Easy Web Development with WaveMaker

Edward Callahan

Chapter No. 10
"Customizing the User Interface with JavaScript"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.10 "Customizing the User Interface with JavaScript"

A synopsis of the book’s content

Information on where to buy this book

About the Author

**Edward Callahan** is an accomplished open source software engineer and consultant living in the San Francisco Bay Area. Currently with Typesafe, Edward was the Scrum Master and a core contributor to the WaveMaker project in VMware's SpringSource division. He led the formation of the WaveMaker developer community and has written countless forum posts, examples, and wiki articles about its development. Prior to joining WaveMaker, he was a Senior Manager of technical support for Progress Software's Enterprise Infrastructure Division. He completed his education at Worcester Polytechnic Institute and Northeastern University.

For More Information:

Easy Web Development with WaveMaker

This book describes an easier way to build custom, modern web applications using WaveMaker Studio, the web development tool for non-professional developers. WaveMaker runs in a browser and provides an intuitive visual development interface. The resultant applications use proven open source libraries, including the Spring framework and the Dojo Toolkit with the Model-view-controller (MVC) pattern. This is professional-grade development made easy.

In this book, we will walk through the entire development process. We will start with using the Studio and planning the application. We will then cover using JavaScript, Java, CSS, relational databases, and web services to provision functionalities and enhance the user experience. We shall conclude with deployment and debugging. By the end of this book, readers will be equipped to experience the joy of having designed, developed, and deployed a web application.

Communities are a wonderful resource for users of open source projects, as the collective wisdom of a community always outpowers the knowledge of any single individual. Readers are encouraged to participate in the WaveMaker developer community at http://dev.wavemaker.com. There, you will not only find documentations and examples, but also a fantastic group of users, new and old, with whom you can discuss, share, and collaborate.

What This Book Covers

Chapter 1, Getting Started with WaveMaker, covers setting up WaveMaker and the example project workspace. It also explores an example application from a user's perspective.

Chapter 2, Digging into the Architecture, explains the application architecture, covering the client, server, and communication between the two.

Chapter 3, Using Studio, walks through every aspect of using Studio, from visually assembling the application to using the binding dialog and code editors.

Chapter 4, Designing a Well-performing Application, covers how to design your application so it performs well and is easy to use.

Chapter 5, Navigating towards Reusability, explains the numerous navigation techniques, such as pages and dialogs, available to developers to create reusable components and a customized user experience.

For More Information:
Chapter 6, *Styling the Application*, covers using CSS with the WaveMaker framework to style applications.

Chapter 7, *Working with Databases*, covers everything you need to know about working with relational databases in WaveMaker.

Chapter 8, *Utilizing Web Services*, explains how to consume web services in your applications.

Chapter 9, *Custom Java Services*, covers everything you need to know to utilize Java for custom server-side functionality.

Chapter 10, *Customizing the User Interface with JavaScript*, explains how to customize applications using JavaScript while leveraging the WaveMaker client framework.

Chapter 11, *Mastering Client Customization*, explores advanced client customization techniques such as custom formatters, custom grid columns, and dynamic page content.

Chapter 12, *Securing Applications*, covers the WaveMaker security model and explains how to secure your application. It also introduces customization of the security configuration for advanced cases.

Chapter 13, *Deploying Applications*, explores the deployment process while reviewing your deployment options.

Chapter 14, *Mobile Deployment*, introduces building a native mobile application from your WaveMaker application using PhoneGap.

Chapter 15, *Debugging*, covers how to troubleshoot and debug applications, from logging to debugging with browser and Java tools.

For More Information:
Customizing the User Interface with JavaScript

We have seen that the use of JavaScript is not required to develop complete applications. However, the WaveMaker client framework provides a rich palette to work with for those who wish to take advantage of JavaScript. This chapter will introduce using JavaScript to customize the user interface of WaveMaker applications.

We'll review the asynchronous part of the AJAX model as well as the component and DOM events that dictate how and when we invoke functions, from button clicks to service call returns. Once we understand events, we will explore some of the things we can do within those events. We will work with `wm.Variables` and see how they act as instruments for getting and setting data values. Not to be left out, we'll also include the role of JavaScript variables in page code.

We will also cover bindings. We'll show you how you can use simple binding expressions and custom JavaScript to enhance the responsiveness of the user interface.

Finally, we'll review some of the resources available to developers, from the client API reference to the component classes themselves. We'll also demonstrate how the community forums are not the only way to learn new JavaScript tricks.

For More Information:
Important warning for browser-executed code

All JavaScript in WaveMaker is executed in the browser. Unless you are deploying your application to a private, secure network only accessible by highly trusted devices, the client must be considered as untrusted. Even when a username and password is required to access the application, the client host could potentially be under malicious control. Never should secrets or security be entrusted to JavaScript. Any system password, proprietary logic, access restriction, or validation done in JavaScript can be circumvented, modified, extracted, or disabled. Such things must be done on the server-side in order to be secure.

Previously, we used the Chrome Developer Tools JavaScript console in Chapter 1, Getting Started with WaveMaker. Malicious users can also use the console to manipulate application components in the very same way. Finally, no matter how secure we could make the client code, any and all logic used in the client can be circumvented by sending a manually crafted XHR request to the server. Therefore, data must be validated on the server side to ensure enforcement of rules.

Client-side customization enables you to deliver a rich, responsive interface to the user. Form validation on the client side is better for the user than server-side validation, but it's not secure. All required data validation must be performed on the server side. You can do some or all of the validation in JavaScript too, but always remember that it is executing on the client side and the client is not to be trusted.

Events

Events are pretty straightforward creatures. We looked at how the binding dialog is really a visually tooled version of dojo.connect() in Chapter 2, Digging into the Architecture. Binding via connect registers a function to be called when an event occurs; "when X happens, please also do Y". We will look at this in detail in a short while.

Asynchronous events

If events are easy to understand, you'd be forgiven for not expecting asynchronous events to be one of most common sources of trouble. Not understanding the impact of the "A is for asynchronous" part of AJAX is a common cause of frustration in accessing service call results.
When the browser invokes an asynchronous server call, for example using a service variable, the browser sends the request off to the server, but it doesn't wait idle until it receives a response. For example, if you used the following code in your application to invoke a service variable and immediately fetch the results, the `result` variable will be undefined nearly every time, if not always:

```javascript
this.serviceVariable1.update();
var result = this.serviceVariable1.getData();
console.log(result);
```

Every once in a while, if the service call can be completed quickly, you might get a value in `result`. Every other time, `result` will be undefined.

To ensure we access server call results at the correct time, WaveMaker provides callback functions (https://en.wikipedia.org/wiki/Callback_(computer_programming)). The callback functions `onSuccess`, `onResult`, and `onError` should always be used when depending on values returned from a service variable. Until these events fire, it cannot be assured that the service variable has returned.

**Event handlers**

When you choose JavaScript from an event menu, such as the `onclick` event of a button, an empty event handler function is created for you:

![Event handler](image)

The empty event handler function is as follows:

```javascript
button1Click: function(inSender) {
},
```

For More Information:

Within these functions, you can utilize any valid JavaScript. The function name is added to the widget definition in the page’s widgets file; for example, Main.widgets.js. This causes the WaveMaker client runtime to connect the custom event handler to the event. For example, button1 in the following code snippet, when clicked, will invoke the function named button1Click:

```
button1: ["wm.Button", {"margin": "4"}, {"onclick": "button1Click"}],
```

This enables the function to be called when the event occurs. This also means that if you change the name of a function and the function name is not updated in the widget definition, the function will not be invoked upon the occurrence of the event.

If you rename the widget using the name editor of the Properties panel, Studio will rename the event handler and update the widget definition. However, if you change button1Click to myButtonClickFunction in the Source tab manually, myButtonClickFunction will no longer be called when button1 is clicked. Either use the name editor of the Properties panel or manually type the function name into the event.

![Event Handler](image)

Entering a function name in events is an easy way to use the same function for multiple event bindings. Instead of repeating the logic in each event handler, define a new function in the page code to perform the common logic and have numerous events call that single function.

For convenience, within page code, this always refers to the current page. From a page named main, main and this are the same such that this===main is true. That is, main and this are strictly equal and refer to the same object.

Another convenience of component event handlers is inSender. The function argument inSender always refers to the sending object; the object on which the event occurred. In the case of our button-click, the button is inSender. If our button is named button1, inSender and this.button1 are strictly equal, or inSender===this.button1.

For More Information:
Other event handler templates have additional parameters that you can utilize. A live form's `onBeforeServiceCall()` function, for example, provides `inSender`, `inOperation`, and `inData`. The following is an example usage of `inData`:

```javascript
  customerEditLiveForm1BeforeServiceCall: function(inSender, inOperation, inData) {
    if (inData.zip.toString().length != 5) {
      app.toastError("Zip code must be 5 digits only");
      // Must be error of "Abort" to stop service call
      throw new Error("Abort");
    }
  }
```

The value of `inOperation` is the name of the operation about to be performed, such as `update`, and `inData` is the data values that are about to be sent to the server. The function template arguments provide quick access to commonly needed components; however, you are not required to use any of the arguments passed to the event handler function. They are for your convenience.

### Binding expressions

Binding expressions are an alternative way to use JavaScript to customize the user experience. Instead of using an event handler function from the **Events** dropdown, binding expressions are accessed using the **Expression** tab of the **Binding** dialog:

![Binding dialog](image)

---

**For More Information:**

Customizing the User Interface with JavaScript

In the preceding screenshot, the title of `customerDetailsPanel` is bound to the following expression:

"Details for " + ${customerDojoGrid.selectedItem.company}

This causes the panel's title to look like the following:

![Details for Cover to Cover Booksellers](image)

Studio informs you of the presence of the expression in the Properties panel by displaying `expr:` and the beginning of the expression in the bound property:

```
Text

title: expr:'Details for' CROSSMARK

Behavior
```

The same result could be achieved by calling `setTitle()` in a custom event handler for the `customerDojoGrid`'s `onSelect()` event:

```javascript
customerDojoGridSelect: function(inSender) {
   this.customerDetailsPanel.setTitle("Details for " + 
inSender.selectedItem.getData().company);
},
```

In the case of customizing the panel's title using the grid's `onSelect` event, unless you use the `selectFirstRow` property or otherwise programmatically select a row, the panel title will first be shown with the title property value. The title will be set only upon grid selection. The same result can be achieved using either binding expressions or event handlers; however, there are some distinct differences between the two.

For one, binding expressions are part of the widget definition, not the page code. This moves the expression out of sight. This keeps your page code cleaner and can be desirable, but it also means that setting a breakpoint in an expression is far more complicated. Setting a breakpoint in an event handler is a simple matter of opening the page in the debugger. On the other hand, setting a breakpoint in an expression is much more difficult, even for experienced users.

For More Information:

Another significant difference between page functions and binding expressions is the context in which they are executed. Because of this difference in context, binding expressions use a $\{\ldots\}$ syntax to refer to components. Within the $\{\ldots\}$ syntax, this refers to the window, not the page. As of WaveMaker 6.5, you can use this in an expression to refer to the page outside of the $\{\ldots\}$ syntax. This means that you can access the data value of an editor named text1 by using this.text1.getDataValue() or ${text1.dataValue}. However, only the $\{\}$ part of the expression is notified of the changes to the evaluated data value. This means that we cannot use the same syntax that we successfully used for the grid’s onSelect handler as our binding expression:

```
"Details for " + this.customerDojoGrid.selectedItem.getDataValue().company
```

The preceding code does not work as an expression. The expression will never evaluate and the title will never be set. Only changes to values referenced within the $\{\ldots\}$ parts of an expression result in the expression being re-evaluated. As this expression lacks a $\{\ldots\}$ evaluation, the expression never evaluates. Furthermore, only bindable properties and values are accessible within the $\{\ldots\}$ syntax. If you want to use a value or property that has not been exposed in the binding dialog, you need to get that value outside of the $\{\ldots\}$ syntax or use a page event handler function instead.

For further information on binding expressions, visit the binding expressions page in the advanced topics documentation at http://dev.wavemaker.com/wiki/bin/wmdoc_6.5/Binding+Expressions+Display+Expressions.

**Setting and getting values**

The getting and setting of values is very common in JavaScript page functions. Whether it is to validate user input, assemble input values to service calls, or generate messages and summaries, page code often reads and writes component values. It is best practice to use setters and getters whenever available. Getters and setters are the functions that get and set values on an object. They almost always use set and get in their name, such as getDataValue(), setCaption(), and setDataValue(). The use of getters, and especially setters, instead of performing an assignment ensures that change notifications and any required refreshing of the component takes place.

Take, for example, a label named labelTitle. We might be tempted to assign the caption to a value as shown in the following line of code:

```
this.labelTitle.caption="Welcome";
```

For More Information:
Now, what happens if we use the preceding line of code to assign a new value to a caption via the console? Well, nothing. We will not see a change in the caption displayed on the screen. Yes, `this.labelTitle.caption` now returns `Welcome`. Even calling `this.labelTitle.getValue("caption")` returns `Welcome`. However, in the case of a label caption assignment, we also need to call `renderLabel()`, as in `this.labelTitle.renderLabel()`, before the caption on the screen changes to the value set by the assignment. Calling `this.labelTitle.setCaption("Greetings")`, on the other hand, changes both the caption property value and the displayed caption with the single call. The `setCaption()` function calls `renderLabel()` for us, so we don't need to call it separately.

**JavaScript variables**

WaveMaker page code is standard JavaScript code executed in a Dojo environment. All the rules of JavaScript apply here. This means you can use JavaScript variables in your page code. JavaScript variables are simply any variable you declare; for example, the following stores the string `Success` in the variable named `result` and then logs the string `Success` to the console:

```javascript
var result = "Success";
console.log(result);
```

JavaScript variables are useful for storing transient values—values that you don't want after the function is complete. For example, consider the following line of code:

```javascript
var State = this.textState.getDataValue();
```

Here we get the text value the user entered into the `textState` editor and store it in a JavaScript variable named `State`. Let's see how even in simple cases there is benefit to using a local variable to store a value. Consider the following code:

```javascript
var State = this.textState.getDataValue();
console.log(State);
```

If nothing further will be done with the `State` data value, it is just as easy and more succinct to perform this in a single line:

```javascript
console.log(this.textState.getDataValue());
```

For More Information:

There is no functional difference between the two versions and readability is not impacted. However, if we check the value first, which is often advisable, the benefit of using a local variable becomes clearer:

```javascript
var State = this.textState.getDataValue();
if(State!=""){
    console.log(State);
}
```

The use of the JavaScript variable saves us the cost of having to get the value from the editor multiple times. It is more efficient to get the data value once and store it locally than to repeatedly fetch the data value from the editor. As an added benefit, the multi-line version provides more flexibility for setting debugger breakpoints.

**wm.Variable**

The other kind of client variable in WaveMaker is `wm.Variable`. A `wm.Variable` component is a non-visual component used for storing data values with type information. A key distinction between a JavaScript variable and `wm.Variable` is that `wm.Variable` is a component present in the project. This enables `wm.Variable` to be part of the project component tree and can be bound to other components, unlike a JavaScript variable.

Service variables and many other components contain or use `wm.Variable` components internally. Being objects, `wm.Variable` components have functions, including `setData()`, `getData()`, `setValue()`, and `getValue()`.

To add a `wm.Variable` component to a project, choose `Variable` from the `Insert` menu:

[Image of the Insert menu with Variable selected]

For More Information:
Upon creation, you will want to immediately specify a value for `type`, whether the variable is to be a list, and if we want to save the values in a cookie:

```
main.varPartNumbers.setValue("dataValue",[1242]);
```

The `isList` feature lets you specify that the variable will contain a list prior to assigning a value, thus enabling you to bind the variable as an array. If you want to retain the `isList` property of `wm.Variable` but need to store a single value in the variable, use an array of a single element. For a `NumberData` type, we can set the data value to an array of a single number such as the following:

```
main.varPartNumbers.setValue("dataValue",[1242]);
```
Type mismatches will cause problems, so choose the type carefully. In addition to the project types, such as the database entity types, the type dropdown includes choices for simple types such as EntryData, NumberData, and StringData, as well as PhoneGap types such as Address and Contact:

Using a wm.Variable component directly is no different than using wm.Variable embedded in service variables. With that said, direct access gives us "more rope with which we can hang ourselves", notably in the topic of our old friend: types. How one sets the data value of a wm.Variable component from code differs between simple and complex types. Examples of service variables returning primitive types include the security service getUserID() and getUserName() functions. Studio, with tooling such as the binding dialog, mostly absolves us of having to know there is a difference. Let's examine this in detail.

**Simple types**

Simple types are the "built-in" types including String, Number, and Boolean. In the type dropdown, these types are shown as AnyData, BooleanData, DateData, EntryData, NumberData, and StringData. These types lack the service name prefix of complex types.
When using a `wm.Variable` component of a simple type, use the `setValue()` and `getValue()` functions with `dataValue` as the first argument. For `setValue()`, the second argument is the new value. Let's see some examples:

- To get the values from a `wm.Variable` component of a simple type, call `getValue()` on `dataValue`:
  ```javascript```
  this.numVar.getValue("dataValue");
  ```javascript```

- Set a `wm.Variable` component of type `StringData` to the string "Hello WaveMaker":
  ```javascript```
  this.stringVar.setValue("dataValue","Hello WaveMaker");
  ```javascript```

- Set a `wm.Variable` component of type `NumberData` that has `isList` checked to an array of integers:
  ```javascript```
  this.numVar.setValue("dataValue",[24,27,32,34]);
  ```javascript```

- Set a `wm.Variable` of type `DateData` to January 1, 2014:
  ```javascript```
  this.dateVar.setValue("dataValue",new Date("01/01/2014").
  ```javascript```
  ```javascript```

**EntryData**

Of the simple types, `EntryData` is among the shortlist of the most useful candidates. A `wm.Variable` component of type `EntryData` is a really easy way to drive a custom dataset for user choices, most commonly as a select menu. The select menu's `dataSet` property is set to the entry data variable. The entry data of the variable can be set statically at design time or dynamically at runtime, or both.

Of the various use cases, the easiest is setting static values in Studio using the `Edit JSON` button in the variable's `Properties` panel:

![Edit JSON Button](Image)
This brings up the Edit JSON dialog:

```
[{
   "name": "GET",
   "dataValue": "HTTP_GET"
},
{
   "name": "POST",
   "dataValue": "HTTP_POST"
}
]
```

Entry data is an array of objects. Each object in the array has a name and a dataValue value. That's all there is to it, an array of value-pair objects.
Syntactically, we have the outermost brackets of the array: [ ]. Within this array, we have objects: [{}, {}, {}, ...]. Each object has a name and a `dataValue` value: 
{"name": "GET", "dataValue": "HTTP_GET"}.

The use of quotes around `name` and `dataValue` is optional with `EntryData`.

This is the same pattern we use to dynamically set the values. We simply pass a JSON structure of the entry data pattern to `setData()`. Consider the days of the week for example:

```javascript
this.varDoW.setData(jsonDoW);
```

If we have a select editor dataset bound to the `varDoW` variable, the select would show the days of the week after executing the preceding functions:

![Select Editor with Days of the Week]

**Complex types**

Complex types are the entity and service types that comprise the project types. We first discussed project types in Chapter 2, *Digging into the Architecture*. The `Wm.Variable` components of complex types are easy to use. Use `setData()` and `getData()` to access the whole object and `setValue()` and `getValue()` to access individual fields of the object.

Consider a variable of the type `Customer` from the `custpurchase` database data model. We can set the value of the `customer` object variable by passing a JSON object, even a partial object, to `setData()`. In the following line of code, we call `setData()` with a partial `Customer` object:

```javascript
this.varCustomer.setData({"city": "San Francisco","state": "CA"});
```
Or, we can set a single field of the customer type by using `setValue()`:

```javascript
this.varCustomer.setValue("state","CA");
```

To learn more about using variables, see the variables page in the documentation at `http://dev.wavemaker.com/wiki/bin/wmdoc_6.5/Variables`.

## Filters

Setting live variable filters using JavaScript can be troublesome for some developers. Filters are nothing more than a member `wm.Variable` named `filter`. This provides us with a few ways to manipulate filters.

The easiest way is to bind the live variable filter to a `wm.Variable` component and manipulate `wm.Variable` as we already have done. The type of `wm.Variable` must match the filter variable type. This means that we can filter on related objects; however, when we create the `wm.Variable` component, it must be for the related type, and we'll bind the related field to the `wm.Variable` component, not to the entire filter object.

Another option is to use `setFilter()`, passing a `wm.Variable` component as the parameter to `setFilter()`. Again, we are setting the desired values of the `wm.Variable` component as any other `wm.Variable`. Just be certain to update the `wm.Variable` values before calling `setFilter()`.

Finally, we can directly manipulate the filter variable using only JavaScript without binding:

```javascript
this.filteredCustomerLiveVar.filter.setValue("company", this.textCompany.getDataValue());
```

However, we choose to manipulate the filter values, as this can provide useful functionality. Consider the following search button click event handler function:

```javascript
buttonSearchClick: function(inSender) {
  //clear previous filter data
  this.filteredCustomerLiveVar.filter.clearData();

  //empty value for editors is empty string
  if(this.textState.getDataValue !== ""){
    this.filteredCustomerLiveVar.filter.setValue("state", this.textState.getDataValue());
  }
}
```

For More Information:

Customizing the User Interface with JavaScript

```javascript
if (this.textCompanyName !== "") {
    this.filteredCustomerLiveVar.filter.setValue("company",
    this.textCompany.getDataValue());
}

// AutoUpdate would fire on state if user set
// both state and name
// call update() here instead
this.filteredCustomerLiveVar.update();
```

First, we clear the filter to ensure we don’t have any leftover values by calling `clearData()` on the filter. Next, we check the editors for values. In this example, we set our editor’s `emptyValue` property to an empty string. So, we check if the editor data value is an empty string. If it is not, we set the corresponding value of the filter. We do this for as many fields as we may have input for. Finally, we fire the live variable using `update()`. Note that we do not use auto update, as that would cause the live variable to fire immediately upon setting the first filter field that had a value, even if the user had set both `state` and `company`. This is yet another example of how not using auto update gives us better control over our data access.

For more examples of using and manipulating filters, visit [http://dev.wavemaker.com/wiki/bin/Dev/Filters](http://dev.wavemaker.com/wiki/bin/Dev/Filters).

Resources

We’ve examined the building blocks of using custom JavaScript page code in a WaveMaker application. Unfortunately, being armed with this knowledge does not preclude you from staring at the flashing cursor of a source editor knowing what you want to do but having no idea of how to achieve it.

Fortunately, there is an array of resources available to us to help guide us to the desired syntax. Let’s review our options.

The console

The Chrome Developer Tools console, and others such as Firebug, provides command-line completions. The console displays a dialog of possible completions. As shown in the following screenshot, we’ve entered `main.varCustomer.setD` and the console is showing us all the members of the main page’s `varCustomer` object that start with `setD`:

For More Information:

Completions are context-sensitive, meaning that when used in a breakpoint of page code, `this` refers to the current and `inSender` refers to the sending object it is the same context as page code. This can be an incredibly useful way to probe some return value or otherwise unknown object. Set a breakpoint or enter the `debugger` keyword in the function you need to figure out. When you get it working, you can often directly paste the code into the Source tab, as the context used for debugging is the same as runtime.

**Trees**

Open the model and services trees when in the source editor to see names and types when editing code. Expanding entities and operations will show you name and type information. In the following screenshot, we have expanded the Customer table so that we can reference its fields:

The model tree and the components tree are also useful to remember names, cases, and spelling.

For More Information:

Customizing the User Interface with JavaScript

Completions
The script Source tab also provides completions. Like the console completions, the Script tab filters to match partial entries. Double-click on an entry from the Completions list to add to it the cursor point in the code:

![Completions list]

JS reference
We've seen the JavaScript client reference as the in-Studio help system. It can be useful on its own as well. The list of classes can be found here at http://dev.wavemaker.com/wiki/bin/wmjsref_6.5/WebHome. Each class has a page with sections for a synopsis, full description, direct methods, and links to parent class methods. Some pages even include examples and code snippets. The client reference is an excellent source for information.

For More Information:
JavaScript `toString()`

When in the Chrome console, the `toString()` function is a convenient way to view a function's implementation without finding the function in the sources. This works best when in debug mode. Without debug loading, all variables are shown as `_1c3` or other less-than-ideal names due to minification. In debug mode, calling `toString()` on a function displays the implementation of the function. As shown in the following screenshot, we've called `toString()` on a label's `setCaption()` function:

```
main.label1.setCaption.toString()
"function (inCaption) {
  if (inCaption == undefined) inCaption = "";
  var innerHTML = this.sizeNode.innerHTML;
  if (inCaption &amp;&amp; dojo.isArray(inCaption)) {
    inCaption = inCaption.join(' ', ');
  } else if (inCaption &amp;&amp; dojo.isObject(inCaption)) {
    inCaption = "";
  }
  this.caption = inCaption;
  this.renderLabel();
  if (innerHTML != this.sizeNode.innerHTML &amp;&amp; (
```

**Source code**

"In vino veritas" would be "in vino" and "code veritas" if the Romans had software. The WaveMaker source code is an excellent reference if you are comfortable with code and an ultimate reference for all others.

The most commonly referenced source for JavaScript developers is the WaveMaker classes in `/lib/wm/base` (https://github.com/SpringSource/wavemaker/tree/master/wavemaker/wavemaker-studio/src/main/webapp/lib/wm/base). As we first saw in Chapter 2, Digging into the Architecture, all the classes for the WaveMaker components are found in `/lib/wm/base`. If a new component lacks documentation, or we just need more information, being able to peek at the source code can be very informative.
Summary
We have just completed our first tour of using JavaScript to customize WaveMaker applications. We started with events. The custom JavaScript we write in our application is invoked in response to events. When working with server response data, we must always wait until the `onSuccess` or `onResult` events to ensure the service call has returned.

We also studied `wm.Variable`, the non-visual variable component. Unlike JavaScript variables, `wm.Variable` can be bound to other components and manipulated via code. We can use `wm.Variable` to customize widget datasets and filter live variables. Finally, we looked at the references available to use. From the API reference to the completions available to us in Studio and the console, there are multiple ways to learn more.

In the next chapter, we will continue our use of JavaScript to customize the user experience. We'll look at some advanced techniques such as creating components at runtime. We will also discuss examples of common operations such as forming display expressions for grid columns and manipulating live form values.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.