JavaScript Library for HTML Canvas

FabricJS

Some of the problems with this native API, first and foremost, the lack of an object model. When we draw a shape on the canvas, we do not get a reference back to it.Therefore we don't get the ability to manipulate individual shapes on the canvas. Therefore we don't get the ability to manipulate individual shapes on the canvas.

Fabric.js is a JavaScript library that provides a high-level API for native Canvas Web API. It abstracts the native Canvas API calls and gives us an **object-oriented model** to work with which solves most of our problems.
With the help of Fabric.js, every shape drawn on the canvas is a JavaScript object with properties and methods. Using these properties and methods we can manipulate the shape on the canvas without having to **redraw** the objects on the canvas manually. Fabric.js takes care of it for us.
The main power of Fabric.js is the **ability to freely move and manipulate objects** (shapes) on the canvas using the mouse pointer or touch gesture.

Integrating FabricJS in WorkSpace -
To get the library .js distribution file, either you can use the fabric.min.js from their official GitHub repository or use a public CDN link.
We can also use a package manager such as **NPM** or **Bower** to install it as a dependency.

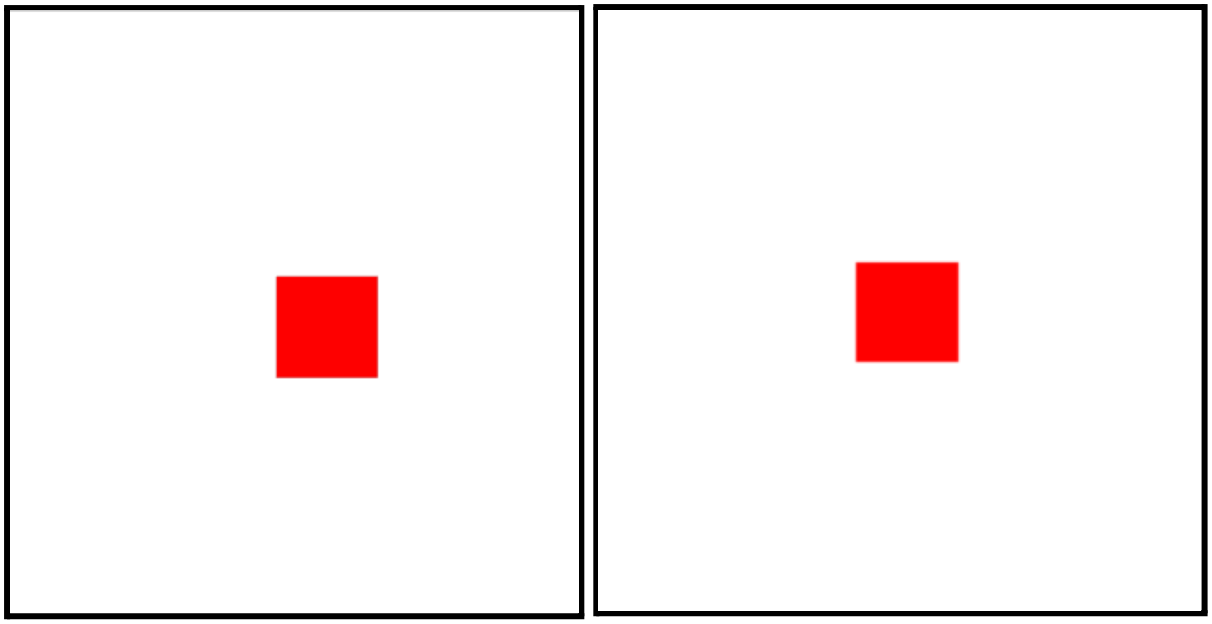
The fabric constant is an object that exposes various properties, methods, and classes of the Fabric.js API. The Canvas class creates a wrapper around the native canvas element and exposes an API to add and remove things from it (*meaning drawing objects on the canvas*). It accepts the <canvas> element's id (*in our case, it is canvas*) to access this element from the DOM.
The Rect class creates a rectangular shape which can be added to the canvas object using the add method provided by it.

Creating a canvas in FabricJS is super simple-

```
<canvas id="canvas" ></canvas>
var canvas = new fabric.Canvas('canvas');
```

Lets see a difference between creating a shape with and without using FabricJS

With FabricJS	Without FabricJS
`var rect = new fabric.Rect({ left: 100, top: 100, fill: 'red', width: 20, height: 20 }); // "add" rectangle onto canvas canvas.add(rect);`	`var ctx = canvasEl.getContext('2d'); // set fill color of context ctx.fillStyle = 'red'; // create rectangle at a 100,100 point, with 20x20 dimensions ctx.fillRect(100, 100, 20, 20);`



In FabricJS we define elements in the form of Objects with different properties and finally add them to the canvas.

Getting familiar with shapes (objects)

Every shape we can possibly draw on the <canvas> inherits from the fabric.Object class. For example, the rect.set() which is a prototype method of fabric.Rect comes from the fabric.Object since Rect inherits from the Object class. Therfore every shape drawn on the canvas is an instance of fabric.Object and we could just it an “object”.

Fabric.js gives us the ability to create **7 primitive** shapes

```
fabric.Line( [ ...points ], { ...options } );
fabric.Circle( { radius, ...options } );
fabric.Rect( { width, height, ...options } );
fabric.Triangle( { width, height, ...options } );
```

```
fabric.Ellipse( { rx, rx, radius, ...options } );
fabric.Polyline( [ ...points ], { ...options } );
fabric.Polygon( [ ...points ], { ...options } );
fabric.Path( commands, { ...options } );
```

1. **Straight Line**

```
const line = new fabric.Line( [ 50, 50, 200, 50 ],
{
 strokeWidth: 5,
 stroke: '#03A87C'
} );
```

2. **Circle**

```
const circle = new fabric.Circle( {
   top: 50, left: 250,
   radius: 50, fill: '#AB47BC'
} );
```

3. **Triangle**

```
const triangle = new fabric.Triangle( {
   top: 60, left: 50,
   width: 50, height: 90, fill: '#5C6BC0'
} );
```

4. **Rectangle**

```
const rect = new fabric.Rect( {
   width: 100, height: 100,
   top: 200, left: 50, fill: '#8BC34A'
```

```
} );
```

5. Ellipse

```
const ellipse = new fabric.Ellipse( {
   rx: 100, ry: 50,
   top: 200, left: 200, fill: '#FFC107'
} );
```

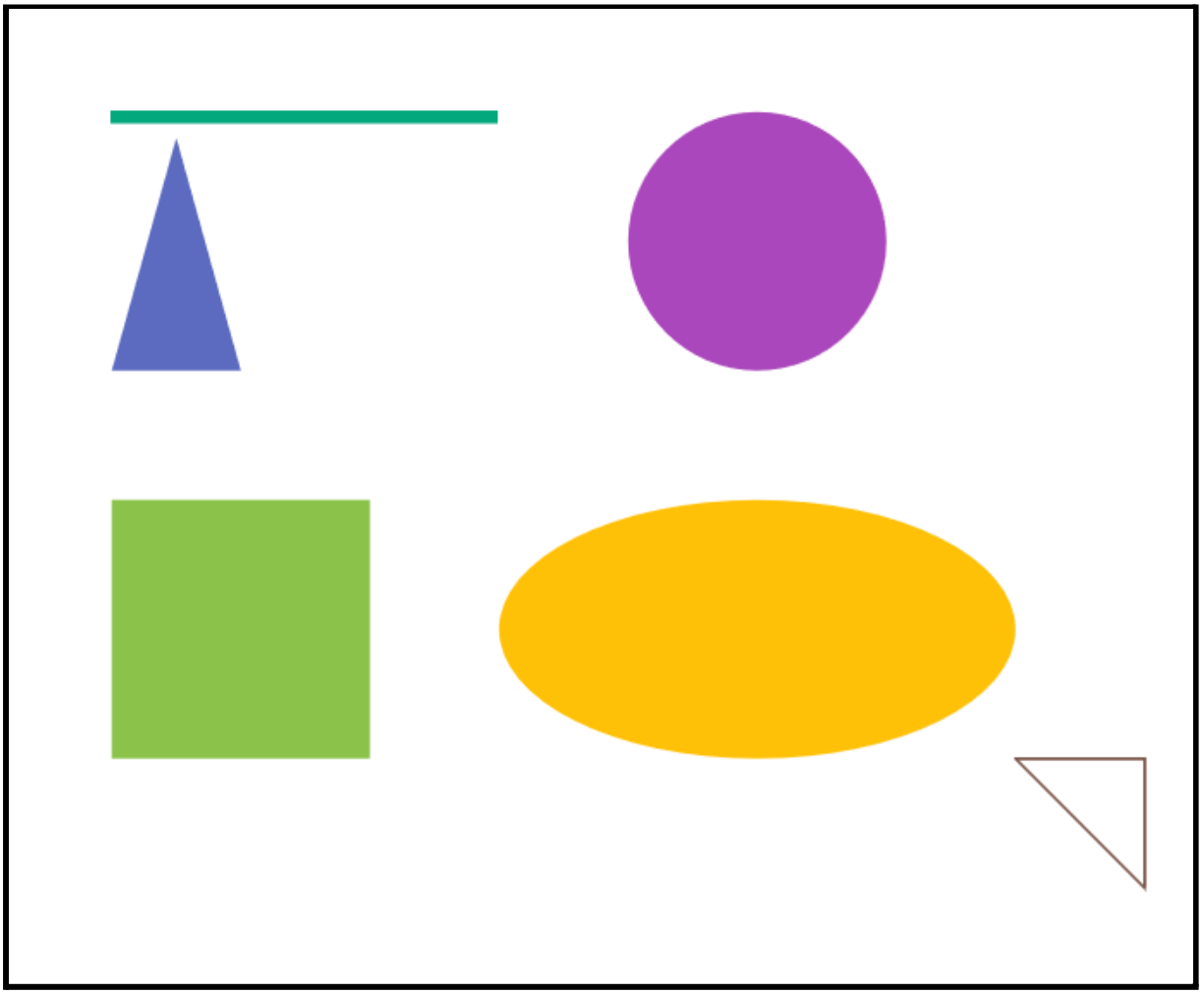
6. Polyline

```
const polyline = new fabric.Polyline( [
   { x: 0, y: 0 }, { x: 50, y: 0 }, { x: 50, y: 50
}
], {
   top: 300, left: 400,
   fill: 'transparent', stroke: '#795548'
} );
```

7. Polygon

```
const polygon = new fabric.Polygon( [
   { x: 0, y: 0 }, { x: 50, y: 0 }, { x: 50, y: 50
}
], {
   top: 300, left: 400,
   fill: 'transparent', stroke: '#795548'
} );
```

Screenshoot of the result -



Code for the following demo ---
https://codepen.io/Devartstar/pen/abpWYxW

Complex Paths and Shapes

So far we have seen classes that produce basic shapes such as circles, triangles, polygons, etc. What if need to draw a complex shape or path such as a Bézier curve or a heart shape that can't be described by the given primitive shapes?

```
var path = new Path( 'commands', { ...options } );
```

There are several properties associated with each object which can be found out in the documentation - http://fabricjs.com/docs/

SVG Manipulation using FabricJS

What is Scalable Vector Graphics (SVG).?

- A language for describing 2D graphics in XML.
- A drawing program, where image can be generated with various popular tools, i.e. drawing application in Google Drive, Inkscape, Illustrator, Corel Draw and lots of others.
- Used to define vector-based graphics for the Web
- Quality of images are maintained, even if zoomed or resized
- Every element & attribute can be animated.
- Perfect for applications with large rendering areas or Maps.

Canvas-loading from a string:

There are two types **JSON & SVG** representation

JSON has fabric.Canvas#loadFromJSON & fabric.Canvas#loadFromDatalessJSON methods.

SVG has fabric.loadSVGFromURL & fabric.loadSVGFromString.

The first two methods are instance methods and are called on a canvas instance directly.The other two methods are static methods and are called on the “fabric” object rather than on canvas.

SVG-loading methods -

There are two methods: string **or URL**. Below describes the example of string method:

```
fabric.loadSVGFromString('...', function(objects,
options) {
   var obj = fabric.util.groupSVGElements(objects,
     options);
   canvas.add(obj).renderAll();
});
```

The first is the SVG string, and the second is the callback function.

Callback is invoked when SVG is parsed, loaded and receives two arguments i.e. objects and options.

Objects, contains an array of objects parsed from SVG i.e. paths, path groups (for complex objects), images, text & so on.

To group those objects into a cohesive collection as in an SVG document use **fabric.util.groupSVGElements** and pass it to both objects and options.

In return, you will receive **fabric.Path** or **fabric.PathGroup,** which can be added onto canvas.

The **fabric.loadSVGFromURL** method **works the same way**, **except to pass a string containing a URL instead of SVG** contents. The SVG needs to conform to the usual SOP rules, as Fabric attempt to fetch that URL via XML Http Request.

How to Handle SVG Image Files with Canvas.?

It is difficult to just draw HTML into a canvas. It essentially requires an SVG image containing the content. To draw HTML content, use a & lt;foreignObject> HTML element, then draw that SVG image into your canvas. It Conveniently load up & draw a SVG image JavaScript element to the canvas.

```
varsource = newImage();
source.src = 'http://example.org/myfile.svg';
source.width = '100';
source.height = '100';
```

I think that combining the functionalities of HammerJS and FabricJS we can achieve all the advanced features we want to create in CircuitVerse.