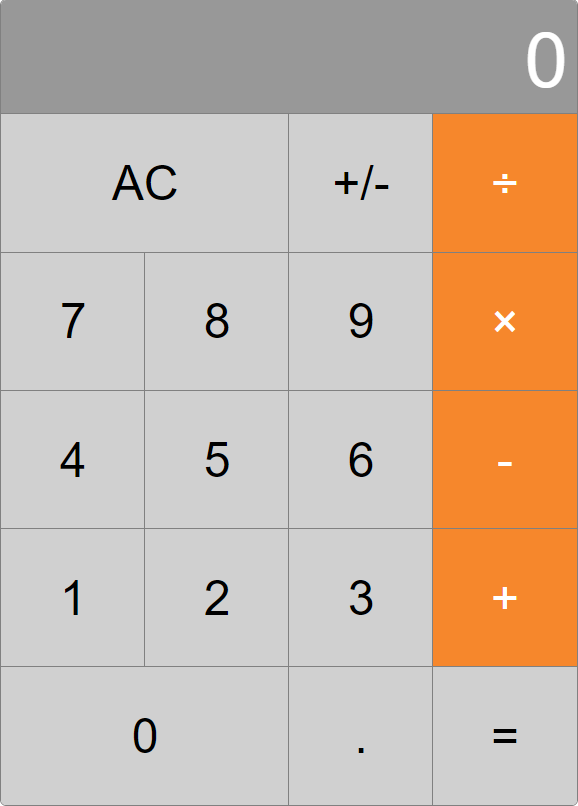
**Build a Calculator with Vue.js**



This workshop will take you through the steps that are required for building a working Calculator with Vue.js.

1. **Initialize a Vue project**

You can create a vue project through your system console or from your IDE. In this workshop, we are going to create it through the system console by following a few easy commands. First, open your console and change the directory to the one in which you want your project to be. Then, you have to install vue-cli globally with the command:

npm install vue

This is done only once, so you won’t have to do it for your future projects. The next step is to create the project:

vue init <template-name> <project-name>

The command pulls a certain template. For example, we can use a webpack-simple template for our purpose:

vue init webpack-simple calculator-workshop

You are going to be asked about the name, description, author and the license of the project. MIT is the default license. The next three commands will be shown to you in the console. First, you must change the directory to the one with your project:

cd <project-name>

Install the dependencies:

npm install

This command will start your local http server, open the browser and your default hosted web page will be shown:

npm run dev

The project is created. You can import it in your IDE. Now, it is left to think of a way to build a simple calculator from scratch and implement some or all of the functionalities it has. You can try to write it on your own. If you have any difficulties, you can follow the steps bellow and you should have a working calculator in the end.

1. **Basic Layout**

Open your project and find your App.vue file. You have three sections – for HTML, script and CSS. You can form the basic layout by your own choice, or you can use the one given bellow:

<template lang="html">  
 <main class="calculator">  
 <div ref="display" class="display">  
 <div class="display-text">  
 <span ref="displayText">{{display}}</span>  
 </div>  
 </div>  
 <div class="buttons">  
 <div class="button-row" v-for="row in buttonRows">  
 <div  
 @click="buttonClicks[button.type](button)"  
 :class="{ operator: button.type === 'operator' }"  
 :style="button.style"  
 class="button"  
 v-for="button in row">  
 {{button.text}}  
 </div>  
 </div>  
 </div>  
 </main>  
</template>

</template>

The “display” class shows the result from our calculations. The “buttons” class warps all of the buttons and inside it, there are classes for the rows and the button itself. Notice, that there are for loops inside the div. They are going to display the buttons, which are an array of arrays. The array is created inside the vue component in the data function:

<script>  
 **export default** {

data: (vm) => ({

buttonRows: [  
 [{  
 text: 'AC',  
 type: 'special'  
 }, {  
 text: '+/-',  
 type: 'special',  
 style: 'flex-basis: calc(100%/2)'  
 }, {  
 text: '÷',  
 type: 'operator',  
 style: 'flex-basis: calc(100%/2)'  
 }],

[{  
 text: '7',  
 type: 'number'  
 }, {

text: '8',  
 type: 'number'  
 }, {

text: '9',  
 type: 'number'  
 }, {

text: '×',  
 type: 'operator'  
 }],  
 [{  
 text: '4',  
 type: 'number'  
 }, {  
 text: '5',  
 type: 'number'  
 }, {  
 text: '6',  
 type: 'number'  
 }, {  
 text: '-',  
 type: 'operator'  
 }],  
 [{  
 text: '1',  
 type: 'number'  
 }, {  
 text: '2',  
 type: 'number'  
 }, {  
 text: '3',  
 type: 'number'  
 }, {  
 text: '+',  
 type: 'operator'  
 }],  
 [{

text: '0',  
 type: 'number'  
 }, {

text: '.',  
 type: 'number',  
 style: 'flex-basis: calc(100%/2)'  
 }, {

text: '=',  
 type: 'special',  
 style: 'flex-basis: calc(100%/2)'  
 }]  
 ]  
 })  
};

</script>

This way you give each button its text and type, as well as a special style if it is needed. The type is required for pointing the actions , which follow after a certain button is clicked.

1. **Style**

Now, you should be able to see in your app’s page, that you have the calculator’s text. Next, you can style it. The code bellow should make the calculator look like the one in the beginning of this document. You can create one of your own choice, if you want.

<style lang="css">  
 body {  
 width: 100vw;  
 height: 100vh;  
 font-size: 2.5vw;  
 display: flex;  
 justify-content: center;  
 align-items: center;  
 }  
 .calculator {  
 width: 30vw;  
 height: calc(30vw \* 1.4);  
 font-family: sans-serif;  
 border-radius: 5px;  
 overflow: hidden;  
 border: 0.5px solid grey;  
 }  
 .display {  
 height: calc(30vw \* 0.2);  
 background: #989898;  
 color: white;  
 font-size: 4vw;  
 display: flex;  
 flex-direction: column;  
 justify-content: flex-end;  
 }  
 .display-text {  
 padding: 0.5vw;  
 text-align: right;  
 }  
 .buttons {  
 height: calc(30vw \* 1.2);  
 text-align: center;  
 }  
 .button-row {  
 height: 20%;  
 width: 100%;  
 display: flex;  
 justify-content: space-around;  
 }  
 .button {  
 display: inline-block;  
 outline: 0.5px solid grey;  
 width: 100%;  
 background: #D0D0D0;  
 display: flex;  
 justify-content: center;  
 align-items: center;  
 }  
 .operator {  
 color: white;  
 background: #F6872C;  
 }  
 .button:active {  
 background: #b5b4b3;  
 }  
</style>

Refresh your app page and see what happens. If you are not familiar with HTML and CSS, you can play with the code a little and try to figure out the functionality of the different properties. Now, you can move on to the next step.

1. **Mathematical Operations and Display Result**

It is highly recommended to try to implement the calculator on your own. If you experience serious difficulties, you can help yourself with the given code and explanations.

You should write methods in the component. The first one will be called buttonClicked and it is going to be responsible for the actions, which follow after a certain button is clicked. When you click a number, you want the next one to append to the previous, and if you click an operator, you want it to do a certain type of calculation. You can pass the data function the view model like this – data: (vm) => {}. Inside it, you can write all of the functions for the different types of clicks – number, operator or special. The equal button, as well as the AC and +/- will be identified as special. You can write the following code above the buttonRows:

data: (vm) => ({  
 display: '0',  
 previousValue: '',  
 currentOperator: '',  
 buttonClicks: {  
 number(button) {  
 **if** (vm.previousValue === **null**) {  
 vm.previousValue = Number(vm.display);  
 vm.updateDisplay('');  
 }  
 **if** (button.text === '.' && vm.display.includes('.')) **return**;  
 vm.updateDisplay(vm.display + button.text);  
 **if** (vm.display.length > 1 && vm.display[0] === '0' && vm.display[1] !== '.') {  
 vm.updateDisplay(vm.display.slice(1));  
 }  
 },  
 operator(button) {  
 **if** (vm.currentOperator) {  
 vm.performOperation();  
 }  
 vm.previousValue = **null**;  
 vm.currentOperator = button.text;  
 },  
 special(button) {  
 **if** (button.text === 'AC') {  
 vm.updateDisplay('0');  
 } **else if** (button.text === '+/-') {  
 vm.updateDisplay(vm.display \* -1);  
 } **else if** (button.text === '=') {  
 vm.performOperation();  
 vm.currentOperator = '';  
 }  
 }  
 },

Look at the code and try to figure out how it works. Notice, that there some methods in this code. The updateDisplay method will update the value that is shown in the display result box and the performOperation method will perform the required mathematical operation. You can write them inside the component, above the data function:

<script>  
 **export default** {  
 methods: {  
 updateDisplay(value) {  
 **this**.display = value;  
 **this**.$nextTick(() => {  
 **let** fontSize = 4;  
 **this**.$refs.display.setAttribute('style', 'font-size:4vw');  
 **while** (**this**.$refs.displayText.offsetWidth + 30 > **this**.$refs.display.offsetWidth) {  
 **this**.$refs.display.setAttribute('style', 'font-size:' + fontSize + 'vw');  
 fontSize -= 0.1;  
 **if** (fontSize <= 0.1) **break**;  
 }  
 })  
 },

performOperation() {  
 **if** (**this**.previousValue) {  
 **const** value = **this**.operations[**this**.currentOperator](+**this**.previousValue, +**this**.display);  
 **this**.updateDisplay(value);  
 }  
 }  
},

And now inside the data, you can write the operations:

operations: {  
 '÷': (a, b) => a / b,  
 '×': (a, b) => a \* b,  
 '-': (a, b) => a - b,  
 '+': (a, b) => a + b,  
 },

This code should provide you with a working and decent looking calculator. If you try to enter a really big number, the font of the number should become smaller and fit on screen. However, there is a bug. When you get the result after an operation, if you try to click a number before you clear the screen, the number will append to the shown result. It is not supposed to work like that. When you click a number after you have received a result, the display should clear and only the number you have clicked should be displayed. You can think of a way to fix that bug. Try all of the operators and see if they are working right. If you can think of a way to refactor the code so that there are no functions inside the data, try it. Make all kinds of changes to this code until you are happy with the result.