Assignment 6 good

- Modify your Lambda function to include the field City for each new record in theStudents DynamoDB table.
- Modify Read in your Lambda function to return the weather of the city assigned to the student's DynamoDB table record. 30 points.
- Add authorization to your API Gateway API. The only valid user is admin and password abc123!@#. 30 points.
- Read the Test Driven Development is the best thing that has happened to software design article and write a summary and opinions about it. 10 points.

Modify your Lambda function to include the field City for each new record in theStudents DynamoDB table.

To do this, we need a python script that supports the city field about to be created/added, for that, we update our already created lambda function from assignment 5 so it adds this new parameter.

We need to make the .py a zip with the new changes:

```
zip CRUD_TableGaby.zip CRUD_table_gaby.py
```

We update the lambda:

```
aws lambda update-function-code --function-name CRUD_TableGaby --zip-
file fileb://CRUD_TableGaby.zip
```

Now we can test our script, we need to have a JSON file with the parameters we want to insert:

```
aws lambda invoke --function-name CRUD_TableGaby --cli-binary-format
raw-in-base64-out --payload fileb:///Users/gs0522/Documents/CETYS/AWS
/CRUD_functions/\create_example.json response.json
```

To verify that the input was correctly made:

```
aws dynamodb scan --table-name Students
```

Modify Read in your Lambda function to return the weather of the city assigned to the student's DynamoDB table record. 30 points.

As seen in class this is the main weather code will be using:

We use a lambda_handler function to coordinate and manage what instructions will be used from the trigger event, it receives two parameters: an event (the request sent by the user) and a context (gives the handler information about what will be executed based on where the function is located).

It receives an id as the event parameter which will be called *matricula*, which will be searched throughout the database table, first, we call a function that will return us the API key (stored as secret); then we have a variable called weather that stores the information given from the student and the api_key, then we have a succes_response object that stores the parameters of that search; if to works we would get a 200 response, otherwise it will be a 400 error response.

```
def get_weather(student, api_key):
    base_url = "http://api.openweathermap.org/data/2.5/weather?q={0}&appid={1}"
    if "city" in student["Item"].keys():
        http = urllib3.PoolManager()
        response = http.request('GET', base_url.format(student["Item"]["city"], api_key))
        return response.data
    else:
        return json.dumps("No city assigned to student")
```

Then we defined a function called **get_weather()** which retrieves the information from the http://openweathermap.com API and the student's object; if the city exists inside the student's table, we make an HTTP call with **urllib3**, a user-friendly HTTP client for Python into our PoolManager that allows for arbitrary requests while transparently keeping track of necessary connection pools. Then we use the **GET()** method and return the data from that response; if no city was found for that specific student it would return “no city assigned to student”.

```
def get_secret():
    secretsmanager = boto3.client(service_name='secretsmanager')
    secret_name = "weather_api_gaby"
    secrets_response = secretsmanager.get_secret_value(SecretId=secret_name)
    return secrets_response['SecretString']
```

Here we will use a secrets manager service, in order to retrieve the value of a given secret, in this case, the weather API key; we have a secret_name stored in AWS Secrets Manager, in this case, weather_api_gaby that calls the OpenWeatherMap service. In the third line, we get the value of the secret with the name of the second line.

```
def get_response(code, body):
    return {
        "statusCode": code,
        "body": body
    }
```

Finally, we have the **get_response()** method that returns the status code (e.g. 200, 400 …) and a body with the information gathered

---

Once we have the code we can start running the following commands:

```
aws secretsmanager create-secret --name api_key_weather_gabyGood --
secret-string f4edb3afca5c9e19aec9f0210b53735b
```

Here we use AWS **secrets manager** (the name of the service that provides encryption to the API key)**create-secret**, (we choose the name of the secret value we are storing), and the **secret string** (the actual secret that will be encrypted and stored).


```
{
    "ARN": "arn:aws:secretsmanager:us-east-1:292274580527:secret:api_key_weather
_gabyGood-PC4NE1",
    "Name": "api_key_weather_gabyGood",
    "VersionId": "aba55d15-8c93-420a-9715-873b2f302e9c"
}
```


We now need to retrieve the contents of the encrypted fields in my weather API:

```
aws secretsmanager get-secret-value --secret-id api_key_weather_gabyGood
```


```
{
    "ARN": "arn:aws:secretsmanager:us-east-1:292274580527:secret:api_key_weather
_gabyGood-PC4NE1",
    "Name": "api_key_weather_gabyGood",
    "VersionId": "aba55d15-8c93-420a-9715-873b2f302e9c",
    "SecretString": "f4edb3afca5c9e19aec9f0210b53735b",
    "VersionStages": [
        "AWSCURRENT"
    ],
    "CreatedDate": "2023-03-25T21:51:37.683000-06:00"
}
```


Now we have to ZIP the Weather lambda function previously explained:

```
Zip -r weather_lambda_gaby.zip weather_lambda_gaby
```

Now we need to upload/create this weather lambda function inside AWS:

```
aws lambda create-function \
--function-name weather_lambda_gabyGood \
--runtime python3.9 \
--zip-file fileb://weather_lambda_gaby.zip \
--handler weather_lambda_gabyGood.main.lambda_handler \
--role arn:aws:iam::292274580527:role/lambda_ice191
```

```
{
    "FunctionName": "weather_lambda_gabyGood",
    "FunctionArn": "arn:aws:lambda:us-east-1:292274580527:function:weather_lambda_gabyGood",
    "Runtime": "python3.9",
    "Role": "arn:aws:iam::292274580527:role/lambda_ice191",
    "Handler": "weather_lambda_gabyGood.main.lambda_handler",
    "CodeSize": 367793,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2023-03-26T04:00:48.603+0000",
    "CodeSha256": "0qDygy/o4oY2hrrBrho1IVW1CtFR3GCWf/9QLSeunDw=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "063ff1f8-815c-466c-8382-f7eea1b865c1",
    "State": "Pending",
    "StateReason": "The function is being created.",
    "StateReasonCode": "Creating",
    "PackageType": "Zip",
    "Architectures": [
        "x86_64"
    ],
    "EphemeralStorage": {
        "Size": 512
    },
```


Now we can invoke our Lambda function to see what response we get inside my response_gaby.json with a success 200:

```
aws lambda invoke --function-name weather_lambda_gabyGood --cli-binary-format raw-in-base64-out --payload '{"id": "009930"}' response_gaby.json
```


```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
(END)
```


Now we can create the rest-API:

```
aws apigateway create-rest-api --name weather_api_gabyGood
```

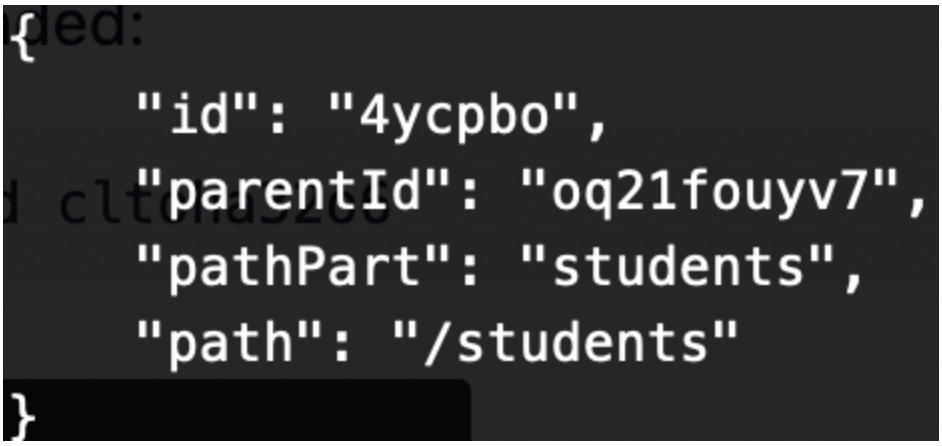

```
{
    "id": "cltoha32o6",
    "name": "weather_api_gabyGood",
    "createdDate": "2023-03-25T22:08:36-06:00",
    "apiKeySource": "HEADER",
    "endpointConfiguration": {
        "types": [
            "EDGE"
        ]
    },
    "disableExecuteApiEndpoint": false
}
```


Now we can see what API call resources were uploaded:

```
aws apigateway get-resources --rest-api-id cltoha32o6
```

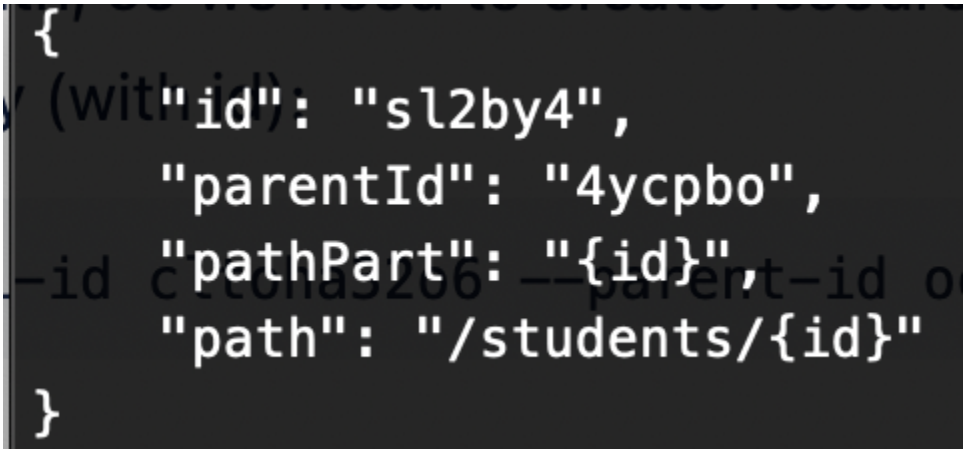
We can see that we only have the id of the main path, so we need to create resources for the Students table and a child to check a specific entry (with id):

```
aws apigateway create-resource --rest-api-id cltoha32o6 --parent-id
oq21fouyv7 --path-part students
```



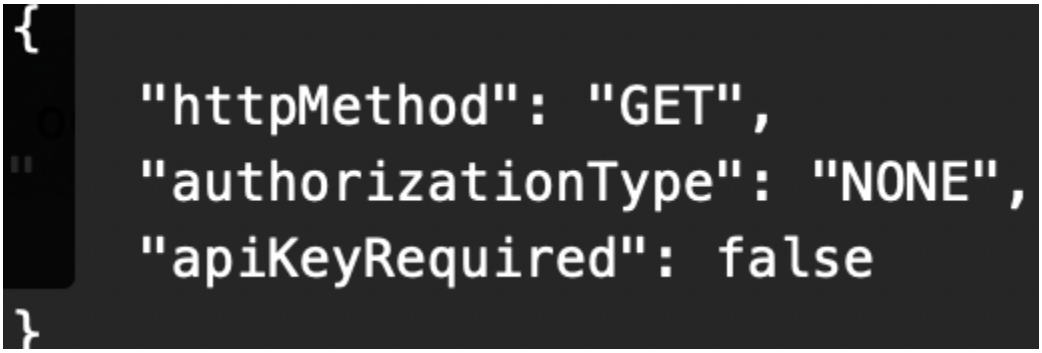
Now we create the child entry for the {id}:

```
aws apigateway create-resource --rest-api-id cltoha32o6 --parent-id
4ycpbo --path-part {id}
```



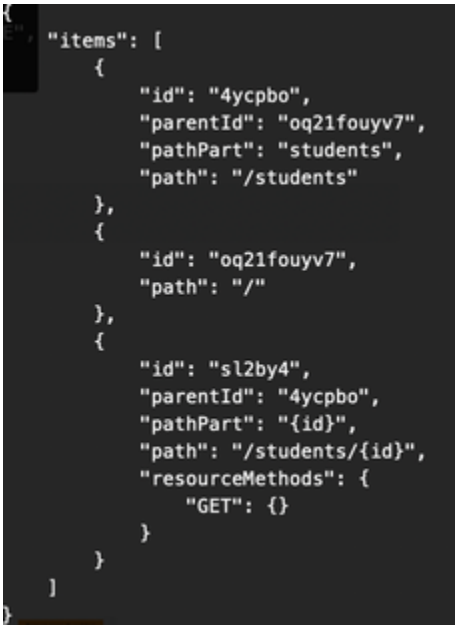
Finally, we create an entry for the GET HTTP method:

```
aws apigateway put-method --rest-api cltoha32o6 --resource-id sl2by4 --
http-method GET --authorization-type NONE
```



So far the resources look like this:

```
aws apigateway get-resources --rest-api-id
cltoha32o6
```



Now we send an AWS integration request, this is an HTTP request that API Gateway submits to the backend, passing along the client-submitted request data, and transforming the data, if necessary:

```
aws apigateway put-integration \
--rest-api-id cltoha32o6 \
--resource-id sl2by4 \
--http-method GET \
--integration-http-method POST \
--type AWS_PROXY \
--uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:
aws:lambda:us-east-1:292274580527:function:weather_lambda_gabyGood
/invocations
```

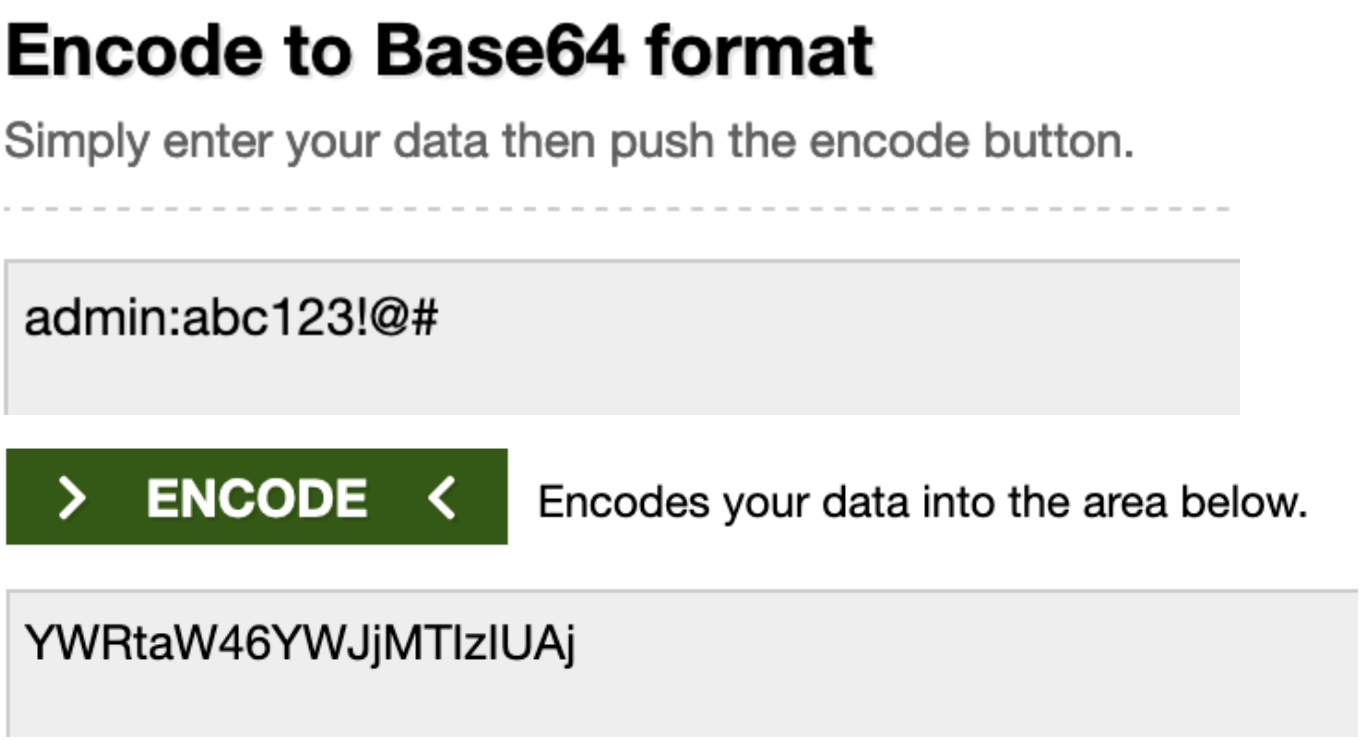
We can do a deployment of our Weather API:

```
aws apigateway create-deployment \
--rest-api-id cltoha32o6 \
--stage-name dev \
--description 'Deploy all the things'
```

Add authorization to your API Gateway API. The only valid user is admin and password abc123!@#. 30 points.

The best option here is to make use of Lambda Authorizer which is an API Gateway feature that uses a Lambda function to control access to my API; when a client makes a request to one of your API's methods, API Gateway calls your Lambda authorizer, which takes the caller's identity as input and returns an IAM policy as output.

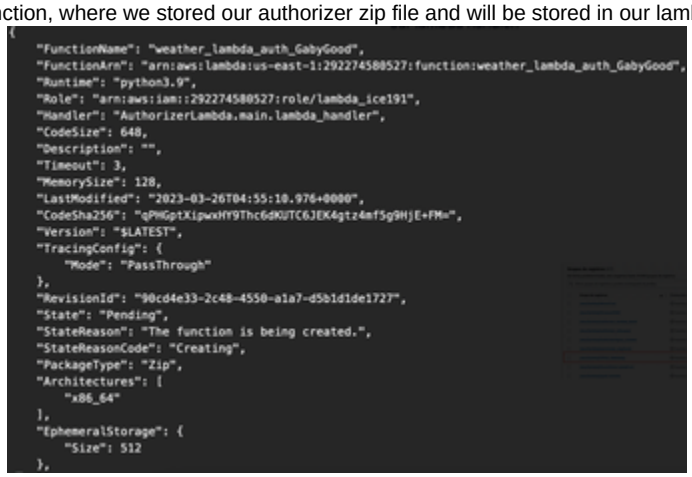
We need to encode our user info:



Now we can run the next command:

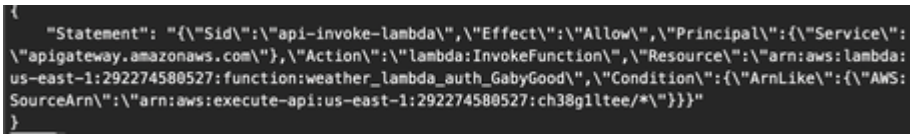
```
aws lambda create-function \
--function-name weather_lambda_auth_GabyGood \
--runtime python3.9 \
--zip-file fileb://authorizer_gaby.zip \
--handler AuthorizerLambda.main.lambda_handler \
--role arn:aws:iam::292274580527:role/lambda_ice191
```

Here we upload the creation of a function, where we stored our authorizer zip file and will be stored in our lambda handler.



Now we use the following command to add permissions:

```
 aws lambda add-permission \
--statement-id api-invoke-lambda \
--action lambda:InvokeFunction \
--function-name weather_lambda_auth_GabyGood \
--principal apigateway.amazonaws.com --source-arn "arn:aws:execute-api:
us-east-1:292274580527:ch38g1ltee/*"
```



`aws lambda add-permission` is used to grant permission to use a function.

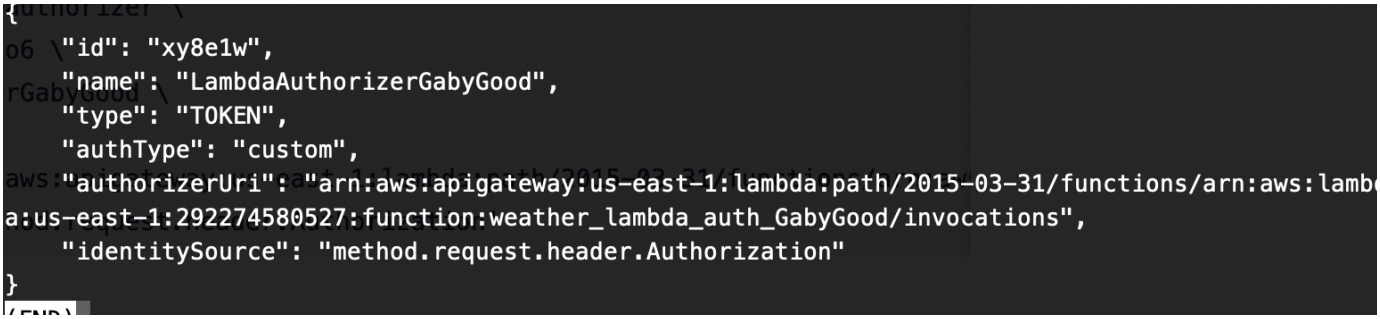
`statement-id` is a statement identifier that differentiates the statement from others in the same policy.

`action` The action that the principal can use on the function, in this case, `lambda:InvokeFunction`.

`function-name` is the name of the lambda that will use this function.

`principal` The AWS service or AWS account that invokes the function.

```
aws apigateway create-authorizer \
--rest-api-id cltoha32o6 \
--name LambdaAuthorizerGabyGood \
--type TOKEN \
--authorizer-uri 'arn:aws:apigateway:us-east-1:lambda:path/2015-03-31
/functions/arn:aws:lambda:us-east-1:292274580527:function:
weather_lambda_auth_GabyGood/invocations' \
--identity-source 'method.request.header.Authorization'
```

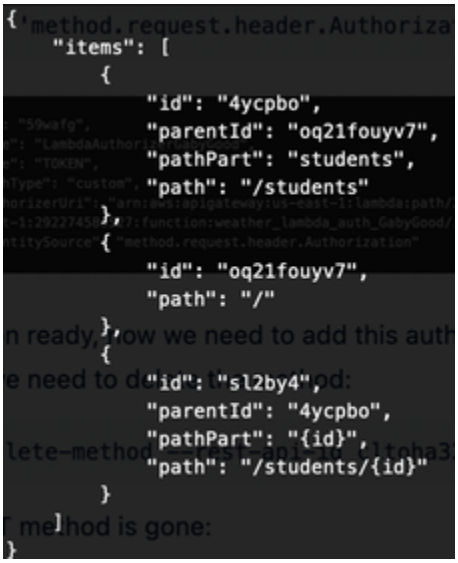


Having this authorization ready, now we need to add this new authorization to our GET method,
which is not possible, we need to delete the method:

```
aws apigateway delete-method --rest-api-id cltoha32o6 --resource-id
sl2by4 --http-method GET
```

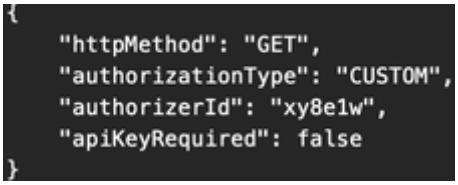
We can confirm the GET method is gone:

```
aws apigateway get-resources --rest-api-id
cltoha32o6
```



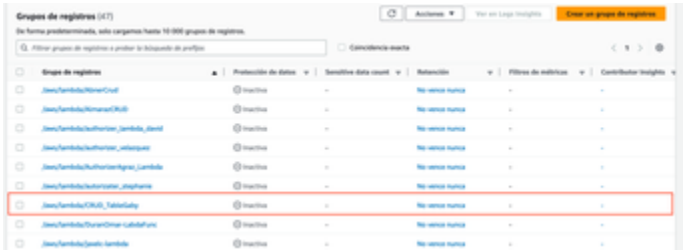
Now we can add the new GET method:

```
aws apigateway put-method --rest-api-id cltoha32o6 --resource-id sl2by4
--http-method GET --authorization-type CUSTOM --authorizer-id xy8e1w
```



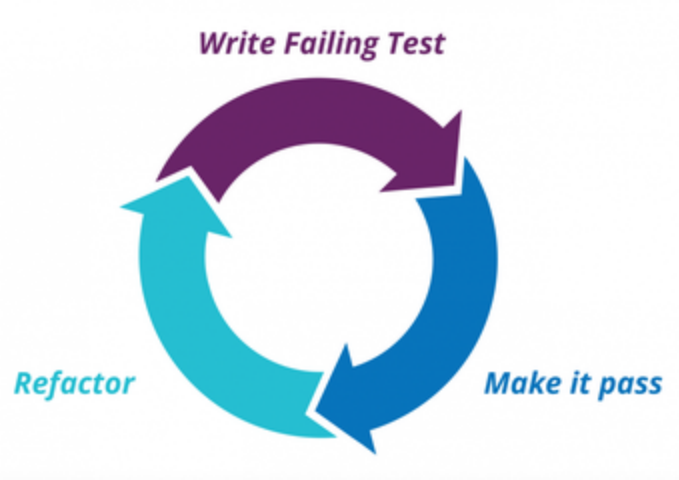
Now we can do a deployment of all the API modifications:

```
aws apigateway create-deployment \
--rest-api-id cltoha32o6 \
--stage-name dev \
--description 'Deploy all the things'
```



Read the Test Driven Development is the best thing that has happened to software design article and write a summary and opinions about it. 10 points.

Test Driven Development (TDD) is an iterative approach to software design, when we write software using this approach the final product is unknown; but it provides the ability to have fast feedback about software design



There are usually two ways a project is made first we write the code and then the tests or vice versa:

- **When tests are driven by code:**

Here we first write the code and then we test it, the “hard time” of this method is deciding when to stop writing tests or deciding how much we want it covered.

- **When code is driven by tests:**

Here we define “what we expect from the code”, this makes tests verify the behavior and not the implementation details, also you will always have up-to-date documentation on how it should behave.

When we have tests that use external libraries, variable test data (e.g. LocalDateTime.now() changing every day), a Bloaded setup (too many dependencies), or Mocking Hell (a small change causes the butterfly effect for both the production code and the corresponding tests), we use TDD to help requirements meet implementation.

I personally have never thought of first thinking and writing the tests and then the coding part, I think it does have some benefits such as having the code make specifically what it needs from the requirements and you do not have to “wonder” how many tests are too much.

I think this is something someone has to get used to, otherwise, it may cause some conflict/confusion at first.

This is great since you first write the test that you know the code MUST fulfill, that way you don't overdo it; in other words, you build your code to pass the tests, with all the requirements already specified 🙂