

SCAVENGER

Submitted by

MIKAYLA TIMM, JAMES HAWTHORNE, JEFFREY MORTON

University of West Florida

2 May 2016

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO.
	ABSTRACT	iii
	LIST OF TABLES	iv
	LIST OF FIGURES	iv
1.	INTRODUCTION	1
2.	PROJECT/SYSTEM REQUIREMENTS	1
	2.1 Project Requirements	1
	2.2 System Requirements	2
3.	TIMELINE	3
4.	DESIGN SPECIFICATIONS	3
	4.1 Overall System Architecture	3
	4.2 Software Specifications	4
	4.3 Communication Protocols	5
5.	TEST PROCEDURES AND RESULTS	6
6.	PROJECT OUTLOOK/CRITICISMS	14

ABSTRACT

The purpose of this project was to create the server back-end for a Scavenger application. Scavenger would be an app that UWF students could use during events such as orientation to familiarize themselves with locations on the UWF campus. To store persistent data, our server uses Google Firebase as a database to store information on location entities, user accounts, and tokens for session management. Our server accepts HTTP requests, using Spring as a request listener, from clients to create new user accounts, login to the system, add new locations to the database, and retrieve location information. Upon resolution of each request, our server returns a JSON object to the client, which the client can then parse and use however it needs to in order to accomplish the goals of the actual Scavenger application. To test our program, we used Postman as opposed to building a full-fledged client, and our test results show that the server functionality we projected to implement is working, though there are some shortcomings to the current system. Improvements on our system include adding encryption, hierarchical user accounts (admin privileges), and adjusting our event listeners to prevent multiple user access from interfering with existing unresolved requests.

LIST OF TABLES

Table 1: HTTP Request Parameters for Adding New User

Table 2: HTTP Request Parameters for Logging In

Table 3: HTTP Request Parameters for Getting Entity Information

Table 4: HTTP Request Parameters for Adding New Entity Information

LIST OF FIGURES

Figure 1: System Architecture

Figure 2: Postman Request and Response for Adding New User

Figure 3: Firebase Update for Adding New User

Figure 4: Postman Request and Response for Logging In

Figure 5: Firebase Update for Logging In

Figure 6: Postman Request and Response for Getting Entity Information

Figure 7: Firebase Update for Getting Entity Information

Figure 8: Postman Request and Response for Adding Entity Information

Figure 9: Firebase Update for Adding Entity Information

1. INTRODUCTION

The purpose of this project was to create the server back-end for a Scavenger application. Scavenger would be an app that UWF students could use during events such as orientation to familiarize themselves with locations on the UWF campus. Scavenger is similar to games such as Ingress and Pokemon Go in the sense that it uses location information from the real world to update aspects of the game. The server will be capable of logging users into the Scavenger service, going through authentication, requesting information on a location in the building, and then displaying the location information. Additionally, the server must allow for new user accounts to be created in the database as well as provide functionality for adding new locations into the database. The client will eventually be capable of scanning a QR code or using GPS information to look up locations and send the unique IDs over to the server over HTTP for getting location information. We are not fully developing the client in this project per se; we are developing the Scavenger server service that will be the backbone of the app and use an outside application to test this service.

2. PROJECT/SYSTEM REQUIREMENTS

2.1 Project Requirements

The server must be able to receive HTTP requests from clients and respond to them with the appropriate information. The server must be RESTful in that it should not keep track of state information for each client. In order to respond to the HTTP requests effectively, the server will save a token to Firebase[2] each time a user logs in. This token must be passed in with the HTTP request parameters each

time the client wants to access location entities or update location entity information in Firebase.

The server must resolve the HTTP requests for creating user accounts, logging in with authentication to the Scavenger service, retrieving location information from the database, and adding new location information to the database. The information returned to the client must be in the form of a JSON object. The client can then use this data however it needs.

2.2 System Requirements

Our server was to be written in Java. Google's Firebase was to be used for our persistent data, including user accounts, location information, and tokens for keeping track of user sessions. We ended up not using Firebase for authentication and chose to implement our own authentication functionality, as Firebase did not work well for authenticating individual users connecting through a Java server. We also chose not to implement a standalone client for testing as we chose to focus on the actual functionality of the server. We used Postman to generate the HTTP requests to test our system.

For deployment, the server must be able to run on any system with a JVM and an internet connection. The client will send HTTP requests containing any parameters necessary for the request they are performing. This will include the authentication token (REST [3]) and location URI for requesting for and updating location entities, or a username and password for creating a new account or logging in. Instead of running on the school's ssh server, we are hosting our own server because we encountered issues with the school's server being able to communicate externally with clients and Firebase. Instead of using two servers, one for communication and one for ID, we implemented a single server with separate

endpoints for authentication and entity requests. The server communicates via HTTP with Firebase to get and update the location information and user information. The server then sends the information back to the client over HTTP in the form of a JSON object.

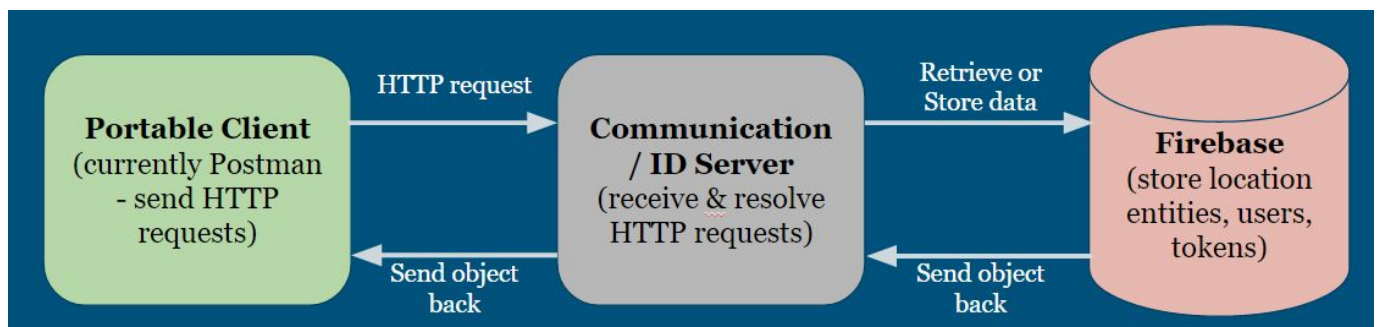
3. TIMELINE

- Project plan completed: 1/27
- Project plan presentation: 2/9
- Project design: 2/17
- Presentation 2 (first results - prototype of functionality): 3/30
- Authentication, server, and database communication working with dummy test client (Postman): 4/23
- Final presentation: 5/2

4. DESIGN SPECIFICATIONS

4.1 Overall System Architecture

Figure 1: System Architecture



As seen in Figure 1, our System architecture consists of 3 main parts: the client, the communication/ID server, and the Firebase database. The client sends a request to one of the endpoints on the communication/ID server. Depending on

which endpoint was used for the request, the server either creates a session token and returns it(/login), checks the session token that was passed(/entity), or creates a new user(/newUser) with username and password combo. To check the token, create a new token, or access user or entity information, the server contacts Firebase using a DatabaseReference object that accesses the child entry, identified either by the token ID, user ID, or entity's unique ID, in the appropriate table (entities, users, tokens). We then bind an event listener to the child and save the value returned by Firebase. Firebase returns a JSON object that the Firebase libraries parse directly into the corresponding Java object that it was initially saved as in the database. Depending on the response from Firebase, we either return the appropriate data for the request that was sent, or a failure message to the client.

4.2 Software Specifications

The Java classes we implemented for data in our program include a User class for logging in, Entity class for location entities, and Token class for storing tokens with expiration times. Our RESTful Web Service Controller resolves HTTP requests from clients for adding new users, logging in, adding new entities, and receiving information on existing entities. We use Spring[1] by Pivotal as one of our utilized technologies for receiving HTTP requests. Spring establishes a listener for HTTP requests and maps them to the appropriate functions using @RequestMapping annotations in our RESTful web service controller. It then binds request parameters to method parameter variables to be used by the controller in resolving the requests. Spring uses the Jackson JSON library to automatically marshal instances of our objects into JSON to be sent back to the client. Our online database is Firebase by Google. The functionalities we used for

communicating between the server and Firebase included retrieving and storing Java objects. Firebase automatically converts Java objects to JSON before storing them in the database, and when data is retrieved, it returns the JSON object as a “DataSnapshot” which can then easily be parsed directly into the appropriate Java object using the `getValue()` method provided by the Firebase libraries.

4.3 Communication Protocols

For communication, we use POST and GET HTTP requests, and we send requests to the endpoint that returns the data we desire. The following are the methods used for each endpoint:

- /login uses GET
- /newUser uses POST
- /entity uses POST to add an entity and GET to get information about an entity

Both the `newUser` and `login` requests require a username and password. The `newUser` endpoint adds a new username and password pair to the database, while the `login` endpoint checks to see if the username and password pair is valid. If the username/password pair is valid the server creates a new token and returns it. Both entity requests require a valid token ID to be passed in with the request. The POST request for an entity also requires information about the entity. The GET request only requires a location ID in addition to the token ID.

5. TEST PROCEDURES AND RESULTS

We used Postman to test our API and serve as our “dummy” client. Postman is used to make HTTP requests to our server and receive responses from the server. To test our functionality, you must first open Postman. The request formats for all of the functionality in our program are described in this section.

Adding New Users: POST <http://scavenger.csproject.org:8080/newUser>

Table 1: HTTP Request Parameters for Adding New User

key	value
username	myuser
password	mypass

A successful request returns the JSON representation of the User object created in the Database as seen in Figure 2 on the following page. Firebase also updates its users table with the username and password parameters entered in this request as seen in Figure 3 on the following page.

Figure 2: Postman Request and Response for Adding New User

The image shows a Postman interface for a POST request. The URL is `http://scavenger.csproject.org:8080/newUser?username=rubberDucky&password=quack123`. The Params tab is active, showing a table with two rows: `username` with value `rubberDucky` and `password` with value `quack123`. Below the table are tabs for Authorization, Headers, Body, Pre-request Script, and Tests. The Authorization tab is selected, showing 'Type' as 'No Auth'. Below that are tabs for Body, Cookies, Headers (3), and Tests. The Body tab is selected, showing a JSON body in 'Pretty' view:

```
{
  "username": "rubberDucky",
  "password": "quack123",
  "visited": [],
  "currentLocation": null
}
```

 The status bar at the bottom right indicates 'Status: 200'.

Key	Value
username	rubberDucky
password	quack123
New key	value

Authorization Headers Body Pre-request Script Tests

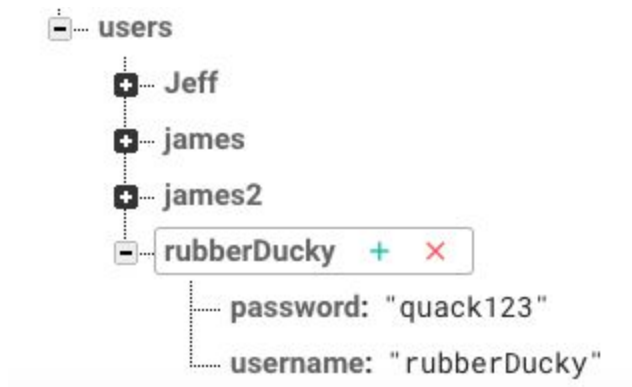
Type No Auth

Body Cookies Headers (3) Tests Status: 200

Pretty Raw Preview JSON

```
1 {
2   "username": "rubberDucky",
3   "password": "quack123",
4   "visited": [],
5   "currentLocation": null
6 }
```

Figure 3: Firebase Update for Adding New User



Logging into the Scavenger service: GET

<http://scavenger.csproject.org:8080/login>

Table 2: HTTP Request Parameters for Logging In

key	value
username	myuser
password	mypass

A successful request returns a unique key (tokenID) for the user session. This tokenID is also saved in Firebase in the Tokens table (Figure 5 on the following page). A successful request and response for login is shown in Figure 4 on the following page. The client will need to pass this token in when getting information on entities from the database or when adding new entities to the database. The token will expire 24 hours from when it was created (this expiration time is stored in database).

Figure 4: Postman Request and Response for Logging In

The image shows the Postman interface for a GET request. The URL is `http://scavenger.csproject.org:8080/login?username=rubberDucky&password=quack123`. The Params tab is active, showing a table with the following data:

Key	Value
username	rubberDucky
password	quack123
New key	value

The Authorization tab is selected, showing "Type" as "No Auth". The Body tab is also visible, showing the response body in "Text" format:

```
1 Successful login. Token ID: 7JYC38JRGAM9JA9EU0rubberDucky
```

The status is 200.

Figure 5: Firebase Update for Logging In

The image shows the Firebase Realtime Database console. The 'tokens' node is expanded, showing a list of tokens:

- 1UFKVWFAJ4FTHUJSC7james
- 4AB7JJVS4NFUDWEQG5Jeff
- 553T0A3Q71HJTHZI8Rjames
- 7JYC38JRGAM9JA9EU0rubberDucky

Getting Entity Information: GET <http://scavenger.csproject.org:8080/entity>

Table 3: HTTP Request Parameters for Getting Entity Information

key	value
tokenID	myTokenID
id	uniqueID

A successful request returns a JSON representation of the Entity object obtained from the database containing information about the queried entity as seen in Figure 6 on the following page. Firebase will then update the current location and the visited locations list with this location Entity for the user in the Users table corresponding to the tokenID passed in as seen in Figure 7 on the following page. The tokenID for the session also must be valid when passed in to retrieve a valid location's information. If not, the server will return an error message to the client.

Figure 6: Postman Request and Response for Getting Entity Information

http://scavenger.csproject.org:8080/entity?tokenID=4AB7JJVS4NFUDWEQG5Jeff&id=uniqueID3


GET ⌵ http://scavenger.csproject.org:8080/entity?tokenID=4AB7JJVS4NFUDWEQG5Jeff&id=uniqueID1

Key	Value
tokenID	4AB7JJVS4NFUDWEQG5Jeff
id	uniqueID1
New key	value

Authorization Headers Body Pre-request Script Tests

Type No Auth ⌵

Body Cookies Headers (3) Tests

Pretty Raw Preview JSON ⌵ 

```
1 {  
2   "roomNum": "435",  
3   "isA": "room",  
4   "contains": "berndok",  
5   "annotatedBy": "Room 435 in Building 4, office of Dr. OK",  
6   "illustratedBy": "dr-ok.jpg",  
7   "uid": null  
8 }
```

Figure 7: Firebase Update for Getting Entity Information



Adding New Entity Information: POST

<http://scavenger.csproject.org:8080/entity>

Table 4: HTTP Request Parameters for Adding New Entity Information

key	value
tokenID	myTokenID
id	uniqueID(of new entity)
isA	room/person/whatever
roomNum	room Num of new entity
contains	Whatever it contains (person, students, etc)
annotatedBy	String with info about entity
illustratedBy	Image.jpg (or similar)

A successful request returns a JSON object of the new Entity object added to Firebase as seen in Figures 8 and 9 on the following page. The user must be already logged in and pass in their tokenID in order to add new Entities to the database.

Figure 8: Postman Request and Response for Adding Entity Information

POST Params

Key	Value
id	uniqueID8
isA	Lecture Hall
roomNum	404
contains	Projector, podium, smartboard, distant learning hardware.
annotatedBy	Room 404 in Building 4, general lecture lab
illustratedBy	lecture-lab.jpg
New key	value

Authorization Headers Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (3) Tests

Pretty Raw Preview JSON

```
1 {
2   "roomNum": "404",
3   "isA": "Lecture Hall",
4   "contains": "Projector, podium, smartboard, distant learning hardware.",
5   "annotatedBy": "Room 404 in Building 4, general lecture lab",
6   "illustratedBy": "lecture-lab.jpg",
7   "uid": "uniqueID8"
8 }
```

Figure 9: Firebase Update for Adding Entity Information

uniqueID8

- annotatedBy: "Room 404 in Building 4, general lecture lab"
- contains: "Projector, podium, smartboard, distant learning hardware."
- illustratedBy: "lecture-lab.jpg"
- isA: "Lecture Hall"
- roomNum: "404"
- uid: "uniqueID8"

6. PROJECT OUTLOOK/CRITICISM

After testing with Postman as our “client”, we determined that the server backend works. A separate client program can be easily created which uses the HTTP requests as described to communicate with the server. The server will respond with json objects in the format shown above which the new client program can then parse for use in the client’s code/UI.

One criticism we have is that currently multiple clients requesting information from the server in quick succession may cause unexpected responses since the server tracks the state of requests with static variables. This issue can be easily resolved by figuring out a way to prevent a calling function from continuing its execution until the event listener for the callback returns results.

Another criticism is that encryption is basically non-existent in our messages sent between server and client. If another user/device intercepts a message from the client to the server, it can harvest the REST token that the client is using to create/access locations and use that same key to falsify legitimate access to the server. This issue can be resolved by either using some sort of running encryption scheme based on date and time, using SSL to encrypt the entire message, or a combination of the two. Google has implemented encryption in its libraries that access Firebase. This means that the communication between server and Firebase is secure, however communication between client and server is still an issue.

Additionally for the final product (production release) users should have an admin flag so that not all users can add locations to the database. This additional requirement for adding locations will keep the database from filling up with excess

data.

7. APPENDICES

Appendix 1: Manuals

Technical/Application Manual:

To continue working on the Scavenger project/build an application around our existing architecture, a client will need to be built that uses the HTTP communication protocol described above to communicate with the server. The client will be able to use some kind of ID (possibly scanned from a QR code) as the lookup id for entities in the database. The IDs of the database entities should be changed to correspond with the matching QR codes.

If new functionality is needed in the Scavenger server, additional endpoints can easily be created by expanding on the existing controller. You can do this by either modifying the functionality in the current methods to change the behavior or by adding additional methods for resolving new types of requests using Spring's RequestMapping annotation (as seen in the current controller). Additionally, admin privileges should be considered in the next stages of implementation. This should be implemented to prevent average users from adding new entities to the database.

Installation Instructions:

To install the Scavenger server, first make sure you have the latest Java Runtime Environment (JRE) installed on the machine you wish to install the server on. Java JRE download can be found at <https://www.java.com/en/download/>. There are several ways to install the server from this point.

Option 1 (easiest method):

1. Download the Scavenger Standalone Server zip from github.
2. Extract the “Scavenger Standalone Server” file to the installation location of your choice.
3. From this point, you have a few more options.
 - a. On a Windows based machine, you can simply run the “StandaloneScavengerServer.bat” batch script to launch the server.
 - b. On a Unix based machine with bash, you can simply run the “StandaloneScavengerServer.sh” bash script to launch the server.
 - c. For a platform independent method, run the command “java -jar gs-rest-service-0.1.0.jar” from the command line.
4. For a platform independent method, run the command “java -jar build/libs/gs-rest-service-0.1.0.jar” from the command line.
5. At this point, your Scavenger server is now running. You can access the Scavenger server at <http://localhost:8080>. If you would like to access the server from outside of the machine it’s running on, you will need to go to your DHCP server and set a static IP for the machine you are going to run the server on. Set appropriate firewall exceptions if necessary for port 8080. The server can then be accessed from LAN with the server machine’s static IP address. If you would like to access the server from outside of the LAN, you will need to portforward port 8080 to the static IP of your server machine from your router’s configuration interface. If all steps listed above have been completed successfully, you should now be able to access the Scavenger server from anywhere on the internet via your WAN IP at port 8080. If you would like to change which port the server is accessible on

from WAN, you will need to modify your portforwarding in your router config.

Option 2 (compile the source code):

1. Download the source code from GitHub (or wherever you have it hosted if you're it in a different location). Copy it to a directory of your choice.
2. Inside the downloaded folder, you should see a directory named "snorlax". Inside this directory, you should also see "build", "gradle", and "src" folders.
3. Install the latest Java JDK (1.8 or later).
4. Install gradle according to the instructions found at <https://gradle.org/install>.
5. Import the project into a Java IDE of your choice (must support Maven).
6. Import the gradle build file. At this point, your project should now be ready to run from within the IDE of your choice. You can access the Scavenger server at <http://localhost:8080>. See Step 5 from install option 1 for instructions on accessing the server from outside of your machine.
7. If you would like to compile a standalone jar executable for the scavenger project, you can build the project into a standalone jar file by running `./gradlew build` from the command line at the root of your project folder.
8. To run the standalone server jar, navigate to the root of your project and run `java -jar build/libs/gs-rest-service-0.1.0.jar`.
9. If you would like to repackage this jar for production deployment, see the following instructions:
 - a. Copy the standalone server jar from `build/libs/gs-rest-service-0.1.0.jar` from your project root directory to a new directory of your choice.

- b. Copy the 2 firebase json authentication files,
“scavenger-6da2c-firebase-adminsdk-uhm2z-ac9e99e869.json” and
“scavengerserver-firebase-adminsdk-qh4li-dbf0d16986.json” to your
new directory.
- c. Run the server from command line in your new directory with the
command “java -jar gs-rest-service-0.1.0.jar”. You can access the
Scavenger server at <http://localhost:8080>. See Step 5 from install
option 1 for instructions on accessing the server from outside of your
machine.

Appendix 2: Declaration of responsibility

Declaration of Responsibility

Mikayla Timm: Sections Abstract, iv (Lists of tables & figures), 1, 2.1-2.2, 3, 4.2, 5, Technical/Application Manual

Signature: *Mikayla Timm*

James Hawthorne: Sections Abstract, 4.1, 4.3, 6, References

Signature: *James Hawthorne*

Jeffrey Morton: Sections Installation Instructions

Signature: *Jeff Morton*

8. REFERENCES

- [1] Spring v1.5.2.RELEASE, license: <https://github.com/spring-projects/spring-boot/blob/master/LICENSE.txt>
- [2] Firebase v4.1.7, license: <https://firebase.google.com/terms/>
- [3] Fielding, R.T. (2000) ‘Architectural Styles and the Design of Network-based Software Architectures’, University of California, Irvine