Socket Sender GUI
The Manual

July 18, 2017

Contents

1 Introduction

This tool, SocketSenderGUI, is a simple message sender that *simulates* messages an app can send to a web renderer. This way, a developer working on the graphics for a customer can avoid actually using the larger app, maybe on a virtual machine, restarting it on every problem, re-playing the scenario that brings to a given screen again and again, modifying other server responses to simulate error messages, etc.

Messages and Sequences

The principal block of data exchange is the *message*. The message can be used to send all the info for a screen, or just for part of a page. It can be in fact any interaction between the app and the renderer. Sometimes the developer will want a sequence of screens to appear one after the other to simulate the ongoing of some simple process.

For this purpose, this tool makes use of *sequences* that are nothing else than lists of messages eventually separated by pauses. For now, sequences can only be composed of two types of elements: messages of course, and a *wait* function that takes a number of *milliseconds* to wait between a step and the next. A future development could introduce the use of function calls directly as parts of a sequence.

We will refer indifferently to a message or a sequence as an *action*. We will say an action has been *picked* when it is the one chosen to be sent to the renderer.

2 The window: the Message Picker

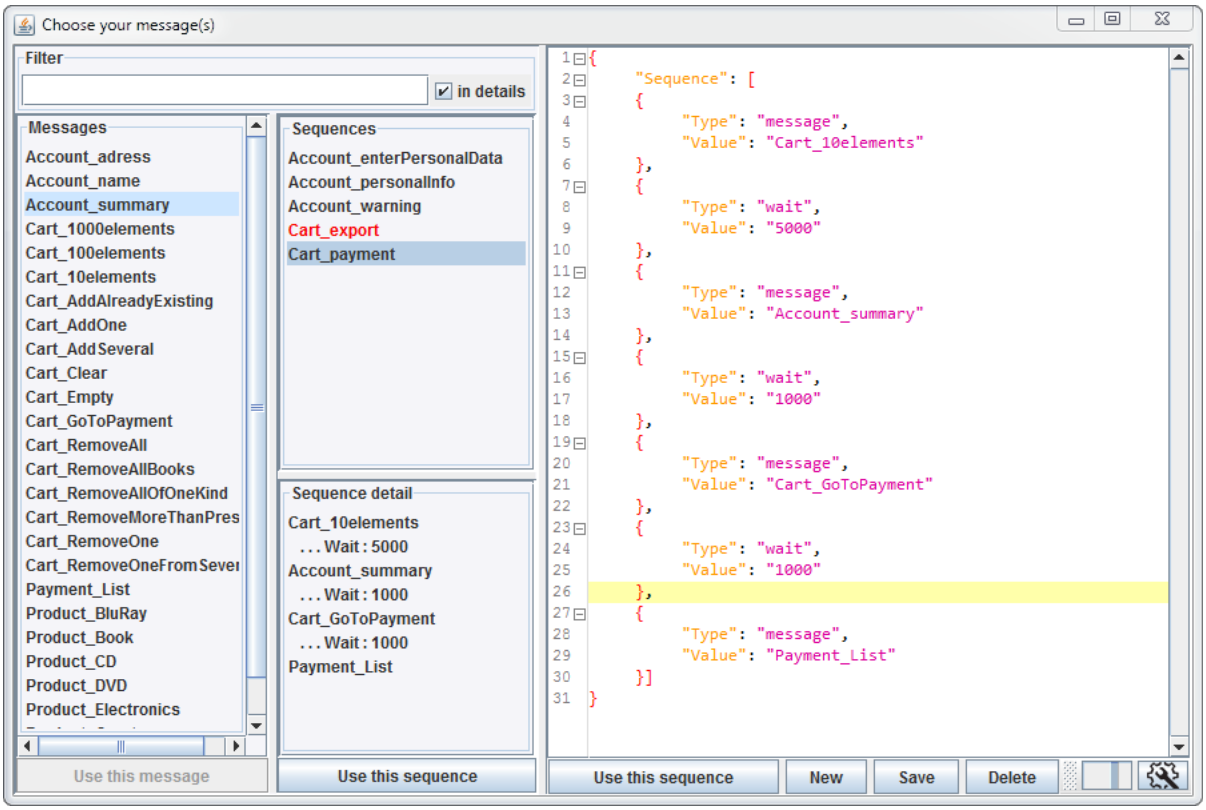


Figure 1: The Main Window

The user interface is quite simple. A single window (Figure 1) allows to manage the collection of messages and sequences and to pick an action from these collections to be sent to the renderer.

The left pane: choose what to send

The left pane is composed of various *lists* and a *filter bar*. The first column lists the saved messages. The second columns, on top, lists the saved sequences. When a sequence is selected, the bottom part of that columns lists the different *steps* of that sequence.

! → Selecting an element in one of the lists does not *pick* it to be used. It just shows its content in the editor (on the right pane, see next section).

Pick a message Clicking the button under a list actually picks the selected element to be sent to the renderer. Notice that buttons are enable or disabled depending on which kind of element is selected.

Current message The *currently used action* has a colored background – light blue –

different from the color of a selected item. When an action is picked, it is saved as current action and automatically loaded on next startup.

Filter — The *filter bar* allows to filter all the lists, removing all messages that do not fit the text. It is case-insensitive and filters both messages and sequences. For more details, see 4.1.

The right pane: the editor and manager

The *editor* is a minimal Json editor (coloring and folding). When an action is selected, its content appears in the editor, this way the developer can check it and modify it.

! → Notice how as soon as the text has been modified, the button underneath changes from "Use this message" to "Use this custom message". This way, the developer is reminded he has modified and not saved the action. A *custom message* or *custom sequence* can be picked and is saved for a next session like any other current message.

Custom action

The buttons under the editor are self-explanatory. For a more detailed overview of the processes of saving, creating or deleting, see respectively 4.3, 4.2 and 4.4.

Status bar — In the mini-status bar on the bottom right, the progress bar shows that the server is *running*: the picked action is sent to the renderer whenever this latter connects to the message server (basically, on every refresh of the page, since we are talking about a browser renderer). The last button on the right opens (and closes) the *config file* in the editor. Its is a Json string itself (see section 3 for details).

3 Configuration

 To open the configuration file, click that button in the the bottom right. It will load the file in a Json editor. To close, click the same

! → button again: it will save automatically the configuration on close.

On first open — A warning message might appear if the configuration is not set yet.

Setting up the configuration means choosing a destination *HTML5 renderer project* and some paths to where to save actions: one for messages, one for sequences, and one for the current action, whatever type it is.

The configuration file

```
{
   "Customer": "SuperStoreApp",
   "Customers" : {
      "SuperStoreApp": {
         "HtmlWebPath" : "C:\\HTMLProjects\\Html5_SuperStore",
         "MessagesPath" : "actions\\messages",
         "SequencesPath" : "actions\\sequences",
         "CurrentActionPath" : "actions\\current"
      },
      "Default": {
         "HtmlWebPath" : "",
         "MessagesPath" : "actions\\messages",
         "SequencesPath" : "actions\\sequences",
         "CurrentActionPath" : "actions\\current"
      }
   }
}
```

In order to be able to work with various projects, SocketSenderGUI can "remember" a *project* per customer. Each project in nothing else than a name and a configuration. This way, each customer can have its own personalized sets of actions.

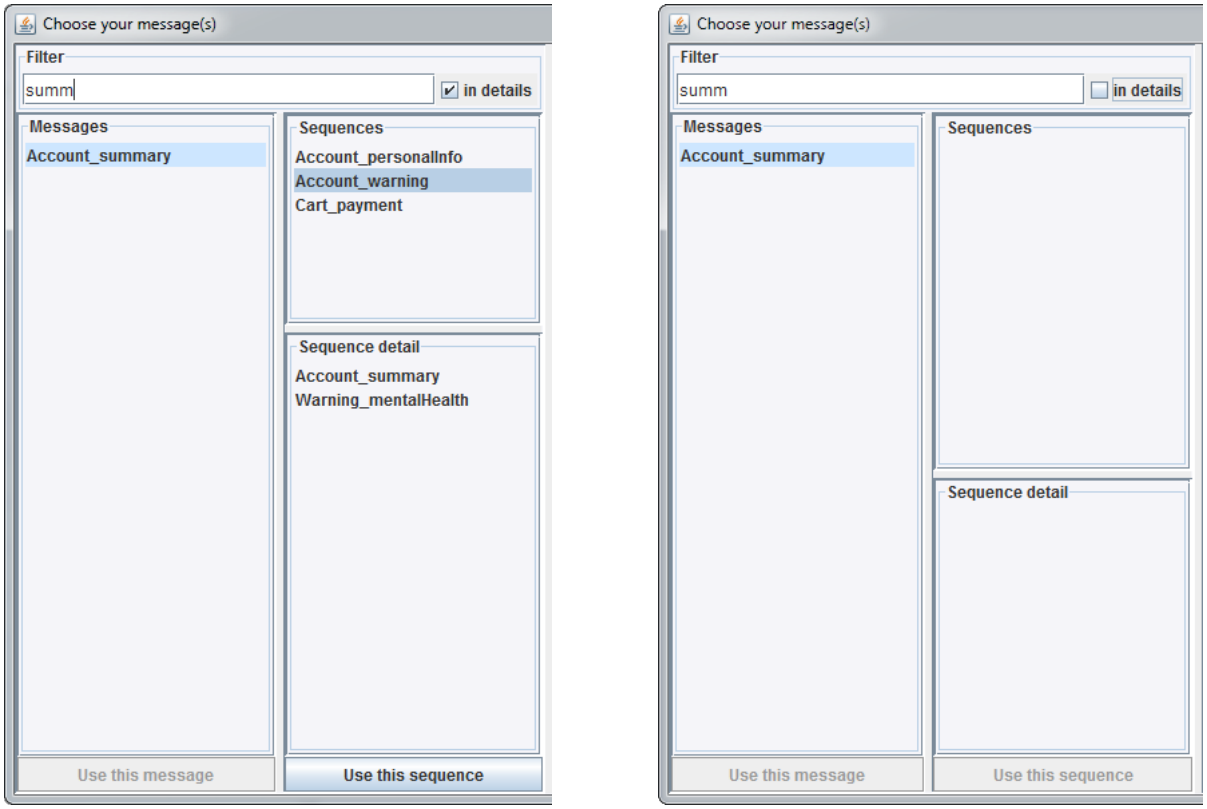
One can just change the value of the `Customer` parameter to one of the `Customers'` name in order to switch from one project to another.

! → All paths can be relative or absolute. Note that the backslashes need to be escaped as the strings are not formatted by the deserializer.

4 Manage messages and sequences

4.1 Filter

The filter is instantaneous: all lists are filtered on every keystroke. The option *in details* lets you choose to keep the sequences which contain a message that corresponds to the filter even if the name of the sequence itself does not.



(a) Keep sequences that contain a filtered message

(b) Only in sequences names

Figure 2: Filter messages and sequences

4.2 Create new messages and sequences

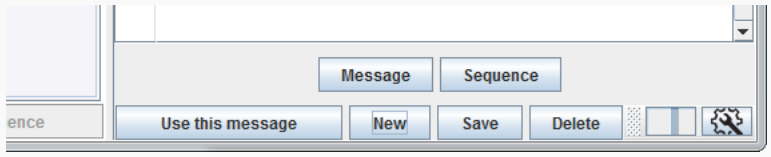


Figure 3: Create new: message or sequence?

Snippets

A single button opens a popup bar to choose the type of the action to create. The editor is filled with the corresponding snippet:

New message

```
{
   "InputElements": [{
      "Text": "",
      "Id": "",
      "Type": ""
   }],
   "OutputElements": [{
      "Text": "",
      "Id": "",
      "Type": ""
   }],
   "Id": "",
   "Language": "ita",
   "Type": "",
   "Number":  151
}
```

New sequence

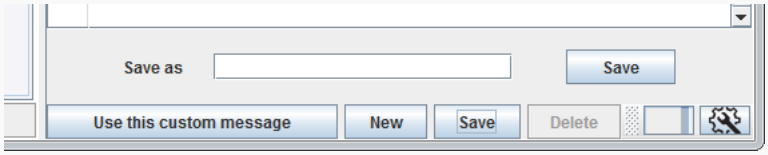
```
{
   "Sequence": [
   {
      "Type": "message",
      "Value": ""
   },
   {
      "Type": "wait",
      "Value": "2000"
   }]
}
```

4.3 Save messages and sequences

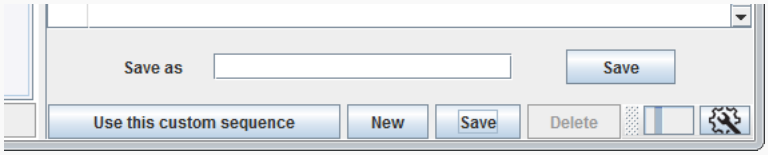
The save button reacts depending on the context. A first element it considers is of course the type of the action. Then it takes into account if the action was created new or is a modification of an existing action (of which it inherits the type – message or sequence).

Save a new action

If it was created from scratch using New , the save button pops-up a bar just asking for a name. The only way to make sure one is saving a message or a sequence is checking that the use button contains the words "custom message" or "custom sequence" (see Figure 4).



(a) Save a new message



(b) Save a new sequences

Figure 4: Save action created from scratch

Modify an existing action

If the action is a modification of an existing action, then the save button open a more complete popup bar offering *two possibilities*:

- Overwrite the original action
- Save it as a new action (of same type)

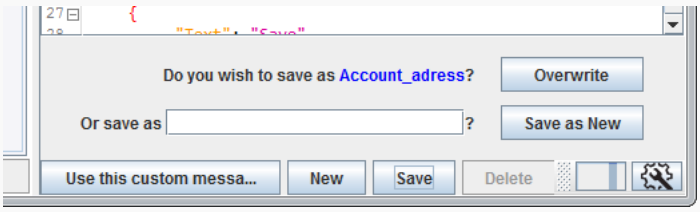


Figure 5: Save a modified message: overwrite or new?

Save as... If an action was not even modified, it is possible to save a copy with a new name.

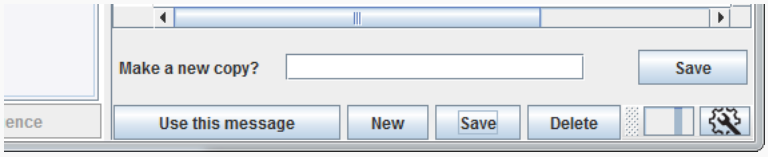


Figure 6: Save an unmodified message: make a copy?

In any case, it is not possible to save an action with the same name as an already existing action of same type. Uniquness of names are case-insensitive[1].

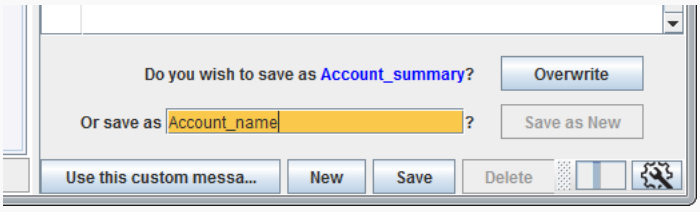


Figure 7: Given name already exists

[1]This is as much a technical limit than a choice to avoid confusion. Technically, actions are saved in separate files `name.json`, and Windows is case-insensitive.

! → No correctness check is made on the grammar of the action: one can save a sequence in a message and vice-versa. Only the Json correctness is checked before saving.

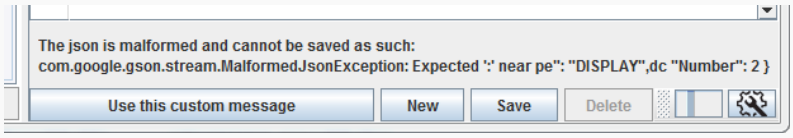


Figure 8: Json is malformed

4.4 Delete messages and sequences

Deleting is quite straightforward with a simple popup asking for confirmation:

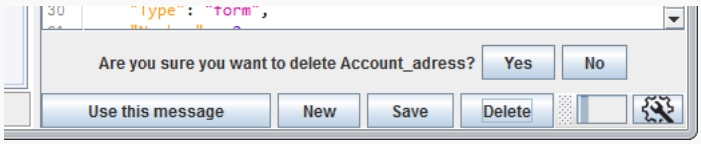


Figure 9: Delete a message or sequence

5 Known Issues

Save When a `message` is selected and New is pressed for the creation of a new `message`, the selected `message` is not unselected. Then, when Save is pressed, it gives the previously selected message as a possible overwrite.

Similar behavior occurs if a `sequence` is selected and a new `sequence` is created.

6 Possible developments

! Refresh Folders are scanned at startup, so a new action or deletion made outside the application is not reflected in the lists (for instance if one updates from a version control system). At least some kind of notification should be used. Popup message? (easy); Autorefresh? (needs some more work). On the other hand, the content are read on each selection. So if a message is being edited, but modified externally, the change is not reflected

Functions For now, SocketSenderGUI allows only messages and pauses in sequences. If needed, it should be rather simple to add calls to Java

functions defined in the code of SocketSenderGUI as part of sequences. This is the reason behind the sequence grammar: to be able to add a type "function".

Functions could have their own list, in which case the editor could show a description of what it does. Or have a popup somewhere with a non-editable, non selectable list of possible functions.

Configuration

It could be possible to define several sources for each action type, with a priority order. This way, one could maintain a set of messages shared between several customers, and others unique to a given customer. In a list of 2 sources, for instance

```
"MessagesPath": {
    "actions\\GroupMessages", "actions\\CustomerMessages" }
```

if a message exists in both sources, the second one, *more specialized* is considered.

It would be interesting to be able to block the possibility to modify, delete or create new messages in some of the sources. This way one can make sure not to modify the *parent* actions, but only the *leaf versions*. In order to avoid changing the flow too much and avoid confusion by asking each time where to save, an easy step to implement the behavior above is to consider all saves and modifications to be made in the last source of the list.

Open usefull folders

It could be handy to easily open folders defined in the configuration, for instance to update messages using TortoiseSVN.

Sequence editor

The editor is now completely text-controlled. We could add click or drag actions, or some kind of way that would add messages into the editor when a sequence is edited.

In this case, if ever functions are also used, a similar action (and therefore list) should exists for them.

Correctness check

Could come in handy to check if the editor is writing a message or a sequence depending on a Json grammar definition. This way, it makes it harder to mistakenly save one in another.