SAGE2 Developer Documentation

Table of contents

- Introduction
  - What is an application
  - Runtime model
  - State
- Getting started
  - Writing an application
    - Name of the application
    - Default Methods
    - Writing Methods
    - Events
    - Application user interface
  - Minimal application
  - application types
    - DOM
    - Canvas
    - SVG
    - D3
    - WebGL
    - three.js
- User Interface
  - Widgets
  - Button
  - Slider
  - Text Input
- Tutorials
  - First application
    - Generating the skeleton
    - Output

instructions.json
Code

1. Introduction

Target audience is a developer for SAGE2 with knowledge of DOM (web programming) and Javascript.

1.1. What is an application

- an application is a Javascript object
    - Javascript program files
    - instructions.json (defined below)
    - .webp/.png/.jpg image for the logo of the application
    - All code written for the application must be in Javascript and stored in the application directory. All server related code will be located in server.js, and executed by running with node.js
- it inherits from the base class SAGE2_App
- overrides a list of methods to define its own behavior
    - init
    - load
    - draw
    - resize
    - move
    - event
    - quit
- Built-in applications (in SAGE2 core, sage2/public/src)
    - image viewer
    - movie player
    - pdf viewer
    - desktop sharing
    - notes
    - streaming application
- Provided applications (folders in sage2/public/uploads/apps)
    - each application in a folder
    - instructions.json
        - settings for this class of application
        - default values
    - main source file
        - main Javascript file loaded by SAGE2 display clients
    - an icon file
        - any valid ‘web’ image (jpeg, png, webp, …)
        - around 512 pixels, square
        - transparency supported
    - dependencies
        - builtins to SAGE2: snap.svg, D3, THREE.js, ..

        - provided by the application: extra Javascript libraries inside the application folder
    - resources
        - images
        - 3D models
        - data files: JSON, CSV, …
    - instructions.json
        - In addition to your application files, you should include a Instructions.json file. This file will act as a configuration file for the application when it is started in the SAGE2 environment. You should include the following in this file:
            - main-script: javascript file with all of our source code for your application
            - icon: image file with your logo to appear as the application icon
            - width/height: width and height of application when it first opens
            - dependencies: dependencies javascript files which would otherwise be added as <script> tags, usually these files are stored in a directory called “scripts” within the application directory.
            - title: name of application as shown on SAGE2.
            - description: your application description
            - author: name and email address usually

Instruction.json fields

Property name	Type	Values	Notes
`main_script`	`string`	`<path>`	The path to the file containing the application definition
`resize`	`string`	`'proportional'`, `'free'`	The resize mode of the window
`width`	`int`		The width of the window in pixels
`height`	`int`		The height of the window in pixels

`animation`	`boolean`		unknown
`sticky`	`boolean`		unknown
`dependencies`	`[string]`	`<path>`, `<uri>`	A list of dependencies of the application
`load`	`object`		An object passed into the `load` function at startup
`icon`	`string`	`<path>`	The file containing an application icon
`title`	`string`		
`version`	`string`		The current version of the application
`description`	`string`		A short description of the purpose of your application
`author`	`string`		
`license`	`string`		The name of the license of the application
`keywords`	`[string]`		A list of keywords associated with the application
`filetypes`	`[string]`		A list of filetypes that the application can open
`directory`	`string`		unknown

1.2. Runtime model

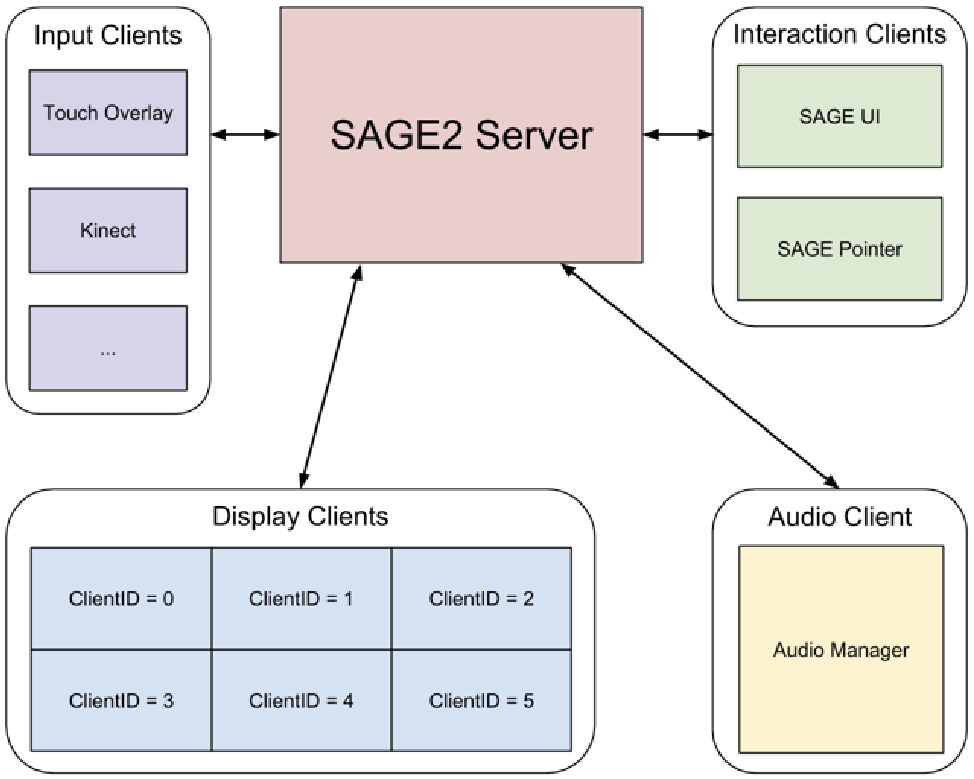


Fig. SAGE2 architecture

- one application object runs in each display client (browser)
- execution driven by the server through RPC calls to each client
    - server maintains a state object for each running application
- Server sends ‘create application’ message to each display client
    - display client creates an instance from the application class (new)
    - call ‘init’
- Server sends ‘draw’ calls to display clients for each applications (as needed to achieve required frame rate)
- Display clients send back message to server before next frame

1.3. State

- synchronized state variable to maintain consistency across displays
- when a display client joins, it receives the state

- state saved in session files
- state is passed when loading a session file and restoring the applications
- state saved automatically on the server

2. Getting started

2.1. Writing an application

While writing a sage2 application refer to the sage2 API wiki found at:

https://bitbucket.org/sage2/sage2/wiki/SAGE2%20Application%20API

It is useful to read one of the provided applications' source code to see how a SAGE2 applications are built. You can find all applications source code in:

<sage2_directory>/public/uploads/apps/

A developer can also place his/her application outside the SAGE2 installation folder and in the Documents/SAGE2_Media/apps folder.

2.1.1. Name of the application

Define your SAGE2 application as a variable. The name of the *var* should be the same as the 'title' component in your instructions.json.

2.1.2. Default Methods

- *Init* - initialize your settings on the application. This is where you set up any controls or buttons/layers that you will access later. This is where your application's starting screen should be coded. The state variable (this.state) is already filled.
- *Draw*
- *Resize* - any code about resizing your application goes here.
    - To learn more about these application methods, refer to the SAGE2 API.

2.1.3. Writing Methods

Define methods in a similar manner to how the default methods are written, this is done in javascript and should follow the usual javascript conventions.

2.1.4. Events

These will include events that will be fired when a user interacts with your application. For example you can cause the application to react to keyboard events - ex. like if the user presses the space-bar, your application will reload, etc.
Define events at the bottom of your application.js file as the following:

event: function(eventType, position, user_id, data, date) {}

eventType - pointerPress, pointerRelease, pointerScroll, keyboard, etc
data.character - to get the character that was pressed on the keyboard

2.1.5. Application user interface

Widget Menu
The widget menu can be started by opening a pointer, and right-clicking on your application while it is running in the display screen of SAGE2. This is handled in the applications controls var (this.controls).

Adding Widget Menu Component
Each item on the widget menu is assigned a sequence number. That number determines where each item lies on the widget menu. For example, to add an up arrow:

```
	this.controls.addButton({type:"up-arrow",sequenceNo:4
```

This will add a button for the up-arrow and put the it in the 4th position on the widget menu. Alternatively, you can set the type like so:

```
	var weatherLabel = { "textual":true, "label":"W", "fill":"rgba(250,250,250,1.0)",
"animation":false};
```

textual means that you want the label to be text based, and then you just set the label to be some text and it will populate that control with whatever you put in there, in this case, it would be “W”.

Any time you want to add another control to the widget menu, make sure the sequence number that you are putting in is not already taken, if it is, you will have to choose another number or rearrange the sequence numbers of all the controls to fit yours in.

2.2. Minimal application

- To generate a new application, run the command
  - ***npm run newapp***
    - Running "prompt:genapp" (prompt) task
    - ? Application name: **mynewapplication**
    - ? Author first name: **Luc**
    - ? Author last name: **Renambot**
    - ? Author email: **renambot@gmail.com**
    -
    - New application done: mynewapplication in sage2/public/uploads/apps/mynewapplication
    - Done, without errors.
  -

2.3. application types

Applications can be of various type. All applications inherit from the class *SAGE2_App.*

All application are attached to the DOM through a HTML ‘div’ element, referred to as *‘this.div’.* It is possible to combine several drawing techniques at once (layers using several ‘div’ elements).

2.3.1. DOM

- pure HTML application
- create elements using the DOM javascript API (*document.createElement, …)*
- Example

2.3.2. Canvas

- 2D pixel drawing
- Immediate mode
- Can be combined with other drawing techniques

2.3.3. SVG

- Scalable graphics is useful for SAGE2 given the size of the displays
- Pure SVG is an option
- The *Snap.svg* library is already loaded inside SAGE2 and is available to the developer
- Example
  - // Make the SVG element fill the app
  - this.svg = Snap("100%","100%");
  - // Adding it to the DOM
  - this.element.appendChild(this.svg.node);
- See *‘snap_one’* example

2.3.4. D3

- Popular D3.js: Data-Driven Documents
  - http://d3js.org/
  - Excerpt: “D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3’s emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation”
- Provide scalable graphics
- Many examples in SAGE2 folder
  - d3_sample, US_weather, evl_photos, …
- Many plotting libraries built on top of D3.js: C3, NVD3, ...

2.3.5. WebGL

- To leverage 3D graphics capabilities of the browser
- WebGL: i.e. OpenGL ES
- Other libraries can be added like ‘lightgl.js’
  - https://github.com/evanw/lightgl.js
  - *A lightweight WebGL librar*y

- Example: *texture_cube*

2.3.6. three.js

- Scene-graph based 3D (camera, lights, animation, material, shaders, …)
- File loader and conversion utilities
- Many plugins and modules available for three.js
- built on top of WebGL
- Example: *threejs_sample*, *threejs_shader*, *threejs_loader_ctm*, …
- **three.js**
- Scene-graph based 3D (camera, lights, animation, material, shaders, …)
- File loader and conversion utilities

3. User Interface

The following write up describes how an application developer can program application specific user interface for custom applications. The task of creating a user interface and associating it with an application is handled by the SAGE2 server. There are two simple steps an application developer has to follow to get controls for an application and make them work.

1. Specify to the server, the controls that are needed by an app.
2. Have mechanisms in the event handler of the app, to handle events generated by users of the application in interacting with these controls.

These two steps are further elaborated upon below.

When done creating UI elements, call the function 'finishedAddingControls' to declare the end of the UI specification.

3.1. Widgets

If you have started writing a custom application for SAGE2 then you must be aware that all SAGE2 applications extend SAGE2_applicationobject. Thus, all custom applications come to posses a "controls" property
which you need to access to specify the user interface elements your application will need.

The different user interface elements that are available are:

- Button
- Slider
- Text Input

3.2. Button

In its simplest form a call to add a button looks like this:

```
this.controls.addButton({label: "Play", identifier: "PlayButton"});
```

The object being passed as the argument to the *addButton* call has two properties: label and identifier. As the name indicates, specifying a label will create a button with the label text displayed on the button. The identifier is the name you are giving to this button. This name will be used to associate action for the button-click event.

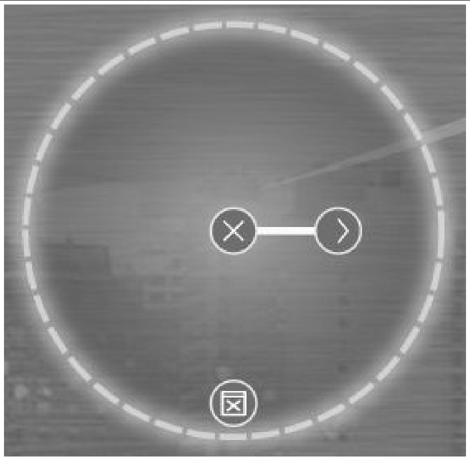
Further, you can specify the position of the button around the radial dial of the user interface by giving a third property to the object being passed:

```
this.controls.addButton({label: "Play", identifier: "PlayButton", position: 1});
```

A value of 1 for position puts a button on the left of the ring at 0 degree to the horizontal and buttons with increasing integer values for position will be placed in clockwise order around the dial. If position is not specified for a button, next available position is assigned to it.

Instead of a label, you can specify an icon for the button by replacing the label field with a type field:

```
this.controls.addButton({type: "next", identifier: "NextButton", position: 7});
```



In the above example, the type field has been set to “next”. This is one of the predefined values that the type property can take(the list of values is given at the end of this document). This set of values is provided for convenience. Alternatively, you can define your own button “type” using “svg” path descriptors.

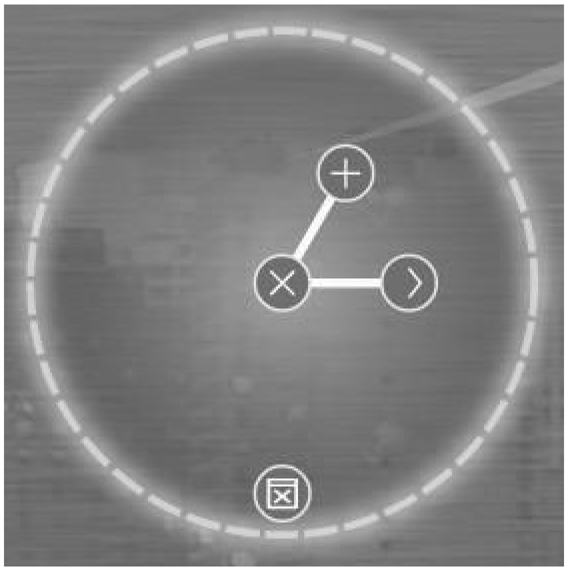
```
var plusButton = {
       "state": 0,
       "from":"m 0 -6 l 0 12 m 6 -6 l -12 0", //svg paths
       "to":"m 6 0 l -12 0 m 6 6 l 0 -12",
       "width":12,
       "height":12,
       "fill":"none",
       "strokeWidth": 1,
       "delay": 600,
       "textual":false,
       "animation":true
};
this.controls.addButtonType("plus", plusButton);
this.controls.addButton({type:"plus", identifier:"Plus3", position: 5});
```

The button type object has several properties that are used to describe the look and feel of the button. “state” refers to the state of animation. It is set to either 0 or 1 depending on whether you want the default look of the button to be drawn using the path element “from” or “to” respectively. When the button is clicked, the look of the button animatedly transitions from one of these states to the other. If state is set to null, then the look of the button transitions from one state to the other and back again upon a click.

The figure below shows the “plus” button added to the user interface. You could directly use the button type object as the value of the type in a call to addButton, like this:

```
this.controls.addButton({type:plusButton, identifier:"Plus3", position: 5});
```

This will create the same result, and you don’t have to add the button type before hand.


Two buttons are provided by default, one at the center of the radial dial to hide the user interface and a second button at the bottom to close the application itself.

3.3. Slider

To add a slider you may call *addSlider* on the “controls” property as shown below:

```
this.controls.addSlider({

        identifier: "MySlider",

        minimum: 5,

        maximum: 10,

        property: "this.options.brightness"

});
```

The object being passed as argument for the addSlider call in the above example has four properties namely: identifier, minimum, maximum, and property. As in case of buttons, identifier is the name that you assign to the slider so as to associate action code with slider events(lock, move, release). “property” field refers to property of the application that you wish to control through the slider. The fully qualified(dot separated) name of the property must be specified as a string against this field. minimum and maximum specify the range of values that the property will take during the course of execution of this app.

Slider from the above example will move between the values of 5 and 10. If these are all the fields that are set, then this interval is broken into 100 parts by default. You may override this behaviour in two ways. You may specify the number of steps your slider should move from minimum before reaching the maximum, or you may specify the amount by which your slider should increment each time it moves. The following examples illustrate these two options.

```
this.controls.addSlider({

        identifier: "MySlider",

        minimum: 5,

        maximum: 10,

        property: "this.options.brightness",

        steps: 10

});
```

Here the slider will have 10 intervals between 5 and 10 where it can halt. In other words, each interval will be 0.5.

```
this.controls.addSlider({
        identifier: "MySlider",
        minimum: 5,
        maximum: 10,
        property: "this.options.brightness",
        increment: 0.2
});
```

Here the slider will have intervals of 0.2 and 25 such intervals. In case you specify both steps value and an increment value, the increment value will take precedence.

List of predefined values for button type:

- *"play-pause"*
- *"mute"*
- *"loop"*
- *"play-stop"*
- *"stop"*
- *"next"*
- *"prev"*
- *"up-arrow"*
- *"down-arrow"*
- *"zoom-in"*
- *"zoom-out"*
- *"rewind"*
- *"fastforward,*
- *"duplicate"*
- *"new"*
- *"closeBar"*
- *"closeApp"*
- *"remote"*
- *"shareScreen"*
- *"default"*

3.4. Text Input

To add a text input to the widget bar, the application may call *addTextInput* function on the controls. addTextInput function takes an object with id as a mandatory property and two optional properties namely *defaultText* and caption. id is a String to associate action code in the app.event handler to the "Enter"(or "Return") key event, which marks the end of the input. The variable *defaultText* can be set to any string value and the widget will be created by placing this string in the text area. The variable *caption* takes a small (length smaller than 6 characters) string and sets a visible label in front of the text input control using this string.

```
this.controls.addTextInput({defaultText: "", caption: "Addr", id: "Address"});
this.controls.finishedAddingControls();

//In the event handler with in the custom app code:
event: function(eventType, position, userId, data, date) {
    if (eventType === "widgetEvent") {
        switch(data.ctrlId){
            case "Address":
                // Code to be executed when Enter key is hit by the user
                break;

            // Other controls follow
        }
    }
}
```

When done creating UI elements, call the function ‘finishedAddingControls’ to declare the end of the UI specification.

4. Tutorials

4.1. First application

Generating the skeleton

```
% npm run newapp

> SAGE2@0.3.0 newapp /Users/luc/Dev/GIT/bitbucket/sage2
> grunt newapp

Running "prompt:genapp" (prompt) task
? Application name: tutorial
? Author first name: Tino
? Author last name: Pizza
? Author email: tino@pizza.com

Running "genapp" task
New application done:  tutorial in
/Users/luc/Dev/GIT/bitbucket/sage2/public/uploads/apps/tutorial
```

```
Done, without errors.
```

Output

instructions.json

```
{
	"main_script": "tutorial.js",
	"icon": "tutorial.png",
	"width": 800,
	"height": 600,
	"resize": "free",
	"dependencies": [
	],
	"load": {
		"value": 4
	},
	"title": "tutorial",
	"version": "1.0.0",
	"description": "SAGE2 application tutorial",
	"keywords": [ "sage2", "tutorial" ],
	"author": "Tino Pizza <tino@pizza.com>",
	"license": "SAGE2-Software-License"
}
```

Code

```
//
// SAGE2 application: tutorial
// by: Tino Pizza <tino@pizza.com>
//
// Copyright (c) 2015
//

var tutorial = SAGE2_App.extend( {
	init: function(data) {
		// Create div into the DOM
		this.SAGE2Init("div", data);
		// Set the background to black
		this.element.style.backgroundColor = 'black';

		// move and resize callbacks
		this.resizeEvents = "continuous";
		this.moveEvents   = "continuous";

		// SAGE2 Application Settings
		//
		// Control the frame rate for an animation application
		this.maxFPS = 2.0;
		// Not adding controls but making the default buttons available
		this.controls.finishedAddingControls();
		this.enableControls = true;

		console.log('tutorial> Load with state value', this.state.value);
	},
        draw: function(date) {
                console.log('tutorial> Draw with state value', this.state.value);
        },

        resize: function(date) {
                this.refresh(date);
        },
        move: function(date) {
                this.refresh(date);
        },

        quit: function() {
                // Make sure to delete stuff (timers, ...)
        },
        event: function(eventType, position, user_id, data, date) {
                if (eventType === "pointerPress" && (data.button === "left")) {
                }
                else if (eventType === "pointerMove" && this.dragging) {
                }
                else if (eventType === "pointerRelease" && (data.button === "left")) {
                }

                // Scroll events for zoom
                else if (eventType === "pointerScroll") {
                }
                else if (eventType === "widgetEvent"){
                }
                else if (eventType === "keyboard") {
                        if (data.character === "m") {
                                this.refresh(date);
                        }
                }
                else if (eventType === "specialKey") {
                        if (data.code === 37 && data.state === "down") { // left
                                this.refresh(date);
                        }
                        else if (data.code === 38 && data.state === "down") { // up
                                this.refresh(date);
                        }
                        else if (data.code === 39 && data.state === "down") { // right
                                this.refresh(date);
                        }
                        else if (data.code === 40 && data.state === "down") { // down
                                this.refresh(date);
                        }
                }
        }
});
```