WWW Script Generator

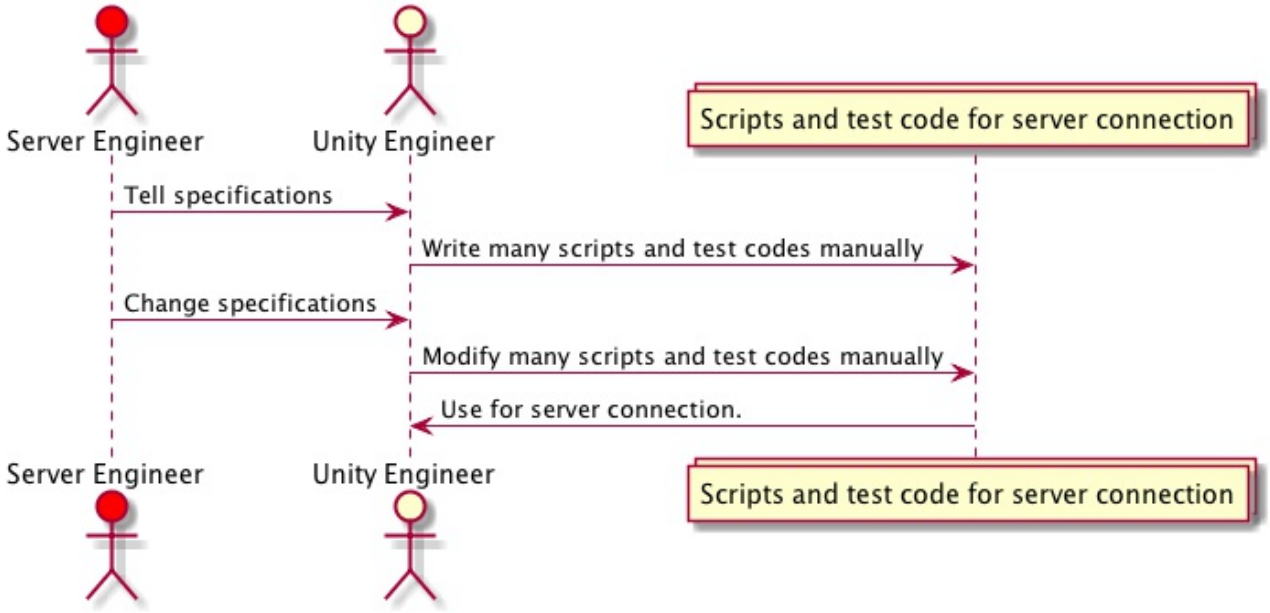
更新履歴

2018.2.9

- 基本設定画面に Handle Cookie と PlayerPrefs key for Cookie の項目を追加
- メソッドの Endpoint に、パラメーターの入る位置を指定する方法の説明を追記
- 'APITaskのカスタマイズ' の説明に '基底クラスのクラス変数'、'基底クラスのクラスメソッド'、'APITask のクラスメソッド' を追加
- 'APITaskのカスタマイズ' の 'CreateHeader' に補足説明を追加

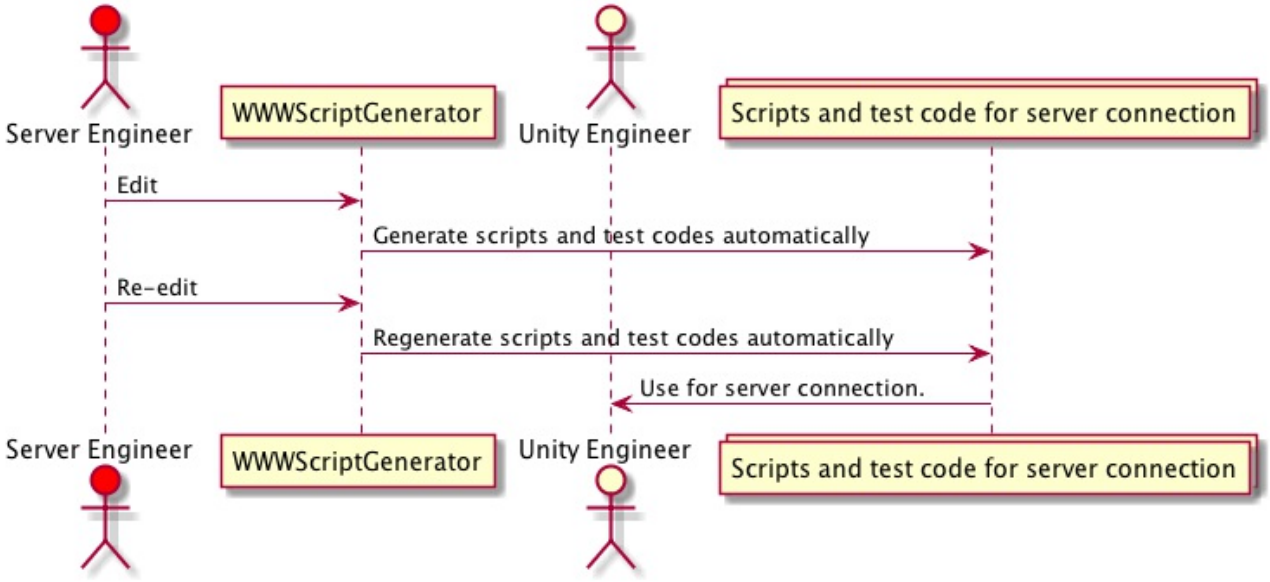
"WWW Script Generator" とは

サーバーとの通信は最近のゲーム開発では必要不可欠になってきています。ですが、Unityの開発者とサーバーの開発者は別の人が担当していることが多く、仕様についてのコミュニケーションが多く発生し、伝達ミスも起きる可能性があります。
また、APIの数が多いとそのために作成しなければならないスクリプトも多くなり、コーディングに膨大な時間を要します。
それに仕様が固まってない段階で開発が始まることも多く、頻繁に仕様変更が発生し、そのたびにUnityの開発者は変更になった箇所を把握しつつスクリプトを修正しなければなりません。
さらに通信テストにも多くの工数を必要とします。

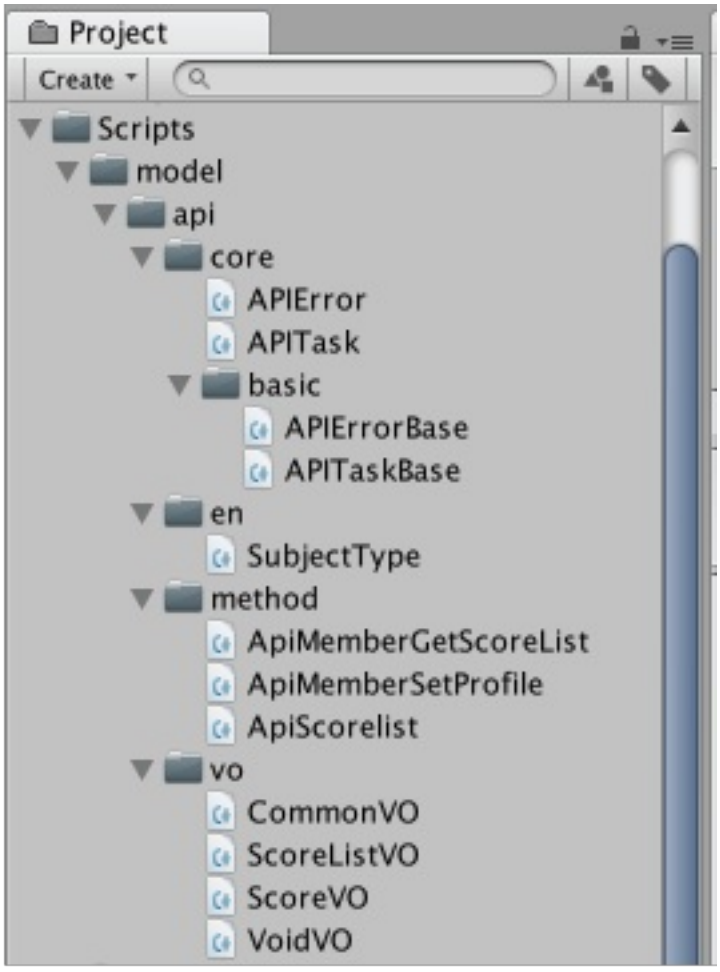


このパッケージを使えば、仕様を設定ファイルにしてサーバー開発者と共有することで、仕様の伝達ミスをなくすことができます。
また、スクリプトの自動生成機能により、スクリプト作成にかかる時間を大幅に短縮できます。
仕様が変更になった場合でも、変更箇所を明確に把握しスクリプトを再生成することで簡単に対応できます。

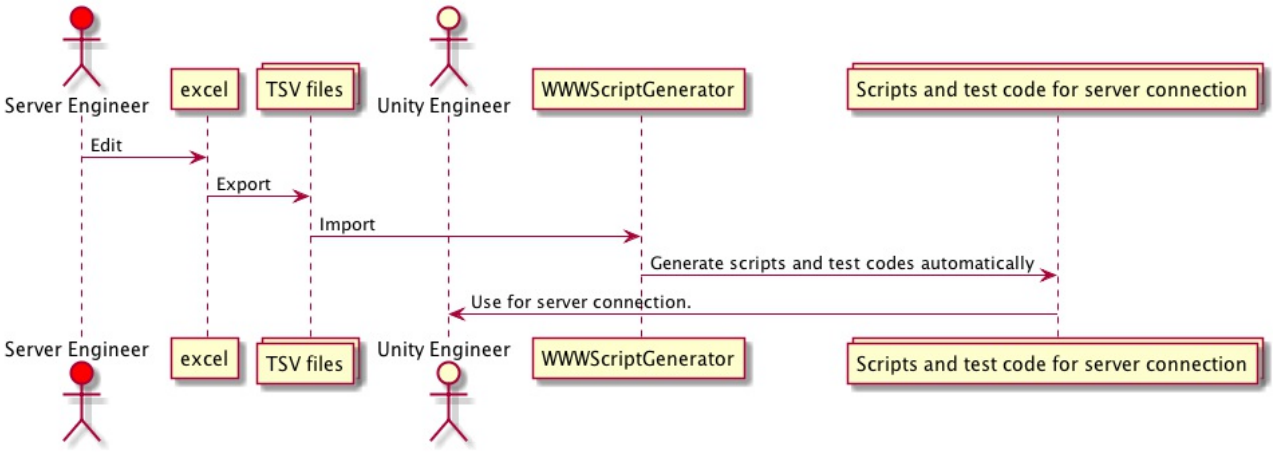
そして、スクリプトと共に生成されるテストコードを使って、通信テストにかかる時間を大幅に短縮できます。



自動生成されるスクリプトはサーバーAPIに特化しているので、他のロジックと明確に分離することができ、可読性が向上します。
通信につかうクラスはUnity標準の"WWW"で、このパッケージ特有の機能は使っていません。生成したスクリプトはパッケージのdllには依存しておらず、書き出されたクラスをカスタマイズすることで、ヘッダー情報や送信データのフォーマットなどを調整できるので、あらゆる通信に対応できます。
JSONをパースするためのモデルクラスも自動生成されるので、JSONをそのまま使うより安全に扱うことができます。



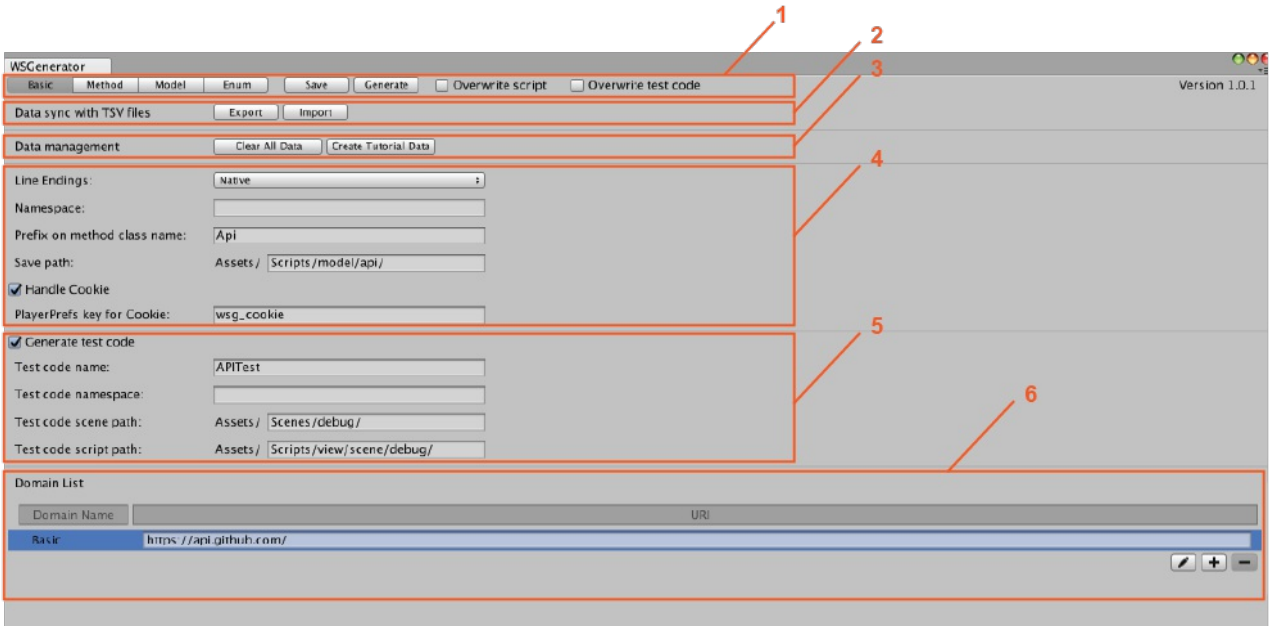
さらに、Pro版ではTSVファイルの読み書きができるので、エクセルなどの外部ツールと連携でき、Unityをもっていないサーバー開発者との連携が可能になります。



画面の説明

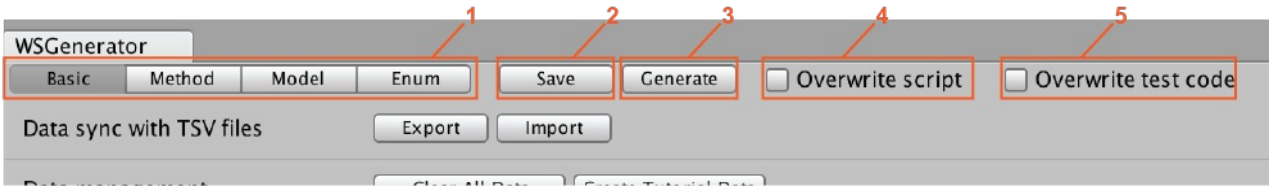
まずはじめに、Unityのメニューから、*"Window / WWW Script Generator"* を選んで、ウィンドウを開きます。

Basic (基本設定)



1. 共通メニュー: 各画面共通のメニューです
2. データ共有設定: 設定ファイルをTSVとして読み込み／書き出しします (※ この機能はPro版限定です)
3. データ管理: データの消去、チュートリアルデータの作成を行います
4. コード生成・挙動設定:　コードの生成や挙動に関する設定です
5. テストコード生成設定: テストコード生成に関する設定です
6. ドメインリスト: 接続先のURIを定義します

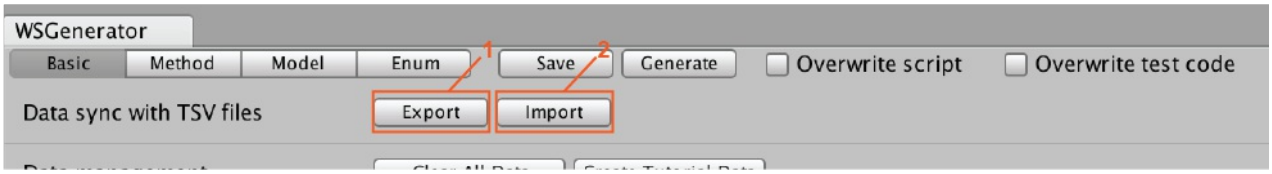
共通メニュー



1. 画面切り替えタブ
    - Basic: 基本設定への切り替え
    - Method: メソッドリストへの切り替え
    - Model: モデルリストへの切り替え
    - Enum: enumリストへの切り替え
2. Save: 設定の保存ボタン
3. Generate: スクリプト生成ボタン
4. Overwrite script: スクリプト生成の際、強制的にカスタマイズ用のcsファイルを上書きします。
5. Overwrite test code: テストコード生成の際、強制的にカスタマイズ用のcsファイルを上書きします。

※ カスタマイズ用のcsファイルついては「スクリプト／テストコードの生成」で説明します

データ共有設定



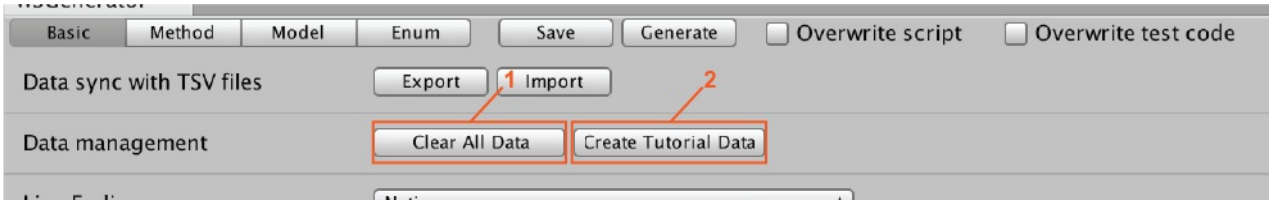
1. Export: 設定ファイルをTSVとして書き出します。
2. Import: TSVファイルを読み込みます。

※ この機能はPro版限定です。

TSVファイルは、それぞれ以下の画面の内容と同等です。

- wsg_domain.tsv: 基本設定のDomain List
- wsg_common.tsv: メソッドリストの共通パラメーター
- wsg_method.tsv: メソッドリストおよびパラメーター
- wsg_model.tsv: モデルリストおよびプロパティ
- wsg_enum.tsv: enumリストおよび値

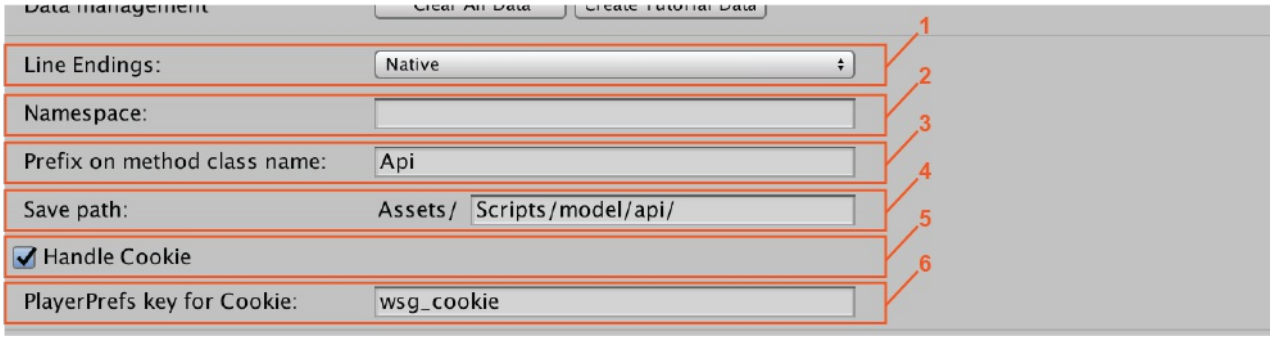
データ管理



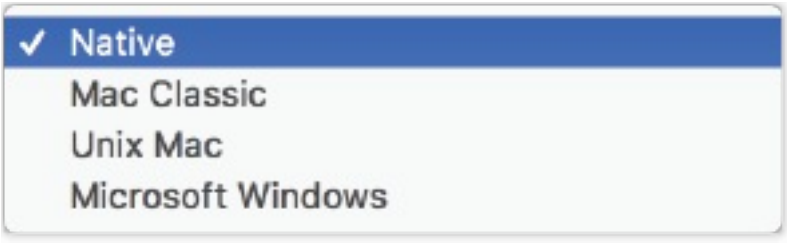
1. Clear All Data: すべてのデータを消去します
2. Create Tutorial Data: チュートリアル用のデータを作成します

※ チュートリアル用のデータ（仮のメソッドとモデルのデータ）は、はじめての方が使用方法を学ぶのに最適です。

コード生成・挙動設定:



1. Line Endings: 書き出すスクリプトの改行コードを指定します。
    - Native: OSの改行コードに合わせます
    - Mac Classic: \r(CR)：古いMac OS（9以前）
    - Unix Mac: \n(LF)：Unix、Mac OS X
    - Microsoft Windows: \r\n(CR+LF)：Windows



2. Namespace:　書き出すスクリプトにつけるネームスペースを指定します。ネームスペースを指定しない場合は空白にします。

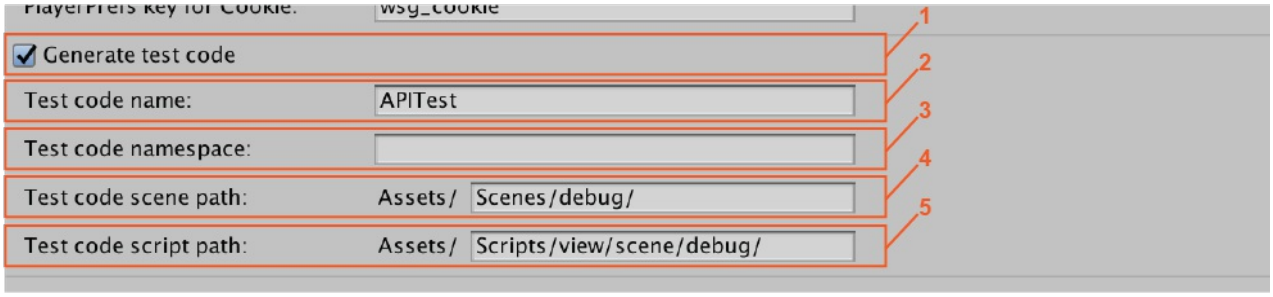
3. Prefix on method class name: 書き出すメソッドのクラス名につける接頭辞を指定できます。
メソッドが大量にある場合、コーディング時に他のクラスと区別して見つけやすくするために接頭辞を指定します。
指定しない場合は空白にします。

4. Save path: 生成するスクリプトを保存するディレクトリを指定します。

5. Handle Cookie: クッキーを扱うかどうかを指定します。 受信したデータのヘッダーに含まれているCookieをPlayerPrefsに保存し、それを送信データのヘッダーに含めます。

6. PlayerPrefs key for Cookie: PlayerPrefsにクッキーを保存するためのキー名を指定します。

テストコード生成設定



1. Create test code: スクリプト生成時に、同時にテストコードも生成するかどうかを指定します。
2. Test code name: テストコードのファイル名（シーン名、スクリプト名）を指定します。
3. Test code namespace: テストコードにつけるネームスペースを指定します。ネームスペースを指定しない場合は空白にします。
4. Test code scene path: テストコードのシーンファイルを保存するディレクトリを指定します。
5. Test code script path: テストコードのスクリプトファイルを保存するディレクトリを指定します。

ドメインリスト

サーバーの接続先を指定します。接続先が複数ある場合は+ボタンで項目を追加します。

1. Domain Name: 接続先の名前
2. URI: 接続先のURI
3. 接続先の名前を編集します
4. 項目を増やします

5. 選択中の項目を削除します

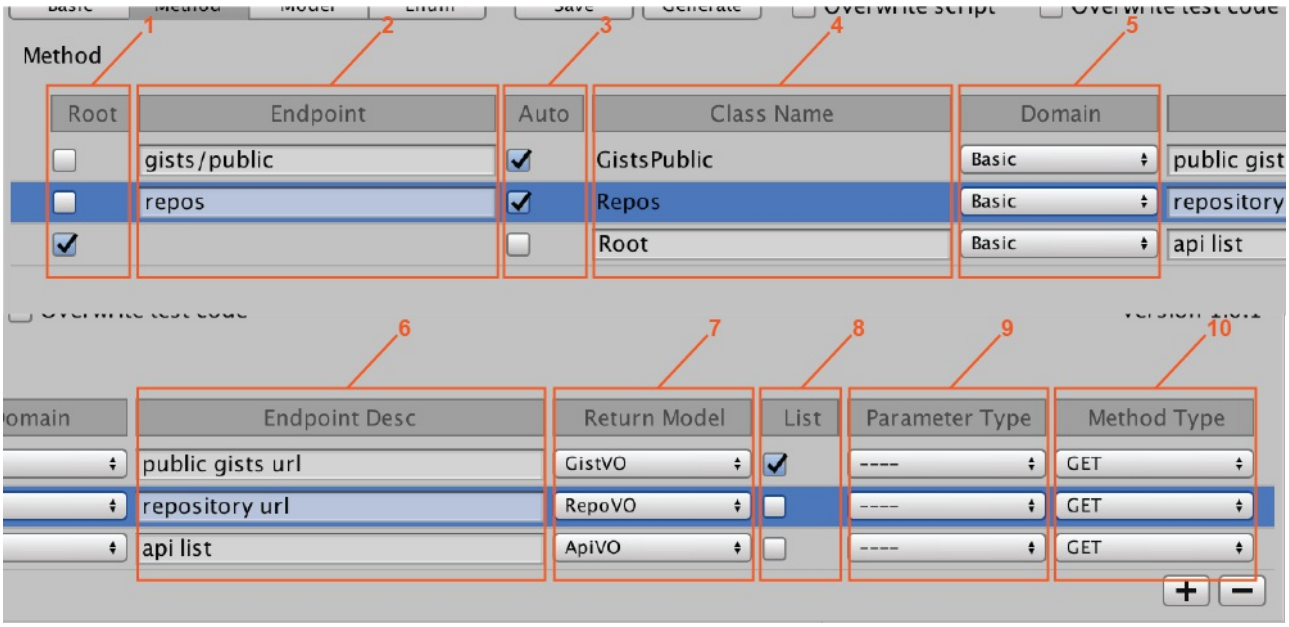
メソッドリスト

生成されるメソッドクラスは、インスタンス化するのではなく、クラスメソッドのAccessを呼んで使用します。例えば ApiScorelist というメソッドを呼ぶ場合、次のように呼びます。

```
ApiScorelist.Access(memberId, (vo, rawText) =>
{
    Debug.Log("Success " + vo.ToString());
}
,
(APIError err, string rawText)
{
    Debug.Log("Error " + err.Code + ", " + err.Message);
}, gameObject);
```

1. Method: メソッドの一覧
2. Parameter: メソッドのパラメータ

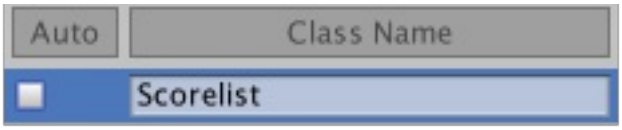
メソッドの一覧

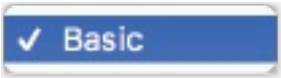


1. Root: エンドポイントが無く、Domainリストで指定したURIをそのまま使用する場合にチェックを入れます。

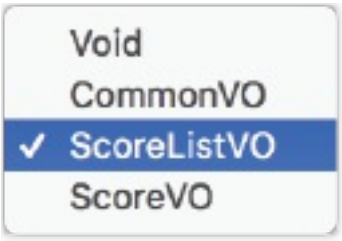
2. Endpoint: メソッドのエンドポイントです。Domainリストで指定したURIに続く文字列を書きます。もしエンドポイントにパラメーターが入る場合、パラメータの「URL」にチェックを入れて、パラメーター名を{}で囲みエンドポイントに入れます。例えば user/getdata/ に続いて user_id という名前のパラメーターが入る場合、エンドポイントは user/getdata/{user_id} になります。パラメーターを省略して user/getdata のままでも、URLにチェックが入っていれば、パラメーターはエンドポイントの後ろにくっつきます。

3. Auto: これにチェックをいれると、エンドポイントから自動でこのメソッドのクラス名を作ります。.jsonや.phpなどの拡張子は自動生成される名前には含まれません。

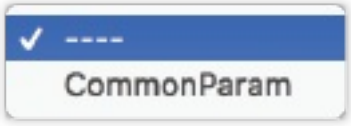
4. Class Name: Autoにチェックを入れてない場合、手動でこのメソッドのクラス名を書きます。


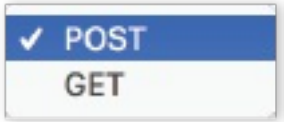
5. Domain: ドメインリストから、どのドメインのURIを使うのかを指定します。


6. Endpoint Desc: このエンドポイントがどのような目的のものかを記述すると、生成されたスクリプトにコメントとして記載されます。

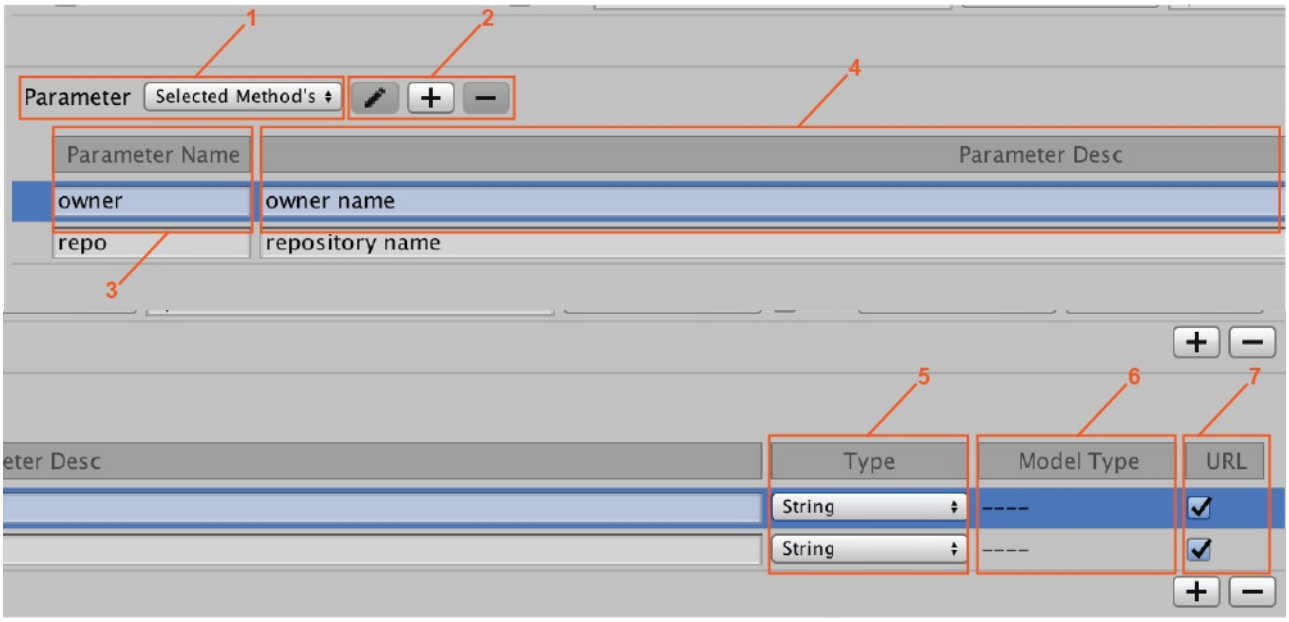
7. Return Model: モデルの中からメソッドの戻り値を指定します。戻り値が無い場合は“Void”を選びます。


8. List: 戻り値のJSONが配列の場合に指定します。
例えばJSONが
`{"list":[{"name":"a"}, {"name":"b"}]}`
ではなく
`[{"name":"a"}, {"name":"b"}]`
のような形で返ってくる場合には、これにチェックを入れます。

9. Parameter Type: 共通パラメーターを指定します。指定しない場合は ---- を選びます。


10. Method Type: 通信方法を POST　か GET かのいずれかを選びます。


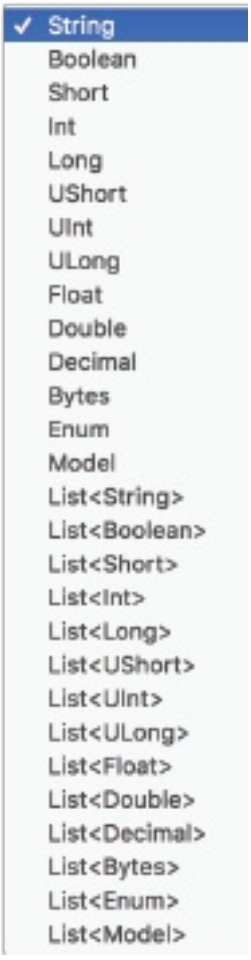
メソッドのパラメータ

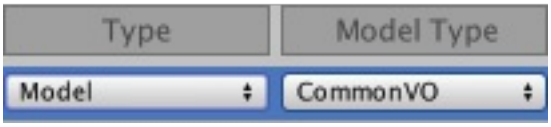


1. いま選択されているメソッド用のパラメーターか、共通パラメーターかを選びます。

   - Selected Method's: いま選択されているメソッドのパラメーター
   - それ以外: 共通パラメーター

2. 共通パラメーターを、編集、追加、削除します。
3. Parameter Name: パラメーター名を指定します。実際にサーバーと通信する際のサーバーに送るキー名と同じにします。
4. Parameter Desc: パラメーターの説明を記述すると、生成されたスクリプトにコメントとして記載されます。
5. Type: パラメーターの型を指定します。



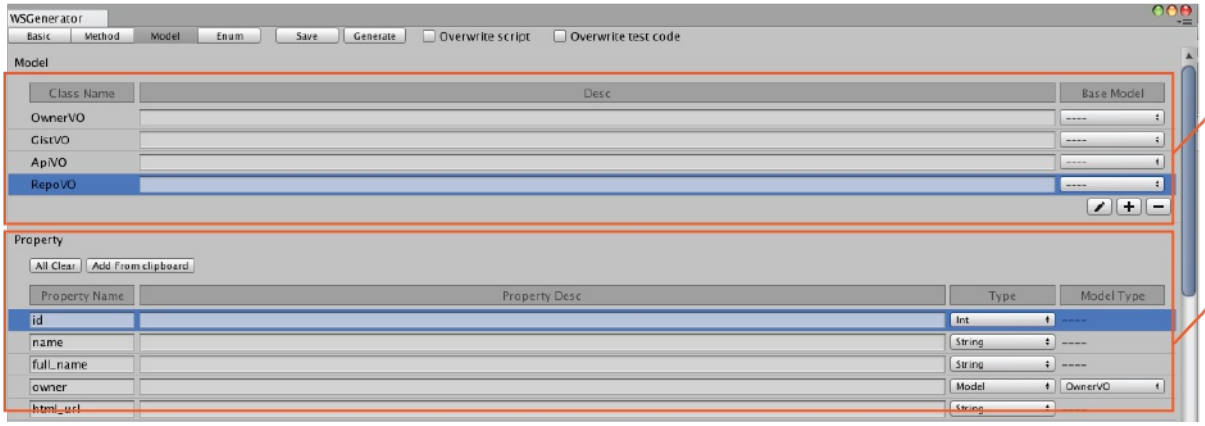
6. Model Type: 型が `Model`, `Enum`, `List<Model>`, `List<Enum>` のいずれかの時、どれを使うのかを選びます。
   `Model`, `List<Model>`の場合はModelの中から選びます。
   

   `Enum`, `List<Enum>`の場合はEnumの中から選びます。
   

7. URL: パラメーターがURLの中に入るのかどうかを指定します
   例えば "param1"というパラメーターの値が"value1"だった場合、実際のURLは
   `https://example.com/value1`
   のようになります。もしエンドポイントにパラメーター名が{}で囲まれて入っていた場合、その位置にパラメーターの値が入ります。

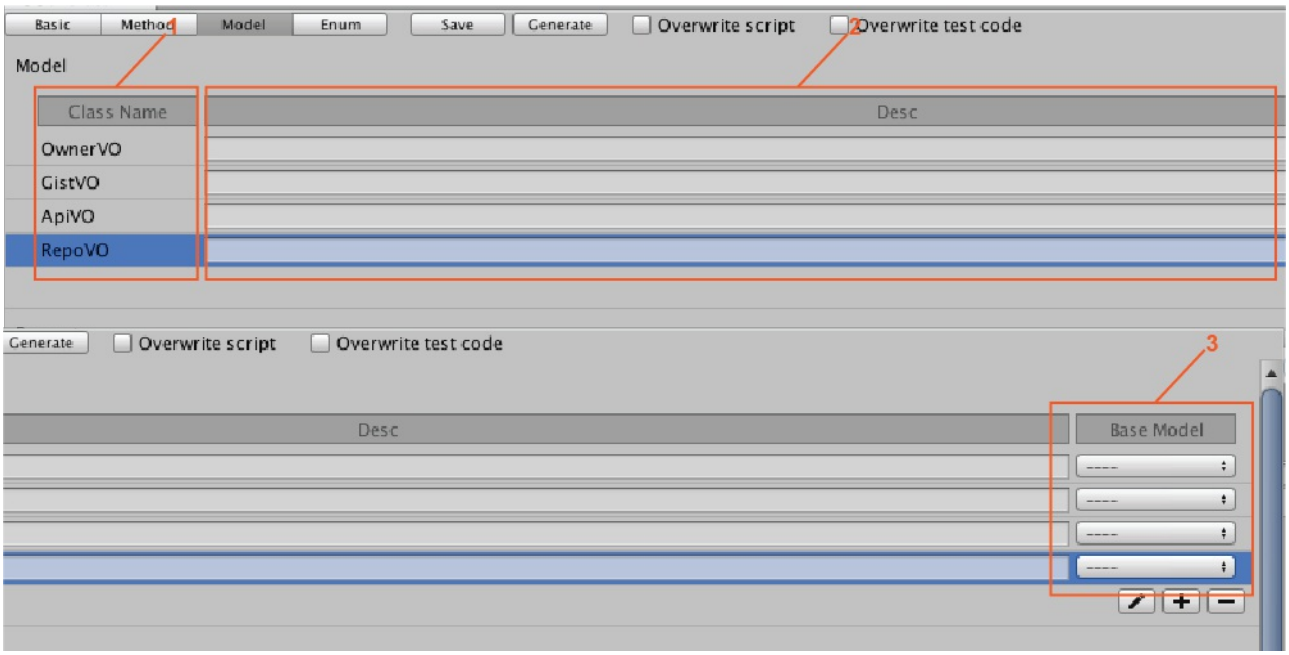
モデルリスト

ここではメソッドのパラメーター、メソッドの戻り値などに使われるモデルを定義します。例に挙げたモデル名の後ろについている“VO”は、“Value Object”の略語で、ロジックを持たない値を格納するためのオブジェクトであることを意味します。モデルは、他のモデルをプロパティに持つことができます。



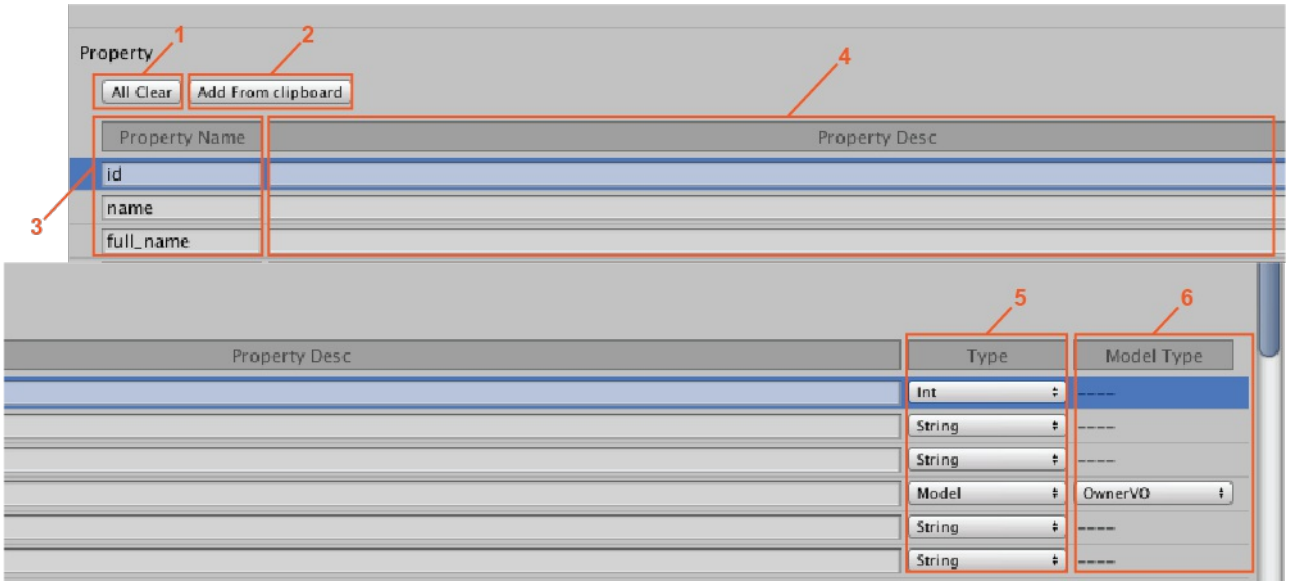
1. モデルの一覧
2. モデルのプロパティ

モデルの一覧



1. Class Name: モデルのクラス名を指定します。
2. Desc: モデルの説明を記述すると、生成されたスクリプトにコメントとして記載されます。
3. Base Model: 共通のプロパティを持つモデルがある場合、共通クラスを作り、継承することができます。指定しない場合は ---- を選びます。

モデルのプロパティ



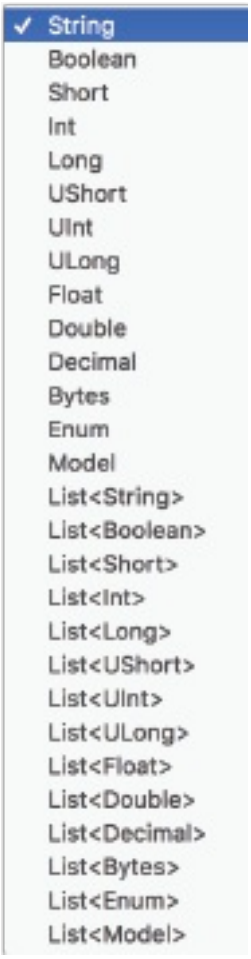
1. All Clear: 選ばれているモデルのすべてのプロパティを削除します。

2. Add From clipbboard: クリップボードの中のJSONの文字列からプロパティを作成します。
   すでにサーバーからの戻り値のJSONが存在する場合に便利です。例えば {"key1":"value1", "key2":"value2"} というJSON文字列をクリップボードにコピーした状態でこのボタンを押すと、"key1"と"key2"が追加されます。
   "key1":"value1", "key2":"value2" のように、"{" と "}" で囲まれていなくても大丈夫です。

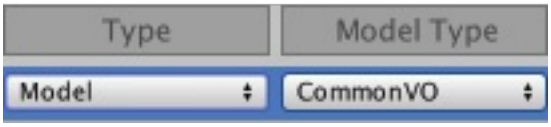
3. Property Name: プロパティ名を指定します。実際にサーバーと通信する際のJSONのキー名と同じにします。
   ※ プロパティ名は、生成されるスクリプトではC#のクラスのメンバ変数になります。なので予約語はプロパティ名として使うことができません。
   (例: public, protected, private)

4. Property Desc: プロパティの説明を記述すると、生成されたスクリプトにコメントとして記載されます。

5. Type: パラメーターの型を指定します。



6. Model Type: 型が `Model`, `Enum`, `List<Model>`, `List<Enum>` のいずれかの時、どれを使うのかを選びます。
   `Model`, `List<Model>`の場合はModelの中から選びます。

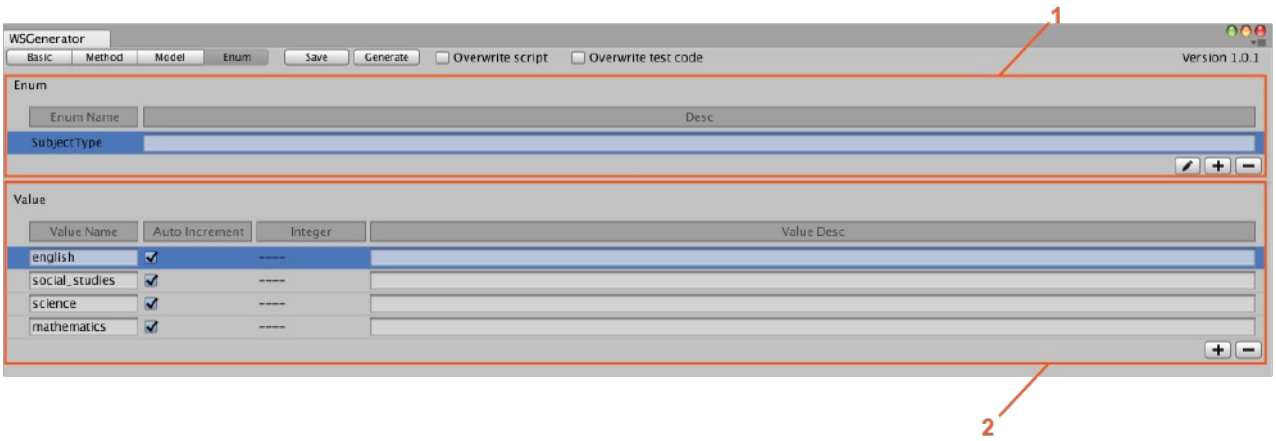


   `Enum`, `List<Enum>`の場合はEnumの中から選びます。

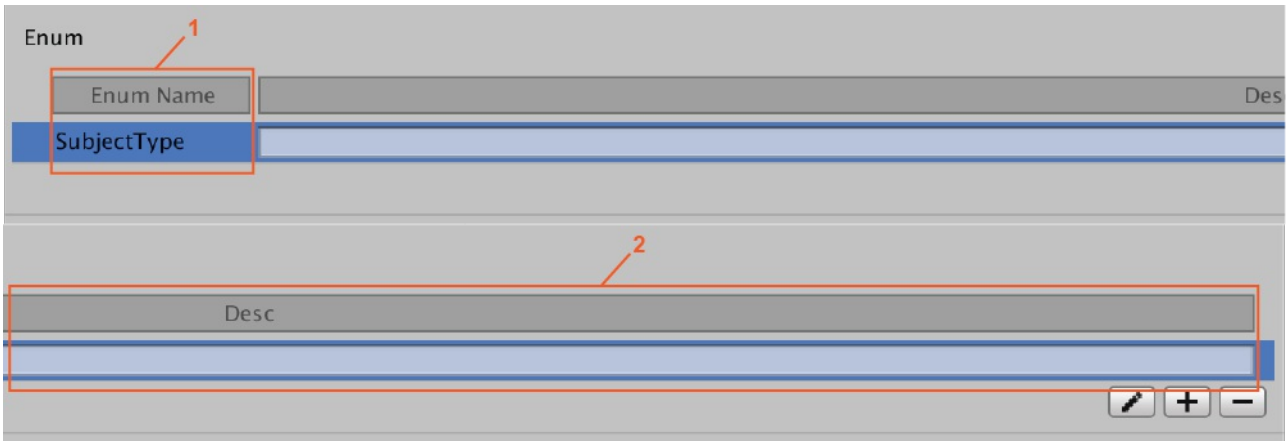
※ プロパティの型は、Listは指定できますが、Dictionaryは指定できません。WWWScriptGeneratorではJsonUtilityを使ってJSONからモデルへのパースを行なっていますが、Dictionary型はJsonUtilityではサポートされていないからです。もしDictionary型のプロパティにしたい場合は、いったんList型としてパースした後、手動でDictionaryに変換すると良いでしょう。

enumリスト

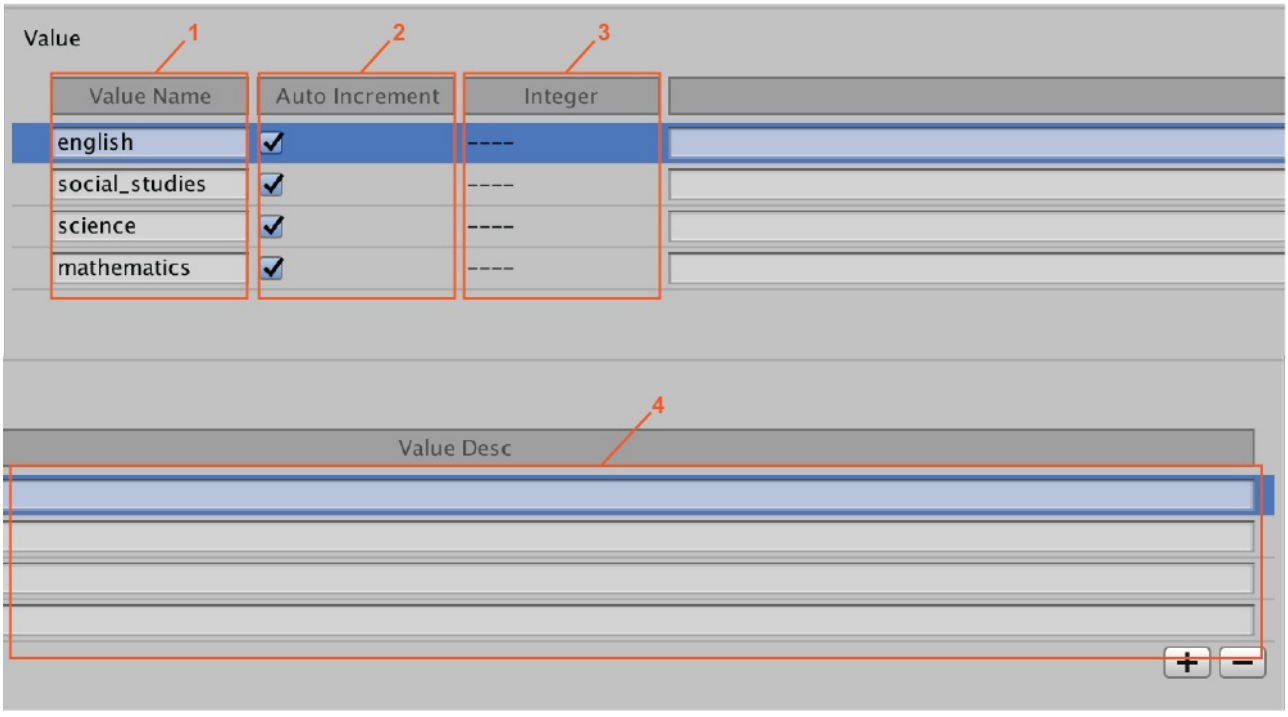
ここでは、メソッドの引数やモデルのパラメーターに使うenumを定義します。例えば科目というパラメーターがあり、0が英語、1が社会を意味する場合、直接数字を扱うよりenumにしたほうが可読性があがります。
整数は通常0からはじまり1ずつ増えていきますが、数字を指定することで英語は1000, 社会は2000 というように固有の数字を指定することができます。



1. enumの一覧
2. enumの値

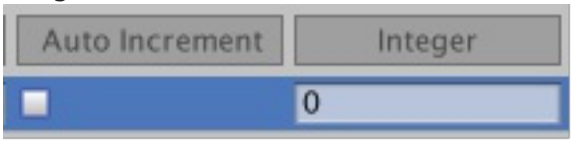


1. Enum Name: enum名を指定します。
2. Desc: enumの説明を記述すると、生成されたスクリプトにコメントとして記載されます。



1. Value Name: enumの値の名前を指定します。書き出されるコードではPascalに変換されるので、ここでは大文字である必要はありません。

2. Auto Increment:　整数の値を指定したい場合はFalseにします。Trueの場合、整数の値は前の値に1を加えた値になります。
3. Integer: 整数の値を指定したい場合、Auto IncrementをFalseにした上で、値を指定します。



4. Value Desc: enumの値の説明を記述すると、生成されたスクリプトにコメントとして記載されます。

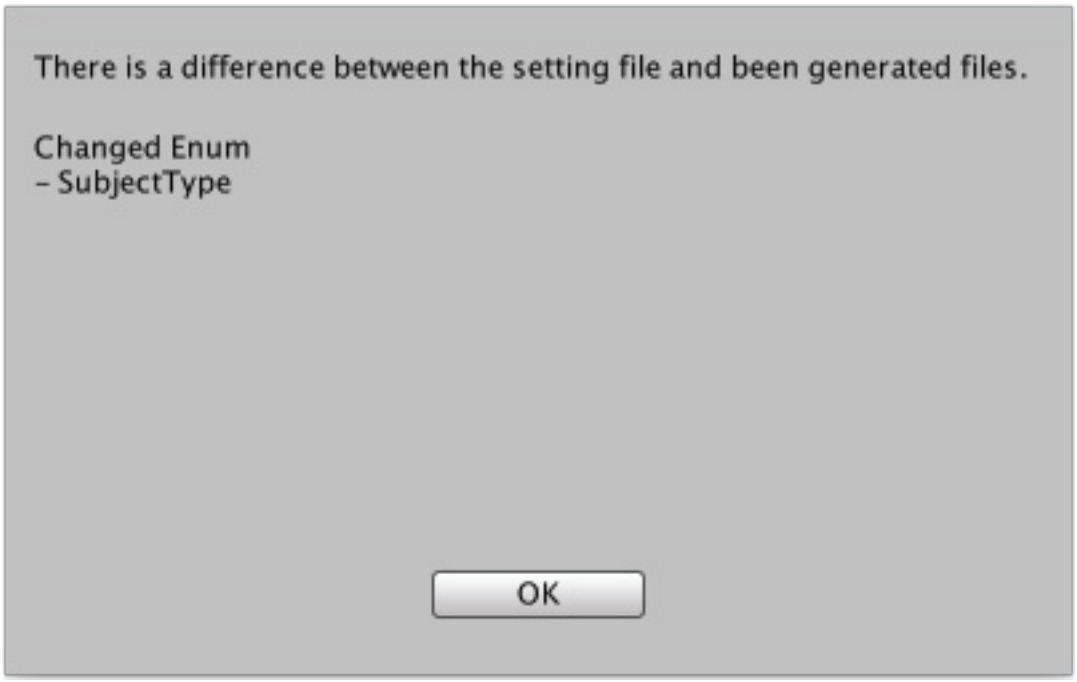
スクリプト／テストコードの生成

Generateボタンを押すと、スクリプトとテストコードが書き出されます。
ディレクトリ “method” には、メソッドリストで定義したメソッド、"vo" には、モデルリストで定義したモデル、"en" には enumリストで定義したenumのスクリプトがそれれぞれ格納されます。
ディレクトリ “vo” の中の VoidVO は、メソッドの戻り値で Void を指定した場合のためのモデルです。
ディレクトリ “core”の中は、サーバー通信を行うためのクラスと、エラー情報のためのクラスです。

※　書き出されるスクリプトは C# です。それ以外の言語には対応していません。

一度スクリプトが書き出されたあと、設定が変更されるたび、またはgitなどで設定ファイルが書き換えらるたびに、前回の設定から変更があったかどうかをチェックし、もし変更があったらそのことをお知らせします。

show detail を押すと、どこが変更になったのかを確認することができます。



スクリプトの書き出し

基本設定の Save Path で指定したディレクトリの下にサブディレクトリが作られ、スクリプトはその下に保存されます。
APITaskBaseがサーバーとの通信処理を行う最も重要なクラスです。このクラスの中でWWWクラスをつかって通信処理を行っています。APITask は APITaskBaseを継承したクラスです。ユーザーはこのクラスでメソッドをオーバーライドすることで、通信処理をカスタマイズすることができます。

また、APIErrorをカスタマイズして、エラーコードをenumで定義することができます。

```
[core]
    APIError.cs // APIErrorBaseを継承したクラス
    APITask.cs // APITaskBaseを継承したクラス
    [basic]
        APIErrorBase.cs // 通信エラーのエラーコードやメッセージ
        APITaskBase.cs // サーバーとの通信処理をおこなうクラスです。
[en]
    <enum名>.cs
[method]
    <メソッド名>.cs
[vo]
    <モデル名>.cs
```

APIError, APITask以外のスクリプトは、書き出しのたびに上書きされます。スクリプトの一番上に

```
// [Caution] This file was made by WWW Script Generator automatically. Don't
modify it.
```

と書かれているように、これらは編集しないでください。もし編集しても、次の書き出しのときに上書きされてしまいます。
ですが、APIError, APITaskはユーザーが編集することができるスクリプトなので、書き出ししても上書きされません。

一度書き出したあと、基本設定で名前空間を変更すると、APIErrorやAPITaskの名前空間と他のスクリプトとの不一致でエラーが起こります。その場合、手動で修正するか、
共通メニューの　Overwrite script にチェックをいれて書き出しを実行し、強制的に上書きします。その際、カスタマイズしたコードは上書きされてしまうのでご注意ください。

APIErrorのカスタマイズ

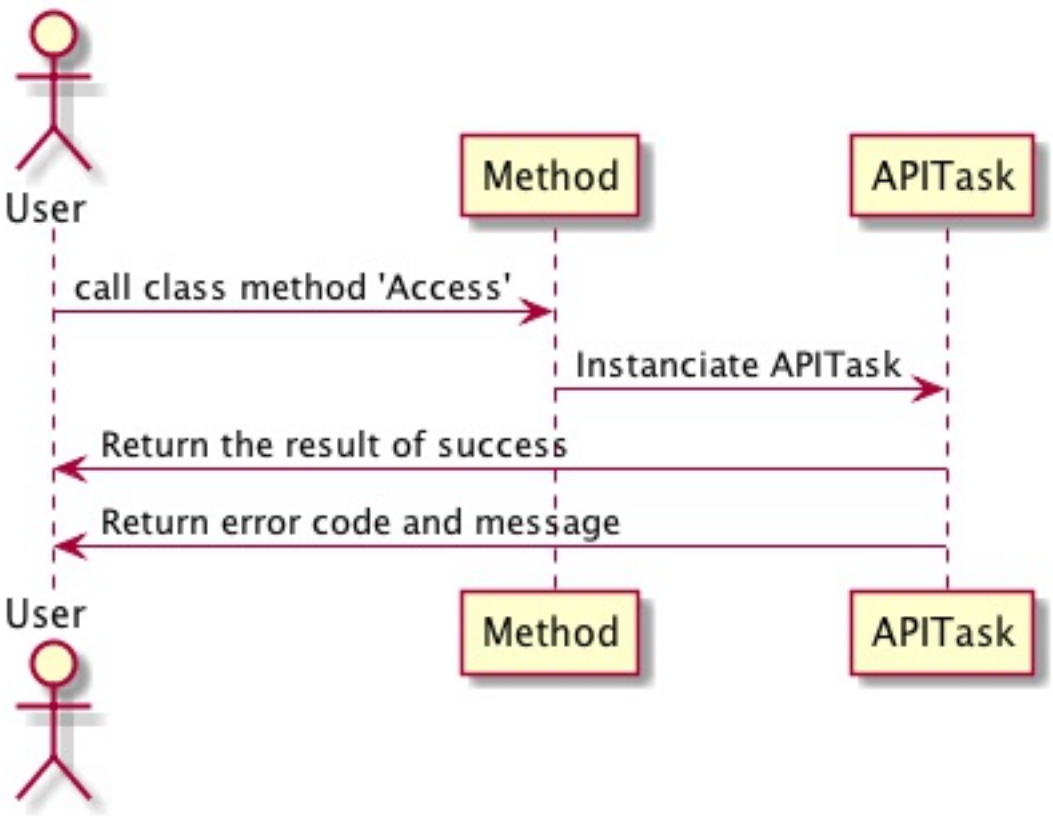
このクラスはエラー状態を表すクラスなので、ロジックを書くことはないと思います。プロパティのCodeはint型ですが、APIErrorの中にエラーコードをenumとして定義し、intにキャストして入れると便利です。

```
/// Error Data for API method
public class APIError : APIErrorBase
{
    // In most cases, when server error occurred, the server returns an error
code.
    // We define that code as an enum or static constant.
    // This is an example and you can modify it.
    //
    // public enum ErrorCodeEnum
    // {
    //     AuthorizationError = 1000,
    //     UnknownMember = 1001,
    // }
    //
    // public ErrorCodeEnum ErrorCode { get { return (ErrorCodeEnum)Code; } set
{ Code = (int)value; } }

    public APIError(int code = 0, string message = null) : base(code, message)
    {
    }
}
```

APITaskのカスタマイズ

このクラスはWWWクラスを使って実際にサーバー通信をおこなうクラスです。ユーザーはそれぞれのAPIメソッドクラスに用意されているクラスメソッドの “Access” を呼び出すことで、APITaskが通信を行い、結果をユーザーに返します。APITaskのインスタンスは一つの通信につき一つ作られ、通信が終わると破棄されます。APITaskはAPITaskBaseを継承しており、主な通信ロジックはAPITaskBaseに書いてあります。
ユーザーはメソッドをオーバーライドし、APITaskを編集することで処理をカスタマイズできます。



- 基底クラスに定義されているenum
    - APIDomainType: ドメイン名がenumとして定義されています

```
public enum APIDomainType
{
    Basic,
}
```

    - APIParamType: 共通パラメーター名がenumとして定義されています。

```
public enum APIParamType
{
    None,
    CommonParam,
}
```

    - APIMethodType: メソッドのPOST/GETが定義されています。

```
public enum APIMethodType
{
    POST,
    GET,
}
```

- 基底クラスのメンバ変数
  このメソッドのメンバ変数、Url、DomainType、ParamType、MethodType、IsList、tmpError、paramTable、paramInURLTable、paramInURLOrderList　は、カスタマイズする処理の条件分岐に使うことができます。
- 基底クラスのクラス変数

    - HandleCookie: Cookieを取り扱うかどうかです。基本設定での値がデフォルトで入っています。
    - KeyForCookie: PlayerPrefsにCookieを保存するためのキー名です。基本設定での値がデフォルトで入っています。

- 基底クラスのクラスメソッド

    - ClearCookie: 保持してあるCookieをPlayer Prefsから消去します。ユーザーがログアウトした時など、必要なタイミングで、APITask.ClearCookie(); や APITaskBase.ClearCookie(); のように呼びます。

      public static void ClearCookie()
      {
      if (!string.IsNullOrEmpty(KeyForCookie) && PlayerPrefs.HasKey(KeyForCookie))
      {
      PlayerPrefs.DeleteKey(KeyForCookie);
      }

      }

- APITask のクラスメソッド

    - OverwriteSetting: Cookieに関する設定を上書きします。デフォルトでは基本設定での値が入っているのでこのメソッドを呼ぶ必要はありませんが、もし動的に変更したいときにはこのメソッドを使います。

      public static void OverwriteSetting()
      {
      // You can overwrite settings such as coockies.
      // e.g.
      // HandleCookie = true;
      // KeyForCookie = "wsg_cookie";
      }

- TimeoutSec: タイムアウトの時間を変更します。30秒にしたい場合は次のように書きます。

```
override protected float TimeoutSec
{
    get { return 30; }
}
```

- GetCommonParamValue: 共通パラメーターのキーに対する値を決めます。
  共通パラメーターで使う値は、PlayerPrefsやファイルなどに保存されていることを想定しています。それらの値を使い、キーごとに適切な値を返します。

```
  override protected System.Object GetCommonParamValue(string key)
  {
      // If common parameter exists, you have to return value for them.
      // Values are assumed to be stored in PlayerPrefs such as member
ID.
      // e.g.
      // switch (key)
      // {
      //     case "member_id": return PlayerPrefs.GetString("member_id",
"");
      //     default: return "";
      // }
      return "";
  }
```

- AddCommonParam: 共通パラメーターの追加処理
  ここでは、WWWFormに追加するまえに、一時的にテーブルに保持しています。ここの処理を変えることはあまりないと思います。

```
  override public void AddCommonParam()
  {
      // You can customize how to handle common parameters
      base.AddCommonParam();
  }
```

- AddParam: 通常のパラメーターの追加処理
  ここでは、WWWFormに追加するまえに、一時的にテーブルに保持しています。ここの処理を変えることはあまりないと思います。

```
  override public void AddParam(string key, System.Object value)
  {
      // You can customize how parameters are added.
      base.AddParam(key, value);
  }
```

- GetBinaryDataFileName: 追加するバイナリデータのファイル名を必要に応じて指定します。

```
  override protected string GetBinaryDataFileName(string key)
  {
      // You can specify a filename for binary data.
      return base.GetBinaryDataFileName(key);
  }
```

- GetBinaryDataMimeType: 追加するバイナリデータのMime Typeを必要に応じて指定します。

```
  override protected string GetBinaryDataMimeType(string key)
  {
      // You can specify a mime type for binary data.
      return base.GetBinaryDataMimeType(key);
  }
```

- WWWForm: 一時的にテーブルに保持してあったデータをWWWFormに追加します。
  通常は、WWWFormに直接、AddField または AddBinaryData メソッドに追加しますが、場合によってはすべてのデータを一つのJSONにして入れたり、暗号化して入れたりなどの処理が必要な場合、ここで変更することができます。

```
override protected WWWForm GetForm()
{
    // You can customize how to generate WWwForm instance.
    return base.GetForm();
}
```

- GetDomainURL: ドメインタイプに対するURIを返します。
  サーバーが開発環境と本番環境とでドメインが分かれている場合などは、ここで条件分岐することができます。
  基底クラスは、ドメインリストで定義したすべてのパターンを返します。このメソッドのドメインタイプとプリプロセッサの値などを条件に、適切なURIを返すように変更することができます。

```
override  protected string GetDomainURL(APIDomainType domainType)
{
    // You can change the domain of url depending on conditions.
    return base.GetDomainURL(domainType);
}
```

- GetProperURL: ドメインのURIとエンドポイントを組み合わせて、最終的なURLを組み立てます。
  メソッドタイプがGETの場合は、パラメーターがURLに含まれます。ここの処理を変えることはあまりないと思います。

```
override protected string GetProperURL()
{
    // You can change the URL depending on conditions.
    return base.GetProperURL();
}
```

- CreateHeader: ヘッダーが必要な場合はここで作成できます。
  Cookieを扱う場合は、基底クラスでヘッダー用のDictionaryインスタンスが作られます。その際はここでインスタンスを作らずに基底クラスで作られたインスタンスを使ってください。

```
override protected Dictionary<string, string> CreateHeader()
{
    // Header information can be added to form depending on conditions.
    return base.CreateHeader();
}
```

- CreateWWW: WWWインスタンスを作成します。ここの処理を変えることはあまりないと思います。

```
override protected WWW CreateWWW(string properUrl, WWWForm form,
Dictionary<string, string> headerData)
{
    // You can customize the WWW instance depending on the condition.
    return base.CreateWWW(properUrl, form, headerData);
}
```

- CheckError: サーバーからのレスポンスの内容から、それがエラーかどうかを判定します。
  タイムアウトになったり、www.errorが返ってきた場合は、このメソッドに到達する前にエラー処理されますが、www.isDoneのあと、サーバーからの戻り値をみて、エラーなのかどうかを判定します。
  JSONの中のcodeというキーの値がエラーコードを示す場合や、successというキーの値がfalseの場合がエラーなど、サーバーの実装により変わりますが、エラーの場合はエラーコードとメッセージが返ってくることが多いと思います。エラーが起きた場合は APIErrorのインスタンスを作り、Code と Message のプロパティをセットして、そのインスタンスを返します。成功した場合は null を返します。

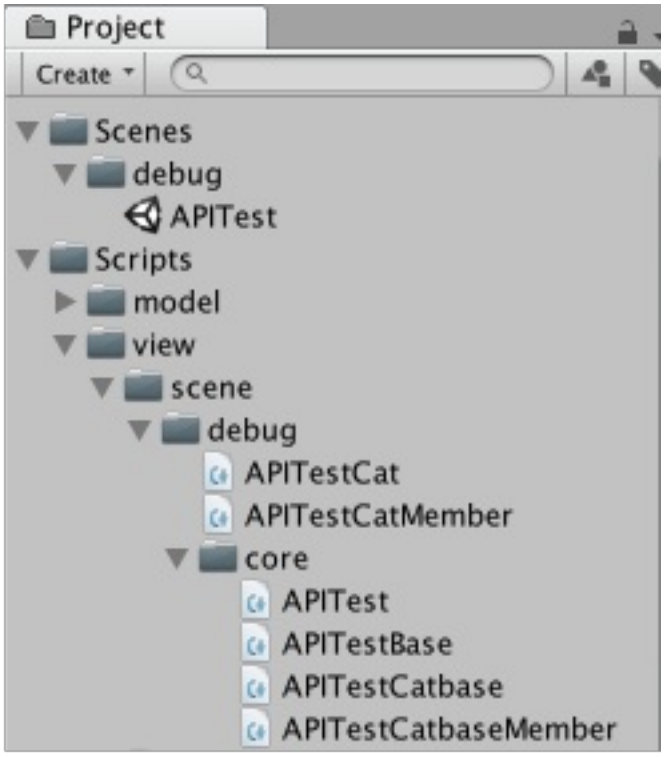
```
  override protected APIError CheckError(WWW www)
  {
      // You can change error codes and messages depending on the server
side specifications.
      // In this example, assuming that "result" and "message" will be
returned from the server.
      // If the value of "result" is 0, it means succeed, otherwise
failure.
      //
      // var jsonStr = Regex.Unescape(www.text);
      // var vo = JsonUtility.FromJson<CommonVO>(jsonStr);
      // if (vo.Result == 0) return null;
      // var err = new APIError(vo.Result, vo.Message);
      // err.ErrorType = APIErrorType.Other;
      // return err;
      return base.CheckError(www);
  }
```

- AdjustReturnValue: 通信が成功した場合の戻り値をここで調整できます。ここの処理を変えることはあまりないと思います。
  すでに retValはwww.textを適切な戻り値のクラスにパース済みですが、戻り値がバイナリだったり、付加情報がある場合などはここで調整できます。

```
  override protected void AdjustReturnValue(System.Object retVal, WWW
www)
  {
      // retVal is VO class which is the return value of method parsed
with www.text.
      // You can adjust the return value by using www before it passed to
the success handler.
  }
```

テストコードの書き出し

シーンファイルは、基本設定の Test code scene path で指定したディレクトリの下に、Test code name で指定した名前で保存されます。
スクリプトは、基本設定の Test code script Path で指定したディレクトリの下に保存されます。



メソッドのカテゴリーとはEndpointが/で区切られている場合の左の文字列です。

```
    APITestCat.cs // APITestCatbaseを継承したクラス
    APITestCat<メソッドのカテゴリー>.cs // APITestCatbase<メソッドのカテゴリー>を継承したクラス
    [core]
        APITest.cs // シーンファイルにアタッチされるメインスクリプト
        APITestBase.cs // 共通処理が書かれた APITestCatbase.cs や APITestCatbase<メソッドのカテゴリー>.cs の基底クラス
        APITestCatbase.cs // Endpointが/で区切られていないメソッドのテストコード
        APITestCatbase<メソッドのカテゴリー>.cs // Endpointが/で区切られているメソッドのテストコード
```

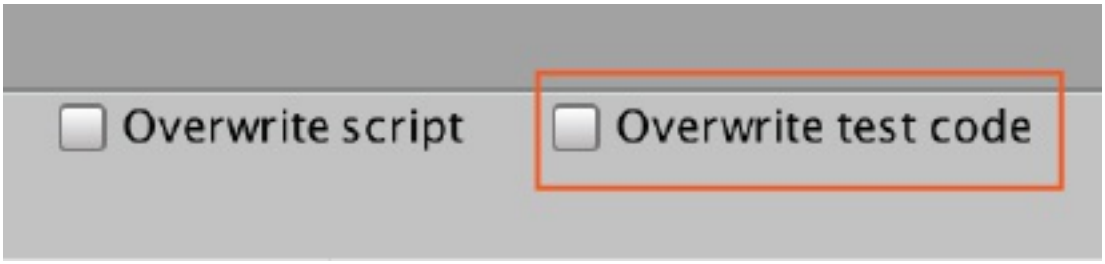
APITestCat.cs, APITestCat<メソッドのカテゴリー>.cs 以外のスクリプトは、書き出しのたびに上書きされます。スクリプトの一番上に

```
    // [Caution] This file was made by WWW Script Generator automatically.
Don't modify it.
```

と書かれているように、これらは編集しないでください。もし編集しても、次の書き出しのときに上書きされてしまいます。
ですが、APITestCat.cs, APITestCat<メソッドのカテゴリー>.cs はユーザーが編集することができるスクリプトなので、書き出ししても上書きされません。

一度書き出したあと、基本設定で名前空間を変更すると、APITestCat.cs, APITestCat<メソッドのカテゴリー>.cs の名前空間と他のスクリプトとの不一致でエラーが起こります。その場合、手動で修正するか、共通メニューの　Overwrite test code にチェックをいれて書き出しを実行し、強制的に上書きします。そ

の際、カスタマイズしたコードは上書きされてしまうのでご注意ください。



APITestCat.cs, APITestCat<メソッドのカテゴリー>.cs　のカスタマイズ

一つのサーバーメソッドに対し、Test<サーバーメソッド名>Execute　と Test<サーバーメソッド名>Success の2つのメソッドが作られます。例えば、Scorelist という名前のメソッドの場合は、TestScorelistExecute と TestScorelistSuccess の2つが作られます。

- Test<サーバーメソッド名>Execute: テスト実行時に呼ばれるメソッドです。
  生成されたばかりのコードにはパラメーターの値が指定指定されてませんので、適切な値を指定します。

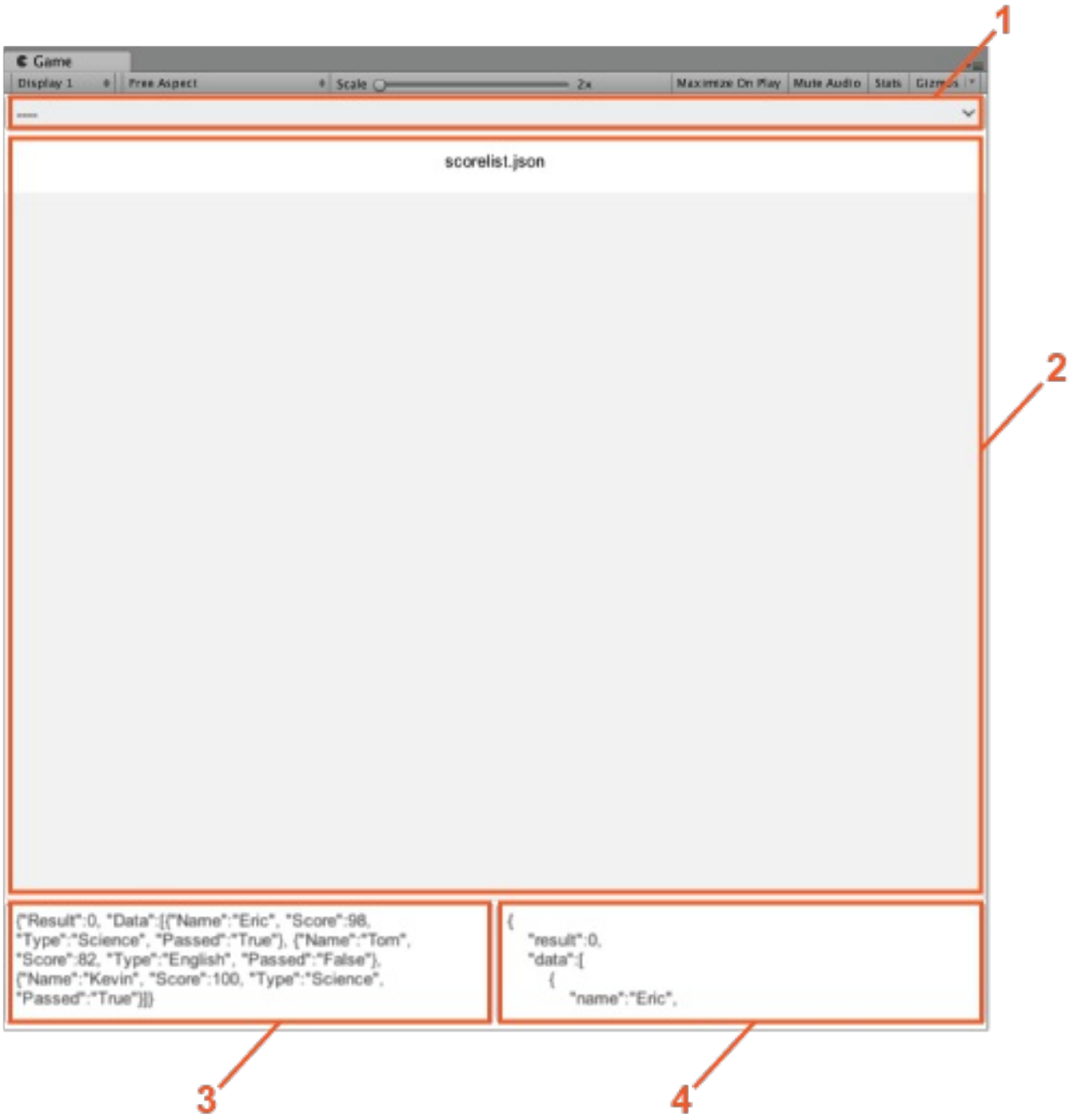
```
 override protected void TestScorelistExecute(string memberId)
 {
     // Set these parameters for method.
     memberId = ""; // Member's ID
     base.TestScorelistExecute(memberId);
 }
```

- Test<サーバーメソッド名>Success: 成功した時に呼ばれるメソッドです。
  戻り値をPlayerPrefsに保存する場合はここで書くことができます。

```
 override protected void TestScorelistSuccess(ScoreListVO vo)
 {
     // If you want to save the property of the return value, you can
write it here.
 }
```

テストコードの実行

エディターで再生することでテストを実行できます。また、ビルドすれば実機でも確認できます。

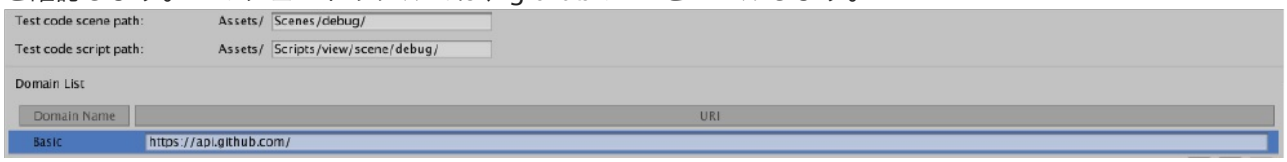


1. メソッドのカテゴリーを選びます。メソッドのカテゴリーとはEndpointが/で区切られている場合の左の文字列です。/で区切られていなければ ----- に分類されます。

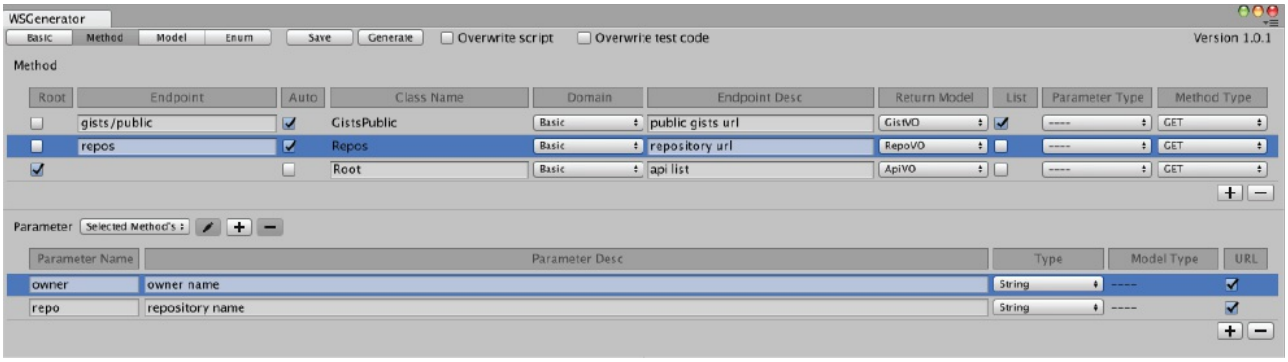

2. メソッドのリストを表示します。項目はそれぞれボタンになっており、押すことでメソッドを実行します。
3. 実行した結果を表示します。戻り値がある場合、そのモデルの内容を表示します。
4. 実際にサーバーから返ってきた文字列をそのまま表示します。これは、期待どおりの結果にならなかった場合の原因調査の手がかりになります。

チュートリアルデータを試してみる

実際にチュートリアルデータを使って試してみましょう。
基本設定画面で Create Tutorial Data ボタンを押してチュートリアルデータを作成します。
Domain List に Basic という名前のドメインの URI が、`https://api.github.com/` になっていることを確認します。このチュートリアルでは、githubのAPIをコールします。



Methodタブをおしてメソッド画面に移動すると 3つのメソッドが作られています。

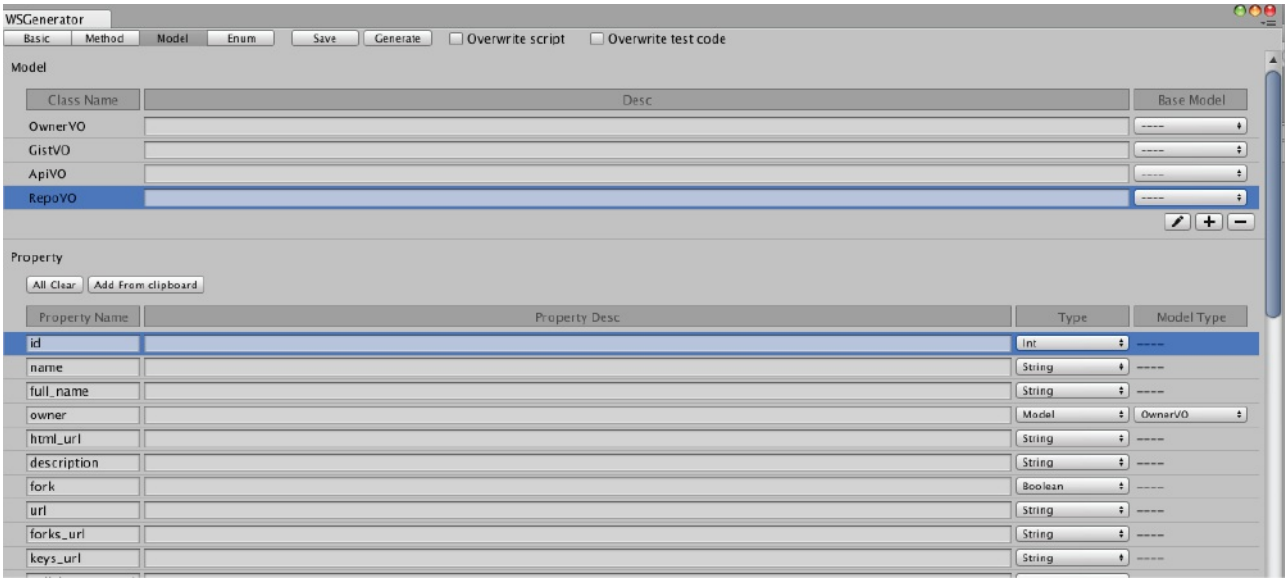


Endpointが　gists/public　の行を見てみましょう。Class Name が GistPublic になっています。これは Auto にチェックが入っているため、Endpointから自動的に作られた名前です。
Return Modelの GistVO は、このメソッドの戻り値のJSONをパースするモデルが GistVO であることを表します。
Listにチェックが入っていますが、これは戻り値のJSONが[で始まる、つまり直接配列が返ってくることを表します。

次に、Endpointが repos の行の左端をクリックしてみてください。行が選ばれ青色になり、下段に owner と repo の 2つのパラメーターが現れます。これは、reposのメソッドを呼ぶには、パラメーターが2つ必要であることを表します。そして、それぞれのパラメーターの URL にチェックがついています。これは、パラメーターがURLに/区切りで直接はいることを表します。

その下のメソッドには Endpointがありません。代わりに Root にチェックが入っています。これは、ドメインの `https://api.github.com/` をそのまま呼び出すことを表します。

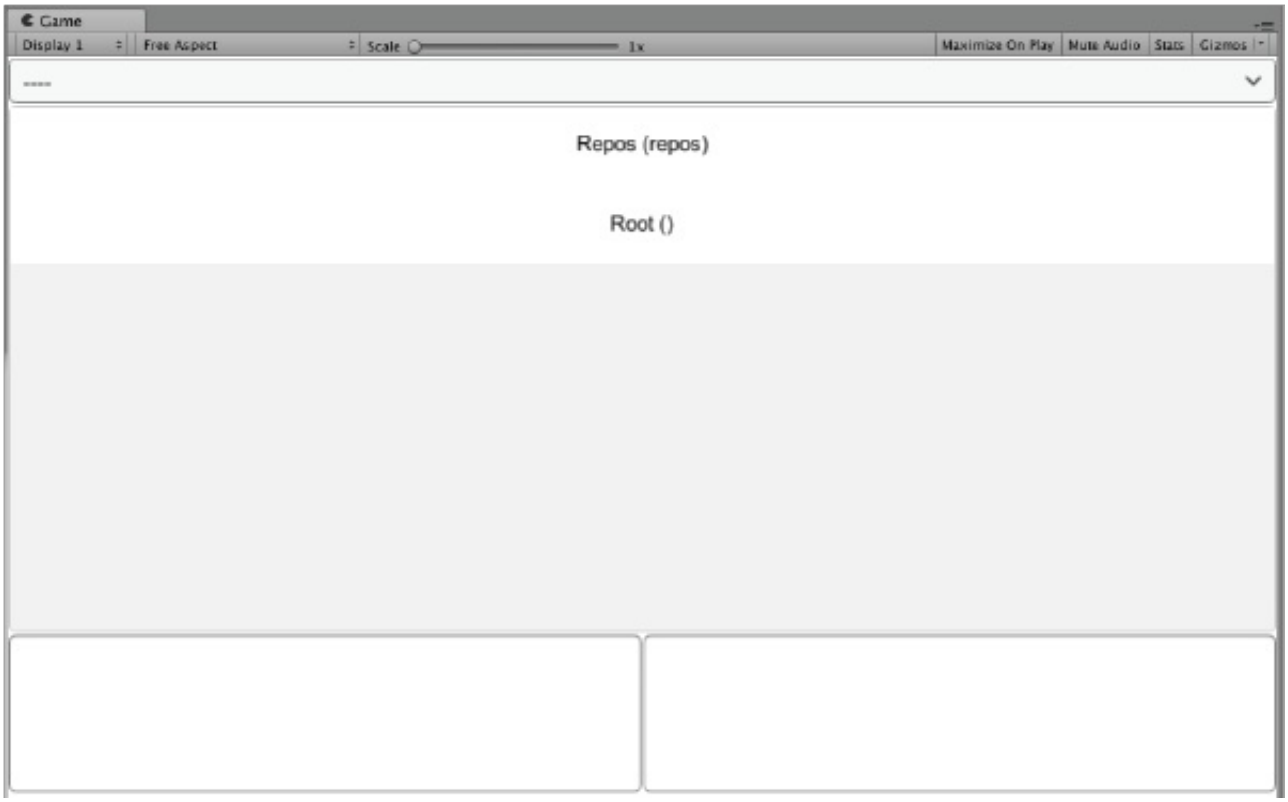
Modelタブをおしてモデルリスト画面に移動すると、4つのモデルが作成されています。



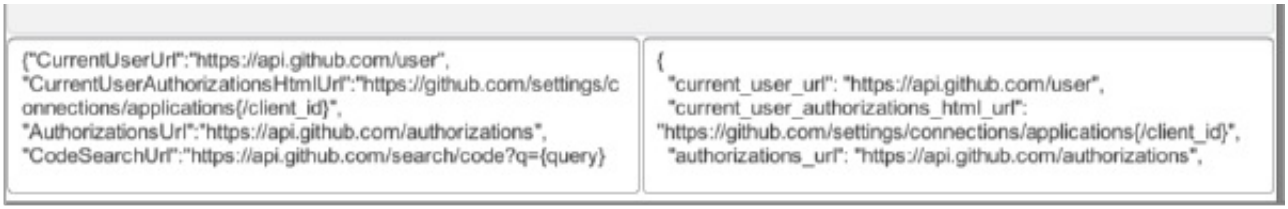
GistVOのプロパティのうち、owner を見てみると、Typeが Model で　Model Type が OwnerVO になっています。これは、GistVO から OwnerVO が参照されていることを表します。また、RepoVO も同様に、owner のプロパティの型が OwnerVO になっており、OwnerVOを参照していることを意味します。

基本設定画面で Generate Test Code がチェックされていることを確認し、共通メニューの Generate ボタンを押します。するとスクリプトとテストコードが書き出されます。Assets/Scenes/debugにあるシーンファイル APITest を開き、Unityエディタのプレイボタンで再生します。
画面には

- Repos (repos)
- Root ()
  が表示されています。



Root をクリックし、しばらくすると下部に結果が表示されます。



Repos は今のままでは押しても失敗します。それはパラメーターが設定されていないからです。パラメーターを設定するにはテストコードをカスタマイズするのですが、テストコードは上記のカテゴリーごとにまとめて書かれています。この２つのメソッドのエンドポイントは/で区切られていないので、カテゴリー名はありません。したがってテストコード名には接尾語がつかないので、APITestCat.cs が対象のコードです。この中の

```
override protected void TestReposExecute(string owner, string repo)
{
    // Set these parameters for method.
    owner = ""; // owner name
    repo = ""; // repository name
    base.TestReposExecute(owner, repo);
}
```

の、owner と repo が空白になっているので、github のオーナー名とリポジトリ名を指定します。（例: owner = "inosyan" repo = "ScrachBand2"）
そして実行すると、結果が返ってくるようになります。

上部のプルダウンメニューで、カテゴリーを ---- から gist に切り替えると、GistsPublic (gists/public) が表示されます。

実行すると、String too long for TextMeshGenerator. Cutting off characters.　のエラーがでますが、これは表示する文字が多いことによるuGUIの問題なので、気にしないでください。戻り値は表示されています。
APITestCatGists.cs を開くと、

```
override protected void TestGistsPublicSuccess(GistVOList vo)
```

の引数の型が GistVOList になっています。これはモデルでは定義していないクラスですが、メソッド画面で List にチェックを入れていて、このメソッドの戻り値が GistVO の配列なので、その戻り値を格納するために自動的に作られたクラスです。プロパティの List に、JSONのパース結果が格納されます。

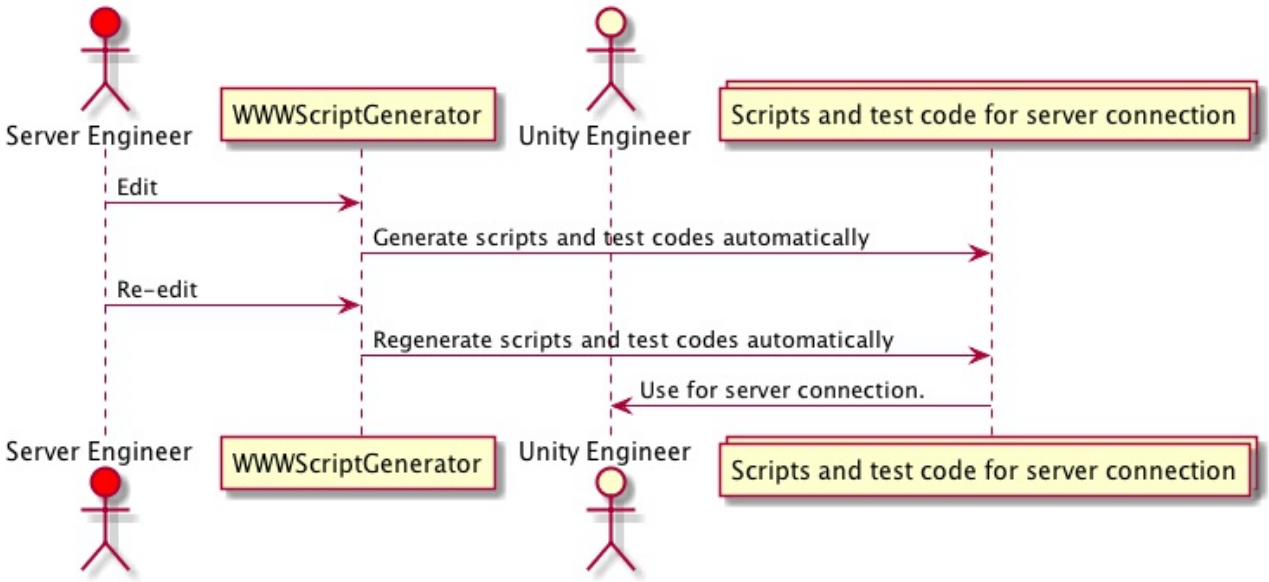
基底クラスの APITestCatbaseGists.cs を見てみましょう。TestGistsPublicExecute() にあるようなコードでメソッドを実行しています。実際にメソッドを使用する際の参考にしてください。voの型はメソッド画面で設定したReturn Modelになります。

```
ApiGistsPublic.Access((vo, rawText) =>
    {
        // 成功処理
    },
    (err, rawText) =>
    {
        // エラー処理
    }, gameObject);
```

サーバーエンジニアとの設定の共有方法

設定ファイルの共有

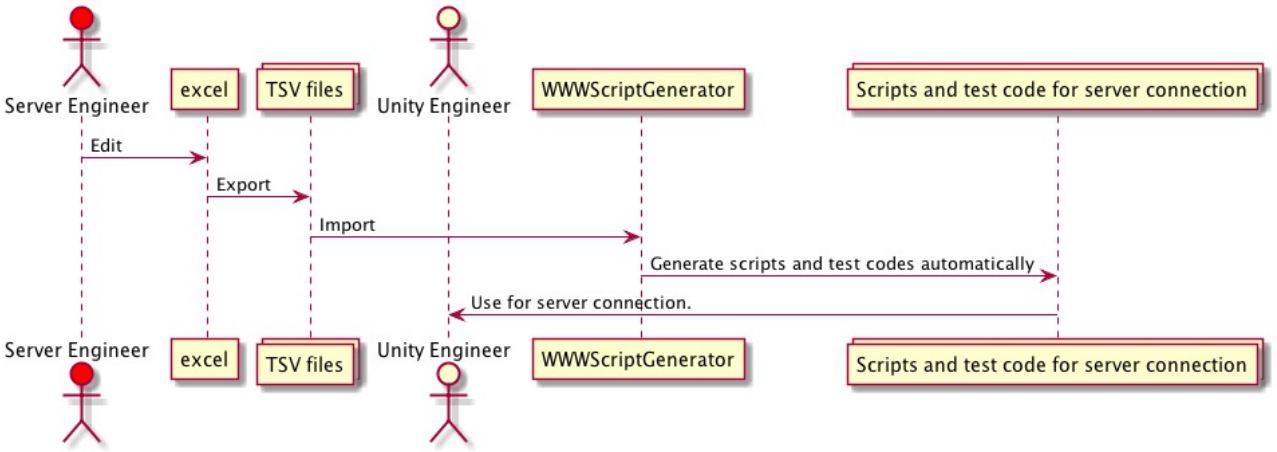
設定ファイルを共有することで、相互に設定を編集することが可能です。
Assets/WWWScriptGenerator/UserData/ の中にある　wwwscriptgenerator_setting.json が設定ファイルです。このファイルをgitなどで共有し、Unityで編集します。



エクセルファイルの共有（Pro版限定）

エクセルファイルを通じて、相互に設定を編集することが可能です。
Assets/WWWScriptGenerator/ の中にある　wsg_template.xlsm がエクセルファイルです。このファイルはオリジナルなので編集せずにとっておき、そのコピーをwsg.xlsmなどの名前で UserData の中に入れて使用します。そのファイルをgitなどで共有し、エクセルからTSVを書き出し、Unityに読み込みます。

このエクセルファイルにはマクロで入力バリデーションをつけてあるので入力ミスによる不具合は未然に防ぐことができます。

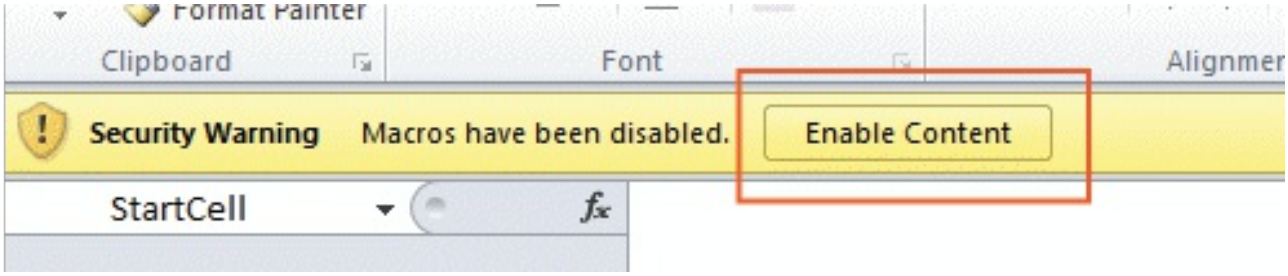


TSVファイルの共有（Pro版限定）

TSVファイルを直接共有することでも、相互に設定を編集することが可能です。ただし、エクセルの場合のように入力バリデーションがないので、選択肢にない値を入れてしまわないように、入力ミスに気をつける必要があります。

エクセルファイルについて

エクセルにはTSVのインポート／エクスポート、および入力バリデーションのため、マクロが含まれています。
なので、エクセルを開いた際に警告が出ますが、マクロを有効にしてください。

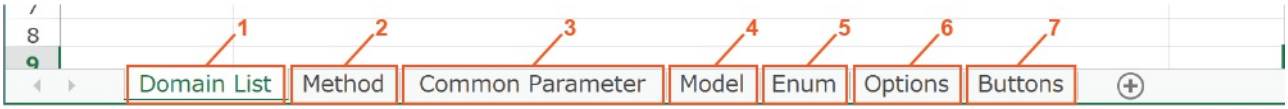


サーバーエンジニアとUnityエンジニアがエクセルを共有することを想定しています。

1. サーバーエンジニアがエクセルを編集し、Unityエンジニアがそれを受け取りUnityにインポートする場合
2. プロトタイプのプロジェクトがあり、Unityエンジニアが設定をUnityからエクスポートし、エクセルをサーバーエンジニアに渡す場合
   いずれの場合でも、エクセルとUnityとで相互に編集することが可能です。

シート

エクセルではシートに分かれています。Unityの設定のうち、サーバーエンジニアと共有する必要のあるものだけが含まれています。

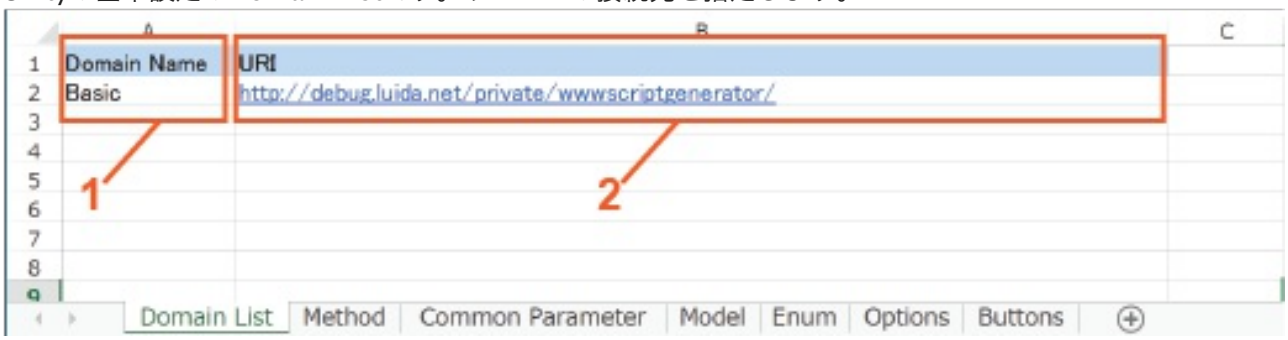


1. Domain List: Unityの基本設定のDomain Listです。サーバーの接続先を指定します。

2. Method: Unityのメソッドリストです。
3. Common Parameter: Unityのメソッドリストの共通パラメーターです。
4. Model: Unityのモデルリストです。
5. Enum: Unityのenumリストです。
6. Options: エクセルの入力バリデーションに使用するシートです。（ユーザーが編集する必要はありません）
7. Buttons: マクロを実行するためのボタンがあるシートです。

ドメインリスト

Unityの基本設定のDomain Listです。サーバーの接続先を指定します。

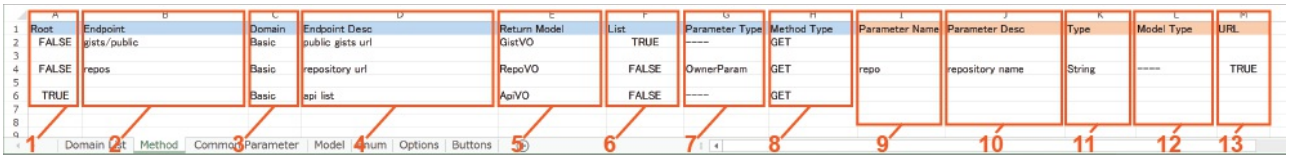


一つのDomain Nameに対し、一つのURIを指定します。項目は行を空けずに書きます。

1. Domain Name: 接続先の名前
2. URI: 接続先のURI

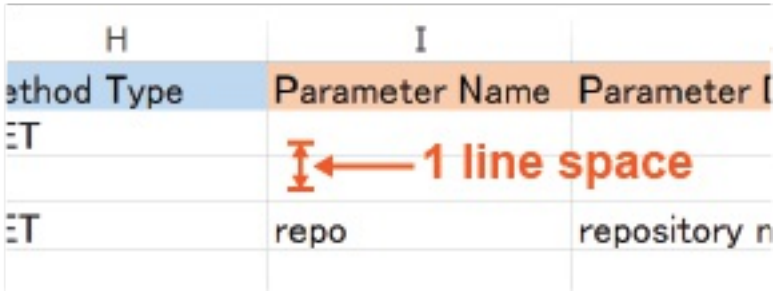
メソッド

Unityのメソッドリストです。
UnityのメソッドリストにあるAuto と Class Name は、Unity側だけで使用する項目なので、エクセルには含まれていません。



一つのEndpointに対し、Domain, Endpoint Desc, Return Model, Parameter Type, Method Type をそれぞれ一つずつ指定します。
パラメーターは一つのEndpointに対し複数指定できます。一つのParameter Nameに対し、Parameter Desc, Type, Model Type をそれぞれ一つずつ指定します。

Parameter Nameと次のメソッドのParameter Name との間は、見やすいように1行あけることができます。詰めて書いても問題ありません。



1. Root: エンドポイントが無く、Domainリストで指定したURIそのままの場合はTRUEにします。

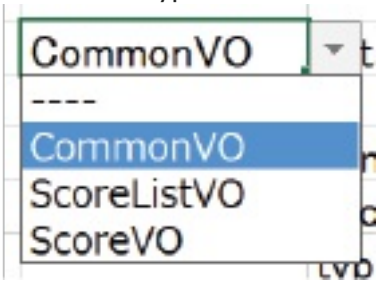
2. Endpoint: メソッドのエンドポイントです。Domainリストで指定したURIに続く文字列を書きます。

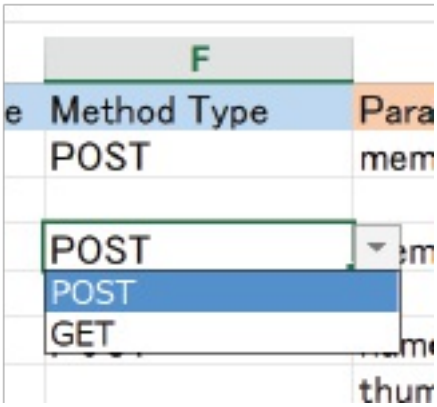
3. Domain: ドメインリストから、どのドメインのURIを使うのかを指定します。

4. Endpoint Desc: このエンドポイントがどのような目的のものかを記述すると、生成されたスクリプトにコメントとして記載されます。

5. Return Model: モデルの中からメソッドの戻り値を指定します。戻り値が無い場合はVoidを選びます。

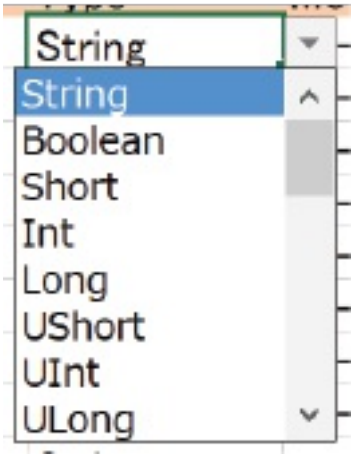
6. List: 戻り値のJSONが配列の場合に指定します。
   例えばJSONが
   `{"list":[{"name":"a"}, {"name":"b"}]}`
   ではなく
   `[{"name":"a"}, {"name":"b"}]`
   のような形で返ってくる場合には、これをTRUEにします。

7. Parameter Type: 共通パラメーターを指定します。指定しない場合は ---- を選びます。
   

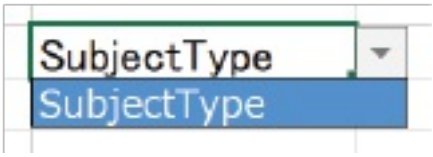
8. Method Type: 通信方法を POST　か GET かのいずれかを選びます。
   

9. Parameter Name: パラメーター名を指定します。実際にサーバーと通信する際のサーバーに送るキー名と同じにします。

10. Parameter Desc: パラメーターの説明を記述すると、生成されたスクリプトにコメントとして記載されます。

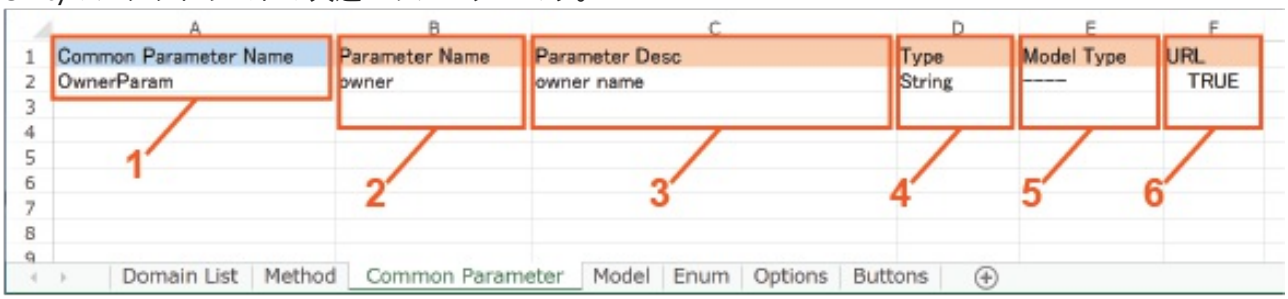
11. Type: パラメーターの型を指定します。


12. Model Type: 型が `Model`, `Enum`, `List<Model>`, `List<Enum>` のいずれかの時、どれを使うのかを選びます。
`Model`, `List<Model>`の場合はModelの中から選びます。


`Enum`, `List<Enum>`の場合はEnumの中から選びます。


13. URL: パラメーターがURLの中に入るのかどうかを指定します
例えば "param1"というパラメーターの値が"value1"だった場合、実際のURLは
`https://example.com/value1`
のようになります。

共通パラメーター

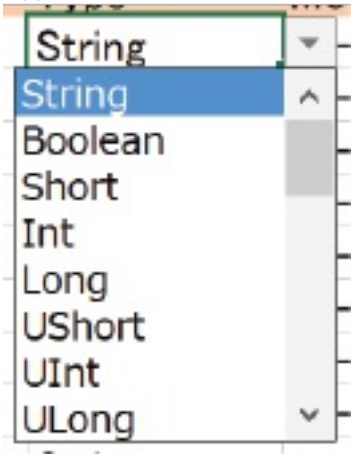
Unityのメソッドリストの共通パラメーターです。


パラメーターは一つのCommon Parameter Nameに対し複数指定できます。一つのParameter Nameに対し、Parameter Desc, Type, Model Type をそれぞれ一つずつ指定します。

Parameter Nameと次の共通パラメーターのParameter Name との間は、見やすいように1行あけることができます。詰めて書いても問題ありません。

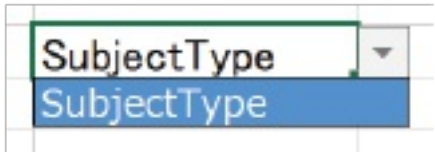
1. Common Parameter Name: 共通パラメーターの名前を指定します。

2. Parameter Name: パラメーター名を指定します。実際にサーバーと通信する際のサーバーに送るキー名と同じにします。
3. Parameter Desc: パラメーターの説明を記述すると、生成されたスクリプトにコメントとして記載されます。
4. Type: パラメーターの型を指定します。



5. Model Type: 型が `Model`, `Enum`, `List<Model>`, `List<Enum>` のいずれかの時、どれを使うのかを選びます。
   `Model`, `List<Model>`の場合はModelの中から選びます。

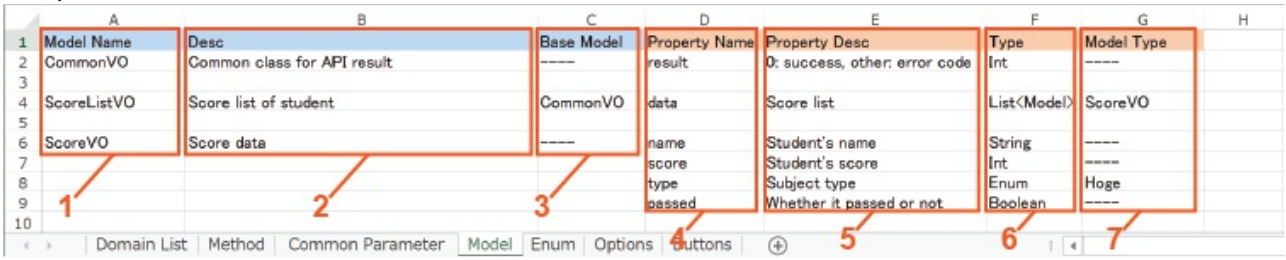
   `Enum`, `List<Enum>`の場合はEnumの中から選びます。



6. パラメーターがURLの中に入るのかどうかを指定します
   例えば "param1"というパラメーターの値が"value1"だった場合、実際のURLは
   `https://example.com/value1`
   のようになります。

モデルリスト

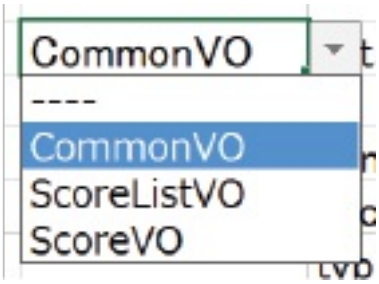
Unityのモデルリストです。



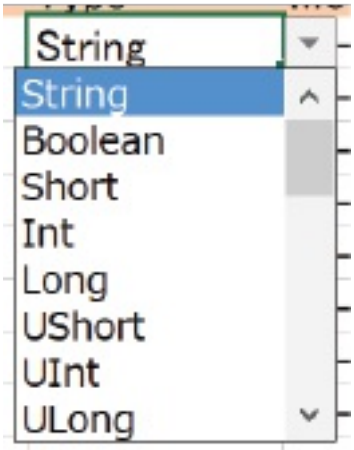
一つのModel Nameに対し、Desc, Base Model をそれぞれ一つずつ指定します。
プロパティは一つのModel Nameに対し複数指定できます。一つのProperty Nameに対し、Property Desc, Type, Model Type をそれぞれ一つずつ指定します。

Property Nameと次のメソッドのProperty Name との間は、見やすいように1行あけることができます。詰めて書いても問題ありません。

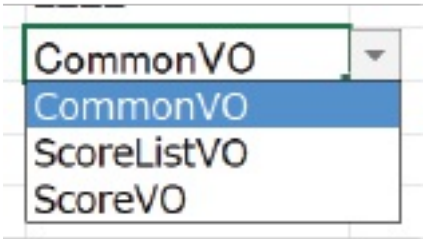
1. Model Name: モデルのクラス名を指定します。
2. Desc: モデルの説明を記述すると、生成されたスクリプトにコメントとして記載されます。
3. Base Model: 共通のプロパティを持つモデルがある場合、共通クラスを作り、継承することができます。指定しない場合は ---- を選びます。



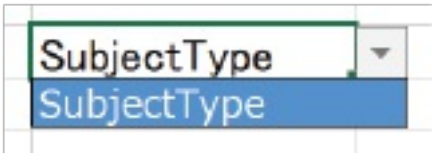
4. Property Name: プロパティ名を指定します。実際にサーバーと通信する際のJSONのキー名と同じにします。
5. Property Desc: プロパティの説明を記述すると、生成されたスクリプトにコメントとして記載されます。
6. Type: パラメーターの型を指定します。



7. Model Type: 型が `Model`, `Enum`, `List<Model>`, `List<Enum>` のいずれかの時、どれを使うのかを選びます。
   `Model`, `List<Model>`の場合はModelの中から選びます。

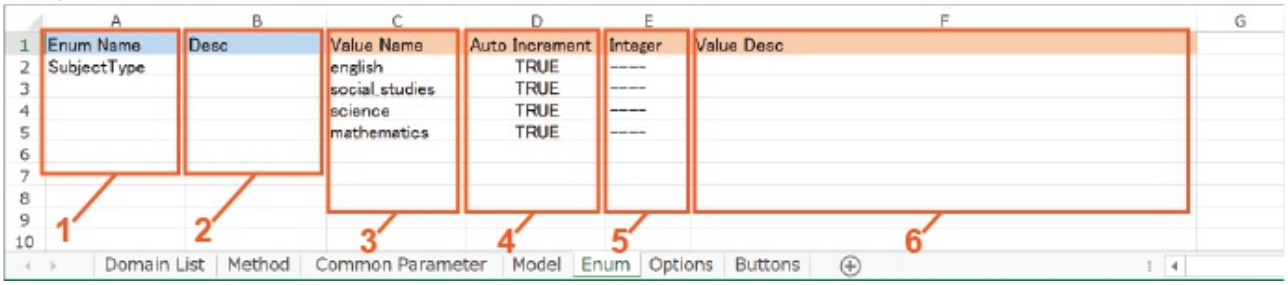


   `Enum`, `List<Enum>`の場合はEnumの中から選びます。



enumリスト

UnityのEnum画面のenumリストです。



	A	B	C	D	E	F	G
1	Enum Name	Desc	Value Name	Auto Increment	Integer	Value Desc	
2	SubjectType		english	TRUE	----		
3			social_studies	TRUE	----		
4			science	TRUE	----		
5			mathematics	TRUE	----		
6							
7							
8							
9							
10							

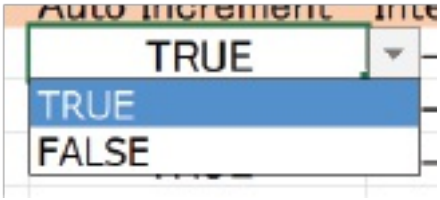
1 2 3 4 5 6

Domain List | Method | Common Parameter | Model | Enum | Options | Buttons

一つのEnum Nameに対し、Desc を一つ指定します。
Enumの値は一つのEnum Nameに対し複数指定できます。一つのValue Nameに対し、Auto Increment Integer, Value Desc をそれぞれ一つずつ指定します。

Value Nameと次のメソッドのValue Name との間は、見やすいように1行あけることができます。詰めて書いても問題ありません。

1. Enum Name: enum名を指定します。
2. Desc: enumの説明を記述すると、生成されたスクリプトにコメントとして記載されます。
3. Value Name: enumの値の名前を指定します。書き出されるコードではPascalに変換されるので、ここでは大文字である必要はありません。
4. Auto Increment:　整数の値を指定したい場合はFalseにします。Trueの場合、整数の値は前の値に1を加えた値になります。
5. Integer: 整数の値を指定したい場合、Auto IncrementをFalseにした上で、値を指定します。



6. Value Desc: enumの値の説明を記述すると、生成されたスクリプトにコメントとして記載されます。

オプション

エクセルの入力バリデーションに使用するシートです。（ユーザーが編集する必要はありません）

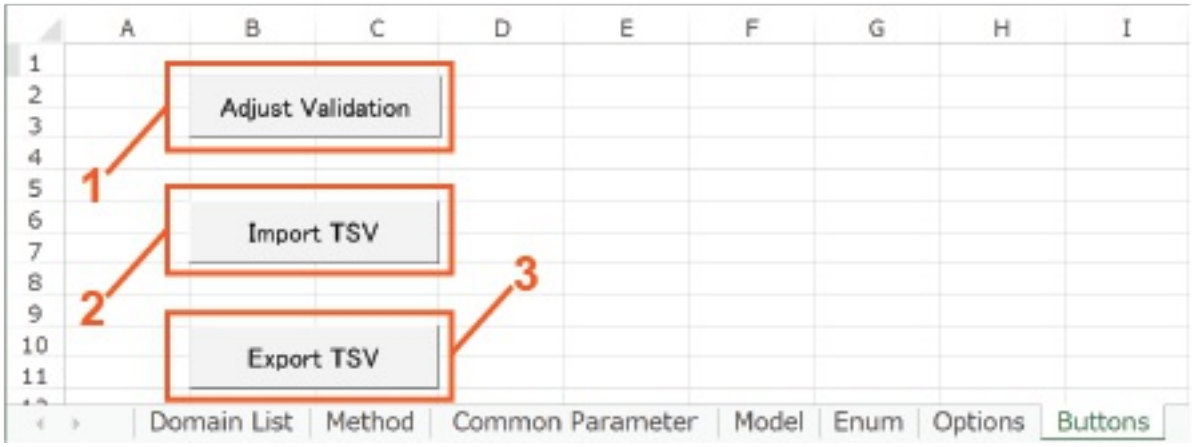
	A	B	C	D	E	F	G	H	I	J	K
1	Domain Name	Method Type	Boolean	Return Model	Parameter Type	Model Type	Enum Name	No Option	Type	Base Model	
2	Basic	POST	TRUE	Void	----	CommonVO		----	String	----	
3		GET	FALSE	CommonVO	CommonParam	ScoreListVO			Int	CommonVO	
4				ScoreListVO		ScoreVO			Long	ScoreListVO	
5				ScoreVO					Boolean	ScoreVO	
6									Float		
7									Bytes		
8									Enum		
9									Model		
10									List<String>		
11									List<Int>		
12									List<Long>		
13									List<Boolean>		
14									List<Float>		
15									List<Bytes>		
16									List<Enum>		
17									List<Model>		
18											

Domain List | Method | Common Parameter | Model | Enum | Options | Buttons

マクロ実行ボタン

マクロを実行するためのボタンがあるシートです。
TSVファイルは、それぞれ以下のシートの内容と同等です。

- wsg_domain.tsv: Doman List
- wsg_common.tsv: Common Parameter
- wsg_method.tsv: Method
- wsg_model.tsv: Model
- wsg_enum.tsv: Enum



1. Adjust Validation: 各シートに入力バリデーションを適用します。行の削除やコピペなどで入力バリデーションが消えてしまった場合、このマクロを実行することで再適用できます。
2. Import TSV: TSVファイルを読み込みます。
3. Export TSV: TSVファイルを書き出します。

その他

入力バリデーションのカスタマイズ

ユーザーが文字入力を行う箇所では入力バリデーションが設けてあり、規定以外の文字列が入らないようにしていますが、これをカスタマイズすることが可能です。
WSGMainWindow.cs の PrepareGUICommonResources のメソッドの中で、入力バリデーション用の正規表現を再定義してください。

- 基本設定用
    - NamespaceRegex: 名前空間の入力
    - PrefixRegex: 接頭辞の入力
    - SavePathRegex: 保存パスの入力
    - TestCodeNameRegex: テストコードの名前の入力
    - TestCodeNameSpaceRegex: テストコードの名前空間の入力
    - TestCodeScenePathRegex: テストコードのシーンの保存パスの入力
    - TestCodeScriptPathRegex: テストコードのスクリプトの保存パスの入力
    - DomainNameRegex: ドメイン名の入力
    - DomainURIRegex: ドメインのURIの入力
- メソッドリスト用
    - EndpointRegex: エンドポイントの入力
    - ParamNameRegex: パラメーター名の入力
    - CommonParameterNameRegex: 共通パラメーター名の入力
- モデルリスト用

    - ModelNameRegex: モデル名の入力
    - PropertyNameRegex: プロパティ名の入力

- enumリスト用

    - EnumNameRegex: enum名の入力
    - EnumValueRegex: enumの値の名前の入力