Using Code for Ciphers

As mentioned in the introduction to ciphers resource, using programming to help visualise and see ciphers in action is paramount to the decrypting process. Throughout these resources, you will gain confidence in programming with ciphers.

The easiest way to visualise a cipher is as a function.

The plain text goes in, the function works its magic, the cipher text comes out.

Except really, the cipher isn't magic at all, and is just a series of instructions to apply to the plain text.

Therefore, in this resource you will be able to use code to see the process of visualising a cipher.

Important Note

All programs will be written in JavaScript, HTML, and using a mini-library called Nanny State. There will be sometimes reference to another mini-library called JOG-List.

No prior knowledge of JavaScript or these libraries are required, as all actions and techniques will be explained. However, in order to make the most of these resources, it is recommended that you are familiar with basic JavaScript and the concepts of programming.

If you are inexperienced in both JavaScript and HTML, you can check out the programming resources for beginners that will help build a basis.

Your First Cipher Program

To keep things simple, we will build a very small app that will allow you to input plain text, press a button to encipher the text, and see the changes.

For the initial program, we will use the imaginary cipher from the introduction resource: replace the T's with ? and reverse the text.

Step 1

First and foremost, we need to set up the initial Nanny State:

```
1  import Nanny from "https://cdn.skypack.dev/nanny-state"
2
3  const View = state => {
4    return state.HTML``
5  }
6
7  const State = {
8    View
9  }
10
11 Nanny(State)
```

In line 1, we have the function ‘Nanny’ being imported, so that we can use it and call it in line 11.

In line 7, we have the initial ‘State’ being set up. As you can see, this is an object data structure. This is where we will host all of our properties (similar to variables), so that we can access and update them.

Any updates to the ‘State’ result in the page being rerendered, meaning this will automatically update the page as well.

In line 3, we have the standard ‘View’. This is a function that takes ‘state’ as a parameter and returns the ‘state.HTML’ method. This calls the inbuilt ‘state’ method that allows us to write HTML directly into the template literals ``, so that it will appear on our page.

If you would like to understand more about Nanny State, make sure you click the link (on Nanny State) above and check out the GitHub page.

Step 2

Next, we need our property called ‘plaintext’. To add this to ‘State’ we just use the standard adding a property to an object in JavaScript.

```
7  const State = {
8    plaintext: '',
9    View
10 }
```

As you can see, the value of ‘plaintext’ has a default value of an empty string (the value you set the property to in ‘State’ is its default value).

Now that we have this, we can start adding some HTML.

In the View function and within the ‘state.HTML’ template literals, you need to add:

```
const View = state => {
  return state.HTML`
  <h1>Create Your Own Cipher</h1>
  <form>
    <input type="text" name="plaintext" placeholder="Enter Message">
    <input type="submit">
  </form>

  <p>${state.plaintext}</p>`
}
```

Firstly, we have a ‘h1’ tag, to create our main heading. This can have any text between the two tags.

Second, we have a ‘form’ tag, with two inputs. One input is where we want the user to input their message (the plaintext). The second input is just a button that will allow the user to submit the message (however, they can also do this through hitting the enter key).

Finally, we have a ‘p’ tag, to create a paragraph where the plain text (and later cipher text) will be shown. The ‘${}’ symbols are used in order to insert changing values into text.

Your screen should look like this:

Step 3

When the user submits their message in the form, we need to grab the message and put it as the value of the ‘plaintext’ property in ‘State’.

To do this we need to add an ‘inline event handler’ (the only type of event handlers that get used in Nanny State), of ‘onsubmit’ to run the instructions we want to execute.

```
const View = state => {
  const enter = event => {
    event.preventDefault()
    state.Update({plaintext: event.target.plaintext.value.toUpperCase()})
    event.target.plaintext.value = ''
  }

  return state.HTML`
  <h1>Create Your Own Cipher</h1>
  <form onsubmit=${enter}>
    <input type="text" name="plaintext" placeholder="Enter Message">
    <input type="submit">
  </form>

  <p>${state.plaintext}</p>`
}
```

As you can see, in the opening ‘form’ tag the ‘onsubmit’ inline event listener has been added. And when the user submits the form, the event handler ‘enter’ will run.

We have to create our own event handlers, and as you can see we execute three instructions in ours.

First, we run the inbuilt ‘.preventDefault()’ method. The default event when a user submits a form is to refresh the page, and hence will always return a 404 error of ‘page not found’. Therefore, by cancelling this event, we can work with the value submitted by the form as we want.

To access the actual value the user has input we need to use the chain:

event.target.[nameOfInput].value

In our case, the ‘nameOfInput’ is ‘plaintext’.

Hence, the message the user has submitted can be accessed by ‘event.target.plaintext.value’ (as you can see).

Therefore, the second step is to replace the current value of the ‘plaintext’ property in State, with the user’s message. We do this by using Nanny State’s inbuilt ‘.Update()’ method.

By writing ‘state.Update()’, we can essentially overwrite any properties we want in the State, and the ones we don’t overwrite will automatically remain the same.

In this case, we need to overwrite the ‘plaintext’ property. Therefore, we use the object literals, {}, and rewrite the property of ‘plaintext’ to have the value ‘event.target.plaintext.value’. However, we also add the method that will change the message to upper case (this will just make things easier for now).

Then finally, we reset the input form so that there is nothing inside.

You can now try out what the code does:

Step 4

Our next step is to add a function that will change the text (acting as a cipher), that will run when we click a button.

The first function we’ll add is the reverse function.

To begin we need to add a button to the HTML:

```
18
19      <h2>Buttons: </h2>
20      <button>REVERSE</button>`
```

Next, we need to add an inline event listener to this button and an event handler called ‘reverseText’. For buttons we use the ‘onclick’ event listener.

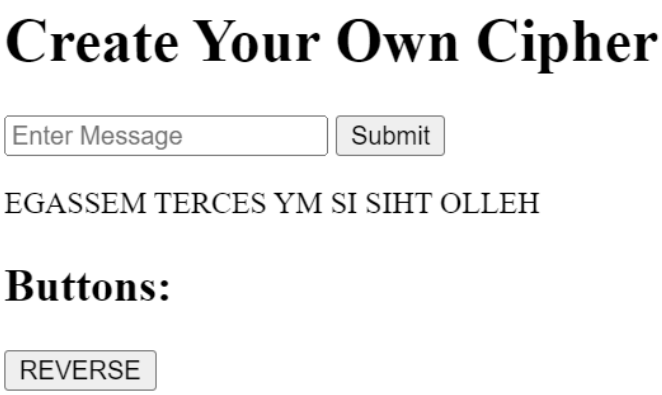
```
import Nanny from "https://cdn.skypack.dev/nanny-state"
import {reverse} from "https://cdn.skypack.dev/jog-list"

const View = state => {
  const reverseText = event => state.Update({plaintext: reverse(state.plaintext)})

  const enter = event => {
    event.preventDefault()
    state.Update({plaintext: event.target.plaintext.value.toUpperCase()})
    event.target.plaintext.value = ''
  }

  return state.HTML`
  <h1>Create Your Own Cipher</h1>
  <form onsubmit=${enter}>
    <input type="text" name="plaintext" placeholder="Enter Message">
    <input type="submit">
  </form>

  <p>${state.plaintext}</p>

  <h2>Buttons: </h2>
  <button onclick=${reverseText}>REVERSE</button>`
}
```

As you can see, the function ‘reverse’ has been imported from the library mentioned above. This function takes a string or a list as a parameter and simply reverses it.

Note: the object literals around ‘reverse’ in the import line are necessary.

Therefore, in the reverseText event handler, we can use the ‘update’ method again and the imported ‘reverse’ function, to change the plaintext so that it is the reverse of the message the user submitted.

As mentioned before, running the ‘update’ method means that the page automatically gets rerendered, therefore you will be able to see the changes to the text immediately:



Step 5

If you want to change the message, the user simply has to resubmit another message. However, it's much cleaner to have an official 'clear' button.

Therefore, you can add a button above all the cipher buttons like so:

```
const View = state => {
  const reverseText = event => state.Update({plaintext: reverse(state.plaintext)})

  const enter = event => {
    event.preventDefault()
    state.Update({plaintext: event.target.plaintext.value.toUpperCase()})
    event.target.plaintext.value = ''
  }

  const clear = event => state.Update({plaintext: ''})

  return state.HTML`
  <h1>Create Your Own Cipher</h1>
  <form onsubmit=${enter}>
    <input type="text" name="plaintext" placeholder="Enter Message">
    <input type="submit">
  </form>

  <p>${state.plaintext}</p>
  <button onclick=${clear}> CLEAR </button>

  <h2>Buttons: </h2>
  <button onclick=${reverseText}>REVERSE</button>`
}
```

The button is called 'CLEAR' and has the event handler 'clear'.

As you can see, the event handler uses the 'update' method again and resets the value of 'plaintext' to its default value of an empty string.

Step 6

Now, we can add as many cipher buttons as we want. To complete the example from the first resource, we'll add a button that will replace all the T's with ?s.

Here is the button and the event handler (working with the same principle as the reverse button.

```
    <button onclick=${replaceT}>Replace T with ?</button>`
```

```
const replaceT = event => state.Update({plaintext: [...state.plaintext].map(character => character === "T" ? "?" : character).join``})
```

The event handler is the most interesting, as we need to do some nifty string manipulation to be able to enact this function.

Firstly, we turn the plaintext into an array. We do this by unpacking the string into a set of array literals: '[...state.plaintext]'.

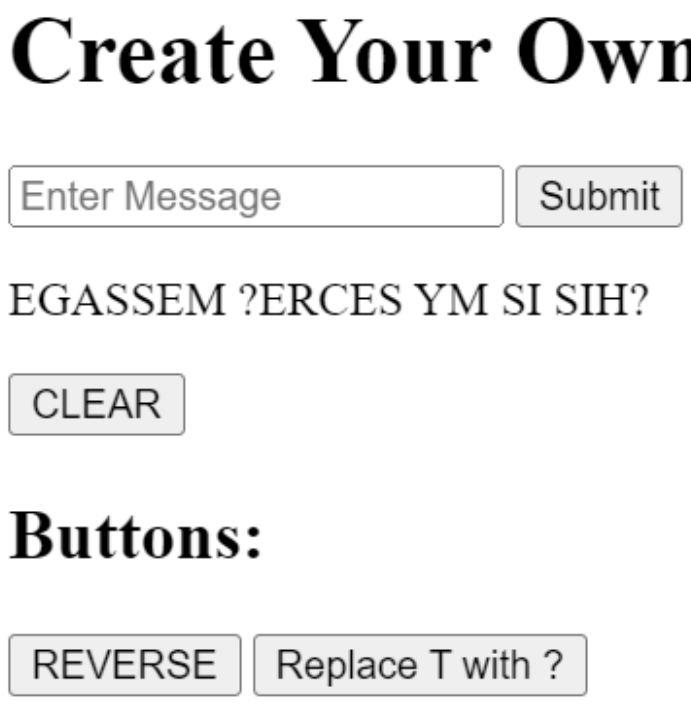
This will turn the message (which is currently a string) into an array where each character of the string is an item in the array. For example, if our message is "HELLO". Then after unpacking into the array, we will have ["H", "E", "L", "L", "O"].

Then, we can use the 'map' method (if you have never met the map method, please check out some of the programming resources as we will be using these a LOT). We map over every character in the array (and hence the string), and if the character is a 'T' then we change it to a '?'. Otherwise we leave the character as it is.

Much like the map method, if you have never met ternary operators please check them out on the programming resources. In short, they are essentially very quick ways of writing an 'if' statement:

Condition ? what to do if true : what to do if false

Now, you should be able to see our made up cipher in action!



Step 7

You can see the final app here.

There is also another function, where all the letters are replaced with their position in the alphabet. See if you can figure out how this is working.

Now you should be able to have a go at adding your own functions, however simple, to make your own unique cipher.

Here are some ideas:

- Replace some other letter with a different symbol
- Replace all the characters with their ASCII code
- Replace all the letters with their morse code value