JavaScript: creare un carousel di prodotti

In questo articolo vedremo come creare un carousel di prodotti con JavaScript.

Il carousel che andremo a creare si estenderà in larghezza per tutta l'ampiezza della finestra del browser e verrà centrato verticalmente.

Ogni elemento del carousel avrà un'immagine, un titolo, una descrizione e un prezzo. Quando si effettua un click sull'elemento, si aprirà una finestra modale che mostrerà l'immagine, il prezzo ed una call to action. La finestra modale potrà essere chiusa sia con un click su un pulsante di chiusura sia con un click su un qualsiasi punto della finestra che non sia il contenuto principale.

Il carousel scorrerà da destra verso sinistra. Il numero di elementi selezionati per lo scorrimento verrà determinato dinamicamente in base alla risoluzione della finestra del browser. Poiché i contenuti verranno reperiti dinamicamente da un'API remota, inseriremo inizialmente dei contenuti statici fittizi come segnaposto e mostreremo un indicatore di caricamento che scomparirà quando la richiesta AJAX verrà completata e le immagini remote saranno state caricate.

La struttura HTML

Partiamo dalla struttura HTML di base.

```
<section id="carousel" class="carousel">
        <nav class="carousel-nav">
            <button type="button" class="carousel-
prev">
                <span class="sr">Previous</span>
                <img src="icons/angle-left-solid.svg"
alt="" aria-hidden="true">
            </button>
            <button type="button" class="carousel-
next">
                <span class="sr">Next</span>
                <img src="icons/angle-right-
solid.svg" alt="" aria-hidden="true">
            </button>
        </nav>
        <ol class="carousel-wrap">
            <li class="carousel-item">
                <img class="carousel-image"
src="images/1.jpg" alt="">
            </li>
            <!--...-->
        </ol>
</section>
<div id="carousel-overlay">
        <a href="#carousel-overlay" id="carousel-
overlay-close">&times;</a>
        <div id="carousel-overlay-content">
            <article id="carousel-overlay-product">
</article>
        </div>
</div>
<div id="loader" aria-hidden="true">Loading...</div>
```

Gli stili CSS

Il primo passo da compiere è definire gli stili CSS globali del documento.

```
@font-face {
    font-family: 'Inter';
    src: url('fonts/Inter-Regular.ttf')
format('truetype');

}

* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

html, body {
    overflow-x: hidden;
}

body {
    width: 100%;
    min-height: 100vh;
    display: flex;
    justify-content: center;
    align-items: center;
    flex-direction: column;
    font-family: Inter, sans-serif;
}

button {
    appearance: none;
    border: none;
```

```
    background-color: transparent;
}

.sr {
    position: absolute;
    top: -9999em;
}
```

`overflow-x: hidden` è fondamentale in quanto non vogliamo che il browser mostri una barra di scorrimento orizzontale, cosa inevitabile in quanto gli elementi supereranno la larghezza complessiva della finestra.

L'elemento `body` è qui un box Flex con layout colonnare in modo da centrare verticalmente il carousel sulla pagina. Ciò è reso possibile anche dal fatto che `body` copre in altezza tutta l'altezza della finestra grazie alla dichiarazione `min-height: 100vh` (l'unità `vh`, ricordiamolo, sta per *viewport height*).

Ora possiamo definire gli stili della barra che contiene i pulsanti "Precedente" e "Successivo".

```
.carousel {
    position: relative;
}

.carousel-nav {
    position: absolute;
    top: 50%;
    left: 0;
    width: 100%;
    transform: translateY(-50%);
    display: flex;
    align-items: center;
    justify-content: space-between;
```

```
    z-index: 2;
}

.carousel-nav button {
    width: 24px;
    height: 24px;
    display: block;
    cursor: pointer;
    background-color: #fff;
    border-radius: 50%;
}

.carousel-nav button img {
    display: block;
    width: 100%;
    height: 100%;
}
```

Il posizionamento assoluto avviene in modo contestuale. Dichiarando `position: relative` sul contenitore principale, la barra di navigazione si posizionerà in relazione a tale contenitore e non in relazione all'intera finestra del browser. Qui la barra viene centrata verticalmente usando insieme le proprietà `top` e `transform`: il 50% dell'una viene compensato dallo stesso valore negativo dell'altra con la funzione `translateY()`. I due pulsanti vengono posti rispettivamente a sinistra e a destra con la dichiarazione `justify-content: space-between` che in un contesto Flex ripartisce uniformemente lo spazio tra gli elementi.

Passiamo quindi a definire gli stili principali del carousel stesso.

```
.carousel, .carousel-wrap {
    width: 100%;
    display: flex;
}
```

```
.carousel-wrap {
    position: relative;
    z-index: 1;
    transition: all 400ms ease-in-out;
    list-style: none;
}

.carousel-item {
    min-width: 16.6666%;
    max-width: 100%;
    margin: 0;
    position: relative;
    cursor: pointer;
    display: block;
    height: 300px;
    overflow: hidden;
}

@media screen and (max-width: 768px) {
    .carousel-item {
        height: 200px;
    }
}
```

La lista ordinata `.carousel-wrap` viene dichiarata come contenitore Flex e viene definita una transizione CSS su di essa che in seguito servirà per animare lo scorrimento orizzontale. Le voci di lista `.carousel-item` hanno una larghezza adattiva specificata in percentuale: la larghezza minima è del 16%, ottenuto dividendo 100 (larghezza totale) per 6 in modo da avere 6 elementi visibili per ogni scorrimento, mentre la larghezza massima, usata sui piccoli schermi, è del 100%. L'altezza sarà di 300 pixel nelle risoluzioni più grandi e 200 pixel sui dispositivi mobile in modalità portrait.

Ciascuna voce di lista crea a sua volta un posizionamento contestuale con la dichiarazione `position: relative`. Poiché gli elementi di ciascuna voce verranno posizionati in modo assoluto, usiamo la dichiarazione `overflow: hidden` per evitare che i contenuti fuoriescano in altezza o larghezza da ciascuna voce.

Definiamo quindi gli stili per gli elementi contenuti nelle voci della lista.

```
.carousel-image {
    display: block;
    width: 100%;
    height: 100%;
    object-fit: cover;
}

.carousel-caption {
    margin: 0;
    opacity: 0;
    display: block;
    width: 100%;
    position: absolute;
    bottom: 0;
    left: 0;
    background-color: rgba(0, 0, 0, 0.7);
    color: #fff;
    padding: 1rem;
    line-height: 1.5;
    transition: opacity 400ms ease-in;
}

.carousel-caption span {
    display: block;
}
```

```
.carousel-caption .carousel-title {
    font-size: 1.2rem;
    letter-spacing: .1rem;
    text-transform: uppercase;
    margin-bottom: 0.35rem;
    line-height: 1;
    display: flex;
    justify-content: space-between;
    align-items: center;
}

.carousel-item:hover .carousel-caption {
    opacity: 1;
}
```

L'immagine del prodotto si adatterà alla larghezza dell'elemento tramite la dichiarazione `object-fit: cover`. Si tenga presente che la proprietà `object-fit` ha effetto solo quando l'immagine ha delle dimensioni dichiarate. Il box di descrizione che contiene il titolo, il prezzo e la didascalia del prodotto è posizionato in modo assoluto nella parte inferiore dell'elemento ed è inizialmente nascosto azzerando la sua opacità. Quando l'utente passa con il mouse sull'elemento, l'opacità viene riportata al suo valore predefinito.

Per regolare questa feature e le dimensioni delle voci del carousel sui dispositivi mobile, dobbiamo aggiungere due Media Query.

```
@media screen and (max-width: 1024px) {
    .carousel-item {
        min-width: 50%;
    }
    .carousel-caption {
        opacity: 1;
    }
```

```
}

@media screen and (max-width: 768px) {
    .carousel-item {
        min-width: 100%;
    }
        .carousel-caption {
            opacity: 1;
        }
}
```

Sugli smartphone (larghezza massima 768 pixel), avremo una sola voce visibile alla volta perché la larghezza minima sarà del 100%. Sui tablet (larghezza massima 1024 pixel), avremo due voci visibili alla volta perché la larghezza minima sarà del 50%. In entrambi i casi il box di descrizione sarà sempre visibile perché su mobile il passaggio del mouse sugli elementi non è presente.

I termini usati, smartphone e mobile, non sono del tutto esatti: in realtà le Media Query in questo caso selezionano la larghezza delloe schermo e non il dispositivo in uso, quindi se si ridimensiona la finestra del browser si otterranno gli stessi risultati anche su desktop.

Dobbiamo ora definire gli stili per la finestra modale a comparsa che verrà mostrata quando si effettua un click su ciascun elemento del carousel.

```
#carousel-overlay {
    position: fixed;
    width: 100%;
    height: 100vh;
    top: 0;
    left: 0;
    z-index: 9999999;
    background-color: rgba(255, 255, 255, 0.7);
```

```
    color: #000;
    display: none;
    transition: opacity 400ms ease-in;
    opacity: 0;
}

#carousel-overlay-close {
    position: absolute;
    top: 1rem;
    right: 1rem;
    text-decoration: none;
    z-index: 1000;
    color: #000;
    font-size: 1.4rem;
}

#carousel-overlay-content {
    position: absolute;
    z-index: 2000;
    padding: 1rem;
    background-color: #fff;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    box-shadow:
            0 2.8px 2.2px rgba(0, 0, 0, 0.034),
            0 6.7px 5.3px rgba(0, 0, 0, 0.048),
            0 12.5px 10px rgba(0, 0, 0, 0.06),
            0 22.3px 17.9px rgba(0, 0, 0, 0.072),
            0 41.8px 33.4px rgba(0, 0, 0, 0.086),
            0 100px 80px rgba(0, 0, 0, 0.12);
    border-radius: 5px;
    width: 40%;
}
```

```
@media screen and (max-width: 768px) {
    #carousel-overlay-content {
        left: 0;
        transform: none;
        width: 100%;
        height: 100vh;
        top: 0;
        border-radius: 0;
        box-shadow: none;
    }
    #carousel-overlay-close {
        z-index: 3000;
    }
}

#carousel-overlay-product {
    display: flex;
    align-items: center;
    flex-wrap: wrap;
}

#carousel-overlay-product .carousel-product-details {
    width: 60%;
    display: flex;
    flex-wrap: wrap;
    justify-content: center;
    align-items: center;
}

#carousel-overlay-product .carousel-product-details
button {
    appearance: none;
    border: none;
```

```
    cursor: pointer;
    padding: 0.85rem 1.2rem;
    background-color: #000;
    color: #fff;
    text-transform: uppercase;
    letter-spacing: .1em;
}

#carousel-overlay-product .carousel-product-details
span {
    font-size: 1.2rem;
    font-weight: bold;
    color: #000;
    display: inline-block;
    margin-right: 0.75rem;
}

#carousel-overlay-product figure {
    width: 35%;
    margin-right: 5%;
}

#carousel-overlay-product figure img {
    display: block;
    max-width: 100%;
    height: auto;
}

@media screen and (max-width: 768px) {
    #carousel-overlay-product {
        height: 100%;
        flex-direction: column;
        justify-content: center;
    }
```

```
    #carousel-overlay-product .carousel-product-
details {
        width: 100%;
    }
    #carousel-overlay-product figure {
        width: 100%;
        margin-bottom: 1rem;
        margin-right: 0;
    }
}
```

La finestra modale è costituita da un elemento di sfondo semitrasparente e dal suo contenuto principale centrato orizzontalmente e verticalmente all'interno dello sfondo. Sui piccoli schermi tale contenuto rimarrà sempre centrato grazie alle proprietà Flexbox, ma l'immagine e il box con la call to action verranno mostrati in verticale anzichè in orizzontale come nelle risoluzioni di schermo maggiori. Ciò è possibile grazie al modello Flex, a patto che ciascun contenitore abbia un'altezzaa dichiarata che prescinda dal calcolo automatico delle altezze intrinsiche di ciascun elemento discendente.

L'ultimo elemento da definire è l'indicatore iniziale di caricamento.

```
#loader {
    position: fixed;
    top: 0;
    left: 0;
    background-color: #fff;
    color: #000;
    z-index: 999999;
    width: 100%;
    height: 100%;
    display: flex;
    flex-direction: column;
```

```
    justify-content: center;
    align-items: center;
    font-size: 1.5rem;
    text-transform: lowercase;
    font-weight: bold;
    letter-spacing: .1rem;
    transition: opacity 400ms ease-out;
}
```

Usiamo spesso il posizionamento fisso nel definire gli elementi a tutto schermo per due motivi: tale posizionamento permette di avere una precedenza visuale sugli altri elementi della pagina e, aspetto non meno importante, annulla lo scorrimento verticale in modo da evitare la comparsa della barra verticale del browser qualora fossero presenti altri elementi sulla pagina oltre al nostro carousel.

Il codice JavaScript

Come prima cosa dobbiamo definire delle funzioni di utility che conterranno la logica delle azioni comuni all'interno del nostro codice.

Partiamo dal reperimento dei prodotti dalle API remote.

```
const getData = async (limit = 24) => {
        try {
            const res = await
fetch(`https://dummyjson.com/products/?
limit=${limit}`);
            return await res.json();
        } catch(err) {
            return [];
        }
    };
```

Usiamo il modello async/await con le Fetch API in modo da evitare callback concatenati restituendo invece i risultati in modo diretto. Ricordiamo comunque che la Promise restituita riporterà l'array dei prodotti in caso di successo o un array vuoto in caso di errore.

Poichè una parte del comportamento del nostro carousel sarà diversa sui dispositivi mobile, dobbiamo definire una funzione che ci permetta di individuare tali dispositivi.

```
const isMobile = () =>
/mobile/gi.test(navigator.userAgent);
```

La funzione ricerca una corrispondenza tra la stringa restituita dalla proprietà `userAgent` e la sottostringa `mobile` o `Mobile`, presente nei browser mobile.

Ora dobbiamo reperire la larghezza complessiva della finestra del browser. Possiamo definire una funzione per questo scopo.

```
const getViewportWidth = () => {
        return (window.innerWidth ||
document.documentElement.clientWidth);
    };
```

L'OR logico presente nella funzione selezionerà la proprietà `innerWidth` dell'oggetto `window` se questa è disponibile o in alternativa la proprietà `clientWidth` dell'elemento radice del documento HTML (in questo caso `html`). Il numero intero restituito rappresenterà la larghezza in pixel della finestra del browser.

Definiamo quindi una funzione che effettua il preload delle immagini remote reperite dalle API.

```
const preloadImage = src => {
        return new Promise((resolve, reject) => {
            const image = new Image();
            image.onload = () => resolve({src,
status: true});
            image.onerror = () => reject({src,
status: false});
            image.src = src;
        });
    };
```

La funzione restituisce una Promise in cui lo stato `resolved` è collegato all'evento `load` dell'immagine corrente e lo stato `rejected` all'evento `error`. In questo modo quando creiamo una nuova istanza dell'oggetto `Image` e le assegniamo la proprietà `src` tramite il parametro della funzione, si innescherà la risoluzione dei due eventi che indicherà l'avvenuto caricamento (`load`) o un eventuale errore (`error`).

Adesso dobbiamo invece definire una funzione che ci permetta di stabilire se un elemento del carousel è visibile nell'area costituita dalla larghezza della finestra (viewport) del browser.

```
const isInViewport = element => {
        if(element === null || !element) {
            return false;
        }
        const rect = element.getBoundingClientRect();
        return (rect.left >= 0 && rect.right <=
(window.innerWidth ||
document.documentElement.clientWidth));
    };
```

Se un elemento ha l'offset sinistro (`left`) uguale o maggiore di 0 e l'offset destro (`right`) minore o uguale della larghezza della finestra del browser, la funzione restituirà `true`, altrimenti `false`. Gli offset sono proprietà dell'oggetto ottenuto invocando il metodo `getBoundingClientRect()` dell'elemento esaminato. Si tenga presente che tali valori saranno uguali a 0 se l'elemento è stato nascosto con la dichiarazione CSS `display: none`, perché in quel caso il modello di layout dell'elemento è stato completamente resettato.

L'ultima funzione di utility da definire è quella che gestisce lo scorrimento orizzontale del carousel.

```
const slideTo = (element = null, offset = 0, delay =
400) => {
        return new Promise(( resolve, reject ) => {
            if(element === null) {
                reject(false);
            }
            const sign = offset === 0 ? '' : '-';
            const rule =
`translateX(${sign}${offset}px)`;
            element.style.transform = rule;
            setTimeout(() => {
                resolve(true);
            }, delay);
        });

    };
```

Il parametro `offset` definisce il valore in pixel con cui spostare il carousel da destra a sinistra. La funzione CSS `translateX()` riceverà un valore negativo (`sign` uguale a `-`) se il valore in pixel è maggiore di 0, altrimenti un valore positivo (`sign` uguale ad una stringa vuota). La funzione restituisce una Promise perché dobbiamo creare un ritardo con la funzione

`setTimeout()` pari al valore in millesimi di secondo del parametro `delay` in modo che lo scorrimento sia sincronizzato con la transizione CSS.

Siamo dunque arrivati al momento di gestire il funzionamento del carousel stesso. Creiamo una classe `Carousel` con questi componenti iniziali.

```
class Carousel {
        constructor(settings = {}) {
            const defaults = {
                container: '.carousel',
                previousBtn: '.carousel-prev',
                nextBtn: '.carousel-next',
                wrapper: '.carousel-wrap',
                item: '.carousel-item',
                overlay: '#carousel-overlay',
                overlayClose: '#carousel-overlay-
close',
                overlayProduct: '#carousel-overlay-
product'
            };
            this.options = Object.assign(defaults,
settings);
            this.container =
document.querySelector(this.options.container);
            this.previousBtn =
this.container.querySelector(this.options.previousBtn
);
            this.nextBtn =
this.container.querySelector(this.options.nextBtn);
            this.wrapper =
this.container.querySelector(this.options.wrapper);
            this.items =
this.container.querySelectorAll(this.options.item);
            this.loader =
```

```
document.getElementById('loader');
            this.overlay =
document.querySelector(this.options.overlay);
            this.overlayClose =
document.querySelector(this.options.overlayClose);
            this.overlayProduct =
document.querySelector(this.options.overlayProduct);
            this.pages = 0;
            this.index = 0;
            this.images = [];

            if(this.container !== null) {
                this.init();
            }
        }

        init() {
            this.setVisibleItems();
            this.setPages();
            this.handleHideOverlay();
            this.onResize();
            this.next();
            this.previous();
            this.getDataItems();
        }
}
```

L'oggetto `settings` passato al costruttore conterrà nelle sue proprietà i selettori CSS che ci permetteranno di creare i riferimenti agli elementi del DOM. L'unica eccezione è l'indicatore di caricamento, che in una situazione reale sarebbe un componente esterno al nostro carousel.

`settings` verrà poi combinato con il metodo `Object.assign()` che permetterà di avere l'oggetto finale `options`. Questo oggetto consente di

fatto di modificare i selettori nel codice HTML e CSS mantenendo intatto il funzionamento del carousel.

`pages` conterrà il numero di pagine in cui verrà diviso il carousel a partire dal numero totale di elementi. Se all'inizio infatti abbiamo 24 elementi, e gli elementi visibili ad ogni scorrimento su un grande schermo sono 6, avremo 24 / 6 pagine, ossia 4. Questo valore sarà diverso su altre risoluzioni di schermo.

`index` è l'indice che verrà incrementato o decrementato di 1 ad ogni click sui pulsanti di navigazione e tiene traccia della pagina corrente del carousel su cui ci troviamo dopo ogni scorrimento.

`images` è un array che memorizza gli URL delle immagini caricate dalle API remote dopo la chiamata AJAX iniziale.

Il primo metodo invocato all'interno del metodo di inizializzazione `init()` è `setVisibleItems()`.

```
resetVisibleItems() {
            const visibles =
this.container.querySelectorAll('.carousel-item-
visible');
            if(visibles.length === 0) {
                return false;
            }
            for(const visible of visibles) {
                visible.classList.remove('carousel-
item-visible');
            }
        }

        setVisibleItems() {
            if(isMobile() && getViewportWidth() <=
768) {
```

```
                return false;
            }
            this.resetVisibleItems();
            for(const item of this.items) {
                if(isInViewport(item)) {
                    item.classList.add('carousel-
item-visible');
                }
            }
        }
```

Questo metodo e il suo helper `resetVisibleItems()` aggiungono o rimuovono la classe CSS `carousel-item-visible` che ha il solo scopo di contrassegnare nel DOM quegli elementi del carousel che sono visibili nella finestra del browser. Tuttavia, se ci troviamo su un dispositivo mobile con una risoluzione inferiore o uguale a 768 pixel, avremo un unico elemento visibile e quindi la logica principale del metodo non può essere applicata.

La classe CSS aggiunta serve principalmente a ottenere il numero di pagine in cui è suddiviso il carousel.

```
setPages() {
            this.pages = this.getPages();
        }

        getPages() {
            const visibles =
this.container.querySelectorAll('.carousel-item-
visible');
            if(visibles.length === 0) {
                return this.items.length;
            }
            return Math.floor(this.items.length /
```

```
visibles.length);
        }
```

Il numero di pagine, come già detto, si ottiene dividendo il numero complessivo di elementi per il numero di elementi visibili, ossia quegli elementi che di volta in volta ad ogni scorrimento avranno la classe CSS `.carousel-item-visible`.

Ora dobbiamo gestire la comparsa e la scomparsa della finestra modale.

```
showOverlay() {
            this.overlay.style.display = 'block';
            this.overlay.style.opacity = 1;
        }

        hideOverlay() {
            this.overlay.style.opacity = 0;
            setTimeout(() => {
                this.overlay.style.display = 'none';
            }, 400);
        }

        handleHideOverlay() {

this.overlayClose.addEventListener('click', e => {
                e.preventDefault();
                this.hideOverlay();
            }, false);
            this.overlay.addEventListener('click',
evt => {
                const targetElement = evt.target;

if(targetElement.matches(this.options.overlay)) {
                    const click = new Event('click');
```

```
this.overlayClose.dispatchEvent(click);
                }
            }, false);
        }
```

L'azione viene registrata principalmente sul pulsante di chiusura posto in alto a destra della finestra. Si tratta semplicemente di gestire le proprietà CSS `display` e `opacity` e il ritardo nella transizione CSS. Per permettere all'utente di chiudere la finestra senza usare necessariamente il pulsante di chiusura, usiamo la event delegation sulla finestra stessa: se l'utente ha cliccato in un punto qualsiasi al di fuori del contenuto principale, inneschiamo l'evento `click` registrato sul pulsante di chiusura. Il punto fondamentale da comprendere è che l'evento principale sul pulsante di chiusura deve essere registrato **prima** della routine con la event delegation.

A livello responsive, dobbiamo ricalcolare il numero di elementi visibili e la paginazione ogni qualvolta viene modificata la dimensione della finestra del browser.

```
onResize() {
            window.addEventListener('resize', () => {
                this.setVisibleItems();
                this.setPages();
            }, false);
        }
```

Ora possiamo definire il comportamento del carousel quando si clicca sul pulsante "Successivo".

```
getOffset(index, width) {
            return Math.round(index * width);
        }
```

```
next() {
            const self = this;
            self.nextBtn.addEventListener('click', ()
=> {
                const pageWidth =
self.getVisibleItemsWidth();
                self.index++;
                if(self.index > self.pages) {
                    self.index = 1;
                }
                slideTo(self.wrapper,
self.getOffset(self.index,pageWidth)).then(() => {
                    self.setVisibleItems();
                });
            }, false);
        }

        getVisibleItemsWidth() {
            if(isMobile() && getViewportWidth() <=
768) {
                const item = this.items[0];
                const rectItem =
item.getBoundingClientRect();
                return Math.round(rectItem.width);
            }
            const visibles =
this.container.querySelectorAll('.carousel-item-
visible');
            if(visibles.length === 0) {
                return 0;
            }
            let width = 0;
            for(const visible of visibles) {
                const rect =
```

```
visible.getBoundingClientRect();
                width += rect.width;
            }
            return Math.round(width);
        }
```

L'offset per lo scorrimento orizzontale viene calcolato moltiplicando l'indice corrente delle pagine con la larghezza totale degli elementi visibili. Tale larghezza si ottiene sommando le larghezze calcolate di tutti gli elementi con classe CSS `.carousel-item-visible`. Su mobile e con una risoluzione inferiore o uguale a 768 pixel, il calcolo viene effettuato prendendo in considerazione solo il primo elemento con classe `.carousel-item-visible`. Qui l'indice delle pagine viene incrementato di 1 e resettato a 1 se supera il numero di pagine disponibili.

La logica del pulsante `Precedente` è identica, ma qui l'indice viene decrementato di 1 e resettato a 0 se è inferiore o uguale a 0.

```
previous() {
            const self = this;
            self.previousBtn.addEventListener('click',
() => {
                const pageWidth =
self.getVisibleItemsWidth();
                self.index--;
                if(self.index <= 0) {
                    self.index = 0;
                }
                slideTo(self.wrapper,
self.getOffset(self.index,pageWidth)).then(() => {
                    self.setVisibleItems();
                });
```

```
            }, false);
        }
```

`getDataItems()`, infine, effettua tutte le operazioni asincrone sul nostro carousel, principalmente quelle relative alla chiamata API ed al preload delle immagini remote.

```
setDataItems(data) {
            if(!Array.isArray(data.products) ||
data.products.length === 0) {
                return false;
            }
            const { products } = data;
            this.items.forEach((item, index) => {
                let caption =
document.createElement('div');
                caption.className = 'carousel-
caption';
                let product = products[index];
                let { title, description, thumbnail,
images, price } = product;
                let displayPrice =
price.toFixed(2).replace('.', ',');
                this.images.push({thumbnail, image:
images.length > 0 ? images[0] : thumbnail });
                caption.innerHTML = `<span
class="carousel-title"><span>${title}</span>
<span>&euro;${displayPrice}</span></span><span
class="carousel-desc">${description}</span>`;
                item.appendChild(caption);
            });
        }
        preloadImages() {
            if(this.images.length === 0) {
```

```
                return false;
            }
            const tasks = [];
            for(const img of this.images) {
                let { thumbnail, image } = img;
                tasks.push(preloadImage(thumbnail));
                tasks.push(preloadImage(image));
            }
            return Promise.all(tasks);
        }

        handleItemsClick() {
            for(const item of this.items) {
                item.addEventListener('click', () =>
{
                    const title =
item.querySelector('.carousel-title');
                    const img =
item.querySelector('img');
                    const childs =
title.querySelectorAll('span');
                    const price =
childs[1].innerHTML;
                    let content = `<figure><img
src="${img.dataset.main}"></figure>`;
                    content += `<div class="carousel-
product-details"><span>${price}</span><button>Buy
Product</button></div>`;
                    this.overlayProduct.innerHTML =
content;
                    this.showOverlay();
                }, false);
            }
        }
```

```
        setItemImages() {
            if(this.images.length === 0) {
                return false;
            }
            this.items.forEach((item, index) => {
                let src = typeof this.images[index]
=== 'object' ? this.images[index] : null;
                if(src !== null) {
                    let img =
item.querySelector('img');
                    img.setAttribute('src',
src.thumbnail);
                    img.setAttribute('data-main',
src.image);
                }
            });
        }

        hideLoader() {
            this.loader.style.opacity = 0;
            setTimeout(() => {
                this.loader.style.display = 'none';
            }, 400);
        }

getDataItems() {
            const self = this;
            getData().then(items => {
                self.setDataItems(items);
                self.preloadImages().then(srcs => {
                    self.handleItemsClick();
                    self.setItemImages();
                    self.hideLoader();
```

```
            }).catch(e => {
                self.hideLoader();
            });
        }).catch(err => console.log(err));
    }
```

Questo metodo, e i metodi accessori, diventano di fatto inutili se il carousel non necessita di avere i propri contenuti caricati in modo asincrono tramite API remote. Ovviamente il preload delle immagini, la gestione della finestra modale e dell'indicatore di caricamento dovranno essere estratti dall'interno del metodo e inseriti nelle posizioni e nell'ordine che gli sono propri.

Demo

JavaScript Product Carousel