Visualizing Data on the Web

Loïc Denuzière
IntelliFactory
loic.denuziere@intellifactory.com

Adam Granicz
IntelliFactory
granicz.adam@intellifactory.com

Anton Tayanovskyy
IntelliFactory
anton.tayanovskyy@intellifactory.com

Abstract

We present a language-integrated technique that can be applied to enlist web-based data visualization libraries in the type-safe discipline of F#, and to use them with various data access mechanisms to visualize data on the web quickly and effectively using WebSharper[1], an open source web framework for F#[2]. This work, especially when applied in combination with F# Type Providers[3], provides excellent capabilities for rapid data exploration, and enables the quick prototyping and development of web-based data visualization applications.

Categories and Subject Descriptors D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming – web and data-centric programming.

D.3.3 [**Programming Languages**]: Language Constructs and Features – F# Type Providers.

H.3.5 [**Information Systems**]: Online Information Services – Web-based services.

General Terms Experimentation, Languages

Keywords data visualization; web programming; F#; type providers; WebSharper; web APIs

1. Introduction

There are a multitude of data visualization libraries and reusable components available for web application development, and advances made in creating stunning data visualizations are constant, driving a strong desire to find ways to quickly embed them in consuming applications. One main complexity in using these web-based libraries in web applications is addressing them with server-side code, and handling the language mismatch between the two tiers.

In this paper, we use WebSharper, an open source web framework for F#, to bridge over the language mismatch; and present a novel approach to describe existing web (JavaScript) APIs in strongly-typed F# code, to be consumed directly in WebSharper applications. This approach can be used to interface with the multitude of client-side data visualization libraries available, enabling developers to visualize large amounts of data in various ways. By providing such a mechanism to interface JavaScript APIs in F# to create strongly-typed bindings directly addressable in an F# web framework, we aim to enable developers and non-developers alike to easily use and benefit from the massive diversity of client-side web functionality including advanced data visualization.

Introducing type safety and functional syntax in JavaScript has been an active topic of interest in recent years, and the development of JavaScript embedded languages such as CoffeeScript[4] and TypeScript[5] are notable examples. In this paper, we are only concerned with bridging to JavaScript libraries through F# code, without requiring modifications to those code bases; although the ideas presented here could easily be adapted to other, or JavaScript-embedded scripting languages such as the ones just mentioned.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DDFP '13, January 22, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1871-6/13/01...$15.00

2. WEB APIs

Web-based APIs and libraries are typically written using JavaScript. This uniformity, reinforced by numerous standards, helps making the Web the powerful tool it is, by providing a common interface for the different aspects of a web application. The typical components of a JavaScript library are simple and well-understood: one or more objects serving as a namespace that contain weakly-typed properties and methods that provide the library's functionality.

However, JavaScript comes with a number of inconveniences of its own. First, the weak typing makes the discovery of a library's protocol difficult. JavaScript has no concept of a class, interface, or any kind of user-defined type, which would provide a complete description of its capabilities. Furthermore, functions hold no information on the types (and sometimes even the number) of arguments they operate on. Instead, developers have to rely on the documentation provided by the library's authors, which comes in wide variations in quality and completeness.

Second, dynamic typing means that many errors are not detected until code is actually run. Since functions do not hold any information about the types of their parameters, one can call a function with erroneous or improper data, without being identified by the browser at loading time. It is not until the function call is executed that the error is caught. This causes serious problems in application robustness, and developers have to rely on unit, functional and other testing methods, with the hope that they are done and applied exhaustively.

2.1 Describing Web APIs in F#

F# provides a static and strong type system that alleviates many of these problems. Built on top of F#, WebSharper is a toolset aimed at building web and mobile applications entirely in F#. One of its main features is an F#-to-JavaScript compiler that enables developers to write the client-side part of their web and mobile applications side-by-side with the server-side part in F#. They can, therefore, get JavaScript's weaknesses out of the way and concentrate on their own code instead.

However, web applications also rely heavily on third-party JavaScript libraries, from addressing simple and ubiquitous chores such as browser-independent DOM manipulation using common-place libraries such as jQuery, to providing more involved functionality such as responsive design, advanced reactive communication, etc. In particular, for our domain of interest, data visualization, there are many libraries available that enable drawing complex graphs, charts, maps and other visualizations in just a few lines of code. In order to take advantage of these libraries while still benefiting from F#'s static typing and other features, WebSharper enables library authors to describe in a strongly-typed manner the interfaces of their JavaScript libraries. This is accomplished using the WebSharper Interface Generator, or WIG for short, a standard and integral part of the WebSharper framework. We call the JavaScript bindings created with WIG as WebSharper extensions, and WebSharper at the time of writing this article comes with over two dozen extensions to popular JavaScript libraries, including advanced data visualization ones, making it easy to interface with them in WebSharper applications.
For example, here is an excerpt from the WIG definition of Rickshaw[6], a JavaScript visualization library:

```
let Graph =
  Class "Rickshaw.Graph"
  |+> [
    // Static members
    Constructor GraphConfiguration
  ]
  |+> Protocol [
    // Methods
    "render" => T<unit> ^-> T<unit>
    "configure" => GraphConfiguration ^-> T<unit>
    "onUpdate" => (T<unit> ^-> T<unit>)
                  ^-> T<unit>

    // Properties
    "renderer" =? Renderer
  ]
```

The above code defines/stubs a JavaScript class named `Rickshaw.Graph`. It defines a constructor for this class, and a number of methods and properties. The various constructs used in these definitions (the combinators `Class`, `Constructor`, `Protocol`, `T<_>` and the operators `|+>`, `=>`, `=?` and `^->`) provide type-safety to the definition language.

Once compiled, this WIG definition makes the defined protocol accessible to WebSharper applications through the same type-checked and type-inferred interface as any F# library. Developers can use auto-completion to get information about the JavaScript library and will be warned by their IDE when using it incorrectly.

2.2 Type Provider for JavaScript APIs

Type Providers are a feature of F# 3.0 which effectively enables developers to generate namespaces and types at compile-time. It is also integrated with Visual Studio's IntelliSense, so that auto-completion and on-the-fly API documentation are also made available by these generated classes and updated when necessary.

Exposing the WIG toolset as a simple Type Provider enables the seamless integration of external JavaScript libraries into consuming WebSharper applications without having to install or manually include a corresponding WebSharper extension project for each, and in essence it helps to remove the build and packaging-related chores and let developers focus on their code instead. Essentially, its core function is to trigger the build system with the right customizations for building WIG-based extensions, and it does not affect the code developers would write otherwise using those extensions.

For instance, assuming that a full Rickshaw WIG definition (contained in `rickshaw.wig`) is placed inside an F# WebSharper web application project, the following code generates a type space representing all Rickshaw code artefacts, and makes them available for the rest of the application:

```
type Rickshaw = JavaScriptWIG<"rickshaw.wig">
```

This way, WebSharper extension definitions can be accessed from globally available URLs (not shown here), or downloaded and changed/adapted locally and consumed from the local version without the need to perform the necessary build and packaging protocol to generate a proper WebSharper extension. This makes the maintenance of these extensions more predictable by encapsulating JavaScript APIs in single, versioned at will WIG definitions without further plumbing, and enables developers to fork their copies and adapt them to their needs, or to fix bugs in publically available versions. WIG definitions are valid F# code and can be type-checked as any other F# code, making updates safe.

The JavaScript WIG Type Provider can also detect changes in the underlying `.wig` files, regenerating the corresponding type space immediately, reflecting any code changes in the integrated development environment that need to be made as a result.

3. Example

We now bring this work together to create a web application that can retrieve data from the web and display it in a client-side interactive visualization widget.

First, we acquire the data using a Type Provider. In this example, we use FSharpx[7]'s excellent Freebase[8] provider. Freebase is a web-hosted community knowledge database covering a wide range of topics. We will use the Science and Technology section of Freebase to visualize the radii of different atoms with respect to their atomic number.

```
let GetAtomRadii() : (string*int*float<_>)[] =
  let db =
    FSharpx.TypeProviders.Freebase.FreebaseData.
      GetDataContext()

  let data =
    query {
      for e in db.``Science and Technology``.
               Chemistry.``Chemical Elements`` do
      where (e.``Atomic number``.HasValue)
      where (e.``Van der Waals radius``.HasValue)
      sortBy (e.``Atomic number``.Value)
      select (e.``Atomic number``.Value,
              e.``Van der Waals radius``.Value,
              e.Name)
    }
  Array.ofSeq data
```

The function `GetAtomRadii` creates a Freebase data context, queries it and returns the queried data as an array of (atomic number, radius, name) triples.

This query is also an example of F# Queries, a feature that uses various metaprogramming techniques to enable developers to author "language-integrated" queries using F# syntax, and translates them to code that calls into the underlying .NET LINQ libraries. Such queries can be executed against any data source that supports a well-defined set of interfaces, typically enabling access through a data context object as shown above.

The Freebase Type Provider used in this example is responsible for generating all the code necessary to access and query the Freebase database, and to represent the entities in it in a strongly-typed manner. The F# Query used to query this data source attempts to talk to this service using the protocol implemented by

the Type Provider. This, depending on the query at hand, may involve multiple service requests, and ultimately the matching dataset is returned and represented as a sequence.

With this data acquired for our running example, it can be passed to the web client for visualization. One of the chores WebSharper automates is the communication protocol between the client and the server code, making this a simple function call from the code that represents the client-side code to the server-side one.

To visualize the data, we will use the library Rickshaw mentioned in Sections 2.1 and 2.2. The code calling Rickshaw uses the WIG-generated bindings previously presented, pulling in the advantage of a type-safe interface and simplifying the writing of the code.

```
[<JavaScript>]
let ShowData data body =
   let points =
      data
      |> Array.map (fun (name, number, radius) ->
         Point(float number, float radius))
   let series =
      Rickshaw.Serie(points, Color = "blue")
   let graph =
      Graph(
         GraphConfiguration(body, series,
            Width = 800,
            Height = 600,
            Renderer = RendererType.Line))
   graph.Render()
```

The `[<JavaScript>]` annotation informs WebSharper to translate this function to JavaScript for use on the client. This visualization can be inserted into a WebSharper application through a WebSharper `Web.Control` type that renders a `<div>` element and calls the `ShowData` function to populate it.

```
[<Sealed>]
type Control(data) =
    inherit Web.Control()

    [<JavaScript>]
    override this.Body =
        Div []
        |>! OnAfterRender (fun div ->
            ShowData data div.Body)
        :> _
```

This type can be embedded into the WebSharper HTML combinators used to build sitelet pages, or directly into legacy ASPX markup. Readers are encouraged to study the standard WebSharper sitelet Visual Studio templates for a full description of sitelets and the above mechanism.

Note that WebSharper automatically serializes data passed to the above `Control` constructor, which means that the integration between the server-side query and the client-side visualization is effectively seamless.

The resulting visualization is shown in Figure 1.

4. Future Work

While interesting and powerful on their own, we believe that the presented techniques can open up more possibilities when made available and integrated within a live web, and cloud-hosted development environment. To further explore such thrilling ideas, our current work is geared towards and focused on the creation of an online integrated development environment (IDE), which can greatly increase the usefulness and practical value of client-side data visualization presented here. In such an environment, a developer can explore the capabilities of both the data provider and the visualization tools, with instant feedback. This opens great perspectives for an accessible, installation-less, multi-purpose, collaborative tool, suitable both for developers and users in data-intensive domains such as finance, medicine, and scientific research.

Extending WebSharper coverage to JavaScript embedded languages such as CoffeeScript and TypeScript also opens up a number of interesting venues. TypeScript, in particular, can be another source of API metadata that could drive the generation of WebSharper extensions, given the more abundant type information available.

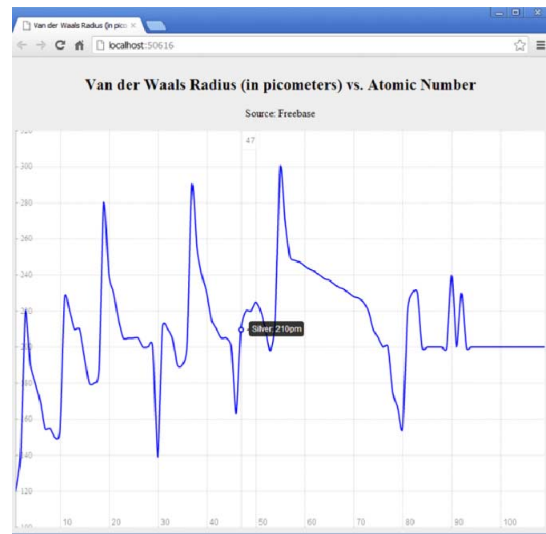

Figure 1. A web page showing the Van der Waals radii of atoms in a simple line graph

5. Conclusions

In this paper, we presented a language-embedded technique to integrate web-based data visualization libraries from strongly-typed interface definitions, and to make them available to an F# web framework in which entire web applications can be authored in pure F# code. Coupled with data-centric F# Type Providers, our JavaScript WIG Type Provider can automate the generation of WebSharper extensions, and enable developers to build data-centric web and mobile web applications that utilize advanced client-side visualization.

The growing number of appealing and feature rich web libraries performing stunning, interactive and rich data visualization is a definite indicator of the shift of attention towards data, information, and its presentation to users on the web, among others. We hope that our work in more readily connecting these libraries into real-life functional web applications further facilitates that shift.

Acknowledgments

We would like to thank the F# Team for their continuous efforts to improve the F# language and the F# compiler, and for introducing such wonderful language features as Type Providers, without which this work would have been much less appealing and readily usable. We believe that the combination of rich web and data-oriented programming has a real possibility to demonstrate that the strengths of applied functional programming are immense in these domains, and bring more recognition to the languages, abstractions, and techniques involved from a wider range of developers and end-users alike.

References

[1] Tayanovskyy, A., Granicz, A. et al. The WebSharper PDF Book. http://websharper.com/websharper.pdf

[2] Syme, D., Granicz, A. and Cisternino, A. 2010. Expert F# 2.0. Apress. ISBN-1430224312.

[3] Syme, D., Battocchi, K., Takeda, K., Malayeri, D., Fisher, J., Hu, J., Liu, T., McNamara, B., Quirk, D., Taveggia, M., Chae, W., Matsveyeu, U. and Petricek, T. 2012. Strongly-Typed Language Support for Internet-Scale Information Sources. Technical Report. Microsoft Research.

[4] The CoffeeScript website. http://coffeescript.org

[5] The TypeScript website. http://www.typescriptlang.org

[6] Shutterstock. Rickshaw: A JavaScript toolkit for creating interactive time-series graphs. http://code.shutterstock.com/rickshaw/

[7] Mohl, D., Riley, R., Scheffer, M. et al. FSharpx. https://github.com/fsharp/fsharpx

[8] Bollacker, K., Evans, C., Paritosh, P., Sturge, T. and Taylor, T. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08). ACM, New York, NY, USA, 1247-1250. DOI=http://doi.acm.org/10.1145/1376616.1376746