

Simple IAP System

Documentation

v4.0

Scripting Reference	1
1 Getting Started	2
2 Creating In-App Products in Unity	3
3 Instantiating Products in the Shop.....	5
4 Customizing IAP Item prefabs	5
5 Programming & IAP Callbacks	7
6 Encrypting device storage.....	8
7 Shop Templates explained	9
8 Receipt Verification	10
9 Contact	12


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

Scripting Reference

www.rebound-games.com/docs/sis

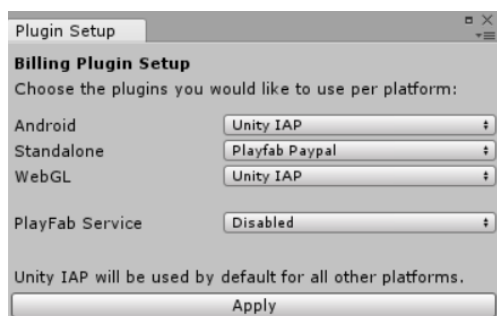
1 Getting Started

For video tutorials, please visit our [YouTube channel](#).

- 1 Simple IAP System requires **Unity IAP** for billing. To enable it, open Unity's services tab . 
Enable your project for Unity In-App Purchasing and import Unity IAP's billing plugins (that's a button in the same window).

Note that **you don't have to** enable Unity IAP if you don't want to charge your users, and only need virtual products and currency - such as buying a sword with coins earned in-game. In this case, just leave Unity IAP disabled in the services window.

- 2 Open our billing plugin setup window under **Window > Simple IAP System > Plugin Setup**. Here you can choose between a selection of billing stores per platform.



- 3 Drag the "IAPManager" prefab from *SimpleIAPSystem > Prefabs > Resources* into the very first scene of your game. Start from the 'AllSelection' scene if you want to run the example scenes of Simple IAP System (the IAP Manager is already in there).
- 4 **Optional:** If you would like to make use of PlayFab services (such as cloud save for in-app purchases, virtual currencies and player data), drag the "PlayfabManager" prefab from *SimpleIAPSystem > Prefabs > Resources* into the very first scene of your game as well.
- 5 Drag the "ShopManager" prefab from *SIS > Prefabs* into the scene where you want to display in-app purchases (your own shop scene, or use one of the templates included).

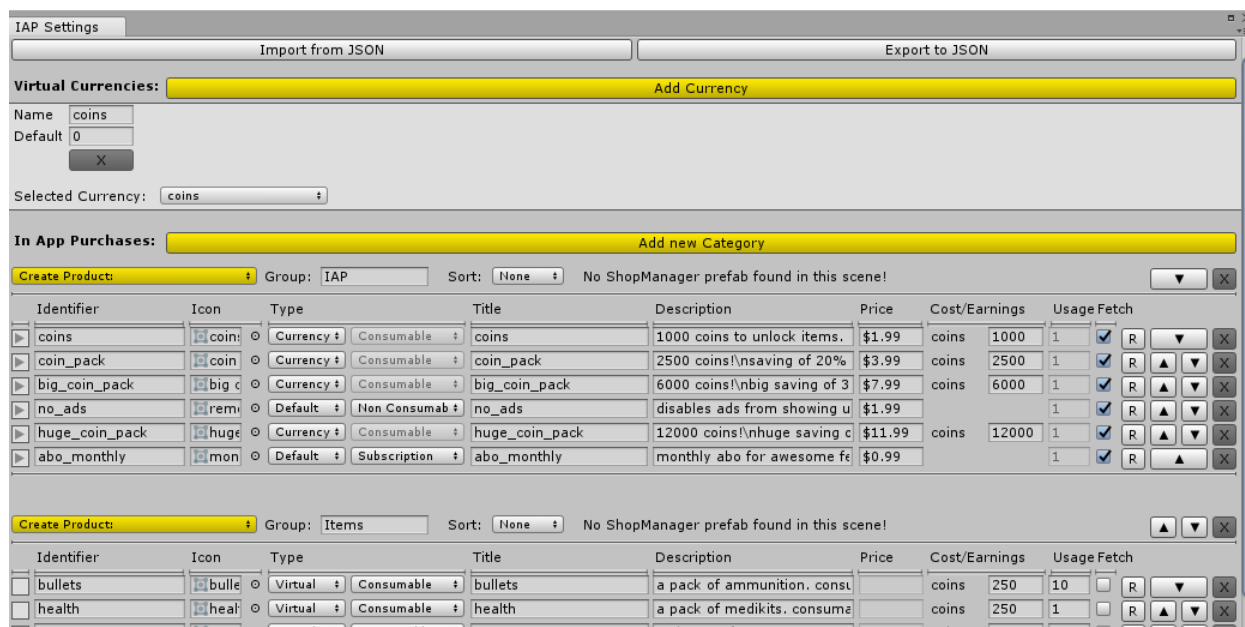
Further billing documentation: Before linking in-app products for real money in Unity, you have to create them in the App Store (Google Play, iTunes Connect, etc.) first. If this is the first time you get in touch with in-app purchases, please have a look at the store-specific guides on [our forums](#) for successful configurations on App Stores.

2 Creating In-App Products in Unity

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



Please open it by navigating to Window > Simple IAP System > IAP Settings.



Virtual Currencies: here you can define virtual currencies along with their starting amount, which can be used for purchasing other virtual products in your app.

In App Purchases: here you specify categories for your in-app products. Each category has a “Create Product” button for adding (real money) In App Purchases (identifier must match your App Stores identifiers), Virtual Currency packs (grant an amount of virtual currency automatically), or Virtual Economy products (virtual products that only exist in your app).

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

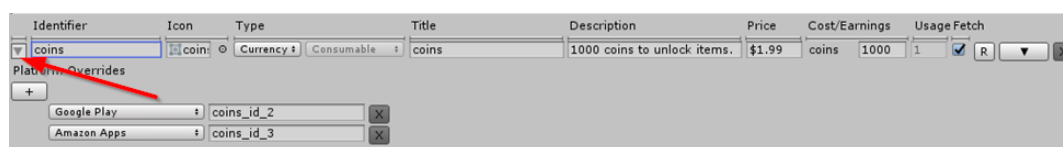
- **Group:** unique name of the group you are adding products to
- **Identifier:** your unique product identifier
- **Icon:** the sprite texture you want to use as the icon
- **Type:** product type in the billing model
 - Consumables: can be purchased multiple times, for real money/currency (e.g. coins)
 - Non-consumables: one time purchase for real money/currency (e.g. bonus content)
 - Subscriptions: periodically bills the player, for real money (e.g. service-based content)
- **Title, Description:** descriptive text for the product in your shop
- **Price:** product price (string value for real money, int value for virtual currency). Virtual products will start as pre-purchased if their price is zero.
- **Cost/Earnings:** product price for virtual economy products (cost), or amount of virtual currency to be granted automatically for purchasing virtual currency packs (earnings).

- **Usage:** When buying consumable products, this is the amount that should be added. Current amount can be accessed via `DBManager.GetPlayerData(product identifier)`. Default is 1. If you set this to 0, nothing is saved as it is treated as consumed immediately. For example: special bullets for selection: amount > 0, temporary power-up bought in-game: amount = 0.
- **Fetch:** whether local product data should be overwritten by fetched App Store data (only applies to real money products). In addition, when using PlayFab, virtual product data will be fetched from the PlayFab dashboard as well.
- **R:** opens the requirement window. The user has to meet this requirement in order to unlock the product first. See the ‘Customizing IAP Item prefabs’ section for more details.

Platform overrides

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 platform overrides are for. They can only be defined for real money IAP products, by expanding the foldout next to them.

In this example, if your product identifier is “coins” on all App Stores except Google Play and Amazon, you can override those by adding them to the platform overrides section:



Some technical prerequisites:

- For virtual products, you will have to specify one or more virtual currencies that should be used first. It is not recommended to rename/remove currencies in production, as this could lead to inconsistency and loss in funds.
- Always keep the IAP Manager prefab in the Resources folder. When making changes to the IAP Settings editor, these changes are automatically saved to the prefab. Thus, if you are upgrading to a new version of SIS, make sure to keep a local copy of your IAP Manager prefab (uncheck the IAPManager prefab & IAPListener script on import).
- For restoring real money purchases in your app, simply add a UI button somewhere in your shop and attach the `UIButtonRestore` component to it. On supported platforms, internally this purchases a product with the identifier “restore”. The IAP Manager will recognize this and call the appropriate restore methods on supported platforms for you. Note that Apple will reject your app if you don’t provide a restore button. On platforms where transactions do not need to be restored, the component automatically disables itself. This is showcased in all example scenes. Alternatively, you could add a consumable IAP product with the identifier “restore”.

3 Instantiating Products in the Shop

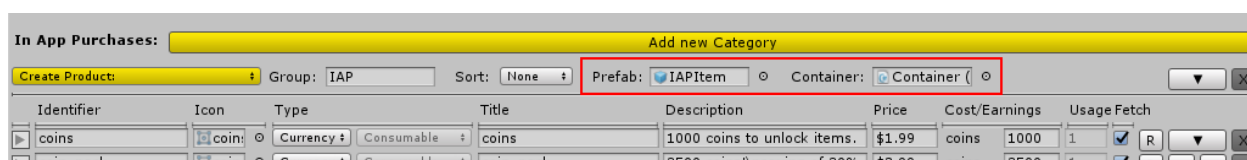
The Shop Manager prefab in your shop scene will do all the work for you. Let's have a look at it.



Please open one of the example shop scenes, e.g. *SimpleIAPSystem > Scenes > Vertical*

For each group in the IAP Settings editor, you only have to define what visual representation (**prefab**) and where (**container**) you want to instantiate your products. There are several pre-defined IAP Item prefabs to choose from, located in the *SimpleIAPSystem > Prefabs > Vertical/Horizontal* folder. When you play the scene, the Shop Manager instantiates this prefab for each item, parented to the container, and Unity's GridLayoutGroup component aligns them nicely in the scene.

Your container transform in the scene needs the IAPContainer component attached to it.



After instantiating your products as items in the scene, the Shop Manager also makes sure to set it to the correct state. This means that the shop item for a purchased product does not show the buy button anymore, or an equipped item has a button to deselect it again and so on.

As public variables, the Shop Manager exposes references to a game object and a UI Text, which are being used for showing a feedback window in case purchases succeed or fail.

4 Customizing IAP Item prefabs

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

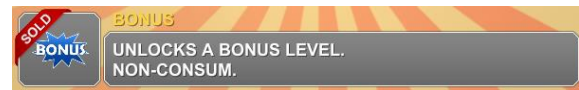


Shop items can have several different states, depending on the previous user interaction with it:

Default (initial state)



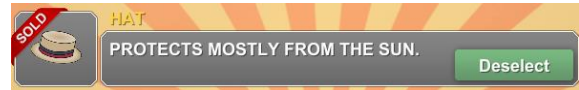
Purchased (user owns this product)



Single Select (unequips others)

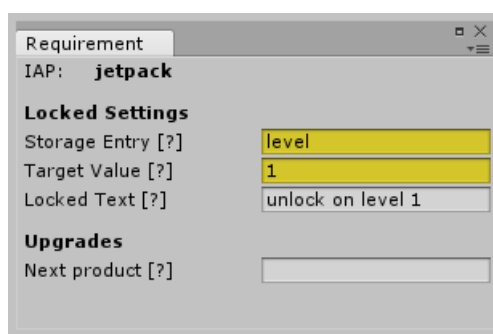


Multi Select (does not unequip others)



Possible states for your item are specified by assigning references to the IAP Item component:

- **Default:** the item needs at least descriptive labels (title/description, price) and a buy button
- **Purchased:** assign a game object to the 'Sold' slot, which gets activated on sold products
- **Single Select:** assign the 'Select Button', but leave the 'Deselect Button' empty
- **Multi Select:** assign both 'Select Button' and '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' instead.



This is the requirement for the jetpack product, defined in the IAP Settings editor. Hover over the labels to see what they mean. For example, here we unlock the jetpack if the player reached level 1.

The vertical/horizontal menu scenes showcase this product in the Custom sub menu.

Upgrades (multiple levels for one product) can be specified here too. Simply enter the product id that comes after this one, and the shop item gets replaced with the next level after the user bought it. Your upgrades do not need to be instantiated in the scene, thus you can leave their Prefab & Container slot empty. The "speed" product is a sample for this, showcased in the Items section.

5 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 IAP Settings Editor)
- adding custom usage amounts for consumable products (via IAP Settings Editor)
- 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, by adding its global identifier (not the platform override) to it.

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 script:

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

At level start:

```
if(DBManager.GetPurchase("no_ads") == 0) //check if product has been bought, or
if(DBManager.isPurchased("no_ads"))      //shorthand for the check above
```

During the game:

```
DBManager.GetPlayerData("bullets").AsInt; //returns remaining amount of virtual product
DBManager.IncreasePlayerData("bullets", -10); //decrease virtual product amount by 10
```

At level end:

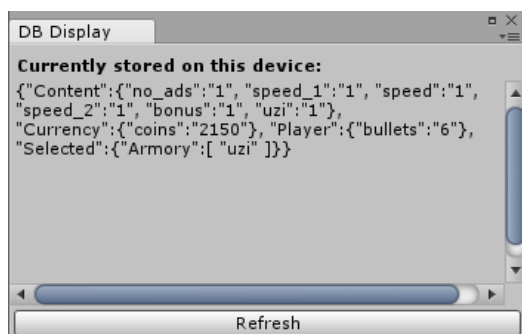
```
DBManager.SetPlayerData("score", new SimpleJSON.JSONData(2250)); //save highscore
DBManager.IncreasePlayerData("xp", 100); //increase current user experience by 100
DBManager.IncreaseFunds("coins", 200); //increase user's virtual currency by 200
```

You probably noticed that several method 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 In App Purchase or custom data related to the player, you should call methods of the DBManager.

Do not forget to include our namespace by adding `using SIS;` at the top of your own scripts. For a full list of available methods, please visit the [scripting reference](#).

6 Encrypting device storage

Optionally you can encrypt the values created by DBManager on the device. On the IAPManager prefab inspector, enable “encrypt” to do so. Also, do not forget to replace “obfuscKey” 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 actually an easier way:

In the Unity Editor, you can see what’s currently stored on your computer by opening our database display window under *Window > Simple IAP System > Show Database*.

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	DHdCnJfdE/vPhR5spmdWoYZVe/xiWi47LkMxdtwnm1T1Ko37Siej7Ka9A9aO3++52k

It is good practice to clean up that entry between IAP testing. Clear your database by opening the *Window > Simple IAP System > Clear Database* window to delete all app-specific data set by SIS.

WARNING: Please be aware that our 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 IAP Editor
- Renaming or removing IAPs in the IAP Editor
- Renaming internal storage paths in the DB Manager
- Toggling the encryption option in the DB Manager

Rebound Games 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 Shop Templates explained

For a unique 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.

- **List** (example scene 'Vertical'/'Horizontal'): these are the most basic samples in SIS and only contain products for real money, no in-game content or currencies. They have a 'Window - IAP' game object with ScrollRects ([?](#)), as well our IAPContainer component attached to the container. IAPContainer dynamically adjusts the cell size of its GridLayoutGroup according to aspect ratios and resolutions of different devices, as well as the width/height of IAP Item prefabs in the scene. IAPContainer also allows you to set the max width/height of an item in case of higher resolutions.
- **Tabs** (example scene 'VerticalTabs'/'HorizontalTabs'): when a single ScrollRect is not enough, maybe several are! These scenes also contain some in-game content products and currencies. Each category has a separate window in the scene, which get activated once the corresponding button on the left side is triggered. For example, the button for Items enables 'Window - Items' but disables all others. These scenes also display amounts of the 'Coins' currency at the top, via the 'UpdateFunds' script attached to a Text component.
- **Menu** (example scene 'VerticalMenu'/'HorizontalMenu'): these are the most complex scenes in SIS and showcase all product types. Instead of just enabling windows, here they are animated. Each window has an Animator component that moves it in or off screen. For example, clicking on the Items button will animate 'Window - Main' off screen and 'Window - Items' will show up. This behavior is handled per button, via two events set up on them. The 'Button - Back' in each window does the exact opposite. Also, pressing 'Button - LevelUp' in the Custom sub menu increases the user level and Shop Manager tries to unlock new items.
- **VR** (example scene 'VerticalVR'): 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 there is a separate VR shop scene, along with additional shop item prefabs (IAPItemVR, IGCIItem_SingleSelectVR) in the *SimpleIAPSystem > Prefabs > Vertical* folder. Components of Unity's official [Virtual Reality Learn Project](#) have been used in order to support Oculus devices. To give you a brief overview of the mechanics used for this VR shop:
 - 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. Our VRGraphicRaycaster is a VR compatible version of Unity's source.

Additional note: if "Virtual Reality Supported" is checked under Player Settings and you are starting the app from the "AllSelection" scene, then the VR shop scene loads automatically.

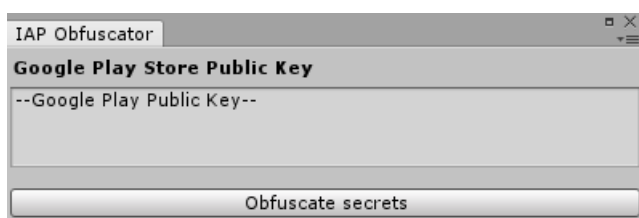
8 Receipt Verification

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.

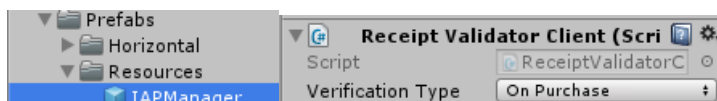
Client-Side Receipt Verification

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 some piracy prevention with just a few clicks:

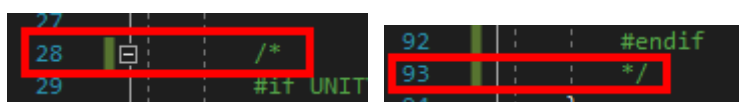
1. Open Unity IAP's Obfuscator under Window > Unity IAP > Receipt Validation Obfuscator



2. Enter your Google Store Key (the same as entered on the IAPManager prefab).
3. Press on the Obfuscate button, this creates additional credential files in your project.
4. Locate the IAPManager prefab and add the ReceiptValidatorClient component to it.



5. Open the ReceiptValidatorClient script and remove the uncomment lines `*/ /*` (this is necessary because otherwise the script would throw errors without the obfuscation files created earlier).



Server-Side Receipt Verification

In this option, the receipt is sent to an external server, which forwards the transaction data to Apple or Google respectively, checks against it and then returns a valid or invalid response to the application. This means that the verification part only happens on the server. Also, if you want to check subscription status and restrict content for expired ones, you will have to use online receipt verification. In order to use this option, some configurations on your server are required, as well as in your iTunes/Google developer account. Due to the advanced settings and experience needed in setting this up, we are offering this as an additional package outside this asset. Please contact us with your invoice number privately via email to request access.

Service 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 their 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. In order to use it:

1. Make sure PlayFab “Full Suite” or “Validation Only” is set in our Plugin Setup window
2. Add the ReceiptValidatorService component to the IAPManager prefab



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.

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 us. You will find all important links in our 'About' window, located under *Window > Simple IAP System*.



If there are any questions, maybe our [FAQ and support forum](#) or [Unity thread](#) has the answer!

For private and/or open questions, you can also email us at
info@rebound-games.com

If you would like to support us on the Unity Asset Store, please write a short review there so other developers can form an opinion. Again, thanks for your support, and good luck!

Rebound Games