REST Inspired Code Partitioning with a JavaScript Middleware

Janne Kuuskeri and Tommi Mikkonen

Department of Software Systems, Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
{janne.kuuskeri,tommi.mikkonen}@tut.fi

Abstract. The Web has become a commonly used programming environment for a number of types of applications, and to a great extent, applications that would have formerly been implemented as desktop systems are now run inside the browser. Numerous approaches have been proposed for application development on the Web, but it is not obvious how the different approaches fit together. In this paper, we introduce a middleware platform intended for composing JavaScript applications. When using the middleware applications are implemented solely using JavaScript in a fashion, where the client side of applications runs inside the browser, whereas the server side gains advantage from a newly emerged activity, CommonJS.

1 Introduction

The software industry is currently experiencing a paradigm shift towards web-based software. The Web has rapidly become a de-facto programming environment for a number of types of applications, and to a great extent, applications that would have formerly been implemented as desktop systems are now run inside the browser.

Unfortunately, the browser was intended to be a hypertext distribution environment, not a runtime system for serious software systems. Consequently, there are numerous restrictions on how the Web should be used as an application platform. One of the fundamental limitations is that if one wishes to build applications relying on client-side functionality, the only programming language that can be used is JavaScript, since it is included in all the major browsers.

At the same time, the Web also lends itself to numerous approaches that can be adapted when developing applications. Perhaps the most prominent and widely accepted approaches are Web Services [2] and Representational State Transfer (REST) [3].

In this paper, we introduce a middleware intended for composing JavaScript applications using a paradigm inspired by REST for client-server communication. Applications are implemented solely using JavaScript, The client side of applications runs inside the browser, whereas the server side gains advantage from a newly emerged activity, CommonJS[1].

[1] http://commonjs.org/

F. Daniel and F.M. Facca (Eds.): ICWE 2010 Workshops, LNCS 6385, pp. 244–255, 2010.

2 Background

Our middleware is built by combining Representational State Transfer (REST) and JavaScript programming language in a new and innovative fashion. Instead of using JavaScript only inside the browser, we have extended its role to server-side programming as well. Moreover, REST based interfaces will be used for enabling the interaction of JavaScript code running in the client and the server.

2.1 REST

Representational State Transfer (REST) as a term, originally introduced by Fielding [1], refers to an architecture suited for distributed hypermedia systems, such as the Web. The architecture consists of clients, which initate requests, and servers, which process requests and reply with suitable responses. A RESTful architecture is commonly thought of as being resource-oriented, where a resource is basically anything that can be considered as a meaningful concept and is addressable by a URI. Requests and responses are then built around the transfer of representations of resources, which in the case of Web are most commonly documents that capture the state of the resource. A RESTful Web API is a web service that is implemented using HTTP. It consists of three elements:

- *Unique Resource Identifier (URI).* Since resources form a key concept in REST, each of them must be addressable.
- *Operations.* To manipulate resources, clients use a set of operations supported by the service. When using HTTP, these operations are associated with HTTP methods, such as GET, PUT, POST, and DELETE.
- *Data representation.* The representation of the data is needed in order to interpret it correctly. This is commonly implemented using Multipurpose Internet Mail Extension (MIME) types.

The simplicity of REST – together with a number of other qualities, such as scalability and generality of interfaces – has made it a commonly advocated approach for developing resource oriented web sites.

2.2 CommonJS

CommonJS is an effort to standardize the runtime environment for JavaScript running outside the browser. Conventionally JavaScript is run inside the browser and thus enjoys the presence of DOM, the document object model. The DOM exposes all relevant parts of the browser and the web page in order to make JavaScript a powerful tool for programmers when creating rich and dynamic client side web applications.

However, JavaScript running outside the browser does not have the DOM in its side and thus lacks the tools to implement any meaningful applications. Most importantly JavaScript lacks the module system so there is no standard way of importing other modules. Furthermore, there are no means to publish an API

for external modules to consume while preserving part of the functionality internal. Finally, standard APIs for accessing the file system, networking, databases, web server and all other resources commonly available in other programming languages are also missing from JavaScript.

The CommonJS initiative is set out to fix the above problems by defining the module system and common APIs and interfaces to external systems. This is done without any modifications to the language and its syntax. Everything is implemented in terms of libraries that applications may import when needed. Moreover, as few globals as possible are exposed into the runtime environment. The most notable globals are the ones needed by the module system (`exports` and `require`). Being true to the nature of JavaScript the module system itself can be implemented in JavaScript using object-oriented design and functional programming with closures. This provides for seamless and natural introduction of interoperable modules to the language. Listing 1 shows an example of exporting and requiring functions.

```
// -- in file called myfuncs.js --
exports.myPublicFunction = function () { ... }

// -- in another file --
var mymodule = require("myfuncs");
mymodule.myPublicFunction();

// or simply:
require("myfuncs").myPublicFunction();
```

Listing 1. Exporting and Requiring

Utility libraries outside the scope of CommonJS, such as different test frameworks, parsers and database adapters, can be constructed as packages and distributed using the package management tool that is also specified by CommonJS. The package manager takes care of the dependencies and automatically installs all required packages, similarly to Python's EasyInstall and Ruby's RubyGems.

3 Architecture

The nature of our implementation (which is described in more detail in the next section) encourages us to work closely – if not directly – with the HTTP protocol. We will need to intercept HTTP requests before they are delivered to applications, and we wish to keep the design simple when implementing the interface. Moreover, the platform should support easy configuration to be able adapt into varying requirements. Driven by these premises, we have chosen to interface directly with the web server and to implement a stack of independent and interchangeable middleware components between the web server and the web application.

3.1 JSGI

JSGI (JavaScript Gateway Interface) is a specification driven by the CommonJS group to standardize the interface between the web server and the web application. The specification defines the format for creating JSGI compatible applications and their request and response objects. The request object can be thought of as the *environment* of the request as it contains all the parameters that CGI specification passes using environment variables, such as the requested URL, request method, and HTTP headers.

A JSGI application is simply a JavaScript function which takes exactly one parameter, the request, and returns an object with three properties: `status`, `headers`, and `body`. Listing 2 shows a minimalistic JSGI application.

```
var helloapp = function (req) {
    return {
        status : 200,
        headers : {"Content-Type":"text/html"},
        body : ["<html><body>Hello world!</body></html>"]
    };
}
```

Listing 2. Simple JSGI Application

3.2 Middleware

Generally the term middleware refers to network oriented integration software. For many web applications and more specifically in the context of this paper, the term "middleware" refers to a sofware component that sits between two other components and performs a task specific to that middleware component either before the HTTP request is given to the web application or after the application returns the HTTP response or both. Granted, this kind of usage could also be defined as layered architecture but we have chosen to use the term middleware as it has become somewhat pervasive in the realm of web server interfaces such as JSGI, WSGI and Rack. Usually a middleware component is generic component that may be applied to any web application but they can also be bespoke components used to configure or amend the web application it supports. A good example of a middleware component is the one that implements HTTP Basic authentication: it is generic to all web applications and it does not affect the behavior of the application itself.

A very useful feature of middleware components is that they may be chained one after the other to compose the desired behaviour for a given web application. This is a powerful paradigm for code reuse and configurability of applications. Fig. 1 shows an example of middleware chaining. In the example Content Length and HEAD middleware components operate on HTTP responses while Basic Authentication takes place before the HTTP request may be delivered to the application. After the authentication middleware component has verified the

client's credentials the request is delivered to the application to carry out the business logic. When the application has completed the request, the generated response is first checked by the HEAD middleware component to make sure that if the requested method was an HTTP HEAD the body of the response must be empty. After this, the content length middleware component calculates the size of the response body and adds the Content-Length attribute to the HTTP headers.

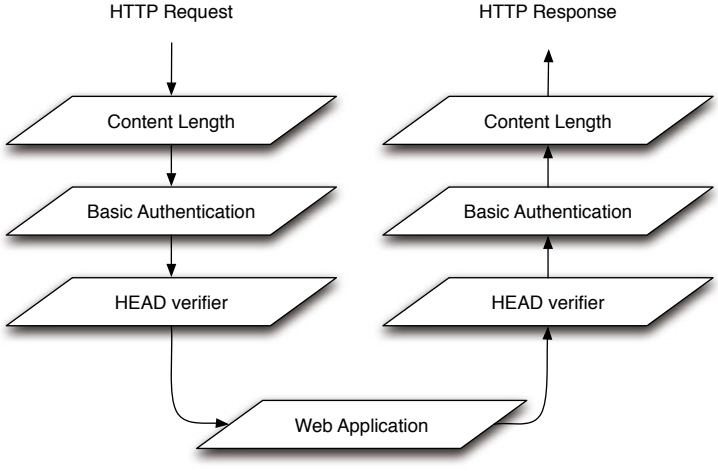

Fig. 1. Middleware Chaining

The JSGI interface has been designed to support middleware chaining. Conventions used for writing JSGI middleware follow the structure of writing JSGI applications themselves. The middleware usually stores the application – which can actually be another middleware component – in its closure in order to call it when the HTTP request comes in. At runtime, the middleware must look like an application to the calling party but at the same time act like the JSGI "framework" to the application it invokes. The common procedure when writing JSGI middleware is to

1. Have a function which takes in the request object.
2. Use the request object to optionally perform some pre-processing.
3. Invoke the original application (or the next middleware component).
4. When the lower level is finished, optionally perform post-processing on the response object.
5. Return the three element array specified by the JSGI to the upper layer.

Listing 3 shows a simple piece of middleware that logs all authentication headers into the application's log file. In the end of the example, we demonstrate how to construct an application with two middleware components.

```
var AuthLogger = function (app, authkey) {
    return function (req) {
        var i, res, auths, authkey = authkey || "authorization";

        // print authorization headers if present
        if (authkey in req.headers) {
            auths = req.headers[authkey].split(",");
            for (i = 0; i < auths.length; i += 1) {
                log.info(trim(auths[i]));
            }
        }
        // invoke the app and return whatever the app returns
        return app(req);
    }
};
exports.app = ContentLength(AuthLogger(helloapp));
```

Listing 3. Logger Middleware

4 Implementation

Our system – a platform codenamed "Groke" – is mainly targeted for, but not restricted to, rich and dynamic single page web applications. Hosted applications may be client side only, server side only, or split so that there are components running on both sides (Fig. 2). The platform helps in hiding the networking details when communicating between the client and the server; even if the application runs solely on the client, it is able to easily consume common services provided by the server side of the platform. The platform is implemented in JavaScript only and consists of mainly the server side components but also has an optional client side library for easy utilization for the developers.

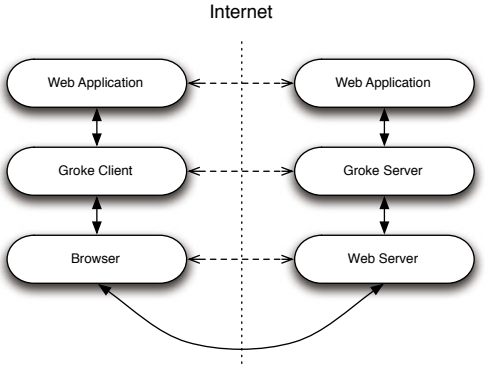

Fig. 2. System Overview

4.1 Partitioning

Code partitioning is implemented using the same `exports/require` mechanism as the module system specified by CommonJS. The platform makes no restrictions as to what may be exported or required remotely. So, a client side application may require and consume any public server side module or function it chooses. The philosophy is that everything that is not explicitly loaded into the client using `<script>` tags or some other means provided by the client environment are left on the server. The server side modules remain accessible to the client (can be called remotely) but they are not transferred over the network. This gives the developers full control over which components should be run on the server and which on the client. At the same time the platform does its best in hiding the networking details and thus minimizing the boilerplate code.

4.2 REST Inspired Interface

By default, the server side of the platform exposes all of its modules and their exposed variables (may they be functions, objects or properties) through a resource oriented interface. That is, the module hierarchy is simply turned into a hierarchy of resources. For example, the `list` function of the `file` module would respond to resource name `/file/list`. In order for a client to actually invoke this function, it needs to send an HTTP POST request to that URL with all the necessary parameters encoded as a JSON string in the body of the request. At the time of writing, the platform also has limited support for exposing objects and anonymous functions that may be created dynamically as part of return values from various operations. Table 1 summarizes the different types of resources exposed by the platform.

Table 1. Resource Identifiers

Resource type	URL scheme	Example
Functions provided by modules.	`/groke/module/` `<module>/<function>`	`/groke/module/` `file/list`
"Constructor" functions, i.e. functions invoked with the `new` operator.	`/groke/ctor/` `<module>/<function>`	`/groke/ctor/` `file/File`
Objects that are exposed as a return value of a function invocation.	`/groke/obj/` `<object-id>/` `<property>`	`/groke/obj/` `8327/write`
Functions exposed as a return values. This is analogous with the previous case.	`/groke/func/` `<function-id>`	`/groke/func/9284`

The interface supports HTTP POST method only. Surely, this seems like a violation to the rules of REST. However, the platform cannot have any understanding about the modules and functions it exposes; it only knows that it serves a set of operations as resources that clients may invoke. Furthermore, these operations may require input parameters which, by REST convention, are POSTed to these operations. Modules and functions as resources also have a static nature in that they cannot be explicitly created, replaced or deleted using the methods of HTTP. Although, as suggested by the Table 1, new objects and functions can be created and exposed dynamically but it is always more or less a side effect of a function invocation issued with HTTP's POST method.

Another controversy is presenting functions as resources. Albeit, the REST guidelines define that a resource can be anything, we do realize that presenting functions as resources makes the interface seem more like an RPC type of service. In the REST world it is highly discouraged to use verbs as the names of resources and this is what inevitably happens when exposing functions as resources. On the other hand this is not the platform's choise, the names of the resources are unknown to the platform when the system starts, everything is exposed dynamically and therefore also the names of the resources are chosen by the programmers providing the operations.

Because of these conflicting aspects we feel that it is not justified to call the interface truly RESTful but something that is inspired by REST.

For a client to consume server's functions by manually sending HTTP queries to various URLs would be quite cumbersome. This is why the platform adds a level abstraction on top of the RESTful communication layer by automatically generating the required code for sending and receiving requests. This is achieved by intercepting the request[2] that originally loads the client side application into the browser. When the request is intercepted by the server side platform, it reads the application from the disk and inspects the code in it. During this inspection the application loader finds all dependent modules and replaces them with generated stubs that are actually proxies to server side implementations. After that the instrumented application is sent to the browser.

The client application is now able to invoke the required functions as if they were local. In the background, however, the function parameters are encoded into JSON and POSTed to the RESTful interface of the server. On the server, the request is unmarshalled and requested operation is carried out. The result is again encoded in JSON along with some platform specific meta data about the result (such as possible exceptions) and sent back to the client.

4.3 CommonJS Compliance

Our implementation conforms to the CommonJS specifications and is built on top of the JSGI specification using middleware components to carry out the processing or requests. As for the runtime environment we have been using libraries

[2] This is an HTTP GET as opposed to POST mentioned earlier, which is used for consuming the RESTful interface after the application has been loaded.

called Narwhal[3] and Jack[4] running on the Rhino JavaScript engine. Narwhal is most likely the most complete implementation of CommonJS specifications currently. It is itself implemented in JavaScript and has support for different JavaScript engines. At the time of writing Narwhal has the best engine support for Rhino.

Jack can be thought of as a reference implementation of the JSGI specification. It is implemented as a CommonJS package and thus can be easily installed into the narwhal environment using the package manager. Jack has support for various web server technologies such as Java Servlets, CGI and the Simple framework [5]. So far, we have been using the Simple framework because of its speed and simplicity.

To enforce code reusability and to follow the JSGI mindset we have implemented several middleware components for the platform. For example, one that automatically invokes appropriate function on the underlying application based on the received HTTP method. This is very useful when implementing RESTful interfaces. Another convenient piece of middleware automatically strips the module and function information from the request URI and function parameters from the request body and hands them to the application as function parameters. The former of these middleware components is presented in Listing 4. The example demonstrates how easy it is to create useful JSGI middleware componentss that web applications may choose to use when appropriate.

4.4 Example Application

Our minimal example application shows how to write an application that makes use of the platform. We have written our own CommonJS compliant implementation of the `XMLHttpRequest` that is able to run on the server. The implementation is written in file called `xhmlhttprequest.js` and small portion of the code is shown in Listing 5. We have left out all the implementation details and most of the functions and properties but for the curious, the `XMLHttpRequest` is implemented using Narwhal's `http` and `event-queue` modules.

Applications running in the client are now able to use the server side XMLHttpRequest as a proxy to access data from other sites. An example client is written in Listing 6. There, the client simply `requires` the `xmlhttprequest` module and starts using it. Even though the application does nothing useful, it demonstrates how easily server side modules become available for the client. Note that the example uses syncrhonous requests, but the platform, and our own XMLHttpRequest both have support for asynchronous requests too.

4.5 Limitations

In the following we describe some of the limitations that the platform still has. To a large extent, they are not much limitations of the approach per se but more limitations of the current implementation.

[3] http://narwhaljs.org/
[4] http://jackjs.org/
[5] http://www.simpleframework.org/

```
// this middleware takes the request method and calls
// corresponding function on the app
var MethodGrab = function (app) {
    return function (env) {
       var method = env.method.toLowerCase();
       if (typeof app[method] === "function") {
           // call the app
           return app[method](env);
       } else {
           // "method not allowed"
           return {
               status: 405, headers: {}, body: []
           };
       }
   };
};

var message = "";

//the application can now have functions like``get'' and ``put''
var myapp = {
    get: function (req) {
        return {
            status: 200,
            headers: {"Content-Type": "application/json"},
            body: [JSON.stringify({"message": this.message})]
        };
    },
    put: function (req) {
      message = JSON.parse(req.body().decodeToString()).message;
        return { status: 204, headers: {}, body: [] };
    }
};

// expose the application with our middleware
exports.app = Jack.ContentLength(MethodGrab(myapp));
```

Listing 4. RESTful Middleware Component

Parameters and Return Values. Currently only basic types and strings are supported in remote function parameters and their return values. This is partly by design because we want to leave all functions to be executed in their intended environment. However, we could allow objects that have no functions (i.e. associative arrays) to be serialized and sent over the network.

Server Push. Ideally, from the communications perspective, the platform should be symmetrical. That is, the server should be able to remotely call client's functions the same way that the client is able to call server's functions. So far, we have been

```
exports.XMLHttpRequest = function () {
    ...
    this.onreadystatechange = null;
    this.open = function (method, url, async, user, pwd) {...};
    this.send = function (data) {...};
    ...
};
```

Listing 5. Example application running on the server

```
// require xhr (the implementation stays on the server)
var xhrlib = require("xmlhttprequest");

var xhrtest = function () {
    // creates new xhr object but leave it on the server
    var xhr = new xhrlib.XMLHttpRequest();
    xhr.open("GET", "http://www.somesite.com/rest/", false);
    xhr.send();
    alert(xhr.responseText);
};
```

Listing 6. Example application running on the client

testing different Comet approaches and the new WebSocket standard but these tests have been implemented as applications utilizing the Groke platform and not as part of the platform itself.

Garbage Collection of Remote Objects. As already mentioned, the platform currently has limited support for objects and functions as part of the RESTful interface. This is mostly due to lack of proper garbage collection of these objects. In the current version, we have degraded into using only timeout based garbage collection of objects. We leave the design and implementation of a proper garbage collection as future work.

5 Conclusions

Many approaches have been proposed for application development on the Web, but it is not obvious how the different approaches fit together. In this paper, we have introduced a middleware platform for composing JavaScript applications using a REST inspired interface. Unlike in many other approaches, we use JavaScript also on the server side to provide a uniform development model.

At present, the implementation is at a level of research software. By the time of the workshop, we hope to be able to release the system in open source under some commonly used license for wider audience. In the process, we have made code contributions to Narwhal and Jack projects but we have not really participated in the CommonJS specification itself.

References

1. Fielding, R.: Architectural Styles and Design of Network-based Software Architectures. Doctoral Dissertation, University of California, Irvine, USA (2000)
2. Papazoglou, M.P.: Web Services: Principles and Technology. Prentice-Hall, Englewood Cliffs (2007)
3. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly, Sebastopol (2007)