Single Page Web App Generator (SPWAG)

Members

Lauren Zou (ljz2112)
Aftab Khan (ajk2194)
Richard Chiou (rc2758)
Yunhe (John) Wang (yw2439)
Aditya Majumdar (am3713)

Motivation

In 2012, HTML5 and CSS3 took the web development community by storm, introducing new elements such as figures and articles as well as innovative features such as animation and transition that aim to introduce more structure to the standard web page and to decrease the amount of JavaScript needed to create fancy visual effects such as fade and slide. With the introduction of new standards came an influx of web applications technologies trying to solve various problems on "thin" clients that were previously solved by "thick" clients. Today, many web developers are familiar with object-oriented CSS languages such as SASS and LESS as well as JavaScript frameworks such as Angular and Ember, not to mention HAML, which is a shorthand way of writing HTML. While these tools aim to improve the programming experience of HTML, CSS, and JavaScript individually, they do not address the integration of these three languages as a whole.

"Single Page Web App Generator" (SPWAG) aims to combine HTML, CSS, and JavaScript into a single and simple language. In SPWAG, the developer does not need to worry about which aspects of his or her webapp needs to be coded in HTML, CSS, or JavaScript. The developer can spend more time focusing on the content of the website. SPWAG will handle cross-browser compatibility, graceful degradation, and producing a visually attractive single page web application.

Language Features

The primary goal of SPWAG is to make writing a single-page dynamic web application easier. SPWAG allows the user to program in a single language and output unified code across three languages (HTML, CSS, and JavaScript). SPWAG provides a simpler, more consistent, and concise syntax for creating animated graphical content which may be viewed via any standards compliant web browser. One of the main advantages of SPWAG is that it outputs into web languages, which makes it cross-platform compatible; one can view a SPWAG application from any operating system, tablet, or mobile device.

In order to standardize SPWAG output, we will make use of open source projects such as LESS (an object-oriented way of writing CSS), jQuery (JavaScript framework), and Prefix-Free (to support cross-browser compatibility for the CSS). Ideally, the LESS should be compiled to CSS before the output is produced.

Representative Programs

The following are programs that we would ideally be able to create using SPWAG. We have also described a high-level overview on how each program would be implemented. The native functions and syntax are explained later on in this proposal.

Animated Slideshow

SPWAG generates one-page animated content, such as slideshows, quickly and efficiently. In this program, we would define a `slide` component that would essentially be a `box()` component stylized to fit the width and height of the window. The slide would also support an `on-click()` function which would use the `hide()` function to hide the current slide and use the `show()` function to show the next slide. Each slide would be able to be customized to contain anything such as images, text, and hyperlinks.

Jeopardy Game

A Jeopardy Game can be constructed using similar functionalities from the Animated Slideshow program. In this case, each question of the game would be a `slide` component as defined in the Animated Slideshow. There would also be a slide that contains several question-picker boxes (which would be custom components) that, when clicked, will show the slide with the question. This is an example of an interactive program that can be created using SPWAG.

Image Gallery

An image gallery application can be easily implemented in SPWAG. The visual format of the image gallery would resemble Google's Image Search, such that there would be several thumbnail images that can be clicked on to produce a larger version of the image. A `thumbnail` component would be a custom `image()` component that has an `on-click()` function which will change the `width()` and `height()` attributes of the image.

Conventions

Here are some conventions for how a SPWAG program would be ideally written. These conventions do not necessarily have to be followed, but are preferred for a cleaner code presentation. SPWAG does not use camelcase for naming variables. Rather, custom names should be separated by a hyphen and should be entirely lower-case (e.g. "box-bold"). Although SPWAG does not require a specific number of spaces per tab, it is convention to only use two spaces to represent a tab. Soft tabs are also preferred over hard tabs. In any case, the compiler for SPWAG will determine what constitutes a single indentation/tab through a preprocessor that analyzes the white space (similar to the preprocessor for Haskell).

Syntax

Overview

SPWAG's basic unit of execution is a **function**. Functions can represent actions to be performed (e.g. `on-click()`), web page **components** (which are analogous to HTML elements), and **attributes** (which are analogous to but not limited to CSS styles that can be applied to the components). Components and attributes are mutually exclusive special subtypes of functions.

Functions are declared and defined using the keyword `define` along with any additional keywords as described in the following table. Note that all function definitions end with the keyword `end`.

Defining Functions

`define comp page()`
`  ...`
`end`
	defines the special component page, which is analogous to the main method in Java; this component is the main "executable" component

`define comp my-comp() isa spwag-comp()`
`  ...`
`end`
	defines a custom component; the `spwag-comp` must be either a custom component already defined somewhere in the code or a native SPWAG component such as `box()` or `image()`

`define attr my-attr()`
`  ...`
`end`
	defines a custom attribute

`define my-funct()`
`  ...`
`end`
	defines a custom function

Rules for Functions, Components, and Attributes

Similar to the structure of elements in HTML, all custom components must extend a custom component that is either another custom component or a native SPWAG component. To extend another component, the keyword `isa` is used. When a custom component extends another component, it automatically inherits the properties of the other component. The custom component can override and append to the properties of the other component with its own properties.

It is illegal to create a custom component with the same name as a native component or another component defined in the same code. For example, a custom component cannot be called `box()` since `box()` already exists as a native component.

`page()` is a special component which must be included in every SPWAG program. The `page()` component serves as an entry point for the program; one may think of this component as analogous to Java's main method. This is the only component that is created using `define comp` but does not require an `isa` keyword to extend to another component. `page()` also cannot be called from another function.

All functions are called from this component during the program execution.

A custom attribute is a collection of other attributes. Custom attributes are used to implement reusable modular styling. However, custom attributes may not contain non-attribute functions.

The body of functions can generally include any other functions as well as other statements. Effects are listed below, where each column represents the entity that is calling the entity in each row.

Calling Functions

Caller	Regular Function	Component	Attribute
Regular Function	Executes the function	Executes the function	Illegal
Component	Binds the component to the nearest outer component	Binds the component to this component	Illegal
Attribute	Binds the attribute to the nearest outer attribute or component	Binds the attribute to this component	Binds the attribute to this attribute
Statements	Performs the statement	Performs the statement	Performs the statement

Data Types

The two main data types in SPWAG are integers and strings. Following JavaScript's syntax, data types are not explicitly declared in SPWAG. In a traditional language such as Java, we have to explicitly state which type a variable is (e.g. `int integer`, `String str`). In SPWAG, it is not necessary to explicitly declare a data type (e.g. `var integer`, `var str`); instead, the data type can be inferred from the declared variable value, which allows us to have fewer reserved keywords. Since the computer screen is split up by discrete, countable pixels, it is not essential to introduce floating point numbers into SPWAG.

Integers can fall under several different integer types: regular integer, pixel, and percentage. A regular integer does not require a suffix, but a pixel integer should always be followed by a 'px' (e.g. 300px) and a percentage integer should always be followed by a '%' (e.g. 100%). Thus, the integer type is implicitly declared.

Strings are identifiable as they are enclosed in double quotes "". Strings are very multi-functional in SPWAG. Strings can be used to define colors (e.g. `color("#F56991")`), messages (e.g. `text("Hello world!")`), and unique ids for components (e.g. `box("my-box")`).

Booleans are handled in a similar way as booleans are handled in C. False is equal to the regular integer 0, and everything else represents true.

Variables

SPWAG allows for variables, which are permitted to refer to strings or integers but not functions. Variables are declared via the `var` keyword.

Keywords

`if <condition>`
`  ...`
`else`
`  ...`
`end`
	if-else conditional

`while <condition>`
`  ...`
`end`
	while loop

`var my-var`	to define a variable

Native Functions

`random(integer)`	takes in an integer and returns a random integer between 0 and integer given in the parameter where the integer given in the parameter is non-inclusive e.g. `random(6)` produces a random integer between 0 and 5
`on-click(function)`	handles on click events
`on-hover(function)`	handles on hover/mouseover events
`link-to(string)`	links to a hyperlink specified in the parameter
`hide(component)`	hides the element given in the parameter
`show(component)`	makes the element given in the parameter visible if it was previously hidden

Native Components

`page()`	main function, root-component
`box(string)`	a box, analogous to an HTML `<div>`; the parameter is the unique id of the element
`text(string, string)`	a floating text element, analogous to an HTML `<span>`; the first parameter is the unique id of the element and the second parameter is the message within the text element

`image(string, string)`	an image, analogous to an HTML `<img>`; the first parameter is the unique id of the element and the second parameter is the url of the image
`field()`	a text input field, analogous to an HTML `<input>`

Native Attributes

`color(string)`	defines the color of the element (accepts RGB, HSL, hex)
`width(string)`	defines the width of the element
`height(pixel)`	defines the height of the element
`font(string)`	defines font family if applicable
`font-size(pixel)`	defines font size if applicable (accepts the font size as a pixel)
`font-decoration(string)`	defines font attribute (accepts “bold”, “italic”, “underline” values)
`margin(pixel)`	defines the amount of outer spacing around the component
`padding(pixel)`	defines the amount of spacing inside the boundaries of the component
`animation(string)`	defines the transition for the component when it is hidden or shown (accepts “slide <direction>” which will slide either up, down, left, or right or “fade” which will fade in/out the element)

Operators

<	less than operator; can only compare variables of the same type
>	greater than operator; can only compare variables of the same type
=	assignment operator
==	equal to
!=	not equal to
+	string concatenation, integer addition; addition for integers, pixels, and percents may only occur with two variables of the same data type; a string can be concatenated to any data type and the result will be a string
-	subtraction; only works for integers, pixels, and percents; subtraction for integers, pixels, and percents may only occur with two variables of the same data type
*	multiplication; only works for integers, pixels, and percents; multiplication for integers, pixels,

	and percents may only occur with two variables of the same data type
/	division; only works for integers, pixels, and percents; division for integers, pixels, and percents may only occur with two variables of the same data type
|	or
&	and

Comments

```
# This is a single line comment.

## This is a multi-lined comment.
   We can use this to comment out code,
   write a poem or a story or some documentation. ##
```

Examples

Creating a Blank Page with a Pink Background

SPWAG

```
define comp page()
  color(“pink”)
end
```

OUTPUT

```
<html>
  <head>
    <style type="text/css">
      body {
        background: pink;
      }
    </style>
  </head>
  <body>
    <script>
    </script>
  </body>
</html>
```

Generate a Number of Colored Circles

SPWAG

```
# Global variables
var number-of-circles = 4;
var circle-radius = 300px;

# Main method
define comp page()
  var i = 0 #=> local variable
  while i < number-of-circles
```

OUTPUT

```
<html>
  <head>
    <style type="text/css">
      .circle {
        border-radius: 100%;
        width: 50px;
        height: 50px;
      }
```

```
    var r = random(256)
    var g = random(256)
    var b = random(256)
    var rgb = “rgb(“ + r + “,“
              + g + “ ,“ + b + “)”
    circle(rgb)
    i = i + 1
  end
end

# Generates circles
define comp circle(color) isa box(color)
  color(color)
  border-radius(100%)
  width(circle-radius)
  height(circle-radius)
end
```

```
    </style>
  </head>

  <body>
    <div class="circle"
style="background:
rgb(204,185,64)"></div>
    <div class="circle"
style="background:
rgb(87,182,158)"></div>
    <div class="circle"
style="background:
rgb(171,72,112)"></div>
    <div class="circle"
style="background: rgb(25,34,123)"></div>
    <script>
    </script>
  </body>
</html>
```

Generate a Button that Shows a Text Box

SPWAG

```
# Main method
define comp page()
  tbox(“textbox”, “Hello world!”)
  hide(“textbox”)
  button(“button”, “textbox”)
end

# Button component
define comp button(id, tbox) isa box(id)
  text(“Button”)
  on-click(show-color(tbox))
end

# Text box component
define comp tbox(id, msg) isa box(id)
  text(msg)
    # make text bold and italicized
    bitext()
end

# Bold, italicized text
define attr bitext()
  font-decoration(“bold”)
  font-decoration(“italic”)
end

# Shows a component and color it pink
define show-color(component)
```

OUTPUT

```
<html>
<head>
  <style type="text/css">
    .tbox span {
      font-weight: bold;
      font-style: italic;
    }
  </style>
</head>
<body>
  <div id="textbox" class="tbox">
    <span>Hello world!</span>
  </div>
  <div id="button" class="button">
    <span>Button</span>
  </div>

  <script>
    $('#textbox').hide();

    $('#button').click(function() {
      $('#textbox').css('background',
'pink');
      $('#textbox').show();
    });
  </script>
</body>
</html>
```

```
  component
    color(“pink”)
  show(component)
end
```