# Installation

1. If you don’t already have Visual Studio, MonoDevelop, or Xamarin Studio installed, install one of the following:
   1. F# Tools for Visual Studio Express 2012 for Web (and walk through of Type Providers) <http://blogs.msdn.com/b/fsharpteam/archive/2012/09/12/announcing-the-release-of-f-tools-for-visual-studio-express-2012-for-web.aspx>
   2. Xamarin Studio - <http://xamarin.com/studio>
   3. MonoDevelop - <http://monodevelop.com/>
2. Install Templates:
   1. If you installed Visual Studio in step 1, get the project templates by following the instructions at <http://bloggemdano.blogspot.com/2011/12/building-aspnet-mvc-4-solution-with-f.html>
   2. If you installed Xamarin Studio or MonoDevelop in step 1, get the project template by following theinstructions at <http://bloggemdano.blogspot.com/2012/12/using-new-aspnet-mvc-4-template-in.html>

# Exercise

1. Open the GuitarExample solution from the GuitarWebApi folder and verify that it compiles/runs.
2. Expand the GuitarExampleWebApp project, and open Global.fs. You’ll see that we’ve already started you on your way by implementing a Guitar data structure, or model, and Guitars module. The Guitar model is very simple and contains a mutable Name property. Note that we could use an immutable data structure, but the default formatters in .NET work best with mutable data structures, so we will use that for now.

The Guitars module has been built in a style familiar to F# developers. We’ve removed the object oriented designs and extracted the functions that were previously embedded within the MVC controllers so that we can clearly see what functions are available for working with our Guitar model.

You should also see a comment indicating the location you should add your web API code, as well as the type declaration for Global, which is an HttpApplication. Most of the Global type is already defined, though you should see a comment in the register function indicating where you’ll need to register your APIs.

1. If you run this application, you’ll find that you don’t see any data in the UI. We haven’t exposed any APIs yet! Expand the GuitarExampleWeb project and open Index.cshtml. Here we see the markup used to display data from the web APIs, as well as the reference to Scripts/script.js.
2. Open the script.js file. This script includes some simple DOM manipulation and ajax calls to retrieve data. You should see a recurring URL structure that uses “guitars” as the entry point URL. We will implement that API first so that we can start retrieving data.

## Add a GET Method

The System.Net.Http library introduced new, statically-typed HTTP request and response message types to .NET. The ASP.NET Web API library builds on System.Net.Http and is a terrific platform for building Web APIs. However, it follows the MVC style, which doesn’t fit well with our Guitars module of functions. The GuitarExampleWebApp includes extensions from the FSharp.Net.Http and FSharp.Web.Http Nuget packages that extend System.Net.Http and Web API classes and allow F# developers to retain their normal style of coding.

Let’s start by implementing the API that will provide the collection of Guitars. Clients will use the HTTP GET method to retrieve this data, which we will provide using Guitars.getGuitars().

1. Create a function that takes an HttpRequestMessage and returns an Async<HttpResponseMessage>:

module Api =

    open System

    open System.Net

    open System.Net.HttpResource

    (\* Guitars API \*)

    let getGuitars (request: HttpRequestMessage) = async {

        return request.CreateResponse(HttpStatusCode.OK, Guitars.getGuitars())

    }

1. Now create an HTTP resource using the route and get helper functions from the HttpResource module we opened above. route creates an HttpResource that can be added to an ASP.NET route table. get is a simple function that creates a tuple containing a System.Net.Http.HttpMethod and a request handler.

let guitarsResource = route "guitars" (get getGuitars)

1. In order to hook up our API, we need only add it into the commented list passed to HttpResource.register:

    member this.Start() =

        let config = GlobalConfiguration.Configuration

        config

        |> HttpResource.register [ **Api.guitarsResource** ]

        |> ignore

1. Try running the application. You should now see data appear in the browser. If you don’t, then you’ll need to add some data. You’ll implement the POST method next.

## Add a POST Method

HTTP defines two methods with which to create new items, POST and PUT. PUT is well defined and will create a new item at a specified URI. PUT, like GET, is idempotent, so it will always do the same thing with the same data. POST is not so well behaved, which is why it has long been abused as the primary HTTP method used in applications. Nevertheless, POST is most often used to create a new item using a “parent” URI as the target, especially when the server will generate a new identifier for the POSTed data. For our purposes, either will work. POST is supported by default on most web servers, so you will use POST in this exercise.

1. Add a function to our Api module called postGuitar:

    let postGuitar (request: HttpRequestMessage) = async {

        let! guitar = request.Content.AsyncReadAs<Guitar>()

        return request.CreateResponse(HttpStatusCode.Created, guitar)

    }

This simple function retrieves the Guitar from the body of the request message and returns the Guitar instance in the response body. Also note that we are returning a status code of 201 Created, the standard status code when the server creates a new resource. You should always return appropriate status codes so that clients that understand HTTP can respond accordingly. You might be surprised how many services stick to only 200 OK, 400 Bad Request, 404 Not Found, and 500 Internal Server Error and miss out on the richness of the HTTP protocol.

1. That’s a decent start, but you have more work to do before you finish. Start by adding the retrieved guitar to the data store:

    let postGuitar (request: HttpRequestMessage) = async {

        let! guitar = request.Content.AsyncReadAs<Guitar>()

**match Guitars.addGuitar guitar with**

**| Some() ->**

**return request.CreateResponse(HttpStatusCode.Created, guitar)**

**| None ->**

**return request.CreateResponse(HttpStatusCode.InternalServerError)**

    }

Guitars.addGuitar returns an Option type. This type provides two options, Some or None, and Some may return a value. postGuitar employs pattern matching rather than an if/then conditional construct in order to determine which value was returned. You’ll see additional examples of pattern matching in the next exercise.

Observe also that postGuitar returns different responses based on the pattern match. In the case of a None, postGuitar returns a 500 Internal Server Error. You may think that a 400 Bad Request status code is more appropriate, but that status code would fit better if the request doesn’t contain all the values necessary to construct a Guitar type.

1. postGuitar currently assumes that the request will have a body containing the values to create a Guitar. Create an Active Pattern to match a correctly formatted request. You’ll use this as another layer of pattern matching that will return the 400 Bad Request response.

    let (|ContainsGuitar|MissingGuitar|) (content: HttpContent) =

        if content.Headers.ContentLength.HasValue &&

           content.Headers.ContentLength.Value > 0L then

            let guitar = content.AsyncReadAs<Guitar>()

            ContainsGuitar(guitar)

        else MissingGuitar

    let postGuitar (request: HttpRequestMessage) = async {

        match request.Content with

        | ContainsGuitar(content) ->

            let! guitar = content

            match Guitars.addGuitar guitar with

            | Some() ->

                return request.CreateResponse(HttpStatusCode.Created, guitar)

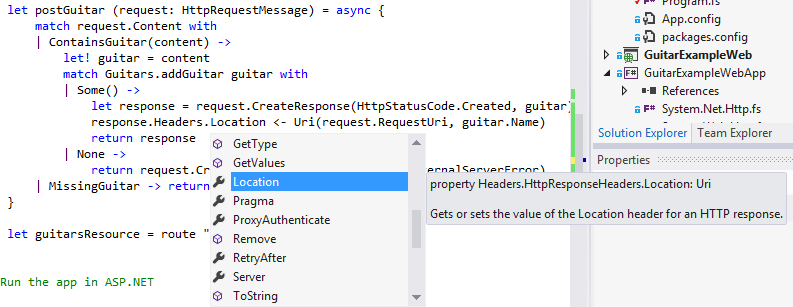
            | None ->

                return request.CreateResponse(HttpStatusCode.InternalServerError)

        | MissingGuitar -> return request.CreateResponse(HttpStatusCode.BadRequest)

Our code has grown a bit, but the logic driving the behavior is nicely factored and easy to read. F#’s Active Patterns provide a means by which to adapt non-functional types to pattern matching. If the HttpRequestMessage and HttpResponseMessage types were F# records, pattern matching would be much easier. This Active Pattern can be used for any HTTP function that needs to retrieve a Guitar from either a request or response message body.

1. HTTP specifies that POST messages should return a Location header containing the URI of the new resource. The System.Net.Http types really shine when working with headers. If you aren’t certain what you need to specify for a header value, the HttpResponseMessage type has a type for you to use. In the case of the Location header, you need to specify a System.Uri.



1. Finally, add the postGuitar function to the Guitars resource:

    let guitarsResource = route "guitars" (get getGuitars **<|> post postGuitar**)

or

    let guitarsResource = **routeResource "guitars" [get getGuitars; post postGuitar]**

You can choose either; they work the same way. The options exist to satisfy preferences for operators or lists.

1. You’ve already registered the resource, so you should be able to run the application and submit a POST to create a new Guitar. Try it now.

## Consume APIs with HttpClient

You’ve now created services, but services are only half the fun of working on the web. In addition to our JavaScript client hosted in HTML, you can use the System.Net.Http.HttpClient to issue requests against web APIs. F#’s Active Patterns provide a terrific mechanism for testing HTTP response messages for expected patterns.

1. Take a look at the Console project in the GuitarExample solution. You’ll see a very simple example of Active Patterns for matching a restricted set of response types. These all return the response headers and content. The main function simply reads the content as a string and prints it to the screen. While a valid use of Active Patterns, it really doesn’t leverage the pattern of Active Patterns.
2. In client scenarios, you’ll most likely find Partial Active Patterns of most re-usable value. Not every request/response interaction will use the same set of expected responses. You can use multiple Partial Active Patterns in the same match expression so long as the parameter type is the same for all. They don’t need to return the same types.
3. Define Partial Active Patterns for all but the Unknown pattern:

let (|OK|\_|) (response: HttpResponseMessage) =

if response.StatusCode = HttpStatusCode.OK then

    Some(response.Headers, response.Content)

else None

let (|BadRequest|\_|) (response: HttpResponseMessage) =

if response.StatusCode = HttpStatusCode.BadRequest then

    Some(response.Headers, response.Content)

else None

let (|NotFound|\_|) (response: HttpResponseMessage) =

if response.StatusCode = HttpStatusCode.NotFound then

    Some(response.Headers, response.Content)

else None

You see some duplication here, but you will more often return back different values in the Some(value) result. The only change you must make in the match expression is to replace the Unknown with an underscore ‘\_’, which is the catch-all.

match response with

    | OK(\_, content) ->

        let! result = content.AsyncReadAsString()

        Console.WriteLine("OK with " + result)

    | BadRequest(\_, content) ->

        let! result = content.AsyncReadAsString()

        Console.WriteLine("Bad Request with " + result)

    | NotFound(\_, content) ->

        let! result = content.AsyncReadAsString()

        Console.WriteLine("Not Found with " + result)

    | \_ -> Console.WriteLine("Unexpected result")

1. Suppose you want to treat JSON data differently from a standard OK response. Perhaps you want OK to return unparsed content for some requests but JSON for requests you know to return JSON data. Create a new Partial Active Pattern that ensures a 200 OK status code and a Content-Type header with a value of “application/json”:

let (|JSON|\_|) (response: HttpResponseMessage) =

if response.StatusCode = HttpStatusCode.OK &&

response.Content.Headers.ContentType.MediaType = "application/json" then

    let content = response.Content.AsyncReadAs<Newtonsoft.Json.Linq.JToken>()

    Some(response.Headers, content)

else None

1. Insert the JSON Partial Active Pattern at the top of the match expression:

match response with

| JSON(\_, content) ->

    let! json = content

    Console.WriteLine("OK with JSON: " + json.ToString())

| OK(\_, content) -> // content removed for clarity

1. Launch the web application, then start the Console project. You may need to change the RequestUri to hit your web API. Change the URI to another web site to see how your Active Patterns respond.

# Other Exercises

1. Using the skills obtained during the lab, extend the application with another Guitar resource exposing GET and DELETE APIs. The DELETE API is called by script.js.
2. Advanced: Replace the HttpResource type with an implementation that replaces the default routing mechanism with Active Patterns. Consider the following uses:
   1. Pattern match against the RequestUri property of the HttpRequestMessage, and parse and return the values of any template placeholders in the Active Pattern result.
   2. Detect Cache-Control headers and cache data locally, only submitting the request again once the max-age has expired
   3. Issue Conditional GET requests using the If-Modified-Since and If-None-Match headers and correctly handling either the 200 or 304 status codes. You should update your content with the contents of a 200 response and do nothing with a 304.
   4. Note that the last two could be performed in a single handler. Consider how you might create re-usable functions to handle these scenarios.