Creating A Client-Side Javascript Shopping Cart

By Gabriele Romanato

HTML, JavaScript, jQuery with 51 Comments February 13th, 2014

Session storage is a new feature introduced by the W3C's "Web Storage[1]" specification. It's supported in Internet Explorer 8+, Firefox, Chrome, Safari and Opera Desktop (for a complete list, please consult "Can I Use[2]"). In this series of articles, we'll cover in depth a practical implementation of session storage by creating a complete e-commerce shopping cart with the `sessionStorage` object and jQuery.

Bear in mind that, in these articles, I'm not going to propose a new technique to replace existing server-side techniques, but rather just a proof of concept of session storage.

Session Storage: A Quick Reminder

We use sessions to store data and share such data across several pages. Usually, a user would pick a product, and we'd save the product's name along with the chosen quantity and price.

Then, the user would fill out a form with their personal information, and we'd save it in the current session before the end of the process, which is typically the checkout page and the subsequent redirection to the payment gateway (for example, PayPal).

Further Reading on SmashingMag:

- Fundamental Guidelines Of E-Commerce Checkout Design[3]
- Reducing Abandoned Shopping Carts In E-Commerce[4]
- Local Storage And How To Use It On Websites[5]
- A Little Journey Through (Small And Big) E-Commerce Websites[6]

How are shopping carts built? PHP, for instance, makes frequent use of associative arrays to create the basic structure of a shopping cart. Associative arrays enable PHP Web developers to keep session data structured and organized.

JavaScript sessions work differently. Generally, a session expires when the user closes their browser (but bear in mind that the concept of "closing a browser" is not clear on mobile devices). When a session expires, all data stored in the session storage of a Web browser is removed. There's no need to explicitly initialize a session because in JavaScript a session takes the form of the global `sessionStorage` object and is always present. It's up to us to write data into the current session.

Session data comes in the form of key-value pairs, and the value of each key may contain only strings. To write data, we can use the `sessionStorage.setItem( name, value )` method:

```
sessionStorage.setItem( "total", 120 );
```

In this case, the key named `total` now contains the value `120` as a string, although we've used an integer in our call to the `.setItem()` method. This value will be available until the session expires, unless we use `sessionStorage.removeItem( "total" )` to remove the named key or we call `sessionStorage.clear()` to entirely remove all keys and values from the session storage.

Note that when a key doesn't exist in session storage, its value is always `null`. Then, when we remove a key from session storage and try again to get its value, we'd simply get `null`.

As you may have guessed, our key now is always available, even as the user navigates the pages of our website. To get its value, we simply write the following:

```
var total = sessionStorage.getItem( "total" );
console.log( total ); // '120', a string
```

We can also update its value by using `sessionStorage.setItem()` again with a new value:

```
var total = parseInt( sessionStorage.getItem( "total" ) );
var quantity = 2;
var updatedTotal = total * quantity;
sessionStorage.setItem( "total", updatedTotal ); // '240', a string
```

Now, the key named `total` has a value of `240` with our last update. Why did we call `parseInt()`? This is a simple technique to convert a numerical string into a true number, ensuring that our calculation will be consistent. Remember that all values in session storage are strings, and our calculations must be between only numbers.

But wait! What about objects? Objects may be stored in session storage by first turning them into JSON strings (with `JSON.stringify()`) and then back into JavaScript objects (with `JSON.parse()`):

```
var cart = {
    item: "Product 1",
    price: 35.50,
    qty: 2
};
var jsonStr = JSON.stringify( cart );
sessionStorage.setItem( "cart", jsonStr );
// now the cart is {"item":"Product 1","price":35.50,"qty":2}
var cartValue = sessionStorage.getItem( "cart" );
var cartObj = JSON.parse( cartValue );
// original object
```

To update our object, we simply extend it and then repeat the procedure above.

Security Considerations

Security is important. If we read the security notes[7] of the W3C's specification, then we'd be aware of the security risks of even a client-side technology such as Web storage.

The US Computer Emergency Readiness Team's technical paper on website security[8] (PDF) clearly states:

> *"Every community organization, corporation, business, or government agency relies on an outward-facing website to provide information about themselves, announce an event, or sell a product or service. Consequently, public-facing websites are often the most targeted attack vectors for malicious activity."*

Even if a browser session ends when the browser itself is closed, malicious attacks can still take place, especially if the browser has been compromised by certain exploits. Moreover, compromised websites can often be used to spread malware that targets particular browsers.

For this reason, **make sure your website is safe** before relying on any technique to store data in the browser. Keeping a website safe is beyond the scope of this article, but by simply following security best practices, you should be able to benefit from Web storage without worrying too much about its security implications.

Our Sample Project: Winery

Our sample project is an online store that sells wine. It's a simple e-commerce website whose only complication is in how its shipping charges are calculated.

In short, wines are sold in packages of six bottles. This means that the total quantity of bottles sold must always be in multiples of six. Shipping charges are calculated, then, according to the total quantity of bottles sold.

Our store will rely on PayPal, so we'll have to create a Business account in PayPal Sandbox to test our code.

The user may add and remove products from their shopping cart, update the cart, change the quantity of each product, and empty the cart. They have to fill a form with their contact information, specifying whether their billing address is the same as their shipping address.

Before being redirected to PayPal, the user will see a summary page with their personal data, their cart, and the cart's total price plus shipping charges.

After completing their purchase, the user should be redirected back to our website. This is **the only step of the process that we can't handle only with JavaScript**. PayPal will send back various data over an HTTP request that has to be processed with a server-side language (such as PHP). If you need more information to get started with this kind of processing, please consult PayPal's tutorial[9].

HTML Structure

Our project is made up of the following sections:

- `index.html`
  This contains the list from which users may add products to their shopping cart, specifying the quantity for each product.
- `cart.html`
  This is the shopping cart page where users may update or empty their cart. Alternatively, they can go back to the main page to continue shopping or proceed to the checkout page.
- `checkout.html`
  On this page, users fill out a form with their personal information — specifically, their billing and shipping addresses.
- `order.html`
  This page contains a brief summary of the user's order plus the PayPal form. Once a user submits the form, they will be redirected to PayPal's landing page.

We'll go over the markup for this project in the following sections.

INDEX.HTML

The main components of this page are the forms that enable the user to add products to their shopping cart.

```
<div class="product-description" data-name="Wine #1" data-price="5">
    <h3 class="product-name">Wine #1</h3>
        <p class="product-price">&euro; 5</p>
        <form class="add-to-cart" action="cart.html" method="post">
            <div>
                <label for="qty-1">Quantity</label>
                <input type="text" name="qty-1" id="qty-1" class="qty" value="1" />
            </div>
            <p><input type="submit" value="Add to cart" class="btn" /></p>
        </form>
</div>
```

The data attributes used here for storing product names and prices can be accessed via jQuery using the .data()[10] and $.data()[11] methods.

CART.HTML

Our shopping cart page is made up of three components: a table with the product's information, an element that displays the subtotal, and a list of cart actions.

```
<form id="shopping-cart" action="cart.html" method="post">
    <table class="shopping-cart">
        <thead>
            <tr>
                <th scope="col">Item</th>
                <th scope="col">Qty</th>
                <th scope="col">Price</th>
            </tr>
        </thead>
        <tbody></tbody>
    </table>
    <p id="sub-total">
        <strong>Sub Total</strong>: <span id="stotal"></span>
    </p>
    <ul id="shopping-cart-actions">
        <li>
            <input type="submit" name="update" id="update-cart" class="btn" value="Update
        </li>
        <li>
            <input type="submit" name="delete" id="empty-cart" class="btn" value="Empty C
        </li>
        <li>
            <a href="index.html" class="btn">Continue Shopping</a>
        </li>
        <li>
            <a href="checkout.html" class="btn">Go To Checkout</a>
        </li>
    </ul>
</form>
```

The table contained in this page is empty, and we'll fill it with data via JavaScript. The element that displays the subtotal works just as a placeholder for JavaScript. The first two actions, "Update Cart" and "Empty Cart," will be handled by JavaScript, while the latter two actions are just plain links to the product's list page and the checkout page, respectively.

CHECKOUT.HTML

This page has four components:

- a table that shows the ordered items (the same table shown earlier in the shopping cart section), plus the final price and shipping charges;
- a form in which the user must fill in their billing details;
- a form with shipping information;
- a checkbox to enable the user to specify that their billing details are the same as their shipping details.

```
<table id="checkout-cart" class="shopping-cart">
    <thead>
        <tr>
            <th scope="col">Item</th>
            <th scope="col">Qty</th>
            <th scope="col">Price</th>
        </tr>
    </thead>
    <tbody>

    </tbody>
</table>

<div id="pricing">
    <p id="shipping">
        <strong>Shipping</strong>: <span id="sshipping"></span>
    </p>

    <p id="sub-total">
        <strong>Total</strong>: <span id="stotal"></span>
    </p>
</div>

<form action="order.html" method="post" id="checkout-order-form">
    <h2>Your Details</h2>
        <fieldset id="fieldset-billing">
            <legend>Billing</legend>
                <!-- Name, Email, City, Address, ZIP Code, Country (select box) -->

<div>
    <label for="name">Name</label>
    <input type="text" name="name" id="name" data-type="string" data-message="This field
</div>

<div>
    <label for="email">Email</label>
    <input type="text" name="email" id="email" data-type="expression" data-message="Not a
</div>

<div>
    <label for="city">City</label>
    <input type="text" name="city" id="city" data-type="string" data-message="This field
</div>

<div>
    <label for="address">Address</label>
        <input type="text" name="address" id="address" data-type="string" data-message="T
</div>

<div>
    <label for="zip">ZIP Code</label>
    <input type="text" name="zip" id="zip" data-type="string" data-message="This field ma
</div>

<div>
    <label for="country">Country</label>
        <select name="country" id="country" data-type="string" data-message="This field m
            <option value="">Select</option>
            <option value="US">USA</option>
            <option value="IT">Italy</option>
        </select>
</div>
</fieldset>
```

```
<div id="shipping-same">Same as Billing <input type="checkbox" id="same-as-billing" value

<fieldset id="fieldset-shipping">
<legend>Shipping</legend>
    <!-- Same fields as billing -->
</fieldset>

<p><input type="submit" id="submit-order" value="Submit" class="btn" /></p>

</form>
```

Data attributes are used here for validation. The `data-type` attribute specifies the type of data we're validating, and `data-message` contains the error message to be shown in case of failure.

I didn't use the email validation built into Web browsers just for the sake of simplicity, but you could use it if you want.

ORDER.HTML

This final page contains a brief recap of the user's order, their details and the PayPal form.

```
<h1>Your Order</h1>

<table id="checkout-cart" class="shopping-cart">
    <thead>
        <tr>
            <th scope="col">Item</th>
            <th scope="col">Qty</th>
            <th scope="col">Price</th>
        </tr>
    </thead>
    <tbody>
    </tbody>
</table>

<div id="pricing">
    <p id="shipping">
        <strong>Shipping</strong>: <span id="sshipping"></span>
    </p>

    <p id="sub-total">
        <strong>Total</strong>: <span id="stotal"></span>
    </p>
</div>

<div id="user-details">
    <h2>Your Data</h2>
        <div id="user-details-content"></div>
</div>


<form id="paypal-form" action="" method="post">
    <input type="hidden" name="cmd" value="_cart" />
    <input type="hidden" name="upload" value="1" />
    <input type="hidden" name="business" value="" />

    <input type="hidden" name="currency_code" value="" />
    <input type="submit" id="paypal-btn" class="btn" value="Pay with PayPal" />
</form>
```

The PayPal form and other elements of this page are initially empty, except for those fields that don't need to be generated dynamically.

JavaScript Code

The CSS layout of this project will have no actual influence on the goal we want to achieve. Even if we disabled CSS entirely, the project would continue to function, thanks to the strong relationship between the HTML's structure and the JavaScript's behavior.

We'll use an **object-oriented approach** because of the complexity of our goals. Our object will be based on a simple constructional pattern and will use both private and public methods.

OBJECT STRUCTURE

Our object has a very simple structure. The constructor function both initializes the top-level element that wraps our DOM's entire structure and invokes the initialization method.

```
(function( $ ) {
    $.Shop = function( element ) {
        this.$element = $( element ); // top-level element
        this.init();
    };

    $.Shop.prototype = {
        init: function() {
            // initializes properties and methods
        }
    };

    $(function() {
        var shop = new $.Shop( "#site" ); // object's instance
    });

})( jQuery );
```

The object's instance is created when the DOM is ready. We can test that everything has worked fine as follows:

```
$(function() {
    var shop = new $.Shop( "#site" );
    console.log( shop.$element );
});
```

This outputs the following:

```
x.fn.x.init[1]
    0: div#site
    context: document
    length: 1
    selector: "#site"
```

Now that we know our object has been instantiated correctly, we can define its properties.

OBJECT PROPERTIES

The properties of our object break down into two categories: first, the properties for handling calculations, forms and validation, and secondly, the references to HTML elements.

```
$.Shop.prototype = {
    init: function() {
        // Properties

            this.cartPrefix = "winery-"; // prefix string to be prepended to the cart's n
            this.cartName = this.cartPrefix + "cart"; // cart's name in session storage
            this.shippingRates = this.cartPrefix + "shipping-rates"; // shipping rates ke
            this.total = this.cartPrefix + "total"; // total key in the session storage
            this.storage = sessionStorage; // shortcut to sessionStorage object


            this.$formAddToCart = this.$element.find( "form.add-to-cart" ); // forms for
            this.$formCart = this.$element.find( "#shopping-cart" ); // Shopping cart for
            this.$checkoutCart = this.$element.find( "#checkout-cart" ); // checkout form
            this.$checkoutOrderForm = this.$element.find( "#checkout-order-form" ); // ch
            this.$shipping = this.$element.find( "#sshipping" ); // element that displays
            this.$subTotal = this.$element.find( "#stotal" ); // element that displays th
            this.$shoppingCartActions = this.$element.find( "#shopping-cart-actions" ); /
            this.$updateCartBtn = this.$shoppingCartActions.find( "#update-cart" ); // up
            this.$emptyCartBtn = this.$shoppingCartActions.find( "#empty-cart" ); // empt
            this.$userDetails = this.$element.find( "#user-details-content" ); // element
            this.$paypalForm = this.$element.find( "#paypal-form" ); // PayPal form


            this.currency = "&euro;"; // HTML entity of the currency to be displayed in l
            this.currencyString = "€"; // currency symbol as text string
            this.paypalCurrency = "EUR"; // PayPal's currency code
            this.paypalBusinessEmail = "yourbusiness@email.com"; // your PayPal Business
            this.paypalURL = "https://www.sandbox.paypal.com/cgi-bin/webscr"; // URL of t

            // object containing patterns for form validation
            this.requiredFields = {
                expression: {
                    value: /^([w-.]+)@((?:[w]+.)+)([a-z]){2,4}$/
                },

                str: {
                    value: ""
                }

            };

            // public methods invocation
    }
};
```

Let’s go over these properties one by one.

Storage and other properties:

- `cartPrefix`
  A prefix to be prepended to the cart’s name key in session storage
- `cartName`
  The cart’s name key in session storage (combines the `cartPrefix` string with the `cart` string)

- `shippingRates`
  The shipping rate key in session storage
- `total`
  The total's key in session storage
- `storage`
  Shortcut to the `sessionStorage` object.
- `currency`
  An HTML entity used to display the current currency in the layout
- `currencyString`
  The current currency symbol used in the element's text
- `paypalCurrency`
  PayPal's currency text code
- `paypalBusinessEmail`
  The email address of your PayPal Business account
- `paypalURL`
  The URL of PayPal's form (defaults to the URL of PayPal Sandbox)
- `requiredFields`
  An object containing the patterns and rules for form validation

References to elements:

- `$formAddToCart`
  The forms for adding products to the shopping cart
- `$formCart`
  The shopping cart form
- `$checkoutCart`
  The checkout's shopping cart form
- `$checkoutOrderForm`
  The checkout's form where users input their personal information
- `$shipping`
  The element that contains and displays shipping rates
- `$subTotal`
  The element that contains and displays the total charges

- `$shoppingCartActions`
  The elements that contain the actions related to the shopping cart
- `$updateCartBtn`
  The button to update the shopping cart
- `$emptyCartBtn`
  The button for emptying the cart
- `$userDetails`
  The element that contains and displays the information entered by the user
- `$paypalForm`
  PayPal's form

All of the elements are prefixed with the `$` sign, meaning that they're jQuery objects. But **not all of these elements are available on all pages**. To check whether a jQuery element exists, simply test its `length` property:

```
if( $element.length ) {
    // the element exists
}
```

Another approach, not used in our project, is to add a particular ID or class to the `body` element and perform actions conditionally:

```
var $body = $( "body" ),
    page = $body.attr( "id" );

    switch( page ) {
        case "product-list":
            // actions for handling products
            break;
        case "shopping-cart":
            // actions for handling the shopping cart
            break;
        case "checkout":
            // actions for handling the checkout's page
            break;
        default:
            break;
    }
```

OBJECT METHODS

The actions of our code take place in our object's methods, which, in turn, can be divided into public and private methods. Private methods operate in the background, so to speak, and help the public methods perform their tasks. These methods are prefixed with an underscore and are never used directly.

Public methods, meanwhile, operate directly on page elements and data, and they're unprefixed. We've already seen the `init()` method, which simply initializes properties and other public methods in the object's constructor function. The other methods will be explained below.

PRIVATE METHODS (HELPERS)

The first private method, `_emptyCart()`, simply empties the current session storage in the browser:

```
$.Shop.prototype = {
    // empties session storage

    _emptyCart: function() {
        this.storage.clear();
    }
};
```

To format a number by a set number of decimal places, we implement the `_formatNumber()` method:

```
/* Format a number by decimal places
 * @param num Number the number to be formatted
 * @param places Number the decimal places
 * @returns n Number the formatted number
*/


_formatNumber: function( num, places ) {
    var n = num.toFixed( places );
    return n;
}
```

This method makes use of JavaScript's toFixed()[12] method of the `Number` object. Its role in our project is to properly format prices.

Because **not all of the prices in our pages are contained in data attributes**, we need a specialized method to extract the numeric portion of a string from text nodes. This method is named `_extractPrice()`:

```
/* Extract the numeric portion from a string
 * @param element Object the jQuery element that contains the relevant string
 * @returns price String the numeric string
 */


_extractPrice: function( element ) {
    var self = this;
    var text = element.text();
    var price = text.replace( self.currencyString, "" ).replace( " ", "" );
    return price;
}
```

Above, `self` is a reference to the `$.Shop` object, and we'll need it every time we want to access a property or a method of our object without worrying much about scope.

You can bulletproof this method by adding a further routine that strips out all trailing white space:

```
var text = $.trim( element.text() );
```

Bear in mind that jQuery's $.trim()[13] method removes all new lines, spaces (including non-breaking spaces) and tabs from the beginning and end of a string. If these white space characters occur in the middle of a string, they are preserved.

Then, **we need two methods to convert strings into numbers and numbers into strings**. This is necessary to perform calculations and to display the results on our pages.

```
/* Converts a numeric string into a number
 * @param numStr String the numeric string to be converted
 * @returns num Number the number, or false if the string cannot be converted
 */

_convertString: function( numStr ) {
    var num;
    if( /^[-+]?[0-9]+.[0-9]+$/.test( numStr ) ) {
        num = parseFloat( numStr );
    } else if( /^d+$/.test( numStr ) ) {
        num = parseInt( numStr );
    } else {
        num = Number( numStr );
    }

    if( !isNaN( num ) ) {
        return num;
    } else {
        console.warn( numStr + " cannot be converted into a number" );
        return false;
    }
},

/* Converts a number to a string
 * @param n Number the number to be converted
 * @returns str String the string returned
 */

_convertNumber: function( n ) {
    var str = n.toString();
    return str;
}
```

Above, `_convertString()` runs the following tests:

1. Does the string have a decimal format? If so, it uses the parseFloat()[14] function.
2. Does the string have an integer format? If so, it uses the parseInt()[15] function.
3. If the format of the string cannot be detected, it uses the Number()[16] constructor.
4. If the result is a number (tested with the isNaN()[17] function), it returns the number. Otherwise, it outputs a warning to the JavaScript console and returns `false`.

By contrast, `_convertNumber()` simply invokes the toString()[18] method to convert a number into a string.

The next step is to define two methods to **convert a JavaScript object into a JSON string** and a JSON string back into a JavaScript object:

```
/* Converts a JSON string to a JavaScript object
 * @param str String the JSON string
 * @returns obj Object the JavaScript object
 */

_toJSONObject: function( str ) {
    var obj = JSON.parse( str );
    return obj;
},

/* Converts a JavaScript object to a JSON string
 * @param obj Object the JavaScript object
 * @returns str String the JSON string
 */


_toJSONString: function( obj ) {
    var str = JSON.stringify( obj );
    return str;
}
```

The first method makes use of the `JSON.parse()` method, while the latter invokes the `JSON.stringify()` method (see Mozilla Developer Network's article on "Using Native JSON[19]").

Why do we need these methods? Because our cart will also store the information related to each product using the following data format (spaces added for legibility):

Key	Value
`winery-cart`	`{ "items": [ { "product": "Wine #1", "qty": 5, "price": 5 } ] }`

The `winery-cart` key contains a JSON string that represents an array of objects (i.e. `items`) in which each object shows the relevant information about a product added by the user — namely, the product's name, the quantity and the price.

It's pretty obvious that we also now need a specialized method to add items to this particular key in session storage:

```
/* Add an object to the cart as a JSON string
 * @param values Object the object to be added to the cart
 * @returns void
 */


_addToCart: function( values ) {
    var cart = this.storage.getItem( this.cartName );
    var cartObject = this._toJSONObject( cart );
    var cartCopy = cartObject;
    var items = cartCopy.items;
    items.push( values );

    this.storage.setItem( this.cartName, this._toJSONString( cartCopy ) );
}
```

This method gets the cart's key from session storage, converts it to a JavaScript object and adds a new object as a JSON string to the cart's array. The newly added object has the following format:

```
this._addToCart({
    product: "Test",
    qty: 1,
    price: 2
});
```

Now, our cart key will look like this:

Key	Value
`winery-cart`	`{ "items": [ { "product": "Wine #1", "qty": 5, "price": 5 }, { "product": "Test", "qty": 1, "price": 2 } ] }`

Shipping is calculated according to the overall number of products added to the cart, not the quantity of each individual product:

```
/* Custom shipping rates calculated based on total quantity of items in cart
 * @param qty Number the total quantity of items
 * @returns shipping Number the shipping rates
 */

_calculateShipping: function( qty ) {
    var shipping = 0;
    if( qty >= 6 ) {
        shipping = 10;
    }
    if( qty >= 12 && qty <= 30 ) {
        shipping = 20;
    }

    if( qty >= 30 && qty <= 60 ) {
        shipping = 30;
    }

    if( qty > 60 ) {
        shipping = 0;
    }

    return shipping;

}
```

You can replace this method's routines with your own. In this case, shipping charges are calculated based on specific amounts.

We also need to validate the checkout form where users insert their personal information. The following method takes into account the special visibility toggle by which the user may specify that their billing information is the same as their shipping information.

```
/* Validates the checkout form
 * @param form Object the jQuery element of the checkout form
 * @returns valid Boolean true for success, false for failure
 */


_validateForm: function( form ) {
        var self = this;
        var fields = self.requiredFields;
        var $visibleSet = form.find( "fieldset:visible" );
        var valid = true;

        form.find( ".message" ).remove();

    $visibleSet.each(function() {

        $( this ).find( ":input" ).each(function() {
        var $input = $( this );
        var type = $input.data( "type" );
        var msg = $input.data( "message" );

        if( type == "string" ) {
            if( $input.val() == fields.str.value ) {
                $( "<span class='message'/>" ).text( msg ).
                insertBefore( $input );

                valid = false;
            }
        } else {
            if( !fields.expression.value.test( $input.val() ) ) {
                $( "<span class='message'/>" ).text( msg ).
                insertBefore( $input );

                valid = false;
            }
        }

    });
    });

    return valid;
}
```

When validation messages are added upon the form being submitted, we need to clear these messages before going any further. In this case, we take into account only the fields contained in a `fieldset` element that is still visible after the user has checked the visibility toggle.

Validation takes place by checking whether the current field requires a simple string comparison (`data-type="string"`) or a regular expression test (`data-type="expression"`). Our tests are based on the `requiredFields` property. If there's an error, we'll show a message by using the `data-message` attribute of each field.

Note that the validation routines used above have been inserted just for demonstration purposes, and they have several flaws. For better validation, I recommend a dedicated jQuery plugin, such as jQuery Validation[20].

Last but not least is **registering the information that the user has entered** in the checkout form:

```
/* Save the data entered by the user in the checkout form
 * @param form Object the jQuery element of the checkout form
 * @returns void
 */


_saveFormData: function( form ) {
    var self = this;
    var $visibleSet = form.find( "fieldset:visible" );

    $visibleSet.each(function() {
        var $set = $( this );
        if( $set.is( "#fieldset-billing" ) ) {
            var name = $( "#name", $set ).val();
            var email = $( "#email", $set ).val();
            var city = $( "#city", $set ).val();
            var address = $( "#address", $set ).val();
            var zip = $( "#zip", $set ).val();
            var country = $( "#country", $set ).val();

            self.storage.setItem( "billing-name", name );
            self.storage.setItem( "billing-email", email );
            self.storage.setItem( "billing-city", city );
            self.storage.setItem( "billing-address", address );
            self.storage.setItem( "billing-zip", zip );
            self.storage.setItem( "billing-country", country );
        } else {
            var sName = $( "#sname", $set ).val();
            var sEmail = $( "#semail", $set ).val();
            var sCity = $( "#scity", $set ).val();
            var sAddress = $( "#saddress", $set ).val();
            var sZip = $( "#szip", $set ).val();
            var sCountry = $( "#scountry", $set ).val();

            self.storage.setItem( "shipping-name", sName );
            self.storage.setItem( "shipping-email", sEmail );
            self.storage.setItem( "shipping-city", sCity );
            self.storage.setItem( "shipping-address", sAddress );
            self.storage.setItem( "shipping-zip", sZip );
            self.storage.setItem( "shipping-country", sCountry );

        }
    });
}
```

Again, this method takes into account the visibility of the fields based on the user's choice. Once the form has been submitted, our session storage may have the following details added to it:

Key	Value
`billing-name`	John Doe
`billing-email`	jdoe@localhost
`billing-city`	New York
`billing-address`	Street 1
`billing-zip`	1234
`billing-country`	USA

PUBLIC METHODS

Our public methods are invoked in the initialization method (`init()`). The first thing to do is create the initial keys and values in session storage.

```
// Creates the cart keys in session storage

createCart: function() {
    if( this.storage.getItem( this.cartName ) == null ) {

        var cart = {};
        cart.items = [];

        this.storage.setItem( this.cartName, this._toJSONString( cart ) );
        this.storage.setItem( this.shippingRates, "0" );
        this.storage.setItem( this.total, "0" );
    }
}
```

The first check tests whether our values have already been added to session storage. We need this test because we could actually overwrite our values if we run this method every time a document has finished loading.

Now, our session storage looks like this:

Key	Value
`winery-cart`	{“items”:[]}
`winery-shipping-rates`	0
`winery-total`	0

Now, we need to **handle the forms where the user may add products** to their shopping cart:

```
// Adds items to shopping cart

handleAddToCartForm: function() {
    var self = this;
    self.$formAddToCart.each(function() {
        var $form = $( this );
        var $product = $form.parent();
        var price = self._convertString( $product.data( "price" ) );
        var name =  $product.data( "name" );

        $form.on( "submit", function() {
            var qty = self._convertString( $form.find( ".qty" ).val() );
            var subTotal = qty * price;
            var total = self._convertString( self.storage.getItem( self.total ) );
            var sTotal = total + subTotal;
            self.storage.setItem( self.total, sTotal );
            self._addToCart({
                product: name,
                price: price,
                qty: qty
            });
            var shipping = self._convertString( self.storage.getItem( self.shippingRates
            var shippingRates = self._calculateShipping( qty );
            var totalShipping = shipping + shippingRates;

            self.storage.setItem( self.shippingRates, totalShipping );
        });
    });
}
```

Every time a user submits one of these forms, we have to read the product quantity specified by the user and multiply it by the unit price. Then, we need to read the total's key contained in session storage and update its value accordingly. Having done this, we call the `_addToCart()` method to store the product's details in storage. The quantity specified will also be used to **calculate the shipping rate** by comparing its value to the value already stored.

Suppose that a user chooses the first product, Wine #1, whose price is €5.00, and specifies a quantity of 5. The session storage would look like this once the form has been submitted:

Key	Value
`winery-cart`	{“items”:[{“product”:”Wine #1″,”price”:5,”qty”:5}]}
`winery-shipping-rates`	0
`winery-total`	25

Suppose the same user goes back to the product list and chooses Wine #2, whose price is €8.00, and specifies a quantity of 2:

Key	Value
`winery-cart`	{“items”:[{“product”:”Wine #1″,”price”:5,”qty”:5},{“product”:”Wine #2″,”price”:8,”qty”:2}]}
`winery-shipping-rates`	0
`winery-total`	41

Finally, our eager user returns again to the product list, chooses Wine #3, whose price is €11.00, and specifies a quantity of 6:

Key	Value
`winery-cart`	{“items”:[{“product”:”Wine #1″,”price”:5,”qty”:5},{“product”:”Wine #2″,”price”:8,”qty”:2},{“product”:”Wine #3″,”price”:11,”qty”:6}]}
`winery-shipping-rates`	10
`winery-total`	107

At this point, we need to **accurately display the cart** when the user goes to the shopping cart page or checkout page:

```
// Displays the shopping cart

displayCart: function() {
    if( this.$formCart.length ) {
        var cart = this._toJSONObject( this.storage.getItem( this.cartName ) );
        var items = cart.items;
        var $tableCart = this.$formCart.find( ".shopping-cart" );
        var $tableCartBody = $tableCart.find( "tbody" );


        for( var i = 0; i < items.length; ++i ) {
            var item = items[i];
            var product = item.product;
            var price = this.currency + " " + item.price;
            var qty = item.qty;
            var html = "<tr><td class='pname'>" + product + "</td>" + "<td class='pqty'><

            $tableCartBody.html( $tableCartBody.html() + html );
        }

        var total = this.storage.getItem( this.total );
        this.$subTotal[0].innerHTML = this.currency + " " + total;
    } else if( this.$checkoutCart.length ) {
        var checkoutCart = this._toJSONObject( this.storage.getItem( this.cartName ) );
        var cartItems = checkoutCart.items;
        var $cartBody = this.$checkoutCart.find( "tbody" );

        for( var j = 0; j < cartItems.length; ++j ) {
            var cartItem = cartItems[j];
            var cartProduct = cartItem.product;
            var cartPrice = this.currency + " " + cartItem.price;
            var cartQty = cartItem.qty;
            var cartHTML = "<tr><td class='pname'>" + cartProduct + "</td>" + "<td class=

            $cartBody.html( $cartBody.html() + cartHTML );
        }

        var cartTotal = this.storage.getItem( this.total );
        var cartShipping = this.storage.getItem( this.shippingRates );
        var subTot = this._convertString( cartTotal ) + this._convertString( cartShipping

        this.$subTotal[0].innerHTML = this.currency + " " + this._convertNumber( subTot )
        this.$shipping[0].innerHTML = this.currency + " " + cartShipping;

    }
}
```

If the cart's table is on the shopping cart page, then this method iterates over the array of objects contained in the `winery-cart` key and populates the table by adding a text field to allow users to modify the quantity of each product. For the sake of simplicity, I didn't include an action to remove an item from the cart, but that procedure is pretty simple:

1. Get the `items` array, contained in session storage.
2. Get the product's name, contained in the `td` element with the `pname` class.

3. Create a new array by filtering out the item with the product's name, obtained in step 2 (you can use $.grep()[21]).
4. Save the new array in the `winery-cart` key.
5. Update the total and shipping charge values.

```
var items = [
    {
        product: "Test",
        qty: 1,
        price: 5
    },
    {
        product: "Foo",
        qty: 5,
        price: 10
    },
    {
        product: "Bar",
        qty: 2,
        price: 8
    }
];


items = $.grep( items, function( item ) {
    return item.product !== "Test";

});

console.log( items );

/*
    Array[2]
        0: Object
            price: 10
            product: "Foo"
            qty: 5
        1: Object
            price: 8
            product: "Bar"
            qty: 2
*/
```

Then, we need a method that updates the cart with a new quantity value for each product:

```
// Updates the cart

updateCart: function() {
        var self = this;
    if( self.$updateCartBtn.length ) {
        self.$updateCartBtn.on( "click", function() {
            var $rows = self.$formCart.find( "tbody tr" );
            var cart = self.storage.getItem( self.cartName );
            var shippingRates = self.storage.getItem( self.shippingRates );
            var total = self.storage.getItem( self.total );

            var updatedTotal = 0;
            var totalQty = 0;
            var updatedCart = {};
            updatedCart.items = [];

            $rows.each(function() {
                var $row = $( this );
                var pname = $.trim( $row.find( ".pname" ).text() );
                var pqty = self._convertString( $row.find( ".pqty > .qty" ).val() );
                var pprice = self._convertString( self._extractPrice( $row.find( ".pprice

                var cartObj = {
                    product: pname,
                    price: pprice,
                    qty: pqty
                };

                updatedCart.items.push( cartObj );

                var subTotal = pqty * pprice;
                updatedTotal += subTotal;
                totalQty += pqty;
            });

            self.storage.setItem( self.total, self._convertNumber( updatedTotal ) );
            self.storage.setItem( self.shippingRates, self._convertNumber( self._calcula
            self.storage.setItem( self.cartName, self._toJSONString( updatedCart ) );

        });
    }
}
```

Our method loops through all of the relevant table cells of the cart and builds a new object to be inserted in the `winery-cart` key. It also recalculates the total price and shipping charge by taking into account the newly inserted values of the quantity fields.

Suppose that a user changes the quantity of Wine #2 from 2 to 6:

Key	Value
`winery-cart`	{“items”:[{“product”:”Wine #1″,”price”:5,”qty”:5},{“product”:”Wine #2″,”price”:8,”qty”:6},{“product”:”Wine #3″,”price”:11,”qty”:6}]}
`winery-`	20

shipping-rates	
winery-total	139

If the user wants to empty their cart and start over, we simply have to add the following action:

```
// Empties the cart by calling the _emptyCart() method
// @see $.Shop._emptyCart()

emptyCart: function() {
    var self = this;
    if( self.$emptyCartBtn.length ) {
        self.$emptyCartBtn.on( "click", function() {
            self._emptyCart();
        });
    }
}
```

Now, session storage has been emptied entirely, and **the user may start making purchases again**. However, if they decide to finalize their order instead, then we need to handle the checkout form when they insert their personal information.

```
// Handles the checkout form by adding a validation routine and saving user’s info in ses

handleCheckoutOrderForm: function() {
    var self = this;
    if( self.$checkoutOrderForm.length ) {
        var $sameAsBilling = $( "#same-as-billing" );
        $sameAsBilling.on( "change", function() {
            var $check = $( this );
            if( $check.prop( "checked" ) ) {
                $( "#fieldset-shipping" ).slideUp( "normal" );
            } else {
                $( "#fieldset-shipping" ).slideDown( "normal" );
            }
        });

        self.$checkoutOrderForm.on( "submit", function() {
            var $form = $( this );
            var valid = self._validateForm( $form );

            if( !valid ) {
                return valid;
            } else {
                self._saveFormData( $form );
            }
        });
    }
}
```

The first thing we need to do is **hide the shipping fields** if the user checks the toggle that specifies that their billing information is the same as their shipping information. We use the `change` event, combined with jQuery's .prop()[22] method. (If you're curious about the difference between `.prop()` and `.attr()`, StackOverflow has a good discussion[23] of it.)

Then, we validate the form by returning a `false` value in case of errors, thus preventing the form from being submitted. If validation succeeds, we save the user's data in storage. For example:

Key	**Value**
`winery-cart`	{"items":[{"product":"Wine #1″,"price":5,"qty":5},{"product":"Wine #2″,"price":8,"qty":6},{"product":"Wine #3″,"price":11,"qty":6}]}
`winery-shipping-rates`	20
`winery-total`	139
`billing-name`	John Doe
`billing-email`	jdoe@localhost
`billing-city`	New York
`billing-address`	Street 1
`billing-zip`	1234
`billing-country`	USA

The final step is the page with the PayPal form. First, we need to display the user's information gathered on the checkout page:

```
// Displays the user's information

displayUserDetails: function() {
    if( this.$userDetails.length ) {
        if( this.storage.getItem( "shipping-name" ) == null ) {
            var name = this.storage.getItem( "billing-name" );
            var email = this.storage.getItem( "billing-email" );
            var city = this.storage.getItem( "billing-city" );
            var address = this.storage.getItem( "billing-address" );
            var zip = this.storage.getItem( "billing-zip" );
            var country = this.storage.getItem( "billing-country" );

            var html = "<div class='detail'>";
                html += "<h2>Billing and Shipping</h2>";
                html += "<ul>";
                html += "<li>" + name + "</li>";
                html += "<li>" + email + "</li>";
                html += "<li>" + city + "</li>";
                html += "<li>" + address + "</li>";
                html += "<li>" + zip + "</li>";
                html += "<li>" + country + "</li>";
                html += "</ul></div>";

            this.$userDetails[0].innerHTML = html;
        } else {
            var name = this.storage.getItem( "billing-name" );
            var email = this.storage.getItem( "billing-email" );
            var city = this.storage.getItem( "billing-city" );
            var address = this.storage.getItem( "billing-address" );
            var zip = this.storage.getItem( "billing-zip" );
            var country = this.storage.getItem( "billing-country" );

            var sName = this.storage.getItem( "shipping-name" );
            var sEmail = this.storage.getItem( "shipping-email" );
            var sCity = this.storage.getItem( "shipping-city" );
            var sAddress = this.storage.getItem( "shipping-address" );
            var sZip = this.storage.getItem( "shipping-zip" );
            var sCountry = this.storage.getItem( "shipping-country" );

            var html = "<div class='detail'>";
                html += "<h2>Billing</h2>";
                html += "<ul>";
                html += "<li>" + name + "</li>";
                html += "<li>" + email + "</li>";
                html += "<li>" + city + "</li>";
                html += "<li>" + address + "</li>";
                html += "<li>" + zip + "</li>";
                html += "<li>" + country + "</li>";
                html += "</ul></div>";

                html += "<div class='detail right'>";
                html += "<h2>Shipping</h2>";
                html += "<ul>";
                html += "<li>" + sName + "</li>";
                html += "<li>" + sEmail + "</li>";
                html += "<li>" + sCity + "</li>";
                html += "<li>" + sAddress + "</li>";
                html += "<li>" + sZip + "</li>";
                html += "<li>" + sCountry + "</li>";
                html += "</ul></div>";

            this.$userDetails[0].innerHTML = html;

        }
    }
}
```

Our method first **checks whether the user has inputted either billing or shipping information or both**. Then, it simply builds an HTML fragment by getting the user's data from session storage.

Finally, the user may buy the products by submitting the PayPal form. The form redirects them to PayPal, but the fields need to be filled in properly before the form can be submitted.

```
// Appends the required hidden values to PayPal's form before submitting

populatePayPalForm: function() {
    var self = this;
    if( self.$paypalForm.length ) {
        var $form = self.$paypalForm;
        var cart = self._toJSONObject( self.storage.getItem( self.cartName ) );
        var shipping = self.storage.getItem( self.shippingRates );
        var numShipping = self._convertString( shipping );
        var cartItems = cart.items;
        var singShipping = Math.floor( numShipping / cartItems.length );

        $form.attr( "action", self.paypalURL );
        $form.find( "input[name='business']" ).val( self.paypalBusinessEmail );
        $form.find( "input[name='currency_code']" ).val( self.paypalCurrency );

        for( var i = 0; i < cartItems.length; ++i ) {
            var cartItem = cartItems[i];
            var n = i + 1;
            var name = cartItem.product;
            var price = cartItem.price;
            var qty = cartItem.qty;

            $( "<div/>" ).html( "<input type='hidden' name='quantity_" + n + "' value='"
            insertBefore( "#paypal-btn" );
            $( "<div/>" ).html( "<input type='hidden' name='item_name_" + n + "' value=''
            insertBefore( "#paypal-btn" );
            $( "<div/>" ).html( "<input type='hidden' name='item_number_" + n + "' value=
            insertBefore( "#paypal-btn" );
            $( "<div/>" ).html( "<input type='hidden' name='amount_" + n + "' value='" +
            insertBefore( "#paypal-btn" );
            $( "<div/>" ).html( "<input type='hidden' name='shipping_" + n + "' value='"
            insertBefore( "#paypal-btn" );

        }



    }
}
```

First, we get some important information from session storage — namely, the shipping rate and the total number of items in the cart. We divide the total shipping amount by the number of items to get the shipping rate for each item.

Then, we set the URL for the `action` attribute of the form, together with our business email and currency code (taken from the `paypalBusinessEmail` and `paypalCurrency` properties, respectively).

Finally, we loop through the items of our cart, and we append to the form several hidden input elements containing the quantities, the names of the products, the number of items for each product, the prices (amounts), and the unit shipping rates.

The monetary values are formatted as `00,00`. Explaining all of the possible values of a PayPal form and the various types of PayPal forms goes well beyond the scope of this article, If you want to go deeper, I recommend the following reading:

- "HTML Form Basics for PayPal Payments Standard[24]," PayPal Developer
- "HTML Variables for PayPal Payments Standard[25]," PayPal Developer

Preview And Source Code

The following video shows the result. I've omitted PayPal's landing page to protect my account's data.

Get the code from the GitHub repository[26]. Just change the `paypalBusinessEmail` property of the `$.Shop` object to your PayPal Sandbox email account.

OTHER RESOURCES

- "DOM Storage Guide[27]," Mozilla Developer Network
- "Introduction to Session Storage[28]," Nicholas C. Zakas
- "Using data-* Attributes[29]," Mozilla Developer Network

(al, ea)

FOOTNOTES

1 http://www.w3.org/TR/webstorage/

2 http://caniuse.com/#feat=namevalue-storage

3 https://www.smashingmagazine.com/2011/04/fundamental-guidelines-of-e-commerce-checkout-design/

4 https://www.smashingmagazine.com/2014/10/reducing-abandoned-shopping-carts/

5 https://www.smashingmagazine.com/2010/10/local-storage-and-how-to-use-it/

6 https://www.smashingmagazine.com/2013/12/e-commerce-websites-showcase/

7 http://www.w3.org/TR/webstorage/#security-storage

8 http://www.us-cert.gov/sites/default/files/publications/TIP-12-298-01-Website-Security.pdf

9 https://developer.paypal.com/docs/classic/ipn/gs_IPN/

10 http://api.jquery.com/data/

11 http://api.jquery.com/jquery.data/

12 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed

13 https://api.jquery.com/jQuery.trim/

14 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseFloat

15 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt

16 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

17 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN

18 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toString

19 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_native_JSON

20 http://bassistance.de/jquery-plugins/jquery-plugin-validation/

21 https://api.jquery.com/jQuery.grep/

22 http://api.jquery.com/prop/

23 http://stackoverflow.com/questions/5874652/prop-vs-attr

24 https://developer.paypal.com/docs/classic/paypal-payments-standard/integration-guide/formbasics/

25 https://developer.paypal.com/docs/classic/paypal-payments-standard/integration-guide/Appx_websitestandard_htmlvariables/

26 https://github.com/gabrieleromanato/jQuery-sessionStorage-shopping-cart

27 https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage

28 http://www.nczonline.net/blog/2009/07/21/introduction-to-sessionstorage/

29 https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_data_attributes

Gabriele Romanato

Gabriele Romanato is a web developer. Contributor of the W3C CSS Test Suite, he is also skilled with jQuery and WordPress. You can find more about him by visiting his blog or his Facebook page. He is also on Twitter.