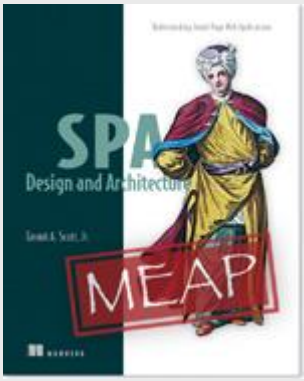

SPA Design and Architecture: The Communication Process

By Emmit A. Scott, Jr

In this article, I discuss some of the basics within the context of a single page application.

Though many concepts around communicating with the server will be the same for any type of web application, let's discuss some of the basics within the context of a single page application.

Choosing a data type

In order for the SPA running in the browser to communicate with a server, both need to speak the same language. So the first order of business is deciding upon the type of data that will be sent and received. To illustrate, we'll use the example of a shopping cart.

When the user interacts with our shopping cart, whether it's adding an item, updating the quantity of an item, or simple viewing the current state of the cart's contents, we're sending and receiving JSON-formatted text. JSON is commonly used by SPAs when communicating with servers, though the data type can be anything from plain text, to XML, to a file.

Even though we're using JSON-formatted text as a common data exchange format, it is merely a representation of a system's native object or objects. For the text to be useful, there are conversions that are happening at both ends. To ensure that the conversions to native objects work, each side must do their part to make sure the agreed upon JSON format is used in the call.

When a call is made to the server, requests can include information about the *Internet Media Types* that are acceptable since a resource can be available in a variety of different languages and media types. The server can then respond with a version of the requested resource that it deems a best fit. This is called *content negotiation*. In the case of our project, though, we are only interested in JSON. To express this, we can explicitly declare an *Internet Media Type* of `application/json` for the exchange.

Internet Media Types

An Internet Media Type (formerly MIME Type) is a standard way to identify the data that's being exchanged between two systems. It is used by many Internet protocols, including HTTP. Internet Media types have the format of `type/subtype`. In our case, we're using a media type of `application/json`. Here, the type is `application` and the subtype is `json`.

Optional parameters can also be added using a semicolon if required. For example, to specify a media type of `text`, with a subtype of `html`, and a character encoding of `UTF-8`, you would use:

```
text/html; charset=UTF-8.
```

Internet Media Types are specified using *HTTP headers*, which are the fields sent in the transmission that provide information about the request, the response or what's contained in the message's body. The *Content-type* header tells the other system what to expect in the request and response. The *Accept* header can also specified in the request to let the server know the media type or types that are acceptable to return.

Once a data type has been selected, an appropriate request method must be used for the call to be successful.

Using a supported HTTP request method

When a client makes a request, it can indicate the type of action it would like the server to perform by specifying the *request method*. In order for the request to be successful, though, the HTTP request method specified in the request must be supported by the server-side code for that call. If it isn't, the server may respond with a 405 Method Not Allowed status code.

Because the HTTP request method describes what should happen to the resource represented in the request, it is often called the "verb" of the call. A request method which doesn't modify a resource, like `GET`, is considered *safe*. Any request method that ends in the same result no matter how many times its call is executed is considered *idempotent*. For example, we'll use `PUT` when the user wants to update the count of a particular item that's in their cart. Because `PUT` is idempotent, we can tell the server we want two copies of Madden NFL 10 times in a row, but after the tenth time we still only have two copies in our cart.

In Table 1 we've defined a few common HTTP request methods used in our shopping cart example. While it's not a comprehensive list, it does represent the ones most commonly used in single page applications.

Table 1 List of common HTTP methods used in an SPA

Method	Description	Example	Safe?	Idempotent?
GET	Normally GET is used to fetch data.	view the shopping cart	Yes	Yes
POST	This method is most commonly used for creating a resource or adding an item to a resource.	add an item to the cart	No	No
PUT	Typically, put is used like an "update or create" action, updating the existing resource or optionally adding it if it doesn't exist.	update the quantity of an item in the cart	No	Yes
DELETE	This is used to remove a resource.	remove an item from the cart	No	Yes

There are other HTTP methods specified the HTTP protocol. For a full list, see: http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods .

The final part of the communication process is the conversion of the data to and from the Internet Media Type sent and received.

The data conversion process

Once the data type is agreed upon, both the client and the server must be configured to send and receive that particular type. In the case of our shopping cart, we're using JSON exclusively. This means that both the code in the browser and the code on the server must be able to convert to and from this text format.

On the client, the ability to convert a JavaScript object to JSON may be built into the MV* framework. If that's the case, it's likely the default and the conversion will happen automatically when you use the framework to make the server request. If it's not built in, the framework may offer a utility for the conversion of its custom types. For the conversion of JavaScript POJOs, you can use the native JavaScript command `JSON.stringify()`. `var cartJSONText = JSON.stringify(cartJSObj);`

On the server, the JSON-formatted text is converted into a native object of the server-side language by a JSON parser that's either built-in or via a third-party library. Like the HTTP method, the exact method for executing the conversion process on the server will vary.

To illustrate the process end to end, let's use the shopping cart update example again. Let's say that the user has increased the quantity of an item in the cart. For the modification to be verified and processed, we will send the updated cart to the server. Figure 1 paints a picture of the conversions that happen at both ends.

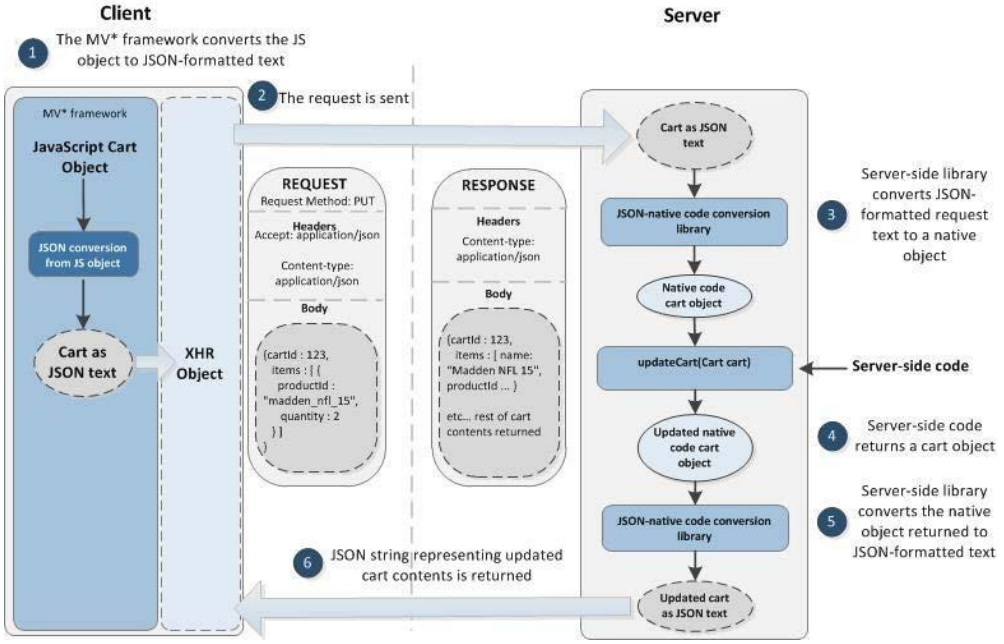

Figure 1 JavaScript objects are converted to JSON and added to the request body for the request. In response, the server sends back the updated cart as JSON using the response body.

Once the update function is called, our JavaScript cart object is converted into JSON-formatted text by the MV* framework. Next, the MV* framework passes the data to the XMLHttpRequest API. Then the JSON payload is sent in the body of the request to the server.

On the client, once the response is received, the returned text is converted once again. This time it is converted back into a native JavaScript object. Often this will be also handled

automatically for you by the MV* framework. If not, you can use the native JavaScript command `JSON.parse()`. `var cartJSObj = JSON.parse(returnedCartJSONText);`

(This article was excerpted from SPA Design and Architecture, from Manning Publications.)