



ACCA7004 WEB DEVELOPMENT/ ACCA7005 WEB SERVICES

REST Implementation Best Practices and Serverless Technology

Abstract

This report will analyse the best practices for RESTful web service implementation and will critically analyse Serverless technologies and the use of the circuit breaker pattern.

Edward Prasad
1809704

Table of Contents

1.	Introduction	3
2.	Best Practices for RESTful Web Service Implementation	3
2.1.	Introduction	3
2.2.	What is REST?	3
2.3.	Best Practices for RESTful Web Services	4
2.3.1.	Richardson Maturity Model and HATEOAS.....	4
2.3.2.	Platform Independence and Stateless.....	7
2.3.3.	Security	8
3.	Serverless Technologies and Circuit Breaker Pattern	10
3.1.	Serverless Technologies	10
3.1.1.	Introduction and Background	10
3.1.2.	What is Serverless?.....	10
3.1.3.	Drawbacks of Serverless	11
3.1.4.	Serverless Conclusion	11
3.2.	Circuit Breaker Pattern.....	13
3.2.1.	Introduction	13
3.2.2.	Circuit Breaker Pattern Use in Web Development.....	13
3.2.3.	Conclusion.....	14
3.	References.....	15
4.	Appendix	17
4.1.	RegnumauctionApplication.java	17
4.2.	GenerateRegNums.java.....	17
4.3.	OffensiveChecker.java.....	18
4.4.	OffensiveInterface.java	19
4.5.	RegnumInterface.java	19
4.6.	RegnumList.java	20
4.7.	SetRegnumVariables.java.....	21
4.8.	Batch.java	22
4.9.	Regnum.java	23
4.10.	Users.java.....	25
4.11.	BatchRepository.java.....	26
4.12.	RegnumRepository.java.....	26
4.13.	UsersRepository.java.....	27
4.14.	RepositoryConfig.java.....	27
4.15.	InsertRegnums.java	27

4.16.	index.html.....	29
4.17.	indexJS.js.....	31
4.18.	SQL Commands	35
4.19.	Application.properties.....	36

1. Introduction

This report will discuss and analyse the best practices for RESTful web service implementation in section 2, while Section 3.1 critically analyses Serverless technologies and section 3.2 discusses the circuit breaker pattern in web development. Any comment about the implementation section of this assignment refers to the Java, HTML, JavaScript and SQL code provided in the appendix of this report.

2. Best Practices for RESTful Web Service Implementation

2.1. Introduction

As with all implementations of programs and services there are certain practices that all developers should follow to ensure they receive the most value as possible from their work. This report will detail some of the best practices when implementing a RESTful web service and to illustrate an implementation of these practices, code snippets will be shown and discussed.

Before the introduction of REST, web services were written under the principles of the Simple Object Access Protocol (SOAP) [1]. SOAP is a protocol that facilitates communication over the internet via HTTP that sends and reads information contained in XML files that are sent as part of a request [2]. As any function calls are described in the XML file, SOAP requests can be encoded therefore increasing security. However, these XML files can become unreadable and complicated to write, which can be a detriment for users. Furthermore, SOAP operations only utilise the HTTP POST function, therefore reducing the flexibility of requests [1].

RESTful web services were introduced to utilise more HTTP requests such as GET, PUT, PATCH and DELETE and will be discussed in more detail below.

2.2. What is REST?

Representational State Transfer (REST) was introduced by Roy Fielding [3] and is a web architecture that is built around hypermedia. There are a number of constraints that, when applied to web services, describe REST.

One of the constraints in REST is the client-server constraint that introduces a separation of concerns between the user and the server [3]. This separation of concerns allows developers to split the development of the front-end web pages and the back-end services so that they can change independently [3].

Another constraint needed for REST is that the service must be stateless. Stateless means that each request from the user to the server must hold all the information needed to process the request and does not require additional information from the server [3]. These practices, and more are discussed henceforth.

2.3. Best Practices for RESTful Web Services

2.3.1. Richardson Maturity Model and HATEOAS

To describe REST and how it evolved from basic and SOAP based interactions, Leonard Richardson introduced a maturity model with different levels each describing extra functionality [4]. The higher the level, the greater the functionality; with SOAP sitting at level zero and REST sitting at level three. Figure 1 illustrates the Richardson maturity model.

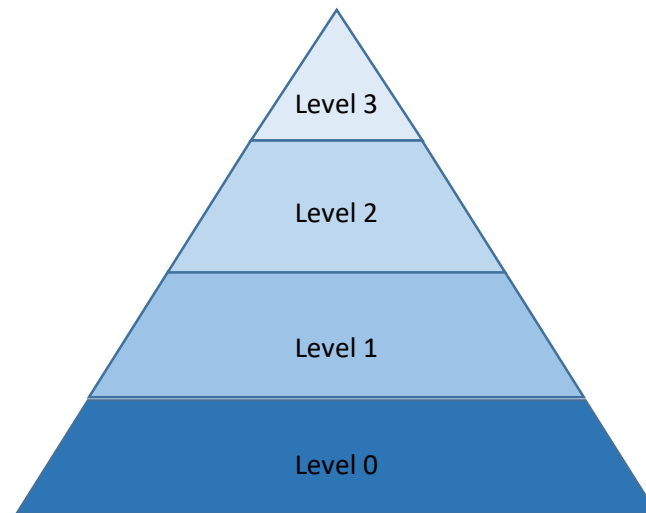


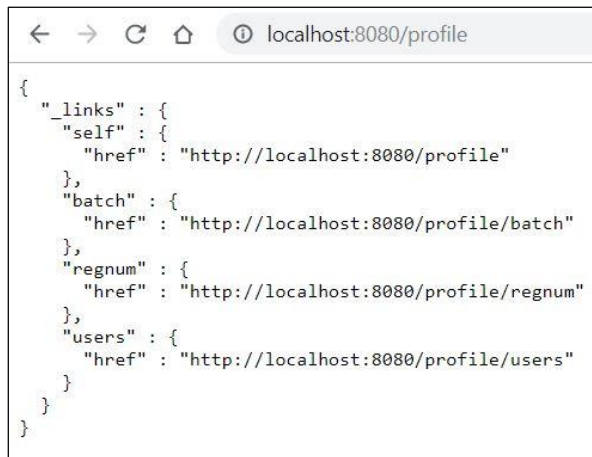
Figure 1. Richardson Maturity Model

Interactions at level zero are the most basic and only utilise the HTTP verb POST to send requests to a single URI [4]. SOAP and other basic requests sit at this level and they contain Plain Old XML (POX) with details of the requests [5]. At this level each request is only ever a POST and are only used to gain access to the service rather than to communicate effectively. This is a detriment because POST requests can modify, create and, in this instance, retrieve information so by only using POST requests it can be difficult to ascertain what use the request is intended for [3], [6]. Only utilising a single URI is bad practice as users must understand that they are required to send requests with different details to the same location, therefore reducing usability [4], [7].

At level one, resources are introduced and as such different URIs are used to perform different tasks [4]. Requests at this level continue to only use POST requests, however, rather than sending an XML file with details of the request, any arguments are passed in through the URI. Level one is more mature than level zero as it splits one large, complex system into multiple smaller functions, therefore improving usability [2], [5].

Level two introduces other HTTP verbs such as GET, PUT, PATCH and DELETE while continuing to use POST [4]. At level zero and level one the POST request was used to retrieve, modify and insert data and the only way to know which method was being used was to examine the XML file. At level two, however, the POST request is used to insert data, while PUT and PATCH are used to modify data, GET is used to retrieve data and DELETE is used to remove data [4], [5]. This separation of functions simplifies the process and improves usability [4]. Level two is close to the full functionality of REST however it relies on the user to understand which methods to use and on which URIs.

Level three describes how hypermedia can be used to provide the user with as much information as possible when using a service [4]. This is achieved through a format called Hypertext As The Engine Of Application State (HATEOAS) which provides links to other resources within the service that could potentially be of interest to the user [4], [6]. Using HATEOAS to inform the user of other resources improves usability and encourages the user to return to the service in the future. Figures 2 through 9 below show different HATEOAS responses from the implementation section of this assignment.

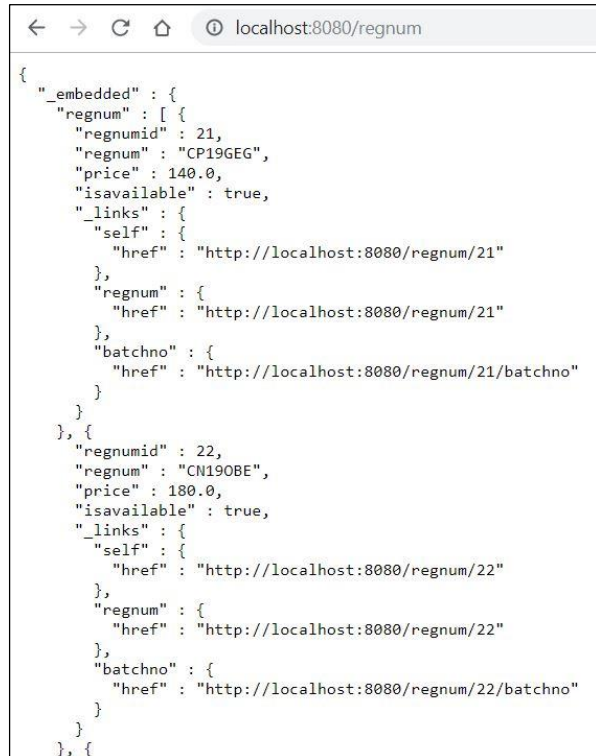


```

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/profile"
    },
    "batch": {
      "href": "http://localhost:8080/profile/batch"
    },
    "regnum": {
      "href": "http://localhost:8080/profile/regnum"
    },
    "users": {
      "href": "http://localhost:8080/profile/users"
    }
  }
}

```

Figure 2. HATEOAS showing other available resources.



```

{
  "_embedded": {
    "regnum": [ {
      "regnumid": 21,
      "regnum": "CP19GEG",
      "price": 140.0,
      "isavailable": true,
      "_links": {
        "self": {
          "href": "http://localhost:8080/regnum/21"
        },
        "regnum": {
          "href": "http://localhost:8080/regnum/21"
        },
        "batchno": {
          "href": "http://localhost:8080/regnum/21/batchno"
        }
      }
    }, {
      "regnumid": 22,
      "regnum": "CN19OBE",
      "price": 180.0,
      "isavailable": true,
      "_links": {
        "self": {
          "href": "http://localhost:8080/regnum/22"
        },
        "regnum": {
          "href": "http://localhost:8080/regnum/22"
        },
        "batchno": {
          "href": "http://localhost:8080/regnum/22/batchno"
        }
      }
    }
  ]
}

```

Figure 3. HATEOAS showing all records in regnum table.

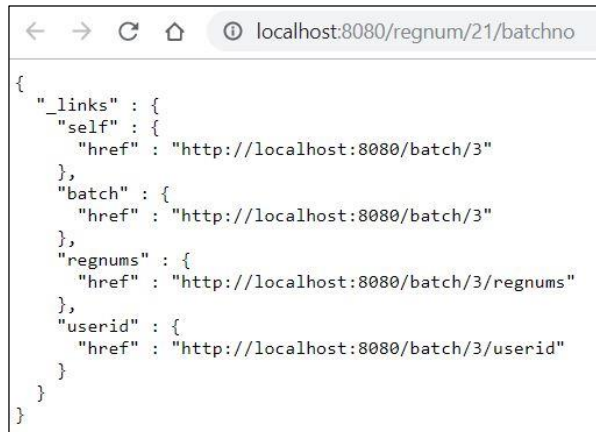


```

} ]
},
{
  "_links": {
    "first": {
      "href": "http://localhost:8080/regnum?page=0&size=20"
    },
    "self": {
      "href": "http://localhost:8080/regnum?page, size, sort",
      "templated": true
    },
    "next": {
      "href": "http://localhost:8080/regnum?page=1&size=20"
    },
    "last": {
      "href": "http://localhost:8080/regnum?page=62&size=20"
    },
    "profile": {
      "href": "http://localhost:8080/profile/regnum"
    }
  },
  "page": {
    "size": 20,
    "totalElements": 1254,
    "totalPages": 63,
    "number": 0
  }
}
}

```

Figure 2. HATEOAS showing extra links available on /regnums endpoint.



```

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/batch/3"
    },
    "batch": {
      "href": "http://localhost:8080/batch/3"
    },
    "regnums": {
      "href": "http://localhost:8080/batch/3/regnums"
    },
    "userid": {
      "href": "http://localhost:8080/batch/3/userid"
    }
  }
}

```

Figure 5. HATEOAS showing batch 3.

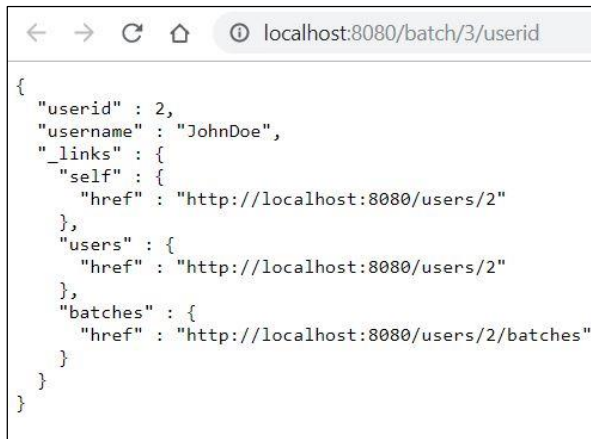


Figure 6. HATEOAS showing user linked to batch 3.

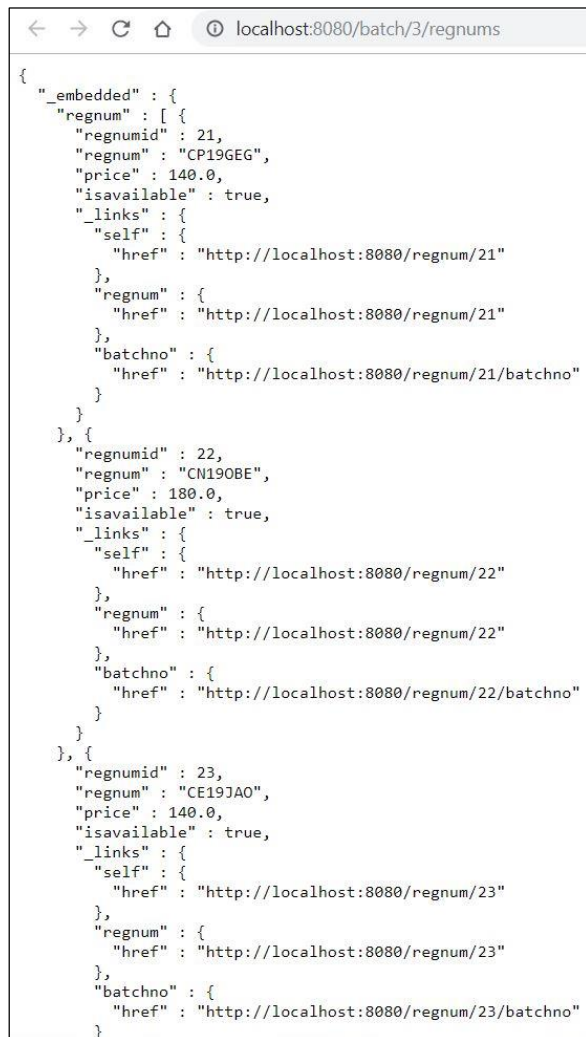


Figure 8. HATEOAS showing regnums linked to batch 3.

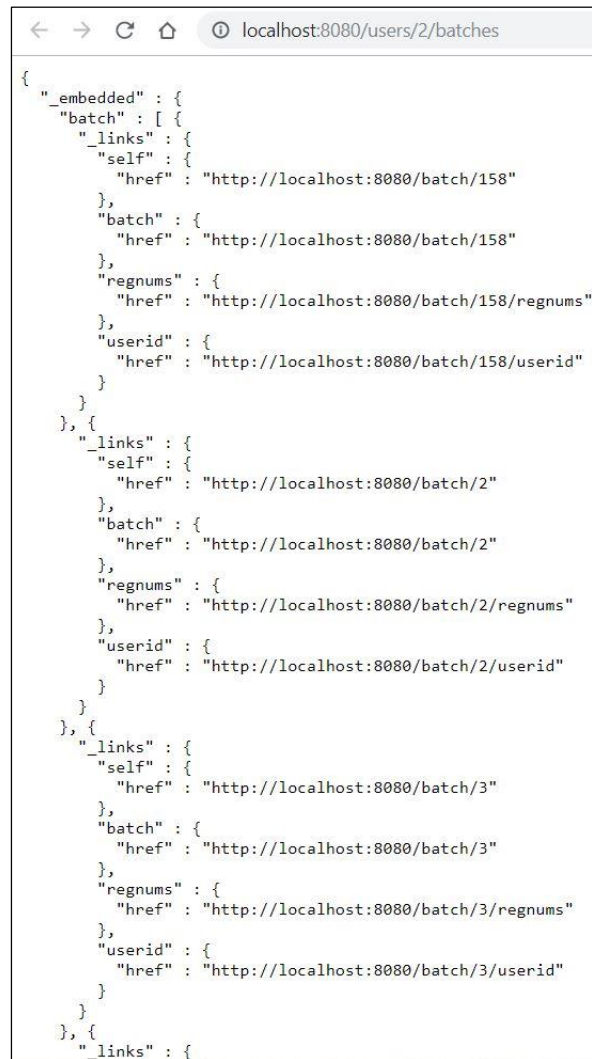


Figure 7. HATEOAS showing batches linked to user 2.

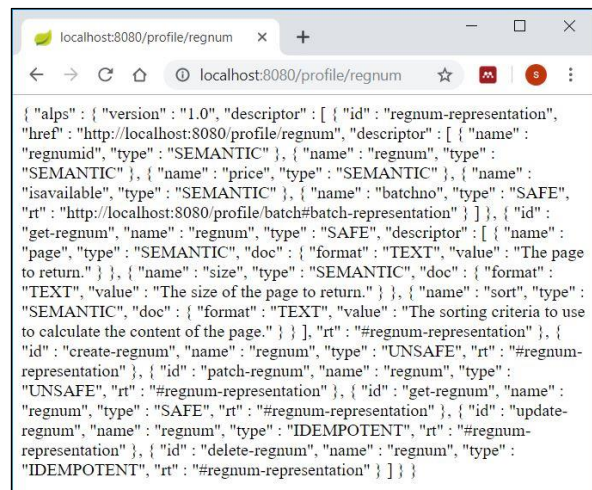
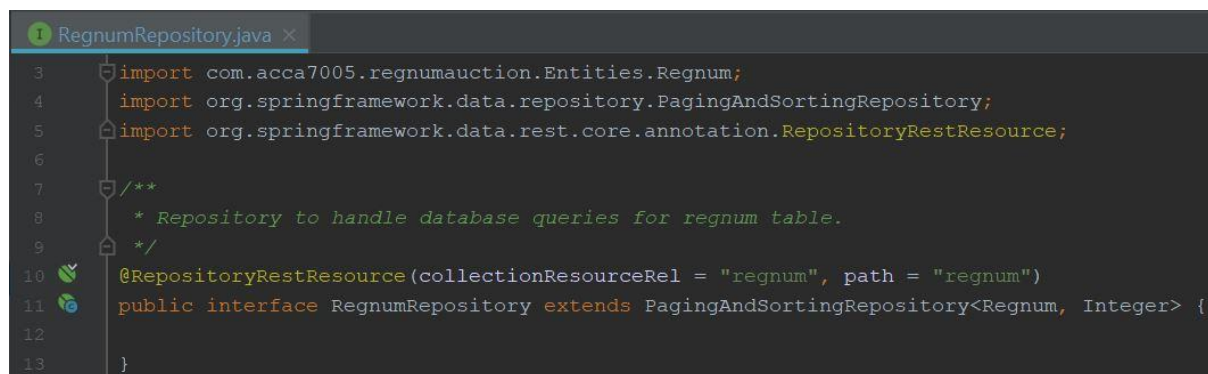


Figure 9. JSON response of regnum profile.

As the implementation was written using Spring, some HATEOAS responses are specifically formed through the Spring framework, such as the profile links as shown in Figure 2. The profile informs the user what resources are available and navigating to a profile URI shows the user key information about that resource, as seen in Figure 9 with the URI `/profile/regnums`. To see the regnum resource itself, the user simply removes the `/profile` from the URI. Figure 3 shows the HATEOAS response for all

records in the regnum table which includes the information stored in the database, such as regnumid and price, but also provides links for the user to follow. In this example, one link directs the user to a HATEOAS response that shows which batch that regnum is relating to (Figure 5), while the “self” link will show all details of that individual regnum record. Figure 4 shows how extra information is presented to the user including how many records are shown on the current page and the total number of elements in that resource. Figure 4 also shows links that are available to the user that allow the user to change how the information is displayed in the browser, including how many items should be displayed. This paging and sorting information is made available thanks to the PagingAndSortingRepository class used in the repository classes of this implementation as seen in Figure 10 below.

Figures 5 through 8 show how a user can navigate a web service when provided with URIs, including how the URI used to get to a resource can be different to the “self” URI of that resource, an example of this is seen in Figure 5. Figures 2 through 10 show how the implementation of this assignment adheres to level 3 of the Richardson maturity model, and therefore to one of the best practices of RESTful web service development.



```
1 RegnumRepository.java x
2
3 import com.acca7005.regnumauktion.Entities.Regnum;
4 import org.springframework.data.repository.PagingAndSortingRepository;
5 import org.springframework.data.rest.core.annotation.RepositoryRestResource;
6
7 /**
8  * Repository to handle database queries for regnum table.
9  */
10 @RepositoryRestResource(collectionResourceRel = "regnum", path = "regnum")
11 public interface RegnumRepository extends PagingAndSortingRepository<Regnum, Integer> {
12
13 }
```

Figure 10. RegnumRepository class.

2.3.2. Platform Independence and Stateless

RESTful web services are written in such a way that they are platform independent. Platform independency means the service can be written in any language and the client can be written in a different language and they are able to communicate without any language related issues [5], [6]. To do this, a web service must use a common media type such as XML or JSON and the client should be able to use the same media type. For most REST web services this commonality between the server and the client is achieved through the standard HTTP verbs of GET, POST, PUT, PATCH and DELETE [5].

To further ensure platform independence, RESTful web services require the requests to be stateless. Being stateless means that all information about the request must be stored client-side and there should be no server-side information that is needed in a request [8]. A benefit to stateless requests is that the service can scale easily as the server does not need to store information about each request. Another advantage of statelessness is that, if there is a fault on the server, it can be replaced without the client knowing anything was wrong [6], [8]. A downside to stateless requests is that clients will often have to send repeated requests to see if a change has occurred and this can lead to less control over consistency as the requests could provoke unforeseen responses [3], [5].

The implementation adheres to statelessness as there is nothing stored in the back-end service or database that is required for requests sent by the client, other than field names which are available to the client through HATEOAS.

```
96  /**
97   * Get all the users in the database and display them in the dropdown list.
98   */
99  function showUsers() {
100    let xmlhttp = new XMLHttpRequest(); //Create new Http request object.
101    xmlhttp.onreadystatechange = function () {
102      if (this.readyState === 4 && this.status === 200) {
103        let users = JSON.parse(xmlhttp.responseText); //Convert JSON to JS object.
104        //For each user in the database, add the username to the drop down list.
105        for (let i = 0; i < users._embedded.users.length; i++) {
106          $("#userSelectDefault").after( nodes: "<option value=\"" + users._embedded.users[i].username + "\">"
107            + users._embedded.users[i].username + "</option>");
108        }
109      }
110    };
111    xmlhttp.open( method: "GET", url: url + "/users", async: true);
112    xmlhttp.send();
113  }
```

Figure 11. GET request from JavaScript client.

```
249  function updateWinningUser(winningUser, batchNo) {
250    getUserId(winningUser); //Get the user id from the database.
251    let userid = $("#userId")[0].value;
252
253    let xmlhttp = new XMLHttpRequest(); //Create new Http request object.
254    //Async = false here to allow second request time to finish.
255    //PATCH to only update some fields, not all.
256    xmlhttp.open( method: 'PATCH', url: url + "/batch/" + batchNo, async: false);
257    //Set headers so service will accept the request.
258    xmlhttp.setRequestHeader( name: 'Content-type', value: 'application/json');
259
260    xmlhttp.onreadystatechange = function () {
261      if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
262        console.log(xmlhttp.responseText); //Log the response.
263      }
264    };
265    //e.g. {"userid":"http://localhost:8080/users/1"}
266    let data = "{\"userid\":\"" + url + "/users/" + userid + "\"}";
267    xmlhttp.send(data); //Send the request to the database.
268  }
```

Figure 12. PATCH request from JavaScript client.

Figure 11 shows a simple GET request to the /users URI (line 111) and uses a for loop to serve the usernames in a dropdown list on a HTML page. Figure 12 shows a PATCH request which is used to update a specified record, in this case the PATCH request is sent to the batch URI with a certain batch number identifier. The PATCH request needs to send information to the service that specifies what field should be updated, and the new data for that field (line 266).

Both Figure 11 and Figure 12 adhere to statelessness as neither require information that is stored on the server without first requesting it. The PATCH request in Figure 12 requires the user id from the database, but first retrieves this via the getUserId method (line 250), rather than needing the server to provide information at the same time.

2.3.3. Security

Another of the best practices for the implementation of RESTful web services is security and authentication. To ensure that a web service is secure and cannot be manipulated by unknown sources, only safe HTTP verbs should be exposed for clients, unless they are authorised to use them

[5]. GET is a safe HTTP verb as it will only ever retrieve information, whereas, POST, PUT, PATCH and DELETE verbs are all unsafe as they can either modify, create or delete information [5]. Sometimes, just hiding HTTP verbs is not enough and an entire URI must be placed behind authentication. This is often done because there is information that is accessible via a GET request that unauthorised users are not permitted to view [3], [6].

The implementation for this assignment does not adhere to this principle because the process to apply security restraints was too complicated in the given timeframe. If there was more time, or if this was to be implemented again in the future, a login system would be introduced to ensure that only authorised users could place bids during an auction.

3. Serverless Technologies and Circuit Breaker Pattern

3.1. Serverless Technologies

3.1.1. Introduction and Background

Before the introduction of Serverless and cloud computing, and the prevalent use of them across the world, companies would have to pay for and maintain a physical server to house databases and facilitate business transactions [9]. A company's servers would have to be located in a building, typically the main office building, and would require a dedicated team to provide maintenance and support for day to day interactions.

A common problem with physical servers is scaling. Scaling is the method of adding or removing servers or capacity depending on the needs of the company. Typical scaling methods include scale-up and scale-down which allow for servers to have their capacity increased or decreased respectively [10]. Companies will often scale-up when they are preparing for a peak in traffic on their server, and will scale-down accordingly after the peak has passed. A more complex method of scaling is known as scaling out, which is where more physical servers are added rather than an increase in capacity [11]. This is a complex process as data must be shared across multiple servers and they must have constant communication to preserve the integrity of the data [10], [11].

The difficulties scaling can impose on a company can include: prolonged maintenance where the servers are offline, potential loss of data when the updated servers are switched on and the cost to scale can be much greater than the return of power or storage [12].

The problem with all hardware is that it has physical limitations and will always, eventually fail [13]. The deterioration of hardware is a substantial issue for companies that use on site servers as implementing replacements is costly and can require the servers to be shut down for prolonged periods of time [14].

The issues with scaling and physical limitations are shortcomings that all companies will face and, without a different solution, production and general workflow will be interrupted. Thus, the Serverless idea was proposed.

3.1.2. What is Serverless?

Serverless technology was introduced to minimise a number of the issues mentioned above and the frustrations they caused. Serverless is the concept of purchasing server capacity from a third party to host web services and databases. Using a third party to host a company's services allows for easy scaling where the company can simply request more or less space and pay accordingly [14]. There are many Serverless hosting companies with whom developers can deploy their code such as Microsoft Azure [15], Google Cloud Platform (GCP) [16] and Amazon Web Services (AWS) [17].

An example of fast and efficient scaling is the Amazon Aurora service which is a relational database engine hosted by Amazon [18]. Companies can use Aurora to hold their databases on Amazon's servers and the database capacity will increase and decrease automatically based on required resources [19]. An advantage to Serverless scaling is that users will be charged based on the amount of time the database is in use and how much capacity is used. Whereas with physical servers, if a company scales-up then they will still have to pay for the larger server capacity even if there is less traffic and resources being used [20].

Serverless computing provides developers with an environment within which they can deploy their code and host services [14]. One way of doing this is known as Functions as a Services (FaaS), which is where developers deploy individual functions to the Serverless platform and define events which

cause the functions to run [21]. The FaaS service (AWS, GCP etc.) constantly checks for the defined events and calls the functions when necessary. This reduces computational requirements as the functions do not have to run indefinitely and developers will only be charged when the functions are called [14].

Serverless FaaS services are used in a multitude of ways with a wide range of complexity. Some functions can be as simple as image rescaling, while others are collections of functions that are all called after a specific event and pass information to and from one another [21], [22]. This variety is not only prevalent in the complexity but also in the types of applications that are deployed in Serverless. Serverless applications can range from simple web sites with a limited number of users to hosting web services of major companies such as the video streaming service Netflix [22].

Per quarter, Netflix provides almost seven billion hours of video to nearly 50 million users and handles extremely large amounts of data [22]. To achieve this, Netflix uses AWS to host their website and databases allowing them to focus on customer interaction in the knowledge that the service will remain online for as long as the AWS servers remain functional [22], [23].

While Serverless computing is a growing technology and is being adopted more and more by companies around the world, there are still many drawbacks that anyone debating Serverless should consider.

3.1.3. Drawbacks of Serverless

A particularly large problem of Serverless technology is the possibility of rewriting large amounts of code to be deployed in a FaaS environment. To allow a service to maintain functions and manage event triggers, functions must be written in a specific way. This can be disastrous for a company who want to deploy a pre-existing application in a FaaS Serverless environment, simply due to the time it would take to rewrite an entire application [21].

Another potential problem to Serverless technology is the fact that after a certain length of time functions that have been triggered are shutdown [21]. This can be a double-edged sword as while it can reduce costs, due to the function being inactive, it must first be written in a way that can handle the scenario where the previous state cannot be recovered [21], [22]. This can be an overly complicated issue for developers since this is not a problem a physical server would encounter as the functions could remain active indefinitely.

A prevailing issue with developers when choosing which Serverless provider to deploy their code with is known as vendor lock. Vendor lock is the concept of being so reliant on one provider, such as AWS, that it is infeasible to move to another provider [21]. The infeasibility of changing providers is due to the specific resources used and the nature in which functions must be written [21], [24].

3.1.4. Serverless Conclusion

Serverless technology is the process by which a company purchases computational space from a third party, to host web services and databases, therefore removing the need for a physical server. Serverless technologies allow for dynamic scaling of services and databases which is cheaper and much less complicated than scaling a physical system.

Serverless is not without its drawback however; the need to rewrite code is a very real issue for developers that want to deploy an already functioning application. In addition to this, vendor locking is a concern for all companies and developers that want to use Serverless. Due to specific resources and requirements, it is can be a very complicated and difficult task to move from one Serverless provider to another.

With respect to web hosting, Serverless technologies provide an easy and cheap option for developers to host web services and websites as functions deployed in Serverless are often event driven. This allows developers to deploy their web service's functions to a Serverless environment and not ever have to concern themselves with the resource management and upkeep of a typical physical server.

3.2. Circuit Breaker Pattern

3.2.1. Introduction

In modern web development it is common for a website, or a service behind a website, to make frequent calls and requests to external sources. When connecting to an external source, there is a measure of uncontrollability that must be accounted for and a reaction must be appointed in the event of the external source failing. A failure could include a network connection error, maintenance from the external source's provider or any number of unexpected scenarios [25].

Web developers must prepare their systems to handle failed requests to external sources and one such way is the retry method. The retry method is as simple as it sounds, the system sends the exact same request to the same external source at the earliest possible instance [25]. A problem with this method is that if an external source is unavailable for an extended period of time, the service will continue to send the request to no avail. If a system does not have a function in place to stop the constant retries, a manual reset is needed to allow the system to continue and to accept the external source is unavailable [26].

Constantly adjusting the system manually to accept that an external source may not respond can be a detriment to developers. To combat this, the circuit breaker design pattern was introduced.

3.2.2. Circuit Breaker Pattern Use in Web Development

Presented by Michael Nygard in his book "Release It! Design and Deploy Production-Ready Software" [27], the circuit breaker pattern is a technique used to prevent a system from continually calling an external source that has failed and is not responding [27]. In the context of electrical components, a circuit breaker is used to detect an overuse of power. Once detected the breaker allows the system to fail, opens the circuit to prevent further failures and then re-closes the circuit once the problem has passed [27].

The principles of the electrical component can be applied to web development and is implemented by placing code that could encounter a failure inside a function that prevents the call being sent when the system is not responding [28]. This strategy is preferable to retries as it stops the calls from occurring, therefore preventing harmful or unexpected responses.

There are three states in which the circuit breaker pattern can exist: closed, open and half-open [27]. In the closed state, the system behaves as normal, sending requests internally and externally. If the call is successful, then no change occurs and the system proceeds as intended. If there is a failure in the system, the circuit breaker keeps track of the number of failures by incrementing the total each time a failure is encountered [27]. This pattern allows the system to continue if a fail occurs, however once the total number of failures has exceeded a maximum allowance, the circuit breaker trips and enters the open state. The pattern can also enter the open state when a specified failure is encountered [29].

While in the open state, any failed call is immediately caught and an exception is thrown, usually in the form of a message to the user detailing the fault [29]. The system remains in the open state for a predetermined amount of time, after which the system is allowed to send one additional request. If the request is successful, the system can progress as normal and the circuit breaker returns to the closed state. Though should the call fail, the circuit breaker returns to the open state [27].

This cyclical process provides developers with a comfortable way to log faults therefore enabling them to implement more exception handlers, making the system more fault-tolerant [27], [30]. The circuit

breaker pattern also makes a system more testable as developers can plan for a failure and then testers can force the circuit breaker to trip to ensure the correct outcome is observed [25]–[27].

A downside to circuit breakers is that they can cause a system's performance to suffer if under stress. Meaning that if there is a large amount of failures being encountered then the circuit breaker will prevent the system from moving forward [26], [27], [31]. This potential drawback should be considered against the business value the pattern provides prior to implementation.

3.2.3. Conclusion

The circuit breaker pattern is used heavily in web development as it is common for websites and services to communicate with external sources. The pattern prevents unexpected errors from external sources from causing the website or service to crash.

The pattern begins in the closed state and allows for a number of failures, though when that number exceeds a certain threshold the breaker trips and enters the open state. While in the open state the system cannot send a request to a failed source. After a given time, the circuit breaker will move into the half-open state where one call to the previously failed source is permitted. If that call succeeds, the breaker returns to the closed state, while if the call fails, the breaker returns to the open state.

The circuit breaker pattern provides developers with a comprehensive way to log information of a system and provides strong testability. However, a consideration must be made between the drop in performance encountered when the system is under stress and the business value the pattern can deliver.

3. References

- [1] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson, "Developing Web Services Choreography Standards - the Case of REST vs. SOAP," *Decis. Support Syst.*, vol. 40, no. 1, pp. 9–29, Jul. 2005.
- [2] I. Salvadori and F. Siqueira, "A Maturity Model for Semantic RESTful Web APIs," in *2015 IEEE International Conference on Web Services*, 2015, pp. 703–710.
- [3] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, Irvine, 2000.
- [4] L. Richardson, "Justice Will Take Us Millions Of Intricate Moves," in *QCon*, 2008.
- [5] B. Mehta, *RESTful Java Patterns and Best Practices: Learn Best Practices to Efficiently Build Scalable, Reliable, and Maintainable High Performance RESTful Services*. Packt Publishing Ltd, 2014.
- [6] S. Parastatidis, J. Webber, and I. Robinson, *REST in Practice*, First. O'Reilly Media, 2010.
- [7] M. Fowler, "Richardson Maturity Model," *MartinFowler.com*, 2010. [Online]. Available: <https://martinfowler.com/articles/richardsonMaturityModel.html>. [Accessed: 05-May-2019].
- [8] T. Yuan *et al.*, "Formalization and Verification of REST on HTTP Using CSP," *Electron. Notes Theor. Comput. Sci.*, vol. 309, pp. 75–93, Dec. 2014.
- [9] P. Wayner, "Serverless in the cloud: AWS vs. Google Cloud vs. Microsoft Azure," 2018.
- [10] M. Hayes, B. Francis, and M. J. Garvey, "Servers Scale Up," *InformationWeek*, no. 663, pp. 77–81, 1998.
- [11] D. Yahalom, "Is Serverless Computing a Game-Changer for the Relational Database Market?," San Mateo, 2018.
- [12] A. Khare *et al.*, *A Fresh Graduate's Guide to Software Development Tools and Technologies: Chapter 6 Scalability*, 2nd ed. 2012.
- [13] P. W. Smith, "On the Physical Limits of Digital Optical Switching and Logic Elements," *Bell Syst. Tech. J.*, vol. 61, no. 8, pp. 1975–1993, Oct. 1982.
- [14] J. Chapin and M. Roberts, *What Is Serverless?* O'Reilly Media, Inc, 2017.
- [15] Microsoft, "Microsoft Azure Cloud Computing Platform & Services." [Online]. Available: <https://azure.microsoft.com/en-gb/>. [Accessed: 29-Apr-2019].
- [16] Google, "Serverless Computing | Google Cloud." [Online]. Available: <https://cloud.google.com/serverless/>. [Accessed: 29-Apr-2019].
- [17] Amazon, "Amazon Web Services (AWS) - Cloud Computing Services." [Online]. Available: <https://aws.amazon.com/>. [Accessed: 29-Apr-2019].
- [18] Amazon.com, "What Is Amazon Aurora? - Amazon Aurora," 2019. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html. [Accessed: 27-Apr-2019].
- [19] "AWS Launches of Amazon Aurora Serverless," *Worldw. Databases*, vol. 30, no. 9, 2018.
- [20] Amazon.com, "How Aurora Serverless Works - Amazon Aurora," 2019. [Online]. Available: <https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-how-it-works.html>. [Accessed: 27-Apr-2019].

- [21] J. M. Hellerstein *et al.*, "Serverless Computing: One Step Forward, Two Steps Back," 2018.
- [22] M. Sewak and S. Singh, "Winning in the Era of Serverless Computing and Function as a Service," in *2018 3rd International Conference for Convergence in Technology (I2CT)*, 2018, pp. 1–5.
- [23] Amazon.com, "Netflix & AWS Lambda Case Study - Amazon Web Services (AWS)." [Online]. Available: <https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/>. [Accessed: 30-Apr-2019].
- [24] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A Mixed-Method Empirical Study of Function-as-a-Service Software Development in Industrial Practice," *J. Syst. Softw.*, vol. 149, pp. 340–359, 2018.
- [25] Microsoft Docs, "Circuit Breaker pattern - Cloud Design Patterns," *docs.microsoft.com*, 2017. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker#solution>. [Accessed: 02-May-2019].
- [26] M. Fowler, "CircuitBreaker," *martinfowler.com*, 2014. [Online]. Available: <https://martinfowler.com/bliki/CircuitBreaker.html>. [Accessed: 02-May-2019].
- [27] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Raleigh, North Carolina, 2007.
- [28] D. Preuveneers and W. Joosen, "Access Control with Delegated Authorization Policy Evaluation for Data-Driven Microservice Workflows," *Futur. Internet*, vol. 9, no. 4, pp. 58–80, 2017.
- [29] F. Montesi and J. Weber, "Circuit Breakers, Discovery, and API Gateways in Microservices," Odense M, Denmark, 2016.
- [30] D. Preuveneers and W. Joosen, "QoC² Breaker: Intelligent Software Circuit Breakers for Fault-Tolerant Distributed Context-Aware Applications," *J. Reliab. Intell. Environ.*, vol. 3, no. 1, pp. 5–20, Jul. 2017.
- [31] H. Subramanian and P. Raj, *Hands-on RESTful Web API Design Patterns and Best Practices: Design, Develop, and Deploy Highly Adaptable, Scalable, and Secure RESTful Web APIs*, First. Birmingham, 2019.

4. Appendix

In this appendix lies all the code written in Java, HTML, JavaScript and SQL used for the implementation section of this assignment.

4.1. RegnumauctionApplication.java

```
package com.acca7005.regnumauction;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RegnumauctionApplication {

    public static void main(String[] args) {
        SpringApplication.run(RegnumauctionApplication.class, args);
    }

}
```

4.2. GenerateRegNums.java

```
package com.acca7005.regnumauction.regnums;

import java.util.Random;

/**
 * Class to generate registration numbers.
 */
public class GenerateRegNums implements RegNumInterface {
    private char[] regnum;

    /**
     * Default constructor for GenerateRegnum class.
     * Initialise regnum and set defaults.
     */
    public GenerateRegNums() {
        regnum = new char[7];
        defaultRegnum();
        generateRegion();
        generateRandomSequence();
    }

    /**
     * Set default values of regnum (C*19**).
     */
    private void defaultRegnum() {
        regnum[0] = 'C';
        regnum[2] = '1';
        regnum[3] = '9';
    }

    /**
     * Select a random letter from the list of available region letters.
     */
    @Override
    public void generateRegion() {
        boolean inBounds = false;
        while (!inBounds) {
            int rand = RandomNumber(REGION_LETTERS.length);
            try {
                regnum[1] = REGION_LETTERS[rand];
                inBounds = true;
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("REGION OUT OF BOUNDS: " + rand);
                inBounds = false;
            }
        }
    }

    /**
     * Select a random letter from the list of available random sequence letters.
     */
}
```

```

@Override
public void generateRandomSequence() {
    for (int i = 0; i < 3; i++) {
        boolean inBounds = false;
        while (!inBounds) {
            int rand = RandomNumber(RANDOM_LETTERS.length);
            try {
                regnum[i + 4] = RANDOM_LETTERS[rand];
                inBounds = true;
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.printf("RANDOM %d OUT OF BOUNDS: %d\n", i, rand);
                inBounds = false;
            }
        }
    }
}

/**
 * Generate a random number utilising the Random class between zero and a maximum.
 *
 * @param max The maximum limit to generate numbers between.
 * @return A random number between 0 and max.
 */
private int RandomNumber(int max) {
    return new Random().nextInt(max);
}

/**
 * Convert the char[] regnum into a string using StringBuilder.
 *
 * @return A String version of regnum.
 */
@Override
public String toString() {
    StringBuilder stringBuilder = new StringBuilder();
    for (char ch : regnum) {
        stringBuilder.append(ch);
    }
    return stringBuilder.toString();
}
}

```

4.3. OffensiveChecker.java

```

package com.acca7005.regnumauction.regnums;

/**
 * Class to check if a regnum is offensive.
 */
public class OffensiveChecker implements OffensiveInterface {

    /**
     * Default constructor.
     */
    public OffensiveChecker() {

    }

    /**
     * Method to check if the three letter sequence of a regnum is offensive.
     * Calls splitRegnum() to split the regnum to easily obtain three letter sequence.
     * Loops through OFFENSIVE_LIST array in interface and checks the three letter sequence
     against each
     * string to see if there is a match.
     *
     * @param regnum The regnum to check.
     * @return True if a regnum is offensive, false if not.
     */
    @Override
    public boolean isOffensive(String regnum) {
        String[] splitRegnum = splitRegnum(regnum);
        //For each offensive word in the array, check if it matches the random sequence
        letters.
        for (String offensiveWord : OFFENSIVE_LIST) {
            if (splitRegnum[1].equals(offensiveWord)) {

```

```

        System.out.println(regnum + " is offensive.");
        return true;
    }
    return false;
}

/**
 * Split the regnum at the digits into two parts: first part = C*19, second part = ***.
 *
 * @param regnum String regnum to split.
 * @return String array of length two containing both parts of the regnum.
 */
String[] splitRegnum(String regnum) {
    return regnum.split("\\d\\d");
}
}

```

4.4. OffensiveInterface.java

```

package com.acca7005.regnumauction.regnums;

/**
 * Interface to define offensive words.
 */
public interface OffensiveInterface {
    /**
     * Final array of offensive three letter combinations.
     * I and Q are not included in the three letter sequence, therefore not included here.
     */
    String[] OFFENSIVE_LIST = {"FUC", "FUK", "FCK", "SHT", "CUM", "VAG", "VAJ", "NZY", "NGA",
        "NGG", "NOG",
        "BUT", "CUM", "BUM", "JEW", "CNT", "CUN", "PIS", "PSS", "TWT", "TWA", "GOD"};

    /**
     * Method to be implemented that checks if a regnum is offensive.
     *
     * @param regnum String to check.
     * @return True if offensive, false if not.
     */
    boolean isOffensive(String regnum);
}

```

4.5. RegnumInterface.java

```

package com.acca7005.regnumauction.regnums;

public interface RegNumInterface {
    /**
     * Final array of individual characters to be used when generating region.
     * I, Q and Z are not allowed in the region tag, therefore not included here.
     */
    char[] REGION_LETTERS = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K', 'L', 'M',
        'N', 'O', 'P', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y'};

    /**
     * Final array of individual characters to be used when generating random three letter
     sequence.
     * I and Q are not allowed in the three letter sequence, therefore not included here.
     */
    char[] RANDOM_LETTERS = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K', 'L', 'M',
        'N', 'O', 'P', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};

    /**
     * Method to be implemented that generates the region tag.
     */
    void generateRegion();

    /**
     * Method to be implemented that generates the random sequence.
     */
    void generateRandomSequence();
}

```

4.6. RegnumList.java

```
package com.acca7005.regnumauction.regnums;

import com.acca7005.regnumauction.Entities.Regnum;

import java.util.ArrayList;
import java.util.List;

/**
 * Class for a list of regnums following the singleton format.
 */
public class RegnumList {
    /**
     * List of regnums.
     */
    private List<Regnum> regnumList;

    /**
     * Instance of regnumList.
     */
    private static RegnumList inst = null;

    /**
     * Private default constructor to follow singleton format.
     */
    private RegnumList() {
        regnumList = new ArrayList<>();
    }

    /**
     * If there is not an instance of of this class then create one, if there is then return
    it.
     *
     * @return The instance of this class to ensure there is only one.
     */
    public static RegnumList getInstance() {
        if (inst == null) {
            inst = new RegnumList();
        }
        return inst;
    }

    /**
     * Add a regnum to the list.
     *
     * @param regnum A regnum to the list.
     */
    public void addRegnum(Regnum regnum) {
        regnumList.add(regnum);
    }

    /**
     * Return the regnum at a certain position in the list.
     *
     * @param id Id of a regnum in the list.
     * @return The regnum at the specified position or null.
     */
    public Regnum getRegnum(int id) {
        if (regnumList.size() <= id || id < 0) {
            return null;
        }
        return regnumList.get(id);
    }

    /**
     * Method to return the full list regnumList.
     *
     * @return Return the full list of regnums.
     */
    public List<Regnum> getRegnumList() {
        return regnumList;
    }
}
```

4.7. SetRegnumVariables.java

```
package com.acca7005.regnumauction.regnums;

import com.acca7005.regnumauction.Entities.Regnum;

/**
 * Class to set a regnums variables.
 */
public class SetRegnumVariables {

    /**
     * Default constructor.
     */
    public SetRegnumVariables() {

    }

    /**
     * Method to set variables of a regnum.
     *
     * @param regnumString Regnum string in form "C*19***" to be stored in regnum.setRegnum.
     * @return Populated regnum object.
     */
    public Regnum setVariables(String regnumString) {
        Regnum regnum = new Regnum();
        regnum.setRegnum(regnumString);
        regnum.setPrice(calculatePrice(regnumString)); //Call calculatePrice to set the price.
        regnum.setIsavailable(true); //Default set to true as is always available until
        purchased.
        return regnum;
    }

    /**
     * Method to assign a price to a regnum dependent on the first letter of the random letter
     sequence.
     *
     * @param regnumString Regnum string of which to check random letters.
     * @return Specified price for regnum.
     */
    private double calculatePrice(String regnumString) {
        //Split the string into "C*" and "***".
        String[] splitRegnum = new OffensiveChecker().splitRegnum(regnumString);
        char[] sequenceSplit = splitRegnum[1].toCharArray(); //Store random letters in a char
        array.
        switch (sequenceSplit[0]) {
            case 'A':
            case 'B':
            case 'C':
                return 100;
            case 'D':
            case 'E':
            case 'F':
                return 120;
            case 'G':
            case 'H':
            case 'J':
                return 140;
            case 'K':
            case 'L':
            case 'M':
                return 160;
            case 'N':
            case 'O':
            case 'P':
                return 180;
            case 'R':
            case 'S':
            case 'T':
                return 200;
            case 'U':
            case 'V':
            case 'W':
                return 220;
            case 'X':
            case 'Y':
            case 'Z':

```

```

        return 240;
    default:
        return 0;
    }
}
}

```

4.8. Batch.java

```

package com.acca7005.regnumauktion.Entities;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

import javax.persistence.*;
import java.util.List;

/**
 * Class to work with BatchRepository to map Batch objects to the database.
 */
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class Batch implements java.io.Serializable {

    /**
     * Primary key for batch table.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(insertable = false)
    private int batchno;

    /**
     * Foreign key from users table.
     */
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "userid")
    private Users userid;

    /**
     * List of regnums assigned to current batch.
     */
    @OneToMany(fetch = FetchType.LAZY, mappedBy = "batchno")
    private List<Regnum> regnums;

    // Getters and Setters.

    /**
     * Get batch number.
     *
     * @return Batch number.
     */
    public int getBatchno() {
        return batchno;
    }

    /**
     * Set batch number.
     *
     * @param batchno New batch number.
     */
    public void setBatchno(int batchno) {
        this.batchno = batchno;
    }

    /**
     * Get id of user.
     *
     * @return Id of user.
     */
    public Users getUserid() {
        return userid;
    }
}

```

```

/**
 * Set new id of user.
 *
 * @param userid New id of user.
 */
public void setUserid(Users userid) {
    this.userid = userid;
}

/**
 * Get list of regnums.
 *
 * @return List of regnums.
 */
public List<Regnum> getRegnums() {
    return regnums;
}

/**
 * Set list of regnums.
 *
 * @param regnums List of regnums.
 */
public void setRegnums(List<Regnum> regnums) {
    this.regnums = regnums;
}
}

```

4.9. Regnum.java

```

package com.acca7005.regnumauction.Entities;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import javax.persistence.*;

/**
 * Class to work with RegnumRepository to map Regnum objects to the database.
 */
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class Regnum implements java.io.Serializable {

    /**
     * Primary key in Regnum table.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int regnumid;

    /**
     * Unique constraint at database level reflected in Column tag.
     */
    @Column(unique = true)
    private String regnum;

    /**
     * Price of regnum to store in database.
     */
    private double price;

    /**
     * Boolean to show if a regnum is available for batch selection or not.
     */
    private boolean isavailable;

    /**
     * Many to one relationship between Regnum and Batch tables.
     */
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "batchno")
    private Batch batchno;

    /**
     * Default constructor.
     */
}

```

```

    */
    public Regnum() {

    }

    ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // Getters and Setters.
    ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    /**
     * Get regnumid.
     *
     * @return Id of regnum.
     */
    public int getRegnumid() {
        return regnumid;
    }

    /**
     * Set regnum id.
     *
     * @param regnumid Id of regnum.
     */
    public void setRegnumid(int regnumid) {
        this.regnumid = regnumid;
    }

    /**
     * Get regnum string.
     *
     * @return Regnum string.
     */
    public String getRegnum() {
        return regnum;
    }

    /**
     * Set regnum string.
     *
     * @param regnum String of regnum.
     */
    public void setRegnum(String regnum) {
        this.regnum = regnum;
    }

    /**
     * Get price of regnum.
     *
     * @return Price of regnum.
     */
    public double getPrice() {
        return price;
    }

    /**
     * Set price of regnum.
     *
     * @param price Price of regnum.
     */
    public void setPrice(double price) {
        this.price = price;
    }

    /**
     * Return state of isavailable.
     *
     * @return Current isavailable state.
     */
    public boolean isIsavailable() {
        return isavailable;
    }

    /**
     * Set state for isavailable.
     *
     * @param isavailable State for isavailable.
     */
    */

```

```

    public void setIsavailable(boolean isavailable) {
        this.isavailable = isavailable;
    }

    /**
     * Get batch number of regnum.
     *
     * @return Batch number of regnum.
     */
    public Batch getBatchno() {
        return batchno;
    }

    /**
     * Set batch number of regnum.
     *
     * @param batchno Id of batch number from batch table.
     */
    public void setBatchno(Batch batchno) {
        this.batchno = batchno;
    }
}

```

4.10. Users.java

```

package com.acca7005.regnumauction.Entities;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

import javax.persistence.*;
import java.util.List;

/**
 * Class to work with UsersRepository to map Users objects to the database.
 */
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class Users implements java.io.Serializable {

    /**
     * Primary key of users table.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int userid;

    /**
     * String username to store in database.
     */
    private String username;

    /**
     * List of batches assigned to current user.
     */
    @OneToMany(fetch = FetchType.LAZY, mappedBy = "userid")
    private List<Batch> batches;

    // Getters and Setters.

    /**
     * Get userid.
     *
     * @return Id of user.
     */
    public int getUserid() {
        return userid;
    }

    /**
     * Set the user id.
     *
     * @param userid Id of user.
     */
}

```

```

    public void setUserid(int userid) {
        this.userid = userid;
    }

    /**
     * Get username of user.
     *
     * @return Username of user.
     */
    public String getUsername() {
        return username;
    }

    /**
     * Set username of user.
     *
     * @param username New username for user.
     */
    public void setUsername(String username) {
        this.username = username;
    }

    /**
     * Get list of batches.
     *
     * @return List of batches.
     */
    public List<Batch> getBatches() {
        return batches;
    }

    /**
     * Set batches.
     *
     * @param batches List of batches.
     */
    public void setBatches(List<Batch> batches) {
        this.batches = batches;
    }
}

```

4.11. BatchRepository.java

```

package com.acca7005.regnumauction.Database;

import com.acca7005.regnumauction.Entities.Batch;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

/**
 * Repository to handle database queries for batch table.
 */
@RepositoryRestResource(collectionResourceRel = "batch", path = "batch")
public interface BatchRepository extends PagingAndSortingRepository<Batch, Integer> {
}

```

4.12. RegnumRepository.java

```

package com.acca7005.regnumauction.Database;

import com.acca7005.regnumauction.Entities.Regnum;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

/**
 * Repository to handle database queries for regnum table.
 */
@RepositoryRestResource(collectionResourceRel = "regnum", path = "regnum")
public interface RegnumRepository extends PagingAndSortingRepository<Regnum, Integer> {
}

```

```
}
```

4.13. UsersRepository.java

```
package com.acca7005.regnumauktion.Database;

import com.acca7005.regnumauktion.Entities.Users;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

/**
 * Repository to handle database queries for users table.
 */
@RepositoryRestResource(collectionResourceRel = "users", path = "users")
public interface UsersRepository extends PagingAndSortingRepository<Users, Integer> {

}
```

4.14. RepositoryConfig.java

```
package com.acca7005.regnumauktion.Database;

import com.acca7005.regnumauktion.Entities.Regnum;
import com.acca7005.regnumauktion.Entities.Users;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.rest.core.config.RepositoryRestConfiguration;
import org.springframework.data.rest.webmvc.config.RepositoryRestConfigurerAdapter;

/**
 * Class to expose the IDs of entities which are usually hidden.
 */
@Configuration
public class RepositoryConfig extends RepositoryRestConfigurerAdapter {

    @Override
    public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config) {
        config.exposeIdsFor(Users.class);
        config.exposeIdsFor(Regnum.class);
    }

}
```

4.15. InsertRegnums.java

```
package com.acca7005.regnumauktion.Database;

import com.acca7005.regnumauktion.Entities.Batch;
import com.acca7005.regnumauktion.Entities.Regnum;
import com.acca7005.regnumauktion.regnums.*;
import org.springframework.stereotype.Service;

import javax.transaction.Transactional;

/**
 * Class to generate random regnums and save them as a new batch.
 * Service tag used to ensure this class runs as soon as the application starts.
 *
 * This class only creates one batch when ran.
 * To create multiple batches, the service must be run multiple times.
 *
 * I attempted to create two batches when this class runs rather than just one, however,
 * a number of error occurred such as:
 * - Two batches would be created but 0 regnums would be added to the first and 20 to the
 * second.
 * - Removing the latest added batch of regnums from the regnumList added 10 to the first
 * batch
 * but only 5 to the second and caused NullPointerExceptions.
 *
 * (Second batch attempt on branch feature/change-for-loop-and-remove-elements-from-list)
 */
@Service
public class InsertRegnums {

    /**
```

```

    * Create object of type RegnumRepository to allow access to database.
    */
    private final RegnumRepository regnumRepository;

    /**
     * Create instance of RegnumList class to store generated regnums before entering into
     database.
     */
    private static RegnumList regnumList;

    /**
     * Create an instance of Batch to be used throughout class.
     */
    private Batch batch;

    /**
     * Constructor for InsertRegnums class.
     * Dependency injection of RegnumRepository and BatchRepository to write to database.
     *
     * @param regRepo    RegnumRepository instance.
     * @param batchRepo  BatchRepository instance.
     */
    public InsertRegnums(RegnumRepository regRepo, BatchRepository batchRepo) {
        regnumRepository = regRepo; //Initialise repository instance.
        batch = new Batch(); //Create a new Batch.
        // Save the batch to the database & autoincrement batchno.
        batchRepo.save(batch);
        generateRegnums(10);
        insertIntoDatabase();
    }

    /**
     * Method to generate 10 random regnums and store them in the RegnumList object.
     */
    @Transactional
    public void generateRegnums(int loopCount) {
        regnumList = RegnumList.getInstance(); //Get the instance of the singleton RegnumList
class.
        for (int i = 0; i < loopCount; i++) {
            //Call GenerateRegNums class to generate random regnum.
            GenerateRegNums generateRegNums = new GenerateRegNums();
            String regnumString = generateRegNums.toString(); //Store regnum string.
            //Set new regnum variables.
            Regnum regnum = new SetRegnumVariables().setVariables(regnumString);

            /*
             * Check if regnum is offensive.
             * If it is ignore and create another, if it is not, add it to the list.
             */
            if (new OffensiveChecker().isOffensive(regnumString)) {
                System.out.println("Regnum is offensive. Not added.");
                i--;
            } else {
                regnumList.addRegnum(regnum);
            }
        }
    }

    /**
     * Method to insert data into the database and generates extra regnums if a duplicate is
     found.
     */
    @Transactional
    public void insertIntoDatabase() {
        for (Regnum reg : regnumList.getRegnumList()) {
            try {
                reg.setBatchno(batch); //Set regnum.batchno equal to current batch.
                regnumRepository.save(reg); //Hibernate takes the Regnum object and inserts it
into the database.
            } catch (Exception e) {
                //Try-catch used to stop the application from crashing if a duplicate is
found.

                generateRegnums(1); //Generate one regnum if a duplicate is found.
                e.printStackTrace(); //Print stack trace to see error.
            }
        }
    }

```

```
}  
}
```

4.16. index.html

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <title>Regnum Auction</title>  
  <link rel="stylesheet"  
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"  
  integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"  
crossorigin="anonymous">  
</head>  
<body>  
<div class="container">  
  <h1>REGNUM AUCTION</h1>  
  <!--Row for vertical inputs and displaying batch.-->  
  <div class="row">  
    <div class="col-4">  
      <!--Vertical to allow items to be stacked above each other.-->  
      <div class="vertical">  
        <div id="totalBatchNo" style="padding-bottom: 2%>Number of Batches To Select  
From:  
          <a id="batchTotal"></a>  
        </div>  
        <!--User enters batch number to display.-->  
        <form action="" id="selectBatch" style="padding-bottom: 2%>  
          Enter Batch Number:  
          <label for="batchBox"></label><input id="batchBox" type="number" value="0"  
min="1">  
          <div id="batchButtonDiv">  
            <!--On click, display regnums relating to selected batch.-->  
            <input id="batchButton" type="button" value="Load Batch"  
onclick="showRegnum()">  
          </div>  
        </form>  
        <div id="usernameDropdown">  
          Select User:  
          <label>  
            <!--Populated with users from database.-->  
            <select name="usernameSelect" onchange="setUser(this.value)">  
              <option id="userSelectDefault" value="">Select a User:</option>  
            </select>  
          </label>  
        </div>  
        <!--Display starting bid. User cannot interact with this element.-->  
        <div id="startingBidDiv" style="padding-bottom: 2%;>  
          Starting Bid: £<label for="startPriceBox"></label><input  
id="startPriceBox" type="number"  
value="0">  
disabled  
        </div>  
        <!--Text box for user to enter a bid.-->  
        <div id="bidLabel">  
          <label>Enter Bid: £<input id="bidValue" type="number" name="bid" min="1"  
max="1000000"></label>  
        </div>  
        <!--Submit bid.-->  
        <div id="bidButtonDiv">  
          <input id="bidButton" type="button" value="Place Bid"  
onclick="checkBidIsNumber()">  
        </div>  
        <!--Display highest bid and user who bid it.-->  
        <div id="highestBidDiv">  
          <h3>Winning Bid:</h3>  
          <label for="highestBid"></label>£<input disabled id="highestBid"  
value="0">  
          <h3>Placed By:</h3>  
          <label for="highestBidder"></label> <input disabled id="highestBidder"  
value="">  
        </div>  
        <!--Display countdown timer. Populated by JS.-->  
        <div id="clockDiv">
```

```

        <h3>Time Remaining</h3>
        <div id="showClock"></div>
    </div>
    <!--Populated by JS.-->
    <div id="showWinnerDiv"></div>
    <!--Reload page.-->
    <div id="startNextAuctionButtonDiv">
        <input id="startNextAuctionButton" type="button" value="Start Next
Auction"
        onclick="location.reload()" ">
    </div>
    <!--Hidden div for JS purposes.-->
    <div><label for="userId"></label><input type="text" id="userId" value=""
hidden disabled></div>
    </div>
</div>

<!--Table to display batches in.-->
<div id="batchTable" class="col-6">
    <table class="table table-striped table-bordered">
        <tr id="tableHeadings">
            <th>Regnum</th>
            <th>Price</th>
        </tr>
        <tr id="fields1">
            <td id="regnumField1"></td>
            <td id="priceField1"></td>
        </tr>
        <tr id="fields2">
            <td id="regnumField2"></td>
            <td id="priceField2"></td>
        </tr>
        <tr id="fields3">
            <td id="regnumField3"></td>
            <td id="priceField3"></td>
        </tr>
        <tr id="fields4">
            <td id="regnumField4"></td>
            <td id="priceField4"></td>
        </tr>
        <tr id="fields5">
            <td id="regnumField5"></td>
            <td id="priceField5"></td>
        </tr>
        <tr id="fields6">
            <td id="regnumField6"></td>
            <td id="priceField6"></td>
        </tr>
        <tr id="fields7">
            <td id="regnumField7"></td>
            <td id="priceField7"></td>
        </tr>
        <tr id="fields8">
            <td id="regnumField8"></td>
            <td id="priceField8"></td>
        </tr>
        <tr id="fields9">
            <td id="regnumField9"></td>
            <td id="priceField9"></td>
        </tr>
        <tr id="fields10">
            <td id="regnumField10"></td>
            <td id="priceField10"></td>
        </tr>
    </table>
</div>
</div>
</div>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></script>
<script src="indexJS.js" type="text/javascript" language="JavaScript"></script>
</body>
</html>

```

4.17. indexJS.js

```
/**
 * When first researching how to use JavaScript to send http requests to the web service,
 * I found a tutorial on W3 schools at the below link that used the XMLHttpRequest object.
 * As this was successful, I continued using this approach.
 * Link to tutorial: https://www.w3schools.com/xml/ajax_xmlhttprequest_send.asp
 */

/**
 * Global variable to store the current user for use in all functions.
 *
 * @type {string}
 */
let username = "";

/**
 * Global variable used for specifying the URL in requests.
 *
 * @type {string}
 */
let url = "http://localhost:8080";

/**
 * On page load:
 * - Call showUsers.
 * - Set the highest bid value to -1.
 * - Start countdown timer.
 * - Hide the button to start the next auction.
 * - Call displayNumberOfBatches.
 */
$(function () {
    showUsers();
    $("#highestBid").value = -1;
    startCountdown();
    $("#startNextAuctionButtonDiv")[0].style.visibility = "hidden";
    displayNumberOfBatches();
});

/**
 * Display the total number of batches available for selection.
 */
function displayNumberOfBatches() {
    let totalBatches = 0;
    let xmlHttp = new XMLHttpRequest(); //Create new Http request object.
    xmlHttp.onreadystatechange = function () {
        if (this.readyState === 4 && this.status === 200) {
            let batchObject = JSON.parse(xmlHttp.responseText); //Convert JSON to JS object.
            for (let i = 0; i < batchObject._embedded.batch.length; i++) {
                totalBatches++; //Count number of batches in database.
            }
            $("#batchTotal")[0].innerText = totalBatches; //Display number of batches.
        }
    };
    //Open the http request.
    //"/batch?page=0&size=300" used to ensure ability to count all batches. Unlikely to be
    over 300.
    xmlHttp.open("GET", url + "/batch?page=0&size=300", true);
    xmlHttp.send(); //Send the request to the service.
}

/**
 * Get all the regnums a certain batch and display them in the table.
 */
function showRegnum(batchNo) {
    if (batchNo == null) {
        batchNo = $("#batchBox")[0].value;
    }

    let price = 0, total = 0;
    let xmlHttp = new XMLHttpRequest(); //Create new Http request object.
    xmlHttp.onreadystatechange = function () {
        if (this.readyState === 4 && this.status === 200) {
            let batchObject = JSON.parse(xmlHttp.responseText); //Convert JSON to JS object.
            //Always 10 here as there are always 10 regnums created.
            for (let i = 0; i < 10; i++) {
                //Check if the selected batch is available, show alert if not available.
            }
        }
    };
    xmlHttp.open("GET", url + "/batch/" + batchNo, true);
    xmlHttp.send();
}
```

```

        if (batchObject._embedded.regnum[i].isavailable === false) {
            alert("Batch has already been selected. Please select another.");
            return; //Exit the function.
        }
        price = batchObject._embedded.regnum[i].price;
        //jQuery did not work here so document.getElementById is used.
        document.getElementById("regnumField" + (i + 1)).innerText =
            batchObject._embedded.regnum[i].regnum;

        //jQuery did not work here so document.getElementById is used.
        document.getElementById("priceField" + (i + 1)).innerText = price;
        total += price; //Add each price to the total.
    }
    $("#startPriceBox")[0].value = total / 10; //Display the start price as the
    average of the batch.
}

};
//Open the http request.
xmlHttp.open("GET", url + "/batch/" + batchNo + "/regnums/", true);
xmlHttp.send(); //Send the request to the service.
}

/**
 * Get all the users in the database and display them in the dropdown list.
 */
function showUsers() {
    let xmlHttp = new XMLHttpRequest(); //Create new Http request object.
    xmlHttp.onreadystatechange = function () {
        if (this.readyState === 4 && this.status === 200) {
            let users = JSON.parse(xmlHttp.responseText); //Convert JSON to JS object.
            //For each user in the database, add the username to the drop down list.
            for (let i = 0; i < users._embedded.users.length; i++) {
                $("#userSelectDefault").after("<option value=\"" +
                    users._embedded.users[i].username + "\">"
                    + users._embedded.users[i].username + "</option>");
            }
        }
    };
    xmlHttp.open("GET", url + "/users", true);
    xmlHttp.send();
}

/**
 * Check if the input of messageIDTextBox is a number, if it is not then display an error
message.
 * Not necessarily needed as #bidValue is a number text box, but left in just in case a user
somehow enters a
 * character that is not a number.
 */
function checkBidIsNumber() {
    let bid = $("#bidValue")[0].value;
    if (isNaN(parseInt(bid))) {
        alert("Please enter a number.");
        return;
    }
    checkBidIsGreater(bid);
}

/**
 * Check if the latest bid placed is higher than the previous highest bid.
 *
 * @param bid Latest bid placed by a user.
 */
function checkBidIsGreater(bid) {
    let currentBid = parseInt($("#highestBid")[0].value);
    let startPrice = parseInt($("#startPriceBox")[0].value);
    if (bid <= startPrice) {
        alert("Your bid is not greater than the start bid.");
    } else if (bid > currentBid) {
        setHighestBid(bid);
    } else {
        alert("Your bid is not greater than the current bid.");
    }
}

/**
 * Check if the user has selected a name from the list.

```

```

* If they have, set the new highest bid and the user who made it.
*
* @param bid Latest bid made that is higher than the previous highest bid.
*/
function setHighestBid(bid) {
    if (username === "" || username == null) {
        alert("Please select a username from the list.");
    } else {
        $("#highestBid")[0].value = bid;
        $("#highestBidder")[0].value = username;
    }
}

/**
 * Update global variable with name of the current user.
 *
 * @param user Username from usernameSelect dropdown list.
 */
function setUser(user) {
    username = user;
}

/**
 * This method starts and displays the countdown.
 *
 * To calculate the correct time conversions from minutes and seconds into milliseconds, I
 * used this tutorial:
 * https://www.w3schools.com/howto/howto_js_countdown.asp
 */
function startCountdown() {
    //Set variables for 1: second, minute and hour in milliseconds.
    let oneSecInMilli = 1000;
    let oneMinInMilli = 1000 * 60;
    let oneHourInMilli = 1000 * 60 * 60;

    let countdown = 30; //Time in minutes for countdown.
    //Set deadline to be the current time plus the countdown time.
    //Need to parse a new Date to ensure countdown displays correctly in HTML.
    //Need to multiply by oneSecInMilli to convert time from minutes to milliseconds.
    let deadline = new Date().getTime() + countdown * oneMinInMilli;

    //Update the countdown every second.
    let interval = setInterval(function () {
        let currentTime = new Date().getTime();
        let timeRemaining = deadline - currentTime;
        /*
         * Math.floor will return the largest integer.
         * 1000 (milliseconds) * 60 (seconds) * 60 (minutes) -> Convert one hour into
         milliseconds.
         * 1000 (milliseconds) * 60 (seconds) -> Convert one minute into milliseconds.
         * timeRemaining % oneHourInMilli = Remaining hours in milliseconds
         * timeRemaining % oneMinInMilli = Remaining minutes in milliseconds.
         */

        /*
         * Without dividing by oneMinInMilli for minutes and oneSecInMilli for seconds,
         * time is displayed in milliseconds in HTML.
         */
        let minutes = Math.floor((timeRemaining % oneHourInMilli) / oneMinInMilli);
        let seconds = Math.floor((timeRemaining % oneMinInMilli) / oneSecInMilli);

        //Display time remaining in minutes and seconds.
        $("#showClock")[0].innerHTML = "<h3>" + minutes + "m " + seconds + "s</h3>";

        //When the timer runs out, display the winning user and bid.
        if (timeRemaining <= 0) {
            clearInterval(interval);
            $("#showClock")[0].innerHTML = "<h3>0m 0s</h3>";
            showWinner();
        }
    }, oneSecInMilli); //Interval set for one second.
}

/**
 * At the end of the auction, display the winning user and bid that are stored in
 * #highestBidder and #highestBid.
 * If there is no user in #highestBidder then print state there is no winner.

```

```

*/
function showWinner() {
    let winner = $("#highestBidder")[0].value;
    let bid = $("#highestBid")[0].value;
    let batchNo = $("#batchBox")[0].value;

    /*
    * If no user has been selected when the timer runs out, display No Winner!
    */
    if (winner === "" || winner == null) {
        $("#showWinnerDiv")[0].innerHTML = "<h3>No Winner!</h3>";
    } else {
        $("#showWinnerDiv")[0].innerHTML =
            "<h3>Winner of batch " + batchNo + " is " + winner + "</h3><h3>With a Winning Bid
of £" + bid + "!</h3>";
        updateWinningUser(winner, batchNo); //Update the database.
        getRegnumsForUpdate(batchNo); //Get regnums to update isavailable field.
    }
    //Display the button to start a new auction.
    $("#startNextAuctionButtonDiv")[0].style.visibility = "visible";
}

/**
 * Method to update the database with the user who won a certain batch.
 *
 * @param winningUser User who won the batch and is stored in the #highestBidder text box.
 * @param batchNo Batch number of regnums to change.
 */
function updateWinningUser(winningUser, batchNo) {
    getUserId(winningUser); //Get the user id from the database.
    let userid = $("#userId")[0].value;

    let xmlhttp = new XMLHttpRequest(); //Create new Http request object.
    //Async = false here to allow second request time to finish.
    //PATCH to only update some fields, not all.
    xmlhttp.open('PATCH', url + "/batch/" + batchNo, false);
    //Set headers so service will accept the request.
    xmlhttp.setRequestHeader('Content-type', 'application/json');

    xmlhttp.onreadystatechange = function () {
        if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
            console.log(xmlhttp.responseText); //Log the response.
        }
    };
    //e.g. {"userid":"http://localhost:8080/users/1"}
    let data = "{\"userid\":\"" + url + "/users/" + userid + "\"}";
    xmlhttp.send(data); //Send the request to the database.
}

/**
 * Method to get the userid of the winning user.
 */
function getUserId(winner) {
    let user = -1;
    let xmlhttp = new XMLHttpRequest(); //Create new Http request object.
    xmlhttp.onreadystatechange = function () {
        if (this.readyState === 4 && this.status === 200) {
            //Parse the JSON response to JS object.
            let users = JSON.parse(xmlhttp.responseText); //Convert JSON to JS object.
            for (let i = 0; i < users._embedded.users.length; i++) {
                if (winner === users._embedded.users[i].username) {
                    user = parseInt(users._embedded.users[i].userid); //Store the userid as an
int.

                    //Store the userid in a hidden text box as return was causing issues.
                    $("#userId")[0].value = user;
                    return;
                }
            }
        }
    };
    xmlhttp.open("GET", url + "/users", false);
    xmlhttp.send();
}

/**
 * Method to get the regnums of a certain batch from the database.
 */

```

```

    * @param batchNo Batch number to get regnums from.
    */
function getRegnumsForUpdate(batchNo) {
    let xmlHttp = new XMLHttpRequest(); //Create new Http request object.
    xmlHttp.onreadystatechange = function () {
        if (this.readyState === 4 && this.status === 200) {
            let batchObject = JSON.parse(xmlHttp.responseText); //Convert JSON to JS object.
            for (let i = 0; i < 10; i++) {
                let regnumid = parseInt(batchObject._embedded.regnum[i].regnumid);
                updateRegnumAvailable(regnumid);
            }
        }
    };
    xmlHttp.open("GET", url + "/batch/" + batchNo + "/regnums/", true);
    xmlHttp.send();
}

/**
 * Method to update the isavailable field of regnums in the database.
 *
 * @param regnumid Current regnum to update.
 */
function updateRegnumAvailable(regnumid) {
    let xmlHttp = new XMLHttpRequest(); //Create new Http request object.
    //Async = false here to ensure the update finishes before updating the next.
    xmlHttp.open('PATCH', url + "/regnum/" + regnumid, false);

    xmlHttp.setRequestHeader('Content-type', 'application/json');

    xmlHttp.onreadystatechange = function () {
        if (xmlHttp.readyState === 4 && xmlHttp.status === 200) {
            console.log(xmlHttp.responseText); //Log the response.
        }
    };

    let data = "{\"isavailable\":\"false\"}"; //{ "isavailable": "false" }
    xmlHttp.send(data);
}

```

4.18. SQL Commands

```

CREATE TABLE users
(
    userid int, --Linked to sequence users_userid_seq that starts at 1 and increments by 1.
    username text,

    CONSTRAINT users_userid_pk PRIMARY KEY (userid)
);

CREATE TABLE batch
(
    batchno int, --Linked to sequence batch_batchno_seq that starts at 1 and increments by 1.
    userid int,

    CONSTRAINT batch_batchno_pk PRIMARY KEY (batchno),
    CONSTRAINT batch_userid_fk FOREIGN KEY (userid) REFERENCES users(userid)
);

CREATE TABLE regnum
(
    regnumid int, --Linked to sequence regnum_regnumid_seq that starts at 1 and increments by 1.
    regnum text,
    price numeric,
    isavailable boolean,
    batchno int,

    CONSTRAINT regnum_regnumid_pk PRIMARY KEY (regnumid),
    CONSTRAINT regnum_batchno_fk FOREIGN KEY (batchno) REFERENCES batch(batchno),
    CONSTRAINT regnum_regnum_unique UNIQUE (regnum)
);

```

The SQL commands will create the necessary tables to allow the program to run correctly, providing they are in a database named regnumauktion.

Due to the nature of the implementation, users will have to be inserted into the users table and can be done so with the following INSERT INTO commands.

```
-- Use this insert statement to add users to the database to then be displayed in HTML.  
INSERT INTO users (username)  
VALUES ('EdwardPrasad'),  
        ('JohnDoe');
```

4.19. Application.properties

```
spring.datasource.driver-class-name=org.postgresql.Driver  
spring.datasource.url=jdbc:postgresql://localhost:5432/regnumauktion  
spring.datasource.username=postgres  
spring.datasource.password=  
spring.jpa.database-platform=postgres  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect  
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true  
spring.jpa.show-sql=false  
spring.jpa.open-in-view=false
```

* represents the password for the server that the regnumauktion database is held in.