RAPID (RAPID API Dialect)

COMSW4115 Proposal

Our Team

- **Manager** Ben Edelstein, bie2103
- **Language Guru** Dan Schlosser, drs2161
- **System Architect** Brendon Fish, bjf2127
- **System Integrator** Nate Brennand, nsb2142
- **Tests / Validation** Brian Shin, ds2791

Introduction

With increased demand in the public and private sector for cloud-connected mobile and web applications has come a rising need for web servers to maintain state across multiple devices and users. Development of web servers is complex, however. Building a web server using modern web server packages requires learning a server-side programming language, and then integrating a web server package and implementing required methods. Furthermore, local testing and development of these servers is excessively complex, as they have numerous dependencies and are difficult to build.

RAPID (RAPID API Dialect) is a programming language intended specifically for the rapid development of web APIs that are compatible with modern standards for data transmission (like REST). Using *RAPID*, developers can easily code and launch a database-backed REST API server that guarantees JSON or XML shapes in responses. *RAPID* compiles to Go[1], for extreme portability and built-in multi-threading.

Language Features

- Statically typed.
- The keywords `class` , `param` , and `namespace` define a "path context". All methods defined with a `class <ClassName> {...}` block are associate with instances of that

class, and yield routes that begin with that class's classname. Similarly, any `http` routes within a `namespace <namespace_name> {...}` block have the namespace name appended to the "path context". Finally, functions defined within a `param <param_name> {...}` block are required to declare a parameter `param_name` , which is also appended to the "path context".Nested blocks are appended from left to right, outside in. See the comments (`//`) in the following code snippet to see the "path context" for each block:

```
// Path context: /
class User {
    // Path context: /user/
    param user_id {
        // Path context: /user/<user_id>/
        namespace books {
            // Path context: /user/<user_id>/books/

        }
    }
}
```

- HTTP and routing primitives. The `http` and `func` are used to define functions. If `http` is used, the method name is appended to the "path context", yielding the complete route. The `func` keyword, which may not be nested within `param` or `namespace` blocks, defines instance methods that may access the class instance using the `self` keyword. If a `http` is left unnamed (see below), it is implicitly named `''` .

```
// Path context: /
class User {
    namespace list {
        http (int max) User[] {
            users = get_a_list_of_users(max)
            return users, 200
        }
    }
}
```

This creates a route `GET /user/list/` on the API, which will require a query string parameter `max` . All routes generated are lowercase.

`http` methods must return a tuple, where the second entry is a HTTP status code.

- Allowed HTTP methods for all `http` routes are implicitly only `GET`, but these may be overridden:

  ```
  http[POST] create() str {
      // this route only accepts POST requests
      return '', 201
  }
  ```

- Implicit JSON and XML handling. Routes accept and return Objects, which are serialized to and decoded from JSON under the hood. Parameters are required to exist in the query string if they are declared as arguments to `http` methods. If the argument name has `JSON`, or `XML` prepended, the request body will be parsed, looking for an object encoded in the specified type. Arguments within the parentheses of a `http` function declaration must follow the order: path, query string, request body. For example:

  ```
  class User {
      str name
      str password
      int id

      param user_id {
          http update_profile_info(int user_id,
                                   bool overwrite=True,
                                   JSON User profile_info) User {
              // update user instance with id `user_id` with profile_info,
              // overwriting if `overwriting` is set in the query string
          }
      }
  }
  ```

  Here `user_id` is a path param, `overwrite` is an optional boolean parameter (with `True` as it's default value) in the query string, and `profile_info` is matched against the JSON-decoded response body to take the form:

  ```
  {
      "name": "AzureDiamond",
      "password": "hunter2",
      "id": 42
  }
  ```

- SQL database-backed (Postgresql). Classes are mapped to SQL tables, and the data is accessed using the build in standard library methods `db_get` , `db_delete` , `db_insert` , and `db_update` . SQL joins are not supported.

Example Programs

1. `hello_world.rapid`

```
http hello() str {
    return "Hello World", 200
}
```

```
$ curl http://localhost:5000/hello
Hello World
```

2. `twitter.rapid`

```
class Tweet {
    str message
    str username

    http list(max=20) Tweet[] {
        tweets = self.db_get({limit: 20})
        return tweets, 200
    }

    http[POST] (JSON Tweet tweet) str {
        self.db_insert(tweet)
        return '', 201
    }
}
```

```
$ curl -X POST -d '{"message": "just setting up my twttr", "username": "jack"}'
$ curl http://localhost:5000/tweet/list?max=2
[
    {
        "message": "just setting up my twttr",
            "username": "jack"
        },
        {
            "message": "you can go hunter2 my hunter2-ing hunter2",
            "username": "AzureDiamond"
        }
    ]
```