

TweeFly 0.13 Documentation

stonedrum.de, 2020

# Introduction

*TweeFly* is an UI based setup tool for interactive stories written with *Twine* and *Twee2*. The idea behind it is to have a tool that generates all the required Twine, CSS and JavaScript files to handles various tasks like creating an inventory, a shop system, or a daytime cycle.

## Editions

There exist two different editions: Free and Professional. The *Professional Edition* includes features that the *Free Edition* does not (surprise). See the following table for all the feature’s differences:

Table 1: Feature comparison of Free and Professional Edition

|  |  |  |
| --- | --- | --- |
| Feature | Free version | Professional version |
| Inventory | X | X |
| Shops | X | X |
| Money | X | X |
| Clothing |  | X |
| Stats |  | X |
| Daytime |  | X |
| Jobs |  | X |
| Characters |  | X |
| Custom captions |  | X |
| Story configuration | X | X |

Both editions can be purchased and/or downloaded from [stonedrum.de](http://www.stonedrum.de).

# Quick start

*TweeFly* is pretty easy to setup. Install the application and start it. You see a *menu* on the left and some *tabs* on the right. First, check the features you want to use in your game (e.g. *inventory*, *shops*, *money* in the *Free Edition*). Some features depend on others and are selected automatically (if you want to use a *shop system* you need to activate the *money* feature as well. It is selected by *TweeFly* if it has not been checked yet).

Now add your items or shops in the corresponding tabs and select *Display link in sidebar* on the *inventory* tab so that you can access your inventory any time in the game. Once you did that, give your story a name in the edit field in the *menu* on the left and name your main story file that has to include the *::Start* paragraph (or the respective first paragraph in *Twine*) which is the entrance point to your story.

There are three options to generate your story:

1. **Generate (Twine)**: Click this button to select a *Twine* story HTML file. *TweeFly* will integrate the code, passages and css definitions into the HTML file. Open the file in Twine and you can start working with TweeFly’s macros.

With version 0.12 TweeFly left the beta state and can be used as is. Nevertheless, there should be a backup for all Twine stories that are generated.

1. **Generate (Twee2)**: Click this button to select a *Twee2* (\*.tw2) file. *TweeFly* will create all the corresponding script and css files around this file. Furthermore, it creates a bat file that you can execute to build your story.
2. **Generate and Build (Twee2)**: Does the same as 2) but builds the story.
3. **NEW Generate Cheat Sheet (HTML)** -P-: Generates an html document that contains all macros provided by TweeFly. This sheet will help you to get started faster and always know what you can do using the framework (BETA).

# TweeFly in-depth guide

In this section we will go through all functions of *TweeFly* and learn how to use them. There are various combinations you can use so this guide will create a scenario where we use them all. From time to time there are differences between the Professional and the Free Edition. Features only included in the Professional Edition are marked with a -P-.

## The menu

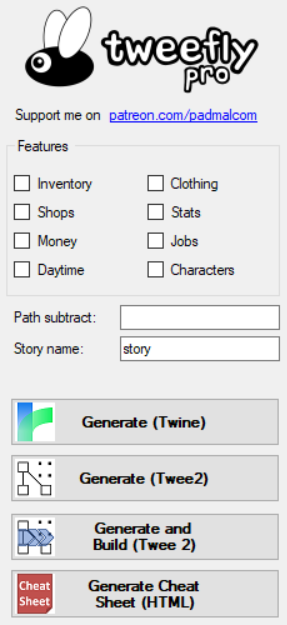
The menu is found on the left of the main window and includes all features that can be selected:

Figure 1: The TweeFly menu

* Inventory
* -P Clothing
* -P Stats
* -P Daytime
* Shops
* Money
* -P Jobs
* -P Characters

As mentioned at the beginning some features require other features. Those are selected automatically by *TweeFly*.

The field **Path subtract** is used when you specify images e.g. to display an item. Most certainly, the URL of an image might differ on the developer's and the player’s computer. As a consequence, using relative paths is the way you should go. When picking an image for an item the dialog adds the entire file path as property. To make this path a relative path, you can leave it as it is and specify a string to subtract from the beginning of the image’s path. An example:

You select an image from:

*C:\Users\developer\mystory\img\chocolate.png*

And specify a subtraction path:

*C:\Users\developer\mystory\*

Then the path referenced in the code generated is:

*img\chocolate.png*

Thus, you can place all your images when packaging your story in a folder *img* and its path is valid whoever plays your story.

We included a place holder called **%APP\_DIR%** which can be used in the path subtract and each field containing an image path. This place holder will always be replaced during design time (not in your game!) by the path of *TweeFly*.

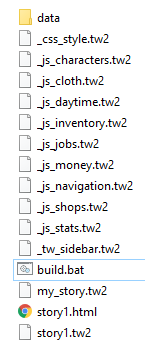
**Story name** specifies the name of the story as it is shown in the sidebar in your game and sets the name of the html file generated.

Figure 2: Folder structure of a TweeFly project

**Main file** is the *Twee2* file (\*.tw2) that contains your story. This main file should contain a paragraph *::Start*. I had to design the program like that to make sure that your story is not overwritten any time you regenerate the *TweeFly* code. So, your main file remains untouched by *TweeFly* but every file around the main file is created once you click *Generate*. If the main file does not exist in the folder where you generate your *TweeFly* code the process generates a valid file for you so that the *Twee2* compiler does not run into errors.

A classic folder structure for a project should look like the image on the left (the folder *data* is added manually and recommended to contain audio and image files).

You can generate your code in several ways. You either click **Generate** to have *TweeFly* only create all the files required for the build. **Generate and build** creates the files and executes *Twee2* to create the html file, too. If you select **Run after generation**, *TweeFly* executes the html file.

[BETA] **Create dummy conf** is a button to fill *TweeFly* with some test values so that you (and I) can quickly test the generation process. It will most certainly disappear from the tool, soon.

## General settings

There are some general settings that occur on nearly every tab and hence need to be explained only once. In the top of the tabs there are checkboxes with the caption **Display (…) link in sidebar**. If this checkbox is checked, *TweeFly* generates a link in the sidebar that can be clicked at any time in the game and opens a menu in the paragraph section, e.g. for the inventory, cloths, or stats.

Another option to show information on those menus is to render them directly in the sidebar. This can be achieved by checking **Display (…) in sidebar**. When rendering menus as such, there is not as much space as in the paragraph section. Thus, the information contained in sidebar menus are less.

If you want to give your player more information in the sidebar menus, check **… sidebar tooltip**. If the player hovers over the icon in the sidebar menu, a popup shows and lists all information you want to give the player (depending on the check list box e.g. *Display in inventory*).

Items, cloths, shops, and characters offer the opportunity to attach three blank skills (**Skill 1 – Skill 3**) so that you can integrate some own logic via *TweeFly*. To activate those skills, just check the boxes and give the skills a name in the *Captions tab*.

As already mentioned, you can take influence on what the player sees in menus or tooltips by checking or unchecking properties in the **Display in …** check list boxes. If you select e.g. *Shop category* in the Display in inventory list, the *shop category* is shown in the inventory table and in the tooltip in the sidebar.

When it comes to editing items, cloths, stats, shops, jobs, or characters each object can be modified in a group box in the bottom of each tab. This box contains all **properties** of each object and gives you hints about the data type, e.g. if the property is a string, a number, or a color. To add a new object, fill each field and click **Add new**. The new item then appears in the list above. To load an item from the list into the group box, double click on the item in the list. If you want to update an item, select an item in the list, change its values and click **Update**. To delete an object, select it and click **Delete**.

## Inventory tab

The *inventory tab* is there to define items and the look of your inventory. Internally, all items defined (no matter if the player owns the item or not) can be accessed via the list in the JavaScript list *state.active.variables.items*. The items actually owned by the user are stored in *state.active.variables.inventory*.

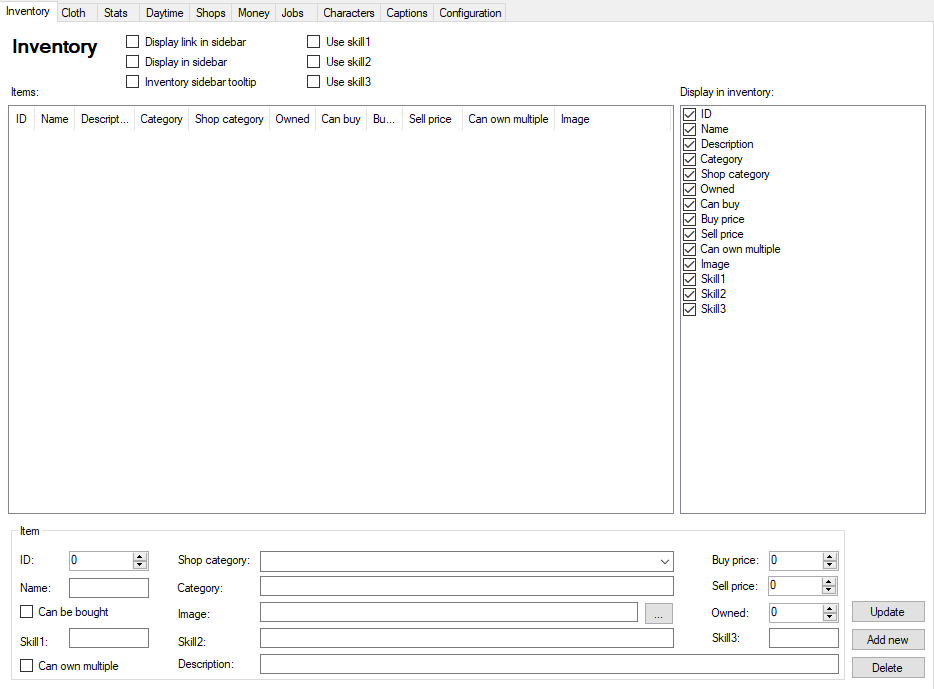


Figure 3:The inventory tab

The list **Items** contains any item specified by you. Let us look at the individual properties:

* *ID*: Each item has a unique ID in form of an integer number.
* *Name*: Name of the item
* *Category*: A free category
* *Shop category*: A free shop category
* *Owned*: How many of these items owns the player at the beginning?
* *Can buy*: Can this item be bought in a shop?
* *Buy price*: The price of the item
* *Sell price*: The money you get for selling this item
* *Can own multiple*: Can the player own multiple of these items?
* *Image*: An image file shown for this item.
* *Skill 1-3*: Three blank skills

TweeFly generates several Macros for you to manipulate the inventory.

Table 2: Inventory macros

|  |  |
| --- | --- |
| Name | Description |
| initItems | Initializes your items (no need to call) |
| initInventory | Initializes your inventory (no need to call) |
| addToInventory | Adds an item to the inventory. Requires two parameters: Item ID and the amount to be added. |
| removeFromInventory | Removes one or multiple items from your inventory.   * If one parameter is specified: All items with the specified ID (parameter 1) are removed. * If two parameters are specified: The amount n (parameter 2) of the item with a specific ID (parameter 1) are removed. |
| inventory | Renders an inventory table for the paragraph section. All properties marked in the *Display in inventory* list are visible. |
| inventorySidebar | Renders a two-column inventory table for the sidebar with a tooltip giving information on each item when the corresponding checkbox is checked. |

A sample configuration with the inventory link in the sidebar, the inventory itself in the sidebar, four items and five selected properties to display in the table will look like the picture below.

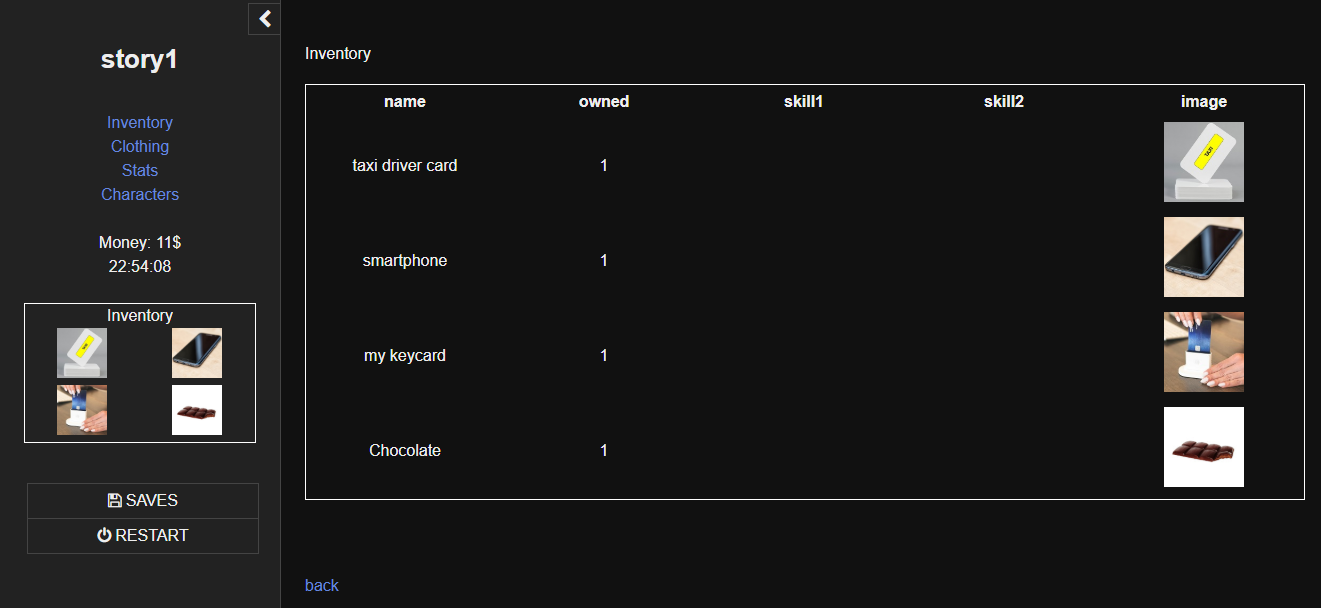


Figure 4: Inventory rendered in sidebar and in paragraph section.

## Clothing tab

The clothing tab is more complex than the inventory since it makes use of three lists:

1. A list for all clothing in the game (*state.active.variables.allClothing*)
2. A list for all cloth owned by the player (*state.active.variables.wardrobe*)
3. And a dictionary for each cloth worn at a specific place (*state.active.variables.wearing*)

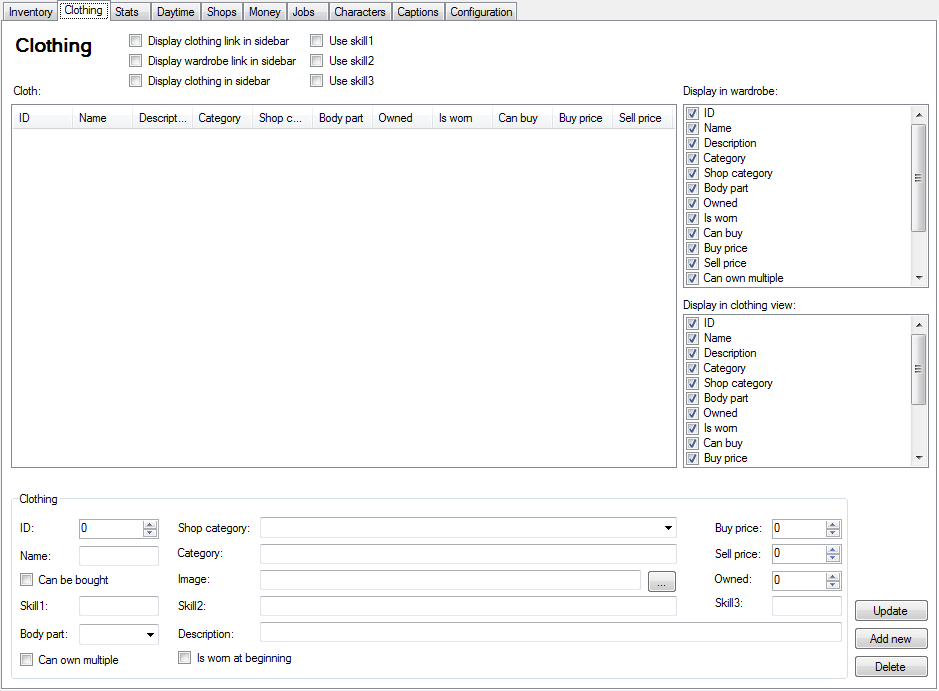


Figure 5: Clothing tab

Clothing has the following properties:

* *ID*: Unique number.
* *Name*: The name of the item.
* *Description*: The item description.
* *Category*: A clothing category.
* *Shop category*: A shop category
* *Body part*: Where this clothing is worn (see the drop down list in the clothing box for possible values).
* *Owned*: How many clothing of this type has the player at the beginning.
* *Is worn*: Is this clothing worn at game start?
* *Can buy*: Can this clothing been bought in shops?
* *Buy price*: How much is this clothing in shops?
* *Sell price*: The money you get for selling this item.
* *Can own multiple*: Can the player own more than one clothing of this type?
* *Image*: An image to display this clothing.
* *Skill1-3*: Three blank skills

The macros used in the clothing system are:

Table 3: Clothing macros

|  |  |
| --- | --- |
| Name | Description |
| initAllClothing | Initializes all clothing in the game. |
| initClothing | Initializes all clothing the player is wearing. |
| initWardrobe | Initializes all clothing in the player's wardrobe. |
| clothing | Prints a table with the clothing the player wears. |
| clothingSidebar | Creates a table of images in the sidebar with the clothing worn by the player. |
| wardrobe | Prints a table with all the player's clothing. |
| addToWardrobe | Adds clothing and expects two parameters: The clothing ID (parameter 1) and the amount to add (parameter 2). |

A sample for displaying the currently worn cloth can be seen here. In the sidebar there are the images of the currently worn cloths and in the paragraph, there is a customized table with name, body part, owned number, and image displayed.

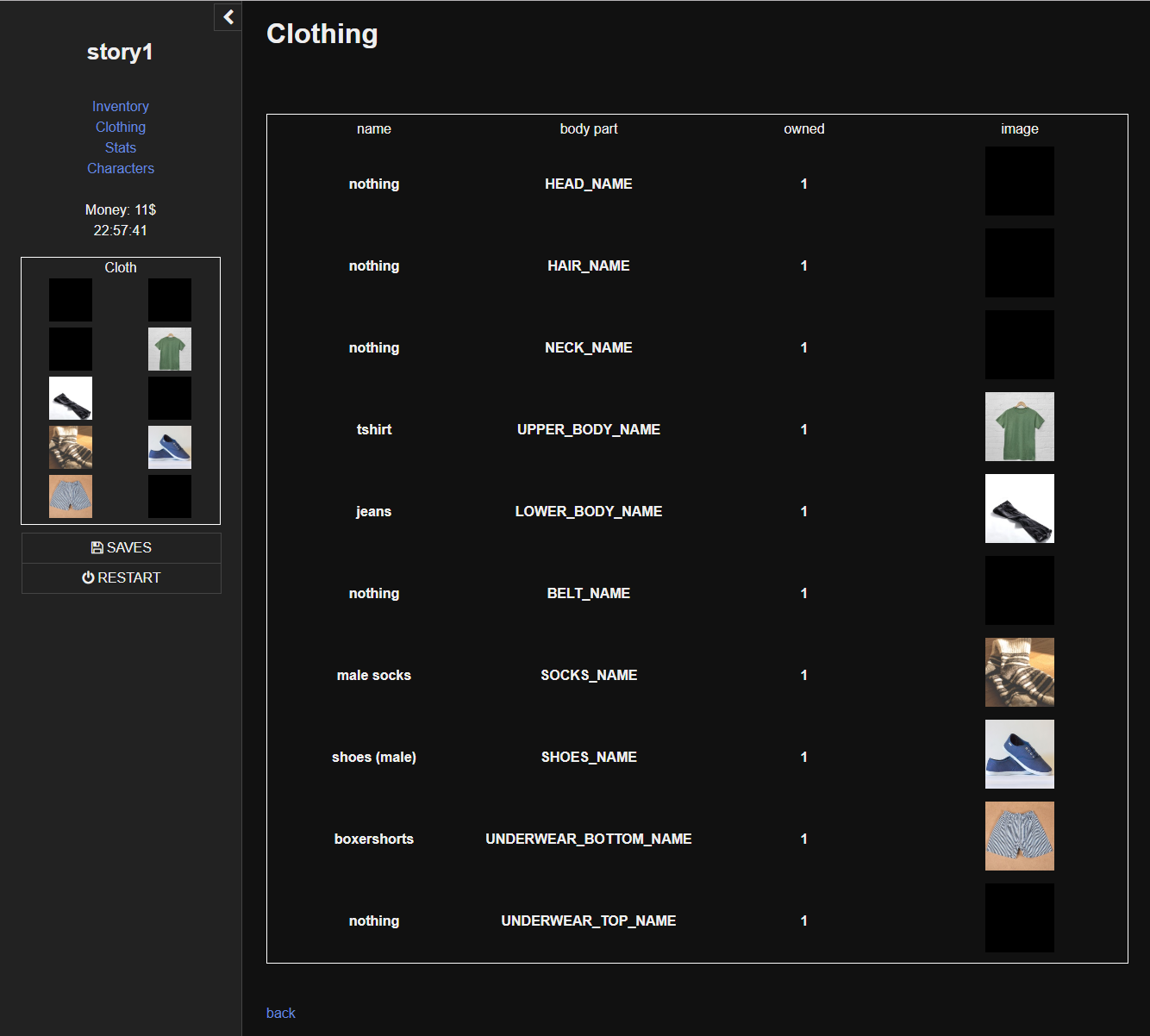


Figure 6: Cloths rendered in sidebar and paragraph section.

## Stats tab

The stats list is managed in one single list (*state.active.variables.stats*). A link to the stats menu as well as a stats list can be seen in the sidebar.

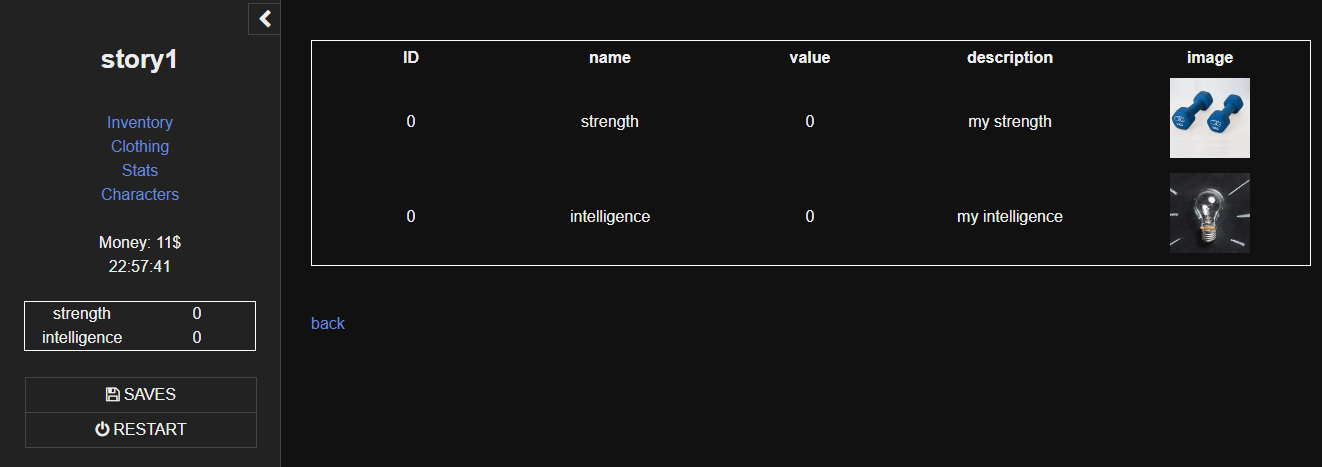


Figure 7: Stats strength and intelligence are shown in sidebar and in paragraph section.

Each stat can have the following properties:

* *ID*: A unique identifier for each stat
* *Name*: The stat’s name
* *Value*: An initial value
* *Unit*: A unit if needed
* *Image*: The image that characterizes this stat
* *Description*: A description

The macros used in the stats system are:

Table 4: Stats macros

|  |  |
| --- | --- |
| Name | Description |
| initStats | Initializes all stats. |
| setStats | Change a stat’s value. The first parameter is the stat ID, the second parameter is the value. |
| getStats | Returns a stat as integer by a given ID (first parameter). |
| addStats | Adds a specific value (parameter 2) to a given stat (ID in parameter 1). |
| stats | Renders a stats table in the paragraph section. |
| statsSidebar | Renders the stats list in the sidebar (just name and value) |

## Daytime

The daytime tab is somewhat different since it does not contain individual objects. The daytime can be shown in the sidebar. The format *TweeFly* uses can be selected in the **Display time format** box. The start time is the time *TweeFly* sets when your game starts.

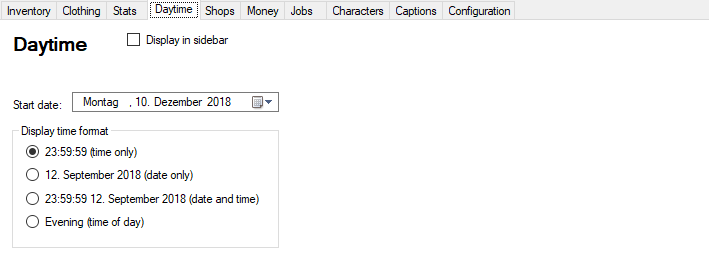


Figure 8: Setting daytime and daytime format.

The macros used in the daytime system are:

|  |  |
| --- | --- |
| Name | Description |
| initDaytime | Initializes the daytime system. |
| getTime | Returns time in format 23:59:59. |
| getDate | Returns date in format 01 August 2018. |
| getDateTime | Returns time and date in format 01 August 2018 23:59:59. |
| getTimeOfDay | Returns Early morning (1-4), Dawn (4-6), Morning (6-11), Noon (11-13), Afternoon (13-16), Evening (16-21), Night (21-24), or Mid-Night (0-1). |
| setTime | Sets the time and expects three parameters: hours, minutes, seconds (parameter 1 to 3). |
| setDate | Sets the date and expects three parameters: year, month, day (parameter 1 to 3). |
| setDateTime | Sets date and time and expects six parameters: year, month, day, hours, minutes and seconds (parameters 1 to 6). |
| addTimeInMinutes | Adds minutes (parameter 1) to current time. |
| addTimeInDays | Adds day (parameter 1) to current date. |

## Shops

Shops are stored in a list (*state.active.variables.shops*) with all their individual properties and a sub-list of items and cloths that are sold in the shop. This items and cloths have to be defined in the *inventory tab* and the *clothing tab* before.

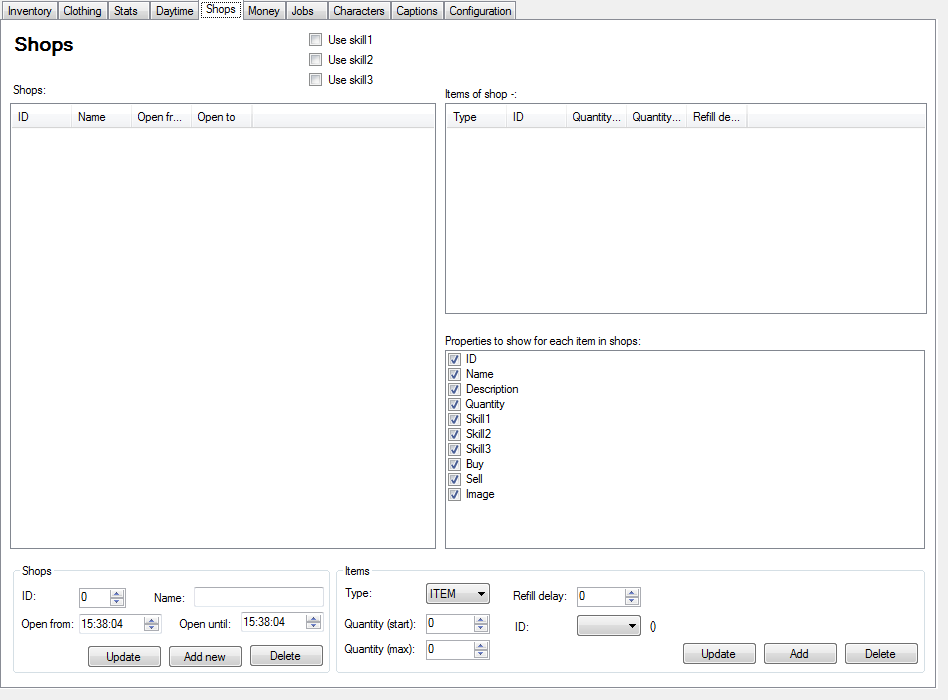


Figure 9: The shop tab

Each shop has the following properties:

* *ID*: A unique Identifier
* *Name*: A name
* *Open from*: When does the shop open?
* *Open until*: When does the shop close?

If a shop should continuously be open set *open from* to 00:00:00 and *open until* to 23:59:59.

The item a shop contains can be seen in the top right list view and contain the following fields:

* *Type*: CLOTHING or ITEM
* *ID*: The Identifier of either item or clothing.
* *Quantity (start)*: How many items of that type are sold at the beginning of the game.
* *Quantity (max)*: How many items may the shop have as a maximum.
* *Refill delay*: How many minutes does it take to refill an item of that kind?

The macros used in the shop system are:

|  |  |
| --- | --- |
| Name | Description |
| initShops | Initializes all shops and their items. |
| shop | Displays a shop with a given ID (parameter 1) in the paragraph section. |

Table 5: Macros of the shop system.

The following sample shop offers to sell and buy three items. If an item cannot be afforded, its link is not acive.



Figure 10: A sample shop.

### Money

The money system uses two variables: *state.active.variables.money* and *state.active.variables.moneyPerDay*.

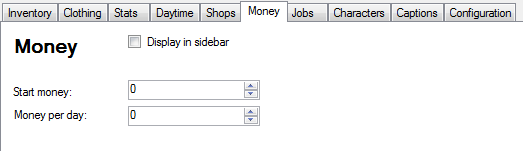


Figure 11: The money tab

Start money is the amount of money a player has at the beginning and money per day -P is an amount the player gets when a day has passed. The money displayed in sidebar can be seen in the figure below. The Unit (here $) can be changed in the *captions tag*.

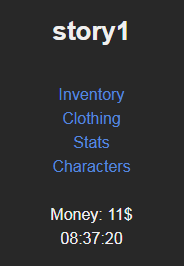


Figure 12: Money in sidebar.

The macros used in the money system are:

|  |  |
| --- | --- |
| Name | Description |
| initMoney | Initializes the money system. |
| printMoney | Outputs the money without the unit. |

Table 6: Macros for the money system

## Jobs

The jobs are stored in a list (*state.active.variables.jobs*) and each job contains next to its properties a list of reward items that are given to the player when the job is done.

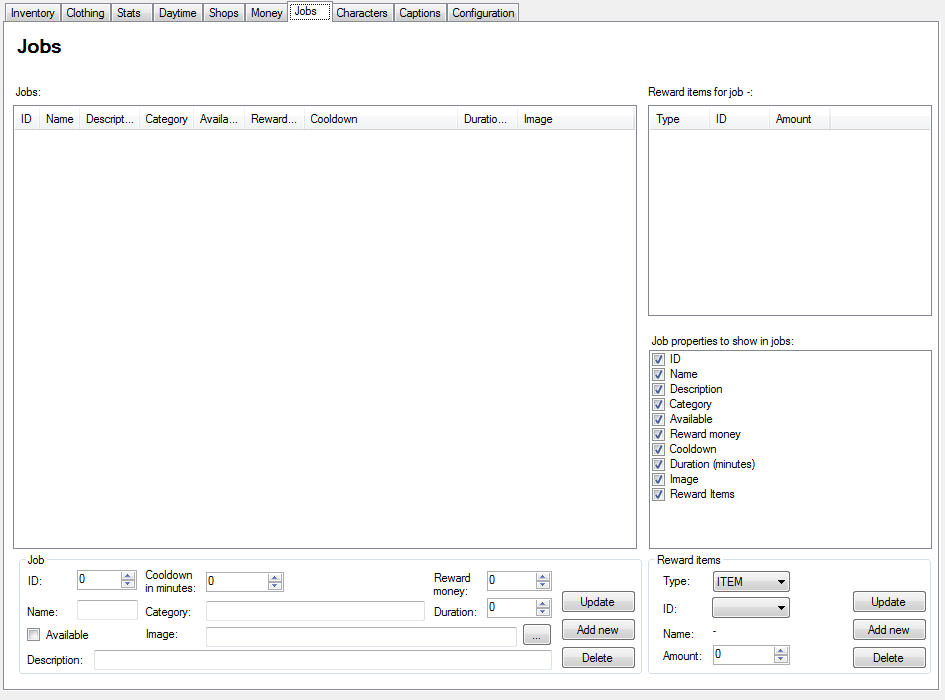


Figure 13: The jobs tab.

Each job has the following properties:

* *ID*: A unique identifier
* *Name*: A name
* *Description*: A job description
* *Category*: A category
* *Available*: An information if the job is available
* *Reward money*: The money the player gets for doing the job
* *Cooldown*: Minutes to wait until the job is available again.
* *Duration*: Minutes it takes to do the job.
* *Image*: An image for the job.

Each reward item has the following properties:

* *Type*: CLOTHING or ITEM
* *ID*: The identifier of the item or clothing referenced.
* *Amount*: The amount of the item the player gets for doing the job.

The macros used in the job system are:

|  |  |
| --- | --- |
| Name | Description |
| initJobs | Initializes all jobs. |
| showJobs | Shows a number of jobs. Has an arbitrary parameter count. Each parameter is a job ID that is displayed in the job list shown. |

Table 7: Macros for the job system.

## Characters

The characters are stored in the list *state.active.variables.characters*.

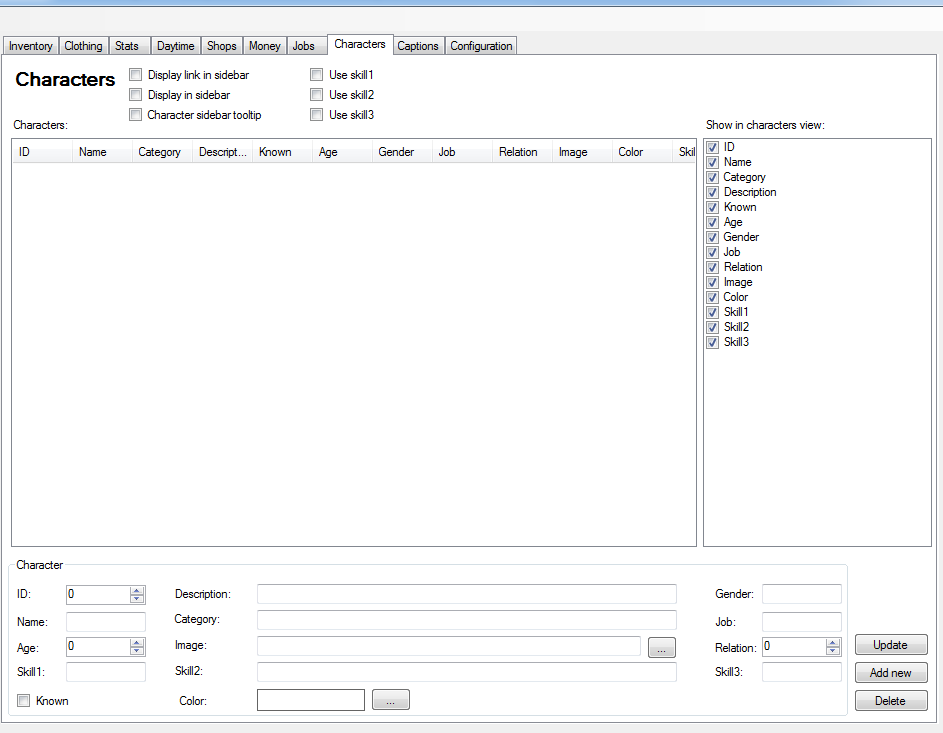


Figure 14: The characters tab.

Each character has the following properties:

* *ID*: A unique identifier.
* *Name*: A name
* *Category*: A category
* *Description*: A description
* *Known*: An information if the character is known. Only known characters appear in the character list.
* *Age*: An age
* *Gender*: A gender field (free text)
* *Job*: A job (free text, not depending on the jobs tab)
* *Relation*: A number to describe the relation to the player.
* *Image*: An image
* *Color*: A color that displays the name of the character in a dialog.
* *Skill1-3*: A blank skill.

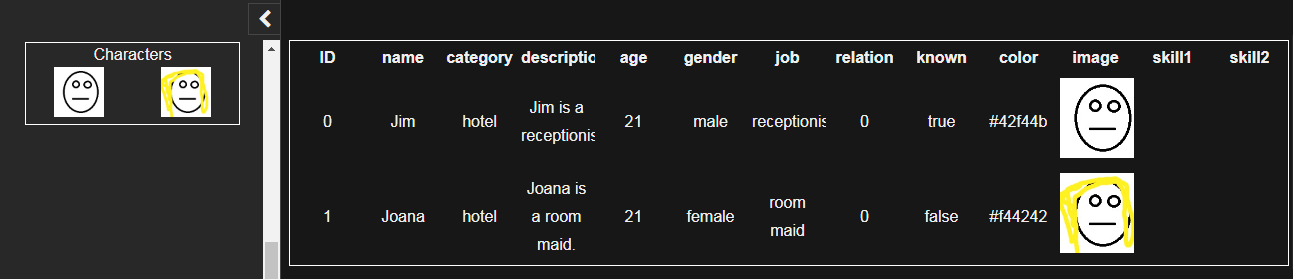


Figure 15: Characters in the paragraph and the sidebar.

The characters system furthermore has a dialog system that can display simple dialogs with the image of the character, his/her name and a text.

|  |  |
| --- | --- |
| Name | Description |
| initCharacters | Initializes the characters. |
| characters | Shows the characters in the paragraph section. |
| charactersSidebar | Shows the characters in the sidebar. |
| setKnown | Sets the character with a specific ID (parameter 1) to the state known. |
| renameCharacter | Sets the name of a character with a specific ID (parameter 1) to a new name (parameter 2). |
| say | Prints a simple dialog box. Requires a character ID (parameter 1) and a text (parameter 2) as input. |

Table 8: Macros of the character system.

## Captions

The *captions tab* allows you to specify individual texts within your story. Instead of changing captions every time in the generated macros, set them here. An example would be the header text of the character table, the currency in the money system or the month names in the daytime system. Using the *captions tab*, you can easily release multi language games (okay, almost, at least you have to translate the story...). To edit a caption, simply double click the caption in the list and type a new value.

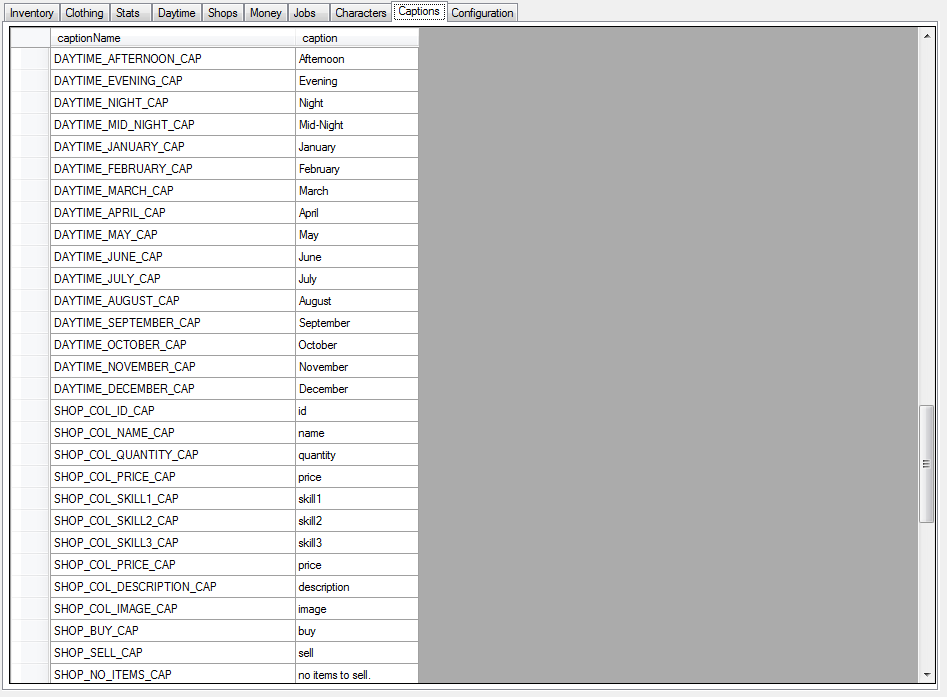


Figure 16: The captions tab.

## Configurations

The configurations tab contains some simple flags and properties for SugarCube which are mostly self explanatory. The changes affect the navigations and the css style file.

NEW The list *Files to include* is a list of additional files (\*.tw2) that need to be included in the final game when compiling. This is mostly used when your story consists of more files next to your main file (e.g. for side quests).

NEW The check box *Show all variables* adds a menu to the game that displays all variables with their states that have been used so far. This is helpful for debug purposes.

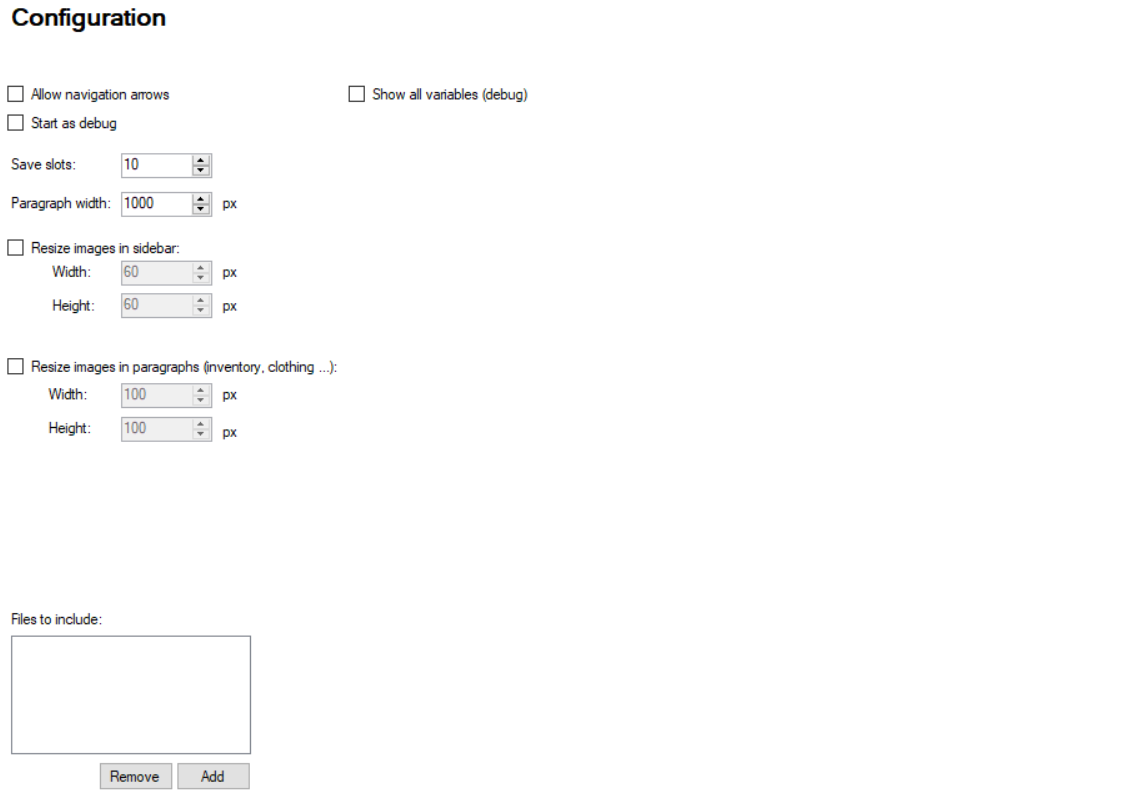


Figure 17: The configurations tab.

# Saving and loading

Each configuration can be saved and loaded easily using the file menu. We offer two file formats. A binary file format *tfc* hides your configuration and does not allow editing without *TweeFly*. The XML format *tfcx* can be edited in an editor -P .

You do not need to ship your *TweeFly* configuration with your game.

# Change Log

This chapter tracks changes in *TweeFly*.

## Release 0.8 BETA

* Initial release.

## Release 0.9 BETA

* Added Twine integration.

## Release 0.9.1 BETA

* Marked auto generated code in Twine files so that they are replaced when TweeFly code is regenerated.
* Added “noreturn” tag to inventory, wardrobe, stats, cloths, characters paragraphs if they did not exist before.

## Release 0.10 BETA

* Fixed a bug in which shops are not updated when items get deleted or the id changed.
* Added Twine support for version 2.3.2
* Added Twine example story (pro only)

## Release 0.12

* added function is\_naked, fixed bug in is\_worn
* added option to set steps visible/invisible
* added option to include further story files
* fixed displaying of jobs
* added undress function
* added macro addMoney
* added function getDaysPassed
* fixed time in addTimeInDays
* fixed bug in clothing

## Release 0.13

* added the generation of a cheat sheet
* added option to include multiple story files (in configuration view)
* added option to display all variables used in game (in configuration view)
* fixed bug with double currency symbols