Elements Unity Plugin

Unity · https://namazustudios.com/docs/game-engine-client-support/unity/elements-codegen/

Github URL:
https://github.com/NamazuStudios/unity-codegen-plugin

Asset Store URL:
https://assetstore.unity.com/packages/tools/integration/namazu-elements-codegen-plugin-for-unity-cross-platform-gbaas-319085

Overview

Requirements

- Unity 2018+
- .Net 4.x enabled in project
- Elements running at an accessible URL
- (Optional) To generate custom application code: Must also have an application created within Elements with the code already uploaded. See the docs here for more info on how to prepare your Element code for client code generation.

Summary

Elements Codegen is a tool that will convert your Elements and application APIs and model definitions and into C# code that is immediately usable.

In addition, there are some convenience classes generated so that you can hit the ground running. These are optional to use, however, so feel free to ignore them if you want to manage everything yourself.

Go to **Window -> Elements -> Elements Codegen** to get started.

You'll need to enter the credentials of an admin/SUPERUSER. If you haven't changed the defaults yet, you should be able to just use root/example as the username/password. See here for more information on accessing the CMS and how to change the default SUPERUSER.

> Important
>
> You must have Elements running at the target root URL. If running locally, then by default this will be `http://localhost:8080`

> Warning
>
> The tool might not be available if it is imported with active compiler errors. If this is the case, please resolve the errors and check again for the tool window.

Generated Code Usage

After generating your code, you can use {packageName}.Client.ElementsClient to initialize the API with the server URL root, and then make any API call. Most properties can be overridden if you prefer to write your own implementation, including object (de)serialization, credentials storage, etc. You can also use the APIs directly if you prefer to manage the requests yourself, or if you prefer a DI based architecture.

Initializing ElementsClient

When you initialize, you will need to specify the root API path. Locally, this will be `http://localhost:8080/api/rest`. Optionally, you can tell ElementsClient to not cache the session if you prefer to do that yourself.

Session Caching

By default, the generated code will use JSON, and store the session in Application.PersistentDataPath. Encryption is not enabled by default.

Request Handling

Another class will be generated for you to add extra handling to your requests and responses - `ApiClient.partial.cs`. This contains two methods - `InterceptRequest` and `InterceptResponse`. By

default, session creating will be handled for you by checking if the response object is of type `SessionCreation` and if so, it will apply the session token to the appropriate request headers for subsequent requests.

See ElementsCodegen/Tests/ElementsTest.cs for an example on how to log in and get the current user (this might be commented out to avoid compiler errors before you generate the Elements API code).

Using the Generated Code

This section covers how to use the C# code generated by the Elements Codegen Tool. It assumes you have already run the tool at least once (**Window → Elements → Elements Codegen**) and that your `Assets/Generated` folder is populated.

Initialization

Singleton (`InitializeDefault`)

The simplest way to get started is to call `InitializeDefault` once at application start. This creates a global instance accessible anywhere via `ElementsClient.Default`.

```
using Elements.Client;

public class AppStartup : MonoBehaviour
{
    void Awake()
    {
        // Pass the root URL of your Elements instance.
        // Locally this is typically http://localhost:8080
        ElementsClient.InitializeDefault("http://localhost:8080");
    }
}
```

`InitializeDefault` is a no-op if a default instance has already been created, so it is safe to call from multiple places.

From any other script:

```
var api = ElementsClient.Default.Api;  // ElementsCoreApi
```

Session Caching

By default, `ElementsClient` persists the session token and profile to disk at `Application.persistentDataPath/ElementsSessionCache_default.json` and restores it on the next launch. You can disable this or use separate cache keys for multiple accounts:

```
// No caching
ElementsClient.InitializeDefault("http://localhost:8080", cacheSession: false);

// Named cache key (useful if you support multiple local profiles)
var client = new ElementsClient("http://localhost:8080", cacheKey: "player2");
```

Sign-up and Login

Registering a new user

`SignUpUserAsync` creates a new account. After sign-up you will typically want to log the user in immediately.

```
using Elements.Api;
using Elements.Model;
using Elements.Client;

public class AuthManager : MonoBehaviour
{
    async void SignUp(string email, string password)
    {
        var request = new UserCreateRequest(
            email: email,
            password: password
        );

        UserCreateResponse newUser = await ElementsClient.Default.Api.SignUpUserAsync(request);
```

```
        Debug.Log($"Created user: {newUser.Id} ({newUser.Email})");

        // Log in right after sign-up
        await Login(email, password);
    }
}
```

`UserCreateRequest` also accepts optional fields such as `name`, `firstName`, `lastName`, and `primaryPhoneNb`.

Logging in

`CreateUsernamePasswordSessionAsync` exchanges credentials for a session. The session token is automatically captured by `ApiClient.partial.cs` and applied to all subsequent requests - you do not need to set any headers manually.

```
async void Login(string userId, string password)
{
    var request = new UsernamePasswordSessionRequest(
        userId: userId,
        password: password
    );

    // SessionCreation is handled transparently; the return value is available
    // if you need to inspect the session directly.
    SessionCreation result = await
ElementsClient.Default.Api.CreateUsernamePasswordSessionAsync(request);

    Debug.Log($"Logged in. Session expires: {result.Session.Expiry}");
}
```

> **Note:** The `userId` field accepts either the user's registered email address or their Elements user ID.

Getting the current user

```
var user = await ElementsClient.Default.Api.GetCurrentUserAsync();
Debug.Log($"Hello, {user.Name}!");
```

Session Management

`ElementsClient` tracks session state. You can query and clear it at any time:

```
var client = ElementsClient.Default;

// Check whether there is a valid, non-expired session
if (client.IsSessionActive())
{
    Debug.Log("Session is active.");
}

// Access the current session and profile directly
Elements.Model.Session session = client.GetSession();
string token = client.GetSessionToken();

// Switch the active profile (updates the Elements-ProfileId header)
client.SetProfile(someProfile);

// Log out (clears session from memory and disk)
client.ClearSession();
```

Using Custom Element APIs

When you generate code for a custom Element (using **Generate Custom Element Code**), an additional API class is created alongside `ElementsCoreApi`. In this example, the Element's serve prefix is `my-game`, and is named `MyGame`, so a `MyGameApi` class will be generated. See the Preparing for Code Generation documentation for more details.

Registering the custom API

Use `CreateAppApi<T>` to instantiate and register a custom API. The `appServePrefix` must match the `dev.getelements.elements.app.serve.prefix` property set in your Element - you can also find it in the **Element Info** panel in the Elements admin console.

`CreateAppApi` takes a factory delegate so that the constructor reference is visible to the linker on IL2CPP platforms (iOS, consoles), preventing it from being removed by managed code stripping.

```
using Elements.Api;
using Elements.Client;

public class AppStartup : MonoBehaviour
{
    void Awake()
    {
        ElementsClient.InitializeDefault("http://localhost:8080");

        // Register the custom Element API. Auth headers and session
        // handling are applied automatically, just like the core API.
        ElementsClient.Default.CreateAppApi<MyGameApi>("my-game", url => new MyGameApi(url));
    }
}
```

`CreateAppApi` both creates the instance and registers it, so you do not need to call `RegisterApi` separately.

Calling custom API methods

Once registered, retrieve the API with `GetApi<T>` and call its methods:

```
public class MatchmakingManager : MonoBehaviour
{
    async void FindMatch()
    {
        var myGameApi = ElementsClient.Default.GetApi<MyGameApi>();

        MultiplayerGameSummary game = await myGameApi.CreateOrFindMatchAsync();

        Debug.Log($"Joined game: {game.Id}");
    }

    async void ChallengePlayer(string opponentProfileId)
    {
        var myGameApi = ElementsClient.Default.GetApi<MyGameApi>();
```

```
        var request = new ChallengePlayerRequest { ProfileId = opponentProfileId };
        MultiplayerGameSummary game = await myGameApi.ChallengeProfileAsync(request);

        Debug.Log($"Challenge sent! Game ID: {game.Id}");
    }
}
```

Registering a pre-constructed API instance

If you need to configure the API before registering it (e.g. to set a custom base URL or timeout), you can construct it manually and pass it to `RegisterApi`:

```
var config = new Elements.Client.Configuration
{
    BasePath = "http://localhost:8080/app/rest/my-game",
    Timeout = TimeSpan.FromSeconds(30)
};

var MyGameApi = new MyGameApi(config);
ElementsClient.Default.RegisterApi(MyGameApi);
```

Raw HTTP Responses

Every generated method has a `WithHttpInfo` variant that returns `ApiResponse<T>` instead of `T` directly. This gives you access to the HTTP status code and response headers without any extra overhead.

```
ApiResponse<UserCreateResponse> response =
    await ElementsClient.Default.Api.SignUpUserWithHttpInfoAsync(request);

if (response.StatusCode == HTTPStatusCode.OK) // 200
{
    Debug.Log($"Sign-up succeeded: {response.Data.Id}");
}
else
{
    Debug.LogWarning($"Unexpected status: {response.StatusCode}");
```

```
}
```

> **Note on exceptions:** `ExceptionFactory` is set to `null` on all APIs registered through `ElementsClient`. This means failed requests will **not** throw an `ApiException` automatically - you should check `ApiResponse.StatusCode` when you need to handle error responses explicitly.

Manual / DI-based Initialization

If you prefer to manage the `ElementsClient` instance yourself - for example, when using a dependency injection framework - skip `InitializeDefault` and construct the client directly:

```
public class GameServices
{
    public ElementsClient Elements { get; }

    public GameServices()
    {
        Elements = new ElementsClient(
            instanceRootUrl: "http://localhost:8080",
            cacheSession: true
        );

        // Register any custom APIs at construction time
        Elements.CreateAppApi<MyGameApi>("my-game", url => new MyGameApi(url));
    }
}
```

Then inject `GameServices` (or `ElementsClient` directly) wherever you need it. All session handling, header injection, and caching behaviour is identical to the singleton path.

Support

For questions or issues, open a GitHub issue with your Unity version, the relevant error logs, and a description of the steps to reproduce.