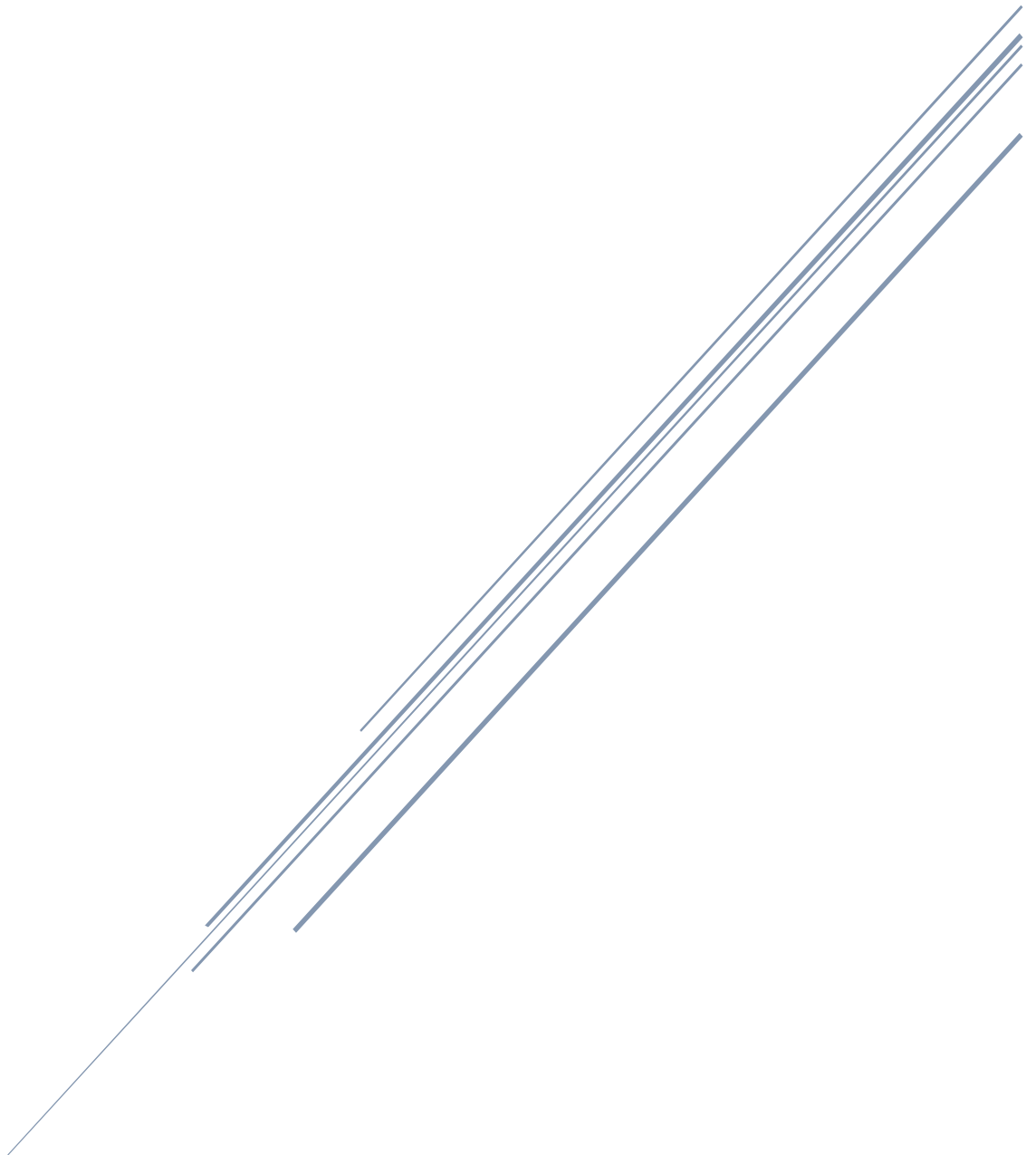


WEB SERVICES AND WEB DEVELOPMENT

Modules ACCA7005 and ACCA7004



1809705

Gomathi Shivakumar

Contents

Introduction.....	3
Constraints	3
Client-Server Model	3
Stateless	3
Layered System.....	3
Cache	4
Uniform Interface	4
Code on Demand	4
Goals for a good REST service:	4
Service Contract.....	4
Resource addressability.....	4
Resources as Nouns:	5
Operations or Methods as Verbs:	5
Misrepresentation of methods:.....	6
Hypermedia as the engine of state(HATEOS).....	7
REST security	9
Conclusion.....	9
Introduction.....	10
Fault Handler.....	10
Deployment Strategies [10]:	11
Advantages:	11
Introduction.....	12
Serverless Computing	12
Time and space	12
The FAAS Market.....	13
Applications	13
Positive impact.....	14
• Functionality	14
• Cost	14
• Programming[16].....	14
• Drawbacks:.....	14
Conclusion.....	14

REST Services

Introduction

A software program that provides its functionality through a technical interface can be termed as a Service. Services provide access to other services or use others to automate tasks and deliver their capabilities. Representational State Transfer (REST) is an implementation method that creates services. It provides a medium with specific characteristics and technology specifications that are specifically suited for services that are distributed hypermedia systems[1]. Roy Fielding describes REST as hybrid web architectural style characterised by its own set of constraints that are combined with other styles[2]. This report talks about the adoption of best practices while designing a REST service.

Web applications leverage REST as an architectural style to deploy robust and scalable programming interfaces APIs[3].

Constraints

Representational state transfer (REST) uses the hypertext transfer protocol (HTTP), and a common exchange format, such as JavaScript Object Notation (JSON) or Extensive Markup Language (XML). It is not possible to talk about practices without the mention of the constraints that collectively define a REST API[3].

Client-Server Model

A client accesses the resources from the server. It is based on separation of concerns. The client is not aware of the implementation details of the server. The User interface is interoperable with different platforms.

Stateless

All information needed for a request from a client to be executed is contained in the request and the response from the service should contain all session state data.[4] Each request is handled in isolation to others.

```
$.ajax({
  type: "Get",
  url: pageurl,
  contentType: "application/json",
  dataType: "json",
  cookie: "sessionid="+u,
}).then(function (data1)
```

The identity of the user, bidding a price is supplied as a cookie with the request so as to update the data store accordingly.

Layered System

Layers can see only its immediate neighbour. Requests hide information regarding the flow of request through the layers allowing for independent development of the layer components. In

this example a layered design has been maintained with no business logic in the model view other than data validation at the client side.

Cache

Response messages are labelled if they are cacheable as true or false. This allows for reuse of previous responses thus improving latency and also allows for certain types of communication as blocked by the stateless nature[2].

Uniform Interface

A uniform technical interface is shared by all consumers and services. The use of HTTP standard facilitates interoperability and platform agnostic services allowing for several resources to be accessed[5].

Code on Demand

This allows for update on logic on either side of the system independent of each other and is optional.

Goals for a good REST service:

Simplicity, Modifiability, Performance, Scalability, Visibility, Portability, Reliability are the design goals of a REST architectural style. Careful application of the constraints to a service will result in these goals being achieved. For instance, caching improves latency performance and simple request-response interaction reduces network overhead. In addition to this, statelessness and layered architecture allows for load balancing and scalability. The uniform contract maintains visibility and portability.

Service Contract

A service contract helps in the consumer understanding the service target and makes it usable. Statelessness influences the contract on the permitted kinds of communication. The contract first approach is considered the best in the design of a REST service as it clearly satisfies the requirements for discovering the resource identifiers, methods, metadata and expected responses. This then gives a clear direction for the developers to ensure that their service meets the contract.

Resource addressability

REST works with managing and exposing resources. Each resource should have a unique identifier, a Uniform Resource Identifier (URI) commonly known as endpoints as the uniform interface[6].

Protocol://Host/Applicationpath/Resourcetype/ResourceID

http://localhost:8080/numberplateproject/users

http://localhost:8080/numberplateproject/batches/1

Cross Origin Policy refers to the policy that web browsers allow transfer of communication between domains of same origins only. This addresses security concerns that relate to illegal hacking of private data being transferred. However, same origin policy restricts open APIs from being reused by a range of consumers. Cross Origin Resource Sharing (CORS) allows javascript to access resources from trusted origins. However, it is only suitable for public APIs in view of the security issues. Same origin relates to same scheme, host and port.

Resources as Nouns:

Resources are either a collection or single entity and the standard practice is to identify the by nouns, especially in plural form.[4] The registration plates are a collection and referred to as numberplates and to find details of a single numberplate, the numberplates/id has been used. Thus, batches represent the different batches, bids represent the collection of all bids placed on all batches and users for registered users/businesses. This satisfies the level 1 criteria of REST by specifying a separate endpoint for each functionality.

```
@PostMapping("/newbids")
```

```
@ResponseBody
```

```
public Bid BidResultGenerator(@RequestBody Bid newBid) {
```

```
@PostMapping("/numberplategen")
```

```
@ResponseBody
```

```
public List<Numberplate> GenerateNumberPlate(@RequestBody  
Regioncode region) {
```

```
function GetBatches(co) {
```

```
$.ajax(  
{  
  cache: false,  
  type: "Get",  
  url: "/batches?sort=auctiondeadline",  
  dataType: "json",  
  contentType: "application/json",
```

All endpoints are nouns with plural for collections such as batches.

Operations or Methods as Verbs:

The standard HTTP methods defined by the verbs relating to common CRUD operations, Create, Read, Update and Delete should be used within the context of their intended meanings.

HTTP method	CRUD operation	
Post	Create	Adds a new user or a numberplate
Get	Read	Reads the details of one or multiple entries in the collection(entity)
Put	Update	Updates one or more entries
Delete	Delete	Deletes one or more entries

```
$.ajax({
  cache: false,
  type: "Post",
  url: "/newbids",
  data: bid,
  contentType: "application/json",
  dataType: "json",
  success: Success,
  error: Error(bid, status)
});
```

```
$.ajax({
  cache: false,
  type: "Post",
  url: "/numberplategen",
  contentType: "application/json",
  data: '{"regioncode":"null","city":"' + $('#city').val() + '"}',
  dataType: "json",
  success: GeneratePlates
});
```

Misrepresentation of methods:

It is recommended that a method should not be used for anything other than its intended action[4]. This means a Get should not be used for adding users. This has implications since, methods are classified as Safe and unsafe. Get and Head are safe. Idempotent methods do not change the state of the resource even if repeatedly requested. Get, Put and Delete are idempotent as opposed to Post. The same request being sent again in these cases would not result in state transition of the resource[4]. But a Post would result with duplicate entries. Hence a Get should never be used for Post. In some cases though where privacy of data is considered, it is in practice to use a Post method to get data, since when querying by a specific attribute reveals the query parameter in the url which is not desirable. However programming own methods into a REST would make the service incompatible with others.

```
$.ajax({
  cache: false,
  type: "Post",
  url: "/users",
  data: user,
  contentType: "application/json",
  dataType: "json",
  success: Success,
  error: Error
});
```

Users as well as Both newbids and numberplategen write data into the database returning JSON objects of the created new entry. Since these create records a POST action is performed returning a 201 code when successful.

Resource representations

Clients only work with representational form of the resources as JSON, XML, text/plain etc.. Hence it is beneficial to explicitly define the representation type so both the client and server clearly know what is being requested and expected.

Headers with status codes and metadata. Request headers specify content type and expected response type in headers. This is a Level 2 compliance[3]

```
responseheaders = new HttpHeaders();  
String value = Integer.toString(user.getUserid());  
responseheaders.set("Set-Cookie", "sessionid=" + value);
```

Hypermedia as the engine of state(HATEOS)

In most cases, resources are interrelated. HATEOS provides for the related resources to be represented as hypertext links. This addresses the stateless nature of sessions. Since no state is managed in the server, it provides a way for the client to navigate through related states by providing the links in the representation[4], [7]. So, if a client requests a batch by id/batches/{id} and then wants to bid on that batch, the hypermedia within the response of the primary request conveys how to bid. A REST API using hypermedia is considered to be at level 3 that spells ease of use and higher quality. A service designed with REST principles and hypermedia enhances the ability to handle changes in the service.[5]

```
$.ajax(  
$.ajax(  
{  
  type: "Get",  
  url: pageurl,  
  contentType: "application/json",  
  dataType: "json",  
  cookie:"sessionid="+u,  
}).then(function (data1) {  
  var hlink = data1._links.regplates.href;  
  
  {  
    type: "Get",  
    url: hlink,  
    contentType: "application/json",  
    dataType: "json"  
  }).then(function (data) {  
    for (var j = 0; j < 10; j++) {  
      $('#reg1').append(  

```

```

    "<p class='plate'>" + data._embedded.numberplate[j].regPlate +
  "</p>")
}

```

In the above code, the pageurl is /batches/{id} which returns a single batch by id that contain a list of regplates. This is represented as a link and fetched lazily. The primary purpose of hypermedia is served by the client not having to construct a url for the regplates but just use the link provided in the response which is assigned to the variable hlink. This also improves performance by not eagerly fetching the entire data but only as and when required[1].

An example of the response to a /users request is shown below containing links to the batches and bids the user has bid for.

```

{
  "userid": 1009,
  "firstname": "Paul",
  "companyname": "Hlghend Motors",
  "addressline1": "Huntly street",
  "addressline2": "",
  "city": "Swansea",
  "postcode": "SA12 5RT",
  "email": "h.motors@email.com",
  "password": "Password1",
  "role": "user",
  "lastName": "Smith",
  "_links": {
    "self": {
      "href": "http://localhost:8080/users/1009"
    },
    "registereduser": {
      "href": "http://localhost:8080/users/1009"
    },
    "batches": {
      "href": "http://localhost:8080/users/1009/batches"
    },
    "bids": {
      "href": "http://localhost:8080/users/1009/bids"
    }
  }
}

```

```
}  
}
```

REST security

User authentication protection against unauthorized access is essential when dealing with databases. One of the many methods available from API keys, openID Connect/OAuth2/SAML (A Security Assertion Markup Language), and session state management can be used[8]. Once the user is authenticated, a cookie is sent back with the response header to be sent back in further request headers.

```
$.ajax(  
{  
  type: "Get",  
  url: pageurl,  
  contentType: "application/json",  
  dataType: "json",  
  cookie: "sessionid="+u,  
}).
```

Conclusion

These best practices address the main design aspects in REST APIs: (i) the modeling of resources, (ii) the identification of resources and the design of resource identifiers (URLs), (iii) the representation of resources, (iv) the definition of (HTTP) operations on resources, and (v) the interlinking of resources[3].

Circuit Breaker Pattern

Introduction

Services in the web need access to multiple services remotely located to provide responses to the client. This is even more important in the case of micro services architectures where they must interact using interface communication processes. REST is one such mechanism that implements remote process invocation [9]. This demands a mechanism for event handling and failure detection to maintain resilience.

Fault Handler

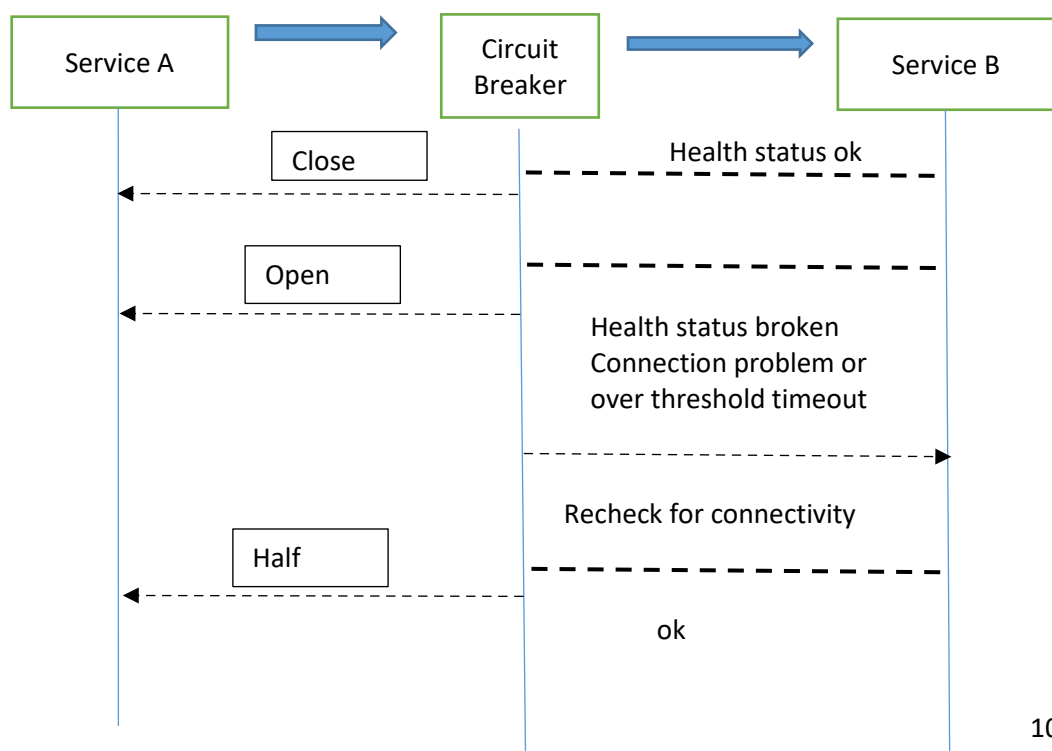
A circuit breaker pattern is similar to a circuit breaker in an electrical system. It is a fail first object that protects the entire system from crashing [10]. When handling requests systems need to collaborate with other services. When a service executes a remote call to another service, if there is a delay due to a failure there is a possibility for the system to crash due to the cascading of failure. A circuit breaker trips the connection after a failure threshold is reached and returns an error message. Fowler says that a circuit breaker monitors for failures. A protected function call is wrapped in a circuit breaker object. At the point of threshold, the circuit is left open and returns an error. The function call is not executed [11]. The health status of services is checked for ok or broken and calls to broken service is prevented. This could also enable the calling service to change its status to broken if it cannot call any other unbroken service. In addition to alerting the broken status, the circuit breaker checks for the recovery of the broken services and tries to reconnect [12].

There are three states that the circuit breaker transfers through

Close State: Normal state. All services are in good health status and remote invocation returns expected response.

Open State: The called service is broken. The circuit is open and returns an error to the calling service.

Half open State: The circuit breaker attempts to check if broken service is recovered and switches to half open. This is to ensure that the called service is able to handle requests again without further failure. The first call is allowed through. Failure results in open state. Success switches to close state.



Parameters control the state transition. For example, monitoring window could be set to 60 seconds with a threshold for number of failed calls to 10% and reset attempt after 20 seconds. Different error types can have different thresholds. Similarly timeout could be set to 30 seconds and often timeouts and executions are dealt with separately[10]. However, these parameters must be considered with the business needs and priorities.

A circuit breaker pattern works effectively when combined with a load balancer. The objective of a load balancer is to establish equal distribution of remote calls. The circuit breaker works with the load balancer and reduces the number of calls to the service with half open status. Butzin et al. state that both the patterns have proven to handle remote call faults very effectively in the Internet of Things[12].

Deployment Strategies [13]:

There are three ways in which Circuit breakers can be deployed.

- Client side - one CB for each client and each service.
- Service side - One CB per service
- Proxy – one CB per client and per service in an intermediary layer

The advantage of the client side circuit breaker is that since no message is received by the target when there is an open state, no implementation is required on the service side for this purpose. However, a client would not be aware of the unavailability of other broken targets and will need to check regularly. A service side circuit breaker does not require access to the client source code but the service still receives the message in open state until it is blocked internally. This suggests slower failure alerts and resource overload. A proxy circuit breaker results in the client and service pointing to the proxy layer with faster fail fasts. The indirect connection makes clients more resilient and the services protected from message overload. No knowledge of source code on the opposite side is required.

Advantages:

Provides system stability. In a distributed system, prevents failures in one component cascading through others. Downtime and slowness is handled efficiently with exception alerts to clients.

Since the circuit breaker prevents calls to broken services, it eliminates wastage of network resources spent on communicating the messages.

The fail fast approach alerts the user to the failure thus preventing them from resubmitting the request causing resource exhaustion. The client is now aware to not waste resource calling a failing service. This could be either due to traffic overload in which case the overloaded services can recover by completing all current requests in the queue[13].

The pattern facilitates interoperability and provides resilience to applications. The ability for any services to be called by the circuit breaker, provides for a layered approach with separation of concerns. The target service can be provided by any vendor.

What makes the breaker valuable is that it also tries to repair by periodically checking the services for recovery, thus making it a self-healing object. This means no additional feature is required to monitor the recovery. The half open state allows for this testing by pinging with a limited number of calls thereby not overloading the target causing further disruption[11].

Monitoring and logging all state changes is valuable to ensure service availability. Testing the states assists in building fault tolerance logic. If a service is unavailable, the response can be fetched from in-memory cache[14].

Aids degrading functionality when the system is under stress. This depends on how the calls are handled at an open state by setting priorities of functions[10].

Circuit breakers are effective at guarding against slow responses, integration points, unbalanced loads and cascading failures[2].

Serverless Technologies

Introduction

Cloud computing is a model where a shared pool of computing resources are rapidly provisioned and released on demand over the internet with minimal management effort or service provider interaction[15]. Characterised by elastic resources, pay as you use and self-service, cloud computing has evolved into three delivery models, Infrastructure as a service(IAAS), Platform as a service(PAAS) and Software as a service. An addition to these models, Serverless architecture takes IAAS to the next level. Microservices and the emergence of serverless technologies have influenced the shift from complex monolith applications to scalable and cost effective cloud based systems[16].

Serverless Computing

Serverless in its true sense is hidden server in the background. It relieves the customers from the burden of provisioning and maintaining infrastructure. Programmers can focus on the application development without worrying about servers[17], [18]. What makes serverless different from IAAS, is that the scaling of infrastructure is dynamic to meet the demands of the application[19][20]. There is no vendor intervention and functionality is provided as a service and language-level abstraction[21].

Function as a Service(FAAS) can be termed as a subset of serverless[22]. Small segments of business logic code or functions are hosted and served on demand, triggered by pre-defined events. Hence, Serverless can be termed as Event driven infrastructure resembling the functional reactive programming due to its stateless nature.

Time and space

Serverless platforms enable cost effective and scalable microservices to be developed. Higher level of abstraction simplifies developing effort and enhances efficient use of the infrastructure. It removes the need for developers to worry about resources and other operational concerns[19]. Small functions that are quick to deploy is highly significant in the case of products that need to meet with client base demands with a quick turnaround time as in the case of online gaming applications. Deployment timeline outweighs optimisation[17]. Applications that involve large volume of high bursts of activities like stock market utilise heavy computation workload at active periods. Even small increases in latency can cause significant repercussions in such cases. Serverless allows for quick upscale of resources and multiple versions of the function to deal with heavy compute thus reducing time[19]. FaaS-hosted components eliminate the need to worry about insufficient memory due to concurrent requests[23]. Consumers only pay for the resources used. With global scaling, the IAAS model cost is per server per centre. With FAAS, the cost is a function of the number of function calls and the runtime[24]. The elasticity nature of automatic scaling up and down means that infrastructure and operational costs are kept optimum with no cost at idle time.

The FAAS Market

Function as a service is deployed at the function level providing logic to connect and extend cloud services. A report produced by Advance Market Analytics estimates that the web based and mobile applications will hold the largest market share in the FAAS market by 2021[25]. The popularity is influenced by easy deployment solutions that optimise performance. Research suggests that microservices deployed exclusively by AWS Lambda per http request, reduces the cost by 77.08%[26]. The market for some of the use cases of serverless which includes, API composition, aggregation, webhooks and event processing is estimated to reach USD7.72 billion by 2021 at an annual growth rate of 32.7%[27]

Applications

Amazon AWS Lambda, IBM Cloud functions, Microsoft Azure functions and Google Cloud functions dominate the serverless providers market. Realtime stream processing[26], cognitive processing, supply chain engineering, Internet of Things and analytics, Mobile backends, Invoice management and several other consumer applications in industries like agriculture, sports, transport news and media are rapidly moving to serverless with success stories[22]. Mike Roberts classifies Backend as a Service (BAAS) serverless. Authentication service like Auth0 is a classic example of integrating authentication functionality as a component with no development time and cost[18].

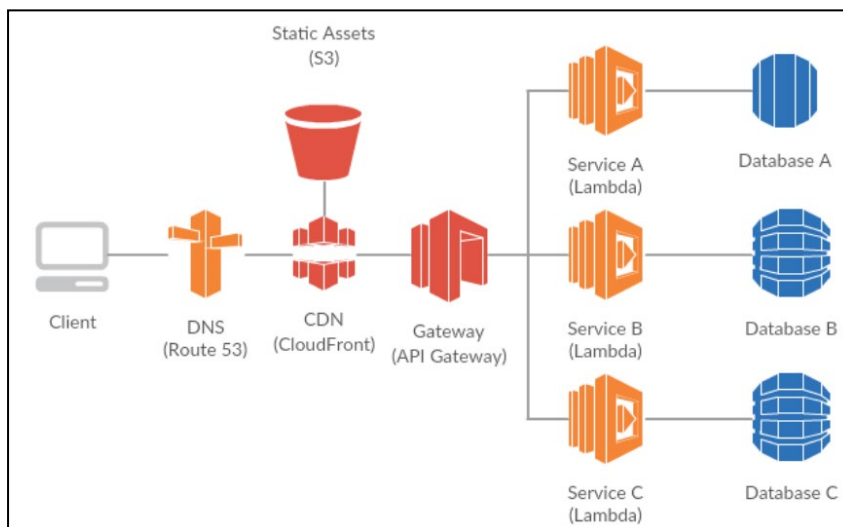


Figure 1:Serverless Framework Integration[28]

Serverless provides for continuous experimentation. The BBC Bitesize implemented a simple view services with API Gateway and Lambda replacing EC2 instances and load balancing. This resulted in easy coding, handling high traffic load with still faster responses, low-cost test environments and delegation of security issues to the provider[29].

Virtual Machines and Containerisation although improved efficiency, have infrastructure constraints. Serverless addresses this with pooled resources including runtime environments and Operating systems[26]. Shoutout, a data driven messaging platform replaced containers with serverless framework. Cost and scalability being their primary objectives, easy command line interface to deploy to multiple regions and roll back and elimination of manual configuration in AWS were achieved with effortlessly combining Node.js with Lambda. [28].

With Rest APIs being the norm for consuming services now, AWS offer BAAS platform for developing and maintaining APIs including authentication. Webhooks and single page applications can be delivered with a combination of serverless API and static S3 bucket content[30] Lambda frontend plugins.

Positive impact

- **Functionality**
 - Zero administration in fully serverless architecture[18]
 - Separation of concerns. Delegation of backend infrastructure, scalability and security responsibilities[16]
 - Better handling of application workload
- **Cost**
 - Server usage to its milliseconds[16]
 - Benefits bursty workloads
 - Auto scaling up and down
- **Programming[19]**
 - Stateless, lightweight and idempotent[21]
 - Consumption of resources only on execution[26]
 - Event based
 - Language agnostic. Parts of applications can be written in any language of choice[21].
- **Drawbacks:**
 - Less control over full application
 - Vendor lock in
 - Hard to debug
 - Cold starts: initial function call takes longer wait for resource allocation
 - The Stateless nature of functions calls for external databases such as DynamoDB and DocumentDB[31]

Conclusion

The benefits of Serverless surpass the drawbacks which could be resolved with further research and evolution. Frameworks such as AWS Serverless Application Model (SAM), Serverless Framework, Stackery, Kubless, Fission and Pulumi enable cross or multi cloud platforms by automating serverless requirements[32][33]. Research suggests that major corporations adopting serverless technologies and succeeding in efficient development and provisioning of their services, play a significant role in the growing trend in the use of serverless[21], [22].

- [1] L. Li and W. Chou, "Design and describe REST API without violating REST: A Petri Net based approach," *Proc. - 2011 IEEE 9th Int. Conf. Web Serv. ICWS 2011*, pp. 508–515, 2011.
- [2] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," no. June, pp. 76–151, 2000.
- [3] R. C. et Al., "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. Lecture Notes in Computer Science," 2016, vol. 9671.
- [4] L. Richardson and S. Ruby, *RESTful Web Services*. 2007.
- [5] Z. Li, X. Li, and M. Xia, "Design and implementation of data access layer based on design pattern," *ICCASM 2010 - 2010 Int. Conf. Comput. Appl. Syst. Model. Proc.*, vol. 14, no. Iccasm, pp. V14-98-V14-100, 2010.
- [6] A. Kobusińska and C. H. Hsu, "Towards increasing reliability of clouds environments with RESTful web services," *Futur. Gener. Comput. Syst.*, vol. 87, pp. 502–513, 2018.
- [7] M. Fowler, "Richardson Maturity Model," *MartinFowler.com*, 2010. [Online]. Available: <https://martinfowler.com/articles/richardsonMaturityModel.html>.
- [8] W. Fu, T. D. Ottavio, and S. Nemesure, "A generic rest api service for control databases *," pp. 465–468.
- [9] C. Richardson, "Circuit Breaker." [Online]. Available: <https://microservices.io/patterns/reliability/circuit-breaker.html>.
- [10] M. NYGARD, *Release It!: Design and Deploy Production-Ready Software. The Pragmatic Bookshelf*, 2007. .
- [11] M. Fowler, "Circuit Breaker."
- [12] B. Butzin, F. Golasowski, and D. Timmermann, "Microservices approach for the internet of things," *IEEE Int. Conf. Emerg. Technol. Fact. Autom. ETFA*, vol. 2016–Novem, 2016.
- [13] F. Montesi and J. Weber, "Circuit Breakers, Discovery, and API Gateways in Microservices," 2016.
- [14] D. TRIVEDI, "Circuit Breakers and MicroServices Arvhitecture," *Constant Contact*, 2017. [Online]. Available: <https://techblog.constantcontact.com/software-development/circuit-breakers-and-microservices/>.
- [15] "Cloud computing: A complete guide tle," *IBM.com*. [Online]. Available: <https://www.ibm.com/cloud/learn/cloud-computing>.
- [16] K. J. P. G. Perera and I. Perera, "A Rule-based System for Automated Generation of Serverless-Microservices Architecture Agile Metrics View project Interactivity of MOOCs View project Indika Perera A Rule-based System for Automated Generation of Serverless-Microservices Architecture," *4th IEEE Int. Symp. Syst. Eng.*, pp. 1–8, 2018.
- [17] N. Savage, "Going serverless," *Commun. ACM*, vol. 61, no. 2, pp. 15–16, 2018.
- [18] M. Roberts, "Serverless Architecture," 2018. .

- [19] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless Programming (Function as a Service)," *Proc. - Int. Conf. Distrib. Comput. Syst.*, pp. 2658–2659, 2017.
- [20] N. Henderson, "Does Going Serverless Save You Money," *Windows IT Pro (online)*, 2017.
- [21] A. Jangda *et al.*, "Formal Foundations of Serverless Computing," 2019.
- [22] M. Sewak and S. Singh, "Winning in the Era of Serverless Computing and Function as a Service," *2018 3rd Int. Conf. Conver. Technol. I2CT 2018*, pp. 1–5, 2018.
- [23] M. Roberts, "Serverless Architecture," 2018. [Online]. Available: <https://martinfowler.com/articles/serverless.html>.
- [24] M. Ast and M. Gaedke, "Self-contained web components through serverless computing," pp. 28–33, 2017.
- [25] M. R. Future, "Function as a Service (FaaS) Market 2018: Global Industry Analysis, Market Size, Share, Trends, Sales Revenue and Comprehensive Research Study Till 2022," *Herald Keeper*, 2018.
- [26] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakarooha, "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms," *Proc. Int. Conf. Cloud Comput. Technol. Sci. CloudCom*, vol. 2017–Decem, pp. 162–169, 2017.
- [27] "Function as a Service Market by User Type, Application, Service Type,etc.," *Market and Market*, 2018. .
- [28] A. Nissanka, "Why We Switched from Docker to Serverless," *Serverless Blog*, 2019. [Online]. Available: <https://serverless.com/blog/why-we-switched-from-docker-to-serverless/>.
- [29] J. Constable, "The Bitesize Serverless Architecture," *BBC Design + Engineering*. [Online]. Available: <https://medium.com/bbc-design-engineering/the-bitesize-serverless-architecture-594e748501e3>.
- [30] S. Infoworld and S. M. Feb, "Build 'em now ! 5 uses for serverless frameworks," pp. 4–7, 2017.
- [31] S. Infoworld and S. M. Jan, "Microsoft Azure Functions locks in on serverless computing," pp. 1–4, 2017.
- [32] E. T. 3-E. D. Guides, "4 Solutions for Serverless Application Deployment: AWS SAM, Stackery, Serverless Framework, and Pulumi," 2019.
- [33] K. Kritikos and P. Skrzypek, "A review of serverless frameworks," *Proc. - 11th IEEE/ACM Int. Conf. Util. Cloud Comput. Companion, UCC Companion 2018*, pp. 161–168, 2019.