Service Oriented Architecture (SOA) Demonstrated as a REST Web Services Game System

Ron Harwood
CAS 703 – Winter 2015
harwood@mcmaster.ca

1. INTRODUCTION

A Service Oriented Architecture (SOA) is an approach in software architecture where individual components can provide (and consume) functionality to (and from) other components. Combined together, these components provide the functionality of a larger software system.

This project demonstrates an SOA through a simple game playing system and its associated infrastructure of services - implemented as a web service infrastructure. The web services use a REST (representational state transfer) [1] API (application programming interface), allowing for human readable URLs and greatly simplify the documentation process.

Two distinct MBSE (model based software engineering) tools have been used to design and help document this project. An ERD (entity relationship diagram) tool for the relational database design and a DSL (domain specific language) for the web service API design.

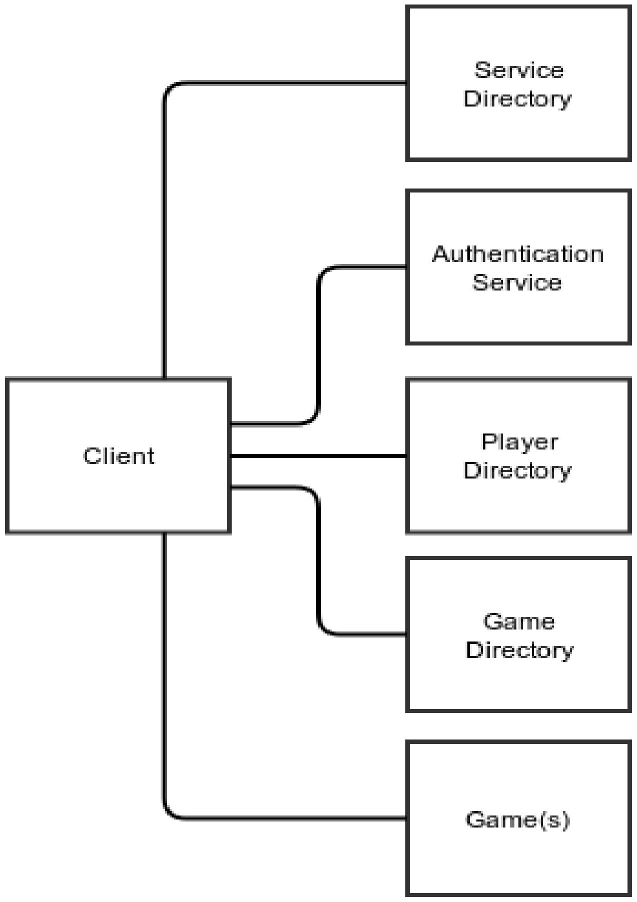

Figure 1: Service Oriented Architecture

2. SYSTEM DESCRIPTION

2.1 Top Level Service Directory

The system uses a top level "Service Directory" in which services are registered in order to provide service details to the other components, or external consumers of the services. Service listings are broken down into a basic categories (subdirectory, game, user oriented or "other"), given a simple service status (open or closed) and provide information about the service (a brief description and URL for accessing the service).

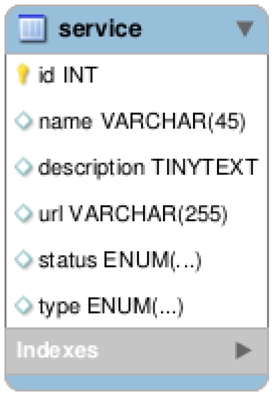

Figure 2: Service Directory Table Design

2.2 User Oriented Services

Two user oriented services are used in the system. An authentication service allows users to register accounts, log in and out and change (or recover) their passwords. A player profile system allows users to optionally provide a homepage for themselves and share their full name with other users of the system. The other services in the system utilize the authentication service (via HTTP cookies containing a username and token) in order to identify users and allow them to access user specific functionality.

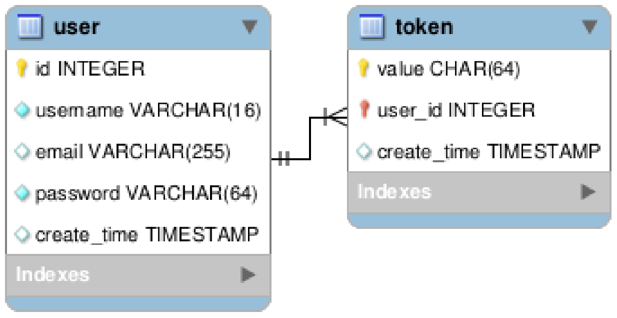

Figure 3: Authentication Service Table Design

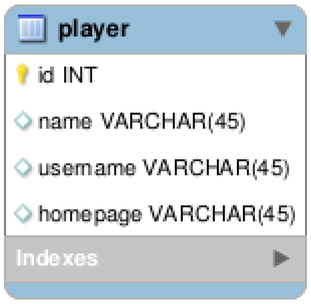

Figure 4: Player Profile Service Table Design

2.3 Game Directory Service

A games directory service provides players with a listing of the different games available in the system - providing a brief description and URL for each game. Additionally, games can be registered with (or removed from) the system through an administration interface, and the API allows game services to do this automatically.

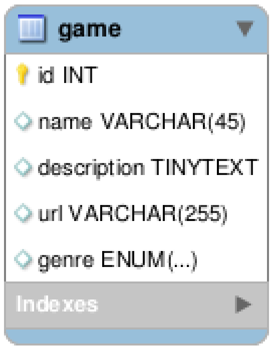

Figure 5: Game Directory Service Table Design

2.4 Demonstration Game Service

One very simple game service has been developed to demonstrate the functionality of the system. This game service allows players to publicly offer, join (accept offers) and play "Tic Tac Toe" with other users of the system.

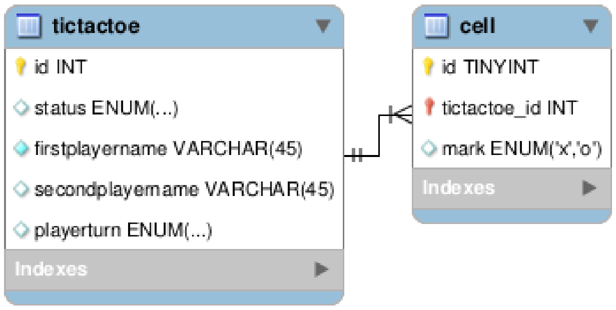

Figure 6: Tic Tac Toe Game Service Table Design

Nearly any turn-based game could be implemented as a game service in the system. Card games and board games lend themselves to this model quite nicely.

2.5 Web Based Front-End

A web based front-end client has been implemented to demonstrate the use of the services (with an administration interface) and the "Tic Tac Toe" game service provides its own web based front-end to facilitate playing the game. However, the system APIs could be consumed by any client (web, mobile, desktop, command line based or embedded systems) allowing users to play games on many platforms.

In order to demonstrate web service functionality and system modularity, the web front-end consumes the system service APIs via HTTP calls, rather than directly via source inclusion or database calls.

3. SYSTEM DESIGN

3.1 Web Services

The API for each service was described using the RAML (RESTful API Modeling Language) [2] DSL. RAML is designed specifically for describing REST and REST-like APIs from scratch. It is hierarchical and optimized for human readability:

```
#%RAML 0.8
---
title: service directory API
baseUri: http://techserv.ece.mcmaster.ca/~harwood/project/
version: v1

/service:
 description: Service directory
 options:
  description: Returns information about the service API
```

REST based services utilize the different HTTP methods (GET, PUT, POST and DELETE being the base methods) as "verbs" for accessing resources identified by URL/URI (uniform resource locator/identifier) [3] - the "verbs" indicate different interactions with the system resources, with the most common mapping being a "CRUD" interface:

POST - create a resource

GET - read a resource

PUT - update a resource

DELETE - delete a resource

The HTTP response codes returned by the application can indicate success, failure or identify additional requirements needed. For example, the authentication service, of the demonstrated system, can return one of three HTTP response codes:

204 No Content - indicating that the account has been successfully created

400 Bad Request - indicating that one of the required parameters is missing or otherwise has a problem

409 Conflict - indicating that the user name requested is already in use

Additional information can be provided in the body of the HTTP response for most (but not all) codes. In the authentication service, "409 Conflict" provides "The username is already registered" as further information in the HTTP response body, while "204 No Content" simply indicates success but provides no further information.

The numeric response codes allow programs to easily determine success or failure of a service request and the response verbage allows programmers to easily debug service interactions.

Accessing system resources via URI/URL allows definition of a standard interface for interacting with a service and allows for "human readable" addresses that programmers and system users can easily understand. For example, the player profile service of the system will return a list of player profiles when a GET request is received at the "/player" URI (the full URI would have the http protocol and server name as well) and will return a single specific player when a GET request is received at "/player/:userName" (where ":userName" is replaced with an actual username, without the colon - a construct of RAML to descript a variable in a URI).

The REST API process follows these steps:

1. Client sends HTTP request for a resource at a URI, using a standard HTTP method – any HTTP cookies and POST/PUT parameters are sent as part of the HTTP headers.
2. The server side code routes the request based on the URI and HTTP method.
3. Any HTTP header information is parsed as part of the request – including any parameters or cookies.
4. An HTTP response is created. This includes HTTP headers, HTTP response code, setting client side cookie data and the response body.
5. The client receives and interprets the HTTP response based on its response code and body content.

The full RAML listing for each service (but without the web based front-end URIs) is available in the source code. Additionally, an interactive web version of the RAML is available as part of the demonstration installation.

3.2 Database

Each web service has its own database back-end, designed using MySQL Workbench [4] - a visual database design tool. The visual interface of MySQL Workbench enables a database designer to create tables and relationships between tables using dynamic ERDs (entity-relationship diagrams) that can be exported as SQL (structured query language) or be used to directly to update a database. During development of the system, changes to the database design were immediately applied to the database using the direct MySQL interface.

3.3 Application Development

The system was written as a LAMP (Linux Apache MySQL PHP) application with the ADOdb Database Abstraction Library [5] and Slim Framework [6].

ADOdb in one of the many libraries available that provide a uniform interface to the variety of database systems that PHP can access. Additionally, the ADOdb Active Record extension [7] (an ADOdb implementation of the active record pattern) allows a programmer to access database tables and rows as PHP classes, objects and sets of objects - greatly reducing the amount of SQL needed in a program and allowing database changes to be immediately reflected in the properties of the classes and objects in the code. One-to-many relationships between database tables allow parent objects to load sets of child objects easily. This can be seen in the "Tic Tac Toe" implementation of the system, where the individual cells of the game board are child objects of an instance of a game - enabling game board data to be retrieved as a property of a game instance object.

The Slim Framework is considered to be a "micro" framework - providing the minimal functionality needed for web based applications and APIs. Slim was selected for this project for its REST oriented approach to responding to HTTP requests. It eases access to HTTP request data (like methods, cookies, post/put parameters and headers) and can quickly assemble HTTP responses components (like response codes, headers and body content). With some refactoring of the system service code, Slim would enable some API responses to be written in a single line of PHP. The RAML example given earlier for the OPTIONS method looks like this when written for Slim:

```
$app->options('/', function () use ($app){
        $app->response->setStatus(200);
                echo "This should return information on how to
use the API.";
});
```

4. FURTHER DEVELOPMENT

Given a larger time frame to develop the system, several potential areas of refinement could be explored. While MySQL Workbench generated the database SQL code from the ERD model, other code generation opportunities are available:

4.1 RAML to PHP Code Generation

With a standardized application framework, a method of transforming RAML specifications into PHP code could be utilized to speed up development. Either adapting existing model transformation tools or writing code to parse RAML and generate Slim Framework code (at the very least the scaffolding of routing statements).

Code generation of this nature is not without pitfalls. For example, any code modified by hand could be wiped out by regenerating code after changing the RAML specs or need to be drastically changed to reflect the model changes. Additionally, the order in which routing statements appear in the code can affect how routes are interpreted. In the "Tic Tac Toe" service source code, if the route "/:instanceId" appeared before the web front-end route "/play", requests for "/play" would be captured by "/:instanceId" and generate an error.

4.2 RAML to Database Mapping

Using RAML to describe the web service APIs accelerated code development and documentation greatly. Exploration of methods to map the web API resources to database entities or application classes and objects could possibly provide another opportunity to hasten the development process.

4.3 Security

Currently, authentication data (username and password) are sent over the network without encryption. Sensitive information, if not all information, should be sent using an HTTPS connection.

The authentication system uses a username/token pair. Currently, the tokens have no expiry but can be deleted with the “log out” feature. While they are uniquely generated values there is still a chance for interception and man-in-the-middle attacks.

A better security solution for a production system should rely on established authentication systems or services.

4.4 Error Handling

In order to make the code easier to read and hasten development, very little error detection and handling code has been added to the system.

A production system would need to detect and gracefully handle service failures and unexpected error codes from web services.

4.5 Service and System Features

The original project proposal had a second demonstration game service – checkers. The decision to not develop this game was made due to time constraints and the feeling that it would add no extra value over the “Tic Tac Toe” demonstration game.

The list of possible features for the system is endless. Extra data fields for player profile information, event logging and game leader boards are just a few features that could be added.

4.6 Automated Testing

Applications like SoapUI [8] can use RAML as a template to automate testing of an API. Since RAML can include expected responses, example data or data ranges, very detailed sets of tests can be designed and executed during development and after changes to API code to ensure that responses are still withing the design specifications.

4.7 Self Documenting API

Using the principles of HATEOS (hypermedia as the engine of state) [10] consumers of the system services would need no prior knowledge of of the operation of the API beyond basic REST standards and the base URI of the service directory. The OPTIONS HTTP method can provide information on each resource and its methods and each resource response body can contain additional information on related resources. For example, the authentication service API could return the following after a successful authentication request:

```
HTTP/1.1 200 OK
Content-Type: application/xml
<?xml version="1.0"?>
<user>
<username>harwood</username>
<token>3e3bb060ec638dbdb61b63b3872f9f17700875442d298fbe994ac46a82a
bedc4</token>
<link rel="tictactoe" href="/tictactoe?userName=harwood" />
<link rel="player" href="/player/harwood" />
</user>
```

This not only indicates the user has been authenticated (along with the token) but also provides links to the player profile resource and a link for listing game instance resources of “Tic Tac Toe” that the player is involved in.

5. CONCLUSIONS

5.1 API and Application Design Tools

There are several design tools that allow a developer to create an application from API to database using only models. They seem to fall into two categories. Either they are very expensive and designed for “enterprise” applications or they are tied to specific and/or proprietary platforms.

Tools that offer code generation from models are also often very platform specific or only generate class stubs (scaffolding).

While RAML and MySQL Workbench accelerated the process of creating the service APIs, significant programming, testing and debugging was still required.

5.2 Learning Opportunities

While the author has been using PHP for over 15 years (and programming in general for more than twice that), new features are introduced to the language on a regular basis. The Slim Framework uses “anonymous functions” (also known as closures) as callback parameters for routing definitions. Anonymous functions were introduced to PHP in version 5.3.0 in 2009 and the author has only just learnt of them now.

This is also the author's first use of RAML, and it will not be the last. RAML has shown itself to be quite useful for describing and designing web based APIs and will be valuable in the author's future projects.

6. REFERENCES

[1] Fielding, R. T.; Taylor, R. N. (2000). "Principled design of the modern Web architecture".
https://www.ics.uci.edu/~fielding/pubs/webarch_icse2000.pdf

[2] RAML (RESTful API Modeling Language) website
http://raml.org/

[3] Fielding, Roy T.; Gettys, James; Mogul, Jeffrey C.; Nielsen, Henrik Frystyk; Masinter, Larry; Leach, Paul J.; Berners-Lee (June 1999). Hypertext Transfer Protocol -- HTTP/1.1. IETF. RFC 2616.
https://tools.ietf.org/html/rfc2616

[4] MySQL Workbench website
https://www.mysql.com/products/workbench/

[5] ADOdb Database Abstraction Library for PHP website
http://adodb.sourceforge.net/

[6] Slim - a micro framework for PHP website
http://www.slimframework.com/

[7] ADODB Active Record documentation web page
http://adodb.sourceforge.net/docs-active-record.htm

[8] SoapUI API testing software website
http://www.soapui.org/

[9] PHP Anonymous Functions documentation web page
http://php.net/manual/en/functions.anonymous.php

[10] HATEOS entry on Wikipedia
http://en.wikipedia.org/wiki/HATEOAS