MirrorBox Designer

Inhalt

Set-Up

- Install node.js (http://nodejs.org/)
- Start the Server by entering
  ```
  node index.js
  ```
  into the terminal in the *server*-folder
- Designer: http://localhost:8888/
- Output for MirrorBox: http://localhost:8888/output.html

The server needs to be restarted each time after changing the code.

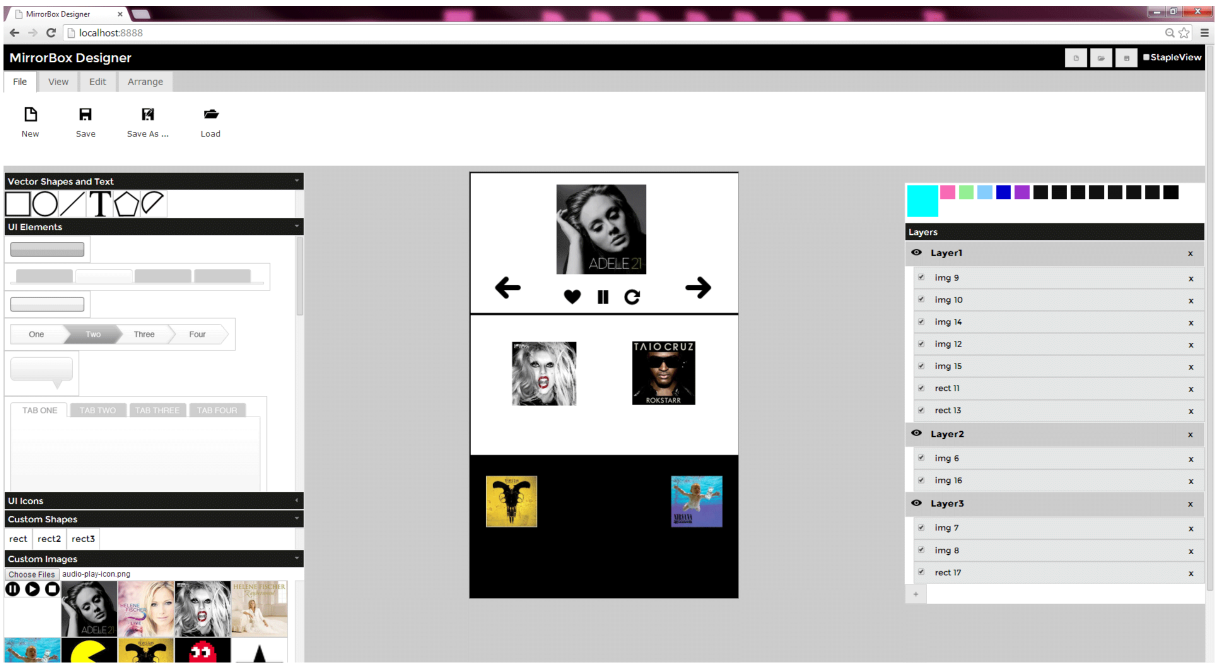
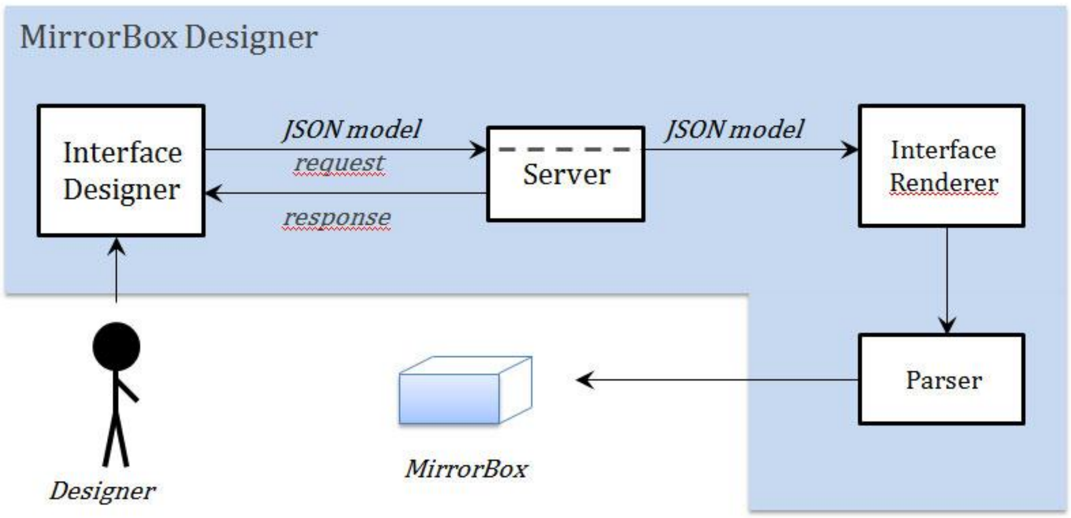


Abbildung 1 MirrorBox Designer UI

Structure


MirrorBox Designer
Interface Designer
JSON model
request
response
Server
JSON model
Interface Renderer
Parser
Designer
MirrorBox

Client

Model View Controller (MVC) architecture

`index.html`	Contains HTML code of the MirrorBox Designer and includes all relevant scripts	View
`index.js`	JQuery Code enhancing the UI with effects and functionality. From here functions in the ViewController.js are called according to events (click, drag and drop, …). No real logic should be implemented here.	View
`viewInititalizer.js`	Called in the very beginning (onload). Initializes the MirrorBox Designer: creates the stage (for createjs), loads all used files (images), defines general settings.	Controller
`viewController.js`	Sends requests to the server (by calling functions from requestBase.js) and is responsible for the content of the UI.	Controller
`requestBase.js`	Sets up all requests sent to the server. Used by the ViewController.js	"Helper"
`editController.js`	As the functionalities in the "edit-tab" are quite extensive, they are handled in a separate file (not in View Controller like the rest).	Controller
`htmlTags.js`	Because of the dynamic nature of the edit-tab, each component of the model is assigned a number of html-parts that are displayed when clicking it. They are assembled here. The html definition can be found in the html folder of the client.	"helper"
`selectedElement.js`	Extends the elements from the model (like with a border around them, scaling functionality)	Model-Extension
`state.js`	Data and functionalities relevant for the current session. Example: currently selected Element, current stage, view, etc.	Model-Extension
`parser.js`	Converts a JSON representation of the model into js and vice versa.	
`outputMain.js`	Equivalent to viewController.js but configured for the needs of the output.	
`outputSate.js`	Equivalent to State.js but configured for the needs of the output.	

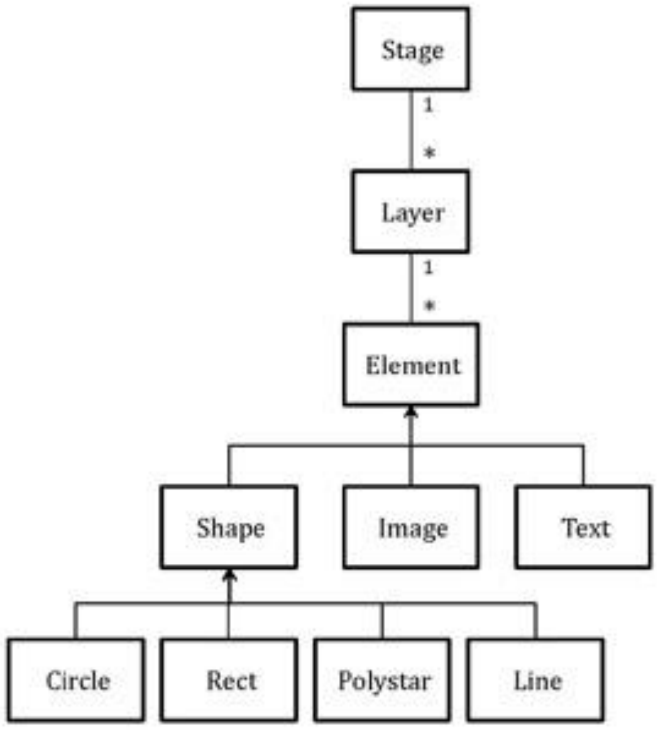
Model

The model is represented two ways:

1. It is formulated in JSON, which allows to permanently store it and transfer it from client to server and vice versa. This data format also ensures that exports to various output devices, such as maybe in later stages, a real stereoscopic 3D display, are easy to handle by creating a parser that translates this into the used language.

2. It is implemented in the native language of the clients (JavaScript - see 5.3) and is enriched with functionality. This representation is used to actually display the data on the UI. To allow a fast transition between them, each element (like circle, rect, etc. see 5.2) of the second representation needs to include the following functions: toModel(), which turns a JSON representation of it into the one used by the clients, toJSON() that performs the opposite operations and toHTML() that returns a HTML representation of the element. Those are called by a parser that is able to convert the models as a whole or parts of it.

General



JSON-Model

Example of a scene with one layer and one circle in JSON.

```
{
   "stage":{
      "layers":[
         {
            "id":2,
            "elements":[
               {
                  "id":3,
                  "x":100,
                  "y":100,
                  "elemIndex":1,
                  "shape":{
                     "fill":true,
                     "color":"rgb(153, 50, 204)",
                     "stroke":{
                        "isStroke":true,
                        "strokeColor":"#00ffff",
                        "strokeStyle":2
                     },
                     "circle":{
                        "radius":50
                     }
                  },
                  "transformations":{
                     "alpha":1,
                     "scaleX":1,
                     "scaleY":1,
                     "rotation":0
                  }
               }
            ]
         }
      ]
   }
}
```

JS-Model Conventions

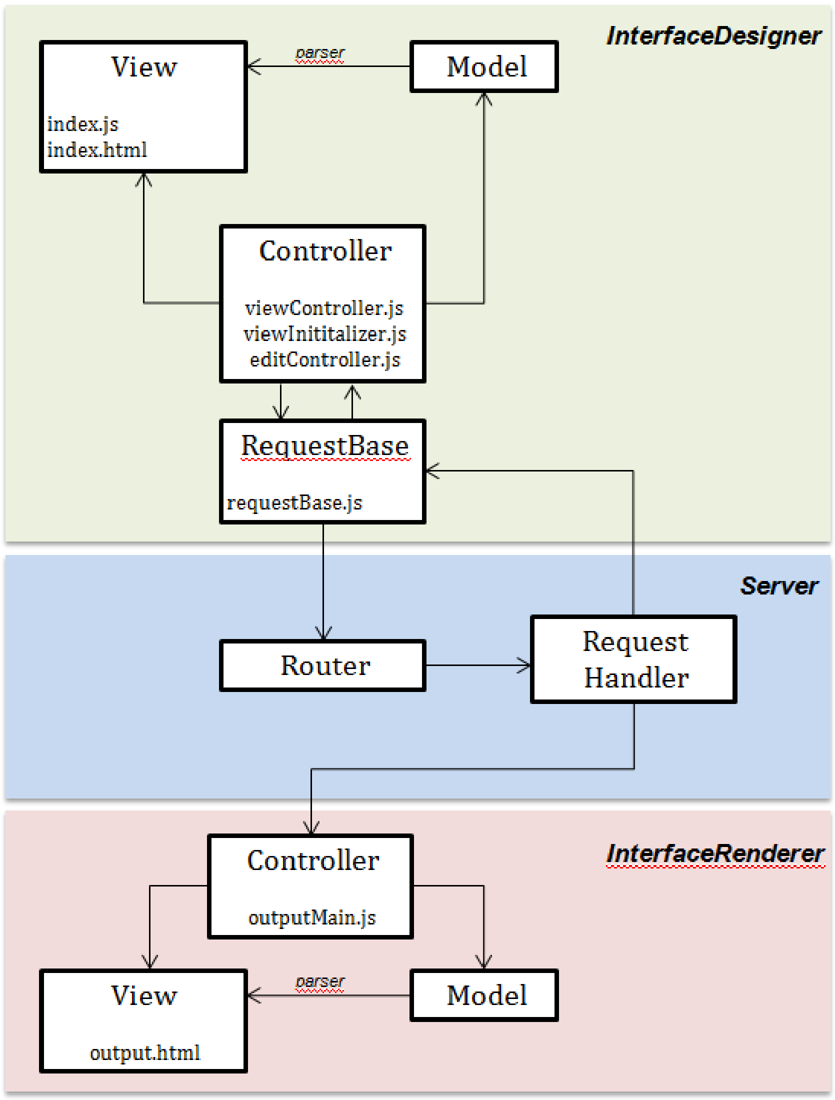
- Names of createJS objects have have the prefix "createjs"
- each element (text, shape, image) has a property createjsShape, which represents the model in the UI with EaselJS.
- properties that can be derived directly from this EaselJS object are not defined redundantly in the element-object. Example: x/y-position.
- properties such as color are defined specifically
- each element has the functions:
  - toJson()   -> turns the object to json
  - toModel()   -> turns json into the object
  - toHtml()   -> returns the html-representation of the element
  - redrawGraphic() -> called when a property changes that requires the graphic to be redrawn
  - getELementProp() -> returns the "type" of the element e.g. "circle" or "rect"

- Containers (Layer, Stage) have a "HashMap" "elements" where the key is the element id in quotation marks and the value the element itself. Also "elements" has an array with the key "ids" with all ids at the correct index. the top element is in front

- Elements from the stage that have a representation in the html-dom have the id: <element-type>_<id>, e.g. "layer_1"

Server

Server structure inspired by "The Node Beginner Book" (http://www.nodebeginner.org/).



Json Requests

Not every operation in the interface designer results in parsing the whole model and sending it to the server. This only happens when the user saves a session, starts a new one or loads an existing one. Changes other than that are transmitted by exclusively sending the adapted content, therefore reducing traffic. There are three possible scenarios for layers and elements. They can be created, updated and deleted.

InterfaceDesigner → Server → InterfaceRenderer

1. Updates

```
{
"update": []
}
```

Layer Update

only changes the index of Layers

```
{
    "update": [
        {
            "layerId": 5,
            "layerIndex": 2
        }
    ]
}
```

Element Update

redraws the whole Element

```
{
    "update": [
        {
            "layerId": 5,
            "elementId": 1,
            "elemnet": {
                "id": 1,
                "x": 400,
                "y": 400,
                "elemIndex": 0,
                "shape": {
                    "color": "#ABECE2",
                    "fill": true,
                    "circle": {
                        "radius": 110
                    }
                },
                "transformations": {
                    "alpha": 1,
                }
        }
```

```
            }
        ]
    }
```

Element Transformation Update

only sets Element-Transformations (no redrawing required)

```
    {
      "update": [
              {
                  "layerId": 5,
                  "elementId": 1,
                  "element": {
                      "id": 1,
                      "transformations":  {
                          "alpha": 1,
                      }
                  }
              }
      ]
    }
```

Element Index Update

```
    {
        "update": [
            {
                "layerId": 5,
               "elementId": 1,
               "action": "oneForward, oneBackward, toFront, toBack"
            }
        ]
    }
```

Element Layer Update

```
    {
        "update": [
            {
                "layerId": 5,
               "elementId": 1,
               "newLayerIndex": 2
            },
        ]
    }
```

2. Adds

Add Layers

```
{
    "add": [
        {
            "layerId": 6,
            "layerIndex": 4,
        }
    ]

}
```

Add Elements

```
{
    "add": [
        {
            "layerId": 5,
           "elementId": 1,
           "element": {
                "id": 1,
                "x": 400,
                "y": 400,
                "elemIndex": 0,
                "shape": {
                    "color": "#ABECE2",
                    "fill": true,
                    "circle": {
                        "radius": 110
                    }
                },
                "transformations": {
                    "alpha": 1,
                }

            }
        },
    ]
}
```

3. Removes

Remove Layers

```
{
    "remove": [
        {
            "layerId": 5,
        }
    ]
}
```

Remove Elements

```
{
       "remove": [
           {
               "layerId": 5,
               "elementId": 3,
           }
       ]
}
```

4. Save

send the whole Stage (the way it is stored)

```
{
  "stage": {
       "w": 1200,
       "h": 800,
       "layers": [
           {
               "id": 5,
               "elements": [
                   {
                       ...
                   }
               ]
           }
       ]
  }
}
```

InterfaceDesigner → Server → InterfaceDesigner

1. ***Change Session Default Settings***

   (only what should be changed is sent)

```
{
    "color": "#ff00ff",
    "currentFile": "./sessions/..",
    "layerInfo": {
        w: 300,
        h: 340
    }
}
```

2. ***Change Element Default Settings***

```
{
    "changeSettings": [
        "circle": {
            "id": -1,
            "x": 100,
            "y": 100,
            "elemIndex": 0,
            "shape": {
                "fill": true,
                "circle": {
                    "radius": 50
                }
            },
            "transformations": {
                "alpha": 1,
            }
        },
        "rect": {...},
        "polystar": {...},
        "img": {...},
        "text": {...}
    ]
}
```

3. ***New Custom Shape***

   …