



BEST PRACTICE IN THE IMPLEMENTATION OF RESTFUL WEB SERVICES

Lee Rodger – 1809707

Abstract

This report outlines and analyses the recommendations in current industry with regards to the development of RESTful web services. Modern day web services rely heavily on these principles and with an ever-evolving industry, it is important that we are able to achieve the best results when using these services.

Contents

1 – Introduction to RESTful Web Services	2
2 – Practices in RESTful WEB Services	3
2.1 – Client-server architecture.....	3
2.2 – Statelessness.....	7
2.3 - Layered Architecture	8
2.4 – Caching	9
2.5 – Code on Demand	10
2.6 – Uniform Interface	10
3 – Conclusion	13
References	14

1 – Introduction to RESTful Web Services

RESTful web services have become an industry standard technology for the development of web applications. It has grown to be considered an easier to use and more powerful platform for developing web services, which has what has made it such a success in the field. Unlike older technologies such as SOAP (Simple Object Access Protocol), REST (Representational State Transfer) offers overall better browser client support due to its collaboration and use of JSON (JavaScript Object Notation).

A RESTful web service sees data and functionality as resources [1] which are accessed using URIs (Uniform Resource Identifiers), a term which is sometimes used interchangeably with URLs but are not necessarily the same thing. A URI however holds more information than just the locator, it can hold the name of a resource also.

2 – Practices in RESTful WEB Services

According to Adamczyk, Smith, Johnson and Hafiz [2], REST principles should be considered guidelines only, as they do not impact how the system works when implemented. However, following the principles will allow for a much easier experience when developing more in depth and complex applications [1] [3], starting with basic principles of object oriented design such as encapsulation and scalability.

According to Giessler and Gebhart [4], there are six key practices which should be adhered to when developing a RESTful web service. These six practices focus on the following areas:

- Client and Server
- Statelessness
- Layered Architecture
- Caching
- Code on Demand
- Uniform Interface

By focusing on these areas and developing a service which follows these closely, the finished product should be that of a robust and extensible web service which would fit well within the modern web.

2.1 – Client-server architecture

The first principle of a RESTful web service is described by Lange [5] as “servers have resources that clients want to use”. Fundamentally this means that the web interface and the server are two completely separate entities which only communicate and do not rely on one another. The server must take care of the back-end (data, rules etc.) and the user-interface, a website in most cases is responsible entirely for the way in which the interaction looks.

A website will typically use a medium of communication to talk to the server and post/get information from it as necessary. The front-end will display this as it sees fit.

A common form of communication between the two entities occurs using ajax. Ajax is a feature of the JQuery package used within JavaScript which uses JSON (JavaScript object notation) to parse information and requests to an API (application program interface). A user will make a request via the front end which will be passed to the server via an ajax request. If successful, the information is returned via a JSON format which can be read and understood by ajax to be displayed in a desired format on the front-end.

The following is an example from my implementation of a RESTful web services within an application to auction number plates.

Select a batch to view



1-26 Search

[Figure 2.1.1] – Batch selection from on the web page (frontend).

The process begins with the user being shown a form by which they can use to search for a specific batch. The batch is valid between the numbers of 1 and 26 as each batch is stored alphabetically by the first value of the registration.

```
<form id="batch-select-form">
  <input type="number" autocomplete="off" min="1" max="26" size="100" id="batch-select-box" placeholder="1-26">
  <button type="button" id="batch-select-button">Search</button>
</form>
```

[Figure 2.1.2] – Batch selection form shown in raw HTML.

The elements of the form are given unique id's so that they can be identified by the JavaScript functions which will make requests based on the information entered.

```
$('#batch-select-form').submit(function (e) {
  e.preventDefault();
  displayBatch(currentBatch);
});

$('#batch-select-button').click(function (e) {
  e.preventDefault();
  displayBatch(currentBatch);
});
```

```
function displayBatch() {
  $('#error-in-ajax').html("");
  var batchNumber = parseInt($('#batch-select-box').val());
  $.ajax({
    type: "GET",
    url: "http://localhost:8080/batch/" + batchNumber + "/plates",
    contentType: "application/json",
    crossOrigin: true,
    dataType: "json",
    success: setBatchData,
    error: showError
  });
}
```

[Figure 2.1.3] – Action listeners and function containing the ajax request written in JavaScript using JQuery.

By using JQuery, the html form elements can be easily accessed directly via the DOM and the displayBatch() function is called which using an ajax GET request with a specific URL to talk to the server and retrieve the plates from a specific batch via the database.

The plates endpoint is called, which is where the server side part of the operation takes over. At the plates endpoint [Figure 2.1.4], we can see that the url returns a JSON structure containing an array of plates, this is the JSON we wish to extract to be able to interpret and display as we wish on our webpage.

```

{
  "_embedded": {
    "plates": [
      {
        "id": 1,
        "reg": "AZ31 YVE",
        "_links": {
          "self": {
            "href": "http://localhost:8080/plates/1"
          },
          "plate": {
            "href": "http://localhost:8080/plates/1"
          },
          "batch": {
            "href": "http://localhost:8080/plates/1/batch"
          }
        }
      }
    ]
  }
}

```

[Figure 2.1.4] – JSON endpoint on the plate auction web service. Format displayed using the 'JSON Viewer Awesome' plugin for Google Chrome.

This endpoint is generated by the web service whereby the data is also stored on a server-side relational database created using PostgreSQL. An example of the plate data stored in the database is shown below [Figure 2.1.5].

	id integer	reg text	batch_id integer
1	1	AZ31 YVE	1
2	2	AF16 EHO	1
3	3	AP04 OZL	1
4	4	AE23 VZE	1
5	5	AU30 YOH	1

[Figure 2.1.5] – The result of a database select query displayed within pgAdmin 4.

Upon the request being received, the JSON is interpreted and HTML is generated in order to display the plates for the selected batch to the user on the front-end [Figure 2.1.6]. The final result is shown below [Figure 2.1.7].

```
function setBatchData(data) {  
    $('#batch-title').html("<br /><h5>Batch " + parseInt($('#batch-select-box').val()) + "</h5>");  
    $('#batch-view').html("");  
    for (i = 0; i < data._embedded.plates.length; i++) {  
        $('#batch-view').append("<div class='plate'>" + data._embedded.plates[i].reg + "</div>");  
    }  
    document.getElementById("bid-view").style.display = "block";  
}
```

[Figure 2.1.6] – JavaScript function which uses JQuery to get DOM elements and update the html with data from the ajax request.

Select a batch to view

Batch 1

AZ31 YVE
AF16 EHO
AP04 OZL
AE23 VZE
AU30 YOH

Current bid: £60

[Figure 2.1.7] – The final result displayed on the front-end webpage for the user.

The whole process for the user to obtain this information was performed with all of the architecture separated into individual sections with unique responsibility. The front-end and the back-end do not rely on one another to operate initially, but by using ajax we can see how easily it is for the two to communicate and work to achieve one goal for the user. There are multiple steps involved but it means that the process is completely separate.

This is advantageous as it makes upgrading the platform much easier in the future. The API can be updated without the frontend needing to be and vice-versa. This also means that as many applications as needed are able to access the service and make requests to interpret the data as they wish to independently.

This implementation could be improved upon but attaching all of the validation such as bid amounts etc. onto the server-side, as opposed to making the majority of checks within the JavaScript.

According to Lange and Subbu [5] [6], business rules should be imposed on the server-side, as this would mean that any application using the API would automatically have the same rules applied to it.

2.2 – Statelessness

A stateless implementation means that all information regarding the client's session is retained entirely by the client, and is not needed by the server at any point [5]. This means that every request which is made must contain all of the necessary information for it to be completed.

This idea is demonstrating within the plate auction application as show below [Figure 2.2.1].

```
function bidOnBatch(num) {
  $('#error-code').html("");
  var number = parseInt($('#bid-amount').val());
  if (currentBid >= number) {
    $('#error-code').html("Bid must be higher than the current bid (£" + currentBid + ")");
  } else {
    $('#bid-amount').val("");

    $.ajax({
      type: "PUT",
      url: "http://localhost:8080/bid",
      data: { "bid": number, "id": num },
      success: function () {
        getBatchAjax();
      },
      error: function () {
        $('#error-code').html("Error when bidding");
      }
    });
  }
}
```

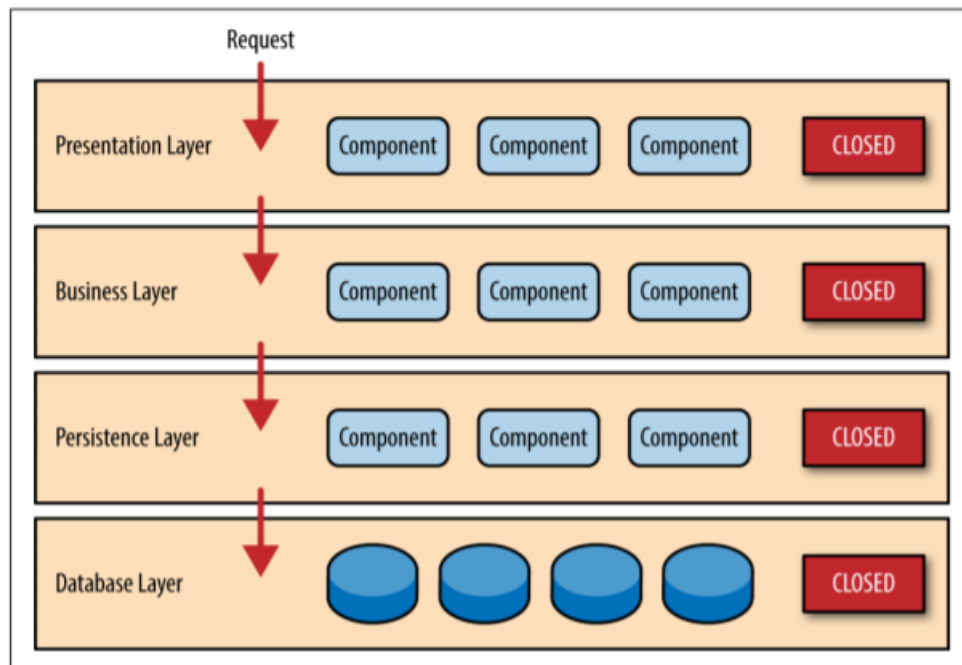
[Figure 2.2.1] – An ajax POST request outlining the data required for the request to be carried out.

This PUT request is being made using ajax. The data attribute is being filled with the necessary data to make the request to the service. The “bid” value is being given an amount which will be applied to the batch. And the “id” is being given a value to identify which batch the bid should be carried out on.

As discussed above, validation is happening within the JavaScript function beforehand which is not ideal for a service which may in the future be consumed by multiple sources, meaning that the rules may not be consistent across each platform.

2.3 - Layered Architecture

The Layered Architecture Pattern conceptualises that each component of an application is split into its own layer, whereby a request starts at the top, and data is passed through each layer to reach the final endpoint [7]. Richards illustrates that there are typically four main layers within an application, however this may vary depending on the specific needs in any given scenario.



[Figure 2.3.1] – Illustration of the layered architecture pattern, taken from ‘Software Architecture Patterns’ by Mark Richards [7].

The above diagram [Figure 2.3.1] by Mark Richards [7] illustrates the flow of data through his proposed four layers within the design pattern. The data begins at the presentation layer (front-end) and finishes in the Database layer (back-end). The pattern is advantageous as it means that if implemented correctly, each layer is completely non-dependent on any other, meaning that changes made to the application should not have any effect in separate layers. This is a similar concept to the client-server principle.

Within the plate auction application, this concept has been utilised as the data made in a request flows through the layers of the application in order.

Current bid: £60

£ 80

[Figure 2.3.2] – The bidding form displayed on the auction webpage where the user can enter an amount to bid.

The process begins in the presentation layer, where the user can enter an amount they wish to bid on a particular batch of number plates [Figure 2.3.2]. When the form is submitted, a PUT request is generated using ajax where the batch id and bid value is passed into the web service [Figure 2.3.3].

```
$.ajax({
  type: "PUT",
  url: "http://localhost:8080/bid",
  data: { "bid": number, "id": num },
  success: function () {
    getBatchAjax();
  },
  error: function () {
    $('#error-code').html("Error when bidding");
  }
});
```

[Figure 2.3.3] – The PUT request built using ajax to send to the web service.

Once the request is generated, a call via the url in the ajax is made to the BatchController within the web-service, moving further down the layers [Figure 2.3.4].

```
@PutMapping("/bid")
public void bid(int id, int bid) {
    Batch batch = batchRepo.findById(id).get();
    batch.setBid(bid);
    batch.setStarted(true);
    batch.setTime();
    batch.addBid();
    batchRepo.save(batch);
}
```

[Figure 2.3.4] – The PUT request endpoint method within the BatchController in the web service.

At this point, the PUT request to place a new bid has been made. The final line of the bid method calls the save method within the batch repository. This is responsible for communicating with the database, whereby the bid value is stored in the data layer, on the batch with the corresponding value [Figure 2.3.5].

	id integer	bid numeric	started boolean	ended boolean	time date	bids integer
1	7	40	true	false	[null]	1
2	2	50	true	false	[null]	4
3	3	10	true	false	[null]	1
4	20	150	true	false	[null]	6
5	25	100	true	false	[null]	1
6	1	80	true	false	[null]	9
7	4	0	false	false	[null]	0
8	5	0	false	false	[null]	0

[Figure 2.3.5] – The updated bid value for batch number 1 (highlighted), updated to the new value of 80. Table generate within pgAdmin 4.

2.4 – Caching

Caching is utilised within RESTful web applications to reduce the number of requests which are made between the client and the server [5]. This overall improves the performance of the system, and becomes more important over time as a system's user base may continue to grow.

By default, some requests are usually cached by the web browser to improve performance, however this can be manually altered during implementation. The plate auction service disabled cache on GET requests as this data is not required to be stored. By not caching the data, the full GET request is made each time the action is called which is ideal for a system where the database may have been updated since a user's last request [Figure 2.4.1].

```
$.ajax({
  cache: false,
  type: "GET",
  url: "http://localhost:8080/batch/" + batchNumber + "/plates",
  contentType: "application/json",
  crossOrigin: true,
  dataType: "json",
  success: setBatchData,
  error: showError
});
```

[Figure 2.4.1] – An ajax GET request showing that caching has been disabled.

2.5 – Code on Demand

As suggested by Giessler, Gebhart [4] and Lange [8], Code on Demand should be considered an optional principle within a RESTful web service. This feature improves flexibility for clients by allowing them to download an applet comprised of JavaScript or Java, sometimes a Flash application, allowing them to run code at runtime.

This is optional as this would only sometimes improve the experience for the client and does not impact the features which make a service 'RESTful'.

The plate auction service does not utilise this feature as it is an optional constraint which may find better use within larger, more complex applications. The simplicity of the auction service means that there would be no real benefit in supplying the client with an application.

2.6 – Uniform Interface

The final key principle of a RESTful web service, which is what allows us all to use a single web browser to use any number of different web applications since the implementation of different components is entirely unique [8].

This final constraint is split up into four sub-constraints, which together form a uniform interface, completing the key principles of a RESTful web service:

Identification of Resources

The REST architecture focuses on the idea that everything is a resource. Each resource needs to be uniquely identifiable (via a URI).

Manipulation of Resources through Representations

This concept tells us that a server should expose a representation of the current state of a resource. This prevents us from using dangerous alternatives such as allowing users to make their own SQL queries.

Within REST, we typically use JSON to expose the state of data, both for GET and POST/PUT requests. The plate auction application utilises JSON for all types of requests which it uses.

```
{
  "_embedded": {
    "plates": [
      {
        "id": 1,
        "reg": "AZ31 YVE",
        "_links": {
          "self": {
            "href": "http://localhost:8080/plates/1"
          },
          "plate": {
            "href": "http://localhost:8080/plates/1"
          },
          "batch": {
            "href": "http://localhost:8080/plates/1/batch"
          }
        }
      }
    ]
  }
},
```

```
$.ajax({
  type: "PUT",
  url: "http://localhost:8080/bid",
  data: { "bid": number, "id": num },
  success: function () {
    getBatchAjax();
  },
  error: function () {
    $('#error-code').html("Error when bidding");
  }
});
```

[Figure 2.6.1] – A batch shown in JSON format and a PUT request passing in data also in the JSON format.

Self-Descriptive Messages

This constraint implies that each request made is accompanied with a description of the type of action which is being made. The four main tasks follow the CRUD design (create, read, update and delete). This is translated into POST (insert new data), GET (read existing data), PUT (update existing data) and DELETE (delete existing data). Each request should be accompanied with an appropriate description.

The plate auction service utilises these messages when performing requests. As shown below [Figure 2.6.2], GET requests are sent to read data and PUT requests are used to update a batch with a new bid.

```
function displayBatch() {
    $('#error-in-ajax').html("");
    var batchNumber = parseInt($('#batch-select-box').val());
    $.ajax({
        type: "GET",
        url: "http://localhost:8080/batch/" + batchNumber + "/plates",
        contentType: "application/json",
        crossOrigin: true,
        dataType: "json",
        success: setBatchData,
        error: showError
    });
}
```

```
$.ajax({
    type: "PUT",
    url: "http://localhost:8080/bid",
    data: { "bid": number, "id": num },
    success: function () {
        getBatchAjax();
    },
    error: function () {
        $('#error-code').html("Error when bidding");
    }
});
```

```
@PutMapping("/bid")
public void bid(int id, int bid) {
    Batch batch = batchRepo.findById(id).get();
    batch.setBid(bid);
    batch.setStarted(true);
    batch.setTime();
    batch.addBid();
    batchRepo.save(batch);
}
```

[Figure 2.6.2] – GET and PUT requests being made by the website and the bidding PUT endpoint within the BatchController in the service.

Hypermedia as the Engine of Application State

This concept suggests that an initial request should be made using a URI, then all other navigational requests should be made using links contained within the URI object.

This concept is implemented within the auction plate service which allows us to instantly make GET requests on the service and then follow links to look further into the data. An example of this is shown below whereby the endpoints contain a list of 'links' which can be accessed by the user [Figure 2.6.3].

```
{
  "id": 1,
  "reg": "AZ31 YVE",
  "_links": {
    "self": {
      "href": "http://localhost:8080/plates/1"
    },
    "plate": {
      "href": "http://localhost:8080/plates/1"
    },
    "batch": {
      "href": "http://localhost:8080/plates/1/batch"
    }
  }
},
```

[Figure 2.6.3] – list of links contained within the plate endpoint.

3 – Conclusion

By utilising all of these concepts (excluding Code on Demand), we can call a web service RESTful. RESTful web services provide us with the current way in which we interact with web services, completely separating client and server-side architecture.

With this, we are easily able to use almost any device and interface to consume a RESTful web service, allowing for complete flexibility when it comes to representing data to the end-user.

References

- [1] Oracle, "What Are RESTful Web Services?," 2013. [Online]. Available: <https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html>. [Accessed 4 April 2019].
- [2] P. Adamczyk, P. H. Smith, R. E. Johnson and M. Hafiz, "REST: From Research to Practice (Chapter 1 - REST and Web Services: In Theory and in Practice)," pp. 1-20, June 2011.
- [3] L. Richardson and S. Ruby, RESTful Web Services, Sebastopol: O'Reilly Media, 2007.
- [4] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger and S. Abeck, "Best Practices for the Design of RESTful Web Services," in *The Tenth International CONferences on Software Engineering Advances*, 2015.
- [5] K. Lange, The Little Book on REST Services, Copenhagen, 2016.
- [6] S. Allamaraju, RESTful Web Services Cookbook, Sebastopol: O'Reilly, 2010.
- [7] M. Richards, Software Architecture Patterns, Sebastopol: O'Reilly, 2015.
- [8] K. Lange, "Kenneth's Lange's Blog," 27 February 2016. [Online]. Available: <https://www.kennethlange.com/what-are-restful-web-services/>. [Accessed 7 May 2019].



SERVERLESS TECHNOLOGIES AND THE CIRCUIT BREAKER PATTERN

Lee Rodger - 1809707

[Abstract](#)

This document outlines the theory and practice behind “serverless” technologies and investigates into their relevance with regards to web hosting. Also discussed is the circuit breaker pattern.

Contents

1 – Introduction.....	2
2 – Serverless technologies.....	3
2.1 – Analysis of the Benefits and Drawbacks of Serverless Technologies.....	3
2.2 – Serverless in Industry	4
3 – Circuit Breaker Pattern.....	5
3.1 – Implementations of Circuit Breakers	6
References	7

1 – Introduction

This report outlines and analyses both serverless technologies and the circuit breaker pattern with regards to modern web hosting and development. Serverless technologies have taken the web hosting industry by storm with leading provider Amazon providing a vast number of services to modern web services.

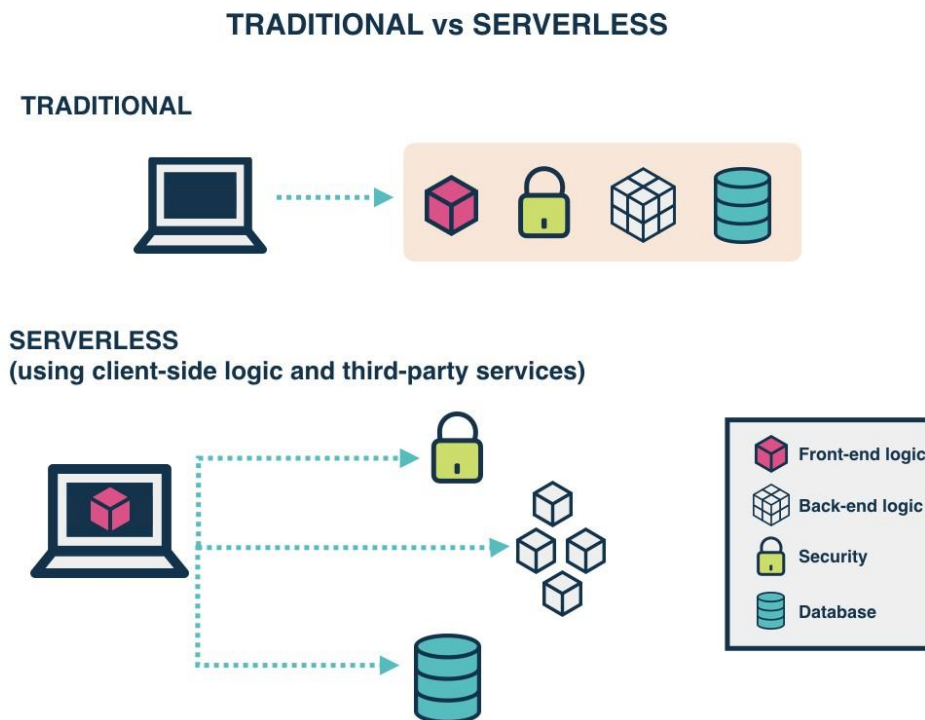
With the amount of requests being made between applications and services, inevitably things go wrong. These occurrences are handled and managed through the implementation of the circuit breaker pattern which will be discussed in detail within this report (Section 3).

2 – Serverless technologies

Serverless technologies are ever present in all modern systems which we use today on the internet. The principle focuses on a cloud based execution model, whereby a cloud provider will run the server and dynamically manage the resources necessary [1].

Before these systems were in place, businesses would often rent or provide their own server space for their online services which would be of a fixed size regardless of the amount of traffic which is received. The constant increase of web users meant that companies had to spend more on buying more server space which would grow continually costlier over time. The business world was in desperate need of a dynamic system whereby they would only be paying for the server space which they need at any given time.

Serverless technologies, unlike the name suggests, do indeed use servers as an integral part of the process. However, it is down to a cloud provider to handle complexity and manage individual servers [2]. Certain events are triggered which call upon the cloud provider to use other services to perform requests for the client.



[Figure 2.1] - A Diagram demonstration the difference between traditional and serverless technologies.

Taken from <https://www.gocd.org/2017/06/26/serverless-architecture-continuous-delivery/>

The above diagram [Figure 2.1] Shows how cloud providers utilise serverless technologies to split up the responsibility of parts of a service, such as security, data storage and logic.

2.1 – Analysis of the Benefits and Drawbacks of Serverless Technologies

Serverless technologies provide us with a multitude of advantages which have improved the way in which web services are hosted and used.

These technologies mean that developers no longer have to deal with the management of servers. They are handled entirely by the cloud provider which makes creation and publishing far easier and quicker than before.

The dynamic nature of serverless technologies mean that businesses will only pay for the amount of server space which they are using at any given time [3]. Times of less server traffic will result in less space being allocated, meaning that these periods will result in less cost, and vice versa. The process of this growing and shrinking of services is also considered 'rapid' meaning that the amount of allocated space at any given time is exactly ideal for the level of demand which the service is receiving.

Contino [3] suggest that security is also a large advantage of these technologies as the cloud provider will always be using up-to-date and reliable techniques. This means that developers can focus more closely on application level security, such as cross-site-scripting [3] and not have to worry about server based security measures.

Although it is beneficial for the cloud provider to provide a lot of functionality for developers, this in turn leads to a major disadvantage. By utilising these serverless systems, developers may become 'locked in' to continue using these with no option to leave due to a lack of understanding of the underlying server architecture. Businesses which intend to venture off and develop their own solutions may not be able to quickly adapt and end using these technologies if they so desire [4].

Although largely beneficial to modern web services which we all depend upon today, the 'locking in' effect that these technologies have on businesses means that they may be taken advantages of with regards to cost, as well as large providers such as Amazon becoming in danger of monopolising the serverless industry.

2.2 – Serverless in Industry

Within industry, Amazon are a leading provider in serverless technologies. Through their Amazon Web Services (AWS) platform. The company claims to deliver a "production serverless application that can run the scale demands of a platform with a broad set of capabilities" [5].

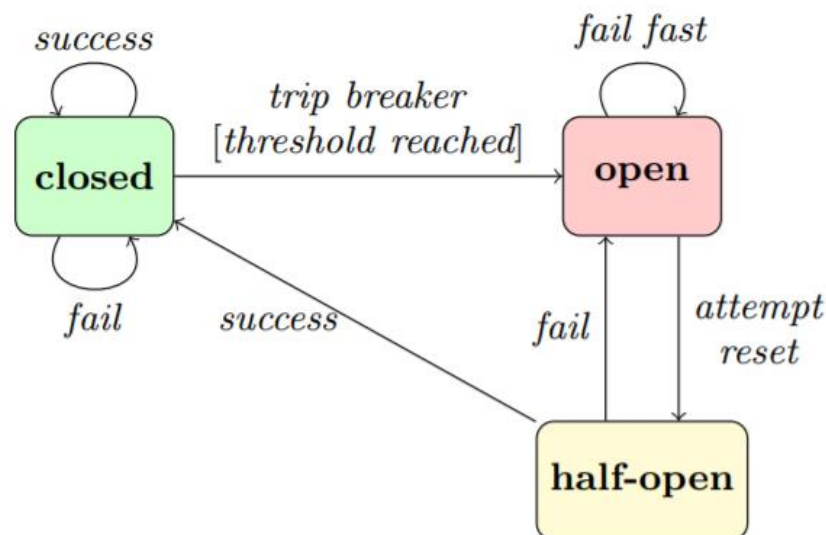
AWS provide a service called 'lambda' which allows users to run code without the need to manage servers and only pay for the computational time which is used [6]. Netflix utilise this technology in order to reduce the costs of running their service. The video platform holds billions of hours of content in over 50 different formats [7], and delivers their service to more than 158 million subscribers [8] worldwide as of the first quarter of 2019.

AWS Lambda is used to effectively transform media at the necessary rate for the number of customers watching media via the platform. The number of concurrent Netflix users will peak and trough for various reasons such as time of day, new show releases etc. The Lambda service will grow and shrink as necessary to provide Netflix with a service which will not waste the business money.

3 – Circuit Breaker Pattern

A circuit breaker prevents an application from attempting to carry out a task which will undoubtedly fail [9]. For example, if an application had to make a database call hundreds of times per second, it would be undesirable to have the same error take place hundreds of times until the connection would inevitably time out. Therefore, an implementation of a circuit breaker would stop this many requests happening entirely by only issuing the requests if the service is available. If not, the circuit should ‘break’ and allow the developer to display a single error message explaining the problem to the user.

Although this pattern is useful in optimising the way in which a service calls another and in displaying any errors, this does have a negative impact on the performance of the system. It will take longer to issue requests as first the circuit breaker will need to determine if the service is ‘alive’ before any operations can be performed. This would happen before each and every operation effectively doubling the requests sent to and from a service such as a database. It is important that developers implement this pattern carefully, without too greatly impacting on the performance of the system by utilising this strategy where it is absolutely necessary.



[Figure 3.1] – A state diagram illustrating the circuit breaker pattern.

Taken from: Circuit Breakers, Discovery, and API Gateways in Microservices, 2016. [9]

The above diagram [Figure 3.1] is a state diagram representing the states in which a service can be in when utilising the circuit breaker pattern [9]. Each of these states holds a different set of rules for how the target service can be used within the system, as discussed by Fabrizio and Weber [9].

Closed

In this state, requests are passed to the target service and any faults increase the failure and timeout counters within the circuit breaker. Once the threshold is met, the pattern moves into the ‘open’ state, meaning that the circuit breaker has been ‘tripped’.

Open

This state stops any request from being passed to the target service, and instead issues the client with a message describing the failure that has occurred. This state will then transition into the 'half-open' state either when the service becomes responsive after periodic 'pinging' or after an amount of time has passed which is to be decided by the developer.

Half-Open

When in this state, only a limited number of calls may be sent to the receiving service. If all of the responses are successful, then the circuit breaker will return to the 'closed' state, whilst also resetting both the timeout and failure counters so that the process can begin again.

3.1 – Implementations of Circuit Breakers

A circuit breaker can be implemented in a number of different useful ways as described by Fabrizio and Weber [10].

A **Client-side** implementation is advantageous as when in an open state, will prevent the target service from receiving any further messages. This assumed however that the clients can be forced into using such implementation, and if used, only the client locally will be aware as to the availability of a service.

Service-side circuit breakers allow for no assumptions when it comes to dealing with the client, as malicious activity need not be accounted for. However, a drawback to this implementation means that extra resources are needed for it to run, and as discussed in section 3, circuit breakers do indeed have an impact on the performance of a service.

Alternatively, a **Proxy** circuit breaker could be implemented which would sit between the service and the client. This would mean that there would exist a circuit breaker for every client and service which would lead to "bottlenecking". This could lead to a slowdown in service requests as the proxy would have to deal with all calls between the client and the service.

Each of these implementations as discussed by Fabrizio and Weber [10] all hold their own advantages and disadvantages, and when designing a service, the way in which a circuit breaker is implemented should be carefully thought out by the developer.

References

- [1] G. Adzic and R. Chatley, "Serverless Computing: Economic and Architectural Impact," Paderborn, 2017.
- [2] Deloitte, "Serverless Computing - Architectural Considerations & Principles," Deloitte Touche Tohmatsu India LLP, 2018.
- [3] Contino, "Introduction to Serverless Computing with AWS Lambda".
- [4] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski and P. Suter, Serverless Computing: Current Trends and Open Problems, 2017.
- [5] Amazon Web Services, "Serverless," Amazon, 2019. [Online]. Available: <https://aws.amazon.com/serverless/>. [Accessed 30 April 2019].
- [6] Amazon Web Services, "AWS Lambda," Amazon, 2019. [Online]. Available: <https://aws.amazon.com/lambda/>. [Accessed 30 April 2019].
- [7] R. Akiwatkar, "10 Practical Examples of AWS Lambda," 24 September 2018. [Online]. Available: <https://www.simform.com/serverless-aws-lambda-examples/>. [Accessed 30 April 2019].
- [8] Statista, "Number of Netflix streaming subscribers worldwide from 3rd quarter 2011 to 1st quarter 2019 (in millions)," Statista, 2019.
- [9] F. Montesi and W. Janine, "Circuit Breakers, Discovery, and API Gateways in Microservices," University of Southern Denmark, Odense, 2016.
- [10] F. Montesi and W. Janine, "From the decorator pattern to circuit breakers in microservices," in *Conference'17*, Washington, DC, 2017.

APPENDIX

Service Code in order Alphabetically of Class Name:

```
package com.example.auction.auction;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.CrossOrigin;

@CrossOrigin(origins = "*")
@SpringBootApplication
public class AuctionApplication {

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

}
```

```
package com.example.auction.auction;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

import javax.persistence.*;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.Date;
import java.util.List;

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
@Table(name="batch")
public class Batch {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private double bid;

    private boolean started;
    private boolean ended;
    private java.sql.Timestamp time;
    private int bids;

    @OneToMany(fetch = FetchType.LAZY, targetEntity = Plate.class, mappedBy = "batch")
    private List<Plate> plates;

    public Batch() {
        this.bid = 0;
        started = false;
        ended = false;
        time = null;
        bids = 0;
    }

    public Batch(double bid) {
        this.bid = bid;
        started = false;
        ended = false;
        time = null;
        bids = 0;
    }

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
        return "Batch{" +
            "id=" + id +
            ", plates=" + plates +
            '}';
    }

    public List<Plate> getPlates() {
        return plates;
    }

    public void setPlates(List<Plate> plates) {
        this.plates = plates;
    }

}
```

```

    public double getBid() {
        return bid;
    }

    public boolean isStarted() {
        return started;
    }

    public void setStarted(boolean started) {
        this.started = started;
    }

    public boolean isEnded() {
        return ended;
    }

    public void setEnded(boolean ended) {
        this.ended = ended;
    }

    public java.sql.Timestamp getTime() {
        return time;
    }

    public void setTime() {
        if(time == null) {
            time = new java.sql.Timestamp(new Date().getTime());
        }
    }

    public void setBid(double bid) {
        this.bid = bid;
    }

    public int getBids() {
        return bids;
    }

    public void addBid() {
        this.bids++;
    }
}

```

```

package com.example.auction.auction;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class BatchController {
    private BatchRepository batchRepo;

    @Autowired
    public BatchController(BatchRepository batchRepo) {
        this.batchRepo = batchRepo;
    }

    @PutMapping("/bid")
    public void bid(int id, int bid) {
        Batch batch = batchRepo.findById(id).get();
        batch.setBid(bid);
        batch.setStarted(true);
        batch.setTime();
        batch.addBid();
        batchRepo.save(batch);
    }
}

```

```

package com.example.auction.auction;

import java.util.ArrayList;

public class BatchGenerator {
    private static ArrayList<String> batch = new ArrayList<>();

    public static ArrayList<String> generateBatch() {

        for (int i = 0; i < 5; i++) {
            batch.add(RegGenerator.generateReg('A'));
        }

        return batch;
    }
}

```

```

    }
}
package com.example.auction.auction;

import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(collectionResourceRel = "batch", path="batch")
public interface BatchRepository extends PagingAndSortingRepository<Batch, Integer> {
}

```

```

package com.example.auction.auction;

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

@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
@Table(name = "plate")
public class Plate implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String reg;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "batch id")
    private Batch batch;

    public Plate() {
    }

    public Plate(String reg, Batch batch) {
        this.reg = reg;
        this.batch = batch;
    }

    @Override
    public String toString() {
        return "Plate{" +
            "id=" + id +
            ", reg='" + reg + '\'' +
            ", batch=" + batch +
            '}';
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getReg() {
        return reg;
    }

    public Batch getBatch() {
        return batch;
    }

    public void setBatch(Batch batch) {
        this.batch = batch;
    }
}

```

```

package com.example.auction.auction;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import javax.transaction.Transactional;
import java.util.ArrayList;
import java.util.List;

@Service
public class PlateFillerService {

    private final PlateRepository plateRepo;
    private final BatchRepository batchRepo;
}

```

```

    @Autowired
    public PlateFillerService(PlateRepository plateRepo, BatchRepository batchRepo) {
        this.plateRepo = plateRepo;
        this.batchRepo = batchRepo;
    }

    @PostConstruct
    @Transactional
    public void fill() {
        Batch batch;

        String reg;

        for(int i = 0; i < 26; i++) {
            batch = new Batch();
            batchRepo.save(batch);
            for(int j = 0; j < 5; j++) {
                reg = RegGenerator.generateReg((char) (i + 65));
                plateRepo.save(new Plate(reg, batch));
            }
        }
    }
}

```

```

package com.example.auction.auction;

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

public class PlateList {
    private static PlateList ourInstance = new PlateList();

    private ArrayList<Plate> plates;

    public static PlateList getInstance() {
        return ourInstance;
    }

    private PlateList() {
        plates = new ArrayList<>();
    }

    public void add(Plate plate) {
        plates.add(plate);
    }

    public void delete(int index) {
        plates.remove(index);
    }

    public void delete(Plate plate) {
        plates.remove(plate);
    }

    public ArrayList<Plate> getPlates() {
        return plates;
    }
}

```

```

package com.example.auction.auction;

import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource(collectionResourceRel = "plates", path="plates")
public interface PlateRepository extends PagingAndSortingRepository<Plate, Integer> {
}

```

```

package com.example.auction.auction;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootApplication
@EnableTransactionManagement
public class RegApplication {

    public static void main(String[] args) {

```

```

        SpringApplication.run(RegApplication.class, args);
    }
}

```

```

package com.example.auction.auction;

import java.util.Random;

public class RegGenerator {
    private static Random rand = new Random();

    public static String generateReg(char c1) {
        char c2 = randomLetter();

        int c3 = randomNumber();
        int c4 = randomNumber();

        char c5 = randomLetter();
        char c6 = randomLetter();
        char c7 = randomLetter();

        String reg = "" + c1 + c2 + c3 + c4 + " " + c5 + c6 + c7;

        return reg;
    }

    private static char randomLetter() {
        return (char) (rand.nextInt(26) + 65);
    }

    private static int randomNumber() {
        return rand.nextInt(10);
    }
}

```

```

package com.example.auction.auction;

import org.springframework.context.annotation.Configuration;
import org.springframework.data.rest.core.config.RepositoryRestConfiguration;
import org.springframework.data.rest.webmvc.config.RepositoryRestConfigurerAdapter;

@Configuration
public class RepositoryConfig extends RepositoryRestConfigurerAdapter {

    @Override
    public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config) {
        config.exposeIdsFor(Plate.class);
        config.exposeIdsFor(Batch.class);
    }
}

```

WEB INTERFACE

HTML

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <link rel="stylesheet" href="styles/bootstrap/bootstrap.min.css">
  <link rel="stylesheet" href="styles/styling.css" />
  <title>Auction Site</title>
</head>
<body>
  <div class="container">
    <div class="row" id="header">
      <h1>Plate Auctions</h1>
    </div>
    <hr />
    <div class="row">
      <div class="col-11 offset-1">
        <h3>Current List</h3>
        <br />
      </div>
    </div>
    <div class="row">
      <div class="col-6 offset-1">
        <h5>Select a batch to view</h5>
      </div>
    </div>
    <div class="row">
      <div class="col-6 offset-1">
        <form id="batch-select-form">
          <input type="number" autocomplete="off" min="1" max="26"
size="100" id="batch-select-box" placeholder="1-26">
          <button type="button" id="batch-select-button">Search</button>
        </form>
      </div>
    </div>
    <div class="row">
      <div class="col-6 offset-1">
        <p id="error-in-ajax" class="error"></p>
      </div>
    </div>
    <div class="row">
      <div class="col-4 offset-1">
        <h5 id="batch-title"></h5>
        <p id="batch-view">

        </p>
      </div>
    </div>
    <div class="row" id="bid-view">
      <div class="col-6 offset-1" id="current-bid">
        <h6>Current Bid: £0</h6>
      </div>
      <div class="col-6 offset-1">
        <form id="batch-bid-form">
          £<input type="number" id="bid-amount" autocomplete="off" min="0"
placeholder="Bid">
          <button type="button" id="bid-button">Bid</button>
        </form>
      </div>
    </div>
  </div>
</body>
</html>
```

```

        </form>
    </div>
    <div class="row">
        <div class="col-6 offset-1 error">
            <h6 id="error-code"></h6>
        </div>
    </div>
</div>
<hr />
<div class="row">
    <div class="col-6 offset-1">
        <h2>Live Auctions</h2>
        <p id="live-auctions">

            </p>
    </div>
</div>
</div>
<script src="scripts/jquery-3.3.1.min.js"></script>
<script src="scripts/bootstrap/bootstrap.min.js"></script>
<script src="scripts/script.js"></script>
</body>
</html>

```

```

.error {
    color : red;
}

.plate {
    background-color : gold;
    border: dashed;
    font-size: x-large;
    text-align: center;
}

.title {

}

#header {
    background-color: gold;
}

```

CSS

```
.error {  
  color : red;  
}  
  
.plate {  
  background-color : gold;  
  border: dashed;  
  font-size: x-large;  
  text-align: center;  
}  
  
.title {  
  
}  
  
#header {  
  background-color: gold;  
}
```

JAVASCRIPT

```
var currentBatch = 0;
var currentBid = 0;

$(function () {

    document.getElementById("bid-view").style.display = "none";

    getLiveAuctions();

});

function getLiveAuctions() {
    $.ajax({
        cache: false,
        type: "GET",
        url: "http://localhost:8080/batch",
        contentType: "application/json",
        crossOrigin: true,
        dataType: "json",
        success: showLive
    });
}

function showLive(data) {
    $('#live-auctions').html("<br /><br />");
    for (i = 0; i < data._embedded.batch.length; i++) {
        if (data._embedded.batch[i].started == true) {
            $('#live-auctions').append("<h5>Batch " + data._embedded.batch[i].id + " | £" + data._embedded.batch[i].bid + "</h5>");
        }
    }
}

function genMulti(data, index) {
    $('#reg-list-' + (index + 1)).html("<h3>List of Plates</h3>");
    console.log("Index " + index);
    var platelist = data._embedded.plates;
    for (i = 0; i < 5; i++) {
        $('#reg-list-' + (index+1)).append(plateList[i].id + " | " + plateList[i].reg + " <br>");
    }
}

function genBid(data) {
    $('#current-bid-1').val(data.bid);
}

function genBatch(data) {
    $('#reg-list-1').html("<h3>List of Plates</h3>");
    var platelist = data._embedded.plates;
    for (i = 0; i < 5; i++) {
        $('#reg-list-1').append(plateList[i].id + " | " + plateList[i].reg + " <br>");
    }
}

function ajaxError() {
    window.alert("error in ajax");
}
```

```

$('#bid-form').submit(function (e) {
    e.preventDefault();
    bidEntered();
});

$('#bid-button-1').click(function (e) {
    e.preventDefault();
    bidEntered();
});

$('#batch-select-form').submit(function (e) {
    e.preventDefault();
    displayBatch(currentBatch);
});

$('#batch-select-button').click(function (e) {
    e.preventDefault();
    displayBatch(currentBatch);
});

$('#batch-bid-form').submit(function (e) {
    e.preventDefault();
    bidOnBatch(currentBatch);
});

$('#bid-button').click(function (e) {
    e.preventDefault();
    bidOnBatch(currentBatch);
});

function displayBatch() {
    $('#error-in-ajax').html("");
    var batchNumber = parseInt($('#batch-select-box').val());
    $.ajax({
        cache: false,
        type: "GET",
        url: "http://localhost:8080/batch/" + batchNumber + "/plates",
        contentType: "application/json",
        crossOrigin: true,
        dataType: "json",
        success: setBatchData,
        error: showError
    });

    getBatchAjax();
}

function getBatchAjax() {
    getLiveAuctions();
    var batchNumber = parseInt($('#batch-select-box').val());
    $.ajax({
        cache: false,
        type: "GET",
        url: "http://localhost:8080/batch/" + batchNumber,
        contentType: "application/json",
        crossOrigin: true,
        dataType: "json",
        success: displayBids,
        error: showError
    });
}

```

```

}

function displayBids(data) {
    $('#current-bid').html("<h6>Current bid: £" + parseInt(data.bid) + "</h6>");
    currentBatch = parseInt(data.id);
    currentBid = parseInt(data.bid);
}

function setBatchData(data) {
    $('#batch-title').html("<br /><h5>Batch " + parseInt($('#batch-select-box').val())
+ "</h5>");
    $('#batch-view').html("");
    for (i = 0; i < data._embedded.plates.length; i++) {
        $('#batch-view').append("<div class='plate'>" + data._embedded.plates[i].reg +
"</div>");
    }
    document.getElementById("bid-view").style.display = "block";
}

function bidOnBatch(num) {
    $('#error-code').html("");
    var number = parseInt($('#bid-amount').val());

    if (currentBid >= number) {
        $('#error-code').html("Bid must be higher than the current bid (£" +
currentBid + ")");
    } else {
        $('#bid-box').val("");

        $.ajax({
            type: "PUT",
            url: "http://localhost:8080/bid",
            data: { "bid": number, "id": num },
            success: function () {
                getBatchAjax();
            },
            error: function () {
                $('#error-code').html("Error when bidding");
            }
        });
    }

    $('#bid-amount').val("");
}

function showError(data) {
    $('#error-in-ajax').html("<h6>Could not locate batch</h6>")
}

```

404 HTML

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="../styles/bootstrap/bootstrap.min.css">
  <link rel="stylesheet" href="error-styles.css">
  <meta charset="utf-8" />
  <title>Not found</title>
</head>
<body>
  <h1>404 Not Found</h1>
  <h5>Sorry, we were unable to locate the page you are looking for.</h5>
</body>
</html>
```

404 CSS

```
.error-text {
  text-align: center;
  color: blue;
}
.blue-text {
  color: blue;
}
```

