The client part is a wrapper over the functions for generating strongly typed content storage classes, user structures, as well as structures for automatic synchronization of client and server models. There is C# in use.

WORKING WITH THE PLATFORM

Working with the server

The server allows you to execute custom scripts in response to requests from game clients. Custom scripts should be in the *customLogic* directory. Here is an example script:

```
1.  module.exports = (user, content, events, custom_parameters) =>
2.  {
3.
4.  }
```

user – the user on whose behalf the script is being executed. For security purposes, scripts can only work and change the data of the user on behalf of whom they are running.
content – game settings. Content is a collection of entities that are addressed by unique keys.
events – event module. Allows you to send events to clients by their unique id.
custom_parameters – any parameters that the client will send.
Scripts can do the following:
- change user model. You can change the model using any methods available in js. For example, you can add an element to a collection or remove that element from it. After the end of the script, the user is automatically saved in the database, and information about the changes to the user's model is sent to the client, which merges it with its local model. Here is an example of how the user model can be updated:

```
1.  module.exports = (user, content, events) =>
2.  {
3.      user.currency = user.currency - 10;
4.  }
```

- return a result of any work. The result of the work is returned to the client through the *return* keyword. Here is an example:

```
1.  module.exports = (user, content, events) =>
2.  {
3.      return {
4.          advice: "Work smart hot hard!"
5.      };
6.  }
```

- throw an exception. Sometimes it is useful to point out to the client that there is an error in the data or that an operation cannot be performed. It can be done like this:

```
module.exports = (user, content, events, health) =>
{
    if (health < 10)
    {
        throw new Error('health is less than 10!');
    }
}
```

- send an event. If necessary, the script can send events to clients. Events can be sent not only to the client on whose behalf the script is running, but also to other online clients. To send an event, you need to do this:

```
module.exports = (user, content, events, message) =>
{
    if (!message)
    {
        throw new Error("Message can't be null!")
    }

    events.send(user._id, message);
}
```

- query game collection data. Game data (monster power, reward for victory) is stored as collections. Access to the first element of the collection is possible through the keyword *first*. It is very handy when working with settings represented by a single entity.

Here is an example of accessing game content from the script's body:

```
module.exports = (user, content, events, weaponId) =>
{
    const settings = content.Settings["first"];
    const weapon = content.Weapon[weaponId];
}
```

Working with the Dashboard

The dashboard is the heart of the platform. It provides game data management. In order for the administrator to gain access to the dashboard, you need to enter admins data in admins.json. The login is

specified in the username field and the password's hash is *hashedPassword*. Hashing uses bcrypt. To verify the hash, you can use https://bcrypt-generator.com/

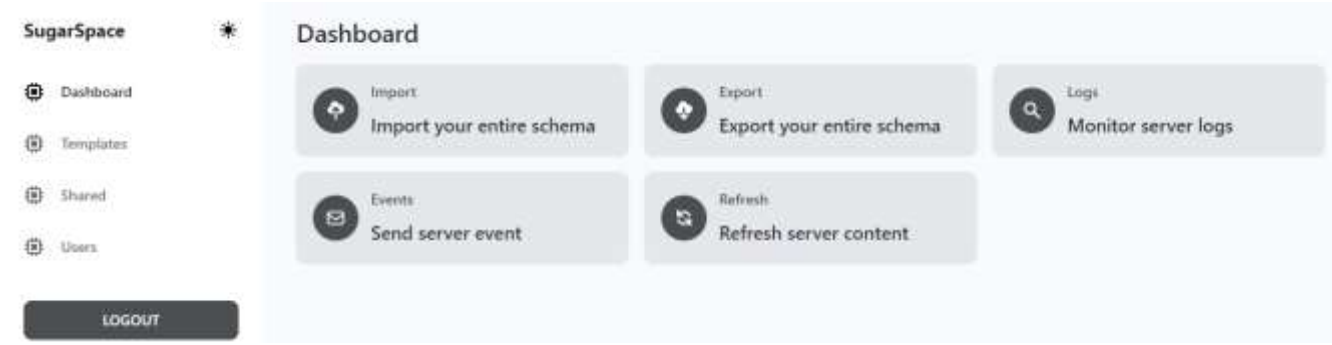
Here is an example of admins.json file:

```
[
    {
        "username": "admin_00",
        "hashedPassword": "$2a$12$Pk4jK"
    },
    {
        "username": "admin_01",
        "hashedPassword": "$BdIkmDjAKQu"
    }
]
```

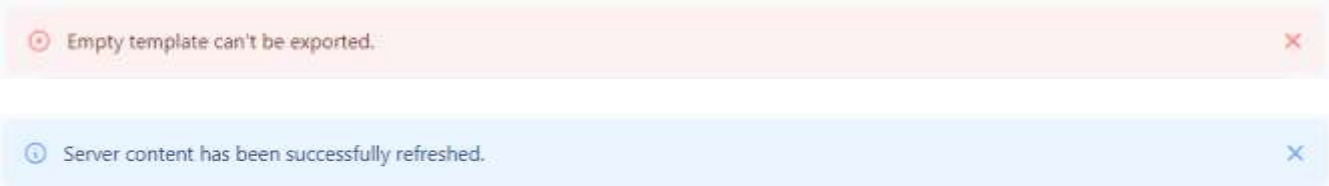
The dashboard has two built-in themes: white and dark. You can switch between them by clicking the button in the form of a sun next to the name of the platform.

Here are two themes we have got:

The work of the dashboard is divided into sections for convenience.



During the operation of the dashboard, situations arise that need to be reported to the user. Notifications are used for this. They look like this:



Common operations

Import – you can restore all game templates from a file.
Export – You can download and save all game templates.
Logs – you can get the log file of the server.
Events – having a user ID, you can send him a message.

Update the content – updates the content cache. In order to optimize the content of the server it is not downloaded from the database for each request, but comes from the cache. Accordingly, if changes have been made, the cache needs to be updated.
Logout - when you are finished working with the dashboard, you can close the session by clicking on *logout*.

Normal templates

This section creates templates for content entities. A template is a blueprint according to which real entities will be created. The template has certain structure. For example, a game ability template might contain fields such as the *duration* of the ability and its *damage*.

Template creation

To create a template, you need to click on the *Create* button in the upper right corner of the section.

When you create a template, you must specify its name, type, and schema. The template name must be unique. It cannot be changed after creation. It is best to use capitalized English words for template names. The template type can be specified as normal (the Shared checkbox is un-checked) and shared (the Shared checkbox is checked).

Normal templates have differences from shared. Entities can only be created based on normal templates. Only shared templates can be used as sub templates for normal and shared templates. In short: most often create normal templates. Shared are needed where it is necessary to use some templates in others. The template has certain structure. The schema of a template defines its essence.

Here is an example of a possible *ability* template schema:

```
{
    "root": {
        "power": "int",
        "duration": "float"
    }
}
```

Template structure formation rules:
- the template scheme is set according to the json formation rules.
- the root element of any template is always an object named *root*. You cannot create a template whose root is an array.
- root elements are strongly typed.
- in a schema, the key defines the name of the element, and the value defines its type.

Template example:

```
{
    "root": {
        "deviceId": "string",
        "name": "string",
        "level": "int",
        "distance": "float",
        "payer": "bool",
        "gender": "enum<Gender>",
        "message": "text",
        "weapon": "link<weapons>",
        "money": "shared<Money>",
        "magic": "custom<Magic>",
        "abilities": "map<string,custom<Ability>>",
        "currencies": "array<custom<Currency>>",
        "branch": "branch<Branching>"
    },
    "Currency": {
        "type": "string",
        "amount": "int",
        "rarity": "string"
    },
    "Ability": {
        "amount": "int"
    },
    "Magic": {
        "amount": "int"
    },
    "Gender": [
        "Male",
        "Female"
    ],
    "Branching": {
        "ABranch": "custom<ABranchData>",
        "BBranch": "custom<BBranchData>"
    },
    "ABranchData": {
        "data1": "int"
    },
    "BBranchData": {
        "data2": "int"
    }
}
```

Data types

Primitive types

int – integer.
long – big integer.
float – floating point number.
string – string in utf8 format.
enum – enumeration, as the type you need to specify the name of an array with all possible enumeration values.
text – the same as string, only the input field is multiline.
link – a reference to an entity in another collection. The same as string, only a field in the form of a list of entities from the specified collection.

Custom types

shared – shared type. The description of the shared type must be an object (json). The description of the shared type must be given as a shared template with the specified name. Used to form common data types for different templates.
custom – custom type. The description of the custom type must be an object (json). The description of the custom type must be given within the template being created. Used to form complex custom data structures.
map – map. The key can only be *int, float, string*. The value can only be *int, float, string, custom*.
array – array. The element can only be *int, float, string, custom*.
branch – It is used when within the same collection it is necessary to store content of different structure. For example, in a weapon collection, each weapon must have a price, but an assault rifle must also have a rate of fire, but a knife does not. As a type in the schema, an object (json) must be specified, each element of which represents a custom object of a possible branch structure.

Template editing

Once created, the template can be edited. To do this, you need to find the desired one in the list of templates and click on the button with a pencil image on it.

Template deletion

To delete a template, you need to find it in the list of templates and click on the trash can button. After confirming the selection, the template will be deleted.

Entities export

To export template entities, you need to find the template of interest in the list of templates and click the arrow button.

Template's entities

After creating a template, functions for working with its entities are available (this is not possible for shared templates).
To see what entities a template has, you need to find it in the list and click on the button with an eye image on it. This will open the entity selection page.

IMPORTANT: if the template has ***title*** element with type ***string***, its entities will use the value of this element to display in the list of entities.

Entities creation

To create an entity, you need to click on the *Create* button. This will open the entity creation page.

Entities editing

To edit an entity, you need to find the required entity in the list and click on the button with a pencil image on it. This will open the *Edit Entity page*.

Entities deletion

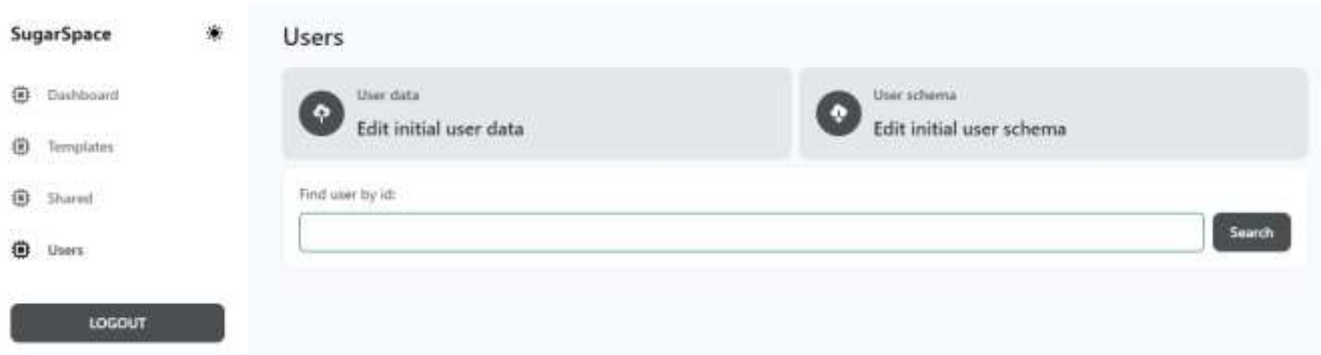
To delete entities, you need to find one in the list of template entities and click on the trash can button. After confirming the selection, the entity will be deleted.

Shared templates

Working with this section is similar to the section of normal templates. The difference is that shared templates are grouped in this section.

Game accounts

This section is required to work with player accounts.



Editing the user structure

The user is also a template. It, like any other template, can be edited to set the desired user structure. Clicking on the *edit initial user schema* button will open the player template.

IMPORTANT: in the structure of any user, the presence of the ***deviceId*** and ***clientVersion*** element is mandatory.

Editing the default user

The first time the client connects to the server, the server creates a new user. It is often necessary to set specific values for user fields other than the default ones. To do this, you need to edit the initial user data. To do this, you need to click on *edit initial user data* the button in the current section.

Editing a game user

Sometimes it is required to correct the data of a particular user. To do this, you need to enter its unique id in the text field and click on the *Search* button. If there is such a user, the edit user page will open. If not, a corresponding notification will appear.

Deleting a game user

Sometimes you need to delete a specific user. To do this, you need to open the page for editing it and click on the button with the trash can. After confirmation, the user will be deleted.

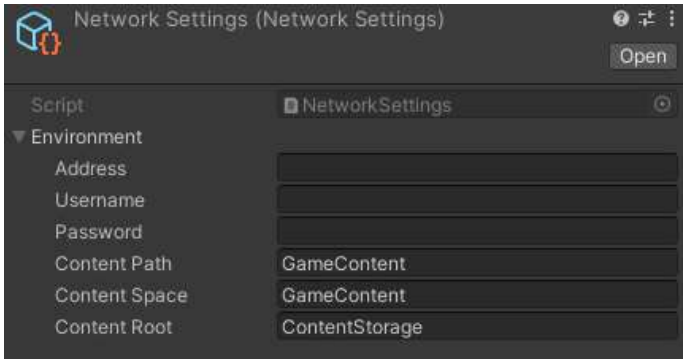
User Import/Export

It is often convenient to save a user model so that you can return to it later and restore it. This can be done using the import/export buttons on the user interface page.

Working on the client side

How to start

The first thing to do is to import the unity package into the client. After that, the *SugarSpace* folder will appear in the project - the root folder of the platform integration.
Then you need to create a settings object. This can be done through the menu: Assets -> Create -> ScriptableObjects -> NetworkSettings. The inspector window for this object looks like this:



Address – server address. Format example: http://000.000.000.000:0000.
Username – dashboard username.
Password – dashboard user password.
ContentPath – relative path to the folder for content generation.
ContentSpace – namespace for generated content.
ContentRoot – class name for content collection.

Content generation

To start generating content according to the schemes specified in the dashboard, you need to execute the following menu item: *SugarSpace -> Generate Content*.
During the generation process, data collection classes will be created, as well as the structures required to synchronize these models between the server and the client.

An example of a generated data collection class:

```
namespace GameContent.UserSpace
{
    using System;
    using System.Collections.Generic;


    [System.Runtime.Serialization.DataContractAttribute()]
    public class Weapon
    {

        [System.Runtime.Serialization.DataMemberAttribute()]
        private string contentId;

        [System.Runtime.Serialization.DataMemberAttribute()]
        private int levelIndex;

        public string ContentId
        {
            get
            {
                return contentId;
            }
        }

        public int LevelIndex
        {
            get
            {
                return levelIndex;
            }
        }
    }
}
```

User deletion

For convenience, you can also delete a user directly from the editor. To do this, execute the following menu item: *SugarSpace -> Remove User*.

Façade creation

All work with the system is done through the facade - the central integration node.

Here is the Facade object example:

```
1.  var networkFacade = new NetworkFacade<
2.              GameContent.UserSpace.UserProxy,
3.              GameContent.UserSpace.User,
4.              GameContent.UserSpace.UserDiff,
5.              GameContent.ContentStorage>(
6.              networkSettings.ServerAddress,
7.              TimeoutSec,
8.              user => new GameContent.UserSpace.UserProxy(user),
9.              (diff, proxy) => proxy.Patch(diff)
10.         );
```

Template parameters:
Proxy – generated proxy class.
User – generated user class.
Diff – generated patch class.
Content – generated content collection class.

Constructor parameters:
serverAddress – server address.
timeout – server response timeout.
proxyFactory – delegate to create a proxy object.
patchFactory – delegate to update the proxy object with a patch.

IMPORTANT: nothing needs to be changed here, except to substitute your own names of the generated classes.

Connecting to the server

After creating the facade, you need to log in to the server as a client and download the content. You can do it like this:

```
var loginResponse = await networkFacade.Login(NetworkTools.DeviceId);

if (!loginResponse.success)
{
    UnityEngine.Debug.LogError(loginResponse.error);
    return;
}

var contentResponse = await networkFacade.LoadContent();

if (!contentResponse.success)
{
    UnityEngine.Debug.LogError(contentResponse.error);
    return;
}
```

To subscribe to events from the server, you need to do this:

```
networkFacade.MessageReceived += OnMessageReceived;

private void OnMessageReceived(string message)
{
    UnityEngine.Debug.Log($"Incoming event : {message}");
}
```

You can get content like this:

```
var weaponId = "61c75856a63d93895732b184";
var weaponContent = networkFacade.GameContent.Weapon.First(item =>
{
    return item.Id == weaponId;
});
```

You can get the current state of the model like this:

```
var weaponId = "61c75856a63d93895732b184";
var weaponModel = networkFacade.UserProxy.Weapons.First(item =>
{
    return item.ContentId == weaponId;
});
```

You can subscribe to model changes like this:

```
networkFacade.UserProxy.CurrencyChanged += OnCurrencyChanged;
networkFacade.UserProxy.WeaponsChanged += OnWeaponsChanged;
networkFacade.UserProxy.WeaponsUpdated += OnWeaponsUpdated;

private void OnCurrencyChanged(int currency)
{
}

private void OnWeaponsChanged()
{
}

private void OnWeaponsUpdated(KeyValuePair<int, Weapon> updated)
{
}
```

When the contents of the collection change, such as when an element is added or removed, *On[collection_name]Changed* is called. When only a collection item is updated, *On[collection_name]Updated* is called.

Working with requests

Request creation

Each request is implemented by inheriting of the platform class. Here is an example request:

```
private class SomeRequest : NetworkProtocol.GameCommonRequest
{
    public SomeRequest() : base("api/someRoute")
    {
    }
}
```

As the api parameter of the base class, you must specify the name of the server script in the format *api/[script_name]*.

Here is an example of a simple request that sends *name* parameter to the server.

```
1.  private class ChangeNameRequest : NetworkProtocol.GameCommonRequest
2.  {
3.      public readonly string name;
4.      public ChangeNameRequest(string name) : base("api/changeName")
5.      {
6.          this.name = name;
7.      }
8.  }
```

Request sending

The request is sent through the facade. Here is an example:

```
1.  public async Task ChangeName(string nickname)
2.  {
3.      var response = await networkFacade.Request<object>(
4.          new ChangeNameRequest(nickname));
5.
6.      if (!response.success)
7.      {
8.          UnityEngine.Debug.LogError(response.error);
9.          return;
10.     }
11. }
```

Response receiving

When sending a request, it is possible to specify the class of the response to it by setting the template parameter of the Network-Facade.Request<T>() method. If the server handler does not send anything back to the client, the template parameter must be object.

An example of a request with a specific response from the server:

```
private class DataDemoRequest : NetworkProtocol.GameCommonRequest
{
    public DataDemoRequest() : base("api/dataDemoRequest")
    {
    }
}

[DataContract]
private class DataDemoResponse
{
    [DataMember]
    public string data;
}

public async Task<string> SendDataDemo()
{
    var response = await networkFacade.Request<DataDemoResponse>(
        new DataDemoRequest());

    if (!response.success)
    {
        UnityEngine.Debug.LogError(response.error);
        return null;
    }

    return response.result.data;
}
```

Notice the response class attributes.

Closing a connection

At the end of the work, you need to release the resources:

```
networkFacade.Dispose();
```

SERVER DEPLOYMENT

This guide will outline the order of the initial deployment of the meta on the host. As an example, the whole process will be outlined from beginning to the end with the necessary settings to improve security and installation of all necessary system components. Setting up CI for the meta is beyond the scope of this guide, as there are a huge number of CI solutions, each with its own documentation. We are using a server running Ubuntu 20.04 (LTS) as an example.

Key generation on the client

The first thing to do is to organize access to the host via ssh for a specific user. To do this, you need to generate a pair of keys. To generate keys, we recommend using git bash as a linux console emulator on windows and the *ssh-keygen command*. The command is executed with the *-t* flag, which specifies the desired type of key encryption. In this case it is RSA.

```
1. > ssh-keygen -t rsa
```

When generating a key pair, you can specify a password to access it to increase security. After that, two files will appear in the directory specified in the key generation process. The one without extension is the private key and should be kept secret. The other one, which has a .pub extension, is the public key that will be used to configure the host to connect via ssh.

Setting up ssh on the host

Now you need to open a terminal on the host you want to access. Usually, having physical access to the machine or root access to the cloud node is sufficient. So, you need to open the terminal as root and start executing commands in it. First of all, it is important to download and install all updates so that you only deal with the latest, most secure software. To update the system, you need to run the apt-get update command.

```
1. > apt-get update
```

After the update, the time of which depends on the speed of the Internet connection and the power of the host, you can proceed with the initial configuration of the host's security. First, we disable the ability to log in to the root, as well as log in to any user by login and password.

This will provide an additional layer of protection. Go to the */etc/ssh* directory and open the *sshd_config* file for editing.

```
1. > nano -l /etc/ssh/sshd_config
```

In the file, you need to find the lines *PasswordAuthentication*, *PermitEmpty-Passwords* and *UsePAM*, *PermitRootLogin* and put *no* in front of them. Some of them can be commented out with # hash tag. In this case, they should also be uncommented. We then resave the file and restart the sshd daemon like this.

```
1. > systemctl restart sshd
```

Now we will not be able to log in as root, as well as any user, even if we know the login and password. Now you can only connect via ssh using a key. If the key is lost, then you will need to physically have access to the host in order to correct the situation.

IMPORTANT: all this must be done during one session, otherwise, when we log out, we will fall into our own trap.

As an additional security measure, you can change the ssh port on which the connection will be made. The standard port for ssh is 22, change it to whatever you want and then simple ssh <user>@<host> will no longer be enough to connect. It is also configured by analogy with the root login and password. That is, open the */ect/ssh/sshd_config* file and find the *Port* line, opposite which is the working port. Don't forget to save the file and restart the daemon. It will now be necessary to connect like this *ssh -p <port> <user>@<host>*. For extra protection in this file, you can also configure the range of ip addresses of clients from which you can make an ssh connection, or even just one ip address. For more information, see the documentation for the daemon and its commands.

Now you should move on to creating a user who will "run" the meta code as a service. In general, it is a good practice to always use a non-root user for further work with the host. It's safer that way. The *adduser <user>* command is used to create a new user. For example, *adduser demo*.

After that, the system will prompt you to select a password for this user, which will need to be duplicated for correction. Also, in the process of creating a new user, a number of optional questions will be asked, such as indicating the full name, room number, work phone. You

might skip all these questions, pressing enter. The final phrase when creating a user is the confirmation of all entered data.

Then the created user must be assigned the rights to execute various commands. This is done by adding it to the *sudo* group with the *usermod* command.

```
> usermod -aG sudo <user>
```

Again, <user> is your username, in our case *demo*. Then it is necessary to give this user the understanding that the owner of the key can connect to him via ssh. To do this, go to the user's home directory with the command

```
cd ~
```

and make sure it has a .ssh directory with an *authorized_keys* file. You can check this with the *ls* command. If there is no such file or folder, then it must be created by assigning secure read and write permissions. This is done in the following way.

```
> mkdir ~/.ssh
> chmod 700 ~/.ssh
> touch ~/.ssh/authorized_keys
> chmod 600 ~/.ssh/authorized_keys
```

Now you need to open the *authorized_keys* file in any text editor and enter the contents of the public key file, which has already been generated by the client. We recommend using *nano* to work with text in the terminal - it is simple, clean and convenient. To open a file, use this command

```
> nano -l ~/.ssh/authorized_keys
```

After that, you need to open the public key file and copy all its contents to the clipboard, go to the console on the server and paste the contents into the *authorized_keys* file using the SHIFT+INSERT combination. Then you need to execute CTRL + X to exit nano, and answer Y when asked about saving.

Tip: You can use ***git bash*** to copy the entire contents of the public key file. This command copies the contents of the public key to the clipboard.

```
> cat ~/.ssh/id_rsa.pub > /dev/clipboard
```

Now the client can access the server and configure it, which is correct and safe. To connect via ssh to the host, you need to open the console, connect the key in git bash, and then connect via ssh. It is done in this way.

```
> eval "$(ssh-agent -s)"
> ssh-add <path_to_your_private_key>
> ssh <user>@<host>
```

After executing these commands, the terminal will connect to the specified user via ssh and will allow you to work with the server remotely.

Installing dependencies for working with the meta

Meta for its work requires a number of dependencies, among which there are mongo and node. Our target configuration worked fine on **mongo 3.6.8** and **node 14.18.0** as well as **npm 6.14.15**, so we recommend installing those versions or higher. Mongo and a package manager (npm) can be installed the usual way, but a node requires more effort.

To install mongo, you can simply run the following command in the terminal, which will require the password from the corresponding user. It is also best to check its version right away to make sure everything is done correctly. You can also immediately check that mongo has started as a service and that everything is fine with it.

```
> sudo apt-get install mongodb
> sudo mongo -version
> sudo systemctl status mongodb
```

The last command will give a part of the log and service information about the running service as a response. Among this information, it is important to find the word *active*, which indicates the correct operation of the service.
Now you can set the node. Ubuntu distributions usually do not contain the latest versions of the node, so we will install it from other repositories. It is done like this.

```
> cd ~
> sudo apt-get install npm
> curl -sL https://deb.nodesource.com/setup_16.x -o nodesource_setup.sh
> sudo bash nodesource_setup.sh
> rm nodesource_setup.sh
> sudo apt-get install nodejs
> node -version
```

After completing all these steps, the node will be installed and, in the console, you can see the version of the newly installed node. Now you can go to the security settings.

Setting up mongo and its security

Mongo's base needs to be protected from a surprise attack. Therefore, it is best to immediately create a user in it, through which it will be possible to interact with its content. This will also allow you to open access to the database from the outside for other software and, if necessary, safely edit it manually. To work correctly, mongo must have a previously created *sugarspace* database. This can be done by opening mongo's terminal and executing a few commands in it.

```
> mongo
```

Now the database opens immediately, because the user has not yet been created and the password has not been assigned to him, but later it will be impossible to read data from it without a login and password. First of all, we will make sure that we can create and use the *sugarspace* base.

```
> use sugarspace
```

After that, you can go to the *admin* database, where we will create our database user.

```
> use admin
> db.createUser(
{
    user: "<user>",
    pwd: passwordPrompt(),
    roles:
    [
        {
            role: "userAdminAnyDatabase",
            db: "admin"
        },
        "readWriteAnyData-base"
    ]
})
> exit
```

After that, the system will ask you for the user's password and finish setting up access to the database. Now there is a user with a password, but the mongo daemon does not know that access should be restricted. It's set up like this. First, let's open the mongo settings file.

```
> sudo nano -l /etc/mongodb.conf
```

You need to find *auth* in the file and set the field to *true*. In order to organize access from other software directly to mongo in the future, you need to configure *bind_ip* in the same file, adding the ip of your host separated by commas. Thus, access to mongo can be organized from the outside, and not just from the same host. After modifying the settings file, you need to save it and restart the mongo daemon with this command.

```
> sudo systemctl restart mongod
> sudo systemctl status mongod
```

The second command is in addition to the first and provides an overview of how the daemon has been restarted.

Setting up ufw

Now you need to configure the firewall in order to close everything unnecessary and open only what will meta use. In essence, we only need three ports - 22 for ssh access, 4000 for the node and 27017 for the mongo in case we decide to organize access to it from the outside.

Important: if ufw is configured incorrectly, it will block all connections, including ssh, shutting down our host completely.

First of all, we prohibit all connections to the input. To do this, we execute the following command.

```
> sudo ufw default deny incoming
```

Then add the ability to access via ssh

```
> sudo ufw allow ssh
```

After that, add port 4000 to access the node for clients and admins with this command

```
> sudo ufw allow to any port 4000
```

Finally, to open the port for mongo, we use this rule.

```
> sudo ufw allow to any port 27017
```

In the final, we simply activate ufw. It is done in this way.

```
> sudo ufw enable
```

Meta service configuration

Meta runs on the server as a service. This means that for it to work successfully, you need to have the meta core code and correctly configure the service config. Then you only need to start the service and the meta will be ready to go.

You should start by copying the meta code to the host using *scp*. In order for scp to work correctly, you need to load the ssh key into git bash using *eval*. This is done in the same way as in the case of a simple connection to the host via ssh. Then we execute the following command on the client machine that has the meta code (not the host):

```
> scp -r <path_to_meta_core> <user>@<host>:
```

Executing this command will copy the entire meta code to the working directory of the user specified as *user* in this case. Then it is

important to go to the directory with the meta code and import the dependencies using the *npm ci* command.

After that, you need to create a service config so that the process will use when it starts. The process will read environment in which it works. Service configs are stored in */etc/systemd/*system. In this directory, you need to create a config file for our service, for example, *sugarbackend.service*. And here is a template of what the contents of this file should be.

```
[Unit]
Description=SugarSpace backend node.js service
After=network.target mongod.service

[Service]
Type=simple
User=root
ExecStart=node /home/root/sugarbackend/index.js
WorkingDirectory=/home/root/sugarbackend
Environment="CONNECTION=mongodb://127.0.0.1:27017"
Environment="SECRET=XXXXXXXXXXXX"
Environment="PORT=4000"
Environment="NODE_ENV=production"
Restart=always
RestartSec=5
```

This is a template that you can customize as needed. Let's go through some of the properties that will need to be configured.
User – user that will be used for running the process.
ExecStart – node location and path to the entry point for the service.
WorkingDirectory – the location of the working directory for the process.
Environment – a list of environment variables that the service uses during its operation.

The file must be saved, and then reload the service configuration with this command.

```
> sudo systemctl daemon-reload
```

After completing all the above steps, the meta server should be up and running!

Useful meta commands

Start, stop and restart

```
> sudo systemctl start sugarbackend.service
> sudo systemctl stop sugarbackend.service
> sudo systemctl restart sugarbackend.service
```

Monitoring service logs directly on the host

```
> journalctl -u sugarbackend.service -f
```

FRONTEND DEPLOYEMENT

The front itself represents an application written in React. It uses client rendering and no server rendering. A feature of the front is that it uses ReactRouter to implement addressing. This will require certain exercises to properly launch the application as a service.

First of all, you need to access the host via ssh. Given that you must first configure the back, and then the front, you already know how to organize access via ssh, make it safe, and so on.

The next step is to copy the front distribution to the host. To do this, as before, you need to use the *scp* command. We will assume that the distribution kit is successfully copied to home directory of the user, on whose behalf the service will be running. We will also use *build* as the name of the front directory, and *demo* as the user.

After that, you need to wrap the distribution in a node application, redefine routing in it, and run the node application as a service. Create a *sugarwrapper_deploy* folder in the user's home directory. While in it, you need to create an empty project. It's all done with this command.

```
1. > cd ~
2. > mkdir sugarwrapper_deploy
3. > cd sugarwrapper_deploy
4. > npm init -y
```

This will create all the necessary files for the wrapper in the user's home directory. Then, the distribution kit of the front is transferred to this wrapper with this command.

```
1. > mv ~/build ~/sugarwrapper_deploy/build
```

After that, you can start writing wrapper code. In the *sugarwrapper_deploy* directory, you need to create an index.js file, and write the following code in it.

```
const express = require('express');
const path = require('path');
const app = express();

const address = path.join(
__dirname, process.env.BUILD_DIRECTORY || 'build');
const port = process.env.PORT || 3000;

app.use(express.static(address));
app.get('/*', function (req, res)
{
    res.sendFile(path.join(address, 'index.html'));
});

app.listen(process.env.PORT || 3000, () =>
{
    console.log(`Wrapper is started on port ${port}
        and serving address ${address}`);
});
```

We save the file and move on to setting up dependencies and the service. In the project console, we execute *npm install express*, and then we start creating the service config. All the services are located in the */etc/systemd/system* directory. We go there and create a *sugarwrapper.service* file. Inside the file we write the following configuration for the service.

```
[Unit]
Description=SugarWrapper frontend node.js service
After=network.target

[Service]
Type=simple
User=demo
ExecStart=/usr/bin/node /home/demo/sugarwrapper_deploy/index.js
WorkingDirectory=/home/demo/sugarwrapper_deploy
Restart=always
RestartSec=5
```

Here it is important to check the paths where you have dependencies, such as the project, distribution, and so on. If something is not configured correctly, the front will not start as it should.

After that, you need to restart the systemctl daemon and start the service with this command.

```
> sudo systemctl daemon-reload
> sudo systemctl start sugarwrapper.service
```

SERVER API

System login

Obtaining a system authorization token.
username – username
password – password
Request

```
POST /api/login HTTP/1.1
Host: 127.0.0.1:4000
Content-Type: application/json
Content-Length: 60

{
    "username":"Admin",
    "password":"Password"
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9"
    }
}
```

isOkay – response status
data.token – system token

Sending an event

Sending an event to the client with custom content.
id – user identifier
message – event custom content
Request

```
POST /api/sendEvent HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 75

{
    "id": "61caaf1f0a8f67b94f3979f6",
    "message": "event payload"
}
```

Response

```
{
    "isOkay": true
}
```

isOkay – response status

Getting logs

Getting server logs.

Request

```
POST /api/getLogs HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Response

```
[Tue, 28 Dec 2021 06:24:29 GMT] id:1 POST 200 /api/getUserModel REQUEST:
{
    "id": "61b0925d88bfdb6c431d34e0"
}
[Tue, 28 Dec 2021 06:24:29 GMT] id:1 POST 200 /api/getUserModel RESPONSE:
{
    "isOkay": true,
    "data":
    {
        "_id": "61b0925d88bfdb6c431d34e0",
        "deviceId": "548D5A1E56FE",
        "nickname": "Marry",
        "level": 10,
        "currencies":
        [
            {
                "type": "Power",
                "amount": 15
            }
        ],
        "clientVersion": "0.0.0"
    }
}
```

Creating a content entity

Creating a content entity of a specific template.
templateName – template name
entity – entity body
Request

```
POST /api/createEntity HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 142

{
    "templateName":"Abilities",
    "entity":
    {
        "title": "Magic Boost",
        "power": 10,
        "price": 20
    }
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61ca0b03a63d93895732b18c",
        "title": "Magic Boost",
        "power": 10,
        "price": 20
    }
}
```

isOkay – response status
data – the body of the created entity

Deleting a content entity

Removing the content entity of a specific template.
templateName – template name
id – entity identifier
Request

```
POST /api/deleteEntity HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 74

{
    "templateName":"Abilities",
    "id":"61ca0b03a63d93895732b18c"
}
```

Response

```
{
    "isOkay": true,
    "data": "61ca0b03a63d93895732b18c"
}
```

isOkay – response status
data – removed entity identifier

Updating a content entity

Update the content entity of a specific template.
templateName – template name
id – entity identifier
entity – entity body
Request

```
POST /api/updateEntity HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 175

{
    "templateName":"Abilities",
    "id":"61ca0b7fa63d93895732b18f",
    "entity": {
        "title": "Magic Boost",
        "power": 10,
        "price": 20
    }
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61ca0b7fa63d93895732b18f",
        "title": "Magic Boost",
        "power": 10,
        "price": 20
    }
}
```

isOkay – response staus
data – updated entity body

Getting a content entity

Getting the content entity of a specific template.
templateName – template name
id – entity identifier
Request

```
POST /api/getEntity HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 76

{
    "templateName": "Abilities",
    "id": "61ca0b7fa63d93895732b18f"
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61ca0b7fa63d93895732b18f",
        "title": "Magic Boost",
        "power": 10,
        "price": 20
    }
}
```

isOkay – response status
data – entity body

Getting content entities

Getting content entities of a specific template.
templateName – template name
pageIndex – page index
pageSize – page size
Request

```
POST /api/getEntities HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 76

{
    "templateName":"Abilities",
    "pageIndex":0,
    "pageSize":10
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "entities": [
            {
                "_id": "61ca0b46a63d93895732b18e",
                "title": "Magic Boost",
                "power": 10,
                "price": 20
            }
        ],
        "totalCount": 1
    }
}
```

isOkay – response status
data.entities – current content entities page
data.totalCount – the number of entities in the entire collection

Create a template

Create a specific template.
name – template name
shared – template type (normal or shared)
schema – template schema
Request

```
POST /api/createTemplate HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 157

{
    "name":"Abilities",
    "shared": false,
    "schema":
    {
        "title": "string",
        "power": "int",
        "price": "int"
    }
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61ca09cfa63d93895732b18b",
        "name": "Abilities",
        "schema": {
            "title": "string",
            "power": "int",
            "price": "int"
        },
        "shared": false
    }
}
```

isOkay – response status
data – body of a created template

Delete a template

Removing a specific template and all of its entities.
name – template name
Request

```
POST /api/deleteTemplate HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 26

{
    "name":"Ability"
}
```

Response

```
{
    "isOkay": true,
    "data": "Ability"
}
```

isOkay – response status
data – name of a deleted template

Update a template

Updating a specific template.
name – template name
shared – template type (normal or shared)
schema – template shema
Request

```
POST /api/updateTemplate HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 157

{
    "name":"Abilities",
    "shared": false,
    "schema":
    {
        "title": "string",
        "power": "int",
        "price": "int"
    }
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61ca09cfa63d93895732b18b",
        "name": "Abilities",
        "schema": {
            "title": "string",
            "power": "int",
            "price": "int"
        },
        "shared": false
    }
}
```

isOkay – response status
data – updated template body

Getting a template

Getting a specific template.
templateName – template name
Request

```
POST /api/getTemplate HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 37

{
    "templateName": "Abilities"
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61ca09cfa63d93895732b18b",
        "name": "Abilities",
        "schema": {
            "title": "string",
            "power": "int",
            "price": "int"
        },
        "shared": false
    }
}
```

isOkay – response status
data – template body

Getting templates

Getting all the templates of a specific type.
pageIndex – page index
pageSize – page size
shared – templates type (normal or shared)
Request

```
POST /api/getTemplates HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 64

{
    "pageIndex":0,
    "pageSize":10,
    "shared": true
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "templates": [
            {
                "_id": "61c74deba63d93895732b179",
                "name": "Money",
                "schema": {
                    "root": {
                        "field": "int"
                    }
                },
                "shared": true
            }
        ],
        "totalCount": 1
    }
}
```

isOkay – response status
data.templates – page with templates
data.totalCount – total number of templates of a specific type

Game authentication

Getting a game token and a player model.
deviceId – player device ID
Request

```
POST /api/gameLogin HTTP/1.1
Host: 127.0.0.1:4000
Content-Type: application/json
Content-Length: 27

{
    "deviceId": "000"
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "user": {
            "_id": "61caae2455fd074320657cba",
            "deviceId": "000",
            "name": "default_user",
            "level": 1,
            "clientVersion": "0.0.0"
        },
        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9"
    }
}
```

isOkay – response status
data.user – player model
data.token – game token

Getting game content

Getting game content.

Request

```
POST /api/gameContent HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Response

```
{
    "isOkay": true,
    "data": {
        "content": [
            {
                "name": "Weapons",
                "entities": [
                    {
                        "_id": "61af4dc7b26ba2405ba8de56",
                        "title": "Axe",
                        "ability": {
                            "title": "Power"
                        }
                    }
                ]
            },
            {
                "name": "Abilities",
                "entities": [
                    {
                        "_id": "61ca0b46a63d93895732b18e",
                        "title": "Magic Boost",
                        "power": 10,
                        "price": 20
                    }
                ]
            }
        ]
    }
}
```

isOkay – response status
data.content – game content

Refresh content cache

Refresh content cache.

Request

```
POST /api/gameRefresh HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Response

```
{
    "isOkay": true,
    "data": {
        "content": [
            {
                "name": "weapons",
                "entities": [
                    {
                        "_id": "61af4dc7b26ba2405ba8de56",
                        "title": "Axe",
                        "ability": {
                            "title": "Power"
                        }
                    }
                ]
            }
            {
                "name": "Abilities",
                "entities": [
                    {
                        "_id": "61ca0b46a63d93895732b18e",
                        "title": "Magic Boost",
                        "power": 10,
                        "price": 20
                    }
                ]
            }
        ]
    }
}
```

isOkay – response status

data.content – game content

Server events

Server events subscription.

Request

```
GET /api/events HTTP/1.1
Host: 127.0.0.1:4000
Content-Type: text/event-stream
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Response

```
retry: 10000

id: 1640679955604
data: "event payload"

id: 1640679963891
data: "event payload"
```

retry – number of milliseconds in case of reconnection
id – event unique identifier
data – custom message content

Getting the content schema

Getting the content schema.

Request

```
POST /api/getSchema HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Response

```
{
    "isOkay": true,
    "data": {
        "templates": [
            {
                "_id": "61ca09cfa63d93895732b18b",
                "name": "Abilities",
                "schema": {
                    "title": "string",
                    "power": "int",
                    "price": "int"
                },
                "shared": false
            }
        ],
        "userTemplate": {
            "_id": "61ca0c0da63d93895732b190",
            "schema": {
                "root": {
                    "deviceId": "string",
                    "nickname": "string",
                    "level": "int"
                }
            }
        }
    }
}
```

isOkay - response status
data.templates - content templates
data.userTemplate - player template

Getting player template

Getting player template.

Request

```
POST /api/getUserTemplate HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61ca0c0da63d93895732b190",
        "schema": {
            "root": {
                "deviceId": "string",
                "nickname": "string",
                "level": "int"
            }
        }
    }
}
```

isOkay – response status
data – player template

Update player template

Update player template.
schema – player template schema
Request

```
POST /api/setUserTemplate HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 330

{
    "schema":
    {
        "root":
        {
            "deviceId": "string",
            "nickname": "string",
            "level": "int"
        }
    }
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61ca0c0da63d93895732b190",
        "schema": {
            "root": {
                "deviceId": "string",
                "nickname": "string",
                "level": "int"
            }
        }
    }
}
```

isOkay – response status
data – update player template

Getting player model

Getting player model.
id – player identifier
Request

```
POST /api/getUserModel HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 42

{
    "id": "61b0925d88bfdb6c431d34e0"
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61b0925d88bfdb6c431d34e0",
        "deviceId": "548D5A1E56FE",
        "nickname": "Marry",
        "level": 10,
        "clientVersion": "0.0.0"
    }
}
```

isOkay – response status
data – player model

Update player model

Player model update. All the fields of the old model are destroyed. There is no data merging.
id – player identifier
model – player model itself (body)
Request

```
POST /api/setUserModel HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 325

{
    "id": "61b0925d88bfdb6c431d34e0",
    "model":
    {
        "deviceId": "548D5A1E56FE",
        "nickname": "Marry",
        "level": 10,
        "clientVersion": "0.0.0"
    }
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "_id": "61b0925d88bfdb6c431d34e0",
        "deviceId": "548D5A1E56FE",
        "nickname": "Marry",
        "level": 10,
        "clientVersion": "0.0.0"
    }
}
```

isOkay – response status
data – updated player model

Deleting a player model by specifying a player id

Deleting a player model.
id - player identifier
Request

```
POST /api/deleteUserModel HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 42

{
    "id": "61b0925d88bfdb6c431d34e0"
}
```

Response

```
{
    "isOkay": true,
    "data": "61b0925d88bfdb6c431d34e0"
}
```

isOkay - response status
data - deleted player identifier

Deleting a player model by specifying device id

Deleting a player model.
deviceId – device identifier
Request

```
POST /api/deleteUserDevice HTTP/1.1
Host: 127.0.0.1:4000
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
Content-Type: application/json
Content-Length: 27

{
    "deviceId": "000"
}
```

Response

```
{
    "isOkay": true,
    "data": {
        "userId": "61a6639149560e279c09147a"
    }
}
```

isOkay – response status
data.userId – deleted player identifier

COMPARISON WITH COMPETITORS

The comparison took into account aspects of the niche that SugarSpace claims. The main characteristics are the management of content and player data, as well as authoritative server logic.
There will be no such aspects as leaderboards, matchmaking, or A/B testing tools in comparison. It is important to keep in mind that the platform, although it does not provide such tools out of the box, is built with extensibility in mind and does not prevent their integration in any way.
Therefore, the comparison will be carried out in terms of managing content and user data, evaluating the work of authoritative server scripts.

Back4App

Cons:
- The documentation is client-oriented - all work with the server is described according to the principle of implementing logic on the client, and the server is just a safe storage.
- Weak security - by default, everything is available and open to any user. This is great at the development stage, but in production it can become a problem.
- Built on the basis of an open source product, which makes regular updates impossible, and the server itself more vulnerable.
- Product support is officially discontinued.
- The service is distributed on a subscription basis.

Pros:
- There is support for the GraphQL Request language.
- The ability to save data in Blockchain.
- Automatic increase in power under load.

GameSparks

Cons:
- Fixed user model, it is impossible to add fields.
- Very weak work with content: you can install data and delete them. That’s it.
- Only two data types are supported: strings and numbers.
- On the client there is no strict typing.

Pros:
- Developed system of rights for dashboard operators.
- Automatic increase in power under load.
- Support for analytics out of the box.

Playfab

Cons:
- Strict restrictions on the number of Requests and on the amount of data of both the content and the user.
- User data is split into player data and game data.
- Content is one collection of key and value pairs.
- No strong typing.

Pros:
- Users from different games are connected to one central account.

ChilliConnect

Cons:
- The most expensive subscription of all.

Pros:
- It is possible to write scripts directly in the dashboard.
- It is possible to create scripts that are activated by time.
- Strong content system. It is possible to write types through the schema.

GameLift

Cons:
- No work with content, only realtime.

Pros:
- Convenient when working in a group with other services from Amazon.

SteamWorks

Cons:
- Focused on the Steam distribution platform.
- The content system is limited by the ability to store user files.

Pros:
- It is possible to integrate only the necessary services.

Our unique pros

- The most mature and advanced system for working with content and data:
    - Convenient work with content for designers.
    - Validation of content data and user models.
    - Implementation of custom types directly in the dashboard.
    - Common user and content data types.
- Generation of typed content classes and user model:
    - Simplification of work with data on the client.
    - No need to write routine code.
- Automatic synchronization of server and client models:
    - Client model update code is generated automatically.
    - Acceleration of development and elimination of synchronization errors.

- Server oriented implementation of authoritative logic:
    - Automatic formation of a diff package to the user model.
    - Easy access to services within the server logic.
    - Easy integration of third-party plugins.

Why choose SugarSpace

- The platform is strictly focused on mobile products.
- Usually, the implementation of such a system for a particular game requires at least a client developer, a dashboard developer and one server developer.
- The implementation of such a system will take from one to two years.
- A universal system requires high expertise of all three developers, while a system for a specific game excludes multiple use.
- The system cannot be used during development, debugging and testing.
- A high probability of writing a suboptimal system under the pressure of the game development deadlines.