CSS Style Sheets in 4D Project Mode

By Shayanna Gatchalian, Technical Services Engineer, 4D Inc.

Technical Note 23-15

Table of Contents

Abstract

User interface design (a.k.a., UI design) has recently become an important pillar in application development, as it directly influences a user's experience. The color scheme, proximity of elements, and text styling all come together to accentuate a certain look and feel. While 4D already has an integrated form editor, it can take time to manually configure the style properties for multiple form elements, especially across various pages and forms. With the style sheet feature in project mode, the CSS language can be used to directly style forms in a more streamlined fashion. This technical note will go over how to implement external style sheets for project-based applications.

Introduction

CSS, short for *Cascading Style Sheets*, is a rule-based style sheet language that declares how document elements are styled and presented. It is most famously known to be one of the three main programming languages in web development (a.k.a., HTML, CSS, and JavaScript). In the context of 4D, however, it became the language of choice for style declarations when project mode was first introduced to the platform.

With a project database's file-based architecture, CSS style sheets presented an alternate way of styling the interface of 4D applications. As mentioned briefly earlier, declaring styles with external style sheets would allow for a more streamlined and organized development experience. Because of its simple rule-based syntax—like the JSON structure of form definitions—, it would also be easy for current 4D developers to pick up on the language. The integration of this feature allows for multiple form elements to be styled all at once with just a couple lines of code.

Overview of Style Sheets in Binary Mode

To better understand the various benefits of CSS style sheets for project databases, it is a good idea to first examine their humble beginnings in binary mode. Below is a screenshot of the *Style Sheets* page in the Toolbox window in designer mode of 4D.

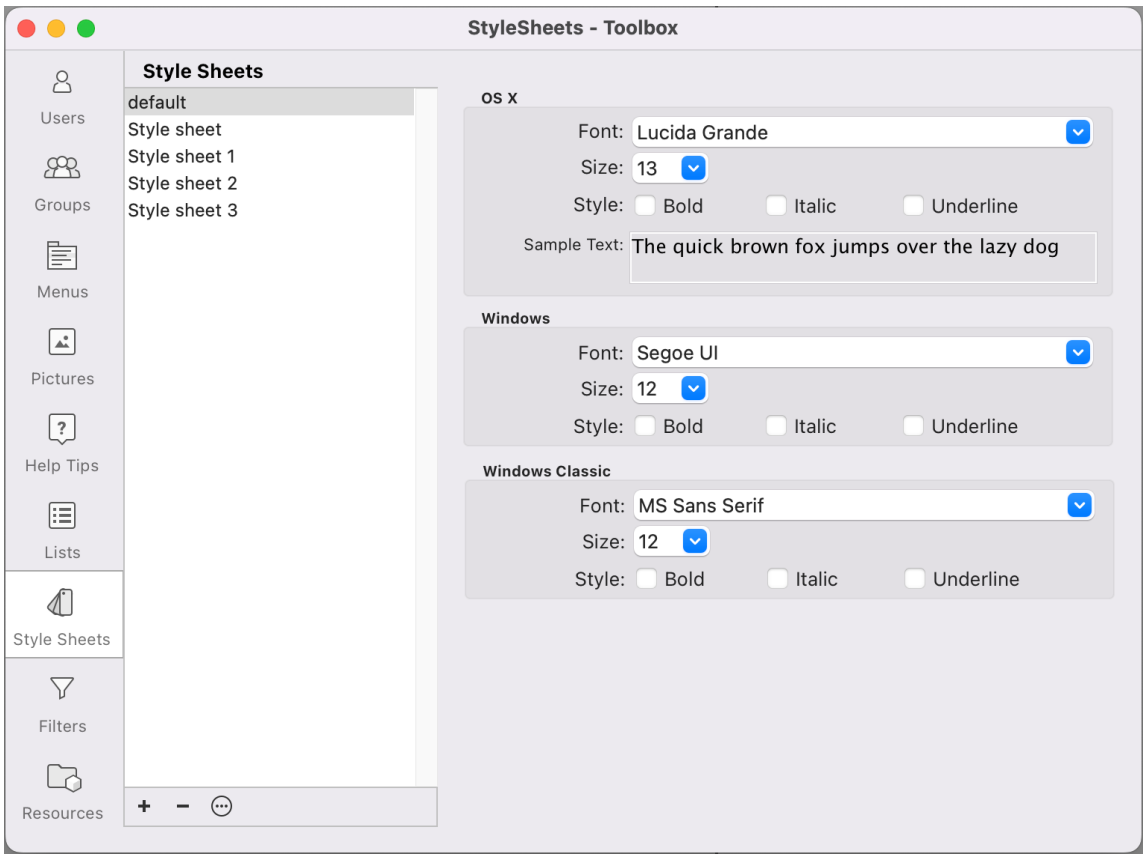

Image 1. The *Style Sheets* page in the Toolbox window (binary mode)

Notice that the only things that can be changed are the font, font size, and font styling for different platforms. Between the MacOS, Windows, and Windows Classic platforms, these settings can then be applied to any form elements that contain text.

Image 2. The *Style Sheet* property for form elements with text (binary mode)

Additional style sheets can also be created or deleted, which allows for separate style declarations to be made for different contexts or forms entirely. As you can see, style sheets in binary mode do not have much going for them; so, the addition of external style sheets in project mode can only build upon this feature.

Documentation on the style sheets feature in binary mode can be found here: https://doc.4d.com/4Dv20/4D/20/Style-sheets.200-6263695.en.html.

Overview of Style Sheets in Project Mode

Unlike binary mode, project mode makes use of external style sheets like those used in web development. With this structure design, these CSS files allow the developer to edit most properties of a form element, not just text styling. Like binary mode, the style sheets feature in project mode is found in the same *Style Sheets* section of the Toolbox window.

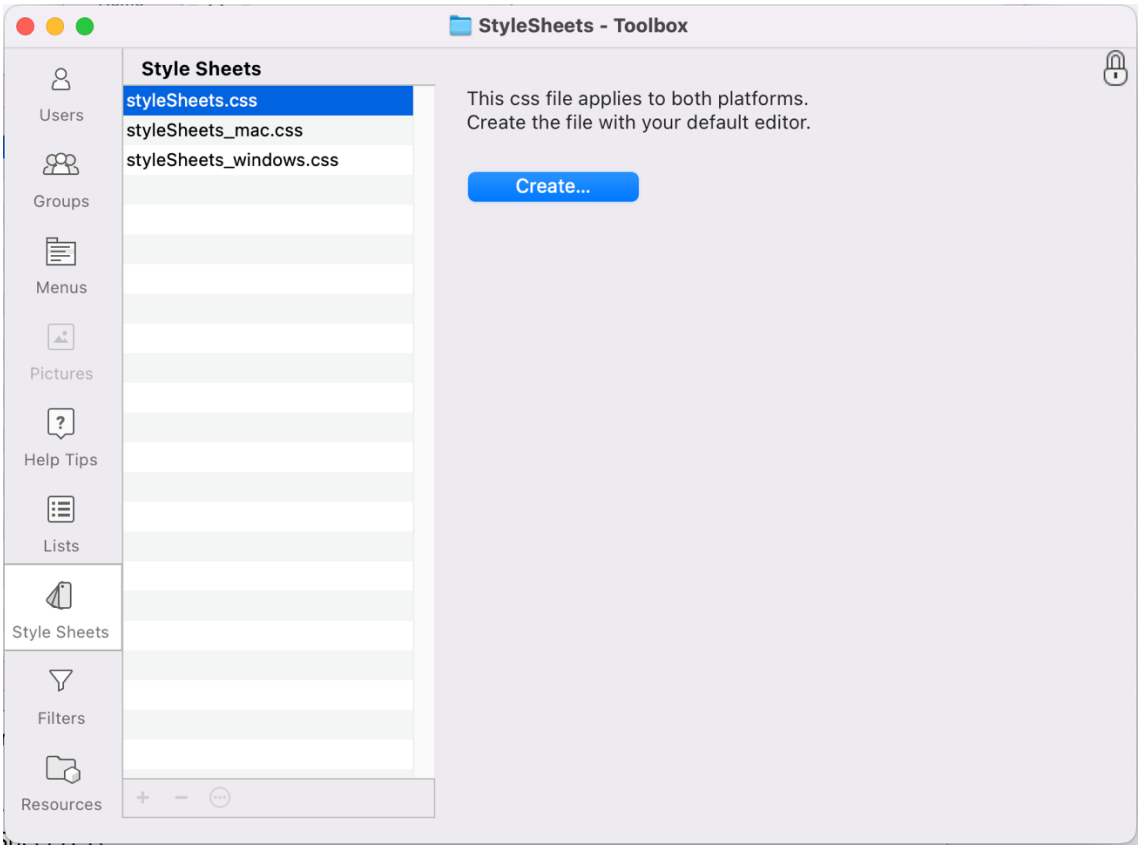

Image 3. The *Style Sheets* page in the Toolbox window (project mode)

As you can see, the list box already includes three style sheets by default:

- styleSheets.css
- styleSheets_mac.css
- styleSheets_windows.css

styleSheets.css is a general style sheet that applies to both Mac and Windows platforms, while each of the other two files will only apply styles on their corresponding platform. These external style sheets are not activated until a CSS file of the same name is created in the *Sources* folder of the database. This can be done by clicking on the *Create…* button when any of the three

style sheets are selected in the Toolbox window; once created, 4D will open the file in your preferred text editor application, and the *Create…* button will change into an *Edit…* button.

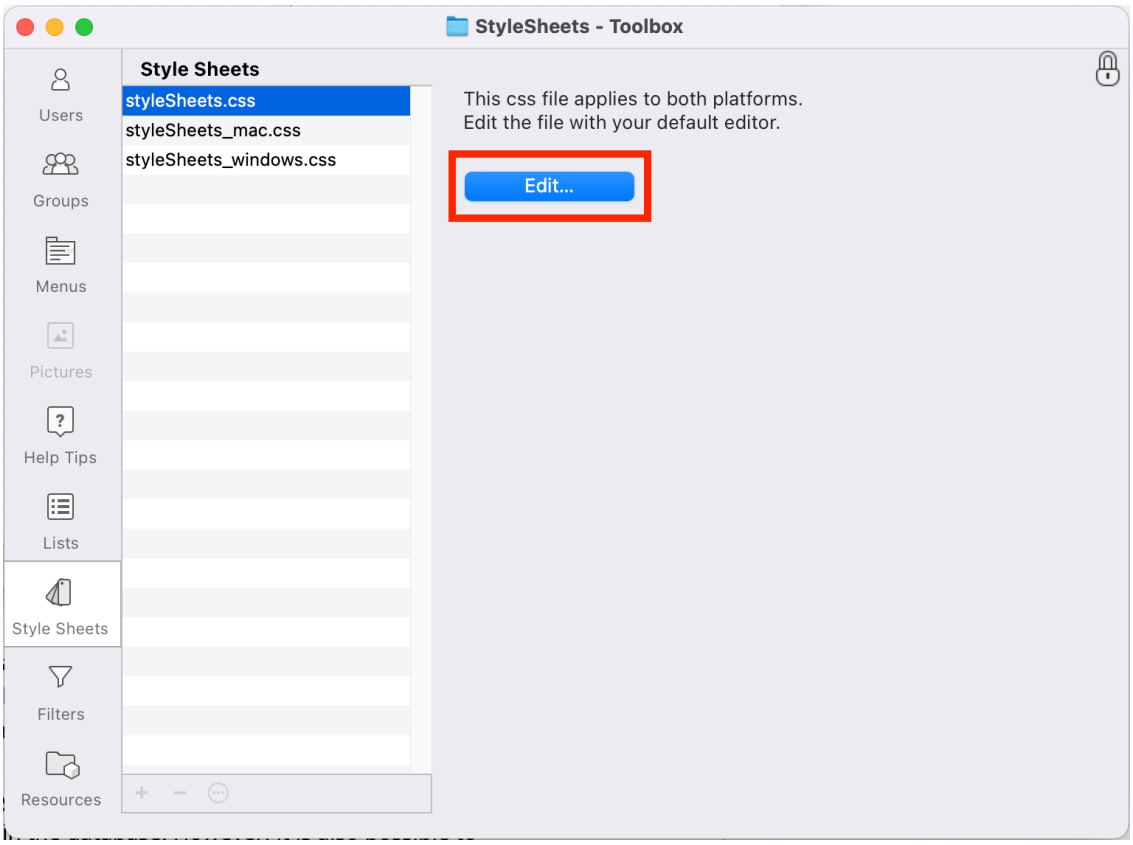

Image 4. The *Edit…* button once a CSS file has been created

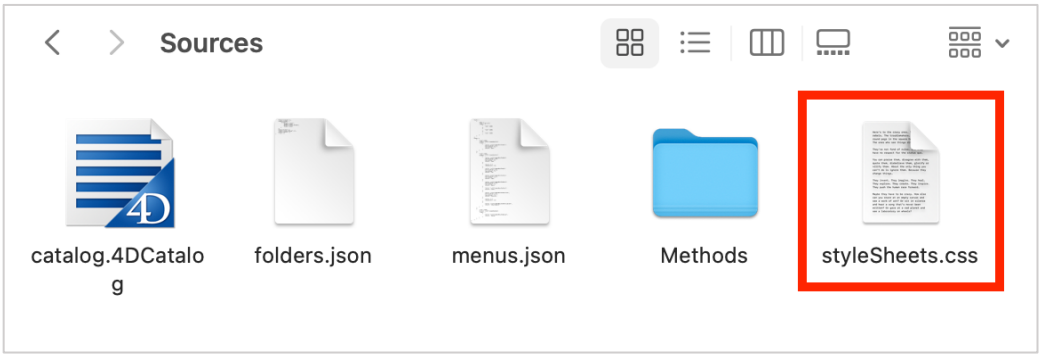

Image 5. The *styleSheets.css* file in the *Sources* folder of the database

Alternatively, you may also manually create the CSS file within the *Sources* folder of your database and 4D will automatically detect and link it. From there, you can implement CSS code to apply styling to your forms.

It is important to note that any style declarations included in any of these default style sheets will apply to all forms within the database. However, it is also possible to implement "dynamic" style sheets that apply to individual forms—this will be discussed later in this technical note.

The full documentation on project mode style sheets can be found on the 4D developer website: https://developer.4d.com/docs/FormEditor/stylesheets/.

What Happens to the Style Sheet When Converting from Binary to Project Mode?

As mentioned before, style sheets are an internal feature in binary databases, while in project mode, they are saved as external CSS files in the project folder. So then, how does 4D convert style sheets between the different structures when upgrading from binary to project mode?

Let's assume we have a binary database with the following style sheets implemented, and we export this to project mode. 4D will generate the *styleSheets_mac.css* and *styleSheets_windows.css* files with each style sheet previously configured in binary mode now saved as classes.

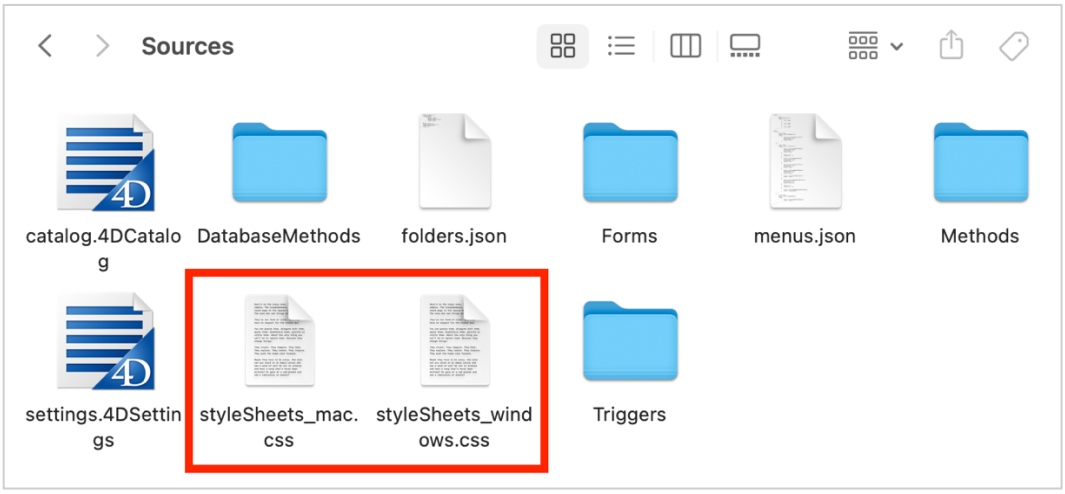

Image 6. The *styleSheets_mac.css* and *styleSheets_windows.css* files once a binary database is converted to project mode

Moreover, for each form element with text, the corresponding class is attached to the *CSS Class* property under the *Objects* section of the *Property List* window; the *Style Sheet* property is also removed from the *Text* section of the property list (see below).

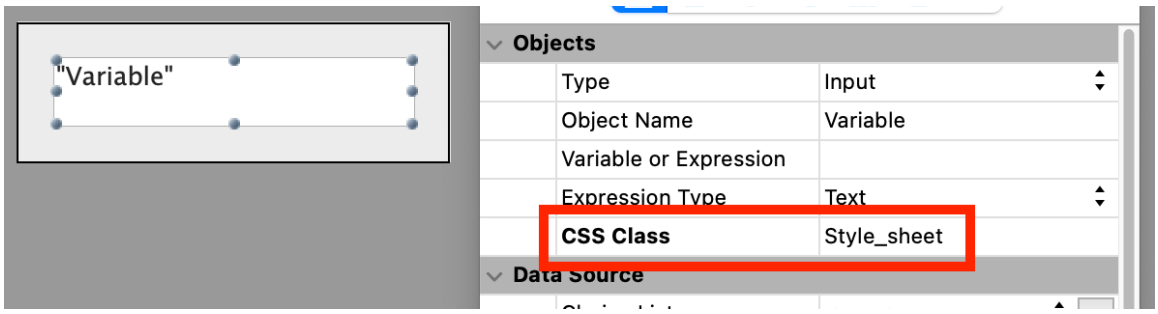

Image 7. The configured *CSS Class* property once the database is exported to project mode

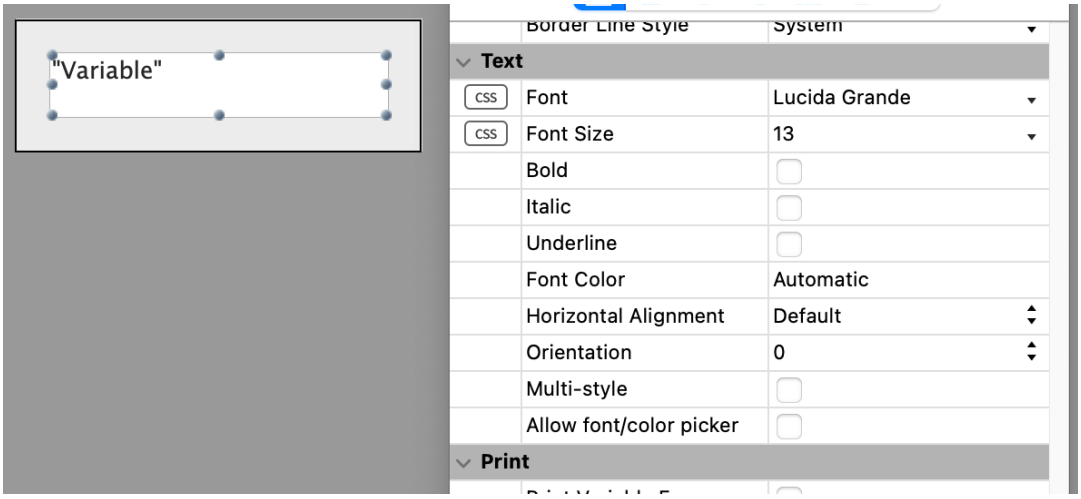

Image 8. The *Style Sheet* property now removed from the project database

Now, any text styling previously configured in binary mode is now formatted to fit the external style sheet structure of project mode.

Implementing External Style Sheets in Project Mode

CSS is a rule-based language that detects elements on a document and applies styling to them. In 4D specifically, style sheets follow the CSS2 syntax, meaning that the program cannot handle newer features offered by CSS3, such as responsive design and animations. Each rule is made of two parts: the *selector* and the *declaration*. The code snippet below shows an example of a CSS rule:

```
.myClass, button {
        stroke: purple;
        fontSize: 25;
}
```

Code Block 1. An example of a CSS rule

The *selector* is what comes before the brackets "{}", and this defines the form elements. In this case, the *myClass* class and *listbox* object type are chosen—selector types will be further discussed in the next section. When multiple selectors are used, each one must be separated by a comma ",".

The *declaration block*, naturally, is what is contained between the brackets "{}". It is made up of individual declarations that configure a specific property of the form element; a semicolon separates each declaration ";" when multiple declarations are used. They will also be explained in a later section of this technical note.

CSS Selectors

In the context of 4D, selectors can be categorized into five main groups: object type, object name, class, all form elements, and specific attributes.

Object Type

The *object type* selector refers to the type of form element (e.g., button, input box, etc.). This corresponds to the *Type* property in the *Property List* window of the form editor.

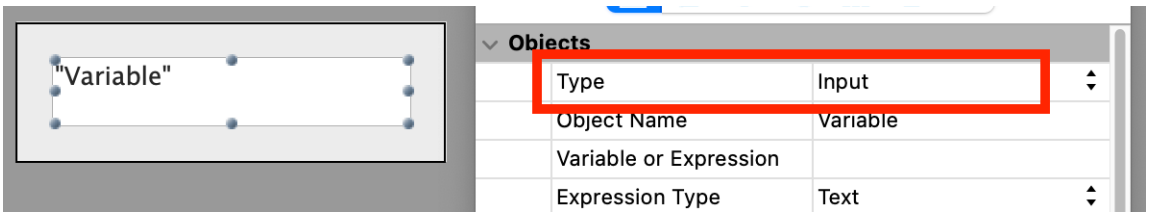

Image 9. The *Type* property of a form element

Within the style sheet, however, the JSON-equivalent value will need to be used to adhere to the JSON structure for 4D dynamic forms. While most form elements have a JSON value of the same name, not all do. For example, web areas have a *Type* value of "Web Area", but the JSON equivalent is considered as "webArea". The following documentation page

lists all the possible JSON values for the form element types in 4D: https://developer.4d.com/docs/FormObjects/propertiesObject#type.

```
input {
        /* add declarations here */
}
```

Code Block 2. A sample CSS rule with the *object type* selector

Object Name

The *object name* selector is essentially the same thing as the ID selector used in conventional CSS style sheets for web development. It corresponds to a specific form element according to a unique identifier (i.e., the *Object Name* property in 4D).

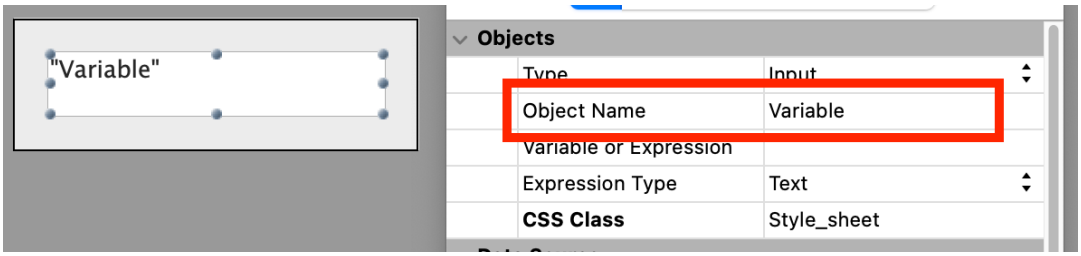

Image 10. The *Object Name* property of a form element

The value of this property must not contain any spaces, and when declaring this selector type, the form element name is preceded by a hash sign “#”.

```
#Variable {
        /* add declarations here */
}
```

Code Block 3. A sample CSS rule with the *object name* selector

Class

The *class* selector type is used to tailor a group of form elements with the same style declarations. For example, a class can be made for main titles while another can be made for subtitles; this ensures that all form elements under a class have the same style parameters. Like the *object name* selector type, a class name is first applied to the form

element within the form editor, under the *Class* property (again, this must not contain any spaces). Moreover, a single form element can be a part of multiple classes; this can be defined in the *Property List* window by separating each class with a space.

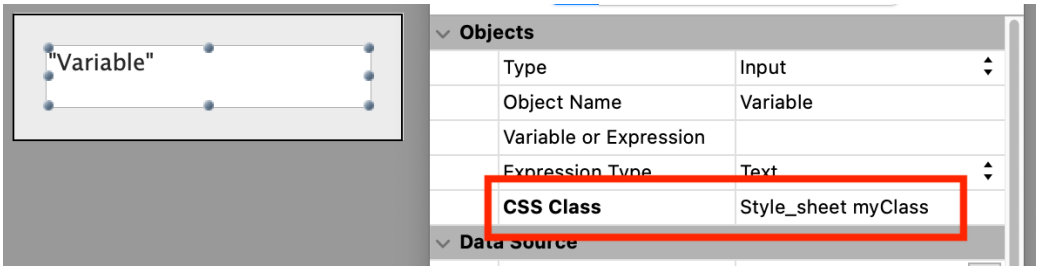

Image 11. The CSS Class property of a form element with multiple assigned classes

This selector can then be used in the external style sheet by inserting a period "." followed by the class name.

```
.Style_sheet, .myClass {
        /* add declarations here */
}
```

Code Block 3. A sample CSS rule with multiple *class* selectors

All Form Elements

To apply style declarations to all form elements, the * selector is used. If a form element does not contain a certain property, it will simply ignore the style declaration.

```
* {
        /* add declarations here */
}
```

Code Block 4. A sample CSS rule with the * selector

Specific Attribute

Here, an attribute is synonymous with a property you would find in the *Property List* window. With the *specific attribute* selector, form elements are chosen based on the attribute defined in the selector. Some syntaxes of this selector type allow the developer to further specify form elements by indicating certain values of the property as well. The CSS

language inherently has several syntaxes of this selector type, but 4D only supports the following four.

[attribute] Selector

This [attribute] selector simply applies styling to all form elements that contain the attribute/property.

```
[text] {
        /* add declarations here */
}
```

Code Block 5. A sample CSS rule with the [attribute] selector

[attribute=value] Selector

The [attribute=value] selector refers to any form elements that contain both the attribute/property and a value strictly equal to that indicated in the selector. For example, let's assume we are working with a *Text* object that simply says "Hello world! My name is John Doe."

```
[text=Hello world! My name is John Doe.] {
        /* add declarations here */
}
```

Code Block 6. A sample CSS rule with the [attribute=value] selector

[attribute~=value] Selector

Like the previous selector, form elements with the specified attribute are chosen; they are then further reduced to form elements where their values contain the one defined in the selector. To match the same behavior in CSS, specified values must be delimited by spaces for the form element to be chosen. Using the same text element previously, searching for the string, "Hello", would successfully style the form element, while searching for "He" would not.

```
[text~=Hello] {
        /* add declarations here */
}
```

Code Block 7. A sample CSS rule with the *[attribute~=value]* selector (successful)

```
[text~=He] {
        /* add declarations here */
}
```

Code Block 8. A sample CSS rule with the *[attribute~=value]* selector (unsuccessful)

[attribute|=value] Selector

The [attribute|=value] selector gathers all form elements whose attribute value either strictly equals a certain value or starts with that value. Unlike the previous selector, the value does not need to be surrounded by spaces. The code snippets below use the same *Text* element from the previous example ("Hello world! My name is John Doe").

```
[text|=Hello] {
        /* add declarations here */
}
```

Code Block 9. A sample CSS rule with the [attribute|=value] selector (using whole word)

```
[text|=Hel] {
        /* add declarations here */
}
```

Code Block 10. A sample CSS rule with the [attribute|=value] selector (using part of a word)

Documentation on supported selector types can be found here: https://developer.4d.com/docs/FormEditor/stylesheets/#style-sheet-selectors.

Selector Type Precedence

With these selector types working on different levels of the form, selectors that apply to the same form elements can clash. Because of this, a priority order is defined so that 4D knows which selector type takes more precedence in this kind of situation. 4D follows the same priority order set by CSS standards, which is listed as follows from most to least priority:

- Object name
- Specific attribute
- Class
- Object type
- All objects

Let's look at an example of a situation where selectors clash:

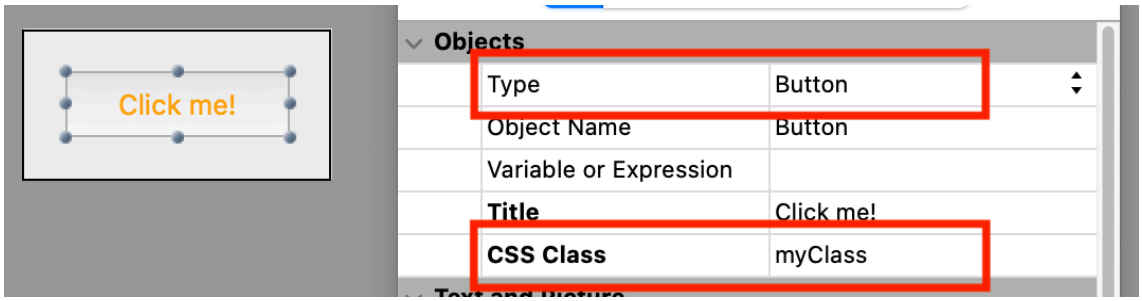

Image 12. A button form element under the *myClass* class with an orange font color

```
button {
        stroke: purple;
}

.myClass {
        stroke: orange;
}
```

Code Block 11. The CSS rules that clash when a button under the *myClass* class is present

Here, a *button* form element is also a part of the *myClass* class. In the style sheet, the CSS rule with the *myClass* class selector includes an orange stroke, while the rule with the *button* object type selector has a purple stroke. Because the *class* selector type has more precedence according to the priority order, it will be the one to apply its styling.

CSS Declarations

As mentioned earlier, CSS declarations define the styling for the form elements indicated in the selector of the CSS rule. A single declaration can be described as a key-value pair. In this case, the key corresponds to a form element's properties in the Property List window while the value dictates how that attribute is configured.

```
/* add selector here */ {
        fontSize: 13;
}
```

Code Block 12. An example of a CSS declaration

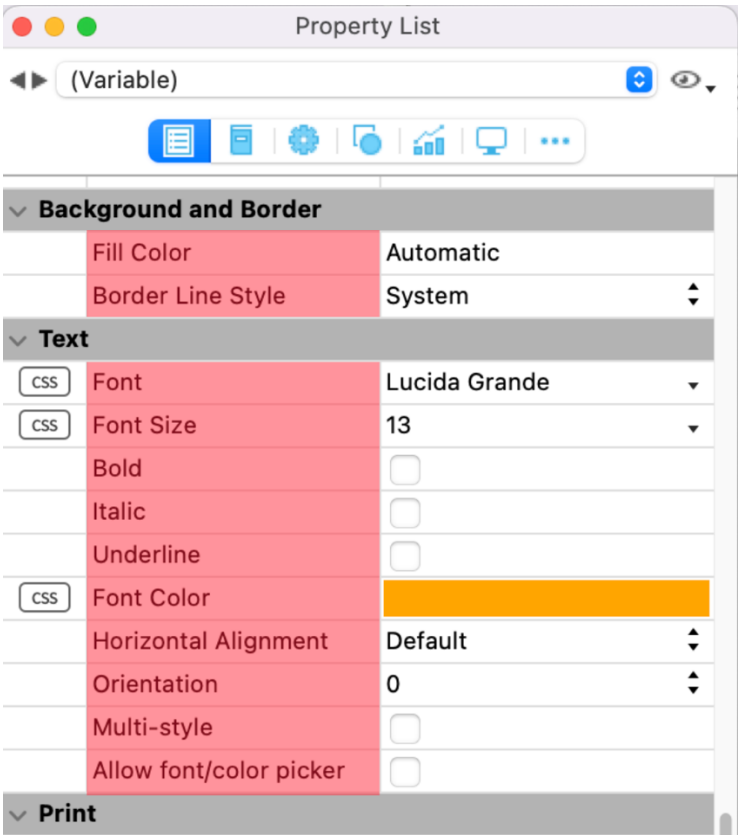

Image 13. The properties column in the *Property List* window

A property's list of accepted values can range anywhere from a string to a number value to a hexadecimal color code. A full directory of properties and corresponding values and value types can be found on the documentation website: https://developer.4d.com/docs/FormObjects/propertiesReference/. You may also look up the

documentation page for a specific form element—referred to as a “form object” on the developer website—and find its list of supported properties: https://developer.4d.com/docs/FormObjects/formObjectsOverview. It must be noted that not all form elements nor properties can be set by CSS style sheets; refer to the documentation first before attempting to style your forms via CSS.

Like selectors, declarations also need to use the JSON values for their attribute names as well as their values when applicable; this way, 4D will be able to correctly evaluate these constants to apply the styling.

Additional information on CSS declarations in 4D can be found here: https://developer.4d.com/docs/FormEditor/stylesheets/#style-sheet-declarations.

CSS Color Constants

For properties that involve color values, 4D has included a CSS color string feature where the style sheet can accept certain string constants that correspond to a specific color shade. For example, the string, “orchid”, can be used in place of “#DA70D6” or “rgb(218, 112, 214)”. This can come in handy in times when a fully custom color palette is not needed. The example below shows a rectangle being colored orchid using this concept.

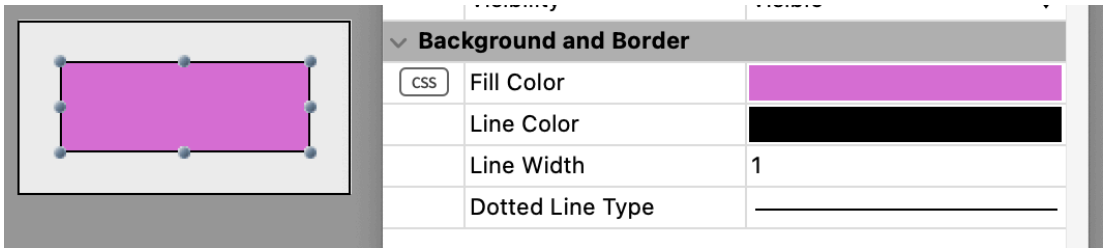

Image 14. A rectangle form element styled with an orchid fill via CSS

```
rectangle {
        fill: orchid;
}
```

Code Block 13. The CSS rule to style the rectangle with an orchid fill color

As a supplement, you can also use the following websites to improve your application’s UI.

- The *HTML Color Codes* website (https://htmlcolorcodes.com/color-names/) lists all the CSS color constants you can use in place of a hexadecimal color code or an RGB value.

- The *Adobe Color* web application (https://color.adobe.com/create/color-wheel) allows you to create a custom five-color palette according to various color rules. One unique function of this website is that it can automatically generate a color palette or gradient using an uploaded image.
- The *Coolers Color Contrast Checker* website (https://coolors.co/contrast-checker/000000-ffffff) improves the accessibility of your application by ensuring foreground and background colors are contrasting enough for users to read text.

CSS Media Queries

Media queries are a CSS feature where you can apply styling depending on the media the application is displayed on. By convention, this would imply electronic device characteristics, but CSS can also handle styling on printed media.

At the time of writing this tech note, 4D has only implemented the *prefers-color-scheme* media query for Mac platforms, which essentially sets different themes for the system's light and dark appearance modes. The following example shows what a media query looks like in the style sheet.

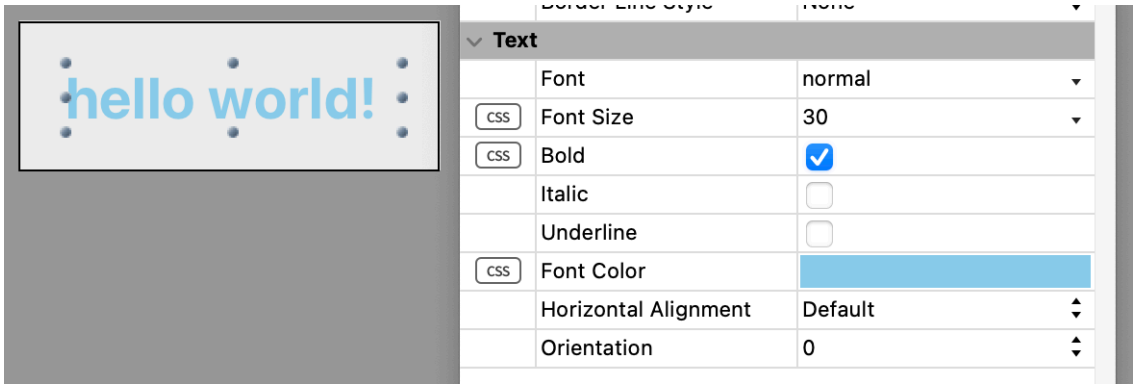

Image 15. A text element styled *prefer-color-scheme* media query set to light mode

```
@media (prefers-color-scheme: light) {
        text {
                stroke: skyblue;
                fontSize: 30;
                fontWeight: bold;
        }
}
```

Code Block 14. CSS media query code with style declarations for light mode

The media query essentially surrounds the set of CSS rules that would be applied if the media query's specified attributes match that of the device. Here, the media query first checks if the color scheme is set to light mode on Mac; since this is true, it applies the blue font color, 30 pt. font size, and bold font styling to the text form element.

Similarly, the following media query does the exact same function, except it applies a red font color if the Mac is set to dark mode.

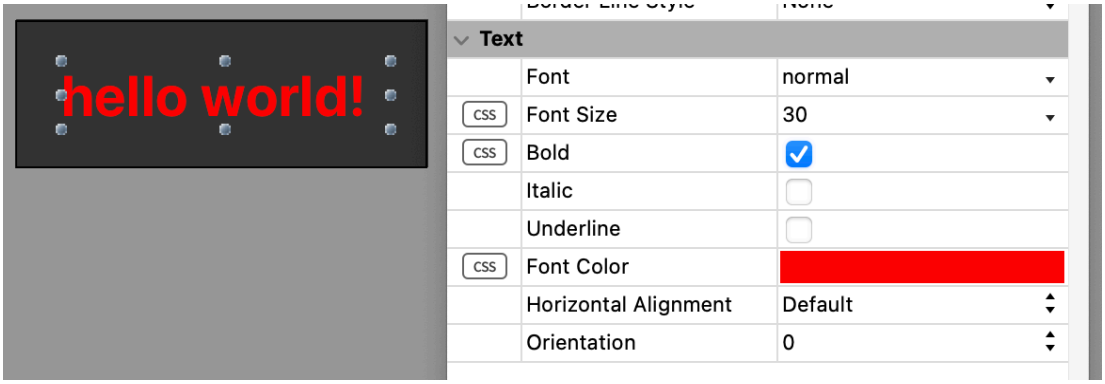

Image 16. A text element styled *prefer-color-scheme* media query set to dark mode

```
@media (prefers-color-scheme: dark) {
        text {
                stroke: red;
                fontSize: 30;
                fontWeight: bold;
        }
}
```

Code Block 14. CSS media query code with style declarations for dark mode

More information on this MSS media query can be found on the developer website: https://developer.4d.com/docs/FormEditor/stylesheets/#media-queries.

Applying the Style Sheets

As a quick reminder, the following are the three default external style sheets in 4D's project mode:

- styleSheets.css
- styleSheets_mac.css
- styleSheets_windows.css

Whenever a change is saved to any of these CSS files, 4D will automatically handle the code and output its styling when the application is executed. You must also ensure that these are in the *Sources* folder, as 4D has designated a specific path to these files. If you are creating a simple application with a limited number of forms, you may implement the CSS just with these default files.

Aside: The CSS Preview Feature in the 4D Form Editor

In previous generations of 4D, any styling configured by external style sheets would not be displayed until the application is executed in runtime. When v18R5 was released, the CSS Preview feature was added for project-mode databases to combat this minor annoyance, by directly presenting the styled form elements in the Form Editor window. A button in the toolbar also allows you to toggle between the three default style sheets, as indicated below.

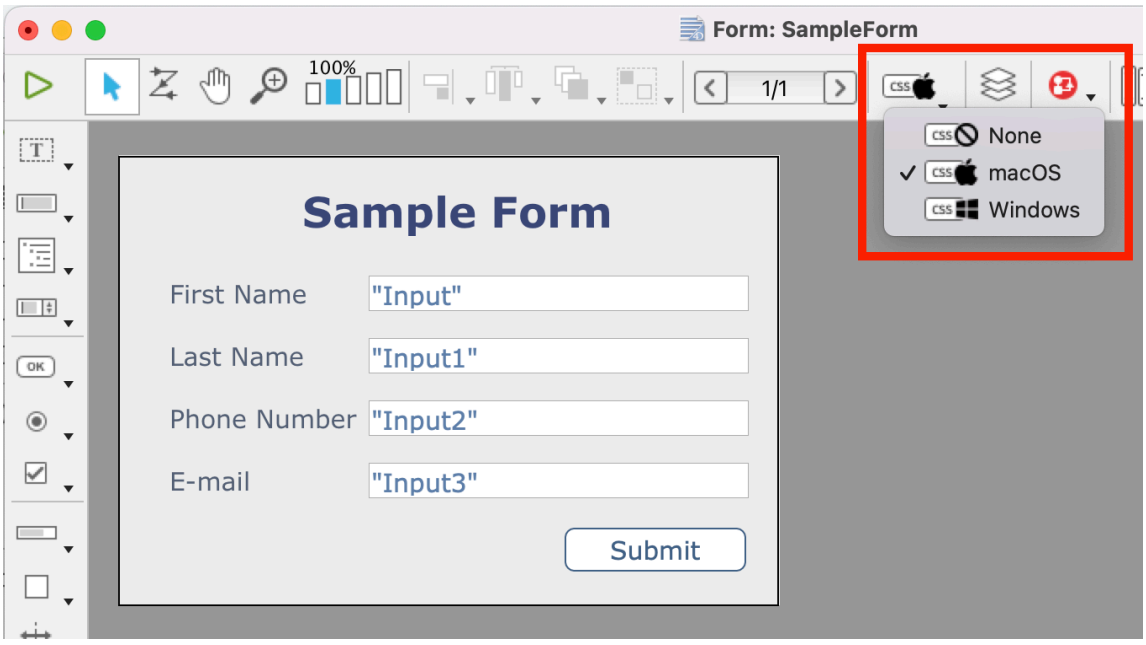

Image 17. The CSS Preview feature in the toolbar of the *Form Editor* window

```
* {
        fontFamily: Verdana;
}

.inputs, .labels, button {
        fontSize: 15;
}

button {
        style: flat;
        stroke: #426185;
}

.inputs {
        stroke: #5E82AD;
}

.labels {
        stroke: #566275;
}

#title {
        fontSize: 25;
        fontWeight: bold;
        stroke: #3D4A7A;
    }
```

Code Block 15. The CSS code used to style the sample form

The *Property List* window also indicates which properties are configured via the style sheet, by displaying a CSS icon to the left of the property name.

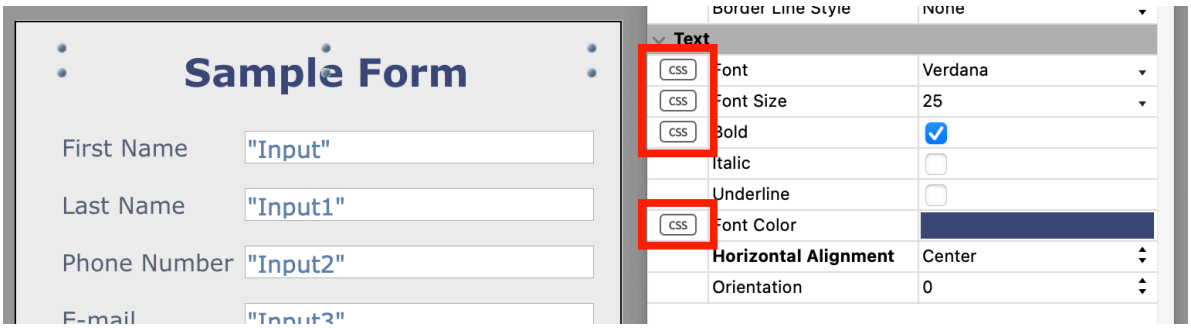

Image 18. The CSS icon indicating the property was configured via CSS

Moreover, if a property was changed via the Form Editor, two things will happen. The property-value pair is explicitly defined in the form definition (a.k.a., the .4DForm file), which essentially describes the attributes of an individual form in JSON format. Then, the property name in the *Property List* window becomes bolded to reflect this change.

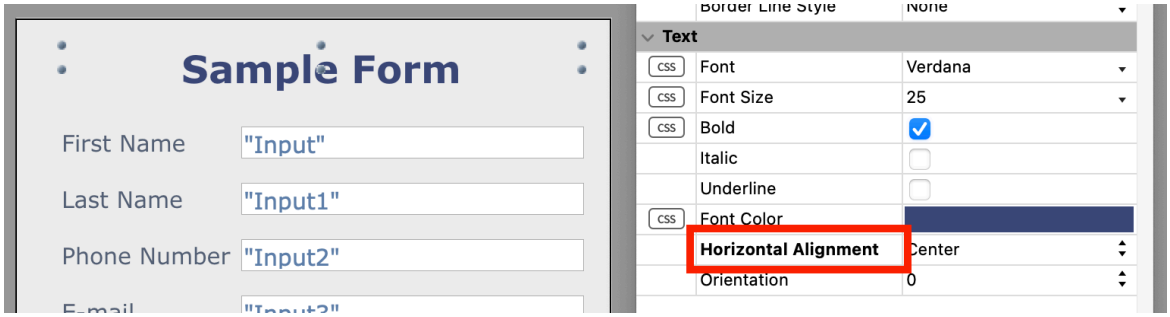

Image 19. A bolded property name indicating it is defined in the form definition

```
. . .
{
        "objects": {
                "title": {
                        "type": "text",
                        "text": "Sample Form\r",
                        "top": 18,
                        "left": 30,
                        "width": 343,
                        "height": 16,
                        "textAlign": "center"
        },
. . .
```

Code Block 16. A part of the form definition, indicating the text alignment property is explicitly defined here

More information on the CSS Preview feature can be found here: https://developer.4d.com/docs/FormEditor/overview#css-preview.

Additional Stylesheets

When it comes to larger databases or those with plans to expand in the future, keeping all the CSS code in three main files can make them too monolithic and prone to clashes; so, it may be a better idea to modularize your style sheets for better organization and maintenance in the long run. The following sections will go over two alternative implementations of external style sheets in project mode.

Importing additional style sheets

If a single default style sheet is getting too long, it can be broken up into separate CSS files. These shorter documents can then be imported into the main style sheet, simply by using the *@import* tag along with the path to the additional style sheet as its parameter. This type of implementation compartmentalizes the different types of CSS styling for better organization. For a basic example of this concept, let's say we create an additional style sheet named *secondary_styleSheet.css* that contains rules about the *.labels* and *.inputs* classes from the previous example; this will then be inserted into the *styleSheets.css* style sheet.

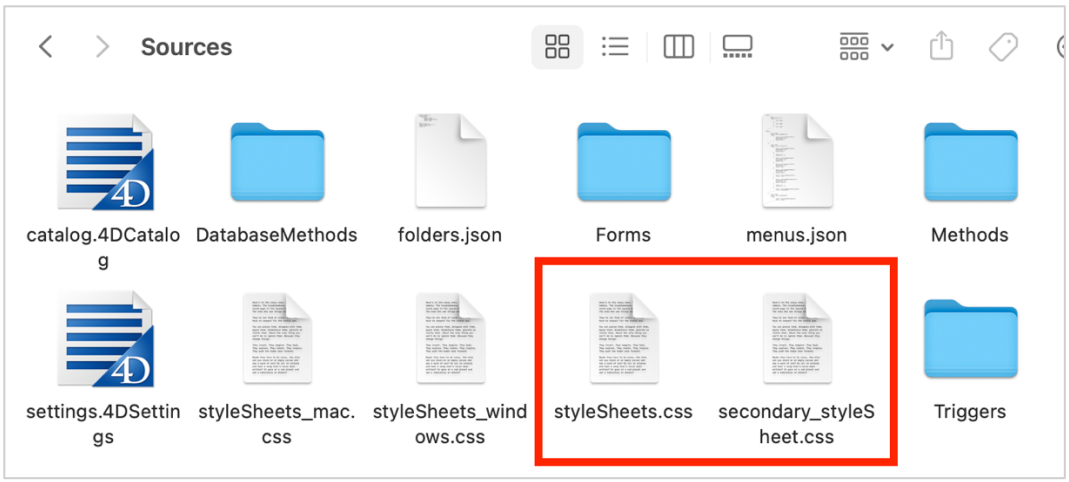

Image 20. An additional stylesheet on the same level as *styleSheets.css*

```
.inputs {
	stroke: #5E82AD;
}

.labels {
	stroke: #566275;
}
```

Code Block 17. The CSS code for the *secondary_styleSheet.css* style sheet

```
@import "secondary_styleSheet.css";

* {
	fontFamily: Verdana;
}

.inputs, .labels, button {
	fontSize: 15;
}
/* … additional CSS code */
```

Code Block 18. The CSS code for the *styleSheets.css* style sheet with the @import tag

Within the default style sheet, the *@import* tag with the path to the secondary style sheet is included at the top of the document to attach its CSS code. If the additional style sheet is saved on the same level as *styleSheets_mac.css*, there are two ways to write the @import tag:

- @import url("secondary_styleSheet.css");
- @import "secondary_styleSheet.css";

For imported CSS files saved in the parent and children folders of the Sources folder, you will need to adjust the relative path accordingly:

To navigate through *children folders*, use the forward slash "/" in the path:

- *Example:* to get the style sheet saved in a child folder under the Sources folder
    - @import url("/myFolder/secondary_styleSheet.css");
- *Example:* to get the style sheet saved in a child folder two levels down from the Sources folder
    - @import "/myFolder/mySecondFolder/secondary_styleSheet.css";

To navigate through *parent folders*, use the two-period ellipsis and forward slash "../" at the beginning of the path:

- *Example:* to get the style sheet saved in the parent folder of the Sources folder
    - @import url("../secondary_styleSheet.css");
- *Example:* to get the style sheet saved two levels above the Sources folder
    - @import "../../secondary_styleSheet.css";
- *Example:* to get the style sheet saved in another folder above the Sources folder
    - @import "../mySecondFolder/secondary_styleSheet.css";

After saving both files, you can navigate back to the form editor to see the changes—restarting the database may be needed.

Although this method compartmentalizes parts of the CSS code, it does not prevent the style sheets from becoming monolithic; during runtime, the broken-down segments of code are simply merged back into the default style sheet. However, combining this concept with the following procedure will fully modularize your style sheets.

“Dynamic” Style Sheets

Typically in web development, style sheets are created for each webpage. While 4D does not inherently allow for this function, there is a workaround discovered by 4D Vice President of Strategy, Thomas Maul. Essentially, he utilizes the JSON pointer concept to attach a single style sheet to the 4D form itself (his 4D forum post can be found here: https://discuss.4d.com/t/tip-use-dynamic-css-to-change-theme-colors-icons-etc/26366).

Before implementing this concept, there are some important things to consider. These dynamic CSS style sheets can be used in conjunction with the three default style sheets and will take the most priority in cases where style declarations clash (style sheet precedence will be further explained in the next section). And unlike with default style sheets, this workaround is not compatible with the CSS Preview feature—meaning you will need to run the application every time you would like to verify any changes to the style sheet. The implementation can be broken down into three parts: the external style sheet, the form definition, and an additional JSON file that acts as a directory.

External style sheet

Simply enough, this CSS file dictates the styling of form elements for the specified 4D form. Unlike default style sheets, these can be named anything other than those of the default style sheets and must be saved in the *Logs* folder of the database. This is because, in client-server databases, this folder has read and write capabilities for both the server and the client.

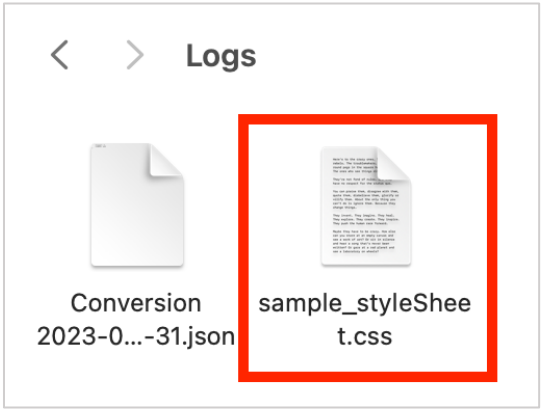

Image 21. An additional stylesheet saved in the *Logs* folder

Form Definition

As mentioned earlier, form definitions are saved as .4DForm files within their own folders under the *Sources* folder of the database. These files define the form

properties in JSON format, from window sizing to its list of form elements. Within this document, a JSON pointer is added to redirect to a file describing the directory of CSS style sheets.

```
"css": {
        "$ref": "/RESOURCES/theme.json#/css1"
},
```

Code Block 18. The JSON pointer to add to the .4DForm file

```
{
        "$4d": {
                "version": "1",
                "kind": "form"
        },
        "css": {
                "$ref": "/RESOURCES/theme.json#/css1"
        },
        "windowSizingX": "variable",
        "windowSizingY": "variable",
. . .
```

Code Block 19. The .4DForm file with the JSON pointer

Directory JSON File

This file defines the paths to each additional style sheet attached to individual 4D forms during the runtime of the application. It can also be named anything and will need to be saved in the database's *Resources* folder. The format simply requires a JSON object for each style sheet, as shown below:

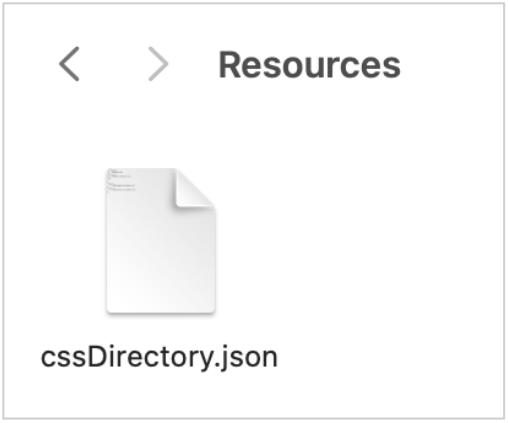

Image 21. The directory JSON file in the *Resources* folder of the database

```
{
	"css1":[
		"/LOGS/sample_styleSheet.css"
	],
	css2":[
		"/LOGS/another_styleSheet.css"
	]
}
```

Code Block 20. The JSON code inside of the directory file

Once all the files are saved, you can run your forms to see the reflected changes. “Dynamic” forms are advantageous in that a CSS file by itself will not affect any forms in the database until it is linked to a specific form. One thing that Thomas Maul emphasizes is that this method can be used to dynamically style 4D forms; for example, a developer can program his or her database to apply different themes or color schemes for each client user. Moreover, when combined with the imported style sheets mentioned earlier, you can effectively modularize parts of your CSS code to be reused across multiple forms.

Style Sheet Precedence

Like selector type precedence, the different style sheets as well as the form definition also have a priority order. From most to least priority, that order is as follows:

- “Dynamic” style sheets
- Form definition

- CSS rules with the !important tag
- Platform-specific default style sheet (styleSheets_mac.css or styleSheets_windows.css)
- General style sheet (styleSheets.css)

In general, the same precedence concept explained in the *Selector Type Precedence* section of this technical note applies here, but the *!important* tag deserves a little explanation. There may be cases where, as the developer, you would like a style declaration in the CSS file to override a property within the form definition—this is where the *!important* tag comes into play. This simple phrase can be added at the end of a declaration to accomplish that very goal.

```
text {
        stroke: navy !important;
}
```

Code Block 21. A CSS rule with the *!important* tag to override the font color

```
. . .
{
        "objects": {
                "Text": {
                        "type": "text",
                        "text": "hello world!",
                        "top": 22,
                        "left": 24,
                        "width": 175,
                        "height": 36,
                        "fontSize": 25,
                        "stroke": "#ffb6c1"
                }
. . .
```

Code Block 22. The form definition with the font color originally set to light pink

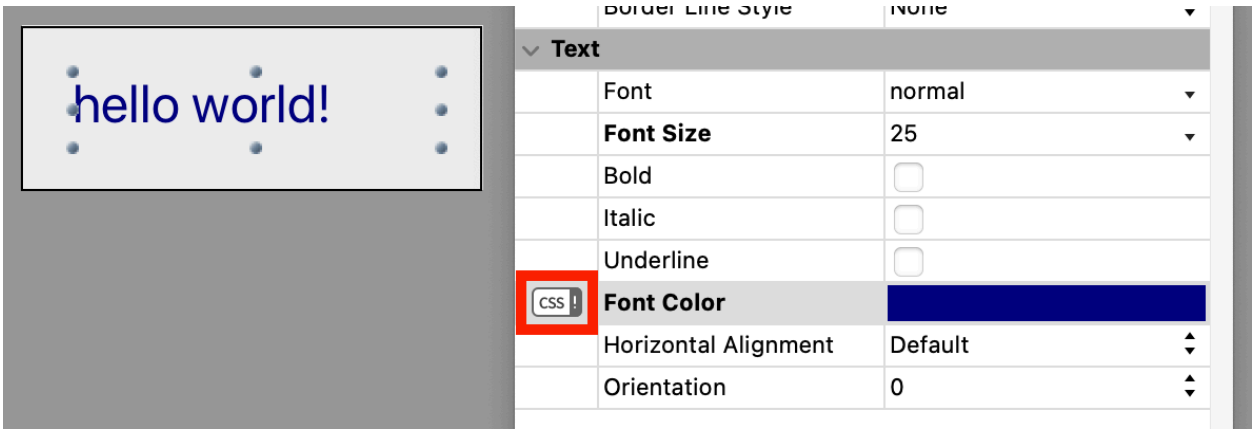

Image 22. The resulting text form element and a CSS! Icon indicating the CSS declaration overwrote the form definition

Conclusion

The integration of external style sheets in project mode unlocks a whole world of possibilities when it comes to styling your 4D applications. As you have read, 4D has come a long way from its simple implementation in binary mode. With the power of external style sheets and CSS, styling forms has become much more streamlined and organized. It also allows for features that are unique to its implementation, such as creating themes for light and dark appearance modes on Mac platforms and “dynamic” style sheets. Although most of the time, developers put more focus on functionality over usability, the latter is still an important aspect of developing a successful application, as it directly contributes to the user’s experience.