Masked Input for Text Fields

Version: 1.4 (2015-02-25)

Description

The Angelwatt Masked Input script provides a way to *add a text format to a text field and enforce it's structure as the user types in values*. The mask shows the general format and gives the user an idea about how they should input the data that your form is requesting of them. A common example of this, is wanting a date to be inputted in a specific format, such as in a month, day, and year order. The common way to portray this to the user is to add some text near the text input that tells or shows the format e.g., MM/DD/YYYY or 04/23/1994. That's good and all, but a text mask can help the user out a little more. The text mask takes your format and throws it *directly inside the text field and allows the user to type over the mask as they enter in the data.* Examples are a better way to get the point across so check out some examples below.

The script has a few parameters that can be set when creating the mask. Some of these include strings that describe the format of the mask, the characters that are allowed to be typed, the characters in the mask that should be treated as separators, and the calling of a function when a non-allowed character is entered. These are elaborated on below in the installation section.

Example

Here are some text fields with different text masks applied to them so you can try them out and see how they react to input. There's a couple date formats, a social security number input, a phone number, and a random game code format (some old games presented you with a 4x4 or 4x5 matrix of numbers and or letters to mark the progress of your game). So, as you can see, the mask can be applied to text areas as well as text inputs. The script allows you to assign a function for when the user enters an invalid character. The below text masks examples show how you can indicate to your user that they entered an incorrect character. These example effects are not included as part of the masked input script.

Date 1	`11/21/2221`	
Date 2	`__12-12-12`	Year, Month, Day
SSN	`...-..-....`	
Phone	`(___) ___-____`	
Time	`HH:mm`	
Game Code	`____` `____` `____`	Allows for uppercase letters and numbers.

Be careful of using this for phone numbers. The format of a phone number is locale specific, so it's only useful if you are only accepting phone numbers from a single locale. You could potentially detect or ask the user about their locale and change the mask for the input accordingly, but that's beyond the scope of this page.

As you can see in some of the examples above, I still provide some extra text to explain the format, at least when it may be unclear what can or should be typed. This is important to remember. A text mask doesn't necessarily remove the need to have explanatory text for the user. For example, if a user has already typed in a date of 01/04/1994, is that Jan. 4th, 1994 or Apr. 1st, 1994? The user may forget what format they were typing it in, and be unconfident that they have typed it in correctly. It's up to you how to solve that, and generally not too big a deal.

Download

License

Files

- JavaScript file. Current version: 1.4 (*12 KB*), Minified (*4 KB*)

Installation

Installation and use of the script is pretty easy. First, we'll add the script to the page needing the mask. Add the below line to the head tag of your page. You'll of course need to change the *src* attribute to reflect the location of where you have saved the script at.

```
<script type="text/javascript" src="file/path/script.js"></script>
```

Next, we need to create each text mask. You can determine where to place this code. It can either go in a separate JavaScript file, or inside a script tag inside the HTML document. That's your choice, but if you reference a HTML node (tag) the code needs to run after the page is loaded. Most JavaScript libraries have easy ways of doing this. Below is an example creation of a text mask for a date field.

```
MaskedInput({
  elm: document.getElementById('mask'),
  format: 'MM/DD/YYYY',
  separator: '\/',
  typeon: 'MDY',
  allowedfx: function(ch, index) {
    var str = document.getElementById('mask').value;
    switch (idx) {
      case 1:
        return ('01'.indexOf(ch) > -1);
      case 2:
        if (str[0] === '1') {
          // Ensure month does not exceed 12
          return ('012'.indexOf(ch) > -1);
        }
        break;
      case 4:
        return ('0123'.indexOf(ch) > -1);
      case 5:
        if (str[3] === '3') {
          // Ensure day does not exceed 31
          return ('01'.indexOf(ch) > -1);
        }
        break;
      case 7:
        return ('12'.indexOf(ch) > -1);
    }
    return true;
  };
});
```

Call the *MaskedInput* function, which is the script class. The function itself takes one argument, an object. This object has multiple parameters. To describe the object, we write it in JavaScript Object Notation (JSON), which is encompassed in curly braces ({}). Inside here we define each parameter of the argument. The object takes name-value

pairs separated by a comma. Each parameter starts with the name, which are pre-defined and explained in more detail below. The name is followed by a colon and then the value for that parameter.

In the above example, the first parameter is named *elm* and has a value of `document.getElementById('mask')`, which represents a DOM node. The second parameter is named *format* and has a value of `'MM/DD/YYYY'`. And so on the parameters go as you need to define them. Not all parameters are needed, but that's all elaborated on below. You can create additional masks by calling the function again as many times as need be.

Mask Settings

There's a number of settings that can defined in the parameters when you create each mask. Below, I define each parameter and how to use it as well as note what parameters are required for it to work. Two of the settings are required, but the others are optional as they have default values associated with them that get applied if they are not defined. The defaults are good enough for numerical data such as dates, times, or SSN. The following settings do not need to be defined in any particular order, which is the benefit of using an object as the argument for the function.

- **elm:** (required) This is the element to apply the mask to. It should be an HTML node. The best way to handle this is to give the text field or textarea an *id* attribute and reference that. You can also use a variable here that references the HTML node.
- **format:** (required) The format is what the user sees in the field before they start typing. It can contain any characters you want, but try to use something that will be meaningful to your users.
- **allowed:** (default: `'0123456789'`) The characters that the user is allowed to type into the field. These can be letters, numbers, or symbols.
- **allowedfx:** (default: nothing) An optional function callback may be provided to add additional validation for allowed characters. The callback sends two arguments; the character being entered, and the current index of the string where it is being entered starting at 1 for the first character. See the above example above for potential implementation ideas. The time input example makes use of this as well. As a note, this callback is only called if the entered character is in the `allowed` set of characters, and only for visible characters. So it won't be called when the user hits the backspace key.
- **separator:** (default: `'\/:-'`) Separator characters show which characters inside the format setting should be treated as a separator so they don't get replaced as the user types in their data. When needing the '/' character remember JavaScript requires it to be escaped as can be shown in the default value.
- **typeon:** (default: `'_YMDhms'`) The typeon setting indicates which characters in the format setting can be typed on top of, as opposed to the separators. Just because a character is defined here, doesn't mean the user will be able to type them, they'll type over them, or reveal them when they delete their data.
- **onfilled:** (default: nothing) When the user has filled in the entire mask this function will trigger. This allows you to take further action based on whether they have completed the field.
- **onbadkey:** (default: nothing) When a user types a character that is not in the allowed group, the text field will not change, it'll just sit there. That isn't always a great thing to do to a user. They could benefit from a hint that they have done something wrong. The *onbadkey* setting allows you to assign a function to execute when a "bad key" has been entered by the user. Some of the examples above demonstrate you can do a shaking animation, or flash a background color, change the color of the text in the field, which gives some hint to the user that they did something wrong. You can either give it the name of the function, or write in an anonymous function. The "bad key" that was entered is also passed along to the function called, which will be the ASCII representation, e.g., 'a', '3', etc. The badkey event is also triggered when the user is at the end of the field so take that into consideration on any actions you take based on the character sent.
- **badkeywait:** (default: 0) This will only be needed when you are making use of the *onbadkey* setting. Depending on the function you run when a user hits a bad key, you may need to lock the text field so the user can't enter another key, which again activates the bad key function. Using the examples on this page, the effect of shaking the input field can get screwed up if the user enters a second bad key before the effect is over, which would cause the text field to stop in the wrong position. So, for this case, I would create a wait period long enough for the effect to complete. During this wait time the user's input will be rejected even for allowed characters. If your function for bad keys does not use an animation, or doesn't require a wait for whatever reason you can leave this setting at the default zero value.

- **preserve:** (default: true) Indicates whether the field should be filled with the mask irregardless of there being any existing text in it or not. When set to true (default), the mask will only be set to the field if it is empty. Helpful when pre-populating the filed using server side code. When set to false, it will always load the mask into the field during page load. Warning, if the supplied text for the field doesn't match up with the mask, I can't guarantee it will behave correctly when the user starts editing the field.

Public Functions

if you assign the MaskedInput function call to a variable it can call some public functions. Calling the set functions will automatically call the resetField function to avoid any invalid input in the field.

- **setEnabled(boolean)** lets you turn the mask enforcement on and off. This can be used to disable the functionality on browsers that aren't supported.
- **resetField()** will reset the field to just the format string. Useful if you're implementing a form reset.
- **setAllowed(string)** allows you to send a string of characters that the user is allowed to use in the field.
- **setFormat(string)** sets the base format for the field.
- **setSeparator** sets which characters are used as separators.
- **setTypeon(string)** sets which characters users are able to type on.

```
// Example using the public functions
var formattedInput = MaskedInput({
  elm: document.getElementById('example'),
  format: '___-__-____'
});
formattedInput.setFormat('__:__:__');
formattedInput.setSeparator(':');
```

Compatibility

This script of course can only work when JavaScript is enabled on a visitor's browser, which means any enforcement you do here will still need to be addressed by a server side language where it can be adequately validated. In its current state, this script would allow 21/21/3434 to be entered in a date field, but that is of course an invalid date. Using the `allowedfx` callback argument you can add extra validation to help improve this. You can also add a *onblur* or *onchange* event handler to the text field to run a function to do some real validation on the field, but of course that will still only work if the user has JavaScript enabled so you still need to validate your data after submission. I'm just throwing that out there so you realize it. I block most JavaScript when I surf the net, and I'm not the only one.

There are a number of incompatibilities when it comes to handling key strokes across browsers and operating systems, but I believe I have accommodated them pretty well. I've mostly checked the most recent versions of browsers except on IE, where I was able to test some older versions. If you find an incompatibility with a browser let me know. No guarantees I'll be able to fix it, but it's good to know where it doesn't work.

Browsers Tested:

- **Mac:** Firefox 4+ (at least), Safari 3+, Opera 11+
- **Windows:** IE 5.5+, Firefox, Safari, Chromium
- **Linux:** only tested Firefox

Known Issues

- Cut and paste don't work from the keyboard, but copy does. They can interfere with the formatting, but also, they're just hard to accommodate. JavaScript has no direct method to access the clipboard so having it be ignored is easiest. Right-clicking on the text or using the menus allows for these commands though.

Version History

- 1.4 (2015-02-25)
    - Added new callback, allowedfx, which gives the developer extra validation that can be performed on an

input.

- 1.3 (2013-08-19)
  - Added new event, 'onfilled' that will trigger when the user has completely filled in the field.
  - Added method to enable/disable the mask enforcement.
- 1.2.4 (2013-07-22)
  - Modified getTextCursor() for IE. Simplified code thanks to a visitor.
- 1.2.3 (2013-06-29)
  - The onbadkey event sends the bad key that triggered the event.
  - Added some adjustments based on JSLint results.
  - Updated some of the doc annotations.
  - Reordered some functions and formatted code some more.
- 1.2.2 (2013-01-10)
  - Added the 'preserve' argument.
- 1.2.1 (2012-08-22)
  - Fixed a IE9 issue caused by a bug in IE9 that cancels the keypress event if the keydown event's default behavior is prevented.
- 1.2 (2012-07-17)
  - Adding onChange event handlers to nodes that had a mask were getting ignored because the browser doesn't JavaScript changing a text field value to be a user change. This has been accommodated trigger onChange appropriately now.
  - Refactored code to be more module.
  - The onKeyDown and onKeyPress events are no longer overwritten, they're simply listened to without interfering with other listeners. The onKeyUp event is no longer listened to.
- 1.1 (2010-04-14)
  - Added functions to set certain properties such as the format, allowed characters, etc. that can be used after initial creation.
- 1.0.1 (2009-11-13)
  - Fixed problem in IE where mask disappears when the field is highlighted, as in just after tabbing to it. No other browser seemed to be effected by this. Lovely IE.
- 1.0 (2009-01-01)-ish
  - First release.

Last updated: 2015-02-25 at 07:20 PST