# Weather Forecasting APIs with Type Providers

## Lab Overview

In this lab session, you will create an API that can check two providers for weather forecasts for a given **US City** and **State**. To do this we will use a number of type providers to quickly and easily build up a set of services that we can compose together.

## Prerequisites

* Visual Studio 2015

## Time Estimate

* 30 minutes

## Table of Contents

[Interop Application Review 1](#_Toc434860015)

[Lab Overview 1](#_Toc434860016)

[Prerequisites 1](#_Toc434860017)

[Time Estimate 1](#_Toc434860018)

[Table of Contents 1](#_Toc434860019)

[Exercise 1: Overall Solution architecture 2](#_Toc434860020)

[Exercise 2: Exploring the F# code 2](#_Toc434860021)

[Exercise 3: Exploring interation points 2](#_Toc434860022)

## Exercise 1: Construct type providers

In this exercise, we will explore the type providers that we will use.

1. Open the Data Access solution.
2. Navigate to the Scripts/WeatherForecasting.fsx script in the DataAccess project.
3. Populate the three outstanding type providers using the URIs supplied. These providers will act as services to retrieve data from: -
   1. A SOAP/WSDL web service to retrieve weather forecasts given a ZIP code.
   2. A web page containing a table to convert city / state to ZIP codes.
   3. A restful URI that returns weather forecasts as JSON.
   4. A restful URI that returns weather forecasts as XML.

## Exercise 2: Exploring the generated types

Explore the generated types using the GetSample() calls. Observe that, for example, the WSDL type provider returns a proxy for all service methods in the WSDL service. The HTML, JSON and XML providers simply parse and provide types for data returned for specific URI. Also notice how the types are generated. You may, for example, see items exposed as collections in F# - this is inferred from the JSON. Similarly, if a property is missing from one element in a collection but present in another, it will show up as an option type.

Identify what data you will need from each service in order to get back a **temperature (as an integer)** and a **description (as a string)**.

In F#, it’s legal to have symbols with spaces in them, as long as they start and end with a double backtick (``) e.g. person.``Name And Address``. You’ll often see this with provided types, but there are other applications for them, too.

## Exercise 3: Create parameterized functions for service calls

In this exercise, we will create some simple wrapper functions that will encapsulate the use of the type providers and return some simple data structures.

1. Implement the getZipCode, getYahooJsonForecast and getYahooXmlForecast functions. You will need to call the Load function on the appropriate type providers in order to load “live” data rather than just the sample data. You will also need to look at the URI structures and replace the appropriate data in the string with the argument supplied e.g. the city and state.
2. The GetZipCode function should return a string option. This is because if you do not supply a valid city / state, it will return invalid HTML. On returning the data from the type provider, before interrogating the Tables.Free Zip Code Finder property, you can check if the Tables.HTML property contains the text “No records match your search parameters.”. If it does, the call has failed and you should return None. Otherwise, simply return the first ZIP code identified in the .Rows property.
3. Test out all three functions to ensure that they work as expected. Note that you will need a Yahoo API key in order to generate correct data from the Yahoo weather API.

## Exercise 4: Compose services together

In this exercise, we will see how we can easily build smaller functions and services together using native F# syntax and libraries. We wish to retrieve weather data from the WeatherWsdl service. However, this only takes in a zip code. So first, we need to call the getZipCode function, which takes in a city and state, and returns an optional zip code. Then, we can pass this into the getCdyneForecast function, which takes in a zip code and returns a weather forecast.

getZipCode : (string, string) -> string option

getCdyneForecast : string -> WeatherReturn

We can plug these together into a new function which takes in the city / state and returns the WeatherReturn forecast by using the >> and Option.map functions together. We can also check if the result from getCdyneForecast was a Success or not using Option.filter. In this way, we only return some forecast if: -

1. The city / state is combination valid.
2. The call to the web service returns a valid (Success) result.

A longhand version is supplied to show you a more explicit way of performing the same logic.

## Exercise 5: Build a top-level API

We will now create a high-level abstraction on top of these lower level functions. It will take in the name of the forecasting service it wishes to query, as well as the city and state. It will return the current temperature and description as a Tuple (if available). The signature is as follows: -

**getWeather** (provider:WeatherProvider) (city:string, state:string) : (int \* string) option

The WeatherProvider type is already supplied in the code. You should pattern match over it, covering all three cases. All paths should return the (int \* string) option i.e. the temperature and description.