

JonesRhysianEssays

by Rhysian Jones

FILE	12010_RHYSIAN_JONES_JONESRHYSIANESSAYS_181292_1512937984.D OCX (1.89M)		
TIME SUBMITTED	01-MAY-2019 03:21PM (UTC+0100)	WORD COUNT	4910
SUBMISSION ID	105986329	CHARACTER COUNT	32809



WEB DEVELOPMENT AND WEB SERVICES PART 2 - REST



Rhysian Jones - 1809710
UWTSD

WEB DEVELOPMENT AND WEB SERVICES PART 2

Rhysian Jones – 1809710

ABSTRACT:

The aim of this report is to analyse the best practice recommendations for implementing RESTful web services by implementing them. The first part of the report will outline key features of a RESTful service and the second part will analyse code snippets against this outline.

CONTENTS:

ABSTRACT:.....	1
REST:	1
RESOURCE ORIENTED ARCHITECTURE:	2
RESOURCES, REPRESENTATIONS, URI:.....	3
HTTP:	4
STATUS CODES:	4
STATES:	4
LINKING (HATEOAS):	5
BUILDING A SERVICE:	6
REFERENCES:	11

FIGURES:

Figure 1: A well-connected web page – every tile is a link	5
Figure 2: A poorly-connected web page - text only, no links	5

REST:

Representational State Transfer is an architectural style for creating web services. This architecture allows systems to communicate over the internet. One system might request a resource held in another system by using REST's operations. These operations are stateless which means that there is no session information kept by the receiver so the operation can be understood without context[1].

RESTful services can be defined by six architectural constraints which are outlined as follows:

Client-server architecture – this deals with the separation of concerns. Separating a system's concerns has many benefits including improving scalability, easing component implementation and allowing for intermediate modules such as firewalls to be introduced without changing the component interfaces. If concerns are separated, individual components can be modified or scaled as needed without needing to alter the whole system[2].

Statelessness – this describes the operations having no session information stored between requests so they need to be understood without context. Each request must contain all necessary information and session information is held only by the client[3].

Cacheability – this concerns responses stating whether they are cacheable to prevent clients from needing to send further requests[4].

Layered system – this describes the use of a layered architecture pattern where components are grouped by function into layers. This can improve scalability as layers can be load balanced individually.

Code on demand – this is an optional constraint which describes client-side scripting (e.g. JavaScript). This allows the service to extend the functionality of a client by transferring code which the client can then execute to receive additional features.

Uniform interface – this is an essential constraint which describes the way all RESTful services communicate using the same operations. The requests must identify individual resources. The request will then return a representation of that resource which can be modified by the client. Every request must contain enough information for the service to understand how to process it. The server provides links dynamically so the client can access all available resources and actions. The links are not hard-coded into the client[5].

In short, REST is a framework which provides a set of predefined operations with which web services can communicate with each other over the internet. It is defined by six constraints which govern how the service should be set up.

RESOURCE ORIENTED ARCHITECTURE:

Resource oriented architecture refers to designing your service around the use of resources[1]. A brief outline of this framework can be summarised as:

- Split your dataset into resources.
- Name all resources.
- Expose the needed HTTP verbs.
- Design accepted representations.
- Design served representations.
- Integrate resources using hypermedia.
- Consider normal flow of events.
- Consider error conditions[1].

RESOURCES, REPRESENTATIONS, URI:

A resource is any item you want to link to. Images, videos, JSON lists, text files, etc. are all resources[1]. Resources are all named. These names are the URIs (Uniform Resource Identifier). Two subtypes of URIs are URNs and URLs. A URN (Uniform Resource Name) is a URI which only specifies the name of the resource and not the address. A URL (Uniform Resource Locator) is a URI which specifies the address of the resource[6].

An example of a URN is *urn:isbn:1-4165-7228-7* which refers to a book by its ISBN number but doesn't tell you how to get it.

An example of a URL is *https://www.uwtsd.ac.uk/staff/tim-bashford/* which refers to a web page containing information about a university lecturer.

Resources are not accessed directly[1]. Instead, representations of the resources are used. A representation is a document in a specified format (e.g. html) which contains information about the resource. If your resource is an object, that object is not sent to the client but rather a representation of that object[6].

Each URI should only refer to one resource but each resource can have more than one URI. Representations can also have their own URIs which is helpful if the resource is passed into other services but representations don't need to be[1]. URIs can be bookmarked, copied and pasted, shared used without having to go through the service's homepage which is referred to as addressability.

A client will access a resource by using the HTTP uniform interface.

HTTP:

HTTP's uniform interface consists of four main HTTP verbs: GET, POST, PUT and DELETE (plus two extras: HEAD and OPTIONS).

A GET request will get the information identified by the URI. This method is idempotent (three GETs in a row will all produce the same result) and it is considered "safe" as it won't change the server state[3].

A HEAD request is a GET request except the message body is not returned in the response. This method is used to get meta-information and is idempotent and "safe".

A POST request will update the identified resource. This is not a "safe" method as it changes the server state and it isn't idempotent as each request will yield a different result.

A PUT request will create a new resource with the given URI. This is also not "safe" but it is idempotent as multiple requests will yield the same result.

A DELETE request will delete the identified resource. This is idempotent but not "safe".

An OPTIONS request will find out the communication options available for the identified resource. This method is idempotent.

GET and POST methods are also cacheable so their responses can be stored[6].

STATUS CODES:

Status codes are the way the service communicates to the client that a request has been successful or unsuccessful in a specified way[6]. In general, a 200("OK") response means that the GET or DELETE request has succeeded and a 201("Created") means a resource has been created. Examples of error codes include 415("Unsupported Media Type"), 400("Bad Request") and 409("Conflict") which can all arise when the representation being sent back by the client is unsuitable in some way[1].

STATES:

RESTful services should be stateless as the server shouldn't store the application state (information about the path the client takes through the application)[1]. The application state stays on the client until it is sent to the server as part of a HTTP request when it is turned into a resource state (information about the resource).

In a stateless application, any application state information the client wants to be considered must be sent as part of a request. This means that authentication info has to be sent with every request because there is no context otherwise for the server to use to infer authorisation[5].

If a service is not stateless, all requests need to be kept together because there is no context without them. This means that you can't just split the requests in half if you need to load balance because you'd need to figure out which requests needed the context of other requests.

In a stateless service, you can easily scale up by adding more servers which don't rely on each other and can function without issue if one stops working.

Statelessness also allows for requests to be re-sent if they time out without having to worry about the session[1].

LINKING (HATEOAS):

Hypermedia as the engine of application state (HATEOAS) refers to request responses containing hypermedia controls or links which the client can use[3][7]. For example, a request to a resource showing a list of student details could also return a hyperlink to the resource that shows the details of the course that student is on. This improves discoverability which means that users can easily tell where they can go and what they can do from each point[8].

If a service doesn't use links, the client will need to build the URIs it wants to visit using predefined rules[1]. This occurs more for web services than on the internet as most people understand it because a human won't understand how to find available resources without links and so won't try.

The image below shows a comparison between a well-connected web page and one that is not very connected at all:

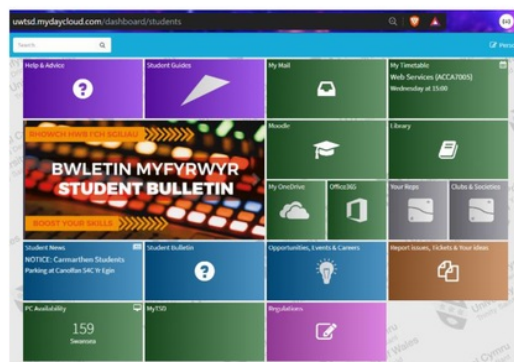


Figure 1: A well-connected web page – every tile is a link [9]



Figure 2: A poorly-connected web page - text only, no links

In the above example, the poorly-connected web service might have resources containing information about each of the films displayed on the home page but there are no links so a human user won't be able to find those resources. A non-human, custom client might know to look for resources and the proper format to find them but it is easier if they're listed for easy access on each resource so the client doesn't need to build URIs from complex rules and so generic clients such as web browsers can access these resources too[10].

Forms can also be used so a user doesn't need to guess the format of a request; the form will do that for them. This is still connected.

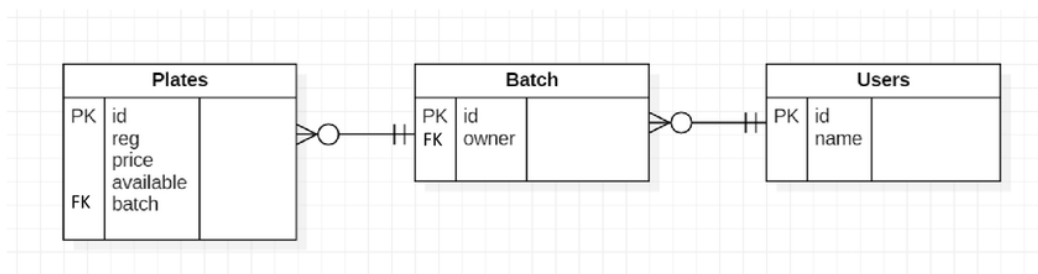
BUILDING A SERVICE:

The RESTful service discussed in this section is a mock number plate auction site. The service consists of a java program which creates number plates, the java package which sets up the repository pattern to connect to the database, a database of plates, a html endpoint UI which allows customers to see the available plates and place bids and some JavaScript to carry out the auction and update the database according to the result.

As discussed above, resource oriented architecture can be broken down into the steps:

- Split dataset into resources,
- Name resources,
- Expose HTTP verbs,
- Design accepted and served representations,
- Hypermedia (links),
- Normal event flow and errors

In this case, the dataset looks like this:



So the resources end up looking like this:

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

@Entity
@JsonIgnoreProperties({ "hibernateLazyInitializer", "handler" })
public class Plates implements java.io.Serializable
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", insertable = false)
    private int id;
    private String reg;
    private double price;
    private boolean available;

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

    //CONSTRUCTORS:
    public Plates()
    {}

    public Plates(int id, String reg, double price, boolean available)
    {
        this.id = id;
        this.reg = reg;
        this.price = price;
        this.available = available;
        //this.batch = batch;
    }

    public Plates(String reg, double price, boolean available)
    {
        this.reg = reg;
        this.price = price;
        this.available = available;
        //this.batch = batch;
    }

    //GETTERS AND SETTERS: REMOVED FOR SIMPLICITY
}
```

This is the Plates class which shows how an entity is set up to be used in conjunction with the repository pattern and hibernate. The corresponding repository interface look like this:

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;
import org.springframework.data.rest.core.annotation.RestResource;
import java.util.List;

@RepositoryRestResource(collectionResourceRel = "plates", path = "plates")
public interface PlatesRepository extends PagingAndSortingRepository<Plates, Integer>
{
}
```

Here, the database table is linked to the resource.

The entity classes and repositories for Users and Batch are laid out similarly.

The HTTP verbs are exposed by the controller:

```
import org.springframework.data.rest.webmvc.RepositoryRestController;
import org.springframework.hateoas.Resource;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.*;

import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;

import java.util.List;

@RepositoryRestController
public class PlatesController
{
    private PlatesRepository repository;

    public PlatesController(PlatesRepository repository)
    {
        this.repository = repository;
    }

    @GetMapping("/plates")
    public List<Plates> all()
    {
        return (List<Plates>) repository.findAll();
    }

    @RequestMapping(value = "/plates/{id}", method = RequestMethod.GET, produces =
    MediaType.APPLICATION_JSON_VALUE)
    Resource<Plates> one(@PathVariable int id)
    {
        Plates plate = repository.findById(id)
            .orElseThrow(() -> new NotFound(id));
        return new Resource<>(plate,
            linkTo(methodOn(PlatesController.class).one(id)).withSelfRel(),
            linkTo(methodOn(PlatesController.class).all()).withRel("plates"));
    }

    @PostMapping("/plates")
    Plates newPlate(@RequestBody Plates newPlate)
    {
        return repository.save(newPlate);
    }
}
```

Here, a repository rest controller is used which takes a repository interface and maps the endpoints. The exposed HTTP verbs are GET and POST.

The controller is also where HATEOAS is implemented to build links between resources. In the `/plates/{id}` mapping, the `linkTo` and `methodOn` functions are used to build these links.

One expected error which is very likely to occur is requesting a plate which doesn't exist. For example, if the system contains two plates with ids 3 and 11 and a request is made at plates/1, the system should return an error containing more information about why this request failed. Here, the controller contains the link to the error here:

```
@RequestMapping(value = "/plates/{id}", method = RequestMethod.GET, produces =
MediaType.APPLICATION_JSON_VALUE)
Resource<Plates> one(@PathVariable int id)
{
    Plates plate = repository.findById(id)
        .orElseThrow() -> new NotFound(id);
    return new Resource<>(plate,
    ...
```

Which calls:

```
class NotFound extends RuntimeException
{
    NotFound(int id)
    {
        super("ERROR: NOT FOUND - id " + id);
    }
}
```

And:

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
class NotFoundAdvice
{
    @ResponseBody
    @ExceptionHandler(NotFound.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    String notFoundHandler(NotFound ex) {
        return ex.getMessage();
    }
}
```

As previously discussed, the six architectural constraints of a RESTful system are:

Client-server architecture (separation of concerns), statelessness, cacheability, a layered system, code on demand and a uniform interface.

The separation of concerns is implemented mostly through the use of the repository pattern. This means that it should be possible to scale up each individual component or introduce new components without altering the existing interfaces.

Statelessness is implemented by having each request contain all the information needed without requiring the server to store session information. In this case, there would be no need to store session information anyway because the system is very simple but in a more complicated system with logins etc., this needs to be accounted for.

Cacheability must be stated in server responses in order to adhere to REST.

The layered system is implemented by using an ORM (hibernate) and the repository pattern but also, in this case, the packages represent groups of functionality. The MakePlates package deals only with generating the number plate objects while the DBPackage package deals with the entity classes, repository interfaces, database connectivity and endpoint mapping.

Code on demand is implemented in this case by using JavaScript to run the auction functionality. Ideally, the auction could be run without JavaScript as well with JavaScript offering optional extra functionality such as cosmetic changes and animations.

The uniform interface (arguably the main point of REST) is implemented by using HTTP verbs to carry out all requests. For example, plates are accessed with /plates/5 or /batch/1/platesList.

These links are not hard coded but are dynamically generated by the system.

So is this service RESTful? – Yes.

Is this service the pinnacle of RESTfulness? – Not even close.

This system could be greatly improved simply by improving the planning. If the resources and representations were better planned, the whole system might make more sense and if the mechanisms used in the whole system were better understood at the beginning of planning, the resources and representations would have been better planned. In short, once you know how to build a RESTful web service, you can properly design a system which will adhere to the REST style while also making sense and being readable. Practice and planning are essential.

REFERENCES:

- [1] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly, 2007.
- [2] J. Badgujar, M. Jailia, and A. Kumar, "Performance metrics of web crawler in client-server and MVC architecture," in *2015 International Conference on Advances in Computer Engineering and Applications*, 2015, pp. 393–398.
- [3] B. Mehta, *RESTful java patterns and best practices : learn best practices to efficiently build scalable, reliable, and maintainable high performance RESTful services*. 2014.
- [4] L. Dewailly, *Building a RESTful web service with Spring : a hands-on guide to building an enterprise-grade, scalable RESTful web service using the Spring framework*. 2015.
- [5] T. M. Nunes, P. P. M. Farias, J. R. V. Dantas, H. A. Lira, B. de A. Muniz, and L. H. S. Felix, "Towards a Distinction between Application State Resources and Data Resources in ROA," in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, 2015, pp. 895–900.
- [6] J. Webber, S. Parastatidis, and I. Robinson, *REST in practice*. O'Reilly Media, 2010.
- [7] J. W. Ng and D. H. Lau, "Going Beyond Web Browsing to Web Tasking: Transforming Web Users from Web Operators to Web Supervisors," in *2013 IEEE Ninth World Congress on Services*, 2013, pp. 126–130.
- [8] R. Oberhauser, "A Hypermedia-Driven Approach for Adapting Processes via Adaptation Processes," in *2015 8th International Conference on Advanced Software Engineering & Its Applications (ASEA)*, 2015, pp. 73–80.
- [9] "UWTSD MyDay - myday." [Online]. Available: <https://uwtsd.mydaycloud.com/dashboard/students>. [Accessed: 01-Apr-2019].
- [10] J. W. Ng, "Adapting REST To REAST, Building Smarter Interactions for Personal Web Tasking," in *2014 IEEE World Congress on Services*, 2014, pp. 38–47.



WEB DEVELOPMENT AND WEB SERVICES PART 2 - REST AND SERVERLESS TECH



Rhysian Jones - 1809710
UWTSD

WEB DEVELOPMENT AND WEB SERVICES PART 3

Rhysian Jones – 1809710

ABSTRACT:

This paper aims to analyse serverless technology and its relevance to web hosting as well as the circuit breaker pattern and its place in web development.

CONTENTS:

ABSTRACT:.....	13
SERVERLESS TECH:	13
-What is it?	13
-Why use it?	14
-Why avoid it?	15
-How to use it:	16
-Web Hosting:	16
CIRCUIT BREAKER PATTERN:.....	17
-What is it?	17
-Why use it?	18
-Why avoid it?	18
-Denial of service:	18
CONCLUSION:	18
REFERENCES:	19

SERVERLESS TECH:

-What is it?

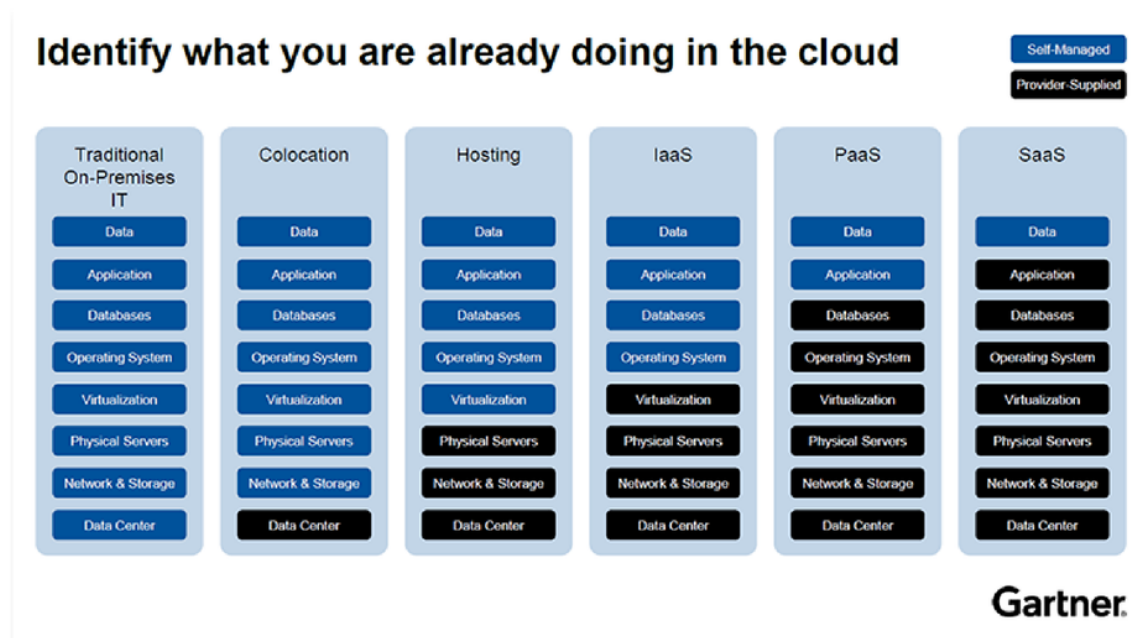
Function as a service (FaaS) describes the practice of outsourcing your infrastructure requirements to a third party company. This company will take over the responsibility of managing the container (server)[1]. One example of a FaaS provider is Amazon Web Services (AWS).

Infrastructure as a service (IaaS) is similar to FaaS but the third party provides infrastructure, virtual machines, file storage, IP addresses and other resources. An example of a provider is Amazon Elastic Compute Cloud (EC2).

Platform as a service (PaaS) provides operating systems, coding environments, databases and web servers[2]. An example of a provider is AWS.

Software as a service (SaaS) provides access to software which you don't need to install. An example of a provider is Office 365.

The image below illustrates the differences between the above mentioned technologies by listing the components provided by the third party vs those developed on site (Note: FaaS could be considered to lie at the right hand side of this image, with every component being provided except for the application which is self-managed):



(image)[3]

-Why use it?

If someone else is handling your server, you don't need to have a big server room in your building which takes up space, is expensive and requires maintenance. You also don't need to keep a server room running continuously which is expensive[1].

If you were to host your own server, you would need to design that system's infrastructure, set it up and maintain it. This requires time and expertise you may or may not have. If you pay someone else to do this, you only need to focus on the code, not the infrastructure.

This practice also means that your service is scalable. If you only need to accommodate a small number of users, you pay for a small container. If your service expands to serve a large number of users, you pay more for a larger container.

If you manage your own servers, you need to load balance and scale yourself but because a serverless system is run in a container rather than a physical machine, an increase in load can trigger more containers to be brought down at any time.

If the servers you manage are located in a low availability region, you might experience an outage which downs your whole system but if your infrastructure is distributed (which it will be if you are using a third party cloud provider as they usually have multiple server hubs in various areas of availability) and one server goes down, the others can pick up the slack and your container can be run on an available server so your system is never unavailable[2].

In short, the benefits of a serverless system include faster development, lower cost, improved scalability, improved availability and removing the need to build and maintain a server room.

-Why avoid it?

If someone else is in full control of your infrastructure, you are not in control of it. For example, if you run a search engine but don't want to log users' location data, you would have trouble avoiding logging this data because it is usually a built in function of the servers[4]. You would need to host your own server in order to be able to control this.

Most providers have a timeout period of a few minutes to avoid long-running processes[5]. This means that serverless architecture will only work if your system needs to be fast and responsive. If your system uses large batch operations, this will not work for you. For example, if you are using your system to compile a large amount of data for scientific purposes, you might send through a large batch of calculations. This can typically take anywhere from half an hour to a few days to complete depending on the complexity and number of calculations. You could not complete this task using cloud computing. In fact, you might consider using a supercomputer such as Supercomputing Wales's (previously HPC Wales) facilities.

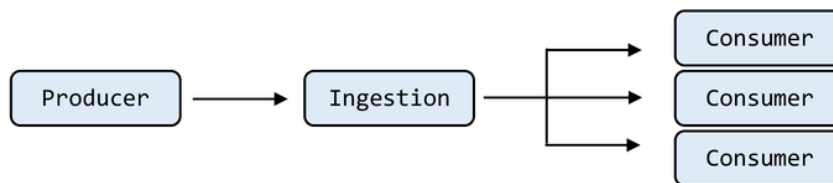
If your system is inactive for a while and then you make a request, the system will experience a cold start and take slightly longer to respond[2]. If you have an infrequently used system which still needs to be immediately responsive, this is a problem. You will need to use a scheduler to periodically trigger your functions and wake them up.

-How to use it:

Serverless applications should be built modularly, with each function being independent of the others. Each function should perform one task and represent one API method[1]. If the architecture is monolithic, functions are not separate. This causes problems because the system becomes a big blob that no-one fully understands, that can't scale up and can't be reused or redeployed. If you instead design your system as a series of microservices made up of individual functions, each component will be independent, clearly defined and reusable. This approach aids in development, deployment and maintenance. A single microservice is easier for other services to consume than a clump of functions which increases efficiency and scalability is helped by the fact that you are able to individually scale microservices as opposed to the entire system at once[2][6]. If your system is made up of multiple microservices or functions, you can deploy each one separately to your cloud server without having to build the whole system. One way to deploy your microservices is through the provider's command line interface[7][8].

Your functions need to consider the events and responses needed. Event-driven collision is a pattern which focuses on events, what they trigger and responses to these events. Functions should only execute when triggered so your system should be loosely coupled.

Event-driven collision can be illustrated by the diagram below:



This shows the path of the events produced by the producer.

-Web Hosting:

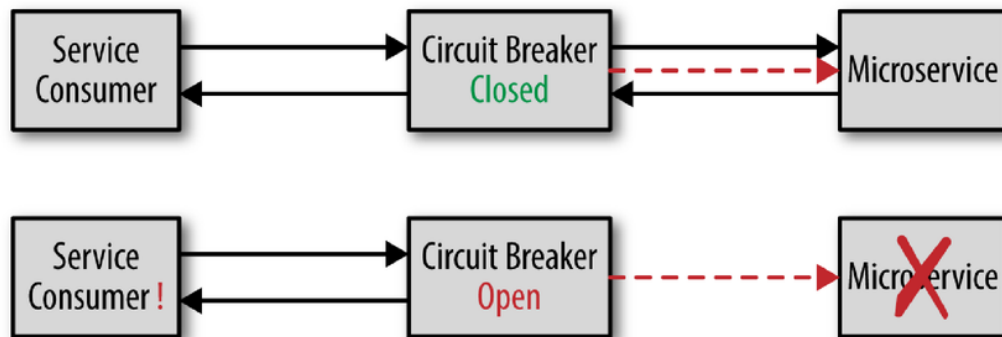
In order to outline the relevance of serverless technologies to web hosting, a real world example can be explored[9].

An application is created which displays news headlines from different outlets. The application consists of a mongodb database, a repository class, an rss feed reader, a class to write to the database, a controller, an application class and an endpoint with a pom file for building and various javascript and html files for the front end. In order to make this service available without users needing to install anything, you would need to use an online database rather than a local one so you need to set up your database on an Atlas server. To deploy the service to the cloud so it is available to users, you can upload your jar file to AWS Elastic Beanstalk (PaaS)[2]. Now, users can access your service without needing to download or install any part of it.

CIRCUIT BREAKER PATTERN:

-What is it?

The circuit breaker pattern is designed to protect services from catastrophic failure in the event that an upstream service fails[10]. It works like a physical circuit breaker where you have a series of trip switches which protect all of your lights from blowing when one does but, in this case, those switches are functions and the lightbulbs are event consumers. A diagram briefly showing this is shown below:



(image)[11]

The circuit breaker can detect that a service isn't responding. If the service isn't responding, the circuit breaker opens its switch (breaking the circuit) which rejects requests going to the unresponsive service. Once the service starts responding again, the switch closes.

The service consumers need to check that the switch is closed before they send requests. If the consumer doesn't know about the circuit breaker, then an interceptor pattern can be used to get around this.

An alternative to the circuit breaker pattern is a timeout pattern where the requests will be sent to the downed service and the consumer will only know that the service is down when the request times out. This has the disadvantage of needing to wait as opposed to immediately knowing a request will fail[12].

There are multiple ways a circuit breaker can monitor a service. One is a heartbeat check which only checks that the service is still running, not if it is responsive to requests. Another is a synthetic transaction sent every few second to see how the service responds to a request. Yet another is real-time monitoring where the transactions are monitored as they interact with the service and if the responsiveness hits a specified threshold, the switch can transition to a half-open state where a fraction of the transactions are allowed through until the service resumes normal responsiveness[13].

-Why use it?

If one of your services stops responding, you want to know. You don't want the consumers to continue to send requests to the service if it has failed. You don't want your request path to be filled with hundreds of failed transactions. You might even want to set up a diversion or an alternative service for these situations. The circuit breaker pattern can tell if your services stop responding and alert you. It can solve the issue of request path crowding. It is probably what you want if your system contains any fragile services or is susceptible to cascading failure[12].

-Why avoid it?

If you add a step in your request path, those requests will probably experience a slight decrease in performance[10].

If your system falls under stress, the circuit breaker will stop some functionality. If this is a problem, the circuit breaker might not be the correct solution[12].

-Denial of service:

Some applications use the retry pattern which retries failed requests until they eventually get a response. This is susceptible to overloading the system with requests which fail and get resent and fail again which means a huge number of requests is in the path all at once. If the system fails, this is a DoS attack.

The retry pattern and the circuit breaker pattern can be used together to result in a system which only resends requests if there are no failed requests still bouncing around[10][13].

CONCLUSION:

In regard to serverless technology, there is certainly a place for it in web hosting for most applications. There are some fringe instances where self-hosting would allow for the necessary fine control such as very high security applications and some scientific applications but for the average weather app, music streaming service or blog, serverless' advantages far outweigh its costs.

For the circuit breaker pattern, it is a highly useful tool if your application is likely to suffer individual service failures, cascading failure or DoS attacks. In fact, there are very few instances where an application wouldn't benefit from this pattern.

REFERENCES:

- [1] M. Stigler, *Beginning serverless computing : developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. Apress L.P, 2018.
- [2] K. Chowhan, *Hands-on serverless computing : build, run, and orchestrate serverless applications using AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions*. Packt Pub, 2018.
- [3] McLellan C, "XaaS: Why 'everything' is now a service | ZDNet," 2017. [Online]. Available: <https://www.zdnet.com/article/xaas-why-everything-is-now-a-service/>. [Accessed: 25-Mar-2019].
- [4] "DuckDuckGo Privacy," 2012. [Online]. Available: <https://duckduckgo.com/privacy>. [Accessed: 25-Mar-2019].
- [5] M. Labouardy, *Hands-On Serverless Applications with Go : Build Real-World, Production-ready Applications with AWS Lambda*. Packt Publishing Ltd, 2018.
- [6] 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)*, 2017, pp. 2658–2659.
- [7] H. Kanikathottu, *Serverless programming cookbook : practical solutions to building serverless applications using Java and AWS*. 2019.
- [8] M. Villamizar *et al.*, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 179–182.
- [9] P. Sbarski, *Serverless architectures on AWS : with examples using AWS Lambda*. 2017.
- [10] H. Subramanian and P. Raj, *Hands-On RESTful API Design Patterns and Best Practices Design, Develop, and Deploy Highly Adaptable, Scalable, and Secure RESTful Web APIs*. Packt Publishing Ltd, 2019.
- [11] M. Richards, "Microservices antipatterns and pitfalls - O'Reilly Media," 2016. [Online]. Available: <https://www.oreilly.com/ideas/microservices-antipatterns-and-pitfalls>. [Accessed: 26-Mar-2019].
- [12] M. T. Nygard, *Release it! : design and deploy production-ready software*. 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 Ltd, 2018.

FINAL GRADE

62/100

GENERAL COMMENTS

Instructor

Part 1(50%): 26

Part 2 (25%): 17

Part 3 (25%): 19

Part 1

Good automatic assigning of prices based on logic. Rudimentary bad language detection. Limited login. Most aspects of the system partially implemented, but a modest way to go to reach a completed system.

Part 2

Generally a good overview of REST, but you should have analysed the "best" practice for developing these services.

Part 3

Very good description of serverless technologies and their importance, but you needed to explain what FaaS is in more detail.

Likewise the CB pattern is well explained

PAGE 1

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



Comment 1

Not really

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20
