**Home**

This is the home page of the website. It starts with a sticky navigation bar on the top; from there down, we have two sections and a footer. The first section is the *hero* section; it contains some key information of the website, such as a short introduction, opening hours of the establishment, and a group photo that presents every member of the bar. The second section contains some customer reviews and a gallery showing some of the bar’s treats. Lastly, on the bottom there is the footer of the website. All these contents were created using HTML, CSS, and a bit of Vanilla JavaScript. Now, for the HTML and CSS parts, there isn’t really much to talk about, it’s just basics; but as far as **JavaScript** goes, first it was used to implemented the smooth scroll effect from the hero section to the reviews section; the image lazy loading effect was also implemented by some JavaScript code.

The smooth scroll effect essentially was handled by a click event. So, first the *check-out-reviews* button is clicked, and that triggers an event in which the ***getBoundingClientRect*** function is called on the desired section, then a *DOMRect* object will be returned (for convenience, let’s call it **domRectObj**) and domRectObj can be used to get the distance from the top of the screen to the top of the desired section by accessing to its *top* property. So with ***domRectObj.top*** and another property under the window object – ***scrollY***, which detects the distance from the top of the page to the top of the screen, we can pass their sum to the ***scrollTo*** function as the value for the *top* property of the argument object that the function requires, and set the *behavior* property to “smooth”. With this setup, every time when the *scrollTo* function gets executed, the browser will scroll from the very top of the page to the top of the section smoothly, so this is how the smooth scroll effect is implemented.

The image lazy loading effect is an effect where images won’t get loaded and displayed on screen until they come into the screen when user scrolls down the page. For achieving such an effect, we will need the section observation functionality provided by the ***intersection observer API***. At first, I attach to every img tag the path of an image loading gif file as the value for its src attribute, so before an img tag enters the designated viewport, these tags will only display the img loading animation. Meanwhile, the path of the real image is stored in the img tag using the *data-src* attribute. When an img tag enters the viewport (or makes the intersection), the observer’s callback function executes; in this function, the path of the real img will be grabbed from the data-src attribute and get fed to the src attribute, so right away the real image will be displayed. And this is how the image lazy loading effect is achieved.

**Gallery**

This is gallery page of the website, here you’ll find many photos about the bar and its lovely customers and members. The images are all embedded in the page using the *img* tag which is contained by the *figure* tag. Then all these figure tags are contained by a gallery container. I made this container a CSS grid container, so all the figure tags are arranged in a 5 x 5 grid. When one of those images get clicked, a lightbox will be opened, which presents the image in a much larger size. The lightbox comes with a left and a right arrow, which direct user in both backward and forward directions, respectively. The first image doesn’t have the left arrow, and the last image doesn’t have the right arrow. On the top right corner, there is a page number indicator. When the user clicks anywhere outside the lightbox, the lightbox will be closed.

How this lightbox is implemented is very simple. A class is created for containing all the html code which makes up this lightbox. The lightbox is basically a div that should cover the entire page with the lightbox content – the clicked image – displayed in the center. When the main support JS script for the gallery page runs, the Lightbox class is imported and its ***activate*** method gets called. This method will select the body tag of the page and insert the lightbox code before its end tag; but by default, the lightbox is not showing because its inline display attribute is set to “none”. When an image is clicked, the *show* method is called and the lightbox will be shown and most of the operations related to the lightbox are handled in the supported JS script. In terms of JavaScript, all these operations are just some basic common DOM manipulations, so there’s no need to go really deep in this aspect.

Another cool feature for this gallery page is the swipe gestures implemented by using two DOM event types - “touchstart” and “touchend”, so on a mobile screen user can swipe left or right for browsing the gallery.

The way how I laid out everything on the gallery page – using a class to create a reusable component with static class – was actually how I developed the food-ordering part of this website, which I will talk more in detail in the next section – Order.

**Order**

The order page is where user enters and orders food from the menu on that page. This is also the most complicated part of developing the website because it involves a lot of programming for taking and managing order work, therefore it can get very complex to look at the code on a very low level; but on a high-level overview, how the system works is actually pretty simple. In general, there are three systems working behind the scene – the order system, the cart system and the payment system. The order system is responsible for taking order from user on a UI and store user order data in local storage; the cart system will then retrieve data from local storage and display the information on the cart UI and is responsible for user to make further options for their order and then move on to the payment phase; lastly, the payment system supports the UI where user can input payment information and make final payment. The three systems are separated, but from the cart UI user can come back to the order UI by clicking the *edit* button of a cart item, so there are two situations where the order system can be used: one, user orders food by a clicking food item on the menu; two, user edits existing cart item by clicking the *edit* button on the cart UI. The payment system will kick once user clicks the “Check Out” button on the cart UI.

The three system adopts one kind of pattern where a main JS file acts as the pivot of all the components, utility functions and event listeners that related to that particular system. In the beginning of the main JS file, a bunch of import statements will be executed to import all the components. All these components were created for making up the UI for that particular system. They are JavaScript classes with a bunch of static methods, and they all have one *activate* method, which will be called in the main JS file after the import statements to inject HTML code to the page. The content created by this HTML code will be not shown at first and will only be shown when the class’s shown method is executed under certain condition. Next in the main JS file, some utility functions will be created for processing data taken from user; the data will be processed here and then be rendered by those static methods provided by the components. So, these utility functions constitute the “*processing layer*” and the components or those JS classes, which make up the UI, are the “rendering layer”. The last part of the main JS file is where the event listeners live. These event listeners altogether are what’s responsible for taking inputs from the user and send the data to the *processing* layer for further processing.