My solution:
To make this Article API, I decided to use Gorilla Mux package to implement a request router and dispatcher for matching endpoint request. As required, I used Golang, a powerful language for building a web service.
Follow the instruction, the JSON format of
Coding process:
Step1: Setup the HTTP server using Gorilla Mux: $go get -u github.com/gorilla/mux
Step 2: Building the frame of source code

```go
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "time"
    "strconv"
    "github.com/gorilla/mux" //using gorilla web toolkit
)
func startWebServer(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Welcome home!")
}

func main() {
    router := mux.NewRouter().StrictSlash(true)
    router.HandleFunc("/", startWebServer)
    log.Fatal(http.ListenAndServe(":8080", router))
}
```

Step 3: Create struct of article and tag name information based on JASON format

```go
/*
    // ********************************************************************************
    An article has some attibutes like below example:
    {
        "id":        "1",
        "title":     "latest science shows that potato chips are better for you than sugar",
        "date":      "2016-09-22",
        "body":      "some text, potentially containing simple markup about how potato chips are great",
        "tags":       ["health", "fitness", "science"],
    }
    // ********************************************************************************
*/
type article struct {
    ID          string `json:"id"`
    Title       string `json:"title"`
    Date        string `json:"date"`
    Body        string `json:"body"`
    Tags        []string `json:"tags"`
}
```

```
/*
    // ========================================================================
    An tagEvent has JSON format like below example:
    {
        "tag":              "health",
        "count":            "17",
        "articles":         ["1", "7"],     //list of article id for last 10 articles for that day
        "related_tags":     ["fitness", "science"], //contains a list of tags that are on the articles
    }
    // ========================================================================
*/
type tagEvent struct {
    Tag             string `json:"tag"`
    Count           string `json:"count"`
    Articles        []string `json:"articles"`
    Related_tags    []string `json:"realated_tags"`
}
```

Step 4: Create simple function in generating and getting 1 article

```
// ========================================================================
// POST /articles handles the receipt of article data in json format and store it
func createArticle(w http.ResponseWriter, r *http.Request) {
    var newArticle article
    reqBody, err := ioutil.ReadAll(r.Body)
    if err != nil {
        fmt.Fprintf(w, "Kindly enter data with the event title and description only in order to update")
    }
    err1 := json.Unmarshal(reqBody, &newArticle)
    if err1 !=nil{
        fmt.Println(err1)
    }
    articlesData = append(articlesData, newArticle)
    w.WriteHeader(http.StatusCreated)

    json.NewEncoder(w).Encode(newArticle)
}

// ========================================================================
//GET /articles/{id} returns the JSON representation of the article
func getOneEvent(w http.ResponseWriter, r *http.Request) {
    articleID := mux.Vars(r)["id"]

    for _, singleArticle := range articlesData {
        if singleArticle.ID == articleID {
            json.NewEncoder(w).Encode(singleArticle)
        }
    }
}
```

Step 5: Making the algorithm for last task, which is the final endpoint, GET /tags/{tagName}/{date} will return the list of articles that have that tag name on the given date and some summary data about that tag for that day.
The algorithm I used in this step was: Firstly, get the tagName and convert the date format from url format “/20190212” to “2019-02-12” as the data stored in the service. Then I compare the date with the date of all the articles in dataset. If they match, try to load all the tags the map. The reason I use map because it is fast to find a tag existence with O(1). And if it exists, we simply remove the target tag and copy all other related_tag into ‘tagMap’ (tagMap helps us to prevent the duplicate articles)

```go
func getTagNameOnDate(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    tagName := vars["tagName"]
    tempdate := vars["date"]
    //Convert date from "20160922" to "2016-09-22" to match with article.Date
    t,_ := time.Parse(layoutUS, tempdate)
    tagDate := t.Format(layoutISO)

    //Init count value and declare a tagMap for storing related_tags, preventing duplicate articles
    var tagOnDate tagEvent
    count:= 0
    tagMap := make(map[string]int)
    for _, singleArticle := range articlesData {
        if singleArticle.Date == tagDate {
            mymap:= make(map[string]int)
            for index, singleTag:= range singleArticle.Tags{
                mymap[singleTag]=index
            }
            if _,found := mymap[tagName]; found{
                count++
                if len(tagOnDate.Articles) <10 {
                        tagOnDate.Articles = append(tagOnDate.Articles, singleArticle.ID)
                    } else {
                        tagOnDate.Articles = tagOnDate.Articles[1:]
                        tagOnDate.Articles = append(tagOnDate.Articles, singleArticle.ID)
                    }

                delete(mymap, tagName)
                for otherTag:= range mymap{
                    if _,found := tagMap[otherTag]; found == false {
                        tagMap[otherTag]++
                    }
                }
            }
        }
    }
    tagOnDate.Tag = tagName
    tagOnDate.Count = strconv.Itoa(count)
    for reTag:= range tagMap{
        tagOnDate.Related_tags = append(tagOnDate.Related_tags, reTag)
    }
    json.NewEncoder(w).Encode(tagOnDate)
}
```

What I leant after finishing this project:
It's the first time I made my own API based on these requirements, it's really interesting when applying all the knowdlege I learnt before to fulfill the requests.
Through the coding process, I have gotten used to JSON file format and making full pack of documentation in effective way.

What I need to improve:

1. **Time management**: I thought that the test has the countdown time so I reserved my time for studying the design pattern in making a web service and microservices. That's why I started accessing the test link from 2:30pm Thursday, 7th, 2019, and supposed to finish it before 5pm but I had finished coding and testing only. I needed more time for documentation. I attended the AWS training series in Amazon Sydney Office and continued working with this project from 10:30pm the same day. I should start earlier and prevent the time conflict during the test (at least 1 day because it's the first time I built my own web api)
2. **Investigating time in using new tool:** Because I could not find the test account as mentioned in the email, I tried to test my article API using Postman tool for API intergration testing. The reason why I chose this tool: Postman is a powerful tool and pretty easy to confirm the status, as well as check the data in JSON format.
3. **If I have more time**, firstly I would like to put the error handling to the project. I would like to create a channel to listen for errors coming from the listener, and use a goroutine to listen for the error (I leant it when attending the course "" in GopherAuCon 2019). It's really nice solution

if I can put all of error handling in this project. Secondly, I would like to add the articles into database using sql. Finally, I also would like to apply all the tests for http handler like this post (https://blog.questionable.services/article/testing-http-handlers-go/)