# Hello World

1. Create a new F# console application.
2. Install version 1.0.0.1 of the FunScript NuGet package with command Install-Package FunScript -Version 1.0.0.1
3. Add an Index.html file to the F# project, replace the content of the file with the content from FunScript\Example 1\FunScriptExample\Index.html and set the Copy to Output Directory property of the file to “Copy if newer”.
4. In the Program.fs, create a module, decorate it with the FunScript.JS attribute, and open a few namespaces as shown below.

[<FunScript.JS>]

module Page

open System.IO

open System.Reflection

open Microsoft.FSharp.Quotations

open FunScript

open FunScript.TypeScript

The FunScript.JS attribute is really just an alias for the ReflectedDefinition attribute in F#. This attribute provides the functionality required to convert the F# code in this module into an AST, which in turn allows FunScript to generate the appropriate JavaScript.

1. You can now use the TypeScript Provider that FunScript provides to generate types from any TypeScript definition files. This allows strongly typed access to any library with a TypeScript definition file. In our example we need to generate types for the JavaScript DOM library and jQuery. Code such as the following provides this functionality (note: you may need to change the path to the TypeScript definition files):

type j = Api<"../../Typings/jquery.d.ts">

type lib = Api<"../../Typings/lib.d.ts">

1. It is now time to do something with the generated Types. We’ll first define an F# function which will be turned into a JavaScript function during the compilation process:

// Define a function that will pop an alert that says Hello World

let hello () =

lib.window.alert("Hello world!")

1. Now we define the “main” function, which will add a click event to the helloWorld anchor element that is defined in our HTML.

let main() =

j.jQuery.Invoke("#helloWorld").click(hello)

1. Finally we need to compile this F# module into JavaScript. The following code accomplishes this:

// Compile to JavaScript

let additionalComponents = FunScript.Interop.Components.all

let source = Compiler.Compiler.Compile(<@@ main() @@>,

components=additionalComponents, noReturn=true)

let sourceWrapped = sprintf "$(function () {\n%s\n});" source // Wrap in jQuery.Ready.

let filename = "tutorial.js" // Specify the desired name of the JS file

System.IO.File.Delete filename // Remove the file if it exists.

System.IO.File.WriteAllText(filename, sourceWrapped) // Write the file to disk

sourceWrapped |> printfn "%A" // Print the resulting JS

System.Console.ReadLine() |> ignore

1. That’s it. We should be able to run the project and see the JavaScript that is generated.
2. To try out your creation, navigate to the bin/debug directory and open the Index.html file in your favorite browser.

# Add a Simple Web Server

Now that you know how to use FunScript to create a simple JavaScript file, let’s add a simple web server so that you don’t have to manually launch the HTML file in a browser.

The simplest way to do this is to use System.Net.HttpListener. This combined with a few extremely cool F# features like Asynchronous Workflow and MailboxProcesser makes for a great simple web server.

Here are a few snippets that should get you drooling:

// Create a MailboxProcessor and start it. The body is an async workflow

let agent = MailboxProcessor<HttpListenerContext>.Start((fun inbox -> async {

// Constantly loop within the async workflow.

while true do

// Check to see if there is a new message to get from the mailbox

// The rest of the code will not execute until a message is received.

let! context = inbox.Receive()

// Get the local path information from the request that was sent as a message

let s = context.Request.Url.LocalPath

// Handle an ordinary file request

let file = root + (if s = "/" then "/index.html" else s)

// Check if the file exists.

if File.Exists(file) then

// If it does, get the extension

let ext = Path.GetExtension(file).ToLower()

// Figure out the content type based on the extension

let typ = contentTypes.[ext]

// Send a reply with the content of the requested file.

context.Response.Reply(typ, File.ReadAllBytes(file))

else

// If the file doesn’t exist, send a response indicating this.

context.Response.Reply(sprintf "File not found: %s" file) }

), tokenSource.Token)

let server = async {

// New up an HttpListener and use “use” so that it’s disposed appropriately.

use listener = new HttpListener()

// Tell the listener what URL it should be associated with

listener.Prefixes.Add(url)

// Start the listener

listener.Start()

while true do

// Asynchronously watch for requests for this listener.

// AsyncGetContext is defined in the Launcher.fs, but is excluded in this

// example for brevity

let! context = listener.AsyncGetContext()

// Send anything that the listener gets to the prev. setup MailboxProcessor.

agent.Post(context) }

// Start the server

do Async.Start(server, cancellationToken = tokenSource.Token)

This is all combined in a nice shared F# file that is provided in the FunScript examples named Launcher.fs.

To plug the Launcher into our previous example, do the following:

1. Add the Launcher.fs file to the project. This file can be found in the ../FunScript/Shared folder.
2. Move the file above the Program.fs file. One easy way to do this in Visual Studio 2012 is to right click on the file and click Move Up (ALT+UP and ALT+DOWN will also work, depending on your VS shortcut setup). Side Bar: You may be wondering why you have to do this. The main reason is that F# tends to lean toward explicity over implicity. Rather than implicitly making everything mutually recursive and opening up problems associated with circular references, F# makes you explicitly specify dependencies and mutual recursion. You’ll see this theme of explicity over implicity in other areas of F# as well such as interface usage and type conversions.
3. Remove all of the following code from our Hello World example:

// Compile to JavaScript

let additionalComponents = FunScript.Interop.Components.all

let source = Compiler.Compiler.Compile(<@@ main() @@>,

components=additionalComponents, noReturn=true)

let sourceWrapped = sprintf "$(function () {\n%s\n});" source // Wrap the generated JS in jQuery.Ready.

let filename = "tutorial.js" // Specify the desired name of the JS file that is generated

System.IO.File.Delete filename // Remove the file if it exists.

System.IO.File.WriteAllText(filename, sourceWrapped) // Write the file to disk

sourceWrapped |> printfn "%A" // Print the resulting JS

System.Console.ReadLine() |> ignore

1. In the place of that code, use the following single line of code:

do FunScriptExample.Runtime.Run(components=Interop.Components.all,

outputFileName="tutorial")

1. Once that is complete, run the project. It will compile the F# code to JS and launch the Index.html file.

Example 2 provides the full example.

# Building a Simple TODO App with jQuery UI and FunScript.

In the 5th chapter of the book “[Building Web, Cloud, and Mobile Solutions with F#](http://www.amazon.com/Building-Web-Cloud-Mobile-Solutions/dp/1449333761/ref=sr_1_1?ie=UTF8&qid=1369603425&sr=8-1&keywords=building+web+cloud+and+mobile+solutions+with+f)”, Daniel Mohl shows examples of how to create the front-end aspects of a simple TODO application with LiveScript, Pit, and WebSharper. In this last exercise, we will build a similar TODO application using FunScript.

1. The first step is to get all the preliminary aspects of your project setup with everything that we learned while completing examples 1 and 2. To do this, create a new F# Application project, install the FunScript NuGet package, and add the html file from [here](https://raw.github.com/ZachBray/FunScript/master/Examples/jQueryUI/index.html) and the CSS file from [here](https://raw.github.com/ZachBray/FunScript/master/Examples/jQueryUI/Site.css). The Copy to Output Directory property for each of these files should be set to Copy always or Copy if newer. We’ll also want to add the Launcher.fs file that was discussed in the Add a Simple Web Server section of this document.
2. We now need to add one more helper module that can assist in one of the callback functions that will be needed for this TODO application. So add a new .fs file named FunctionHelpers.fs and move it above the Program.fs file.
3. Now add the following F# code. Note: While you really only need the TupledDelegate type definition that contains ‘a,’b,’c, we’ve included the other overloads to show how to handle callback functions with various numbers of parameters.

[<FunScript.JS>]

module FunScript.FunctionHelpers

type TupledDelegate<'a,'b> = delegate of 'a -> 'b

type TupledDelegate<'a,'b,'c> = delegate of 'a \* 'b -> 'c

type TupledDelegate<'a,'b,'c,'d> = delegate of 'a \* 'b \* 'c -> 'd

type TupledDelegate<'a,'b,'c,'d,'e> = delegate of 'a \* 'b \* 'c \* 'd -> 'e

type TupledDelegate<'a,'b,'c,'d,'e,'f> = delegate of 'a \* 'b \* 'c \* 'd \* 'e -> 'f

let immediateFn tupledDelegate = unbox <| (fun () -> tupledDelegate)()

1. Now for the fun part. We start by adding the F# that you have no doubt become familiar with during the last two examples to the Program.fs. There is a slight variation to the modules that are opened, but other than that it is very similar to what you’ve already seen. Note: You may have to change the path to the TypeScript definition files.

[<FunScript.JS>]

module Page

open System.IO

open System.Reflection

open Microsoft.FSharp.Quotations

open FunScript

open FunScript.TypeScript

open FunScript.FunctionHelpers

type ts = FunScript.TypeScript.Api<

@"../../Typings/jquery.d.ts

../../Typings/jqueryui.d.ts

../../Typings/lib.d.ts" >

1. The first “real” code defines a function that will take care of adding TODO cards to the different container elements.

let addTasksToElement (elementSelector:string) tasks =

let tasks = tasks

|> Array.mapi (fun i task ->

"<div class='ui-widget-content draggable'>" +

task + "</div>" |> box)

ts.jQuery.Invoke(elementSelector).append tasks |> ignore

1. Next, you add a function that will be used to initialize the task card text. In a “real” application, this information would come from a data store.

let populateTasks () =

let tasksToDo =

[| "Persist the tasks to a data store."

"Add new tasks."

"Remove a task." |]

let tasksDone =

[| "Allow tasks to be moved to done."

"Add dynamic population of tasks." |]

addTasksToElement ".tasksNotStarted" tasksToDo

addTasksToElement ".tasksDone" tasksDone

1. Now it’s time to add a function that takes care of setting up the drag and drop event handlers. The drop event handler uses the TupledDelegate type and one of the immediateFn functions that you created in step #3.

let initDragAndDrop () =

let dragSettings = ts.Draggable(revert = "invalid", cursor = "move",

helper = "clone" )

ts.jQuery.Invoke(".draggable").draggable(dragSettings) |> ignore

let dropSettings = ts.Droppable(hoverClass = "ui-state-active",

accept = ".draggable")

dropSettings.drop <- immediateFn

<| TupledDelegate<ts.Event, ts.DroppableEventUIParam,

ts.JQuery>(fun e ui ->

ui.draggable.appendTo(e.target))

ts.jQuery.Invoke(".droppable").droppable(dropSettings) |> ignore

1. Lastly, the main function is defined and the Launcher is used to handle compilation.

let main() =

populateTasks()

initDragAndDrop()

do FunScriptExample.Runtime.Run(components=Interop.Components.all,

outputFileName="page")

# Other Exercises

1. Using the ASP.NET MVC skills that were gained during the MVC lab, extend the TODO application to save/restore the task cards.
2. Build out a simple shopping cart with FunScript using the example from jQuery UI as a starting point: <http://jqueryui.com/droppable/#shopping-cart>