# BotBoard Project

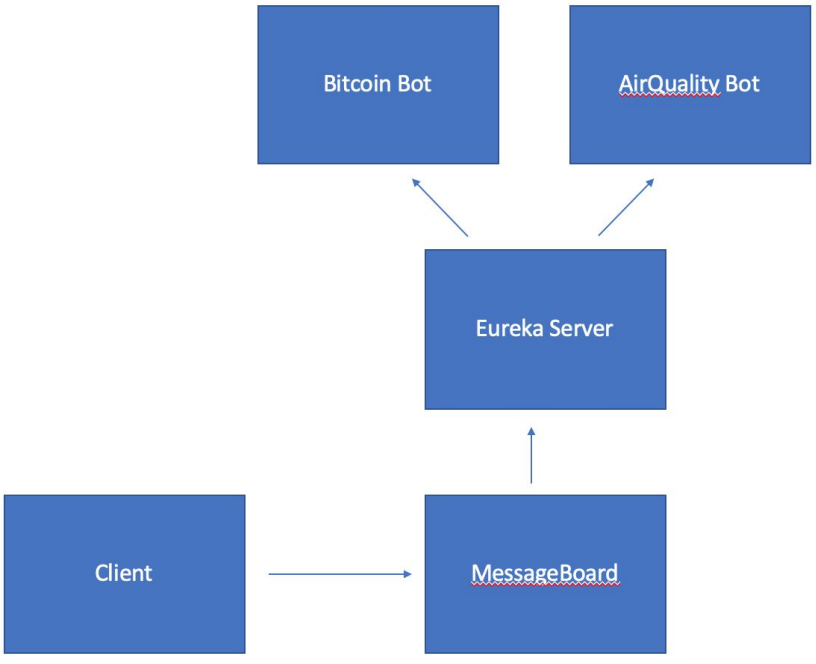
**1. Overview of Problem**

At the most fundamental level our project is to build a web service that is a message board. Our board works such that multiple users may “post”(write/add) messages to different “threads”(lists of messages to which new messages are appended to). A user may post messages to an existing thread, or create a new one. We wanted to build our message board system so that it is not limited by location or machine, i.e., is not an entirely standalone application but it is distributed, so that a user may access the board from any location on any machine.

What separates our “BotBoard” from a simple fundamental message board is the functionality and services it provides in the form of “Bots”. When a user writes *@SomeBotName*(e.g. @Bitcoin) in their message body, if there is a bot service with that name then that Bot will append a post to the user’s post containing a message with some information. For example, if a user writes @Bitcoin in their post, the bitcoin bot will append a post with the current conversion of 1 euro to bitcoin. Furthermore we want the ability to add/remove new Bot services for the clients to use without having to restart our BotBoard(server).

The advantage of a message board not limited by location which can be accessed by any machine, over a message board which does not have those properties, is plainly clear. To be able to achieve the former, we need a distributed message board application with a kind of client-server model. In a similar fashion, if our “Bots” are distributed so that we may independently run them on separate machines so long as they are under the same IP address so that the client only need one url to use the system. This means we will then have a more flexible and versatile system which also puts less stress on the server machine. Such a distributed solution to the question of “Bots” is also necessitated by our requirement that we are able to add/remove “Bot” services without having to restart the server. Thus it is appropriate to have a distributed service registry in place which the “Bots” can connect to the registry to be “visible” by the client for use in the system.

**2. System Architecture (max. 1 page)**



Our system comprises of a server, messageboard, bitcoin bot and an airquality bot. All 4 components are built using Spring Boot to run them as rest webservices.

The server is our service registry for the system. It is built using Netflix's Eureka service discovery system on top of Spring Boot. The server application starts up a Eureka Server for different parts of our system to connect to, to be able to be 'discovered' by the clients for use.

The messageboard is our base server API for our system. It holds the threads and post in memory and exposes endpoints that can be use to add new posts(POST), retrieve existing ones(GET), update an existing post(PUT), or delete existing ones too(DELETE). It also makes calls to the bots based on the command put into the body of a post added to the thread. Clients use the messageboard endpoint to connect to, to POST/GET/DELETE/PUT new posts and threads.

The bitcoin bot is one of the two bots we have currently built into the system. It registers itself with the Eureka server available for the messageboard to call it. It returns the price of 1 euro in Bitcoin.

The airquality bot is the second bot we have built into the system. It also registers itself with the Eureka service. It returns the air quality value of the nearest city based on IP address of the ec2 instance. We hope to expand this further to make greater use of the airquality API that we are using to allow deeper customisation of what airquality data is retrived.

**3. Technology Choices**

| Technology | Motivation for Use |
|---|---|
| AWS Platform | By hosting the server on the cloud, our BotBoard service may be accessed by any client device with an internet connection. |
| REST | Because it is the best. |
| Spring Framework with Maven | Greatly simplifies the use of Java Servlet and other technologies, especially when it pertains to using the Eureka Service Registry. |
| Netflix Eureka | Dynamic Service Discovery. |
| Swagger | Automatically generates REST API descriptions. The Swagger UI allows for testing of the API, and effectively doubles as a client-side web interface. |

**4. The Team**

| Student Number | Name | Assigned Task(s) |
|---|---|---|
| 13315756 | Conal O'Neill | - Netflix Eureka service discovery with Spring<br>- Configure AWS EC2 server<br>- Integrate and Implement "Bots" |
| 17459182 | Nikolai Gladychev | - Develop server API<br>- Implement server API<br>- Swagger addition |

**4. Task List**

**4.1 Task 1: Develop Server API**

We want to build the server as a web service, as then our system, BotBoard, may be accessed by any client with an internet connection. So the server API consists of the REST methods available to the client. A way to fully support all the functionality of the message board with only CRUD methods had to be found. The client needs to be able to retrieve all of the information on BotBoard as well as append messages, and potentially change some information. The best way to do this, it was decided, was to use URL path variables to access different parts of the board:

1. Root URL: “host:9000/board/” would be the root URL which supports a GET method to return a list of threads, and a POST method to add new threads. This root URL returns only a list of threads(not the posts), since we envision clients not wanting all the threads as well as all of their contents as the result of one GET request.
2. Thread URL: “host:9000/board/{threadid}” would be the unique path of an individual thread, and would similarly support GET, POST requests but for a list of posts, rather than a list of threads. It would also support a DELETE request which would remove the whole thread at the URL.
3. Post URL: “host:9000/board/{threadid}/{postid}” would be a non-essential but potentially useful path of a unique post. It would support a GET request(to return the post), and a PUT request so that modifying a post after its initial posting is possible.

After we had worked on the project for some time, it became clear that there would also be another GET request for the URL: “host:9000/swagger-ui.html”, which would return a Swagger UI webpage of the automatically generated Swagger specification. Because the API may be tested from here, it also doubles as our client web-interface.

**4.2 Task 2: Implement Server API**

The next step was to implement the server API in Java. Initially it was implemented with the RESTlet framework, however was later re-written using the Spring framework with minor alterations(to support Eureka, and because some RESTlet extensions behaved strangely and/or had poor documentation). The implementation logic was largely unchanged however. We had three Java classes to handle the requests for each of the URL patterns described in Task 1, a class to handle the overall BotBoard data structure(also launching it on the server), and classes for the post and thread data structures.

The main data structure is a java LinkedList of the MessageThread class, which contains the posts(Post class) of the thread as well as information about the thread itself. Because threads may be deleted, but their ids are unique, it was necessary to have a implementation method to find the position of a MessageThread in the list based on its unique id. A similar method was needed to find the positions of Post objects. We used the Jackson library for serialisation/deserialization of java objects to JSON, for interaction between the BotBoard web service and the client. Junit tests aided in the development of the implementation to ensure existing functionality wasn’t broken. The following was implemented:

For the Root URL we created the MultipleThreadController class. The GET method returns a list of the thread information of each thread but not the posts themselves(this is aided by the MessageThreadInfo class). The POST method takes a Post object, and if the thread id value of that object is 0, then it creates a new thread with that Post as it’s first Post. If the thread id is not 0, and matches an already existing thread, then the Post is added onto the end of that thread. If the thread id is not 0 and does not match an existing thread an error response is given. Special consideration had to be given to giving new threads a unique thread id not already on the board, and to the special case when the board has no threads.

For the Thread URL we created the ThreadController class. It was necessary to write checks for the path variable {threadid}, that it is indeed a thread that currently exists, otherwise an error response is given. The GET method returns a list of all the posts in the thread. The POST method adds a post to the end of the list with correct unique post id. The DELETE method removes the thread at this URL from the root data structure(if it exists).

For the Post URL we created the PostContoller class. It was again necessary to write checks for the path variables. The GET method takes the unique post id(path variable {postid}), finds the unique post in the root data structure and returns it. The PUT method takes a Post object, and checks that its thread id and post id values match the path variables, otherwise it assumes the client has made some error in their request and returns an error response. It then finds the Post within the root data structure and then updates it with the given Post object. The DELETE method deletes the unique post object(according to the path variables) and removes it from the root data structure. A special case is where there is only one post in the thread in which case deletion is restricted. This was done because we do not want the case where a thread has no posts, and so to delete such a post, the entire thread must be removed.

**4.3 Task 3: Host Server on AWS EC2(max. 1 page per task, but prefer ½ page)**

For our system to be the distributed easily accessible one that we envisioned, we need to host our system with a cloud computing provider for it to be accessible through the internet. We chose an AWS EC2 instance for this as it gave us control to implement the code as we needed to as well as allowing us to have on demand compute power should we need it.

We have the system running as background processes on the instance so that we can log out of the instance and keep the system running as all data is held in memory. This was one of the difficulties we faced as we had no running logs when testing the system so had to remain logged in and running it from the command line to test. It was only when we finished it that we could then use the nohup command to run the system in the background to be able to log out without stopping the system running.

**4.4 Task 4: Integrate and Implement Bots**

The bots are quite straightforward and there is not that much code to them, however the difficulty came in trying to access the external APIs to be able to get the data for the bots. Eureka by default is a service discovery system for an internal system so it can act as a load balancer. However we needed to edit the configuration and default set up of the bot client Rest controller so that it could access the external API and not be locked inside the Eureka system.

The bots are called from the MessageBoard app. When a new post is added to a thread, the app searches the body of the new post to try find a match to the regex. This regex is built to find occurrences of '@' followed by a word e.g. "@bitcoin". It then takes this word and searches the list of registered services on the base Eureka Server. If it finds a service with the same name as the word found then it makes a call to that service. That service then makes the call to the external API and returns the data back to the MessageBoard which appends the data as a post to the thread.

**4.5 Task 5: Use Swagger for REST API Documentation and as a Web UI**

We made use of Swagger for documentation of our system. The swagger documentation is created through use of annotations on each of the method that are exposed as endpoints for the service. These annotations allow us to add custom descriptions and response codes to the documentation. This swagger documentation is available in either a json form or in a web UI. The json form is available at 'host:port/v2/api-docs' and the web UI is available at 'host:port/swagger-ui.html'

There is a swagger documentation generated for each of the four parts of the system, the base eureka server, the messageboard service, and each of the the two, bitcoin and airquality bots.

**5. Reflections**

Overall we feel the project was a success and works well, in the sense that we achieved what we initially set out and proposed to do. Certainly we are pleased with the suite of technologies we were able to bring together to enhance our project. Given more time and more manpower, of course there is much more we could add to this project. We could elaborate and polish the details of distribution, develop a more complete or standalone client interface, and we could include authentication for users on BotBoard so that certain requests can only be accessed by a certain class of users. One feature we did not have the time to make is to provide the client a live list of the bots currently available for use.

Ultimately, the technologies we used were appropriate for our project. Initially we used the RESTlet framework, but for our uses found the Spring framework to be more useful. We used REST which was certainly the right decision. In terms of REST being the easy and simple choice, as it is advertised, is something we personally verified as being true. We were able to very rapidly create the Server API and implementation, and our project is not complex enough where we at any point felt SOAP could offer us advantages over REST. Hosting the server on the cloud with AWS is certainly more versatile than having a dedicated server, as we do not personally have access to a reliable, and in general our interaction with this server was smooth.

We will admit to some mistakes of the kind: “in retrospect we would have approached it differently”. With the technologies new to us that we used, and a somewhat “learning on the fly” kind of approach, we sometimes went down the wrong path for a certain time. In terms of productivity this has advantages, but it means that best practices weren’t necessarily always followed, and maybe a suboptimal sub technology was chosen. An example of this is using path variables instead of query parameters for “host:9000/board/{threadid}/{postid}”. Query parameters may have worked better and followed best practice. And now we also understand why it is best practice, as we had some trouble in our approach where we could not make the URLs simply “host:9000/{threadid}/{postid}” as this would block the client from accessing “host:9000/swagger-ui.html”. So in retrospect we may have researched the technologies a bit more before using them, but then there is the question of “when is enough” as too much research is as bad as too little.

One of the biggest advantages and benefits we got in this project was by using Netflix Eureka as a service discovery. This gave us the ability to start and stop the bots on the fly without having to restart either the base eureka server or the messageboard service either. It also means that we can restart either the base eureka or messageboard services while keeping the bots running as well as once the base eureka server starts back up it will re-register all the existing services that are still running and register any new ones as well. This ability has been invaluable for our system as it gives up the power to add new bots in the future without restarting anything or having to add in additional code to be able to ‘call’ the bots from the new posts added. Eureka also means that if a bot is called but is not running and not registered with the base eureka server then the system does not crash, it carries on like a bot was not called. If we had to hardcode in the urls for the bots then if the bot is not running the system would crash.