# Tagged Templates

the ins and outs of tagged templates using a literalFragments array, and the tag & format functions

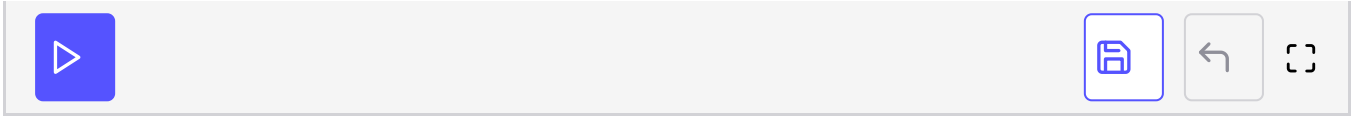
A template tag is a function performing a transformation on a template literal, returning a string. The signature of a tag function is:

```
tagFunction( literalFragments, ...substitutionValues )
```

- `literalFragments` is an array of Strings that store fragments of the template literal. We use substitutions to split the original template literal.
- the rest parameter `...substitutionValues` contains the values of `${...}` substitutions.

For the sake of simplicity, suppose that we only use alphanumeric characters in template substitutions. To understand how `literalFragments` are constructed, study the behavior of the JavaScript `split` method:

```
let emulatedSubs = '${sub1}abc ${sub2} def${sub3}'
        .split( /\${\w*}/ );
console.log(emulatedSubs);
```



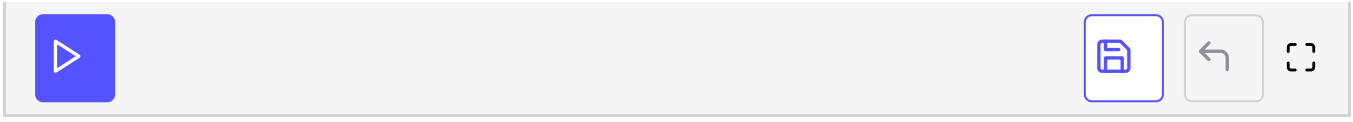
> ["", "abc ", " def", ""]

The `emulatedSubs` array contains all text fragments in order. The `i`th element of `emulatedSubs` is before the `(i+1)`th argument of the tag function, representing the substitution `${sub_i}`.

Let's now observe how the real `literalFragments` are constructed. We will create a tag function that prints all of its arguments. Let's execute this tag function on the template `${sub1}abc ${sub2} def${sub3}`.

```
let sub1=1, sub2=2, sub3 = 3;
( (x, ...subs) => {


        console.log( x, ...subs );
    })`${sub1}abc ${sub2} def${sub3}`
```



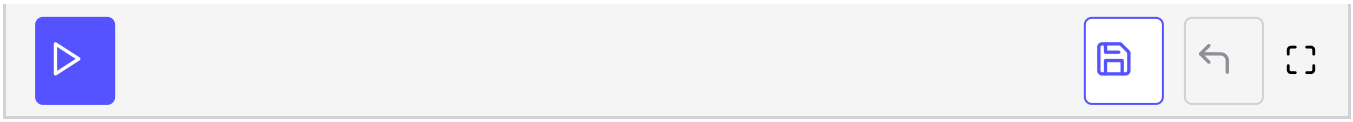
```
> ["", "abc ", " def", "", raw: Array[4]] 1 2 3
```

There is one small difference in the construction of the `literalFragments` array: the array has an associative property `raw`, containing the same four literal fragments as raw values.

As a first real example, let's create a salutation tag.

```
let salutation = literalFragments =>
    'Hello, ' + literalFragments[0];

console.log( salutation`Ashley` );
```



```
> "Hello, Ashley"
```

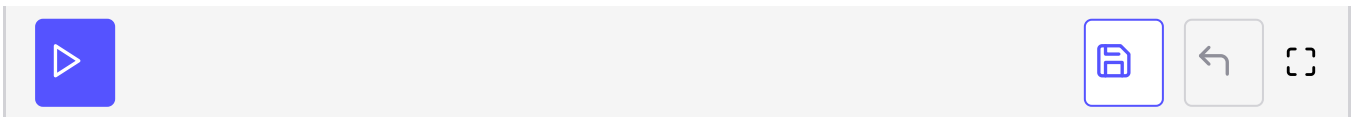
If variable substitutions occur inside the template literal, their values can also be manipulated using tag functions.

```
let price = 5999.9;
let currencySymbol = '€';
let productName = 'Titanium Toothbrush';

let formatCurrency = function( currency, amount ) {
    return amount.toFixed(2) + currency;
}

let format = (textArray, ...substitutions) => {
    let template = textArray[0];
    template += substitutions[0];
    template += textArray[1];
    template += formatCurrency( substitutions[1], substitutions[2] );
    template += textArray[3];

    return template;
};
```



In the `format` function, we can access all variable substitutions and template fragments, and we can concatenate them in any order. Substitutions come from evaluating the values of `productName`, `currencySymbol`, and `price` in the scope of the template evaluation.

The result of the tagged template above looks like this:

```
<div class="js-product">
    Product: Titanium Toothbrush
</div>
<div class="js-price">
    Price: 5999.90€
</div>
```

Even though in this specific case, the `format` function relies on knowledge of the structure of the template, in some cases, templates have a variable number of substitutions. One of the exercises will require you to create tag functions handling a variable number of substitutions.

Now, let's solve some exercises on this before moving on to new concepts.