

web services

by Yang Zhang

FILE	19082_YANG_ZHANG_WEB_SERVICES_254686_1027454731.DOCX (508.01K)		
TIME SUBMITTED	14-MAY-2020 11:56PM (UTC+0100)	WORD COUNT	4476
SUBMISSION ID	127678789	CHARACTER COUNT	25344

Web Services

Web Development

Student Name: Yang Zhang

StudentNo: 1904077

Contents

Best practice in the implementation of RESTful web services	3
“Serverless” technologies and their relevance to web hosting	10
The circuit breaker pattern and the rationale for its implementation in modern web development	13

Table of Figures

Figure 1: SeverResponse Class	6
Figure 2: addCommodity Class	7
Figure 3: Error	7
Figure 4: ResponseCode Class	8
Figure 5: setHeader function	8
Figure 6: RedisUtil Class	9
Figure 7: LoginController Class	9

1. Best practice in the implementation of RESTful web services

1.1 The introduction of REST and RESTful

Representational state transfer (REST), to be precise, is Resource Representational State Transfer. That is, the state transition of resources in a certain form of expression in the network, which belongs to a method for network design and development. Essentially, it is a way to use URLs to obtain resources. REST is not a standard, but one of the design styles of the framework for building Web services. It combines HTTP, URI, XML, JSON, HTML and other popular standards and protocols to carry out operational design, so that web development has achieved the purpose of lightweight, cross-platform and cross-language, greatly reducing the development complexity And improve the scalability of the system.

REST defines a set of architectural constraints and principles, and applications or designs that satisfy these constraints and principles are RESTful. Restful Web service is a common REST application. It is a web service designed using REST. REST-style Web services are a resource-oriented architecture (ROA). RESTful can be defined in XML format or JSON format based on HTTP.

- Resources

"Resource" is one of the keys to RESTful design. In the RESTful concept, various information (such as text, pictures, audio, and video) can be regarded as resources, and they all correspond to specific URIs (uniform resource locators). The URI provides each resource with its address or unique identifier.

- Presentation layer

For the above-mentioned "resources", developers or customers may require different rendering effects, and a variety of presentation forms can be used. For example, text can be presented in JSON format, XML format, HTML format or even binary format, and pictures can be displayed using PNG or JPG. These representations are called "Presentation layer".

- State transition

Generally speaking, resources are located on the server side. Because the code language written by the client and the server is different, if the client wants to add, delete, modify, and query the resources and data of the server, it is necessary to convert the state of the resource back and forth. This process is "state transition".

Take the HTTP protocol as an example (RESTful is not only applicable to the HTTP protocol, but is often set for the HTTP protocol). The client can modify the resources on the server through some operations. This process is called "state transition of the presentation layer". In HTTP, four common operation methods are provided: GET, POST, PUT and DELETE. These four operation modes correspond to the four basic operations of acquiring resources, creating new resources (which can also be used to update resources), updating resources, and deleting resources.

1.2 RESTful architecture



The RESTful architecture is designed by modifying and improving the MVC architecture. It provides different interfaces to define, can connect different services, improve development efficiency and versatility, and provide higher reliability and consistency for components across multiple services.

In the RESTful architecture, if you want to perform a series of operations on the specified URL resource, the browser can use the following four request methods: POST (add data, you can also

update data), DELETE (delete data), PUT (update data) and GET (view data). Therefore, RESTful manages and views various data and resources based on URI, which has the advantages of strong scalability and clear structure.

The RESTful architecture divides the server into a front-end server and a back-end server. The front-end server delivers the presented interface effect to the browser, that is, the view (excluding the model); the role of the back-end server is to provide an interface to connect to the front-end server. Its workflow is: the browser requests a view from the front-end server and initiates an interface request to obtain the model through the AJAX function contained in the view.

1.3 Design guidelines of REST and RESTful

The REST architecture approach describes six constraints. It was originally proposed by Roy Fielding in his doctoral thesis and defines a RESTful style benchmark.

- Unified interface: connect the interface between the client and server, simplify and separate the frame structure, so that each part can independently reach or view the effect of each part. The following are the four principles of the unified interface:
 1. Based on resources: Different resources need to be uniquely identified by URI. The representation returned to the client and the resource itself are conceptually different. For example, the server does not directly transfer a database resource. However, some HTML, XML, or JSON data can display some database records. Whether to use Finnish to express or UTF-8 encoding depends on the request and the details of the server implementation.
 2. Manipulate resources by representation: When the client receives the representation of the resource containing metadata, the client can operate the resources of the server with certain information under the authority.
 3. Self-describing information: Each piece of information contains enough data to confirm how the information should be handled. For example, different parsers need to be used to confirm the type of network media. The response also indicates their caching capabilities.
 4. Hypermedia is the application state engine (HATEOAS): The client transmits the state through the body content, query string parameters, request header and URI (resource name). The server transmits the status to the client through the body content, response code and response header. This technology is called hypermedia (or hypertext link). In addition to the above, HATEOS also means that links can also be included in the returned body (or head) when necessary to provide a URI to retrieve the object itself or associated objects.

Uniform interface is a necessary criterion for the design of each REST service.

- No status: In REST, statelessness plays an extremely important role. When processing a request, the status change can be included in the request itself, or it can be part of the URI, query string parameter, body, or header. The URL only and represents a resource or an address pointing to a specific resource, and its state details are stored in the main body. After that, through the header, status and response body, the client accepts the related status or resources transmitted by the server. In REST, if you want to keep the user status unchanged under multiple requests, the client needs to include all the client's information to complete the request, and there may be a case of resending the request. Since the server does not need to maintain, update, or transfer session state, statelessness is further expanded. In addition to

this, the load balancer need not worry about conversations with stateless systems. The difference between states and resources is that for the state (application state), the server needs to complete the data required by the request in the current session or request; while for the state (resource state), the resource defines the data characterizing the resource (Like the data information in the database). It can be concluded that the application state is data that changes as the client and the request change. In contrast, the resource state is unchanged for the requesting client. Therefore, try to ensure that the server does not need to maintain application state under multiple requests.

- Bufferable

Usually, the client has a cache effect on the response content of the page. Therefore, the response should be defined as cacheable implicitly or explicitly. If it is not cacheable, you need to pay attention to the old data or dirty data to respond to the client's request. Designing a better-managed cache will partially or completely eliminate the interaction between the client and server, greatly improving performance and scalability.

- C-S architecture

The design of the unified interface allows the client and server to work independently, without some redundant restrictions and communication. For example, the data does not need to be stored on the client, and can be kept inside the server, which allows the client code to be transplanted and used again. The server does not need to consider the user's interface and user status, making the server more simple and easier to expand. Ensure that the interface is not changed, and the server and client can be developed independently.

- Layered system

In general, the client cannot determine whether the connection to the server is direct or indirect. The intermediary server can expand the scalability of the system by enabling load balancing or providing a shared cache. However, when you pay attention to layering, you must also consider the security strategy.

- On-demand coding (optional)

The server can execute logic and send it to the client, which can help it temporarily expand and customize functions. Related examples are compiled component Java applets and client-side script JavaScript.

Following the above principles, consistent with the REST architectural style, allows various distributed hypermedia systems to have the desired natural attributes, such as high performance, scalability, simplicity, variability, visualization, portability and reliability. However, it is worth noting that among the design guidelines in the REST architecture, only On-demand coding is optional. If a service violates any of the other guidelines, strictly speaking, it cannot be called a RESTful style.

1.4 The advantages of RESTful

- Transparency. Resources can be manipulated in the semantics of the http protocol itself, and resources are open to the public.
- Can be designed based on the original semantics of the HTTP protocol.



- No status. When the browser calls the interface data to the server to view, modify and other operations, there is no need to refer to the context and current state, simplifying many cumbersome processes.
- HTTP itself provides a variety of content negotiation methods through business-independent middleware, which can be used to implement concurrent control of caching or resource modification.

1.5 Implement REST framework

1) Unified response structure

The REST framework wants to achieve independent work related to the front end and the back end, so the JSON response structure returned must be the same. Indicates that each REST request will return a uniform structure JSON response structure. In order to map the above JSON response structure in the framework, write a Serverresponse class corresponding to this design:

```
public class ServerResponse<T> implements Serializable {

    private int status;
    private String message;
    private T data;

    private ServerResponse(int status) { this.status =status; }

    private ServerResponse(int status,T data){
        this.status =status;
        this.data=data;
    }

    private ServerResponse(int status,String msg,T data){
        this.status =status;
        this.data=data;
        this.message=msg;
    }

    private ServerResponse(int status, String msg){
        this.status = status;
        this.message = msg;
    }

}
```

Figure 1: SeverResponse Class

This class inherits the parent class of sequencers and has the functions of verification and translation conversion. It can be realized to return a series of values to be displayed in the browser.

2) Data flow in front and back

The browser sends an HTTP request (contains JSON format parameters). After the server receives the request, the JSON parameters are converted into ordinary Java objects before they can be run or modified on the server side; for another example, received on the server side After reaching the data, the data is now a Java object. But because the browser needs a JSON string, the format needs to be converted. The entire process must consider the conversion between HTTP messages and Java objects.

Taking the example of the function of creating a product, directly using the `@ResponseBody` annotation in the method, you can convert the Java object into a specific HTTP message. The following code snippet:

```
@PostMapping("addCommodity")
public ServerResponse addCommodity(@RequestBody FullCommodity fullCommodity) {
    Commodity commodity = new Commodity();
    commodity.setTitle(fullCommodity.getTitle());
    commodity.setDescription(fullCommodity.getDescription());
    String itemId = commodity.getTitle() + "_" + System.currentTimeMillis();
    commodity.setItemId(itemId);
    String skuId = fullCommodity.getSkuId();
    CommodityImages commodityImg = new CommodityImages();
    commodityImg.setItemId(itemId);
    commodityImg.setPicAddr(fullCommodity.getImg());
    commodityImagesService.insert(commodityImg);

    CommodityCategory commodityCate = new CommodityCategory();
    commodityCate.setItemId(itemId);
    commodityCate.setCategoryId(fullCommodity.getType());
    commodityCategoryService.insert(commodityCate);
    List<Commodity> commodities = new ArrayList<>();
    String[] split = skuId.split(",");
    for (String s : split) {
        CommoditySku skuInfo = commoditySkuService.queryById(s);
        skuInfo.setItemId(itemId);
        commoditySkuService.update(skuInfo);
        commodity.setSkuId(s);
        Commodity insert = commodityService.insert(commodity);
        commodities.add(insert);
    }
    return ServerResponse.createBySuccess("success", commodities);
}
```

Figure 2: addCommodity Class

3) Deal with abnormal behavior

In order to analyze and improve some errors that may occur, the reason and status code for displaying the error message are displayed in the browser. Code show as below:

```
public static <T> ServerResponse<T> createByError(){
    return new ServerResponse<T>(ResponseCode.ERROR.getCode(), ResponseCode.ERROR.getDesc());
}

public static <T> ServerResponse<T> createByErrorMessage(String errorMessage){
    return new ServerResponse<T>(ResponseCode.ERROR.getCode(), errorMessage);
}

public static <T> ServerResponse<T> createByErrorCodeMessage(int errorCode, String errorMessage){
    return new ServerResponse<T>(errorCode, errorMessage);
}
```

Figure 3: Error

Among them, this design sets this class as an enumeration, there are four status codes returned: success, error, login, illegal parameters. The above exception analysis is to obtain the status code from this class. Or you can verify the data to avoid illegal parameters.

```
package com.lz.sys.utils;

public enum ResponseCode {

    //成功
    SUCCESS( code: 0, desc: "SUCCESS"),
    //错误
    ERROR( code: 1, desc: "ERROR"),
    //登录
    NEED_LOGIN( code: 10, desc: "NEED_LOGIN"),
    //非法参数
    ILLEGAL_ARGUMENT( code: 2, desc: "ILLEGAL_ARGUMENT");

    private final int code;
    private final String desc;

    ResponseCode(int code, String desc) {
        this.code = code;
        this.desc = desc;
    }

    public int getCode() { return code; }

    public String getDesc() { return desc; }
}
```

Figure 4: ResponseCode Class

4) Solve cross-domain issues

For example, if the front-end application is a static site and deployed in the `http://web.xxx.com` domain, but the back-end application publishes the REST API and is deployed in the `http://api.xxx.com` domain, it needs to be resolved Cross-domain issues. This design is to set the header for the response to achieve cross-domain:

```
private void setHeader(HttpServletRequest request, HttpServletResponse response) {
    //跨域的header设置
    response.setHeader("Access-control-Allow-Origin", request.getHeader("Origin"));
    response.setHeader("Access-Control-Allow-Methods", request.getMethod());
    response.setHeader("Access-Control-Allow-Credentials", "true");
    response.setHeader("Access-Control-Allow-Headers", request.getHeader("Access-Control-Request-Headers"));

    response.setHeader("Content-Type", "application/json;charset=UTF-8");
    response.setStatus(HttpStatus.OK.value());
}
```

Figure 5: setHeader function

5) Provide security mechanism

Because in the process of obtaining services, it may not be able to interact normally. Therefore, this design refers to the RedisUtil class in the login interface, which writes a series of functions such as cache expiration time to provide a security mechanism. When logging in, password

comparison and so on are performed, and corresponding exceptions are thrown if they fail. Upon successful login, a token is generated and stored in the redis database. code show as below:

```
@Component
public final class RedisUtil {
    @Resource
    private RedisTemplate<String, Object> redisTemplate;

    public Set<String> keys(String keys){
        try {
            return redisTemplate.keys(keys);
        } catch (Exception e){
            e.printStackTrace();
            return null;
        }
    }

    public boolean expire(String key, long time) {
        try {
            if (time > 0) {
                redisTemplate.expire(key, time, TimeUnit.SECONDS);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

Figure 6: RedisUtil Class

```
ServerResponseBean serverResponseBean;
if (!currentUser.isAuthenticated()) {
    serverResponseBean = pLogin(currentUser, usernamePasswordToken);
} else {
    currentUser.logout();
    serverResponseBean = pLogin(currentUser, usernamePasswordToken);
}
result = serverResponseBean.getStatus();
System.out.println(result);
if (result.equals(SUCCESS) || result.equals(REPEAT_LOGIN)) {
    String token = "Zhou"+UUID.randomUUID()+"";
    stringRedisTemplate.opsForValue().set(token,serverResponseBean.getData().toString(),Duration.ofMinutes(30L));
    // stringRedisTemplate.opsForValue().set("token",token);
    Map<String,Object> map = new HashMap<>();
    map.put("user",serverResponseBean.getData());
    map.put("token",token);
    return ServerResponse.createBySuccess(serverResponseBean.getMsg(), map);
} else {
    return ServerResponse.createByErrorMessage(serverResponseBean.getMsg());
}
}
```

Figure 7: LoginController Class

2. “Serverless” technologies and their relevance to web hosting

2.1 “Serverless” technologies

2.1.1 The introduction

Serverless refers to reducing the complexity of traditional systems by eliminating the need to run servers and manage infrastructure. It eliminates the need for developers to write redundant code and spends time on the design difficulties and key points. If developers use serverless computing services and utilize a variety of powerful one-time APIs and Web services, it is very helpful to build a loosely coupled, scalable and efficient architecture. In this way, developers do not need to think too much about the server and infrastructure, but focus on the code, which is the ultimate goal of serverless.

Serverless technology is one of the ways to design applications. It is server-side logic implemented by developers running in stateless computing containers. It is triggered by an event and is completely managed by a third party. The true meaning of serverless technology is that it really reduces the requirements and limitations of developers on servers, but this does not mean completely eliminating the relationship with the server, but relying on third-party resources server backend, such as Amazon Web Services (AWS) Lambda, computing service execution code. It contains two different but related areas, namely; BaaS (back-end as a service), FaaS (function as a service).

- BaaS (back-end as a service): Refers to related applications that use third-party applications or services to handle server-side logic. Such applications are based on cloud databases (such as Google Firebase) or authentication services (such as Auth0 or AWS Cognito), and consist of a large number of third-party cloud servers and APIs. Therefore, in the application, the service provider controls the logic and status of the server. For example, our typical single-page application SPA and mobile APP rich client application. Such applications allow the front-end and back-end to communicate with each other based on RestAPI calls, and only need to call the service provider's API to implement corresponding functions, such as general authentication, cloud data / file storage, message push, and application data analysis.
- FaaS(function as a service): Serverless indicates that the development is event-triggered code. Developers can directly deploy service business logic code and run it in a stateless computing container provided by a third party. This architecture is often called FaaS. Developers can only focus on writing business code without considering server issues at all, and code execution is triggered by events. Among them, AWS Lambda is a good example of FaaS implementation.

The serverless application architecture is an application that combines BaaS and FaaS. Users or developers can only focus on the business logic code of the application, write functions to run it on the FaaS platform at a granularity, and integrate with BaaS third-party services to form a complete system. There is no need to consider the server during the whole system.

2.1.2 Difference from traditional model architecture

- Traditional architectural pattern

The traditional architecture model usually uses the C / S architecture. In Web services, the front end sends HTTP requests, and the server accepts the signal and starts processing. The data is pressed before reaching the database, which may pass through multiple application layers, and finally the back end will return a response. The development process is as follows: The first step is to design the page to meet the customer's needs. The second step is to develop and design

the server and front end independently. The third step is to deploy the server after it has been developed. The fourth step is to connect the front end and the back end to debug and run the results. The fifth step is to test and go online. Finally, maintain this. Therefore, in the entire development process, different roles are required to complete different tasks. Originally, the communication cost of each member is high, and many problems of operation and maintenance (such as server load balancing, Huanchun, data redundancy, etc.) need to be considered.

- Serverless architecture

In a serverless architecture, the business logic of an application is to use the FaaS architecture to divide it into multiple independent functional components and API services. In FaaS, the implementation of the back-end functions is divided into functions. Developers only need to write functions and deploy them to serviceless. Because there is no need to consider any operations on the server, the entire process can only be handed over to the front-end engineer, and there is no need to communicate with other engineers to form a communication cost. Front-end engineers generally write programs for back-end services in a serverless environment, the most commonly used is AWS Lambda, because AWS has no restrictions on languages and can be written in languages that are not used.

2.1.3 The advantages

- 1) No management services required - No server requirements and no server maintenance or software installation.
- 2) Flexible scalability - Automatically perform expansion or contraction of the application, or change its resource usage to adjust the corresponding required capacity without adjusting the number of servers.
- 3) High availability - Serverless applications have built-in high availability and fault tolerance. By default, services that run applications automatically provide high availability.
- 4) No extra wear and tear - No need to reserve capacity for computing and storage services. If the code is not running, there is no charge.
- 5) Low cost - There is no need to build servers and infrastructure to save a company a lot of costs (such as operating costs, development costs), and the project can be quickly launched and have security.

Using serverless technology means that developers can spend more time researching product code without having to think too much about cloud or local servers or runtime issues.

2.1.4 The disadvantages

- 1) Not suitable for long-running applications - Because the serverless technology is only run when the request comes, it is "sleeping" at other times. Waiting for the next request, the application will take time to start, this process is called cold start. So serverless architecture is not suitable for applications that run continuously for a long time and need to process a large number of requests.  3
- 2) Completely rely on third-party services - If a third-party service contains all of our application-related services, there may be hidden security risks, so we can choose non-critical APIs or services stored on serverless.

- 3) Lack of debugging and development tools - If problems or errors occur, it is difficult to locate somewhere.
- 4) Cannot be used for high concurrency applications - If you want to start a process for every request, it will cost a lot of money, and when the traffic bursts at a time, it is easy to have a timeout problem.



2.1.5 Suitable for serverless application scenarios

The serverless architecture has the following characteristics: it achieves fine-grained computing resource allocation, does not require pre-allocated resources, has a truly high degree of scalability and flexibility, and is used on demand and billed on demand. Therefore, it is suitable for any event-driven use case. Such as the Internet of Things, mobile applications, Web-based applications and chatbots. The event refers to the user's behaviour (such as pressing a button), the sensor or the data flow in the system.

2.2 The relevance to web hosting

1) Conceptual aspect:

Serverless technology is a simple, efficient, safe, and reliable computing service with elastic and scalable processing capabilities. The management method is simpler and more efficient than physical servers.

Web hosting refers to allocating a certain amount of disk space on a network server. Users can rent this space to place sites and application components to provide necessary data storage and transmission functions. Also known as "web space"

2) Method of operation:

The Web hosting can be directly operated in the control panel. Serverless technology needs to configure its own environment, which mainly includes the three core elements of computing, storage and network.

3) Core products:

Serverless technology mainly includes cloud server rental services for SME users and high-end users and elastic computing platform services for large and medium-sized Internet users. The web hosting is opened from the server, a server can be divided into multiple web hosting.

4) Permission settings:

From the perspective of operation permissions, the permissions of serverless technology will be more abundant, and serverless is equivalent to multiple server connections, with independent IP and bandwidth, you need to install an operating system to configure the operating environment.

5) Security:

Serverless technology is more secure than web hosting, because it has an independent operating system, and the system will automatically perform image backup to prevent downtime and be more stable and safer.

The web hosting is to divide a certain disk space on the network server for users to place sites, application components, etc., and provide the necessary site functions and data storage and transmission functions.

3. The circuit breaker pattern and the rationale for its implementation in modern web development

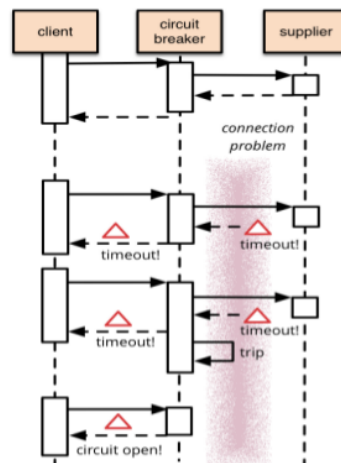
3.1 The introduction and the rationale in modern web development

The circuit breaker pattern belongs to a design mode of modern software development. It encapsulates the services that need to be protected into a circuit breaker object, and investigates whether it is malfunctioning, if the number of failures reached reaches the set threshold. If there is a call to this service later, the circuit breaker will directly return an error to the caller after obtaining this information, and will not execute this service again. So as to achieve the role of protection. At the same time, the circuit breaker mode also plays a good role in the problem of failures that need to be solved for a longer time.

In a distributed environment, especially in a distributed system with a microservice structure, a software system often needs to use another remote system. Since the callee may be another process or another host across the network, it is easy to appear When the call fails. Or hang without any response until it times out. It may even affect the entire distributed system from the timeout of a service and consume a lot of resources of the entire distributed system. Finally, the system is paralyzed. The circuit breaker mode is to prevent this cascading chain reaction.

3.2 Basic structure

The structure of the circuit breaker is as follows:

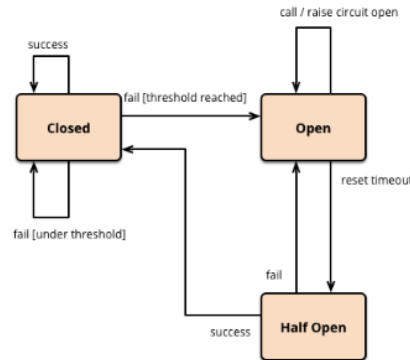


It has two basic states-close and open, and a basic trip action:

- Close state, the service request initiated by the client to the supplier passes through the circuit breaker without errors, and it sends the return value of the supplier to the client.
- Open state, after the service request initiated by the client to the supplier, the circuit breaker disconnects the link between the client and the supplier, and does not pass the request to the supplier, but directly returns the client.
- Trip: In the close state, if the supplier repeatedly reports errors continuously, after the set value is reached, the circuit breaker will trip, and the circuit breaker state will change from close to open.

3.3 The extended mode

In the basic circuit breaker mode, once the circuit breaker is in the open state, the supplier will not be called for protection. But if after the supplier resumes service, you need to reset the circuit breaker. Therefore, an extended design is needed—the circuit breaker continuously detects the service of the supplier, and the detection result is restored, and the state is set to close. The state of the circuit breaker at retry is half-open. The extended structure diagram is as follows:



3.4 Suitable scenarios for circuit breakers

For sudden failures that cannot be efficiently and quickly checked and repaired, this type of service is very suitable for using the circuit breaker mode. Otherwise, the ping-pong effect is likely to occur.

An application calls a remote service or accesses a shared resource. The probability of this operation failing is extremely high, and the power-off mode is also required.

3.5 Scenarios where the circuit breaker is not suitable

If the processing application accesses local dedicated resources, such as data structures in memory. This situation is usually not suitable, the use of circuit breakers will only increase system overhead.

It is also not suitable as an alternative to exception handling in application business logic.

3.6 Problems and considerations

- Exception handling. The circuit breaker invoking service has a situation that the service is abnormal and cannot be accessed, so the application prepares a processing plan in advance.
- The type of exception. There are many reasons for the failure of the request, and some may cause some serious problems.
- Log. Circuit breakers can record failed or successful requests, and administrators can monitor and control the health of operations.
- Recoverability. It is necessary to design its recovery model in advance.
- The operation that failed the test. When the circuit breaker is open, you can use a counter or start a timed task to specify the time for switching the state, and let the remote service determine whether the service is already accessible. In addition to this, it is also a feasible method to recall the service that failed before and to call the special operation provided by the remote service.

References

- [1] Marvin, REST style framework actual combat [Online]. Available: <https://www.cnblogs.com/softidea/p/8144890.html>. [Accessed: Apr.18.2020]
- [2] Xing Li, Detailed RESTful architecture [Online]. Available: <https://www.runoob.com/w3cnote/restful-architecture.html> [Accessed: Apr.20.2020]
- [3] Yin Guan, Research on Web Service Technology Based on REST Architecture, Computer Engineering, 15 Feb 2016 [Accessed: Apr.21.2020]
- [4] Jinya Pan, Research and Application of Spring-based REST Web Service, Xi'an University of Electronic Technology, 20 Feb 2019 [Online]. [Accessed: Apr.25.2020]
- [5] Minna Liu, Suping Huang, Yanxiang Li, Serverless computing: next-generation cloud infrastructure, Computer & Network, pp43-44, 2019 [Accessed: May.02.2020]
- [6] Zhanbing Li, Five tips that serverless technology gods have to say [Online]. Available: <https://forum.huawei.com/enterprise/zh/thread-427687.html> [Accessed: May.03.2020]
- [7] Jeongchul Kim, Kyungyong Lee, The difference between web hosting and server [Online]. Available: <https://www.west.cn/docs/zt18121001> [Accessed: May.05.2020]
- [8] Duanxz, Circuit Breaker Pattern [Online]. Available: <https://recomm.cnblogs.com/blogpost/5183752> [Accessed: May.06.2020]
- [9] Phlsheji, Cloud computing design mode-circuit breaker mode [Online]. Available: <https://www.cnblogs.com/bhlsheji/p/5095844.html> [Accessed: May.06.2020]
- [10] Cunxie Dian, Simple implementation of circuit breaker pattern in Spring [Online]. Available: <https://blog.csdn.net/cunxiedian8614/article/details/105697829> [Accessed: May.07.2020]

FINAL GRADE

72/100

GENERAL COMMENTS

Instructor

Implementation (50): 30
REST (25): 21

Serverless (25): 21

Feedback

Please use referencing.

REST is discussed in good detail but I would have liked more discussion of FaaS and how it works. Benefits of both are also discussed.

Server side has been implemented quite well, the client side appears to be an incomplete Vue application.

PAGE 1

PAGE 2

PAGE 3

 **Comment 1**
Refereces

PAGE 4

PAGE 5



Comment 2

References

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11



Comment 3

The resources are typically left warmed up to process subsequent response to avoid this delay

PAGE 12



Comment 4

A little more explanation required

PAGE 13

PAGE 14

PAGE 15
