Add Backengine to your Unity project

Prerequisites

- Install or update Unity Editor to its latest version or at least 2018.4 lts version.
- Sign in to Backengine (If you don't have an account, just register one, it's FREE).

STEP 1: Create a Backengine App

Before you can add Backengine to your Unity project, you need to create a Backengine App to connect to your Unity project.

Create Backengine App

1. In the Backengine homepage, click `Dashboard`.
2. Login to dashboard with your account credentials.
3. In the Backengine dashboard, click `Apps`, then click `Create An App` or (+) button.
4. Insert you App information.
5. Click `Add App`.

Backengine automatically provisions resources for your Backengine App. When the process completes, you'll be taken to the overview page for your Backengine App

STEP 2: Add Backengine Unity SDK

You can add Backengine Unity SDK to your Unity project using the Unity Assets Store, or you can install the SDK manually.

Using the Unity Assets Store

- In your Unity Editor, select menu `Windows/Asset Store` to open Asset Store tab.
- In Asset Store page, search `Backengine SDK` and go into it.
- Click `Download` then `Import` to import Backengine Unity SDK to your Unity project.

Manual installation

- In the Backengine homepage, click SDKs menu.
- Click Unity SDK icon to download lastest `Backengine Unity SDK` package.
- In your open Unity project, navigate to Assets > Import Package > Custom Package.
- Select the `BE_Unity_SDK.unitypackage` file.
- In the Import Unity Package window, click `Import`.

Add App secret to your Unity project

- Back to your Backengine App overview page
- You can see an `App Secret` under your App Name
- Copy that `App Secret` string.
- In your Unity project, select `BackConfig` in `BackEngine/Resources` folder.
- Paste `App Secret` string into `App Secret` field.

Add Backengine to your C++ project

Coming soons.

Add Backengine to your Android app

Coming soons.

Add Backengine to your iOS app

Coming soons.

Backengine - Codeless Backend Solution

Nowadays almost all apps and games (apps) need to store data online. It is almost standard. It makes it easy for users to use the application in many places, many devices.

Backengine provides an easy, no-code solution for building a simple back-end for the database and authentication.

Key capabilities

You can create apps that require user authentication. Store data online. Access and edit data on Backengine dashboard or on the application through the APIs.

Backengine can handle most apps without coding logic on the server.

Self-defined user authentication model

You can almost define how you can authenticate users in your apps without having to program logic on the server.

Backengine will allow you to define an Auth table for user authentication. You can define all the data fields you want and determine which one to use to authenticate the user. Therefore, you can authenticate the user normally such as using username / password or simply use the user device id.

Flexible data retrieval

Define data `Schema` and retrieve data dynamically from the apps through the API.

Backengine SDKs will help you access and edit data flexibly without knowing the database languages. All you need is to focus on building user-side apps.

Understand Backengine App/ Schema/ Document

To build and manage data with backengine. You need to understand some of the concepts that backengine uses.

Backengine App

Every application used Backengine API needs a corresponding Backengine App created on the Backengine dashboard.

Application connecting to backengine via API should provide backengine api a valid `App Secret` of the corresponding Backengine App.

Schema

The schema is like a document describe how you store a specific data type. It also includes other special descriptions so that Backengine can understand and process your data.

There are 3 types of schema:

Normal Schema

This is the default schema type, it does not limit data access and editing rights. It requires providing the app's `App Secret` properly.

It is popular with public data that does not belong to the user, and anyone can access and edit it.

Restrict Schema

This type of schema describes data that cannot be edited by the user via the API. It can only be edited directly on the Backengine dashboard.

It is suitable for fixed data such as level data, or configs.

Auth Schema

This is a special type of schema that allows you to define how the application authenticates users.

You can authenticate your end users the way you want by defining a schema of this type.

Each application has only one `Auth Schema` .

Relation Schema

This type of schema can only be created when there is at least one `Auth Schema`

The data in this schema will always be associated with a `Document` of `Auth Schema`

All the data that end users create in the `Relation Schema` will be refer to a `Document` in the `Auth Schema` .

This is exactly how Backengine authenticates users. A user needs to provide a correct `Document` in the `Auth Schema` , then that user has rights to access to the data refer to that `Document` .

Document

A `Document` is a data record stored as described by a `Schema` . It is like a row in the data table.

All application data will be stored as documents. Based on the `Schema` s you have defined before. Backengine will know how to store and process these data the way you want.

Using Backengine to create a simple login/register and leaderboard app

To **CONTINUE you should finish `*STEP 1*` in `Getting started` session of this documents for your plaftorm.****

Prepair Backengine Project

Create `USERS` schema

- Click `View` button in your app created in above step.
- In your app overview page, click `Create a schema` or (+) button to create a new schema.
- Enter `users` in `Schema Name` box.
- Select `Auth Schema` checkbox to define this is an `Auth Schema` type.
- Click `Next`.

In next steps we create `fields` for `users` schema.

Create `email` field

- Click [+] button on the top right.
- Input `email`, select `String` type, select `Unique`, `Require`, `For Auth`, do not select `Encrypted`.
- Click [✔] to save.

Create `password` field

- Click [+] button on the top right.
- Input `password`, select `String` type, select `Require`, `Encrypted`, `For Auth`, do not select `Unique`.
- Click [✔] to save.

Create `name` field

- Click [+] button on the top right.
- Input `name`, select `String` type, select `Require`, do not select `Encrypted`, `For Auth`, `Unique`.
- Click [✔] to save.

Then click `[CREATE SCHEMA]` button. You should see `users` schema after processing.

Create `SCORE` schema

- In your app overview page, click `Create a schema` or Plus Button button to create a new schema.
- Enter `score` in `Schema Name` box.
- Select `Edit Require Auth` checkbox to define this is an `Relation Schema` type.
- Click `Next`.

In next steps we create `fields` for `users` schema.

Create `score` field

- Click [+] button on the top right.
- Input `score`, select `Number` type, select `Require`, do not select `Encrypted`, `For Auth`, `Unique`.
- Click [✔] to save.

Create `user` reference field.

- Click [+] button on the top right.
- Input `user`, select `Reference` type, select `Require`, do not select `Encrypted`, `For Auth`, `Unique`.
- Select `users` in `Ref` column.
- Click [✔] to save.

Then click `[CREATE SCHEMA]` button. You should see `score` schema after processing.

Download Model Scripts to using in client project.

After created schemas, you can download model scripts belong to your schemas that we generated for you. In `Apps` page. Look at your project. You should see `[Download Models]` button. Click to download.

Using Backengine in Unity sample

You should finish `*STEP 2*` at Add Backengine to your Unity project

Now you can try our leaderboard demo included in `Unity SDK`.

- In Unity Editor, open `Login_Leaderboard_Demo` scene at `Backengine\Demo\Scenes`
- Click register
- You should see a leaderboard here with some score records.
- Click `[+1 POINT]` to add some score then click `[SUBMIT SCORE]`. You should see your score in the leaderboard.
- Now try stop and open `Login_Leaderboard_Demo` again. Login with your registered user before to see if all thing go right.

Understand Leaderboard demo.

Now we will take a closer look to understand how this demo works

First, look at `Backengine\Demo\Scripts\Models` folder. We should see `UserModel.cs` and `ScoreModel.cs` here.
These scripts are `BEModel` class, the model class for the Users and Score schema that you created earlier on the Backengine dashboard.
These classes contain exactly the properties that correspond to the schema fields.
You can create them yourself or simply download them from the Backengine dashboard after creating Schemas.

Register

First, open `UIRegister.cs` Look at

```
UserModel model = new UserModel();
model.email = Email.text;
model.name = Name.text;
model.password = Password.text;
```

We are creating an UserModel with data from input here.

Then we will make a request to the Backengine server.

```
BERequest.Instance.InsertAuth(model, (error, response) => {
  if (!error)
  {
    UserModel user = response.data;
  }

});
```

We call `BERequest.Instance.InsertAuth` to create a new document in `USERS` schema, which is an `Auth Schema` type, that we created on Backengine dashboard.
Pass `UserModel` we created before as parameter. And if there is no `error` we could receive an user data created in `response.data` .

Login

Now, open `UILogin.cs` to see how to authenticate a user.

Look at

```
var requestData = new RequestData<UserModel>();
requestData = requestData.Where(x => (x.email==Email.text) && (x.password== Password.text));
```

To create data for your request, you should create an instance of `RequestData` .
The code above, mean you are creating a `Condition` with an `UserModel` where `email==Email.text` and `password==Password.text` .

Next

```
BERequest.Instance.Auth(requestData, (error, response) =>
  {
    if (!error){
      UserModel user = response.data;
    }
  });
```

Now we call `BERequest.Instance.Auth` to perform user authentication with the requestdata we created.

If you do not want to use `query style` code . You can also make requests using the UserModel like this.

```
UserModel user = new UserModel();
user.email = Email.text;
user.password = Password.text;

BERequest.Instance.Auth<UserModel>(user, (error, response) => {
  if (!error){
    UserModel user = response.data;
  }
});
```

Fetching scores board

Open `UILeaderboard.cs`

```
var requestData = new RequestData<ScoreModel>();
requestData.GetField(x=>x.score).GetRefs(x=>x.user).Sort(x=>x.score, SortType.Desc).Take(1,20);
BERequest.Instance.SelectMany(requestData, (error, response) => {
  if (!error){
    List<ScoreModel> scores =response.data;
    myScore = scores.Find(score => score.userRefs.id == Manager.User.id);
    if (myScore != null)
      BestScore.text = "BestScore: " + myScore.score.ToString();
    else BestScore.text = "BestScore: 0";
  }
});
```

To fetch documents from `SCORE` schema we call `BERequest.Instance.SelectMany` . You should pass a `RequestData` to let server know how want to retrieve data.

```
var requestData = new RequestData<ScoreModel>();
requestData.GetField(x=>x.score).GetRefs(x=>x.user).Sort(x=>x.score, SortType.Desc).Take(1,20);
```

We created a `RequestData` type `ScoreModel` . We want to get `score` field so we call `GetField(x=>x.score)` then we want get `user` field.
If you remember, we defined `user` field in `SCORE` schema as a `Reference` field. That means `user` field should contains `id` from `USERS` schema.
But we don't want only get `user's id` , we want get all user info belong with `score` . To do that, we should use `GetRefs(x=>x.user)` .
Then we wouldn't get `user` field as `user's id` anymore, we would get `userRefs` field as `UserModel` object.
Use `Sort(x=>x.score, SortType.Desc)` to sort data descending by `score` field.
Last, use `Take(1,20)` to get first 20 documents.

Submit a score

In `UILeaderboard.cs` look at `OnSubmitScore` method

```
if (myScore != null){
  myScore.score = currentScore;
  BERequest.Instance.UpdateOne(myScore, (error, response) => {
    if (!error){
      RequestScores();
    }
  });
}
else{
  myScore = new ScoreModel();
  myScore.user = Manager.User.id;
  myScore.score = currentScore;
  BERequest.Instance.Insert(myScore, (error, response) => {
    if (!error){
      myScore = response.data;
      RequestScores();
    }
  });
}
```

In this method, we check if we fetched myScore before
Then we use `BERequest.Instance.UpdateOne` to update myScore ,a `ScoreModel` , to `SCORE` schame. else if we didn't get myScore, that means we hasn't submited a score before.
Then we use `BERequest.Instance.Insert` to insert new score to `SCORE` schema.

Delete a score

Open `ScoreItem.cs` and look at `Delete` method

```
BERequest.Instance.DeleteOne(scoreModel, (error, response) => {
});
```

To delete exactly your score, Call `BERequest.Instance.DeleteOne` and pass `scoreModel` as param.
You should pass exact your score, because `SCORE` schema is a `Relation Schema` with `Require Edit Auth` .

If you pass other score as param, you should receive a response with `403 error code`.

> In this demo we showed you how to authenticate users using `Auth Schema`.
> How to select, insert, update and delete data with Unity SDK.
> For details on how to use it, and all methods in the Unity SDK. You can find them all at `Unity SDK API References`