

Web Services

by Adam Peter Davies

FILE	2784_ADAM_PETER_DAVIES_WEB_SERVICES_181292_2039850596.PDF (1.22M)		
TIME SUBMITTED	08-MAY-2019 11:22PM (UTC+0100)	WORD COUNT	7899
SUBMISSION ID	106704274	CHARACTER COUNT	44446



Part I: Analysis on Best Recommendations in the Implementation of RESTful Web Services

(ACCA7005) Web Services & (ACCA7004) Web Development

by Adam Peter Davies (P147908)
MSc Software Engineering
University of Wales Trinity Saint David

2019

Table of Contents

1	Introduction	1
2	Background Information	1
2.1	Web Services Overview	1
2.2	RESTful Web Services	2
3	Discussion	3
3.1	URI Formatting	3
3.2	HTTP Methods	5
3.3	Use of Sub Resources	6
3.4	HATEOAS	6
3.5	Paging and Sorting	7
3.6	HTTP Headers	8
3.7	Use of an ORM	8
3.8	Stateless and Cache	8
3.9	Error Handling	8
3.10	Versioning	10
4	Conclusion	10
	References	11
	Bibliography	14
	Appendix	15

Table of Figures

1 RESTful Web Service Architecture 2

Table of Tables

1 Common HTTP Error Codes 9

1 Introduction

The purpose of this report is to discuss the best practice recommendations for the implementation of RESTful web services. The report is structured with firstly a section on related background information, which briefly overviews definitions of a web service and the concepts of RESTful web services. This is followed by the main body of the report, which presents a discussion on analysing and evaluating the implementation of best practice recommendations found from researched literature. All findings are reflected upon at the end of the report.

2 Background Information

This section of the report presents the background information to the reports areas of focus. Provided is a brief overview of web services followed by an overview of general RESTful web service concepts.

2.1 Web Services Overview

A web service is a program which is delivered and can be accessed over the internet adopting various protocols, such as Hypertext Transport Protocol (HTTP), Transmission Control Protocol (TCP) etc. [1]. Web services use standardized means of communicating and use languages such as Extensible Markup Language (XML) and JavaScript Object Notation (JSON). Due to the communication language being standardized it allows the cores of a web service application to be written in any language. This flexibility allows web service applications to communicate regardless of what language each service is written in [2].

There are many ways a web service can be implemented to interact with the web, these include Representational State Transfer (REST), Simple Object Access Protocol (SOAP), Remote Procedure Protocol (RPC) and Hypermedia As The Engine Of Application State (HATEOS) [3]. Although these methods of implementation are comparable they are all varying technologies with conflicting definitions in researched literature. For example, SOAP is considered as a standard to follow for the communication between web services, whereas REST is considered an architectural style [4]. However, they are still compared by many even though they can be implemented together [4, 5]. REST and SOAP are the two mainly used approaches to interface web services to the web [6]. It is important to note that all the aforementioned approaches can be combined.

2.2 RESTful Web Services

As mentioned previously Representational State Transfer (REST) is an architectural style which sets design principles and rules for developing web services. Web services which adopt a REST architecture are also known as RESTful web services (RWS). The term REST was defined by Fielding [7] in 2000 where he discussed the use of REST principles (known as HTTP object model at the time) in the development of HTTP 1.1 and Uniform Resource Identifiers (URI) standards. Fielding observed that developing distributed systems without constraints or limits means that resulting systems are difficult to extend and maintain [8]. This led to the introduction of REST constraints and principles.

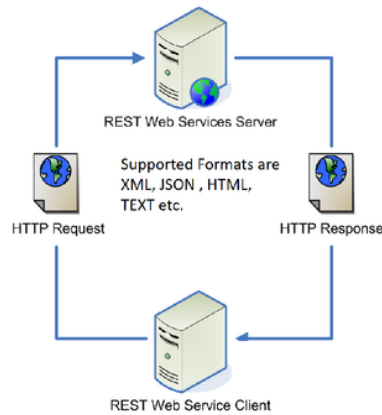


Figure 1: RESTful Web Service Architecture [9]

A key concept of RESTful services is that they are resource focused, meaning that data and functionality are considered as resources which are accessed through a URI. Resources are invoked upon by using well-defined HTTP methods [8, 10]. There are many other key concepts which Fielding conceptualized to define a RESTful system. These principles are summarized as follows:

- **Strict use of HTTP Methods:** RESTful web services feature the explicit use of HTTP methods as defined by RFC 2616 [11]. This means that HTTP methods should only be used to perform their counterpart CRUD operations, in a one-to-one mapping principle [12]. For example, a GET request on the server should not create a new resource.
- **Stateless:** This principle means that the server does not store any of the client's session data server side. REST uses state transfer to effectively pass the session around instead of directly storing session data server-side [12]. This allows RESTful web services to be scalable and cope with high-performance demands.

- Use of URI to expose resources: In a RESTful web service, the use of a URI is used to directly control actions made upon a resource [12, 13]. The URI can be thought of as a way of identifying a resource.
- Client-server: This constraint allows the server and client to be loosely coupled. This means the client does not need to know the implementation of the server, and the server does not need to know how the client handles the data [8]. This reduces dependency between the two layers.
- Layered: A REST-based service should implement various layers which can enhance scalability and improve performance. It also allows for various components to be split into specific layers increasing the maintainability of a system.

3 Discussion

Present day RESTful web services provide a lightweight, maintainable and scalable architecture for developing web services. They are gaining in importance due to the implementation of technologies such as Cloud Computing, Internet of Things (IoT) and Microservices. However, many existing REST services are implemented by following no true framework. Thus resulting in desirable qualities being omitted from many current systems [14]. A study by Palma et al [15] who proposed a means of testing Linguistic antipatterns implemented within RESTful web services, highlights that the majority of tested APIs showed URI syntactical errors, and do not comply to RESTful architecture standards [15].

There are many sources which propose the best practices for the development of RESTful services, however many of the sources are conflicting [16]. All best practices investigated will be discussed in the following sections, using pertinent literature to highlight views on the recommendations.

3.1 URI Formatting

Formatting of URIs in the implementation of a RESTful service is key, as the services are built around URIs to uniquely identify the resources of the application [17, 18]. Furthermore, it is what allows consumers of the web service to access and expose resources [11]. This is why making endpoint URIs readable and easy to understand fundamental in building a successful RESTful implementation.

One of the recommendations for URI formatting is to remove the use of trailing forward slashes in any endpoints. Masse [17] believes that this is an important factor as every resource should be uniquely identified, meaning every character in a URI is used for this.

RESTful web services are noun centralized for the naming of URI endpoints [19]. Vinoski [20] believes that URLs are names, not verbs. Edstrom and Tilevich [21] express that "...REST is commonly described as applying verbs (HTTP methods) to nouns (URIs)". This means that HTTP CRUD methods should not be used in the naming of URIs. The use of plural nouns should be used for collection names and store names, whereas singular nouns should be used for document names [15, 22]. By applying these naming practices it can make it easier for consumers to access specific resources [22].

With longer named resources it is important to consider the use of word breaks to ensure readability is maintained. Masse [17] believes that underscores should not be used as browsers can partially obscure the view so it seems they are not present. He believes that hyphens (-) should be used instead.

The use of lower case characters is preferred. Although host components of the URI are not case-sensitive, resource identifiers are and should be kept consistent as defined by 3986 standards [15, 17]. Additionally, the removal of file extensions at the end of a URI is considered desirable. RESTful services should rely on HTTP headers to distinguish the content media-type in the form of an acceptance header [17]. Shown below are examples of these best practice recommendations implemented.

```
//Trailing slashes
localhost:8080/business-accounts/ //Incorrect URI (use of trailing /)
localhost:8080/business-accounts //Correct use

//Verbs
localhost:8080/get-user-accounts //Incorrect URI (use of verbs)
localhost:8080/user-accounts //Correct use

//Case
LOCALHOST:8080/registration-plates //Correct use (capitalization on host)
localhost:8080/registration-Plates //Incorrect use (mixing cases)
localhost:8080/registration-plates //Correct use

//Breaks
localhost:8080/user_account/bid_win Incorrect use of underscores
localhost:8080/user-account/bid-win //Correct use of hyphens
```

3.2 HTTP Methods

A key factor for implementing HTTP methods in a RESTful web service is that method operations must be explicit [18]. This means that a HTTP method used to evoke an endpoint must relate to the RFC 2616 definition. Outlined in the following describes how each HTTP method should be used:

- GET: A GET must be used to retrieve a resource, whether it be from a single resource such as a document or from a collection. They must not change the state of a collection [20].

```
/**
 * Get request mapping for '/' endpoint
 * @return html home page acting as the landing page for the site
 */
@RequestMapping("/")
public String home(){
    return "home";
}
```

- PUT: A PUT must be used to update a sorted resource within a collection [23].

```
@PostMapping("/auction")
public void updateExpiredBatches(){
    //Set up initial page values
    int page = 0;
    int pageSize = 10;
    //get the first page from auctionGroup repository query
    Page<AuctionGroup> auctionGroupsPage1 = auctionGroupRepository.findAll(PageRequest.of(page, pageSize));
    int numberOfPages = auctionGroupsPage1.getTotalPages();
    List<AuctionGroup> auctionGroups = new ArrayList<>(); //define new list of auctionGroups

    //Loop around each page adding each element to the list
    for(int i = 0; i < numberOfPages; i++){
        Page<AuctionGroup> auctionGroupsPages = auctionGroupRepository.findAuctionGroupsByExpiredIsFalse(
            PageRequest.of(i, pageSize));
        auctionGroups.addAll(auctionGroupsPages.getContent()); //add all page content to list
    }
    //loop around each entry in list and check if the listing is expired
    for(int i = 0; i < auctionGroups.size(); i++){ //if listing is expired then update the
        // auction group by id updating expired field to true
        if(auctionGroups.get(i).getDeadlineDate().isBefore(LocalDate.now())){
            (auctionGroups.get(i).getDeadlineDate().isEqual(LocalDate.now())) &&
            (auctionGroups.get(i).getDeadlineTime().isBefore(LocalDate.now()))){
                auctionGroupRepository.updateAuctionGroupExpiredByAuctionGroupId( expired: true,
                    auctionGroups.get(i).getAuctionGroupId());
            }
        }
    }
}
```

- POST: A POST must be used to create a new resource within a collection. It is also usually considered good practice to invoke any functionality controllers using POST methods [23].

```
@PostMapping("/registration/new-user")
@ResponseBody
public String addNewUserRegistration(@RequestBody UserAccount userAccount){
    List<String> errors = new ArrayList<>();
    errors = userAccountRegistrationErrorCheck(errors, userAccount);
    JSONObject[] userAccountErrorObj = new JSONObject[errors.size()];
    userAccountErrorObj = convertToJSONObject(errors, userAccountErrorObj, 0);
    JSONObject userObj = new JSONObject();
    userObj = convertToJSONArray(errors, userObj, 0, "userAccountMessages", userAccountErrorObj);
    if(userObj.getJSONArray( key: "userAccountMessages").length() == 0){
        String password = userAccount.getPassword();
        String encryptPassword = new BCryptPasswordEncoder().encode(password);
        userAccount.setPassword(encryptPassword);
        userAccountRepository.save(userAccount);
        return userObj.toString();
    }
    return userObj.toString();
}
```

- DELETE: A DELETE must be used to completely remove a resource from a collection.

```
@DeleteMapping("/admin/user")
public void removeUser(@RequestParam int userId){
    userAccountRepository.deleteUserAccountById(userId);
}
```

These principles may seem self-explanatory, however, there are still implementations even from large enterprises who do not apply this practice sufficiently [22]. Implementing endpoints which correctly apply the related HTTP method can improve the ease of consumption of a web service, as it allows consumers to know the outcome from the server request. This allows RESTful web services to be predictable which improves consumption and maintainability [24, 25].

3.3 Use of Sub Resources

As mentioned throughout, RESTful web services are resource driven and therefore should be designed with ease of access in mind. One method of making it easier for consumers is to use sub resources [18]. Sub resources can be thought of as links to parent resources and can be used to allow consumers an easier understanding of how the service is structured. An example of a sub resource structure is shown below.

3.4 HATEOAS

Hypermedia as the Engine of Application State (HATEOAS) is a defining architectural style of a RESTful web service [7, 26]. The principle implements the use of hyperlinks to control the resources the client side application accesses. By applying this style it allows links to other related resources to be used in response objects. This is regarded as a best practice by many as it can improve the consumers' navigation through the developed service [27, 28]. HATEOAS can make a RESTful web service dynamic and responsive [29]. This is achieved through allowing the client to enter the web service through a URI and navigate to discover various resources without prior knowledge to the URI schema [30, 31].

Abrams et al [32] believe that HATEOAS is one of the most misunderstood terms in developing RESTful web services. They believe that HATEOAS is one of the most powerful best practises as it can allow flexibility for the server to change without breaking its clients. However, it is believed that HATEOAS is not implemented to the full extent in many systems as it is difficult to implement especially in that of existing systems [32, 33]. Shown below is an example implementation.

```
"links": [  
  {  
    "rel": "self",  
    "href": "http://localhost:8080/user-accounts?page,size,sort",  
    "hreflang": null,  
    "media": null,  
    "title": null,  
    "type": null,  
    "deprecation": null  
  },  
  {  
    "rel": "profile",  
    "href": "http://localhost:8080/profile/user-accounts",  
    "hreflang": null,  
    "media": null,  
    "title": null,  
    "type": null,  
    "deprecation": null  
  },  
  {  
    "rel": "search",  
    "href": "http://localhost:8080/user-accounts/search",  
    "hreflang": null,  
    "media": null,  
    "title": null,  
    "type": null,  
    "deprecation": null  
  }  
],
```

3.5 Paging and Sorting

Another best practice is the use of range-based responses. A range based response is a response in which the consumer can control the number of results which are returned. This is implemented in the form of pagination. Pagination allows the client or server to specify the number of data items returned splitting items up into pages. This allows the client to select how much of the data they wish to view by passing in limit (amount of data) and offset (starting point) parameters [28].

Pagination can be used with filtering and sorting which can narrow down requested data even more. This makes it easier for the client to consume responses and reduces the need for client-side logic [34]. Another benefit of implementing paging, sorting and filtering is it can increase a service's performance. This is achieved as only the information required by the client is sent by the server. It also becomes important for mobile based applications as consumers will only want to display smaller amounts of information, which is where filtering becomes important [18, 28].

```
/**  
 * Get request mapping on endpoint '/auction/listings' which returns auction listings in pages  
 * @param page this is an int value which defines the page the user wants to view  
 * @return A page object of AuctionGroup which holds all entries on that page  
 */  
@RequestMapping("/auction/listings")  
@ResponseBody  
public Page<AuctionGroup> populateAuctionListings(@Param("page") int page){  
    Pageable pageable = PageRequest.of(page, size: 10); //fixed size of page make page request  
    // with page parameter  
    //find auctions which are not expired and add them to page object  
    Page<AuctionGroup> pageObj = auctionGroupRepository.findAuctionGroupsByExpiredIsFalse(pageable);  
    return pageObj;  
}
```

3.6 HTTP Headers

When designing RESTful web services it is recommended that you use HTTP for defining serialization formats. Masse [17] believes that HTTP headers in a complying REST service must make use of and send content type and accept headers. He believes this is important as both the client and server rely on a content type value, to tell them how to process the sequence of bytes in a response body. Accept headers are used to define what language is to be accepted [17].

3.7 Use of an ORM

This recommendation is mainly to do with the way the backend RESTful service is implemented and does not affect the end consumer. Object Relational Mapping is a software tool which maps data from an object oriented environment to a database environment. This is recommended as it can speed up the development process and can eliminate repetitive code being written, complying with DRY (Don't Repeat Yourself) design principle.

3.8 Stateless and Cache

A key principle of RESTful web services is that they are stateless. This means that services should be designed not to hold any of the client's data server side. Every request the client makes should be independent and stand-alone [?].

Allamaraju [35] believes another key principle is of caching. Caching can improve performance and increase service reliability as it can store key information in cache. The problem with cache is that if data is persisted for a period of time it may become out of date. This can lead to the problem of data consistency [21].

3.9 Error Handling

It is important to consider implementing error handling in a RESTful web service. Default stack trace errors are usually hard to read which is not ideal for consumers of the service. It is important to handle HTTP error codes so that a standardized mean of communicating errors to the consumer is made [36]. Although it is important to handle as many error codes as possible there are 10 fundamental HTTP error codes which should be handled. These are shown in the table below (*table 1*).

Table 1: Common HTTP Error Codes [37, 38]

Code	Status	Description
200	OK	Working
201	OK	New resource created
204	OK	Resource deleted
304	Not Modified	Client can use cached data
400	Bad Request	The request made was invalid
401	Not Authorised	The request requires an authorised user
403	Forbidden	The server understood the request, but access is denied
404	Not found	There is no resource behind the URI
422	Unprocessable Entity	Server cannot process the entity
500	Internal Server Error	Generic server error (this response should be avoided)

Error codes should not just be returned in a response with only the HTTP code. They should be returned in a custom error payload. This payload should include information such as the type of error, the HTTP error code and a message of how the error occurred [35]. An example of a custom error code implementation is shown below.

```
//Example of Error Payload
{
  "timestamp": "2019-05-08T19:03:49.724+0000",
  "message": "Username is not found : adam",
  "details": "uri=/user-account/bid-win/user-wins"
}
```

```
@ExceptionHandler(UserNotFoundException.class)
public final ResponseEntity<ErrorDetails> handleUserNotFoundException(UserNotFoundException ex,
                                                                    WebRequest request) {
    ErrorDetails errorDetails = new ErrorDetails(new Date(), ex.getMessage(),
        request.getDescription(b: false));
    return new ResponseEntity<>(errorDetails, HttpStatus.NOT_FOUND);
}
```

3.10 Versioning

Over time web services experience changes in both requirements and technology. To keep up with these changes versioning should be applied to the developed service. Versioning allows changes to be made to a service without affecting or distributing the current consumers.

There are various ways of implementing versioning. One of which is to pass a version parameter into a URI [17]. This is done for each individual resource endpoint. When an endpoint is invoked with a particular version it will comply to that versions operation and schema, thus allowing multiple versions to be accessed.

```
//Example versioning in URI
localhost:8080/user-accounts/4?version=1.1
```

4 Conclusion

RESTful web services are growing due to their scalability and lightweight nature which makes them useful in cloud and micro service environments. A RESTful web service is defined by mainly the following principles, that they are stateless, feature explicit use of HTTP methods, are client-server modelled, are URI and resource driven. Although this is what defines a RESTful service there are many current services developed implementing REST architectures which do not comply fully with some of these principles.

There are many recommended best practices for developers to follow to achieve a RESTful web service. Although there are authors who believe that there is confusion with recommended practices, there are others which have slightly different views. The research literature found, shows that there seems to be some coherence even though there is no real set standard. The discussed best practices all have relevance in achieving a RESTful web service. In many studies exploring whether these practices are achieved in current systems show that the majority have syntactical errors in URIs. This is surprising as REST services are URI driven which is an important feature. All practices are influenced with the consumer in mind to make consumption as easy as possible. In conclusion, it is believed that REST does require more standardization for best practices. Recent studies show that even large organizations do not apply these best practices fully even though they are vital to achieving REST principles.

References

- [1] M. Kalin, *Java Web Services: Up and Running: Up and Running*. O'Reilly Media, 2009.
- [2] S. A. McIlraith and D. L. Martin, "Bringing semantics to web services," *IEEE Intelligent systems*, vol. 18, no. 1, pp. 90–93, 2003.
- [3] P. K. Potti, S. Ahuja, K. Umapathy, and Z. Prodanoff, "Comparing performance of web service interaction styles: Soap vs. rest," in *Proceedings of the Conference on Information Systems Applied Research ISSN*, vol. 2167, 2012, p. 1508.
- [4] S. Mumbaikar, P. Padiya *et al.*, "Web services based on soap and rest principles," *International Journal of Scientific and Research Publications*, vol. 3, no. 5, pp. 1–4, 2013.
- [5] S. Kumari and S. K. Rath, "Performance comparison of soap and rest based web services for enterprise application integration," in *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2015, pp. 1656–1660.
- [6] P. A. Castillo, J. L. Bernier, M. G. Arenas, J. Merelo, and P. Garcia-Sanchez, "Soap vs rest: Comparing a master-slave ga implementation," *arXiv preprint arXiv:1105.4978*, 2011.
- [7] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000, vol. 7.
- [8] J. Purushothaman, *RESTful Java Web Services*. Packt Publishing, 2015. [Online]. Available: <https://books.google.co.uk/books?id=f9ZOCwAAQBAJ>
- [9] R. Pattamsetti, *Distributed Computing in Java 9*. Packt Publishing, 2017.
- [10] M. K. Saad, R. Abed, and H. M. Hamad, "Performance evaluation of restful web services for mobile devices," *Performance Evaluation of RESTful Web Services for Mobile Devices*, 2010.
- [11] A. Rodriguez, "Restful web services: The basics," *IBM developerWorks*, vol. 33, p. 18, 2008.
- [12] C. Pautasso, "Restful web services: principles, patterns, emerging technologies," in *Web Services Foundations*. Springer, 2014, pp. 31–51.
- [13] F. Belqasmi, R. Glitho, and F. Chunyan, "Restful web services for service provisioning in next generation networks: a survey," *IEEE Communications Magazine*, 2011.
- [14] F. Haupt, F. Leymann, A. Scherer, and K. Vukojevic-Haupt, "A framework for the structural analysis of rest apis," in *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2017, pp. 55–58.

- [15] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, “Are restful apis well-designed? detection of their linguistic (anti) patterns,” in *International Conference on Service-Oriented Computing*. Springer, 2015, pp. 171–187.
- [16] T. Fredrich, “Restful service best practices,” *Online*. [http://www.restapitutorial.com/media/RESTful Best Practices-v1](http://www.restapitutorial.com/media/RESTful%20Best%20Practices-v1), vol. 1, 2012.
- [17] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O’Reilly Media, 2011. [Online]. Available: <https://books.google.co.uk/books?id=eABpyTcJNIC>
- [18] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percanella, “Rest apis: a large-scale analysis of compliance with principles and best practices,” in *International Conference on Web Engineering*. Springer, 2016, pp. 21–39.
- [19] X. Feng, J. Shen, and Y. Fan, “Rest: An alternative to rpc for web services architecture,” in *2009 First International Conference on Future Information Networks*. IEEE, 2009, pp. 7–10.
- [20] S. Vinoski, “Restful web services development checklist,” *IEEE Internet Computing*, vol. 12, no. 6, pp. 96–95, 2008.
- [21] J. Edstrom and E. Tilevich, “Reusable and extensible fault tolerance for restful applications,” in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2012, pp. 737–744.
- [22] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, “Are rest apis for cloud computing well-designed? an exploratory study,” in *International Conference on Service-Oriented Computing*. Springer, 2016, pp. 157–170.
- [23] M. Stowe, *Undisturbed REST: A Guide to Designing the Perfect API*. Lulu.com, 2015. [Online]. Available: <https://books.google.co.uk/books?id=Gg0sCgAAQBAJ>
- [24] Z. Krzysztof, *Providing Web Services of Established Quality with the Use of HTTP Requests Scheduling Methods*, ser. Agent and Multi-agent Systems: Technologies and Applications. Springer, 2010.
- [25] P. Adamczyk, P. H Smith, R. Johnson, and H. S. Munawar, “Rest and web services: In theory and in practice,” *REST: From Research to Practice*, pp. 1–20, 06 2011.
- [26] K. R. Page, D. C. De Roure, and K. Martinez, “Rest and linked data: a match made for domain driven development?” in *Proceedings of the Second International Workshop on RESTful Design*. ACM, 2011, pp. 22–25.

- [27] F. Haupt, D. Karastoyanova, F. Leymann, and B. Schroth, "A model-driven approach for rest compliant services," in *2014 IEEE International Conference on Web Services*. IEEE, 2014, pp. 129–136.
- [28] B. Varanasi and S. Belida, *Spring REST*. Apress, 2015.
- [29] F. Haupt, M. Fischer, D. Karastoyanova, F. Leymann, and K. Vukojevic-Haupt, "Service composition for rest," in *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*. IEEE, 2014, pp. 110–119.
- [30] F. Somoza Alonso *et al.*, "Development of a restful api: hateoas & driven api," 2015.
- [31] M. Fischer, "Model-driven code generation for rest apis," Master's thesis, 2015.
- [32] H. Abrams, P. Ignatius, R. Philyaw, and M. Waschke, "Rest and the cloud," *CA Technology Exchange*, vol. 1, no. 3, pp. 4–12, 2011.
- [33] F. Haupt, F. Leymann, and K. Vukojevic-Haupt, "Api governance support through the structural analysis of rest apis," *Computer Science-Research and Development*, vol. 33, no. 3-4, pp. 291–303, 2018.
- [34] B. Mehta, *RESTful Java Patterns and Best Practices*, ser. Community experience distilled. Packt Publishing, 2014. [Online]. Available: <https://books.google.co.uk/books?id=oDaUBAAQBAJ>
- [35] S. Allamaraju, *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. O'Reilly Media, 2010. [Online]. Available: <https://books.google.co.uk/books?id=LDuzpQlVuG4C>
- [36] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, ser. Addison-Wesley signature series. Addison-Wesley, 2011. [Online]. Available: https://books.google.co.uk/books?id=tK3_vB304bEC
- [37] J. Manoj R and A. Chandrasekhar, "An authentication system of web services based on web server log analysis," *International Journal of Engineering and Technology*, vol. 5, pp. 4786–4793, 01 2013.
- [38] S. Jauker. 10 best practices for better restful api. [Online; accessed 2019-04-06]. [Online]. Available: blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/

Bibliography

- [1] J. Cooper, *Java Design Patterns: A Tutorial*. Addison-Wesley, 2000.
- [2] S. Lee, J.-Y. Jo, and Y. Kim, "Method for secure restful web service," in *2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS)*. IEEE, 2015, pp. 77–81.
- [3] D. Riehle and H. Züllighoven, "Understanding and using patterns in software development," *Theory and practice of object systems*, vol. 2, no. 1, pp. 3–13, 1996.
- [4] M. Pasternak, "Abstracting restful web service http between a client and a server," Jan. 5 2016, uS Patent 9,231,819.
- [5] Y. Elshater, P. Martin, and E. Hassanein, "Using design patterns to improve web service performance," in *2015 IEEE International Conference on Services Computing*. IEEE, 2015, pp. 746–749.
- [6] A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: a survey of techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 33, 2016.
- [7] M. Yi, B. Liu, and Y. Zhang, "Browser design and analysis based on java language," in *6th International Conference on Electronic, Mechanical, Information and Management Society*. Atlantis Press, 2016.

Appendix



Part II: Analysis on Serverless Technologies in Web Hosting & the Implementation of the Circuit Breaker Pattern in Modern Web Development

(ACCA7005) Web Services & (ACCA7004) Web Development

by Adam Peter Davies (P147908)
MSc Software Engineering
University of Wales Trinity Saint David

2019

Table of Contents

1	Introduction	1
2	Background Information	1
2.1	Serverless	1
2.1.1	What is Serverless?	2
2.1.2	Function as a Service (FaaS)	2
2.1.3	Event Driven Computation	3
2.2	Circuit Breaker Pattern	3
2.2.1	How does it work ?	3
2.2.2	Implementation	4
3	Discussion	5
3.1	Serverless	5
3.2	Circuit Breaker Pattern	7
4	Conculsion	8
	References	9
	Bibliography	12

Table of Figures

1	Traditional Architecture v Serverless Architecture	2
2	Circuit Breaker State Diagram	4

1 Introduction

Serverless technologies are fast gaining momentum with many enterprises adopting Serverless architectures. With the rise in Serverless technologies, there are many emerging design patterns which are being adopted in modern day web development, one of these is the Circuit Breaker pattern. The purpose of this report is to critically analyse the use of Serverless in web hosting and the use of the Circuit Breaker pattern in modern day web development. The report is structured with firstly a general background, which presents the definitions of Serverless technology and the concepts of the Circuit Breaker design pattern. This is followed by the main body where a discussion is made critically evaluating the previously mentioned areas. All findings and explored opinions are concluded at the end of the report.

2 Background Information

This section of the report provides the background to the use of Serverless technology in web hosting and general principles of the Circuit Breaker pattern.

2.1 Serverless

In the early days of the web, companies had to own the physical hardware required to run a server infrastructure. This is known as a client-server architecture. In this architecture, the server manages and delivers all the resources the client requires. The client consumes and processes these resources [1]. This architectural model is expensive to run and is time-consuming in terms of deployment. Cloud computing was developed to overcome the difficulties traditional distributed infrastructures possess [2]. Cloud-based architectures use servers which are hosted by a service provider and are accessed over the internet. There are many architectures which are branched under cloud infrastructures, one of these is Serverless.

Serverless technology is emerging as the latest captivating approach to deploy and host applications within the cloud [3, 4]. The growth of Serverless technology within IT is expected to rapidly increase from the present to the near future. It is expected that IT spending on Serverless computing will reach \$8 billion per annum by 2021 [5, 6].

2.1.1 What is Serverless?

Serverless is a cloud computing model in which the cloud provider manages and dynamically allocates the server side resources [7, 8]. Fundamentally, this is the idea of adding a layer of abstraction on top of the cloud computing model. Unlike traditional server-client models, it runs off stateless containers with a combination of third-party services, client-side logic and service-hosted remote procedure calls also known as Function as a Service (FaaS). This, in turn, removes the need for continuously running a system where an event-driven computation can be employed [7]. The term Serverless is misleading as the architecture still uses physical servers to operate.

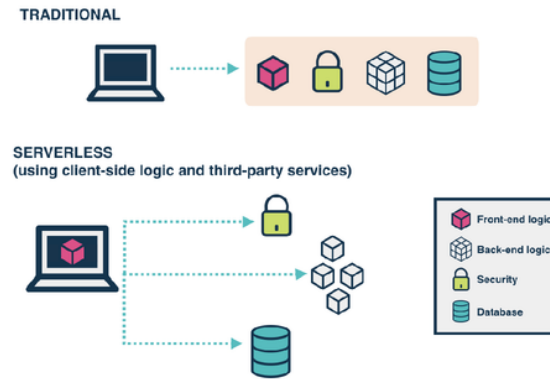


Figure 1: Traditional Architecture v Serverless Architecture [9]

2.1.2 Function as a Service (FaaS)

Function as a Service (FaaS) is the latest category in Serverless and cloud computing platforms. The idea of FaaS is that it allows developers to write functions without thinking or worrying about the infrastructure. Hellerstein et al [10] defines FaaS as the following:

“Traditional programming is based on writing functions, which are mappings from inputs to outputs. Programs consist of compositions of these functions. Hence, one simple way to program the cloud is to allow developers to register functions in the cloud, and compose those functions into programs.”

Functions are triggered via events and are executed on trigger. FaaS providers monitor the run time of these events where they compute a price for running that function. By using the concept of FaaS it allows Serverless providers to offer a *'pay-as-you-go'* scheme [8]. Current providers of FaaS are AWS Lambda, Azure Functions, Google Cloud Functions (GCF) and IBM Cloud Functions.

2.1.3 Event Driven Computation

Event-driven computation is an architecture pattern used commonly in many Serverless applications. The architecture enables systems to be loosely coupled and therefore ensuring services are independent of one another. Additionally, it ensures functions are executed only when invoked and are triggered independently [11]. Serverless applications are built upon multiple functions (FaaS). Event-driven architecture allows these multiple functions to be used coherently.

2.2 Circuit Breaker Pattern

The Circuit Breaker pattern is a software design pattern which is commonly used in modern day web development [12]. The pattern can be thought of in terms of its electrical component counterpart. In electrical systems, a circuit breaker is a form of protection mechanism which interrupts current flow when an overload of current is detected. The software pattern works of the same basic principle as its electrical component counterpart, as it monitors for any failures which may reoccur [13].

There are situations in all aspects of distributed systems where errors occur. Common errors include timeout, temporarily unavailable resources and slow connection faults. The Retry pattern is a commonly used pattern in these situations, as it monitors for faults and tries constantly re-running operations until the error is resolved. However, there are situations where errors occur due to unexpected events. Errors may be as small as a partial loss in connection to complete service failure [14]. For these situations, it is not appropriate to constantly retry to fix the error. This is where the Circuit Breaker pattern is used and is thought of as a quick way of gracefully dealing with unexpected faults in a distributed system. Additionally, it provides a means of not allowing failures to reach out of its boundaries and bring the whole system down [15, 16].

2.2.1 How does it work ?

A Circuit Breaker functions by passing calls and performing transactions to a target service in which the responses are monitored for failures. The number of failures which are received are computed until it reaches a set threshold, which trips and fundamentally breaks the connection between service and client [15]. Any future communication between the client and faulting service will return a failure [12]. The Circuit Breaker pattern is commonly depicted as a finite state machine where it has three states which are outlined as follows:

- Open: Requests are not passed to the intended service. A failure status is sent immediately to the client [14].
- Closed: The requests are processed through to the service. The Circuit Breaker increments a counter for any failure messages received. If the number of failure messages or timeout counter exceeds a set threshold then the system transitions into an open state [15].
- Half-Open: This state only allows a limited number of requests to the intended service. If any of these requests return a successful response then the Circuit Breaker assumes the problem causing the error has been fixed, transitioning to a closed state. If the client receives any failure responses then it transitions to an open state [17].

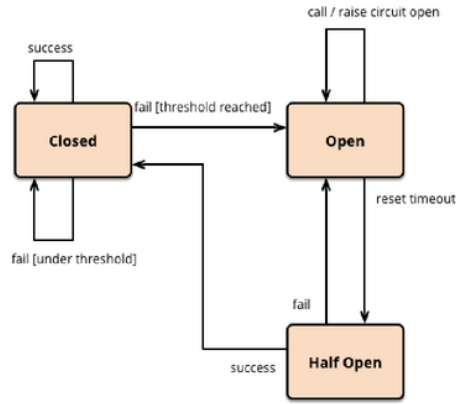


Figure 2: Circuit Breaker State Diagram [18, 19]

2.2.2 Implementation

There are many methods in which the Circuit Breaker pattern can be implemented. There are three main approaches these being client-side, service-side and proxy circuit breakers.

- Client-side: This approach deploys circuit breakers client side, with each client layer having its own breaker. The separated circuit breakers intercept each client call to the various external services.
- Service-side: This approach deploys circuit breakers service side. This approach means that all client requests are processed through an internal circuit breaker.
- Proxy: This approach combines both client and service side circuit breaker implementations. The Circuit Breaker is deployed in between every client and every service which handles all incoming and outgoing requests. Both client and service breakers must be closed for any requests to go through [15].

3 Discussion

This section of the report presents a discussion in analysing Serverless technology in web hosting and the implementation of the Circuit Breaker pattern in modern day web development.

3.1 Serverless

Serverless web hosting technologies bring many advantages to both consumer and provider. One of the main advantages of Serverless architectures is that developers do not need to be concerned about system infrastructure. This is due to the fact that Serverless providers perform all the resource allocation and run the application automatically. It allows developers to concentrate on developing the business logic of a system [4, 11]. To deploy the software all developers have to do is upload it to a service provider, such as Google Cloud and use a related FaaS model. Additionally, it allows developers to develop code in a shorter amount of time over traditional architectures [20].

Another major advantage to consumers is that the cost of deployment is reduced significantly. This is due to the 'pay-as-you-go' schemes which services providers provide through event-driven architectures. By using this consumers of Serverless hosting only get billed when a function is called [10]. By having the ability to dynamically allocate resources on run time, providers allow consumers not to be concerned with peak activity spikes and extra computational power, as it is allocated automatically [21].

This leads on to the scalability benefits of a Serverless architecture. Serverless architecture is hugely scaleable as no matter what load your application requires Serverless providers will ensure the system copes with the load [22, 3]. An example of this would be if a start off business developed a new web service where they designed a traditional system to cope with the estimated load. If suddenly the service kicks off and becomes popular the designed system will need upgrading and modifying to cope with the unforeseen load. This is time-consuming and costly. This estimation of service activity is removed as the service computes how much is required through the activity on deployed functions.

Serverless technologies also provide benefits to end users of a web service. Since there is more time for developers to focus on developing business logic, it means that there is more time to concentrate on user requirements. This can allow it to work well in software developed through an agile methodology, as set requirements do not have to depend on whether the infrastructure can cope with demanding user requirements.

Another advantage to the end user is that of latency due to server locations. As providers allow consumers to deploy in multiple locations, which means that systems with global users can benefit from increased performance capabilities, as they can connect through various points [23].

Serverless also brings advantages to the provider. A large advantage to the provider is the stateless nature of the technology. This stateless nature gives providers more control of the stack in which they can easily maintain and optimize the platforms [4].

As discussed there are many benefits which Serverless technology bring, however, many authors consider the drawbacks which they may provide. Eivy [24] believes that although Serverless brings economical advantages such as scalability and deployment cost reduction, he states that it is important to consider the execution behaviour and workload volumes to gain these advantages [24]. This is only partially true as many providers compensate for this through event-driven architectures [11].

Another area which is discussed by many authors is the security issues Serverless technologies possess. Serverless applications consume a large range of data from various sources and access points. This is a concern as data travels through a variety of functions and third-party sources, which could potentially lead to various access points revealed if an attack took place and makes data more exposed [25].

Due to functions relying on third-party sources and libraries it is difficult to fully test security in an application in a Serverless environment [25]. This again leads to security concerns as full testing coverage cannot be applied.

Baldini et al believe that using FaaS can constrict the development of an application. As they believe many service providers may not have relevant or support up to date libraries for particular languages and frameworks [4]. Although this is true to some extent it is believed that Serverless is heavily invested and languages and frameworks are continuously being updated [21].

3.2 Circuit Breaker Pattern

There are various implementations which can be made of the circuit breaker pattern. With its implementation it brings advantages. One of the main advantages is the fact it can gracefully halt reopening connection to a service. This is a major advantage as the system does not constantly try to reconnect, allowing the communicating service to resolve by reducing the load.

Another benefit of the pattern is as it monitors for system failure, when the service fails you can create custom handling logic. This allows developers to use custom operations, for example, there could be an alternative gateway to an alternative service which the breaker could redirect to. This allows software to not be completely shutdown from one failure.

Another advantage is that of monitoring. The circuit breaker pattern can monitor failures and in turn can log errors. This is important as it can help resolve underlying problems services may have [26].

There are many authors which describe implementing patterns such as the Circuit Breaker, allows applications to be resilient, self-healing and anti-fragile [15]. However, Ibryam [27] believes this is not true. He believes that although a circuit breaker has auto-recovery and self-healing characteristics, he also believes it can only protect and recover from failures related to service interactions [27]. It will not protect from logical failures such as infinite loops, code bugs or memory leaks. To cope with these issues he suggests using live health checks such as Kubernetes monitoring, to monitor the health of deployed containers.

Although this is true it does not mean that the circuit breaker pattern is flawed. It has its job and it does it well, provided it is implemented with considerations for error handling, logging and resets. From this brief analysis the circuit breaker can be seen as widely used in modern day development including within Serverless technologies. By implementing a circuit breaker pattern in a Serverless architecture it can provide an economic advantage by monitoring the status of third party services. If a third party service fails, it can shut off connection to it. This in effect will shut down use of the function running with the provider.

One of the considerations which must be made when implementing a circuit breaker is to wrap individual components in their own breaker. This is done to effectively not share the state of the breaker around. It is considered an anti-pattern if components share the same circuit breaker and is not recommended [28].

Other considerations in the implementation of a circuit breaker are manual override. It is beneficial to provide manual override in systems where recovery time fluctuates greatly for failing operations. Small variations in connectivity can also falsely trigger a circuit breaker to open or close. This is why it is beneficial to have an administrative override to allow a user to reset or set a circuit breaker in the previously mentioned situations [14].

The circuit breaker pattern should not be implemented in either a situation for handling access to local private resources in an application, such as in memory sources. It also should not be used to handle exceptions in business logic. Both of these implementations would add irrelevant overhead to a system [14].

4 Conculsion



With the growth of Serverless technologies in web hosting it is important to have design patterns to protect from failures and provide economic benefits. The main advantage of Serverless technologies is the fact that they are more economical and scalable than traditional architectures. This is due to automatic/dynamic resource allocation and 'pay-as-you-go' schemes.

The problem arises in a Serverless system if there are time out or unexpected errors. As providers charge per function execution and run time if a system constantly tries to reconnect to a third-party service it reduces the effect of this economic advantage. To prevent this the circuit breaker pattern is used. The circuit breaker pattern can improve the economy for applications deployed in a Serverless environment by shutting the system off when a third party service fails or is unresponsive. In conclusion, it is believed that the circuit breaker pattern is one of the most important design considerations when using a variety of third-party services in a Serverless environment.

References

- [1] G. Adzic and R. Chatley, “Serverless computing: economic and architectural impact,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 884–889.
- [2] M.-G. Avram, “Advantages and challenges of adopting cloud computing from an enterprise perspective,” *Procedia Technology*, vol. 12, pp. 529–534, 2014.
- [3] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, “Serverless computing: An investigation of factors influencing microservice performance,” in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 159–169.
- [4] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, “Serverless computing: Current trends and open problems,” in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.
- [5] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uă, and A. Iosup, “Serverless is more: From paas to present cloud computing,” *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.
- [6] marketsandmarkets. Function-as-a-service market by user type (developer-centric and operator-centric), application (web & mobile based, research & academic), service type, deployment model, organization size, industry vertical, and region - global forecast to 2021. [Online; accessed 2019-04-21]. [Online]. Available: www.marketsandmarkets.com/Market-Reports/function-as-a-service-market-127202409.html
- [7] M. Stigler, *Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. Apress, 2017.
- [8] K. Chowhan, *Hands-On Serverless Computing: Build, run and orchestrate serverless applications using AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions*. Packt Publishing, 2018. [Online]. Available: <https://books.google.co.uk/books?id=YzhmDwAAQBAJ>
- [9] R. Weston. Serverless architectures and continuous delivery. [Online; accessed 2019-04-21]. [Online]. Available: www.gocd.org/2017/06/26/serverless-architecture-continuous-delivery/
- [10] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, “Serverless computing: One step forward, two steps back,” *arXiv preprint arXiv:1812.03651*, 2018.

- [11] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 405–410.
- [12] M. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2018.
- [13] O. Michalski and S. Demiliani, *Implementing Azure Cloud Design Patterns: Implement efficient design patterns for data management, high availability, monitoring and other popular patterns on your Azure Cloud*. Packt Publishing, 2018.
- [14] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson, "Cloud design patterns," *Microsoft Corporation*, 2014.
- [15] F. Montesi and J. Weber, "Circuit breakers, discovery, and api gateways in microservices," *arXiv preprint arXiv:1609.05830*, 2016.
- [16] W. Hasselbring, "Software architecture: past, present, future," in *The Essence of Software Engineering*. Springer, Cham, 2018, pp. 169–184.
- [17] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.
- [18] M. Fowler. Circuit breaker. [Online; accessed 2019-04-21]. [Online]. Available: martinfowler.com/bliki/CircuitBreaker.html
- [19] K. Hole, *Anti-fragile Cloud Solutions*, 03 2016, pp. 47–56.
- [20] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: The prospect of serverless scientific computing and hpc," in *Latin American High Performance Computing Conference*. Springer, 2017, pp. 154–168.
- [21] C. Gurturk, *Building Serverless Architectures*. Packt Publishing, 2017.
- [22] D. Zanon, *Building Serverless Web Applications*. Packt Publishing Ltd, 2017.
- [23] H. Kanikathottu, *Serverless Programming Cookbook: Practical solutions to building serverless applications using Java and AWS*. Packt Publishing, 2019.
- [24] A. Eivy, "Be wary of the economics of" serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.
- [25] S. Chaudhary, G. Somani, and R. Buyya, *Research Advances in Cloud Computing*, 03 2018.

- [26] P. Chelliah, S. Naithani, and S. Singh, *Practical Site Reliability Engineering: Automate the process of designing, developing, and delivering highly reliable apps and services with SRE*. Packt Publishing, 2018. [Online]. Available: <https://books.google.co.uk/books?id=ei59DwAAQBAJ>
- [27] B. Ibryam. It takes more than a circuit breaker to create a resilient application. [Online; accessed 2019-04-24]. [Online]. Available: dzone.com/articles/it-takes-more-than-a-circuit-breaker-to-create-a-r
- [28] V. Pacheco, *Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices*. Packt Publishing, 2018.

Bibliography

- [1] N. P. Saraiya, L. M. Matter, and A. Rishi, "Method and system of scaling a cloud computing network," Mar. 24 2015, uS Patent 8,989,187.
- [2] D. Puthal, B. Sahoo, S. Mishra, and S. Swain, "Cloud computing features, issues, and challenges: a big picture," in *2015 International Conference on Computational Intelligence and Networks*. IEEE, 2015, pp. 116–123.
- [3] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2658–2659.
- [4] S. K. Mohanty, G. Premsankar, M. Di Francesco *et al.*, "An evaluation of open source serverless computing frameworks," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2018, pp. 115–120.
- [5] M. Roberts and J. Chapin, *What is Serverless?* O'Reilly Media, Incorporated, 2017.
- [6] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design," *information and software technology*, vol. 82, pp. 1–18, 2017.

FINAL GRADE

95/100

GENERAL COMMENTS

Instructor

Part 1(50%): 46

Part 2 (25%): 24

Part 3 (25%): 25

Part 1

Excellent registration/login system built on Spring Security. Working auction system – client sided end. Bad word detection on last 3 letters. A very good system.

Part 2

Very well written discussion of REST. I would have liked some discussion of design patterns but you did mention ORM. etc.

Part 3

Good, very detailed.

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30



Comment 1

conclusion



Comment 2

Good point!!

PAGE 31

PAGE 32

PAGE 33

PAGE 34
