A Beginner's Guide to Web Scripting with Lua and Fengari.

Fengari allows lua to be used as a web-scripting language *i.e.* we can use lua instead of javascript (JS) to control the behaviour of our web pages and we have access to all of the available JS libraries.

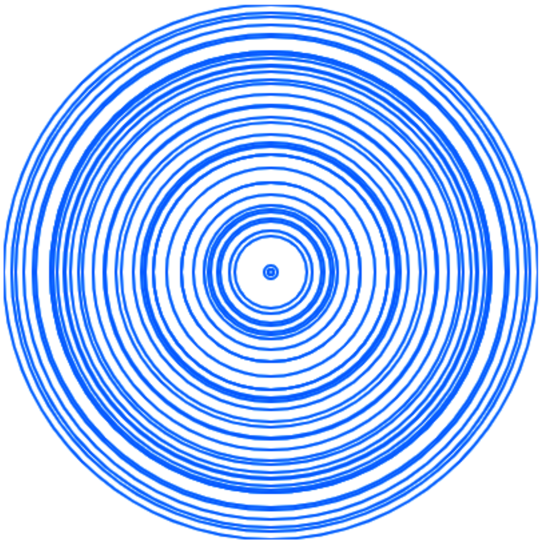
To run fengari, you need to download the file *fengari_web.js* from the *src* folder here. Note that I have renamed it so that the **hyphen** in the original file name is replaced by an **underscore** to stop linux being confused, since the hyphen can be interpreted as a shell command option. It's probably best to use Chrome and in the drop-down settings menu go to ***More Tools*** and then ***Developer Tools*** where you can select the Chrome **Console**, which shows you any errors and printed output from the scripts.

Example 1. Drawing and animation with the html canvas.

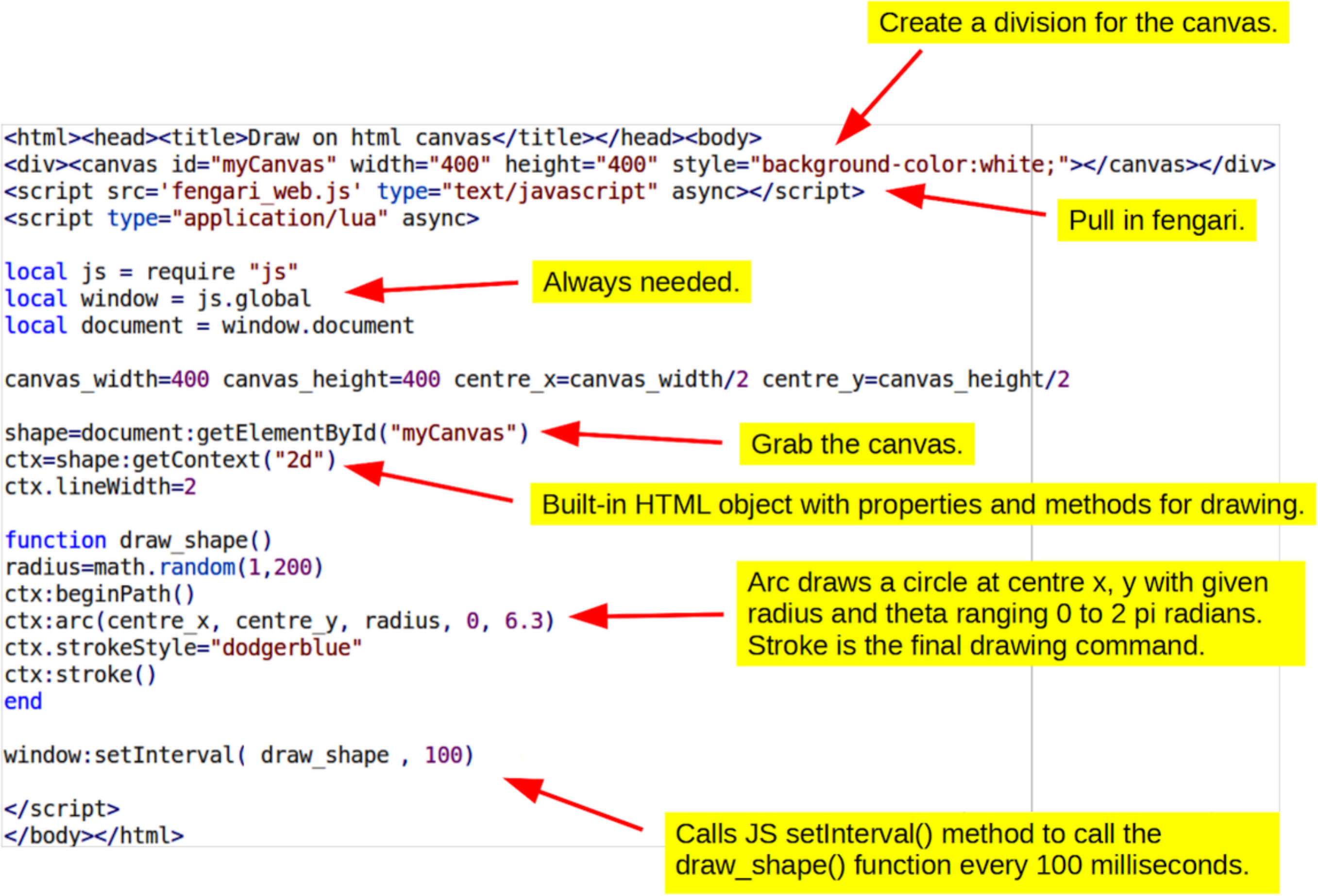
Traditionally a job for JS *e.g.* as introduced here, we can achieve the same thing with lua and fengari. Looking at the JS examples in the link, we can convert these commands to lua with some swapping around of dots and colons. The example below draws concentric circles of random radius on the canvas, a new circle appearing every 100 milliseconds.

```
<html><head><title>Draw on html canvas</title></head><body>
<div><canvas id="myCanvas" width="400" height="400" style="background-color:white;"></canvas></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js = require "js"
local window = js.global
local document = window.document

canvas_width=400 canvas_height=400 centre_x=canvas_width/2 centre_y=canvas_height/2

shape=document:getElementById("myCanvas")
ctx=shape:getContext("2d")
ctx.lineWidth=2

function draw_shape()
radius=math.random(1,200)
ctx:beginPath()
ctx:arc(centre_x, centre_y, radius, 0, 6.3)
ctx.strokeStyle="dodgerblue"
ctx:stroke()
end

window:setInterval(draw_shape, 100)

</script></body></html>
```

The output is animated but a snapshot of it looks something like this:



An explanation of the lua code is given below.



```
<html><head><title>Draw on html canvas</title></head><body>
<div><canvas id="myCanvas" width="400" height="400" style="background-color:white;"></canvas></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js = require "js"
local window = js.global
local document = window.document

canvas_width=400 canvas_height=400 centre_x=canvas_width/2 centre_y=canvas_height/2

shape=document:getElementById("myCanvas")
ctx=shape:getContext("2d")
ctx.lineWidth=2

function draw_shape()
radius=math.random(1,200)
ctx:beginPath()
ctx:arc(centre_x, centre_y, radius, 0, 6.3)
ctx.strokeStyle="dodgerblue"
ctx:stroke()
end

window:setInterval( draw_shape , 100)

</script>
</body></html>
```

Create a division for the canvas.

Pull in fengari.

Always needed.

Grab the canvas.

Built-in HTML object with properties and methods for drawing.

Arc draws a circle at centre x, y with given radius and theta ranging 0 to 2 pi radians. Stroke is the final drawing command.

Calls JS setInterval() method to call the draw_shape() function every 100 milliseconds.

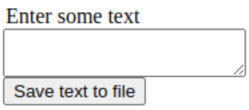
The numerous other drawing methods of the canvas can thus be accessed through fengari. Note that a small minority of these methods need parameters to be given as a JS array. JS arrays are of the form: [12, 3, 3] and this one could, for example, be created with fengari using the command *window:Array(12, 3, 3)*. An array containing a single text element is created in the example below.

Example 2. To save text entered in a textbox to a file.

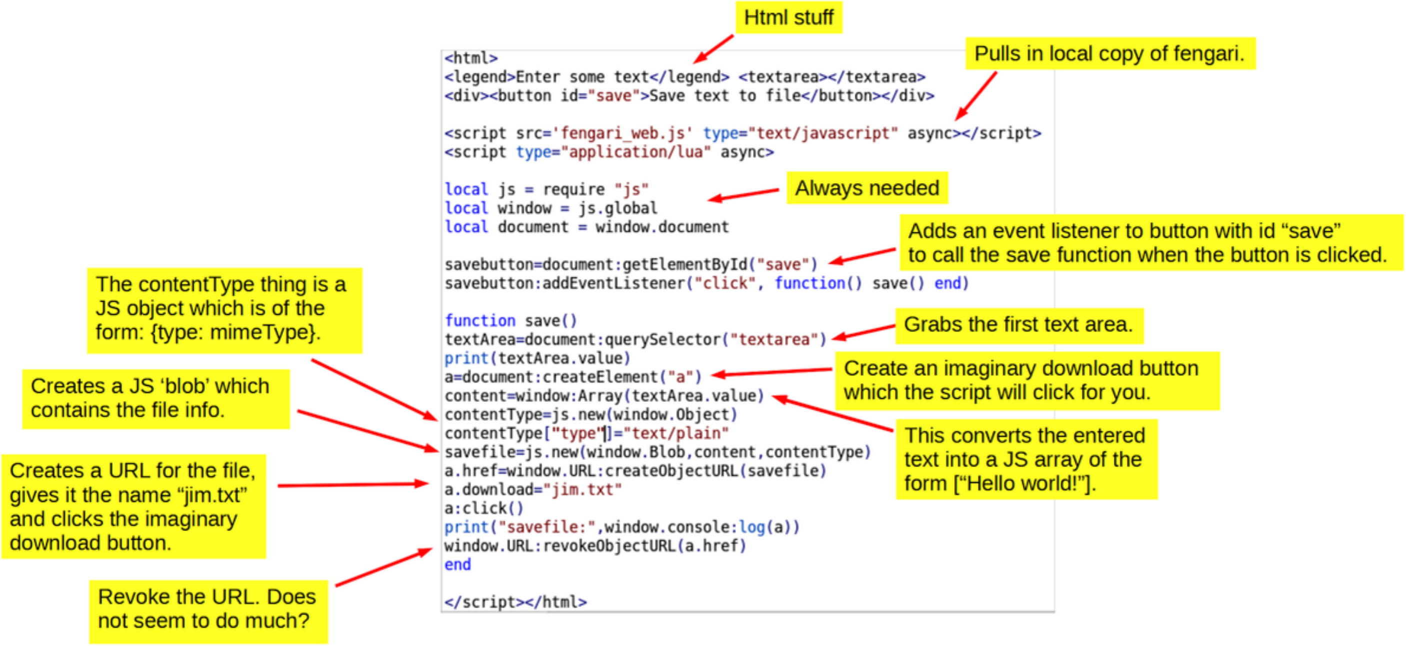
```
<html>
<legend>Enter some text</legend> <textarea></textarea>
<div><button id="save">Save text to file</button></div>
```

```
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js = require "js"
local window = js.global
local document = window.document

savebutton=document:getElementById("save")
savebutton:addEventListener("click", function() save() end)

function save()
textArea=document:querySelector("textarea")
print(textArea.value)
a=document:createElement("a")
content=window:Array(textArea.value)
contentType=js.new(window.Object)
contentType["type"]="text/plain"
savefile=js.new(window.Blob,content,contentType)
a.href=window.URL:createObjectURL(savefile)
a.download="jim.txt"
a:click()
window.console:log(a)
window.URL:revokeObjectURL(a.href)
end

</script></html>
```

The html above generates a very simple form in which one can enter text and save it to a local file (jim.txt).



Here is some explanation of what the lua script does.



```
<html>
<legend>Enter some text</legend> <textarea></textarea>
<div><button id="save">Save text to file</button></div>

<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js = require "js"
local window = js.global
local document = window.document

savebutton=document:getElementById("save")
savebutton:addEventListener("click", function() save() end)

function save()
textArea=document:querySelector("textarea")
print(textArea.value)
a=document:createElement("a")
content=window:Array(textArea.value)
contentType=js.new(window.Object)
contentType["type"]="text/plain"
savefile=js.new(window.Blob,content,contentType)
a.href=window.URL:createObjectURL(savefile)
a.download="jim.txt"
a:click()
print("savefile:",window.console:log(a))
window.URL:revokeObjectURL(a.href)
end

</script></html>
```

Html stuff

Pulls in local copy of fengari.

Always needed

Adds an event listener to button with id “save” to call the save function when the button is clicked.

Grabs the first text area.

Create an imaginary download button which the script will click for you.

This converts the entered text into a JS array of the form [“Hello world!”].

The contentType thing is a JS object which is of the form: {type: mimeType}.

Creates a JS ‘blob’ which contains the file info.

Creates a URL for the file, gives it the name “jim.txt” and clicks the imaginary download button.

Revoke the URL. Does not seem to do much?

Note that in the line:

savebutton:addEventListener("click", function() save() end)

we could have said simply:

savebutton:addEventListener("click", save) as in Example 1 and this would work fine, too. However by wrapping the function in a function we can add other lua commands and extra functions to the button's callback, if needed, e.g.

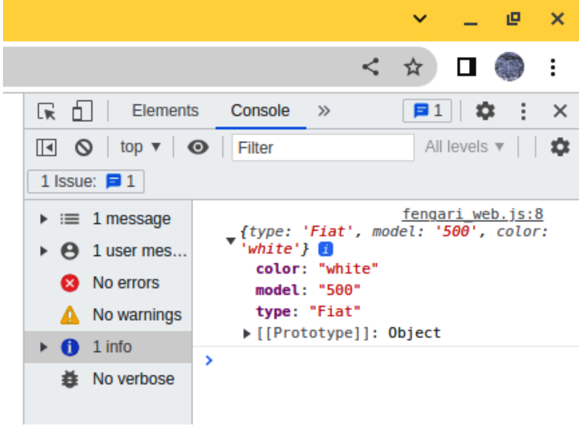
savebutton:addEventListener("click", function() save() print("Done") end)

For this reason, we will stick to the slightly longer wrapped format for the remaining examples.

In this example, in addition to making a JS array, we also made a JS object. JS objects are of the form: *car = {type:"Fiat", model:"500", color:"white"}* so we could make the same thing with lua and fengari using the following commands:

car=js.new(window.Object)
car["type"]="Fiat"
car["model"]="500"
car["color"]="white"

In JS you can print the contents of an object by using the *console.log(object_name)* command. However, in lua we need to use the following syntax: *window.console:log(object_name)* instead. For example, with the car object above, the command *window.console:log(car)* would give us the following information about it in the Chrome console.

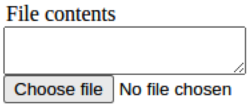


```
fengari_web.js:8
▾{type: 'Fiat', model: '500', color: 'white'}
   color: "white"
   model: "500"
   type: "Fiat"
  ▸[[Prototype]]: Object
```

Example 3. To choose a file and display it in a web page.

The following code makes a simple file chooser button as shown beneath the black box:

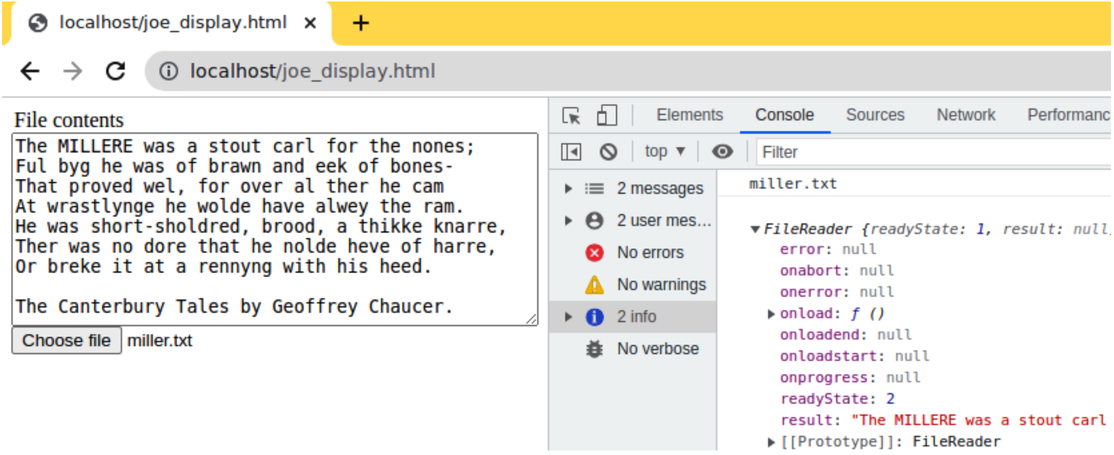
```
<html>
<legend>File contents</legend> <textarea></textarea>
<div><input type="file" id="file"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>
local js = require "js"
local window = js.global
local document = window.document

fileChooser=document:getElementById("file")
fileChooser:addEventListener("change", function() read() end)

textArea=document:querySelector("textarea")

function read()
myFile=fileChooser.files[0]
print(myFile.name)
reader=js.new(window.FileReader)
reader.onload=function() textArea.value=reader.result end
reader:readAsText(myFile)
window.console:log(reader)
end

</script></html>
```



File contents

Choose file No file chosen

This allows us to select a local text file and display its contents in the web page, *e.g.* in the picture below, we have chosen a random file called *miller.txt*. Of course, we may need to resize the text area by clicking and dragging in the lower right corner to display the entire file.



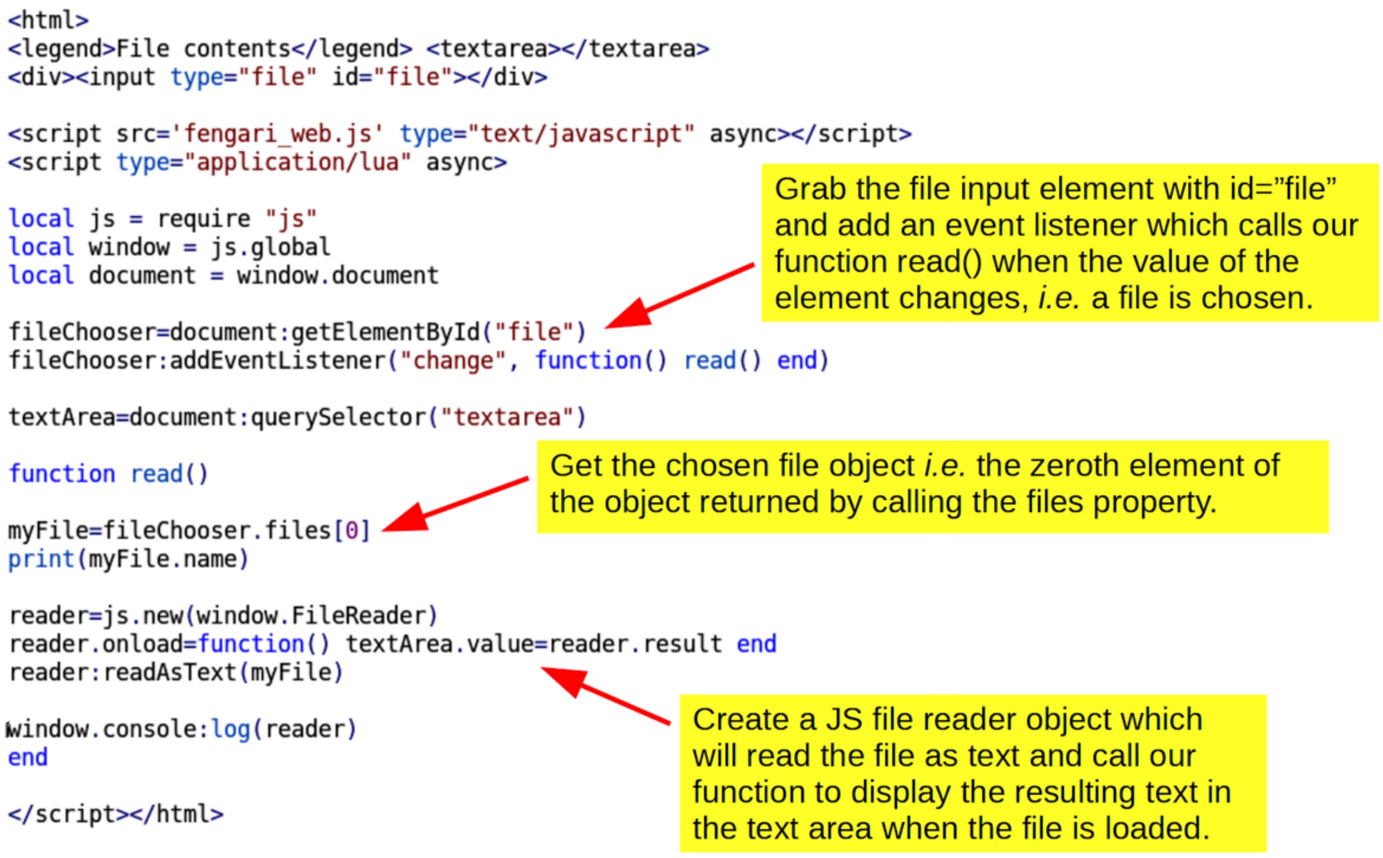
File contents

```
The MILLERE was a stout carl for the nones;
Ful byg he was of brawn and eek of bones-
That proved wel, for over al ther he cam
At wrastlynge he wolde have alwey the ram.
He was short-sholdred, brood, a thikke knarre,
Ther was no dore that he nolde heve of harre,
Or breke it at a rennyng with his heed.

The Canterbury Tales by Geoffrey Chaucer.
```

Choose file miller.txt

```
miller.txt

▼FileReader {readyState: 1, result: null,
   error: null
   onabort: null
   onerror: null
  ▶onload: ƒ ()
   onloadend: null
   onloadstart: null
   onprogress: null
   readyState: 2
   result: "The MILLERE was a stout carl
  ▶[[Prototype]]: FileReader
```

An explanation of the code is given below.



```
<html>
<legend>File contents</legend> <textarea></textarea>
<div><input type="file" id="file"></div>

<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js = require "js"
local window = js.global
local document = window.document

fileChooser=document:getElementById("file")
fileChooser:addEventListener("change", function() read() end)

textArea=document:querySelector("textarea")

function read()

myFile=fileChooser.files[0]
print(myFile.name)

reader=js.new(window.FileReader)
reader.onload=function() textArea.value=reader.result end
reader:readAsText(myFile)

window.console:log(reader)
end

</script></html>
```

Grab the file input element with id="file" and add an event listener which calls our function read() when the value of the element changes, *i.e.* a file is chosen.

Get the chosen file object *i.e.* the zeroth element of the object returned by calling the files property.

Create a JS file reader object which will read the file as text and call our function to display the resulting text in the text area when the file is loaded.

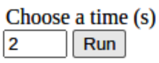
Example 4. Display an hourglass while a process is running.

Often you want to have a web form with a submit button that starts a client-side process and while this is running we want to disable the form and display an hourglass to inform the user that things are happening in the background. In this example we use an animated gif hourglass.gif which must first be downloaded from the link to your working directory. The lua script in the html file looks like this:

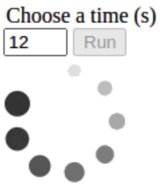
```
<html><legend>Choose a time (s)</legend>
<input type="number" min="2" max="12" value="2" step="2" id="timer">
```

```
<button id="run">Run</button><div id="info" style="height:100px;"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>
local js=require "js"
local window=js.global
local document=window.document

info=document:getElementById("info")
runButton=document:getElementById("run")
runButton:addEventListener("click", function() start() end)

function start()
runTime=document:getElementById("timer")
runTime=1000*runTime.value
runButton.disabled=true
info.style.backgroundImage="URL(hourglass.gif)"
info.style.backgroundRepeat="no-repeat"
window:setTimeout(function() stop() end, runTime)
end

function stop()
runButton.disabled=false
info.style.background="none"
end

</script></html>
```

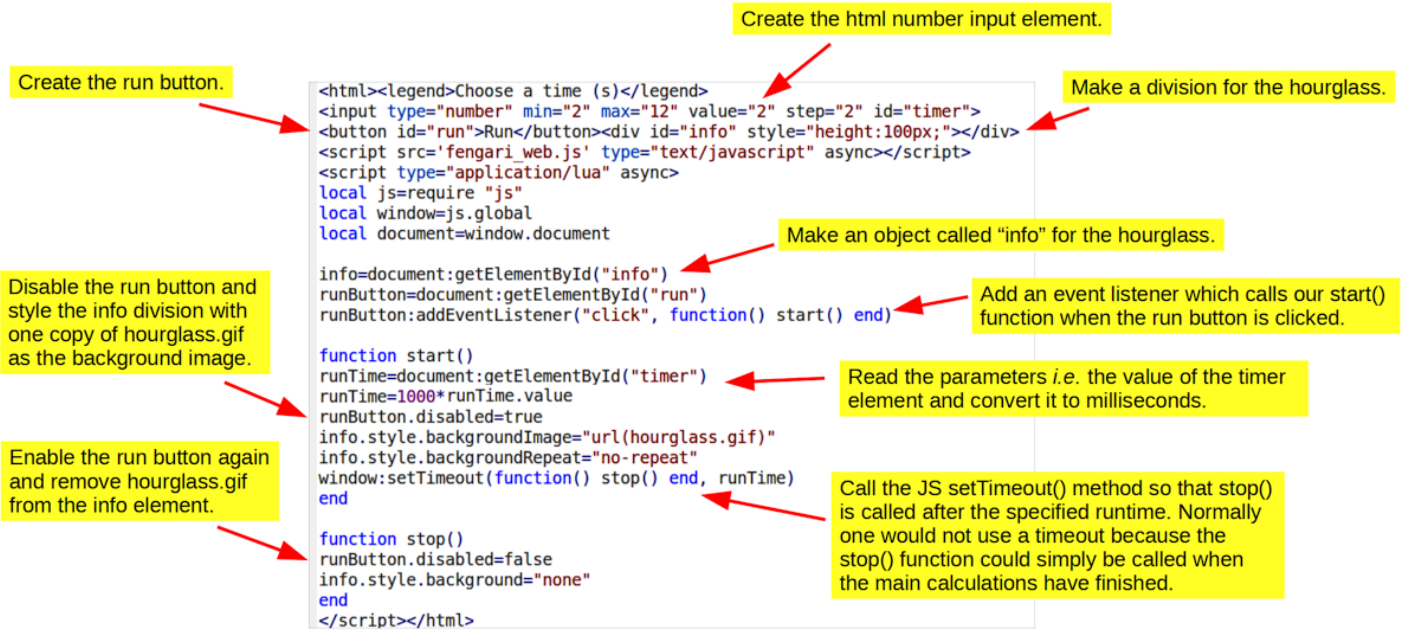
It gives us a mini-form as follows:



and when running it looks like this:



Note that the run button is disabled while the background process is running. An explanation of the code is given below.



Create the html number input element.

Create the run button.

Make a division for the hourglass.

```
<html><legend>Choose a time (s)</legend>
<input type="number" min="2" max="12" value="2" step="2" id="timer">
<button id="run">Run</button><div id="info" style="height:100px;"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>
local js=require "js"
local window=js.global
local document=window.document

info=document:getElementById("info")
runButton=document:getElementById("run")
runButton:addEventListener("click", function() start() end)

function start()
runTime=document:getElementById("timer")
runTime=1000*runTime.value
runButton.disabled=true
info.style.backgroundImage="url(hourglass.gif)"
info.style.backgroundRepeat="no-repeat"
window:setTimeout(function() stop() end, runTime)
end

function stop()
runButton.disabled=false
info.style.background="none"
end
</script></html>
```

Make an object called “info” for the hourglass.

Disable the run button and style the info division with one copy of hourglass.gif as the background image.

Add an event listener which calls our start() function when the run button is clicked.

Read the parameters *i.e.* the value of the timer element and convert it to milliseconds.

Enable the run button again and remove hourglass.gif from the info element.

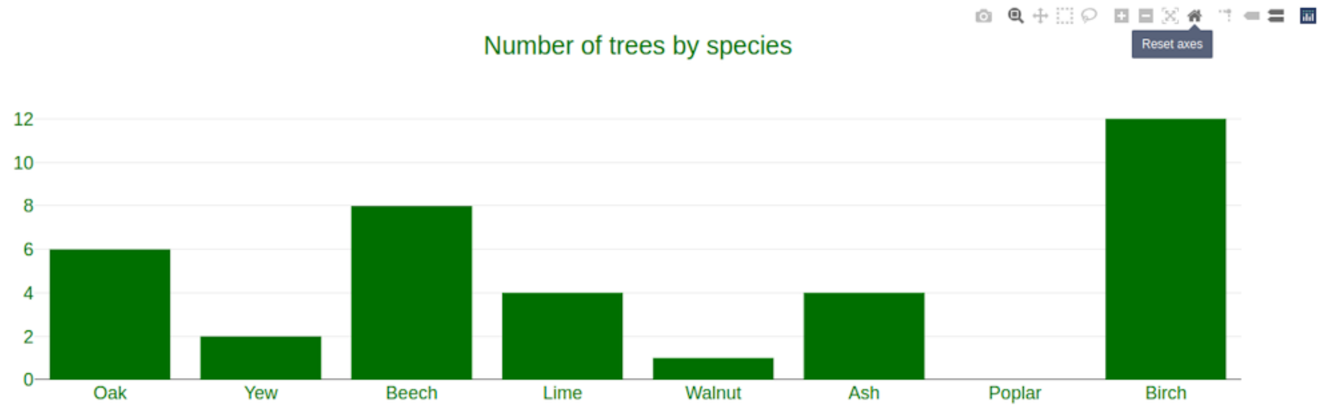
Call the JS setTimeout() method so that stop() is called after the specified runtime. Normally one would not use a timeout because the stop() function could simply be called when the main calculations have finished.

Example 5. Using the interactive graph plotting tool Plotly.

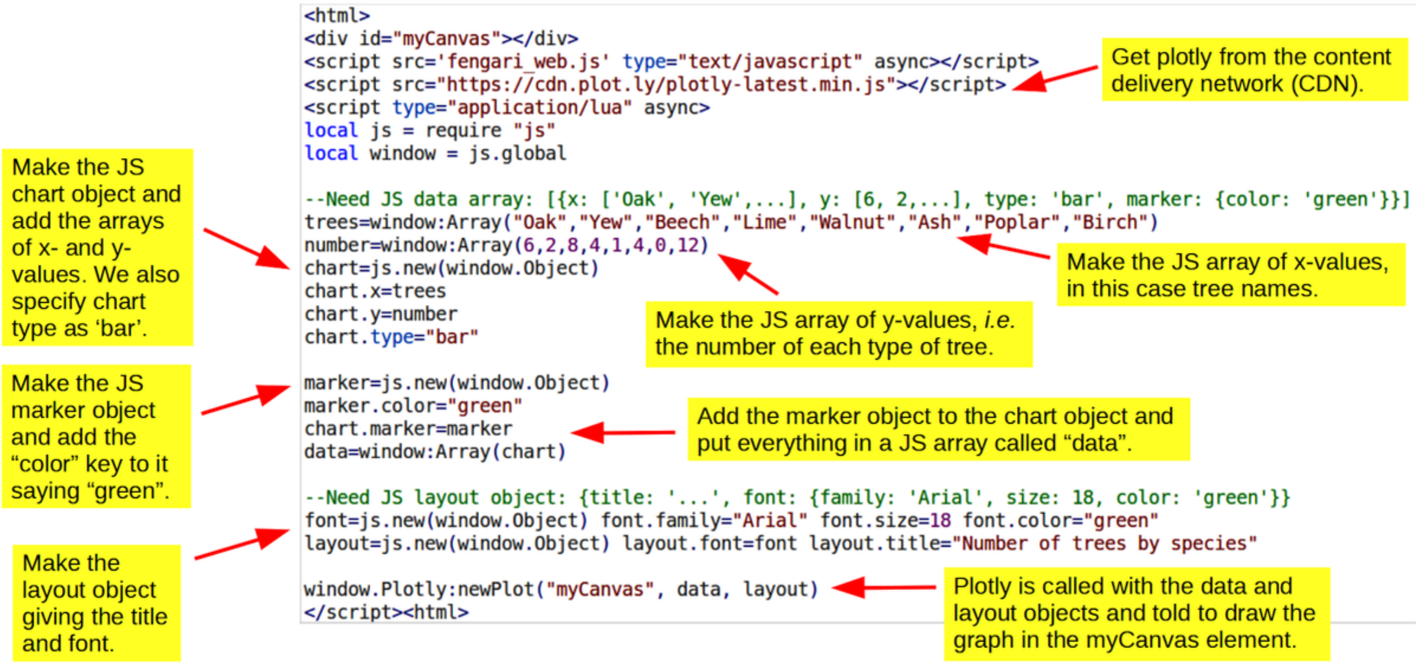
This example will illustrate how to draw highly interactive graphs in a web page using the plotting tool Plotly. Imagine that we want to draw a bar chart showing the number of trees of different types in a small wood. The data are as follows: oak: 6, yew: 2, beech: 8, lime: 4, *etc*. In the code below you will see in the lua comments that Plotly has fairly complex data structures *i.e.* you need to assemble the data and plotting instructions into nested objects and arrays but this can be done fairly easily with lots of short commands, some of which I have put side-by-side on the same line.

```
<html>
<div id="myCanvas"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
<script type="application/lua" async>
local js = require "js"
local window = js.global

--Need JS data array: [{x: ['Oak', 'Yew',...], y: [6, 2,...], type: 'bar', marker: {color: 'green'}}]
trees=window:Array("Oak","Yew","Beech","Lime","Walnut","Ash","Poplar","Birch")
number=window:Array(6,2,8,4,1,4,0,12)
chart=js.new(window.Object)
chart.x=trees
chart.y=number
chart.type="bar"

marker=js.new(window.Object)
marker.color="green"
chart.marker=marker
data=window:Array(chart)

--Need JS layout object: {title: '...', font: {family: 'Arial', size: 18, color: 'green'}}
font=js.new(window.Object) font.family="Arial" font.size=18 font.color="green"
layout=js.new(window.Object) layout.font=font layout.title="Number of trees by species"

window.Plotly:newPlot("myCanvas", data, layout)
</script><html>
```

Anyway, the result is quite nice and, as you can see below, Plotly provides little on-screen tools for zooming and panning the graph, *etc.*, which will impress the users of your web site!



An annotated version of the code is given below.



```
<html>
<div id="myCanvas"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
<script type="application/lua" async>
local js = require "js"
local window = js.global

--Need JS data array: [{x: ['Oak', 'Yew',...], y: [6, 2,...], type: 'bar', marker: {color: 'green'}}]
trees=window:Array("Oak","Yew","Beech","Lime","Walnut","Ash","Poplar","Birch")
number=window:Array(6,2,8,4,1,4,0,12)
chart=js.new(window.Object)
chart.x=trees
chart.y=number
chart.type="bar"

marker=js.new(window.Object)
marker.color="green"
chart.marker=marker
data=window:Array(chart)

--Need JS layout object: {title: '...', font: {family: 'Arial', size: 18, color: 'green'}}
font=js.new(window.Object) font.family="Arial" font.size=18 font.color="green"
layout=js.new(window.Object) layout.font=font layout.title="Number of trees by species"

window.Plotly:newPlot("myCanvas", data, layout)
</script><html>
```

Get plotly from the content delivery network (CDN).

Make the JS chart object and add the arrays of x- and y-values. We also specify chart type as 'bar'.

Make the JS array of x-values, in this case tree names.

Make the JS array of y-values, *i.e.* the number of each type of tree.

Make the JS marker object and add the "color" key to it saying "green".

Add the marker object to the chart object and put everything in a JS array called "data".

Make the layout object giving the title and font.

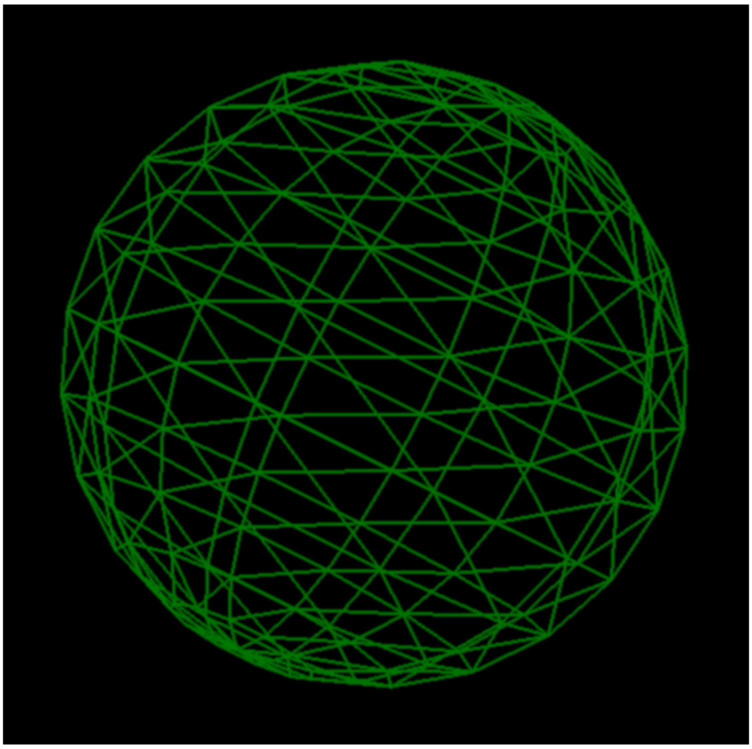
Plotly is called with the data and layout objects and told to draw the graph in the myCanvas element.

Example 6. Drawing animated 3D graphics with the three.js library.

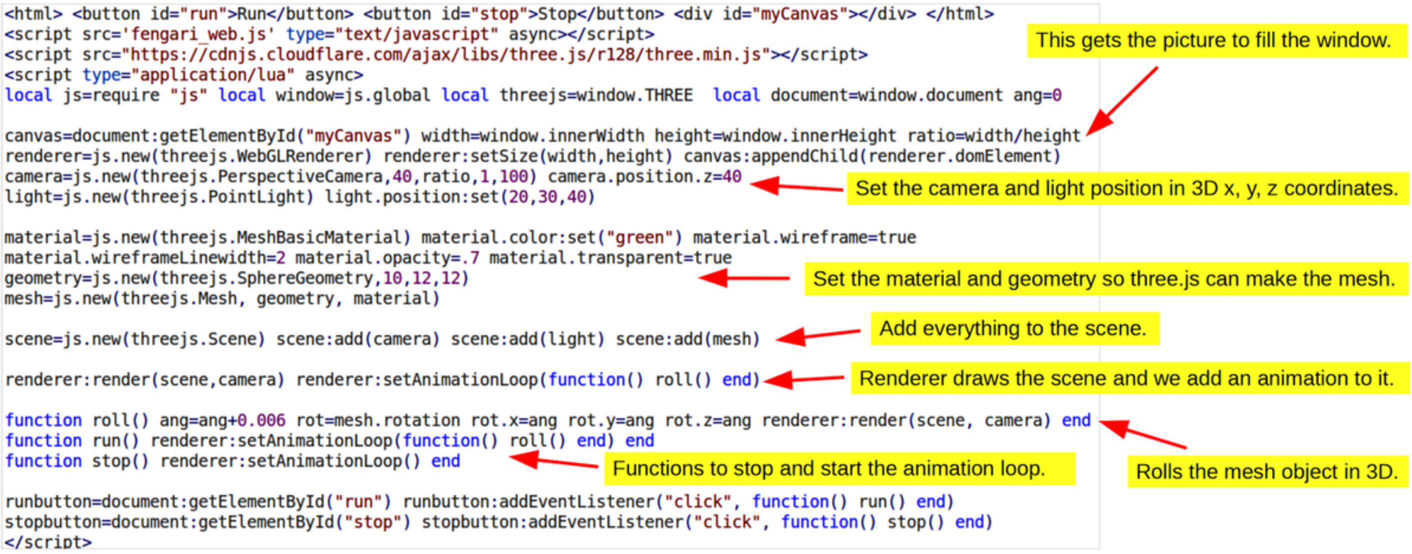
Drawing 3D objects generally requires lots of code but three.js simplifies things considerably, although in the example below I still had to combine quite a few lines together to keep everything within one page. However, the commands are grouped together logically, I hope, to indicate roughly the process of rendering a scene, which in this case is a rolling wireframe sphere.

```
<html> <button id="run">Run</button> <button id="stop">Stop</button> <div id="myCanvas"></div> </html>
<script src='fengari_web.js' type="text/javascript" async></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/three.min.js"></script>
<script type="application/lua" async>
local js=require "js" local window=js.global local threejs=window.THREE  local document=window.document ang=0

canvas=document:getElementById("myCanvas") width=window.innerWidth height=window.innerHeight ratio=width/height
renderer=js.new(threejs.WebGLRenderer) renderer:setSize(width,height) canvas:appendChild(renderer.domElement)
camera=js.new(threejs.PerspectiveCamera,40,ratio,1,100) camera.position.z=4
light=js.new(threejs.PointLight) light.position:set(20,30,40)

material=js.new(threejs.MeshBasicMaterial) material.color:set("green") material.wireframe=true
material.wireframeLinewidth=2 material.opacity=.7 material.transparent=true
geometry=js.new(threejs.SphereGeometry,10,12,12)
mesh=js.new(threejs.Mesh, geometry, material)

scene=js.new(threejs.Scene) scene:add(camera) scene:add(light) scene:add(mesh)

renderer:render(scene,camera) renderer:setAnimationLoop(function() roll() end)

function roll() ang=ang+0.006 rot=mesh.rotation rot.x=ang rot.y=ang rot.z=ang renderer:render(scene, camera) end
function run() renderer:setAnimationLoop(function() roll() end) end
function stop() renderer:setAnimationLoop() end

runbutton=document:getElementById("run") runbutton:addEventListener("click", function() run() end)
stopbutton=document:getElementById("stop") stopbutton:addEventListener("click", function() stop() end)
</script>
```

The actual animation looks something like this which may not be very impressive as a still image, but it is quite hypnotic when animated. There are two html buttons (not shown) which are coded to turn the animation on or off.



The rendering process (*i.e.* drawing, painting or photographing the scene computationally) consists of defining a *camera* and a *light source* in a 3D coordinate system as shown here. We must also specify the *material* of the object being 'photographed' and its *geometry* - roughly speaking its shape. The program calculates a *mesh* which is like a polygon model of the object's surface, based on its geometry and the chosen material. In the language of three.js, we create a *scene* and add the *camera*, *light* and *mesh* to it before issuing a final command to *render* the scene. Of course it is insanely complicated and I can only pretend to understand it, but I have annotated the main steps below.



```
<html> <button id="run">Run</button> <button id="stop">Stop</button> <div id="myCanvas"></div> </html>
<script src='fengari_web.js' type="text/javascript" async></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/three.min.js"></script>
<script type="application/lua" async>
local js=require "js" local window=js.global local threejs=window.THREE  local document=window.document ang=0

canvas=document:getElementById("myCanvas") width=window.innerWidth height=window.innerHeight ratio=width/height
renderer=js.new(threejs.WebGLRenderer) renderer:setSize(width,height) canvas:appendChild(renderer.domElement)
camera=js.new(threejs.PerspectiveCamera,40,ratio,1,100) camera.position.z=40
light=js.new(threejs.PointLight) light.position:set(20,30,40)

material=js.new(threejs.MeshBasicMaterial) material.color:set("green") material.wireframe=true
material.wireframeLinewidth=2 material.opacity=.7 material.transparent=true
geometry=js.new(threejs.SphereGeometry,10,12,12)
mesh=js.new(threejs.Mesh, geometry, material)

scene=js.new(threejs.Scene) scene:add(camera) scene:add(light) scene:add(mesh)

renderer:render(scene,camera) renderer:setAnimationLoop(function() roll() end)

function roll() ang=ang+0.006 rot=mesh.rotation rot.x=ang rot.y=ang rot.z=ang renderer:render(scene, camera) end
function run() renderer:setAnimationLoop(function() roll() end) end
function stop() renderer:setAnimationLoop() end

runbutton=document:getElementById("run") runbutton:addEventListener("click", function() run() end)
stopbutton=document:getElementById("stop") stopbutton:addEventListener("click", function() stop() end)
</script>
```

This gets the picture to fill the window.

Set the camera and light position in 3D x, y, z coordinates.

Set the material and geometry so three.js can make the mesh.

Add everything to the scene.

Renderer draws the scene and we add an animation to it.

Functions to stop and start the animation loop.

Rolls the mesh object in 3D.

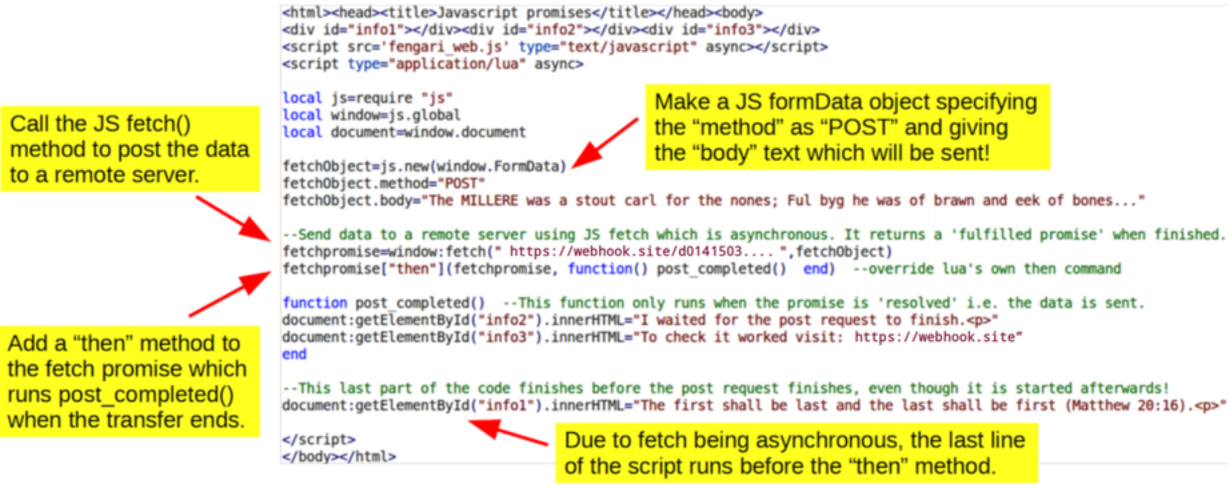
Example 7. Handling JS promises in file transfers.

In this example we will use webhook which is a website specifically for testing post and get transfers. We can easily set up a temporary URL and any data we send to it will appear in the corresponding web page. The example code below contains the temporary URL that I set up for this tutorial, so you will need to set up your own and use that instead by altering the https address in the script accordingly. Keep the webhook URL open in one browser tab and load the script in another. **N.B.** With some browsers (*e.g.* Chrome) this example only works properly if you tick the "CORS Headers" box, as shown highlighted in the screenshot of the webhook site a few paragraphs down.

The purpose of this example is to demonstrate that some JS processes are allowed to be asynchronous, *i.e.* they start and run at their own speed, independently of the rest of the code. The reason for this is that some processes, such as transferring data to or from a web server, take a finite amount of time to complete, so there is not much point holding up the rest of the script while this is happening. However, if the script needs to process data after it has been received then it must, of course, wait for the transfer to complete. JS allows for this with 'promises' *e.g.* the JS *fetch* method which we use in the below example to post data to an external server returns a promise which, assuming no errors occur, will either be 'pending' during the transfer or 'fulfilled' once the transfer is complete. JS promises have a *then* method which is executed when the promise is fulfilled and, as shown below, we can attach a lua function to this which will run only when the transfer is complete.

```
<html><head><title>Javascript promises</title></head><body>
<div id="info1"></div><div id="info2"></div><div id="info3"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js=require "js"
local window=js.global
local document=window.document

fetchObject=js.new(window.FormData)
fetchObject.method="POST"
fetchObject.body="The MILLERE was a stout carl for the nones; Ful byg he was of brawn and eek of bones- That proved wel,

--Send data to a remote server using JS fetch which is asynchronous. It returns a 'fulfilled promise' when finished.
fetchpromise=window:fetch("https://webhook.site/d0141503-e6da-4b3a-8720-af5906e4af3e",fetchObject)
fetchpromise["then"](fetchpromise, function() post_completed()  end)  --override lua's own then command

function post_completed()  --This function only runs when the promise is 'resolved' i.e. the data is sent.
document:getElementById("info2").innerHTML="I waited for the post request to finish.<p>"
document:getElementById("info3").innerHTML="To check it worked visit: https://webhook.site"
end

--This last part of the code finishes before the post request finishes, even though it is started afterwards!
document:getElementById("info1").innerHTML="The first shall be last and the last shall be first (Matthew 20:16).<p>"
</script></body></html>
```

One issue with the JS promise then() method is that in lua the word "then" has its own meaning in the familiar "if then" statement for logical tests. However, lua's syntax can be overridden locally as shown in the line beginning *fetchpromise["then"]...* to provide a "then" method, which in this case is the lua function *post_completed()*. Some more notes on the code are provided below.



```
<html><head><title>Javascript promises</title></head><body>
<div id="info1"></div><div id="info2"></div><div id="info3"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js=require "js"
local window=js.global
local document=window.document

fetchObject=js.new(window.FormData)
fetchObject.method="POST"
fetchObject.body="The MILLERE was a stout carl for the nones; Ful byg he was of brawn and eek of bones..."

--Send data to a remote server using JS fetch which is asynchronous. It returns a 'fulfilled promise' when finished.
fetchpromise=window:fetch(" https://webhook.site/d0141503.... ",fetchObject)
fetchpromise["then"](fetchpromise, function() post_completed()  end)  --override lua's own then command

function post_completed()  --This function only runs when the promise is 'resolved' i.e. the data is sent.
document:getElementById("info2").innerHTML="I waited for the post request to finish.<p>"
document:getElementById("info3").innerHTML="To check it worked visit: https://webhook.site"
end

--This last part of the code finishes before the post request finishes, even though it is started afterwards!
document:getElementById("info1").innerHTML="The first shall be last and the last shall be first (Matthew 20:16).<p>"

</script>
</body></html>
```

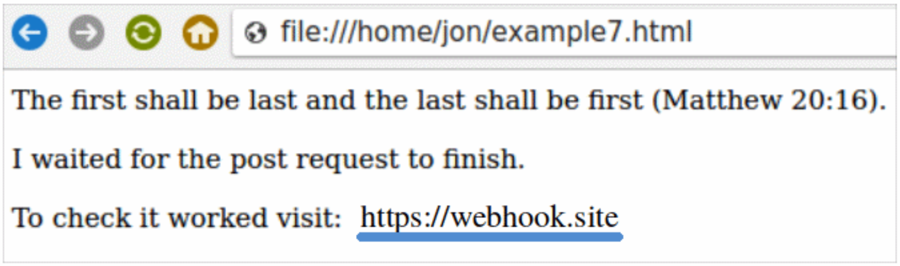
Call the JS fetch() method to post the data to a remote server.

Make a JS formData object specifying the "method" as "POST" and giving the "body" text which will be sent!

Add a "then" method to the fetch promise which runs post_completed() when the transfer ends.

Due to fetch being asynchronous, the last line of the script runs before the "then" method.

In a web browser the script loads as follows and if you copy and paste the web address which is underlined in blue below or simply switch to the webhook tab you can check that the full text has been posted to the remote server.



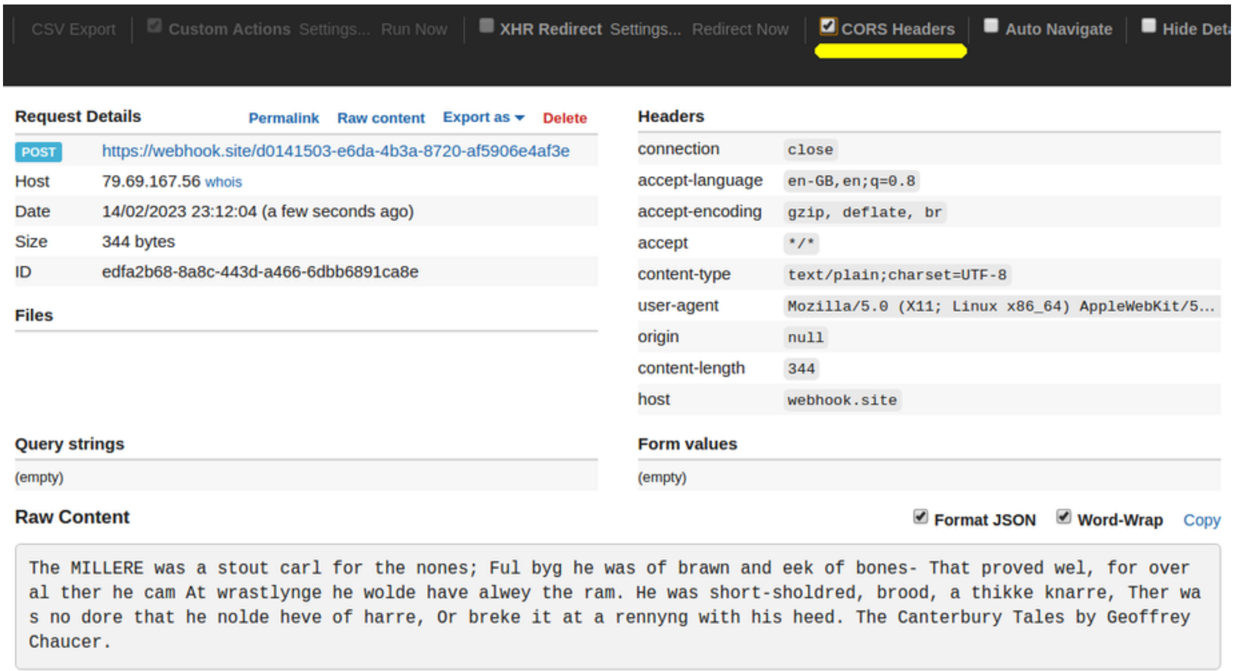
file:///home/jon/example7.html

The first shall be last and the last shall be first (Matthew 20:16).

I waited for the post request to finish.

To check it worked visit: https://webhook.site

Keep clicking the browser refresh button, green circle in the above picture, and you will see new transfers in the webhook tab as they arrive.



Request Details

POST	https://webhook.site/d0141503-e6da-4b3a-8720-af5906e4af3e
Host	79.69.167.56 whois
Date	14/02/2023 23:12:04 (a few seconds ago)
Size	344 bytes
ID	edfa2b68-8a8c-443d-a466-6dbb6891ca8e

Files

Headers

connection	close
accept-language	en-GB,en;q=0.8
accept-encoding	gzip, deflate, br
accept	*/*
content-type	text/plain;charset=UTF-8
user-agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/5...
origin	null
content-length	344
host	webhook.site

Query strings

(empty)

Form values

(empty)

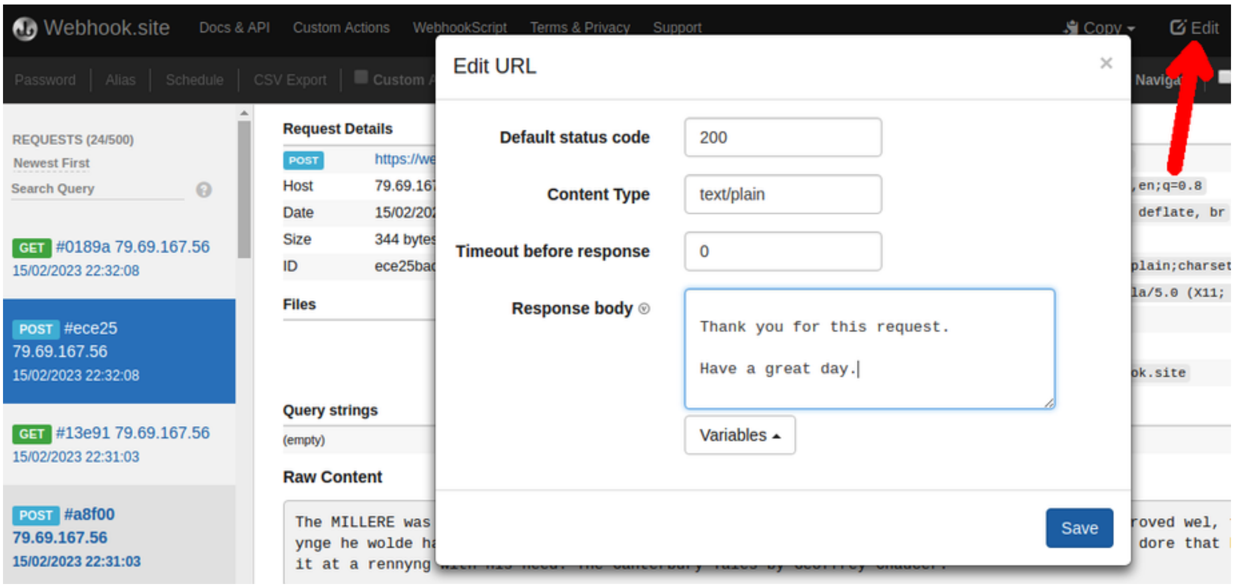
Raw Content

```
The MILLERE was a stout carl for the nones; Ful byg he was of brawn and eek of bones- That proved wel, for over
al ther he cam At wrastlynge he wolde have alwey the ram. He was short-sholdred, brood, a thikke knarre, Ther wa
s no dore that he nolde heve of harre, Or breke it at a rennyng with his heed. The Canterbury Tales by Geoffrey
Chaucer.
```

You can see the text we sent in the "Raw Content" box. Note the "CORS Headers" checkbox which needs to be clicked with Chrome.

Example 8. Chaining server requests.

In the previous example we used the fetch method to post data to a remote server. In this example we will do the same but we will also get the response from the server and display it in an inline frame, or iframe. Iframes are very useful for displaying data from a server in the current webpage, otherwise the response would automatically appear in a new browser tab. Anyway, firstly we need to visit webhook and set up temporary URL as before. However, this time we also need to set a server response which can be done by clicking the Edit button, shown by a red arrow in the figure below, and entering some text for the "Response body".



Edit URL

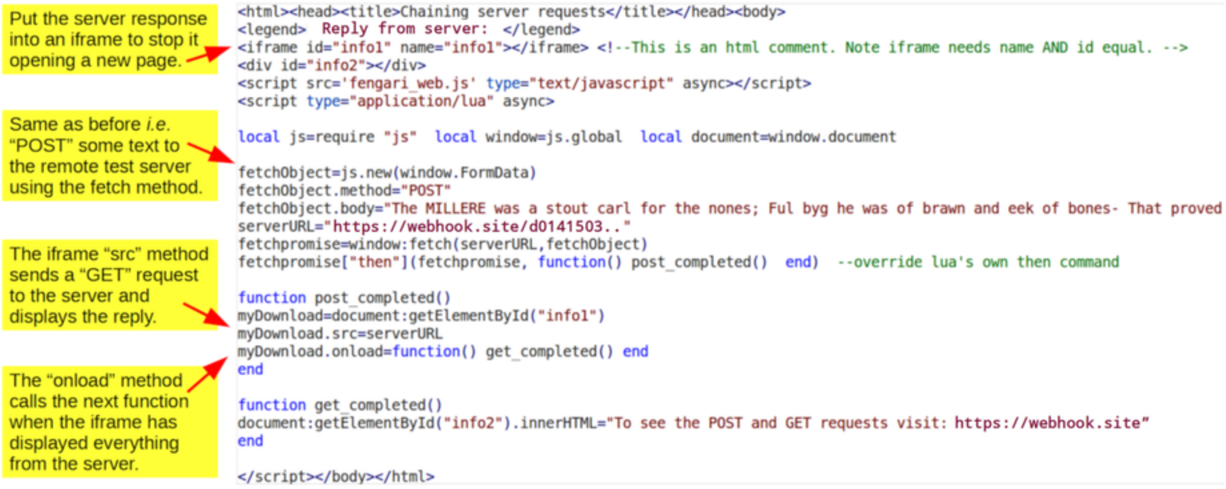
Default status code	200
Content Type	text/plain
Timeout before response	0
Response body	Thank you for this request. Have a great day.

We also need to copy and paste the temporary URL into the script, as shown below.

```
<html><head><title>Chaining server requests</title></head><body>
<legend>Reply from server:</legend>
<iframe id="info1" name="info1"></iframe> <!--This is an html comment. Note iframe needs name AND id equal. -->
<div id="info2"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js=require "js"  local window=js.global  local document=window.document
```

```
fetchObject=js.new(window.FormData)
fetchObject.method="POST"
fetchObject.body="The MILLERE was a stout carl for the nones; Ful byg he was of brawn and eek of bones- That proved wel
serverURL="https://webhook.site/6e3bbc0d-c108-4e37-bb72-e44caf972b86"
fetchpromise=window:fetch(serverURL,fetchObject)
fetchpromise["then"](fetchpromise, function() post_completed()  end)  --override lua's own then command

function post_completed()
myDownload=document:getElementById("info1")
myDownload.src=serverURL
myDownload.onload=function() get_completed() end
end

function get_completed()
document:getElementById("info2").innerHTML="To see the POST and GET requests visit: https://webhook.site"
end

</script></body></html>
```

After the data are sent to the server using the fetch promise, a function is called to get the server reply using the iframe's *src* (source) method. Then when the iframe is fully loaded, its *onload* method is used to call the final function in the script. An explanation of these steps is given below.



Put the server response into an iframe to stop it opening a new page.

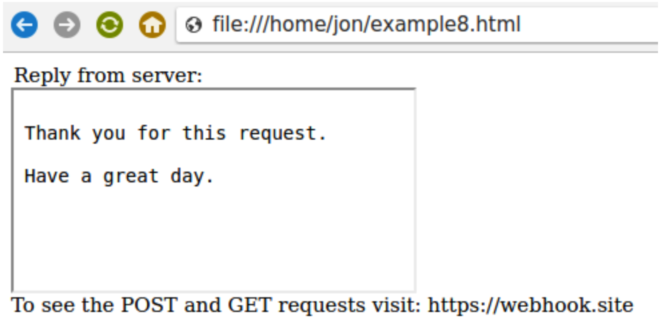
Same as before *i.e.* "POST" some text to the remote test server using the fetch method.

The iframe "src" method sends a "GET" request to the server and displays the reply.

The "onload" method calls the next function when the iframe has displayed everything from the server.

```
<html><head><title>Chaining server requests</title></head><body>
<legend>  Reply from server:  </legend>
<iframe id="info1" name="info1"></iframe> <!--This is an html comment. Note iframe needs name AND id equal. -->
<div id="info2"></div>
<script src='fengari_web.js' type="text/javascript" async></script>
<script type="application/lua" async>

local js=require "js"  local window=js.global  local document=window.document

fetchObject=js.new(window.FormData)
fetchObject.method="POST"
fetchObject.body="The MILLERE was a stout carl for the nones; Ful byg he was of brawn and eek of bones- That proved
serverURL="https://webhook.site/d0141503.."
fetchpromise=window:fetch(serverURL,fetchObject)
fetchpromise["then"](fetchpromise, function() post_completed()  end)  --override lua's own then command

function post_completed()
myDownload=document:getElementById("info1")
myDownload.src=serverURL
myDownload.onload=function() get_completed() end
end

function get_completed()
document:getElementById("info2").innerHTML="To see the POST and GET requests visit: https://webhook.site"
end

</script></body></html>
```

You should be able to confirm that everything appears in the web page in the right time sequence. The legend and the empty iframe appear first, then the server message eventually arrives in the iframe followed almost instantly by the final message at the bottom of the page.



Reply from server:

```
Thank you for this request.

Have a great day.
```

To see the POST and GET requests visit: https://webhook.site

If you look at the webhook site which you created for this exercise, you should be able to see the POST and GET requests appearing on the on the left hand side of the page, as shown in the first screenshot in this section.

In this example the server has just sent a plain text message, but the normal situation would be for the server to send some html that would be displayed in the iframe. It is also possible for the server to send back a file and this could then be downloaded to the local disk by the client browser script, *e.g.* by using a truncated version of the *save()* method of Example 2. In this case there is no need to make a blob object, etc., since *a.href* can simply be set to the relative path and name of the desired file on the remote server (relative, that is, to the server script).

In the last two example scripts, if you were to delete all of the chaining commands, these small test cases would probably still work perfectly well. However, remember that we are transferring just a few lines of text but when large amounts of data (*e.g.* megabytes) are being transferred, the use of "then" and "onload" methods is crucial to ensure everything proceeds in a synchronized way. I hope the last two examples give an idea of how asynchronous processes can be handled in a fengari web script.

Footnote.

The files used in the examples above can be obtained as a zip file here.

This is very much a guide for beginners by a beginner so it will be nowhere near perfect.

Please do let me know of any typos, errors and omissions.

Things which I still do not really understand (sorry) and should be looked at in future are:

- The JS API, push, test, proxy, etc.
- Using Webpack.
- Lots more...

Acknowledgements.

I am very grateful to Kartik Agaram for pointing out an improvement to Example 1.