WebLang

Ryan Bernstein (rb3234), Brendan Burke (btb2121), Christophe Rimann (cjr2185),
Julian Serra (jjs2269), Jordan Vega (jmv2177)

09/25/17

1 Introduction

Programming is becoming increasingly tied to the web. The vast majority of commercial applications, from social media to cancer detection mobile applications, involve some sort of server backend communicating with clients via the HTTP protocol. These servers are also increasingly reliant on others, as more and more services communicate with other services in order to richly serve a user. The de facto standard for much of this communication is Representational state transfer, or REST. These RESTful API's are becoming in many cases more important than the websites they help serve [1]. Many successful companies have made entire businesses out of doing nothing but connecting different APIs (ifttt [2], zapier [3], and Microsoft Flow [4] are three examples). These services begin to approximate what we'd like to accomplish with our language (seamless interaction between different web services), but they are nonetheless quite rigid in their implementations.

2 Purpose

Our project aims to provide an easy mechanism to allow programmers to take part in building and consuming RESTful applications. We'd essentially like to build a language that serves as a replacement for services like ifttt by offering significantly more power and flexibility while hopefully keeping the structure of the language intuitive and simple. There will be two primary components of the language: an API ingester (i.e. a very simple web server that can easily be configured to listen for certain calls, aka webhooks), and an API client (to send API calls and receive the responses). In this way, users of the language will very easily be able to create applications that interact between different APIs, allowing these applications to both listen and initiate. Potential use cases include: easy creation of bots (for example chat bots or trading bots), simple hosts for websites, or even entire web applications.

3 Components

As mentioned in section 2, there are two primary components to WebLang.

3.1 Server

For the purposes of this discussion, we will assume that the server's root lives at localhost:8000. The server is defined as a collection of functions. Each function represents an endpoint; the name of the function is in fact the endpoint: for example, a function such as search() would be available at localhost:8000/search. As such, in weblang, functions will be referred to as endpoints. The primary endpoint modifier is the "method"; we'd hope to support as many of the HTTP verbs as possible, but at the very least GET and POST. A potential return type could serve to automatically set HTTP headers, although this would be a stretch goal (for example, if the return type is json, returning a string would automatically encode it in json and set content encoding etc). The parameters an endpoint takes in are: the header, and the body. How the body will be ingested is yet to be determined; they could either be named parameters (a user would have to specify what parameters they are looking for, like a defined API), or kwargs like python (although this would mean typing would need to be dynamic and therefore open a whole new can of worms).The endpoint body simply performs some action, and then returns data to the user. All returns are thus HTTP Responses; if the user is not expecting a response, no return statement is necessary.

3.2 Client

The client component is really nothing more than a hyper specific standard library that allows the user to interact with other services. Those with experience in web programming will find our goals very familiar: whether it be requests, urllib, HttpComponents, or even socket manipulation directly, we'd like to abstract away all the nasty bits of HTTP requests to allow programmers to easily interact with other applications. Our hope would be to link a library like CPR (C++ requests). To explain the functionality, we go to object oriented terminology: users would create an "object" representing a root url, then access specific attributes representing HTTP verbs available, then call the endpoint method (as mentioned above). Parameters represent the body of the request. For example, if we have an object goog for the url http://google.com, a search for weblang would look like: goog.GET.search(q="weblang"). Furthermore, on creation of the UrlRoot object, default values for all requests sent to that object can be added (for example, authentication).

4 Types

Our ideal scenario for types would be only strings - we'd love to build this as exclusively a string manipulation language, given that that is how information is transferred via REST. Unfortunately, we understand that that's not very useful. Ideally, we'd still like strings to be our only primitive, with specific "endpoints" within different libraries handling all other data types (we will mention this in section 5.1). In this way, although there is only one primitive, the language is extensible enough to cover a variety of scenarios.
Further types would include dictionaries (likely an implementation of hashmaps, like in python), and lists (once again, influenced by python), as these are crucial for collecting data.

5 Specific mechanics

We mention the following specific mechanics because we think they are really exciting.

5.1 Importing/libraries

There are no libraries, and thus no imports. All libraries built for weblang would take the form of server code (written in any language) that communicates via the HTTP protocol. In this way, we can easily write a flask server in python to perform various tasks that acts as a "library" for weblang. These libraries can then be run locally (or in the cloud), allowing them to interact with the user's program. As such, there is syntactically and mechanically no distinction between a library and a urlroot: they are identical. Furthermore, because of this interoperability, weblang theoretically supports all types that exist in languages capable of producing a web server: although weblang itself can only handle strings, it can pass off different computations to its "libraries" as strings, and receive their responses as strings as well. If the user would like to add two floats, they can easily use the Math library in Java to do so, assuming that someone has created a Spark application to service weblang.

5.2 Local function calls

There will exist a special urlroot within weblang, local, that represents the namespace that the program is currently operating in. As such, once again, there is syntactically no differentiation between calling an endpoint that exists locally and an endpoint that is across the world. Mechanically, there will be a distinction in order to speed up the system.

6 Code example

An example program is seen below. It waits for a service to send a request to it (as a webhook) with new email data (an example of this would be the mailgun API), and then creates a new post with the subject of the email onto the myemails subreddit.

```
Server WebServer{
    POST process_email(headers, email_body, email_subject, email_sender){
        auth = {'api_key':"key", 'api_secret':"secret"};
        reddit = new UrlRoot(auth = auth, reddit.com);
        data = {'subreddit':'myemails','post':email_subject}
        response = reddit.POST.new_post(data);
        return response.code;
    }

}
```

Figure 1: WebLang Sample program

7 Conclusion

We're very excited about weblang. We see it as a necessary language that is also very approachable: with only one type and one type of object (urlroot), there isn't a whole lot necessary to hit the ground running. However, its interoperability makes it incredibly flexible (albeit quite slow).

References

[1] Debow, B. Why you need to think about APIs before Websites (https://www.wired.com/insights/2014/11/apis-before-websites/)

[2] IFTTT helps you do more with the services you love. Connect Amazon Alexa, Facebook, Twitter, Instagram, Fitbit, Slack, Skype, and hundreds more. ifttt.com

[3] Zapier makes it easy to automate tasks between web apps. Zapier.com

[4] Automate tasks by integrating your favorite apps with Microsoft Flow. Make repetitive tasks easy with workflow automation. flow.microsoft.com.