Pointshop 2 Documentation

Release 2.2.4

Kamshak

June 21, 2015

Contents

Pointshop 2 is the next generation shop system for Garry's Mod. Its features include:

- Live item editing
- Lua-less creation of items
- Inventory and Item slots
- PAC3 Integration

The script can be found at https://scriptfodder.com/scripts/view/596

DLCs available:

- Boosters: https://scriptfodder.com/scripts/view/648
- Gambling: https://scriptfodder.com/scripts/view/645

This is the documentation site for Pointshop 2. You can find guides and explanation of various options here. If you are a developer you will also find detailed instructions for interacting with the script.

CHAPTER 1

Getting Started

1.1 Installation

To install Pointshop2 simply extract the zip you downloaded into the addons folder. This will give you two folders: *addons/pointshop2* and *addons/libk*.

Next install PAC3, the newest version can always be found here. Download and extract the zip, too. This will give you *addons/pac3-master*.

Final structure when everything is installed:

- **garrysmod**
  - **addons**
    - libk
    - pac3
    - pointshop2

1.2 Configuration

Pointshop2 requires no initial configuration! Once it is installed you are ready to go! Restart your server and press F3 to open the menu. You can find all settings under the "Management" tab.

1.3 FastDL

Unless players already have the files from other places, they will see errors or texture problems if there is no FastDL system. If you haven't already set this up, see here. Pointshop2 automatically uses resource.AddFiles to add all required files to the server downloads. To avoid errors, please upload the following files to your fastdl server:

- addons/pointshop2/materials
- addons/libk/resource

Make sure to upload them into your fastdl's root folder, i.e. *addons/pointshop2/materials* to *fastdl/materials*, NOT *fastdl/addons/pointshop2/materials*.

1.4 Advanced Configuration

There are a few options in Pointshop2 that can be used to integrate the script more tightly into your system.

1.4.1 MySQL Setup

By default SQLite (via sv.db) is used to store all player and pointshop data. If you have a MySQL server you can also use MySQL with the script. This has the advantage of being more efficient for large amounts of data, enabling you to share items across multiple servers and allowing you to display data on the web.

Switching from SQLite to MySQL means that the pointshop is reset - the data is not transferred across. If you wish to keep your data you have to manually export it.

To enable MySQL please follow these steps:

1. **Install the gmsv_mysqloo module**: Download the module from facepunch and follow the installation instructions for your operating system at the bottom of the post.
2. **Enable MySQL within LibK**: LibK is used for all database operations of Pointshop2. To enable MySQL support go into the configuration file *addons/libk/lua/libk/server/sv_libk_config.lua*. Set `LibK.SQL.UseMysql = true` and update the remaining settings with your database connection details. If you are hosting the database on a different machine than the gamemserver, make sure to allow external connections to the database.
3. **Test the configuration**: After a server restart Pointshop2 will now connect to MySQL. If there are any errors when connecting to the database they will be shown in the server console and logged to garrysmod/data/LibK_Error.txt serverside.

1.5 Troubleshooting and reporting bugs

When you are experiencing issues with pointshop 2 please follow these steps. For a fast solution include as much information as possible. Report bugs and problems only through scriptfodder tickets.

1. **Turn on debug mode**: Follow the steps outlined here to enable verbose logging: *Developer Options*.
2. **Create a minimal test case**: Try to reduce the steps needed to create the problem. Once you have found the quickest reliable way to create a problem note down the steps in a step by step fashion.
3. **Capture the load log**: When the script loads it outputs a lot of information to the console. In your report include this, from both server and client console. If something goes wrong during load other errors can happen. The load log is printed on connect (clientside) and after a map change (serverside).
4. **Include client and server console logs**: Include the client and server console logs from when the issue happens. Include a bit before and after, including too much is not a problem. Use a pastebin like http://privatepaste.com/ for storing the information.
5. **Include server configuration**: Are you using MySQL or SQLite? Do you use any custom extensions or any DLC? Which administration mod do you use? Which gamemode do you run?

CHAPTER 2

Pointshop2-Docs

Docs for Pointshop2.

These are the Pointshop2 documentation files. On commit the documentation is automatically updated.

The generator used is the Sphinx documentation generator, which allows a markdown style notation. For more information please refer to the official documentation. If you don't want to read 100 pages check otu the quickstart guide

CHAPTER 3

Item Creation

3.1 Trail Creation

The pointshop comes with a few trails readily available and it is very easy to add your own. If you wish to do this, simply place the files inside materials/trails and as long as this is correct, they will be picked up. When making your own, it is recommended to use a size of 128x128 and in VTF format, as .png can lead to crashes. When handling the .VMT, use these specifics to avoid blurring or blocky previews:

```
"UnlitGeneric"
{
"$basetexture" "trails/dollar"
"$vertexalpha" 1
"$vertexcolor" 1
}
```

To get your image to an actual Pointshop item, go to the second tab called Management. Now you should be on a tab called 'Create Items', with options to select the type. Go to Trail to bring up the basic settings menu, where you can personalize the item you are going to make. Once you are happy with the name and price, click the image preview next to the file path. This will bring up a box displaying all functioning materials inside /trails. Hover over them to play an animated preview, which will give some indication of how they look in-game. Once happy, save the item and if successful, will be found in Uncategorised Items, located inside the Manage Items tab. The last step is to move it into a category, which can achieved via drag and drop.

3.2 Accessory/Hat Creation

Hats in pointshop2 are PAC3 items. As well as the ordinary single prop-based hats you'll find in any Pointshop, here you also have the ability use PAC. This allows you to create animated hats, add particles, position hats interactively onto different models and even create full armor or accessories. This can all be done without any lua knowledge and allows you to easily create hats that are absolutely unique to your server.

When creating a hat you have two options:

1. Using the ordinary PAC3 editor and importing the outfits into Pointshop 2
2. Using the PAC3 editor embedded within the Pointshop2 hat editor

Generally it is recommended to use the second way, this allows you to use the full editor and gives you more options. A common approach is to first create the PAC outfits in sandbox and then use the Hat Creator within the Management Tab to create a pointshop item from it. In gamemodes that do not derive from sandbox (for example TTT or Murder) the PAC editor cannot be used. In this case you can still use the Pointshop2 Hat Editor to create and position hats.

3.2.1 Introduction into PAC

PAC (short for Player Appearance Customizer) is an addon made by CapsAdmin. It allows you to fully customize your looks.

First of all, grab the addon from the Workshop here. Once it's installed, load up Garry's Mod in sandbox and press c (or whatever key is bound to open the context menu). Underneath the small button to change your player model, you'll find another that opens up the PAC editor.

In short, PAC is an addon that allows you to build costumes out of props and various other tools available, such as lighting, particles and even text. To use it, you choose a prop from the sandbox spawn menu, use sliders to move it into a location you're happy with and repeat until finished. The first thing you'll want to do after installing is turning on advanced features. Do this by going to third tab, options, and hitting the second part down, you'll then have easy access to everything.

From here, what you make with PAC is entirely up to you, so it's recommended to just play around until you feel familiar. One aspect that makes costume creation a lot easier is the ability to t-pose your model, meaning they do not move around which can cause issues getting positions exactly right. Access this via the fourth tab, players. It's the first button and can be changed upon demand, if you wish to preview your costume in an animated manner.

When working on PAC items that are not head-based, it is vital to change the bone to one that correctly suits the position of your costume. For example, if you are placing a model on the player's chest, go the orange orientation tab of PAC and click the button for it, which will bring up a list of bones that the prop can be attached to. In this scenario, you would click 'chest' or 'amulet' although it will really depend on what you are making. Failure to do so will usually cause costumes to fall out of place when moving around or in different standing positions.

Further information, tutorials and example items:

- The official Facepunch thread
- The official Github repository
- PAC3 Beginners FAQ
- Tutorials and information

3.2.2 The Accessory/Hat Maker

One of the main features of Pointshop2 is to make item creation much easier and fluid. The Accessory/Hat Maker is the tool used to create PAC3 powered Hats and Accessories. This includes Pets, Hats and various other items that modify the look of the player.

The editor can be accessed within the Management tab under "Create Items". Select Accessory/Hat and the editor will open. To create an item follow these steps:

1. **Fill out the basic information**: Fill out the basic fields such as name, point costs and a description.
2. **Create or import the base outfit**: To do this, simply click on the Open Editor button and select the respective option. This outfit is applied to all models by default. CS:S Playermodels usually require slightly different positioning of the items.
3. **Add model specific outfits**: To modify the item positions for CS:S or other playermodels, click the "Add" button beneath the main editor button. You can now choose to create an outfit for all CS:S models or choose a model manually. After selecting this option you can clone the base outfit and adapt the positions. Please not that the outfit is only cloned when you click this button, if you change the main outfit after cloning the changes will not automatically apply to all model specific outfits. In order to fix this simple reclone the outfit by selectiong the option within the model specific outfits table.
4. **Create a shop icon**: Icons for PAC items are automatically generated. To specify from where the icon should look at the item you can use the icon editor. Within the item positioner you will usually click on the "Icon

Snapshot" button. This will initialize the icon for you. To fine tune the icon's view you can use the sliders next to the icon.

5. **Create an inventory icon**: To update the inventory icon follow the same procedure as for the shop icon. Please note that creating a new icon snapshot will overwrite previous changes. It is recommended that you use the sliders for the inventory icon after creating the shop icon.

3.2.3 Slots

To avoid clipping and keep everything organized, items are categorized by different slots, which can be viewed via the inventory tab. This allows for multiple accessories on the player, such as head, pets, etc. Items are not set to a single slot, meaning they can be used in multiple areas if the user wishes to do so.

To assign an item to a slot simply check the checkboxes in the item editor. Only slots that were created for Accessory/Hat items can be used, so a PAC item cannot be put into a Trail slot.

CHAPTER 4

Developer Information

Pointshop2 was designed with modification, customization and extension in mind. You are welcome to modify or add to the script through coderhire jobs, scripts or private modifications. The following pages are designed to help you work with the addon. Whether you are extending or modifying the script, please follow the guildelines explained here.

4.1 Terms, Agreement and General Guidelines

Pointshop 2 is designed to be a quality script. To maintain this experience to the user addons and modifications need to conform to a high technical quality standard. You are obliged to follow and adhere to these terms, otherwise you are not permitted to modify, use or look into the source code of pointshop 2.

4.1.1 Code Review Process

To enforce the quality requirements to Pointshop 2 addons, it is mandatory to have it reviewed by the author of Pointshop2. Independent of the addon's size and function it is necessary to get approval prior to publishing the script. Pointshop 2 is designed to be extended and modified through external scripts, which is why as a developer you will receive support working with the codebase.

The code review is **free for publicly released scripts** and is available at a **fee of 10$ for every commercial script** (this includes coderhire jobs and scripts). This fee is used to maintain and update the developer documentation, add new functions and APIs for you to use and to provide premium support and advice when writing plugins. It also serves as a guard to protect against "5 second scripts" and low quality, hacked together scripts. Please note that if you do not comply to this agreement, your script will not be approved.

During the code review the code will be checked for exploits, problems with the existing codebase and incompatibilities with existing and future addons. You will receive a written summary of the review and proposed changes. After fixing these problems you will receive a green light to release the addon. Follow up code reviews and code reviews for updates are free.

Support and information during the creation of a script are free. If you have any problems or a question about the codebase or require additional functionality do not hesitate to get in touch via steam, coderhire PM or email.

To summarize: For free (public scripts) or a small fee (commercial scripts) you get:

- Personal support
- Code review and quality assurance
- Compatibility testing against all addons in the program
- Listed on the script's page and officiall endorsement by Pointshop 2

- Promotion through Pointshop 2 communication channels

Requirements that will be checked:

- Support for both, MySQL and SQLite
- Configuration possible through GUI, no lua configs
- Possibility to completely reset the addon
- Adherence to modification guidelines

4.2 Developer License

If you do not plan to use pointshop 2 on a server but want to create an addon to it, you can request a discounted developer license. The license is available for 10$ and includes a **free** code review for a single addon. To request a license please contact Kamshak via e-mail. Please include your experience/previous work and a rough description of what you want to add to the shop.

4.2.1 Modification Guidelines

1. **Never modify code files**: When extending or modifying pointshop 2 the first guideline is to never modify any of the core files. You can use hooks and global functions to extend the script or modify functionality. This makes sure that there are as little conflicts between addons as possible. If you need a specific hook or if modification of core files is a much easier solution to a general problem, please get in touch.
2. **Skin your panels only in derma skins**: To make sure that skins can provide a consistent experience, it is important that Paint functions are never directly overwritten. Instead use Derma_Hook and extend the base skin. This gives skin creator the chance to support your addon without messy hooks and overwrites.
3. **Avoid global variables**: Use local variables or create a single global table for your addon to store globals. This avoids cluttering the global namespace, is faster and avoids conflicts.
4. **Avoid direct database queries**: Pointshop 2 makes heavy use of LibK as a database abstraction layer. This enables easy swapping of the databases and makes it possible to avoid many security issues. While you are not required to use LibK models you are encouraged to do so as it automates table creation and removal and allows you to hook into the export, import and backup mechanisms of Pointshop 2 without additional work.
5. **Use lowercase file and folder names**: To keep linux compatibility please use lowercase file and foldernames.

4.2.2 Getting Started

To get started with Pointshop 2 development, first read the information on module creation. Modules are the main way to modify and extend Pointshop 2. They provide many automated facilities that make it very easy to extend the script.

After that a good way to start is to check out the examples.

4.3 Developer Options

To make development and debugging of the script easier there are a few options for developers. If you experience any errors please also turn these settings on as it will help track errors down.

Within *addons/libk/lua/libk/shared/2_sh_libk.lua* LibK developer settings can be configured.

4.3.1 Defaults

LibK.Debug = false

LibK.LogLevel = 2 –Requires Debug

LibK.LogSQL = false

4.3.2 Developer

LibK.Debug = true

LibK.LogLevel = 4 –Requires Debug

LibK.LogSQL = true

`LibK.LogSQL` logs every query that is generated and sent to the database. This can slow down the server significantly and creates large log files. Only use it if needed.

4.3.3 The reload command

For easier development the `pointshop2_reload` command to fully reload the script was added. It requires LibK to be in debug mode as well as the user to be an administrator. The command will reload every part of the script, including the database. You can use this to quickly test changes without having to change the map. The command only works for existing files, when adding new files you have to do a map change.

Note: The `pointshop2_reload` command is currently broken on linux as file changes are not properly picked up by the game.

4.3.4 Examples

Todo

Add Examples

CHAPTER 5

Module Creation

Modules are the best way to extend pointshop. Through them new item creators can be added.

5.1 Structure

The module structure is very simple. Each module is a folder within the *lua/ps2/modules* folder. Every file in this folder is recursively loaded by the module loader. The realm is determined by the file prefix (sh, cl, sv). Client files are automatically AddCSLuaFile'd.

Within this folder the module description file *sh_module.lua* needs to be placed. Here you set up the module table (`MODULE`) which contains information such as the author and name of the module, then register it using `Poinsthop2.RegisterModule()`.

5.2 Adding custom Item Types

Note: Creating custom item types is an advanced topic and requires an understanding of object oriented patterns. It is recommended to look at the items included in the script as an example when reading this section. Good candidates are the Trail and Playermodel items, which show very simple items. A complex example is provided with the Accessory/Hat, which uses multiple data models and a complex item creator.

Users should be able to create and modify items through the ingame gui.

To understand the way items are created in Pointshop2 it is important to know that each item is handled on it's own and is unique. This means that even if the user owns ten items of the same type each is saved and handled individually. Technically, each item is an instance of a item class. Thus each "item" that is displayed for purchase at a shop is actually an *item class*. Instances of this class are the items which can be found in a player's inventory.

Internally this is achieved using the following components:

- **Persistence**: The persistence is the data model that is used to save all user defined properties (for example the model for a playermodel item, but also the item name and description). On server start the persistence is read and classes are dynamically generated.
- **Item Base**: The item base defines the functionality of an item. This is defined in a lua file within **lua/kinv/items/pointshop**. Item bases can use mixins and extend existing bases. All item bases should extend `base_pointshop_item`.
- **Item Creator**: This is the derma control which is used to create or modify an item. It inherits from `DPointshopItemCreator`.

Note: Unlike in pointshop 1, in pointshop 2 an item is an instance of an item class. A shop item is the representation of that class.

A **blueprint** is defined as the persistence, item base and creator for an item type.

5.3 Creating a persistence

The first step when creating a custom item type is to create it's persistence. A persistence needs to be a LibK model, which is done by including the `DatabaseModel` mixin.

5.3.1 Creating the data model

You can use this template:

```
Pointshop2.ExamplePersistence = class( "Pointshop2.ExamplePersistence" )
local ExamplePersistence = Pointshop2.ExamplePersistence

ExamplePersistence.static.DB = "Pointshop2" --Use the database configured for Pointshop2


ExamplePersistence.static.model = {
    tableName = "ps2_examplepersistence", --needs to be unique
    fields = {
        --holds the reference to the basic information (description, name, price)
            itemPersistenceId = "int",

        --Define your custom fields here
            property1 = "string"
    },
    belongsTo = {
        -- Makes LibK automatically join the basic information table each times
        -- it is received from the database
            ItemPersistence = {
                    class = "Pointshop2.ItemPersistence",
                    foreignKey = "itemPersistenceId",
                    onDelete = "CASCADE" --Persistence is deleted when the base is deleted. This is
            }
    }
}

ExamplePersistence:include( DatabaseModel ) --include the DatabaseModel mixin
```

It is important to specify the `DB` property to `Pointshop2`. This makes the model save to the database as configured for Pointshop2.

The model can be customized to contain as many fields as you need. If you need to save tables or nested data, consider joining another model (and creating a new belongsTo relationship) or simply use a field type that is serialized (json or luadata).

After doing this, a table will automatically be created and the model can now be used with LibK, which means that no queries have to be written to save or update items.

5.3.2 Implementing saving and updating logic

Note: LibK makes heavy use of *promises*. Using promises is required when saving or modifying models. They allow easy handling of asynchronous processes wihtout the need of messy nested callback chains. The promises script used (by Lexic) follows the javascript promises specification and the jQuery interface. More information: General introduction, The jQuery interface documentation

When a pointshop item is created using an Item Creator, the persistence is passed a "save table". This table's structure is defined entirely by your creator. Usually it simply contains the model fields. The same function is called for updating items once they are modified. For this the static function `createOrUpdateFromSaveTable` has to be added. It creates (or on update retrieves) an instance of the own and any required models and then saves it to the database. All fields that the user can configure when creating a custom item need to be included into the model.

Create a new file within your module called **sh_model_<itemname>persistence.lua**.

For simple items you can follow this template:

```
function ExamplePersistence.static.createOrUpdateFromSaveTable( saveTable, doUpdate )
    -- Firstly, save or update the basic item information.
    local promise = Pointshop2.ItemPersistence.createOrUpdateFromSaveTable( saveTable, doUpdate )
    :Then( function( itemPersistence )
        // First we fetch or create our persistence instance
            if doUpdate then
                --We need to update an existing item.
                --Find the instance by using the itemPersistenceId and return it.
                    return ExamplePersistence.findByItemPersistenceId( itemPersistence.id )
            else
                    local exampleInstance = ExamplePersistence:new( )
                    exampleInstance.itemPersistenceId = itemPersistence.id
                    return exampleInstance
            end
    end )
    :Then( function( exampleInstance )
        // Then we update all fields
            exampleInstance.property1 = saveTable.property1

            // And save changes to the database
            return exampleInstance:save( )
    end )

    return promise
end
```

This concludes all of the serverside code that is needed for handling the creation and modification of items.

5.4 Creating the item base

The next step is to create the item base for your item type. To do this, create a new file within **lua/kinv/items/pointshop**. The name should be `sh_base_<itemname>.lua` you can also put your file into a subdirectory. Inside of the item base you can now overwrite any of the pointshop base functions and add item hooks as required.

Todo

Item hook explanation

ITEM.static.**generateFromPersistence**(*itemTable*, *persistenceItem*)
Decodes all information from the persistenceItem and adds fields and methods to the itemTable field.

itemTable: A table containing the created class. **persistenceItem**: An instance of this item's persistence.

The next step is to make sure that the item persistence can be loaded into a valid instance of the item base. This is done by using the static `generateFromPersistence` method. This method is pretty much the opposite of the persistence's createOrUpdateFromSaveTable method (with the exception that a valid item class is created instead of a save table). To generate the item class first call the super class' method by invoking `ITEM.super.generateFromPersistence( itemTable, persistenceItem.ItemPersistence )`. Next simple copy your item's properties over to the item class. You should set these to to the `itemTable.static` table since they belong to a class itself and not an instance (which would be an instantiated item in the player's inventory).

Next you need to link the base to the persistence. To do this simply define a static `getPersistence` method which returns the persistence class used.

Example:

```
ITEM.PrintName = "Pointshop Example Base"
ITEM.baseClass = "base_pointshop_item"

function ITEM.static.getPersistence( )
    return Pointshop2.ExamplePersistence
end

function ITEM:OnEquip( ply )

end

function ITEM:OnHolster( ply )
    --note that ply == self:GetOwner()
end

function ITEM.static.generateFromPersistence( itemTable, persistenceItem )
    ITEM.super.generateFromPersistence( itemTable, persistenceItem.ItemPersistence )
    itemTable.property1 = persistenceItem.property1
end
```

Within the item base you can also specify your own, custom icon controls for both, the shop and the inventory.

5.5 Adding the clientside creator

The last step is to create a custom editor control, which is shown when clicking the create item button. This is very easy to do, simply create a new file inside your module, called `D<youritem>Creator`. It should inherit from `DPointshopItemCreator` and overwrite the `SaveItem(saveTable)` and `EditItem(persistence, itemClass)` methods. The `SaveItem` method populates the save table passed as argument with the settings set in the item creator. The `EditItem` method poulates the editor with the settings stored in the persistence. For ease of access the relevant itemClass is also passed as data from the persistence might be accessible easier in there.

Example template:

```
local PANEL = {}

function PANEL:Init()
    self.textEntry = vgui.Create( "DTextEntry" )
    self:addFormItem( "Property 1", self.textEntry )
end
```

```
function PANEL:SaveItem( saveTable )
    self.BaseClass.SaveItem( self, saveTable )
    saveTable.property1 = self.textEntry:GetText( )
end

function PANEL:EditItem( persistence, itemClass )
    self.BaseClass.EditItem( self, persistence.ItemPersistence, itemClass )

    self.textEntry:SetText( persistence.property1 )
end
vgui.Register( "DExampleCreator", PANEL, "DItemCreator" )
```

5.6 Putting it all together: The blueprint

The only thing left to do now is to link the item to the menu and register it with the modules. This is done within sh_module.lua. Simply define all of your components in a *Blueprint*.

Example:

```
MODULE.Blueprints = {
{
    label = "Example Item",
    base = "base_example", --The name is deduced from the filename
    icon = "pointshop2/playermodel.png", --Icon
    creator = "DExampleCreator"
},
```

5.7 Creating a slot for your item

Slots are created using the function *Pointshop2.AddEquipmentSlot()*

Example:

```
Pointshop2.AddEquipmentSlot( "Example", function( item )
    --Check if the item is an example item
    return instanceOf( Pointshop2.GetItemClassByName( "base_example" ), item )
end )
```

5.8 Adding custom Settings

Pointshop 2 has a builtin, extensible settings system. A module can add custom settings buttons to the builtin settings tab (Management -> Settings) which can then be used to create a GUI. The system first initializes the settings from the Lua table and copies the defaults, then reads settings from the database. To create custom settings you need the following components: the settings table, a settings button and a settings editor.

5.8.1 The Settings Table

The settings table is a table defined inside of *sh_module.lua*:

```
MODULE.Settings = {}
MODULE.Settings.Server = {}
MODULE.Settings.Shared = {}
```

The table is devided into server and shared settings. Shared settings are synchronized with all clients, server settings are only available on the server. Each of these tables can contain multiple :lua:class:‘SettingsCategory‘s. A category consists of a path, an info table and a number of Settings attached to it. Example:

```
MODULE.Settings.Server.Kills = {
    info = {
        label = "Kill Rewards"
    },
    DelayReward = {
        value = true,
        label = "Delay Rewards until round end",
        tooltip = "Use this to prevent players to meta-game using the kill notifications. Kill points
    },
}
```

In this example a server-side category “Kills” is created, with the label “Kill Rewards” and a single (boolean) setting called DelayReward. The path of this setting would be “Kills.DelayReward”. You can add as many categories and settings as you like. Be careful not to define a setting in both, the shared and server table with the same path which could lead to conflicts.

5.8.2 Settings Button

The next step is to define a button which can be used to open the settings editor. This is also done within *sh_module.lua*.

Example:

```
MODULE.SettingButtons = {
    {
        label = "Point Rewards",
        icon = "pointshop2/hand129.png",
        control = "DTerrortownConfigurator"
    }
}
```

This defines a button with the label “Point Rewards” and the icon “pointshop2/hand129.png”. On click a DTerrortownConfigurator control is created. The control should implement the *Configurator* interface. For details see the next section.

5.8.3 Adding the Configurator

Within your module create a new clientside file where you define the configurator control. The configurator control is a derma control which has the methods of the *Configurator* interface.

The easiest way is to simply create a control inheriting from `DSettingsEditor` and using the method `AutoAddSettingsTable`. This automatically populates the settings window with the appropriate input elements for each type you supplied in the settings table.

Example:

```
local PANEL = {}

function PANEL:Init( )
    self:SetSkin( Pointshop2.Config.DermaSkin )
```

```
    self:SetTitle( "TTT Reward Settings" )
    self:SetSize( 300, 600 )

    self:AutoAddSettingsTable( Pointshop2.GetModule( "TTT Integration" ).Settings.Server, self )
    self:AutoAddSettingsTable( Pointshop2.GetModule( "TTT Integration" ).Settings.Shared, self )
end

function PANEL:DoSave( )
    Pointshop2View:getInstance( ):saveSettings( self.mod, "Shared", self.settings )
end

derma.DefineControl( "DTerrortownConfigurator", "", PANEL, "DSettingsEditor" )
```

This example adds all, shared and server settings to the configurator and sends them to the server on save. This is all that is needed to create modifiable, synchronized settings that are saved to the database and can be changed using an ingame editor.

5.8.4 Accessing the Settings

To use the settings in your script simply use *Pointshop2.GetSetting()*.

5.9 Adding custom Tabs

It is possible to add new tabs to various sections of the shop. You can add a tab to the top navigation by using *Pointshop2:AddTab()*. Inside of the inventory tab you can add pages to the side navigation by using *Pointshop2:AddInventoryPanel()*. It is also possible to add new pages to the side nav of the management tab by using *Pointshop2:AddManagementPanel()*.

CHAPTER 6

Getting and Setting player points

In Pointshop 2 points are stored in a player's wallet (*Wallet*). A wallet is a table that contains points and premiumPoints.

Wallets are only networked to their owner and to all admins by default. If you want to display points on a scoreboard, you need to tick the setting "Broadcast Wallets" in the General Settings. This will send everyone's wallet to everyone.

Use ply.PS2_Wallet to access a player's wallet. To manipulate a wallet use

PLAYER:PS2_AddPremiumPoints()

PLAYER:PS2_AddStandardPoints()

There is also a console command available to give points to a specific steamid. You can use this with your donation system:

ps2_addpoints <steamId> <currencyType> <points>
Gives points to a specific SteamID. Works even if the player has never joined the server.

steamId: SteamID of the player in the STEAM_x_x:xxxxxxxxxx format

currencyType: Currency to give. Can be points or premiumPoints

points: Amount of points to give

CHAPTER 7

Creating Pointshop Skins

Pointshop 2 skins are implemented as derma skins. This means that the Paint functions of Panels are passed to a skin. If you are creating a skin you cannot overwrite or modify any of the files but have to do your skinning entirely through a skin.

The name of the skin which is used is set within the main configuration file at *lua/ps2/shared/sh_config.lua*. The default flatui skin can be found in *lua/ps2/client/cl_dermaskin_flatui.lua*. The inventory is seperately skinned through the skin configured in *lua/kinv/shared/sh_config.lua*, the default skin can be found in *lua/kinv/client/cl_dermaskin.lua*

7.1 Types of Hooks

There are two types of hooks that you can use to customize looks and behaviour:

1. **Layout Hooks**: These are always called after initialization of the component. You can use this hook to replace components with custom components or to reposition components.
2. **Paint Hooks**: These work like the normal panel:Paint hooks, use this to customize the appearance of the different panels.

If you are missing a hook on a component that you want to customize, please send Kamshak a pm on coderhire.

7.2 Fonts

Fonts are defined as skin properties and can be customized through a custom skin as well.

CHAPTER 8

Developer API Reference

Use this to look up classes or methods.

Note: Not every class is documented here. Look at the source code for usage examples. This is intended to explain data structures that might not be obvious at first as well as documenting the public interface.

8.1 Modules

class `Blueprint`
A blueprint defines all properties needed for a custom item.

class `SettingsCategory`
A table representing a category that contains one or more settings. The category's path is indicated by it's key in the table. The `info` key is used to provide a readable name for the category to be used in the configurator component. It contains multiple Settings, the path is again provided by the key.

class `MODULE`
The module information table. Contains information such as a module's blueprints and settings. This is to be defined in the module's *sh_module.lua*

`Name`
The name of the module.

`Author`
The author of the module

`RestrictGamemodes`
A table containing gamemodes that this module is restricted to. The module will only be loaded if the active gamemode is in the list.

`Blueprints`
A table containing *`Blueprint`* s that players can use.

`Settings`
A table containing the settings of the module. A seperate table for each realm exists. `Settings.Server` is only sent to admins when changing the settings, `Settings.Shared` is synced with all clients. Accessing Settings is done by calling *`Pointshop2.GetSetting()`* . Each realm settings table defines Settings that can be accessed and configured easily and can contain multiple `SettingsCategory` s. Settings registered this way are automatically saved the the database when changed.

SettingsButton
A table containing information about the settings button that will show up in the management tab. The table has three elements:

`label`: The label of the button `icon`: The icon of the button `control`: The derma control that is created when clicking the button

`Poinsthop2.`**`RegisterModule`**(*MODULE*)
Registers a *MODULE* to be used with Pointshop 2.

`Pointshop2.`**`GetSetting`**(*moduleName*, *path*)
Retrieves a setting value that was defined in MODULE.Settings. Automatically uses the default or database saved value.

- **modName**: The MODULE.Name of the module where the setting is defined
- **path**: The category and name of the setting, seperated by a "."

Example:

```
print(Pointshop2.GetSetting("TTTIntegration", "RoundWin.Innocent"))
```

`Pointshop2.`**`AddEquipmentSlot`**(*name*, *itemValidFunction*)
Registers a new equipment slot.

Name:label of the slot that is shown underneath the slot's panel in the inventory. **itemValidFunction**: A function that takes an item as an argument and returns whether or not it can be equipped in the slot.

`Pointshop2:`**`AddTab`**(*label*, *controlName*, *shouldShow*)
Adds a new tab to the top navigation of the pointshop.

- **label**: The label of the tab.
- **controlName**: The derma control that is created as panel.
- **shouldShow**: **optional** A function returning whether or not the player should be able to see this tab.

`Pointshop2:`**`AddManagementPanel`**(*label*, *icon*, *controlName*, *shouldShow*)
Adds a new tab to the side navigation of the management panel.

- **label**: The label of the tab
- **icon**: The tab's icon
- **controlName**: The derma control that is created as panel
- **shouldShow**: **optional** A function returning whether or not the player should be able to see this tab

Example:

```
derma.DefineControl( "DPointshopManagementTab_Settings", "", PANEL, "DPanel" )

Pointshop2:AddManagementPanel( "Settings", "pointshop2/advanced.png", "DPointshopManagementTab_S
    return PermissionInterface.query( LocalPlayer(), "pointshop2 managemodules" )
end )
```

`Pointshop2:`**`AddInventoryPanel`**(*label*, *icon*, *controlName*, *shouldShow*)
Adds a new tab to the side navigation of the management panel.

- **label**: The label of the tab
- **icon**: The tab's icon
- **controlName**: The derma control that is created as panel
- **shouldShow**: **optional** A function returning whether or not the player should be able to see this tab

class `Configurator`
Interface for Configurator controls. Used by the settings section.

`SetModule`(*module*)
Passes the *MODULE* table to the control.

`SetData`(*data*)
Passes the Settings as retrieved from the server to the control. data contains Server and Shared settings merged together.

8.2 Player integration

class `Wallet`
In Pointshop 2 points are stored in a player's **wallet**. A wallet is a LibK model instance, you can use :save() on it after changing it.

`points`
The standard points of a player.

`premiumPoints`
The premium (donator) points of a player

PLAYER`:PS2_AddStandardPoints`(*points*, *message*, *small*)
Awards points to a player. If a message is specified it will show up in their pointfeed.

points: Amount of points given **message**: Message displayed to the pointfeed **small**: Use the small print (for bonus points related to a primary event)

PLAYER`:PS2_AddPremiumPoints`(*points*)
Adds premium points to a player's wallet.

points: Amount of points given

B

C

I

M

P

S

W