

# Simple IAP System

## Documentation

V5.2

Docs .....	1
Scripting Reference .....	1
1 Getting Started .....	2
2 Creating In-App Products in Unity .....	3
3 Instantiating Products in the Shop .....	5
4 Customizing Shop Items .....	6
5 Example Scenes .....	7
6 Encrypting device storage .....	13
7 Receipt Validation .....	14
8 Programming & IAP Callbacks .....	16
9 Contact .....	18

Thank you for buying Simple IAP System!  
Your support is greatly appreciated.

## Docs

<https://flobuk.gitlab.io/assets/docs/sis/>

## Scripting Reference

<https://flobuk.gitlab.io/assets/docs/sis/api/>

# 1 Getting Started

## Install Instructions

<https://flobuk.gitlab.io/assets/docs/sis/install/>

For video tutorials (might be outdated), please visit the [YouTube channel](#).

### Custom Extensions

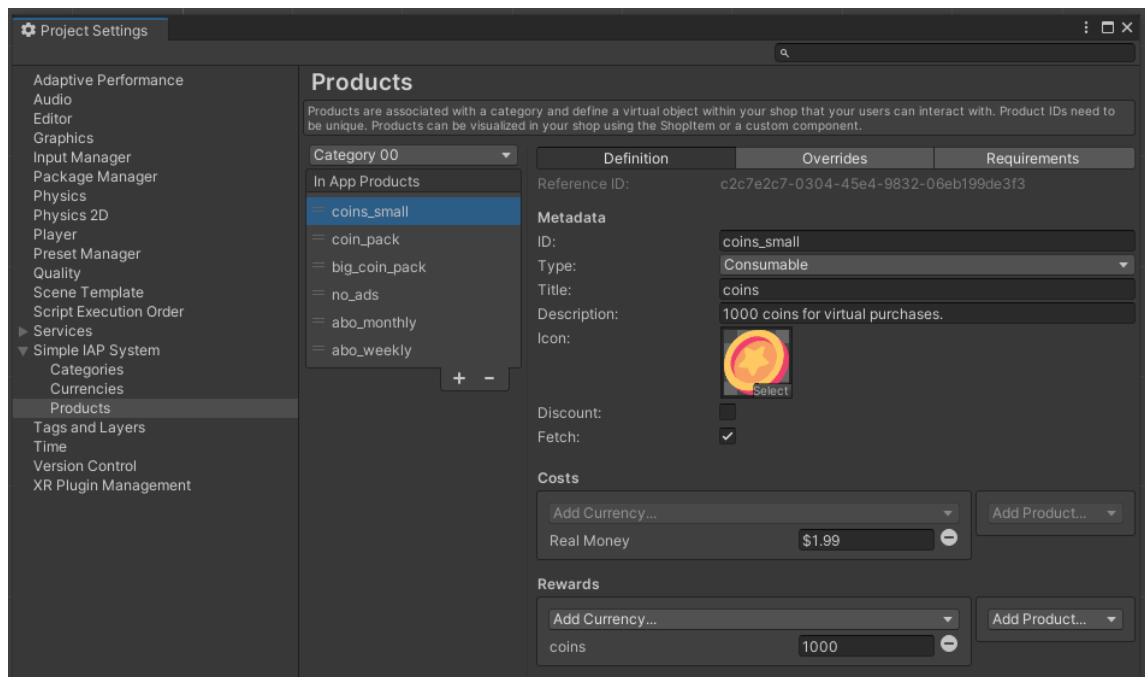
Scripts for more specialized use cases, such as Virtual Reality applications and third-party SDKs, are shipped in a separate package and can be imported via buttons in the Setup tab.

### App Store Guides

Before linking in-app products for real money in Unity, you have to create them in the App Store (Google Play, iTunes Connect, etc.) first. If this is the first time you get in touch with in-app purchases, please have a look at the store-specific guides on [the docs pages](#) for successful configurations on App Stores, but also guides for third party integrations.

## 2 Creating In-App Products in Unity

In Unity, the Project Settings tab is your main spot for managing IAPs, be it for real or virtual money.



**Categories:** here you specify groups for your in-app products. Categories are used for separating several products into sections, which you can later instantiate into your shop UI.

**Currencies:** here you can define virtual currencies along with their starting amount, which can be used to purchase other virtual products in your app.

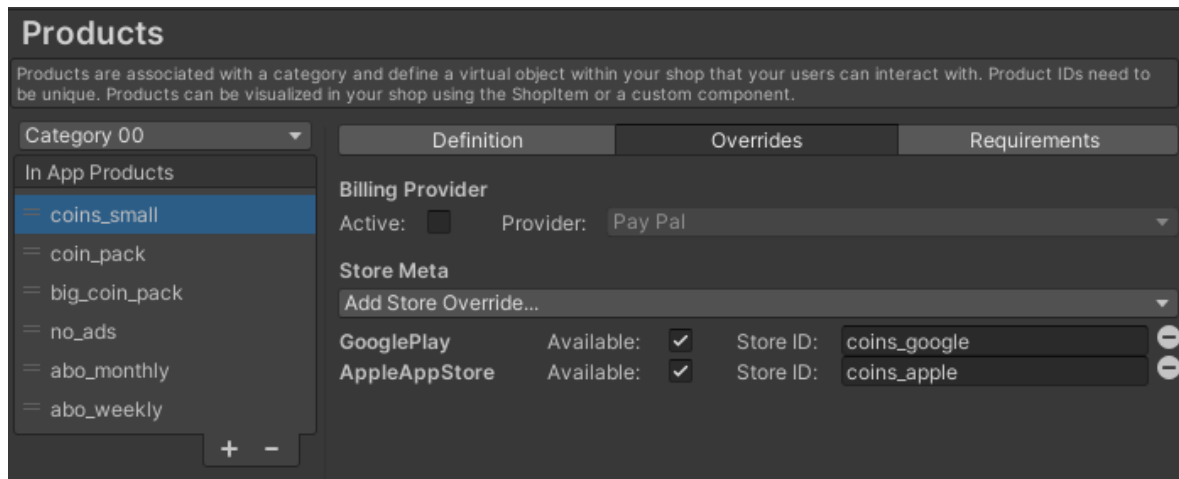
**Products:** this section allows adding in-app products that can be bought with real money (identifier must match your App Stores identifiers), in exchange for virtual currency (only exist within your app), or even physical goods using third-party payment providers.

These are the product variables which need to be defined in the editor:

- **ID:** your unique product identifier
- **Type:** product type in the billing model
  - Consumables: can be purchased multiple times, for real money/currency (e.g. coins)
  - Non-consumables: one time purchase for real money/currency (e.g. bonus content)
  - Subscriptions: periodically bills the user, for real money (e.g. service-based content)
- **Costs:** price for real money / virtual currency, or other products that will be consumed on purchase.
- **Rewards:** amount of virtual currency, or other products to be granted automatically. Please have a look at the sample scenes and compare the different settings for products that are non/consumables, consumed immediately or saved for later.
- **Fetch:** whether local product data should be overwritten by fetched App Store data (only applies to real money products). In addition, when using third party services (PlayFab), virtual product data will be fetched from their dashboard as well.

## Overrides

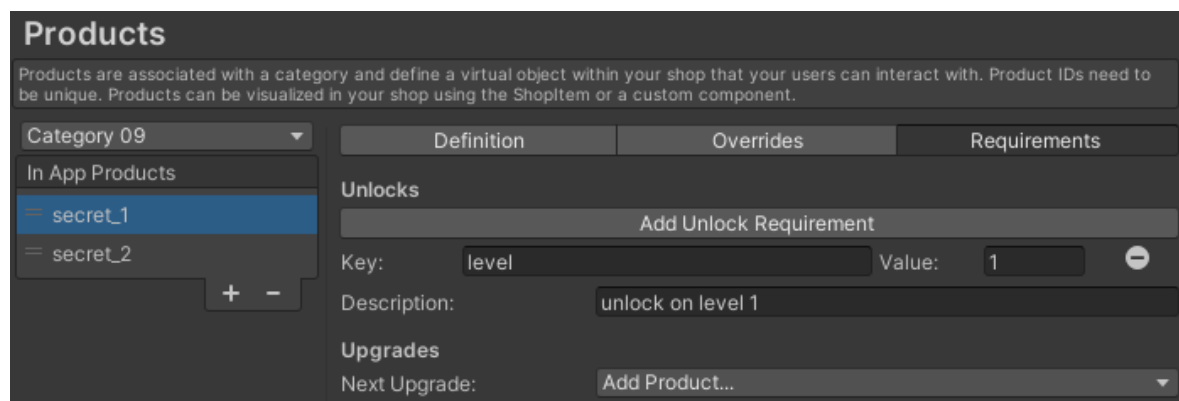
- **Billing Provider:** this allows specifying an independent, third-party billing provider when trying to purchase this product. If you would set “PayPal” to Active here, this basically converts the product into an out-app purchase. This can be used for selling physical goods or donations.
- **Store Meta:** if you are deploying to several App Stores and have different identifiers for the same product, you need a way to ‘merge’ them somehow. That’s what this section is for. In this example, if your product identifier is “coins\_small” on all App Stores except Google Play and Amazon, you can override those by adding them to the platform overrides section:



Additionally, if the product is not available on some stores, you can set the **Available** checkbox to OFF. The product is then not passed into Unity IAP and will not get auto-instantiated when the category is loaded in your store.

## Requirements

- **Unlocks:** the user has to meet one or more Key-Value pairs in PlayerData in order to unlock the ability to purchase this product. Description is displayed on the “Locked Label” Text in a ShopItem.
- **Upgrades:** you can have chains of products which should only be available one after another, after purchasing the base/previous product. Subsequent upgrades of a product do not need to be displayed in the shop - the base product will be replaced by the current upgraded product.

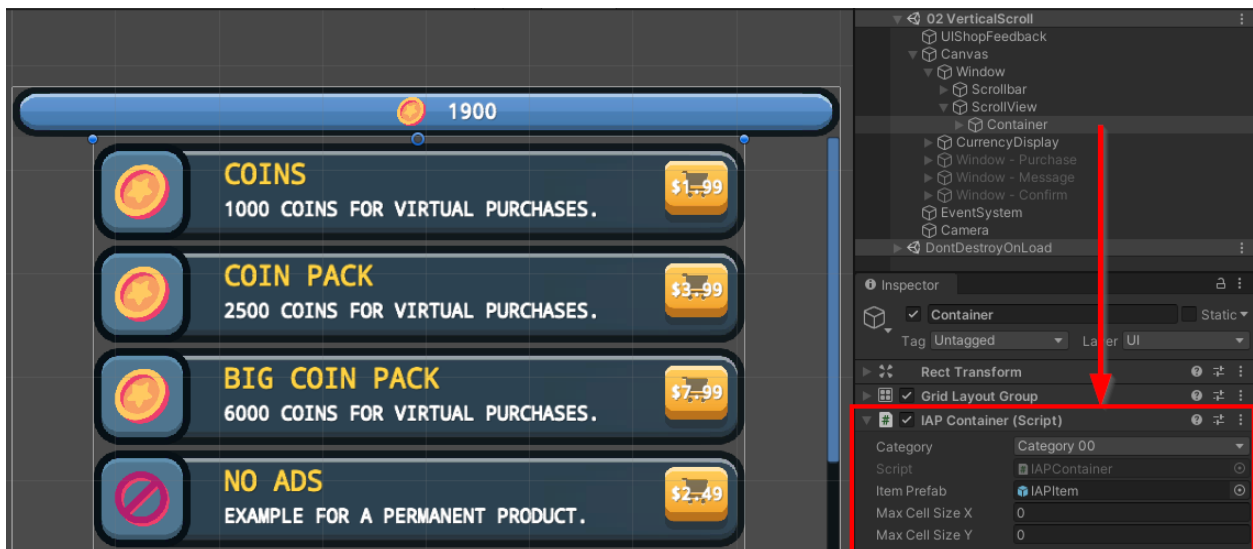


### 3 Instantiating Products in the Shop

The **IAPContainer** component in your shop scene will do all the work for you. Let's have a look at it.

🔗 Open one of the example scenes, *SimpleIAPSystem* > *Scenes* > *VerticalScroll*

When you run the scene, the corresponding category IAPs are instantiated and parented to the container, and Unity's **GridLayoutGroup** component aligns them nicely on the **ScrollView**.

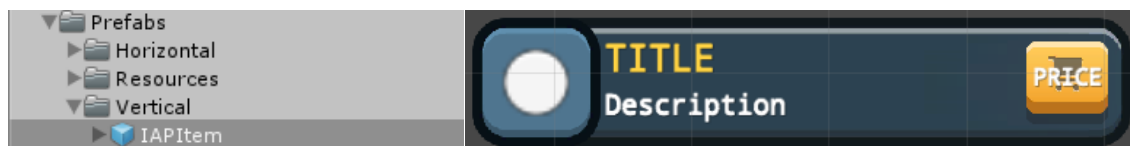


You only have to define what visual representation (**Item Prefab**) and from what group (**Category**) you want to instantiate your products. There are several pre-defined IAP Item prefabs to choose from, located in the *SimpleIAPSystem* > *Prefabs* > *Vertical/Horizontal* folders.

After instantiating the products as items in the scene, the **IAPManager** makes sure to set them to the correct purchase state, based on the user's inventory. This means that the shop item for an already purchased product does not show the buy button anymore, or an equipped item has a button to deselect it again and so on.

## 4 Customizing Shop Items

As mentioned in the last chapter, ShopItem prefabs visualize IAP products in your shop at runtime. These prefabs have an ShopItem2D component attached to them, that has references to every important aspect of the item - to descriptive labels, the buy button, icon sprite and so forth. Based on the product's inventory state, shop items show or hide different portions of their prefab instance.



Shop items can have several different states:

<b>Default (initial state)</b> 	<b>Discounted (highlighted)</b> 
<b>Purchased (user owns this product)</b> 	<b>Single Select (unequips others in category)</b> 
<b>Multi Select (does not unequip others)</b> 	<b>Locked (requirement not met)</b> 
<b>Bundle Preview</b> 	

Item state is defined by user inventory and what references you assigned to the ShopItem component:

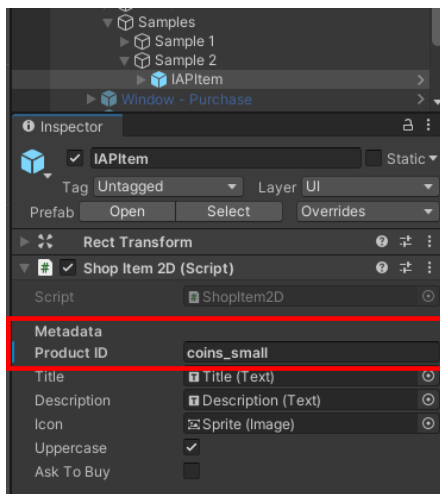
- **Default:** the item needs at least descriptive labels (title/description, price) and a buy button
- **Discounted:** assign a game object to the 'Discounted' slot, which gets activated manually (enabling the 'Discount' checkbox on the product page) or by doing promotional sales on third-party services
- **Purchased:** assign a game object to the 'Sold' slot, which gets activated on purchased products
- **Single Select:** assign the 'Select Button', but leave the 'Deselect Button' empty
- **Multi Select:** assign both 'Select Button' and a 'Deselect Button'
- **Locked:** prepare your prefab for the locked state first by disabling the buy button, descriptive labels etc. and showing the 'Locked Label' and 'Hide On Unlock' game objects. If the item gets unlocked, it hides 'Hide On Unlock' and shows 'Show On Unlock' assignments instead
- **Preview:** in case the product grants other products as reward, you might want to allow users to preview the product's contents before they buy it. Assign a 'Preview Button'

Please note that all UI elements in Simple IAP System use Unity's default UI package. Explaining how to work with UI in Unity would be out of scope for this documentation, but if you are not familiar with it, please refer to the [official documentation](#) or [video tutorials](#) provided by Unity.

## 5 Example Scenes

For a consistent design, it is highly recommended to import your own images and build shop scenes and IAP Item prefabs to match your game's style. This drastically improves conversion rates. Let's go over the demonstrated functionality of the templates included. Each example scene showcases a unique feature of the asset:

### 01 Buttons



If you do not want to use automatic instantiation, or only have one or two products, you can also drag & drop a product into the scene directly:

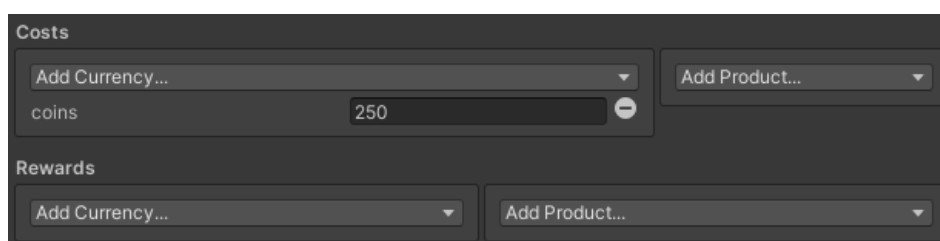
- With a button calling `IAPManager.Purchase(productId)`
- Or using a ShopItem prefab and entering the product ID in the inspector, so that the connection to the referenced product in the Project Settings can be established

### 02 VerticalScroll / 03 HorizontalScroll

These scenes showcase automatic ShopItem instantiation for a category, using the IAPContainer component in a ScrollView. The concept of the IAPContainer has been explained [in this chapter](#). At the top of these scenes, you can also see a display of the current currency amount, retrieved and set via the 'CurrencyContainer' script.

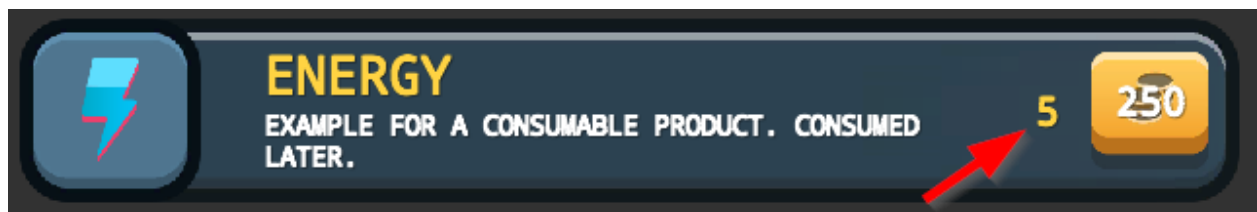
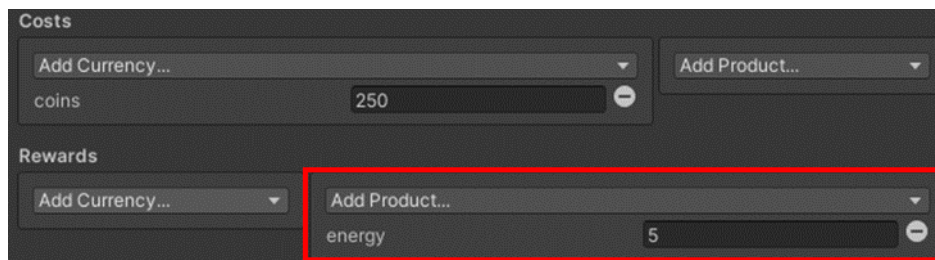
### 04 VirtualPurchase

This scene demonstrates purchasing a consumable or non-consumable product in exchange for virtual currency (coins). If you do not have enough coins yet, you can 'buy' some in the previous scenes. Since the consumable product has no reward, it is consumed instantly and therefore not persisted.



## 05 VirtualPurchaseCount

There is only one product in this scene, granting charges to be consumed later via a rewarded product amount. The current amount purchased is stored locally and displayed next to the product price.

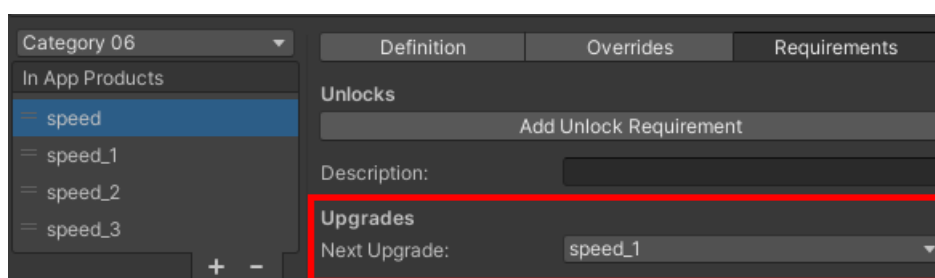


You can consume them later programmatically, by calling `IAPManager.Consume(productID, amount)`.

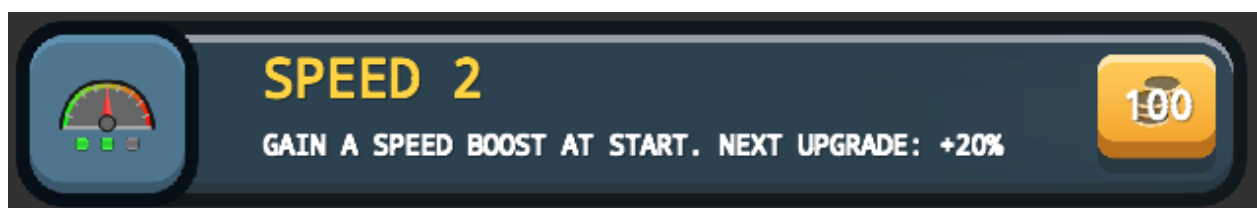
## 06 Upgrade

The category for this scene has 4 products, however in the scene you can only see one: that's because we do not make use of automatic instantiation, but placed a prefab in the scene and assigned the base product 'speed' to it in the inspector.

We do it this way because not all products should be visible at the same time, only the currently active upgrade. An alternative is to use different categories, i.e. placing all the upgrade products in a separate category. On each respective product page, the next product is assigned:



If you buy the base product in the game, it is swapped out with the next upgrade, and so forth.

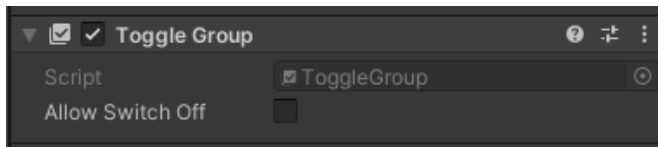




## 07 SingleSelect

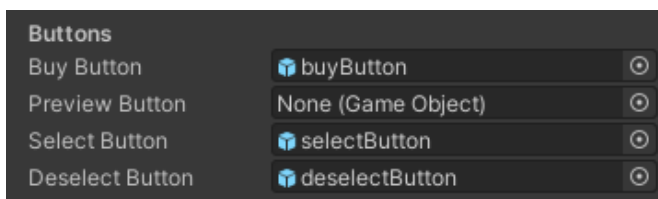
A single-select group can have one product selected at all times. If you purchase and select another product in the same group, the previous one gets deselected. There is no button to deselect a product. And since the first one has no cost defined, it gets selected automatically as soon as the scene loads.

In addition to the item prefabs not having a deselect button, on the IAPContainer game object there is one additional script responsible for the single-selection behavior: the ToggleGroup component.



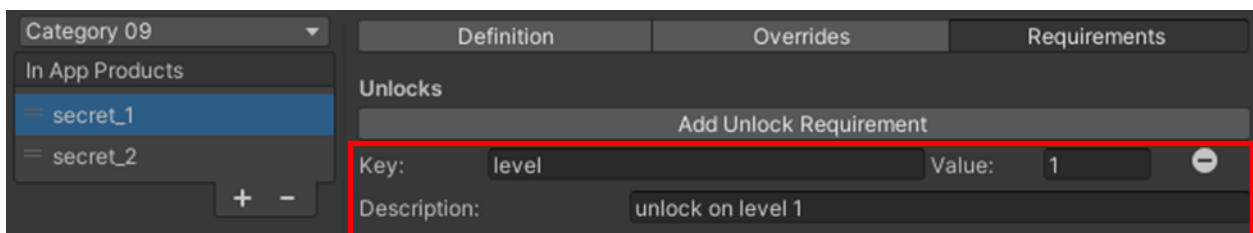
## 08 MultiSelect

In contrast to the single-select group, a multi-select group does not need a ToggleGroup component at all. The instantiated item prefabs just have both a select and deselect button, that's all there is to it.

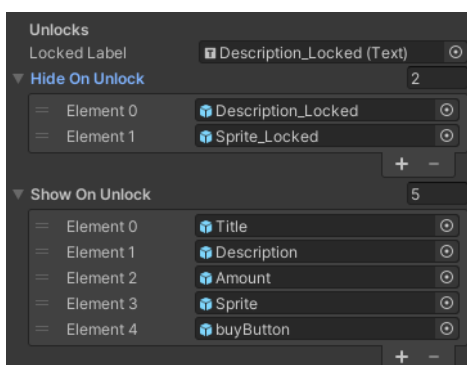


## 09 Locked

This scene features two locked products, which have to be unlocked by reaching a specific criteria. The criteria is the 'level' value, which needs to be 1 or 2 respectively. For testing purposes, there is a button in the scene that let's you increase the level value.



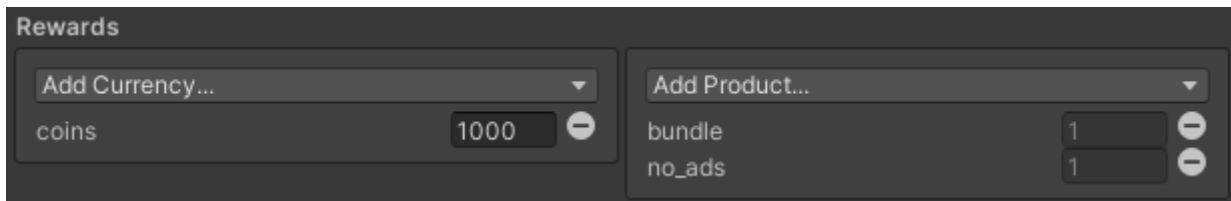
Requirements make use of local PlayerData, modified via DBManager.SetPlayerData/AddPlayerData.



The item prefabs for the locked behavior have the 'Locked Label' assigned, as well as game objects to show/hide in a locked or unlocked state.

## 10 Bundle

A bundled product allows you to grant multiple other products with only one purchase, e.g. with a cheaper price. This works by adding more rewards in addition to the product itself (if it's a non-consumable). Note that it is usually not a good idea to mix & match consumable and non-consumable rewards, as users might try to restore them multiple times in order to receive an endless amount.



The scene contains the 'Window - Preview' prefab to display the bundle's contents.

## 11 RestoreTransactions


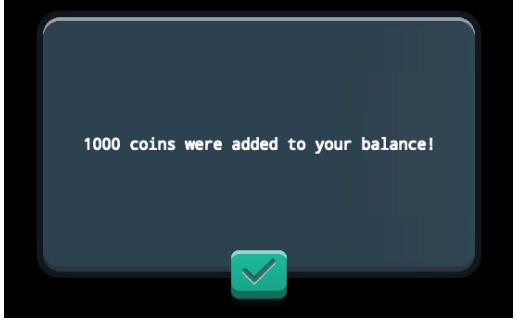
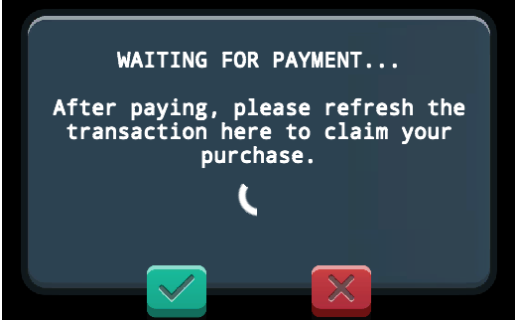
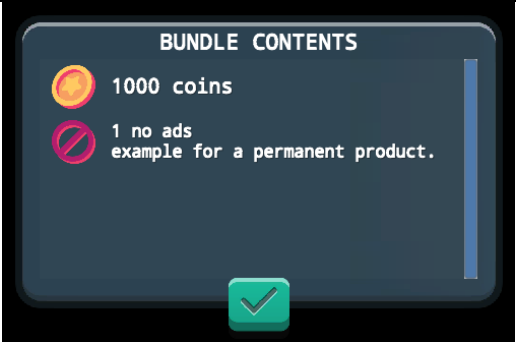
This scene has two buttons:

- Clear PlayerPrefs: despite the title, this button not only clears all local storage but also all internal transactions saved by Unity IAP, so they can be re-processed and treated as new transactions when restored. This should not be used in a production app and is only here for debugging purposes.
- Restore Transactions: having a restore button in your game is mandatory, so users can easily reclaim their previous purchases. Note that all progress earned and saved locally via the DBManager is lost eitherway, as well as consumable purchases, since the App Stores do not keep track of them.

### UI Window Prefabs

The example scenes not only contain products, but some windows for extended functionality, too. Note that each of these windows can be customized, replaced by your own or removed completely.

- **UIShopFeedback:** this prefab contains references to all windows. As such, its only purpose is to manage and present feedback to the user in different scenarios. This acts as the main interface between user actions invoked by other scripts.

<ul style="list-style-type: none"> <li>• <b>Purchase:</b> presents a confirmation popup, to avoid purchases or spending virtual currency by mistake. This is optional but highly recommended, since virtual purchases do not have any other native confirmation popup.</li> </ul>	 <p>A dark blue rounded rectangle with the title 'PURCHASE' at the top. In the center is a red heart icon with the word 'health' in yellow text below it. At the bottom are two buttons: a green checkmark and a red 'X'.</p>
<ul style="list-style-type: none"> <li>• <b>Message:</b> displays a feedback text in case a product has been bought, consumed or otherwise failed to process. You can also call <code>UIShopFeedback.ShowMessage</code> manually to show a message at any other time.</li> </ul>	 <p>A dark blue rounded rectangle with the text '1000 coins were added to your balance!' in the center. At the bottom is a green checkmark button.</p>
<ul style="list-style-type: none"> <li>• <b>Confirm:</b> tells that a purchase transaction is currently in process, with a close button to cancel it early. This is needed on payments happening outside the app, as with PayPal. The transaction is either refreshed automatically or by the user to verify its completion.</li> </ul>	 <p>A dark blue rounded rectangle with the title 'WAITING FOR PAYMENT...' at the top. Below it is the text 'After paying, please refresh the transaction here to claim your purchase.' and a loading spinner icon. At the bottom are two buttons: a green checkmark and a red 'X'.</p>
<ul style="list-style-type: none"> <li>• <b>Preview:</b> shows contents of a product bundle, i.e. if a product contains other products as rewards. This is enabled from the preview button of a <code>ShopItem</code>.</li> </ul> <p>Note that you should avoid adding consumable products to a bundle that can be restored, since users could try to get unlimited amounts for them.</p>	 <p>A dark blue rounded rectangle with the title 'BUNDLE CONTENTS' at the top. Below it are two items: '1000 coins' with a gold coin icon and '1 no ads example for a permanent product.' with a red 'X' icon. At the bottom is a green checkmark button.</p>

## Custom Extensions Example Scenes



### VR

The default shop scenes don't work in virtual reality for two reasons: input is not a regular click or touch and the UI canvas needs to be in world space. That's why this scene is separate, along with additional shop item prefabs (IAPItemVR, IGCItem\_SingleSelectVR). Components of Unity's [VR Samples project](#) have been used in order to support VR devices. To give you a brief overview of the mechanics used for this scene:

- The MainCamera has several VR scripts attached along with a reticle & selection radial
- Each UI button requiring user confirmation (such as the buy buttons of shop items) have the VRInteractableItem and a BoxCollider component attached to them. This is so that the VREyeRaycaster script on the MainCamera actually recognizes them as confirm buttons
- VRInteractableItem has several UnityEvents mapped to different actions in the inspector

To make UI interaction (hover, clicks) work in VR, the default GraphicRaycaster Unity attaches to a Canvas is of no use. Instead, our VRGraphicRaycaster is a VR compatible version of Unity's source.

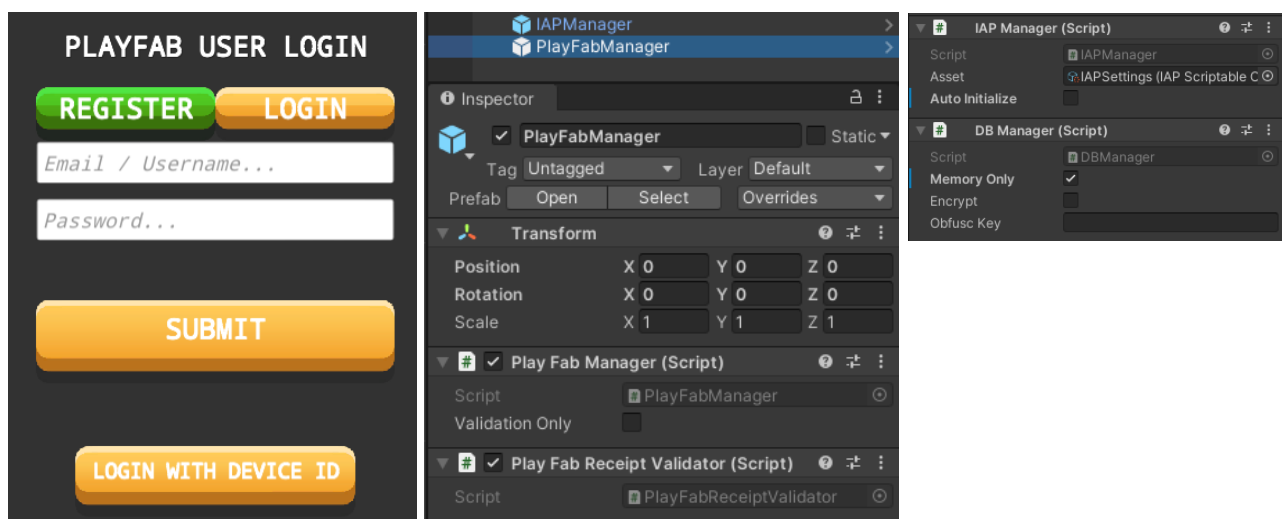


### PlayFab

When using PlayFab, the first scene of your app should offer the possibility to sign-in your users (unless you are using PlayFab for 'Validation Only'). This example scene provides a sample login form. Besides the IAPManager prefab, it also contains the PlayFabManager prefab.

The most important part in this scene are the slightly modified IAPManager settings. Since when logging in, we receive all product data and user inventory from PlayFab, the IAPManager should not initialize itself at app launch, but wait for PlayFab to finish initialization. Additionally, we would not want to save any data locally, and keep everything on PlayFab's servers instead. For this reason, IAPManager's 'Auto Initialize' checkbox is disabled and DBManager's 'Memory Only' checkbox should be enabled.

See the [PlayFab integration guide](#) for more details.

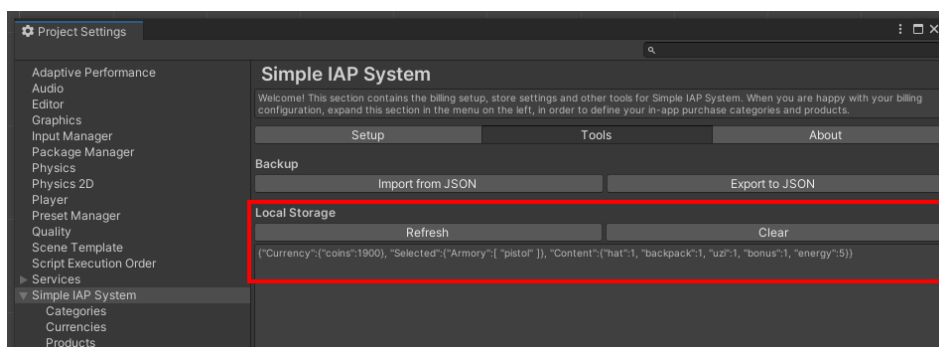


## 6 Encrypting device storage



Optionally you can encrypt the values created by DBManager on the device. On the IAPManager prefab inspector, toggle “Encrypt” to do so. Also, do not forget to replace “Obfusc Key” with your own encryption key (8 characters on iOS/Android, 16 characters on Windows Phone 8). While other techniques are more secure, many App Stores require an encryption registration number (ERN) when submitting your app with those standards. This technique does not require an ERN.

If you are about to submit your app to Apple’s App Store and Apple asks you whether your app contains encryption, click YES. If they ask you whether your app qualifies for any exemptions, click YES again and you’re done.

As the Simple IAP System database is stored in Unity’s PlayerPrefs, you can have a look at it in your registry file. But there is an easier way too: in Unity, you can see what’s currently stored on your device by opening our database display under *Edit > Project Settings > Simple IAP System > Tools*.



In the registry, not encrypted vs encrypted data looks like this:

 data_h2087377941	{\"Content\":{\"no_ads\":\"false\", \"abo_monthly\":\"false\", \"bonus\":\"false\", \"pistol\":\"true\", ...
 data_h2087377941	DHdCnJfciE/vPhR5spmdWoYZVe/xiWi47LkMxkbtwnm1T1Ko37Siej7Ka9A9aO3++52k

It is good practice to clean up that entry by pressing the “Clear” button between IAP testing.

**WARNING:** Please be aware that the PlayerPrefs database implementation (DB Manager) may require a one-time only setup of variables. If you change their values again in production (live) versions, you will have to implement some kind of data takeover for existing users of your app on your own. Otherwise you will risk possible data loss, resulting in dissatisfied customers. Examples:

- Renaming or removing a virtual currency in the Project Settings
- Renaming or removing IAPs in the Project Settings
- Renaming internal storage paths in the DBManager
- Toggling the encryption option or changing obfuscation key in the DBManager

The asset author will not be liable for any damages whatsoever resulting from loss of use, data, or profits, arising out of or in connection with the use of Simple IAP System.

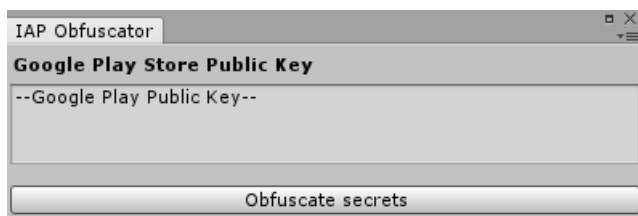
## 7 Receipt Validation

With your IAPs set up, you may want to add an extra layer of security to your app, which prevents hackers to just unlock items by using IAP crackers, sending fake purchases or simply overwrite its local database storage. Receipt verification could help at fighting IAP piracy. With Simple IAP System, you have several options on how to use receipt verification in your app.

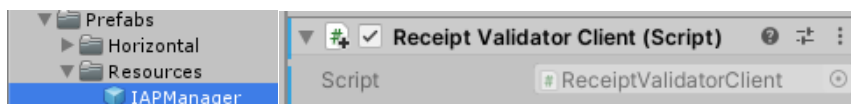
### Client-Side Receipt Verification

This option utilizes the bundle and App Store developer key to verify that the receipt has been created by your app. While doing this check locally (on the client's device) results in a security flaw, this is a very fast approach to add at least some piracy prevention with just a few clicks:

- 1 Open Unity IAP's Obfuscator under *Services > In-App Purchasing > Receipt Validation Obfuscator*, enter your Google Store Key (only if you are deploying to Google Play) and press on the Obfuscate button. This creates additional credential files within your project.



- 2 Locate the IAPManager prefab and add the ReceiptValidatorClient component to it.



- 3 Open the ReceiptValidatorClient script and remove the uncomment lines `//` at line 9 (this is necessary as the script would throw errors without the obfuscation files created before).

```

6  #if UNITY_ANDROID || UNITY_IPHONE
7  //Create your obfuscator secrets
8  //before enabling local receipt validation
9  //define VALIDATION_CLIENT
10 #endif

```

### Server-Side Receipt Verification: <https://flobuk.com/validator>

Being the most secure validation method, we are offering a platform for receipt validation, without requiring your own servers! It supports validation of all products types, detecting fake receipts, active or expired and billing issues within the user's subscription cycle, so you can always let them know to take action in order to stay subscribed. Additional security measures are implemented to ensure a transaction is only redeemed once across your app, preventing duplicate purchase attempts.

### **Service-Side Receipt Verification (required when using PlayFab)**

This option utilizes the PlayFab API + servers for receipt validation. Therefore, you need an active PlayFab developer account (free tier is sufficient) to use this option. Same as server-side, the verification part is not in the hands of your users. The receipt is sent to PlayFab servers on purchase, which validates the transaction with Apple, Google or Amazon respectively, but also checks that it is unique and has not been used before. Since a receipt can only be validated once, this option is not suited for validating active or expired subscriptions, due to the fact that they will be rejected as duplicate. Nevertheless, it is still more secure than client-side validation if you do not intend to implement subscriptions and are using PlayFab anyway.

In order to use it, please see the [PlayFab integration guide](#), section “Using receipt validation only” for reference.

Note that the validation logic for PlayFab has been optimized cost-wise: PlayFab calculates billing for additional API limits based on monthly active users (MAU) using your app. With the “Validation Only” implementation, PlayFab users will only be created at the time they actually make an in-app purchase (and nothing else), so your MAU count stays as low as possible.

### **Custom Receipt Verification**

If you would like to use your own, custom verification on your server, the IAPManager also offers two events that fire either on initialization or on a purchase event:

```
public static event Action receiptValidationInitializeEvent;  
public static event Action<Product> receiptValidationPurchaseEvent;
```

The event on initialization is fired when the IAPManager is successfully initialized and received a list of already purchased products. You could use this event to re-validate subscription receipts or stored local receipts. The purchase event delivers the Unity IAP Product including receipt on a new purchase.

Your script can either subscribe to one, or both events, depending on what you would like to validate. For code samples, please see the ReceiptValidator implementations mentioned previously.

## 8 Programming & IAP Callbacks

Note that most of the programming is already handled for you internally, such as:

- setting a non-consumable/subscription product to ‘purchased’ after it has been bought
- granting virtual currency after a ‘Currency’ product has been bought (via Project Settings)
- adding custom usage amounts for consumable products (via Project Settings)
- subtracting currency on virtual product purchases, and more.

If you would like to present a nice feedback window to the user, then that’s still something you would add in the **IAPListener** (because you can define the text yourself). The **IAPListener** script has a *HandleSuccessfulPurchase* method, which is where you say what happens when a user buys a product, and the *HandleSuccessfulConsume* method which is fired when a product was consumed successfully.

Below are some examples what you can do with programming at any point in your game. All code snippets are examples listed with their proposed script location.

**In the IAPListener: HandleSuccessfulPurchase**

```
case "coins_small":
    //show feedback with custom text
    ShowMessage("1000 coins were added to your balance!");
    break;

case "health":
    //for example, in case product was bought not in the shop,
    //but during the game to receive a second chance or revive
    if(UnityEngine.SceneManagement.SceneManager.GetActiveScene().name == "MyGameScene")
    {
        //try to consume product immediately
        IAPManager.Consume("health", 1);
    }
    else
        ShowMessage("Medikits were added to your inventory!");
    break;
```

**In the IAPListener: HandleSuccessfulConsume**

```
case "health":
    //for example, add health to your custom player class
    Player.GetInstance().AddHealth(75);
    break;
```

Again, you do not want to handle any currency amounts or granting products in the **IAPListener** manually - as written above, this is all handled for you automatically. When looking at the “coins\_small” code, note that only a message is displayed to the user - the amount is added internally.

An exception to this is if you are offering in-app purchases in a popup during the game, or maybe your shop scene is included in the game scene. The “health” code example above first checks for the current scene, where the product has been bought. If you are in the game scene, it tries to consume the product immediately. What should happen is then handled in *HandleSuccessfulConsume*.



To make the update process of Simple IAP System easier, you might want to create your own listener script, subscribe to only the IAPManager methods needed for your game logic and attach it to the IAPManager prefab, instead of using the default IAPListener script.

There are also a range of cases where you want to call IAP methods on game launch, at certain events or checkpoints reached when playing, with some examples listed below.

### In your game logic

At level start:

```
if(DBManager.IsSelected("weapon1")) //check if product was selected
    Player.InstantiateWeapon(...) //give the corresponding weapon to the player

//returns remaining amount of virtual product, display it on a UI label text
label.text = DBManager.GetPurchase("energy").ToString();
```

Before showing ads:

```
if(DBManager.GetPurchase("no_ads") > 0) //check if product has been bought, or
if(DBManager.IsPurchased("no_ads")) //shorthand for the same check
```

During the game:

```
IAPManager.Consume("bullets", 10); //decrease virtual product amount by 10
```

At level end:

```
DBManager.SetPlayerData("score", new SimpleJSON.JSONData(2250)); //save highscore
DBManager.AddPlayerData("xp", 100); //increase current user experience by 100
DBManager.AddCurrency("coins", 200); //increase user's virtual currency by 200
```

You probably noticed that several methods make use of the `DBManager`. The `DBManager` manages our `PlayerPrefs` database and keeps track of purchases, selections, virtual currency and other player-related data. Basically, whenever you want to modify purchase data, you should call methods of the `IAPManager`. For any custom data related to the player, you should call methods of the `DBManager`.

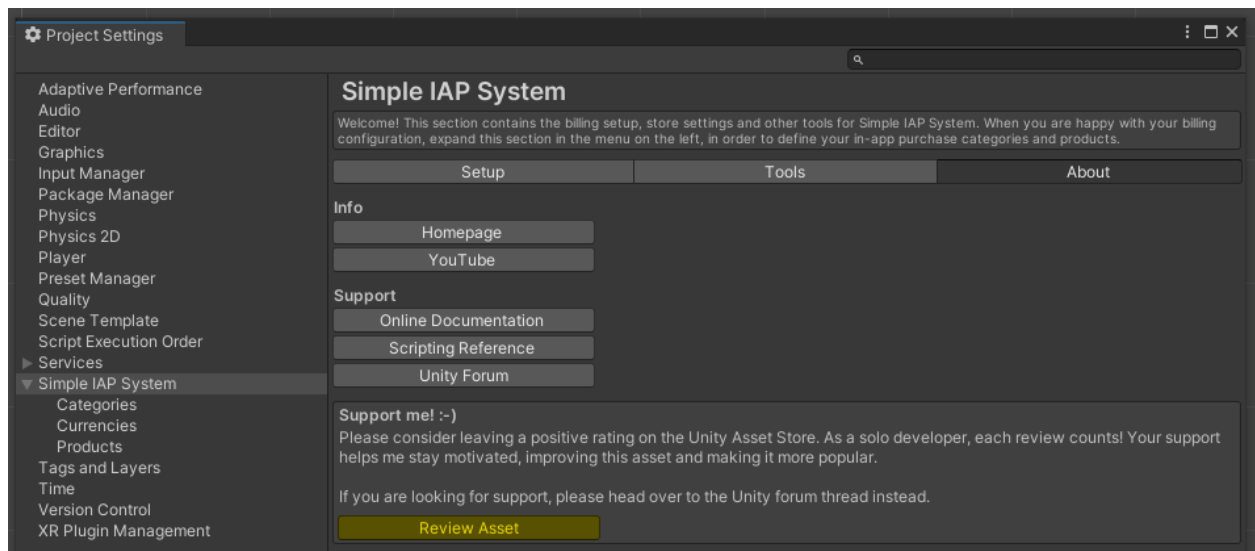
Do not forget to include the namespace by adding `using SIS;` at the top of your own scripts. There are a lot more actions and methods available for integration in your app, which should be flexible enough to cover almost all use cases related to in-app purchases. If a method can be accessed in a static context (e.g. `IAPManager.xxx` oder `DBManager.xxx`), it is designed for you to make use of it.

For a full list, please visit the [Scripting Reference](#).

## 9 Contact

As full source code is provided and every line is well-documented, please feel free to take a look at the scripts and modify them to fit your needs.

If you have any questions, comments, suggestions or other concerns about our product, do not hesitate to contact me. You will find all important links in the 'About' tab, located in the Project Settings under *Simple IAP System*.



Simple IAP System is on the Unity Asset Store since 2013, but I will keep maintaining it and adding more features, as long as users - like you! - are using it.

If you would like to support me on the Unity Asset Store, please add a rating or write a short review there so other developers can form an opinion. And if you would like to support me outside of the Asset Store, please consider using my [Server-Side Receipt Validation Service](#).

Again, thanks for your support, and good luck with your apps!

FLOBUK