# Jenkins:

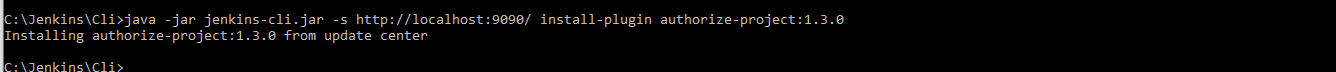
To start Jenkins: .$ java -jar jenkins.war

If default port (8080) is used by another process, mentions the port as well

To start Jenkins: .$ java -jar jenkins.war –httpPort=9090

**To Install Plugins via CLI ()**

java -jar jenkins-cli.jar -s http://localhost:9090/ install-plugin authorize-project:1.3.0



**Clarifications** :

What is SCM in Jenkins?

# GIT:

Difference between HTTP & SSH

## Install and getting started:

Today we will learn

1. How to install Git on windows

2. Adding project/files to git for tracking

3. Git commands

4. Pushing project to remote repository(github)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Step 1 : check if git is already installed

git --version Step

2 : download and install git

Step 3 : add your project to git

Step 4 : commands

- git config --global user.email "yourGitHub@email.com"

- git config --global user.name "yourGitHubusername"

- git init

- git status

- git add

- git commit -m “…..”

- git remote add origin [https://github.com/RaghavAutomation/R...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbVRCMmpTM0lJMmVnTTJUaE16cG1SNWUtRkhRZ3xBQ3Jtc0ttekV1TDFma2V4YzltcnFNaGJrLWdCd3Z4VGc0OURQVldpVXNxWDFZQzFnOG9oVmVVWDRiejl1S1RSeE5BeFNHcjJaamVkaFFzTGJNdlN5Y2dpdzlUWEVOYXZTdlY2N040bnBUV3I0c19UbC1GVzFOVQ&q=https%3A%2F%2Fgithub.com%2FRaghavAutomation%2FRepo2.git)​

- git push -u origin master

- git log

- git —help

## Branching and Merging:

## Add a remote repository:

# Adding a remote repository in Git?

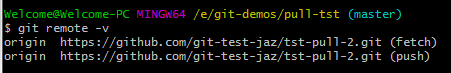
In this tutorial, I will explain how to add a new remote repository, a branch, and file in the remote repository from the terminal (Git Bash).

**Adding the remote repo**

If you are working on Git terminal, then you may check the current repo (if any is added/set) by using this command:

*$ git remote –v*

This should display the URL of the current remote repository as shown below for our test purpose:



If you require adding a new repository then use the **git remote add** command as follows:

*$ git remote add origin https://github.com/user\_name/remote\_repo.git*

For example, this command adds our test remote repository (bootstrap.git):

*$ git remote add origin https://github.com/git-test-jaz/bootstrap.git*

If Git terminal returns a fatal error message like this:

fatal: remote origin already exists.

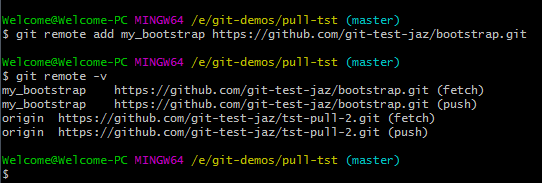
You may use some other name than the origin. For example:

*$ git remote add my\_bootstrap https://github.com/git-test-jaz/bootstrap.git*

For testing, if the remote repo is added or not, run the remote –v command again i.e.

*$ git remote –v*

The result with our test:



You can see the remote repo with the name of my\_bootstrap is added.



**Did you Know?** The origin is just an alias to the remote repository. You may use any other name as an alias as in above example for the remote repositories. Similarly, you may use other commands with that name other than origin .e.g git push origin/other\_name etc.

### **How to add a branch while having multiple repositories?**

In this section, I will show you how to add a branch in a remote repository while having multiple branches.

For that, first I added another repository by using this command:

*$ git remote add my\_test https://github.com/git-test-jaz/hello-git.git*

So, we have now three repositories as shown below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | my\_bootstrap    https://github.com/git-test-jaz/bootstrap.git (fetch)    my\_bootstrap    https://github.com/git-test-jaz/bootstrap.git (push)    my\_test https://github.com/git-test-jaz/hello-git.git (fetch)    my\_test https://github.com/git-test-jaz/hello-git.git (push)    origin  https://github.com/git-test-jaz/tst-pull-2.git (fetch)    origin  https://github.com/git-test-jaz/tst-pull-2.git (push) |

Our target is to create a new branch, adding a file in the my\_test repository. The branch name is “tst\_multiple\_br” and I will add a text file (tst1.txt).

First, creating the branch locally:

*$ git branch tst\_multiple\_br*

This is followed by checking out this branch:

*$ git checkout tst\_multiple\_br*

Now add the text file in that branch locally:

*$ git add tst1.txt*

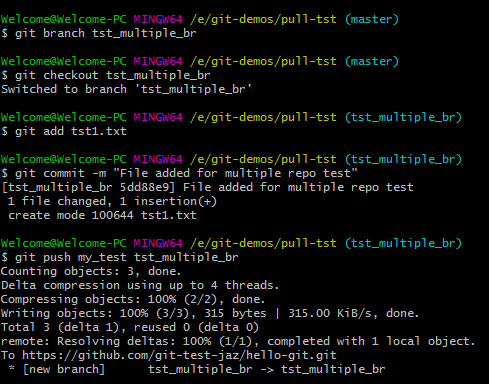
Committing the file added:

*$ git commit -m “File added for multiple repo test”*

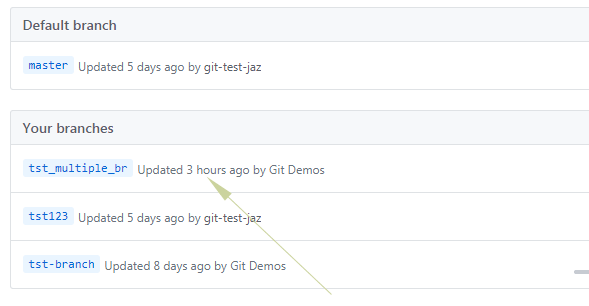
And finally pushing this branch and file in our target remote repository i.e. my\_test

*$ git push my\_test tst\_multiple\_br*

All these commands in the terminal with output:



This is the online view of the remote repository after adding the branch/file:



# Interview:

## Rest vs Soap:

Choosing between these two lies on the hands of the programming language we use, the environment in which we are going to use and the requirements of the application.

As one REST API tutorial put it: SOAP is like an envelope while REST is just a postcard.

Certainly, a postcard is faster and cheaper to send than an envelope, but it could still be wrapped within something else, even an envelope.

You can just read a postcard too, while an envelope takes a few extra steps, like opening or unwrapping to access what is inside.

***SOAP***:

It is the heavy weight choice for the web services access and provides the following advantage over REST:

1. Language, platform & transport independent (REST requires use of HTTP)
2. Build-in error handling
3. Standardized
4. Works well in distributed enterprise environments (REST assumes direct point-to-point communication)
5. Provides significant pre-build extensibility in the form of the WS\* standards
6. Automation when used with certain language products

***REST***:

It is easier to use for most part and is highly flexible. It has the following advantages when compared to SOAP:

1. Easier to use, no expensive tools to interact with rest services
2. Smaller learning curve
3. Efficient (SOAP uses structured XML messages for all but rest can use smaller message formats)
4. Closer to other web technologies in design philosophy
5. Fast (no extensive process required)

### Links:

<https://www.guru99.com/rest-api-interview-question-answers.html>  
https://www.soapui.org/learn/api/soap-vs-rest-api/

## JMS

## WSDL

### WSDL is a contract between you (the provider of the service) and every single one of your customers (consumers of the service).

[**The WSDL**](https://www.soapui.org/soap-and-wsdl/getting-started.html) defines every aspect of the SOAP message. It is even able to define whether any element or attribute can appear multiple times, if it is required or optional, and can even dictate a specific order the elements must appear in.

It is a common misconception that the WSDL is a requirement for a SOAP service.

SOAP was designed before the WSDL, and therefore the WSDL is optional. Although, it is significantly harder to interface with a web service that does not have a WSDL.

On the other hand, if a developer is asked to interface with an existing SOAP web service, he only needs to be given the WSDL, and there are tools that do ***service discovery*** - generate method stubs with appropriate parameters in almost any language from that WSDL.

Many test tools on the market work in the same way - a tester provides a URL to a WSDL, and the tools generate all the calls with sample parameters for all the available methods.

## SWAGGER

Swagger is an Interface Description Language for describing RESTful APIs expressed using JSON.

Swagger (okay, now the "Open API Initiative"... more on that later!) is a framework for *describing your API using a common language* that everyone can understand. Think of it as a ***blueprint for a house***.

### The Rise of Design-First APIs

With this blueprint in mind, there are two main ways to take advantage of Swagger:

* *Top-down approach, or design-first*. This means you are using Swagger to design your API before you have written any actual code.
* *Bottom-up approach, or code-first*. This means you have already written the code for your API, and you will be using Swagger to document your API.

### Link:

https://blog.readme.com/what-is-swagger-and-why-it-matters/  
<https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/>

## IIB

### Load Balancing IIB:

### Cluster IIB:

### Security in IIB:

### Authentication & Authorization:

Authentication is for Identity management, tells us it really someone who says it is them.

Authorization is for access management, defines who has access to what?

Regression Testing & Black Box Testing

### What is a web service?

Simply put, a web service is a resource that’s made available over the internet. Therefore, web services, by definition, require a **network**.

### What is an API?

An API is an interface that can be used to program software that interacts with an existing application. In practice, an API is “a set of functions and procedures” that allow you to access and build upon the data and functionality of an existing application.

What is a logical tree?

How do you connect with database from Compute node?

Difference between Compute Node and Mapping Node?

### Types of queues available in MQ

### Type of variables in IIB

### Timer Nodes

### File Handling in IIB

### What is the branching strategy in any version control?

A branching strategy is a convention, or a set of rules, that describes when branches are created, naming guidelines for branches, what use branches should have, and so on.

#### Git:

There are various Git branching strategies (workflows) such as

**Git** Flow – (When multiple versions required in Production)

**GitHub** Flow – (When single version required in production)

**GitLab** Flow – (environment branches)

##### Git Flow:

it is based in two main branches with infinite lifetime:

* master — this branch contains production code. All development code is merged into master in sometime.
* develop — this branch contains pre-production code. When the features are finished then they are merged into develop.

During the development cycle, a variety of supporting branches are used:

* feature-\* — feature branches are used to develop new features for the upcoming releases. May branch off from develop and must merge into develop.
* hotfix-\* — hotfix branches are necessary to act immediately upon an undesired status of master. May branch off from master and must merge into master and develop.
* release-\* — release branches support preparation of a new production release. They allow many minor bugs to be fixed and preparation of meta-data for a release. May branch off from develop and must merge into master and develop.

###### Advantages

* Ensures a clean state of branches at any given moment in the life cycle of project
* The branches naming follows a systematic pattern making it easier to comprehend
* [It has extensions](https://github.com/nvie/gitflow) and support on most used git tools
* It is ideal when there it needs to be multiple versions in production

###### Disadvantages

* The Git history becomes unreadable
* The master/develop split is considered redundant and makes the Continuous Delivery and the Continuous Integration harder
* It isn’t recommended when it need to maintain single version in production

##### GitHub Flow

The GitHub Flow is a lightweight workflow. It was created by [GitHub in 2011](http://scottchacon.com/2011/08/31/github-flow.html) and respects the following 6 principles:

1. Anything in the master branch is deployable
2. To work on something new, create a branch off from master and given a descriptively name(ie: new-oauth2-scopes)
3. Commit to that branch locally and regularly push your work to the same named branch on the server
4. When you need feedback or help, or you think the branch is ready for merging, open a [pull request](http://help.github.com/send-pull-requests/)
5. After someone else has reviewed and signed off on the feature, you can merge it into master
6. Once it is merged and pushed to master, you can and should deploy immediately

###### Advantages

* it is friendly for the Continuous Delivery and Continuous Integration
* A simpler alternative to Git Flow
* It is ideal when it needs to maintain single version in production

###### Disadvantages

* The production code can become unstable most easily
* Are not adequate when it needs the release plans
* It doesn’t resolve anything about deploy, environments, releases, and issues
* It isn’t recommended when multiple versions in production are needed

##### GitLab Flow

The GitLab Flow is a workflow created by [GitLab in 2014](https://about.gitlab.com/2014/09/29/gitlab-flow/). It combine [feature-driven development](https://en.wikipedia.org/wiki/Feature-driven_development) and feature branches with issue tracking. The most difference between GitLab Flow and GitHub Flow are the environment branches having in GitLab Flow (e.g. staging and production) because there will be a project that isn’t able to deploy to production every time you merge a feature branch (e.g. SaaS applications and Mobile Apps)

The GitLab Flow is based on 11 rules:

1. Use feature branches, no direct commits on master
2. Test all commits, not only ones on master
3. Run all the tests on all commits (if your tests run longer than 5 minutes have them run in parallel).
4. Perform code reviews before merges into master, not afterwards.
5. Deployments are automatic, based on branches or tags.
6. Tags are set by the user, not by CI.
7. Releases are based on tags.
8. Pushed commits are never rebased.
9. Everyone starts from master, and targets master.
10. Fix bugs in master first and release branches second.
11. Commit messages reflect intent.

###### Advantages

* It defines how to make the Continuous Integration and Continuous Delivery
* The git history will be cleaner, less messy and more readable (see [why devs prefers squash and merge, instead of only merging, on this article](https://softwareengineering.stackexchange.com/questions/263164/why-squash-git-commits-for-pull-requests))
* It is ideal when it needs to single version in production

###### Disadvantages

* It is more complex that the GitHub Flow
* It can become complex as Git Flow when it needs to maintain multiple versions in production

#### Reference Links:

https://pradeeploganathan.com/git/git-branching-strategies/

<https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf>

### How do you handle errors in IIB?

### What is scalability and why we need that?

**Cloud scalability in cloud computing** refers to the ability to increase or decrease IT resources as needed to meet changing demand.

### What is caching?

### Pub/Sub:

#### Difference Between Messages and Topics

A queue means a message goes to one and only one possible subscriber.

A topic goes to each subscriber.

**Topics (**1-to-many**)**

In JMS, a Topic implements *publish and subscribe* semantics. When you publish a message, it goes to all the subscribers who are interested - so zero to many subscribers will receive a copy of the message. Only subscribers who had an active subscription at the time the broker receives the message will get a copy of the message.

You can also call this the 'broadcast' model.

**Queues (**1-to-1**)**

A JMS Queue implements *load balancer* semantics. A single message will be received by exactly one consumer. If there are no consumers available at the time the message is sent it will be kept until a consumer is available that can process the message. If a consumer receives a message and does not acknowledge it before closing, then the message will be redelivered to another consumer. A queue can have many consumers with messages *load balanced* across the available consumers.

##### Reference Link:

https://tomd.xyz/jms/

#### Reference Link:

https://www.mqtechconference.com/sessions\_v2013/MQTC-PubSub\_Part\_I.pdf

### Transactions in IIB

When you design a message flow for transaction, you need to consider the transaction property at two places – first the transaction property at message flow level and secondly the transaction mode property at individual node level.

Let’s look into each of these property in detail:

**Message flow transaction at flow level**

A message flow transaction starts at the input node of the flow and ends at the last downstream node in the flow – when it finishes the processing of the message. The message flow may be accessing zero or more external resources such as DB, MQ etc when involving in a transaction.

The message flow transaction setting is configured by the input node’s transaction property. If the transaction mode of the input node is set to ‘Yes’ or ‘Automatic’, then the flow level transaction mode is set to ‘YES’ – Means the flow can participate in broker coordinated transaction. The setting done at input node will determine the rest of the message flow’s transactional behavior.

**Message flow transaction setup at node level**

You can configure each individual nodes to be configured to participate in a broker coordinated transaction or not. Not all nodes support transactions. But those nodes which does not support transactions (File nodes), still contains a transaction property to configure. If it is an input node which does not support transaction then the transaction property is to configure the remaining nodes behavior in the flow.

If the input node is one among which supports transactions, then the default property set by the node for the Transaction mode property is ‘Yes’. If the node is not an input node and supports transactions, then the Transaction mode property set by default is ‘Automatic’ – Means this node will follow the message flow transaction property which again determined by the input nodes transaction mode.

If the node is not an input node and does not support transactions, then the transaction mode property will not be available to configure.

**Note**: HTTP by default will not support transaction. Hence SOAP over HTTP also wont support for coordinated transactions. But SOAP over JMS will do support coordinated transaction in broker.

Why doesn’t file node support transactions?

Operation Mode in SOAP Request Node?

ESQL COMMIT inside the compute node?

**Database transactions**

Database transactions by default support broker coordinated transactions. For Globally coordinated transactions, only DB2, Oracle and Sybase database will support this function.

By default, when accessing the database via compute node, all the database operations performing will be under sync point and will commit when the flow thread finishes the successfully. If any exceptions happened after the compute node and are not handled in downstream nodes and is handled only in Input node or not handled at all, the DB operations performed in the compute node will be rolled back. There are certain points to be noted when need to commit the DB operations individually:

Using the transaction mode of compute node as ‘Commit’ in one node and ‘Automatic in previous or other nodes will make the message flow transaction in an inconsistent state and will cause exception if the DSN used in both compute nodes are same.

If need to commit or handle database operations in one of the compute nodes out of broker transactions, recommended to use a different DSN name for those compute nodes even though all nodes may be referring to the same database.

Else, use ESQL COMMIT inside the compute node to commit the Database transactions of a particular node separately/ immediately.

The problem with setting transaction property as ‘Commit’ will commit all transactions which are performed till that compute node (MQ PUT, Previous DB Operations with transaction as ‘Automatic’) resulting the message flow transaction in an inconstant state. Design the flow and nodes properly when dealing with transaction to  
make sure your message processed as expected and avoid unexpected behaviors.

#### globally coordinated transactions

Whether to perform a globally coordinated transaction is a business decision and should be configured separately. Additional configuration at BAR file and transaction manager (ex: MQ) should be done to enable this. Should consider the performance trade off when setting up the globally coordinated transactions.

When need to setup nodes to perform out of the broker coordinated transactions (commit/rollback independently), special care needs to take, else the message flow transaction become inconsistent state.

#### Reference Link:

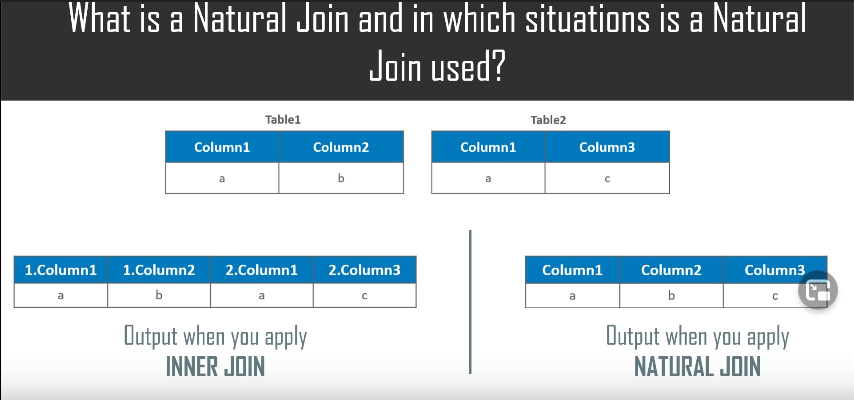
https://eaideveloper.wordpress.com/2015/06/03/message-flow-transactions-in-iib/

## DDL & DML difference:

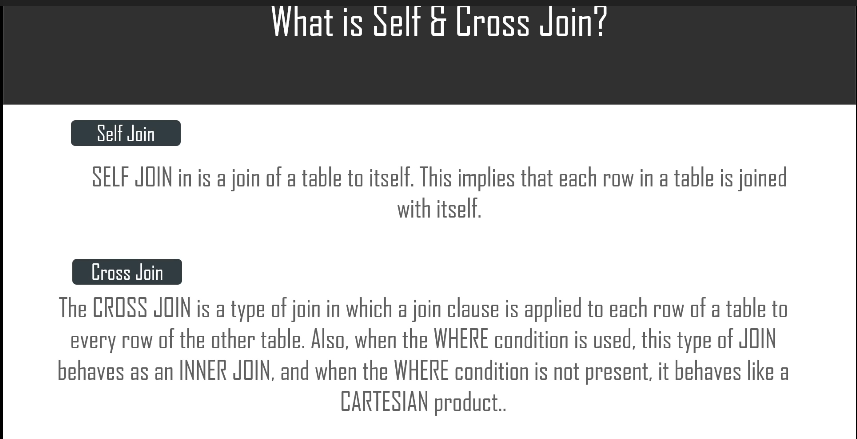
## DB Joins:

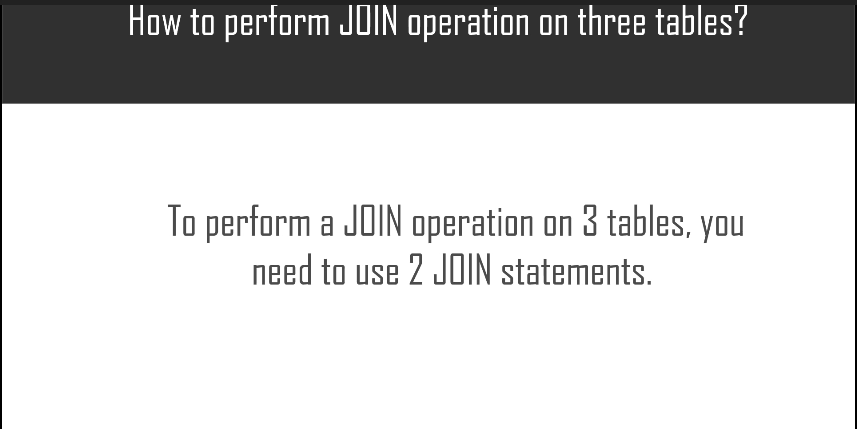
**Natural Joins:**

To avoid the redundancy of columns that will be produced by Inner Joins.



Cross Join & Self Join:





## Micro Services:

* Technology diversity, e., Microservices can mix easily with other frameworks, libraries,  and databases
* Fault isolation, e., a process failure should not bring the whole system down.
* Greater support for smaller and parallel team
* Independent deployment
* Deployment time reduce

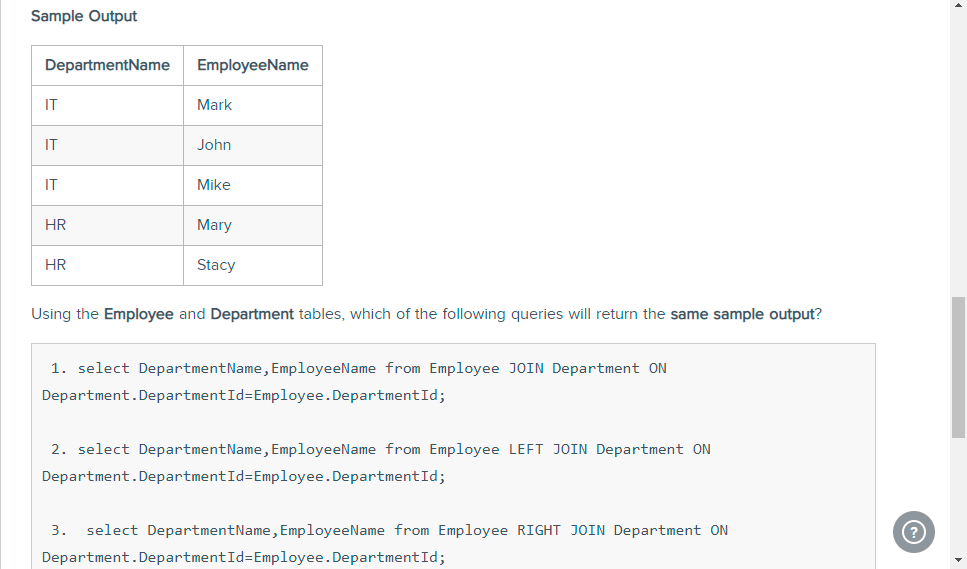
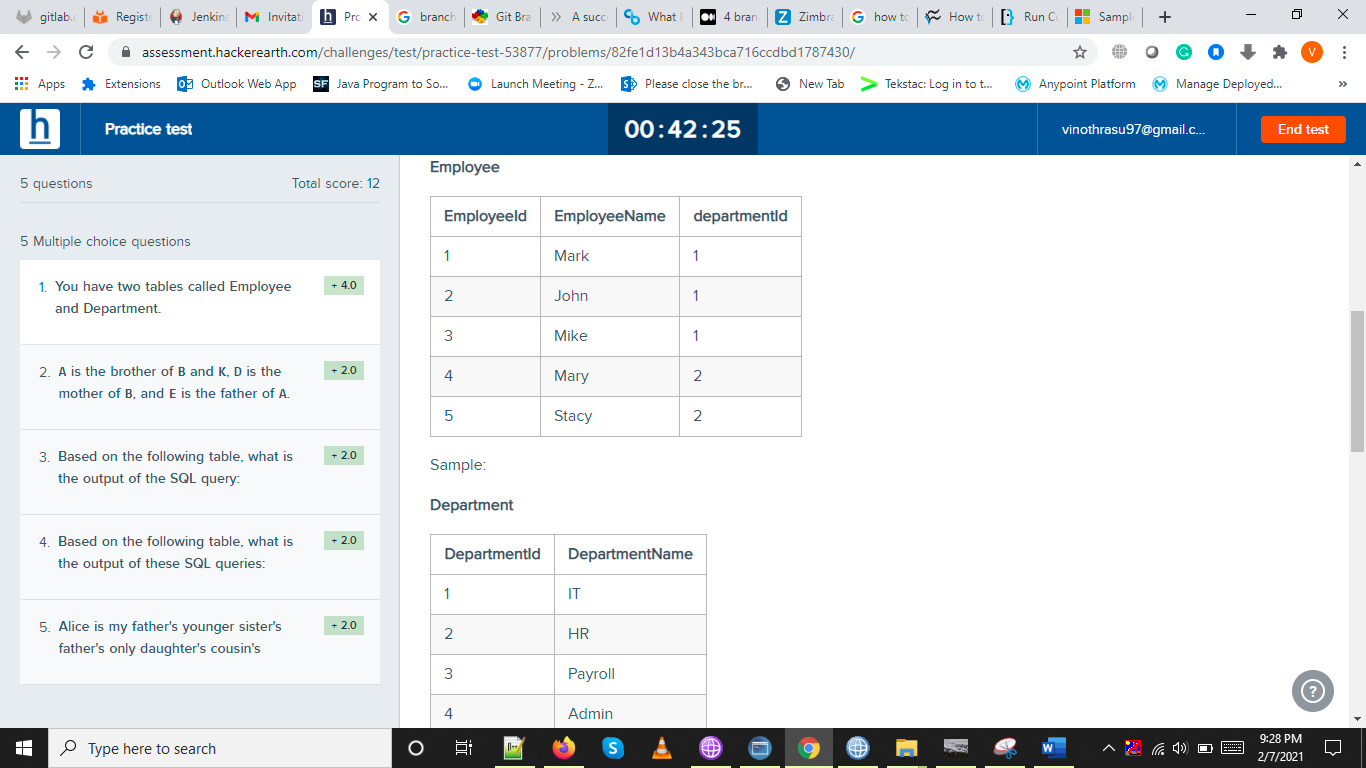
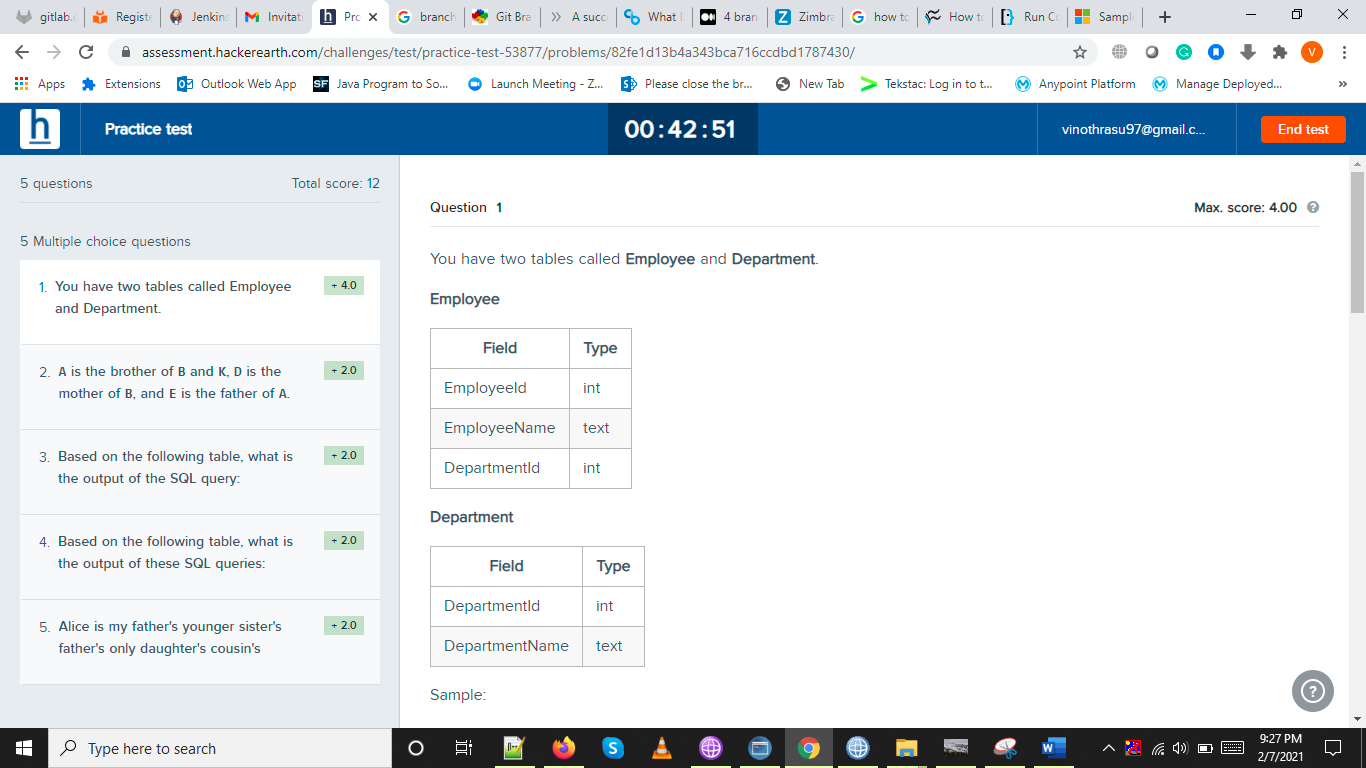
**What are main differences between Microservices and Monolithic Architecture?**

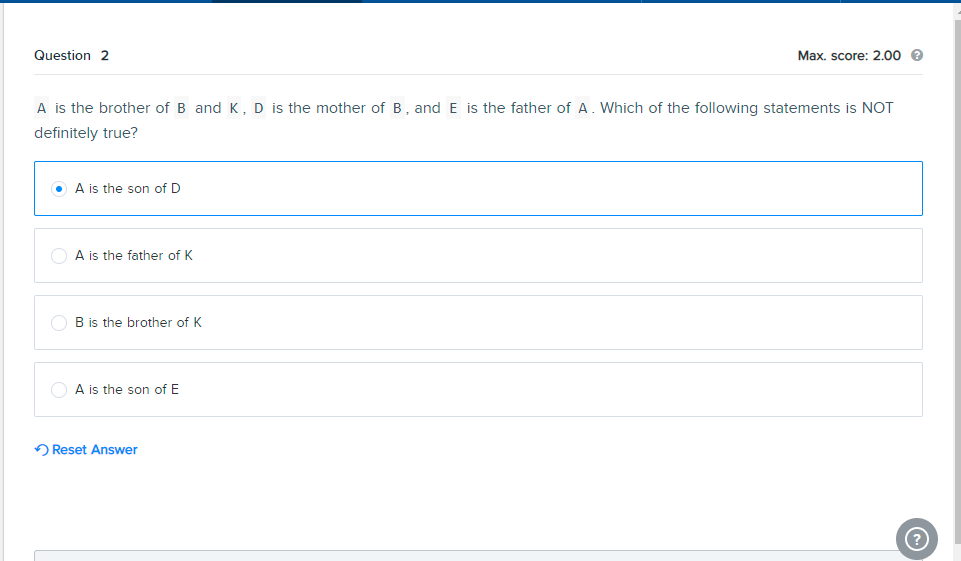
|  |  |
| --- | --- |
| **Microservices** | **Monolithic Architecture** |
| Service Startup is fast | Service startup takes time |
| Microservices are loosely coupled architecture. | Monolithic architecture is mostly tightly coupled. |
| Changes done in a single data model does not affect other Microservices. | Any changes in the data model affect the entire database |
| Microservices  focuses  on products, not projects | Monolithic put emphasize over the whole project |

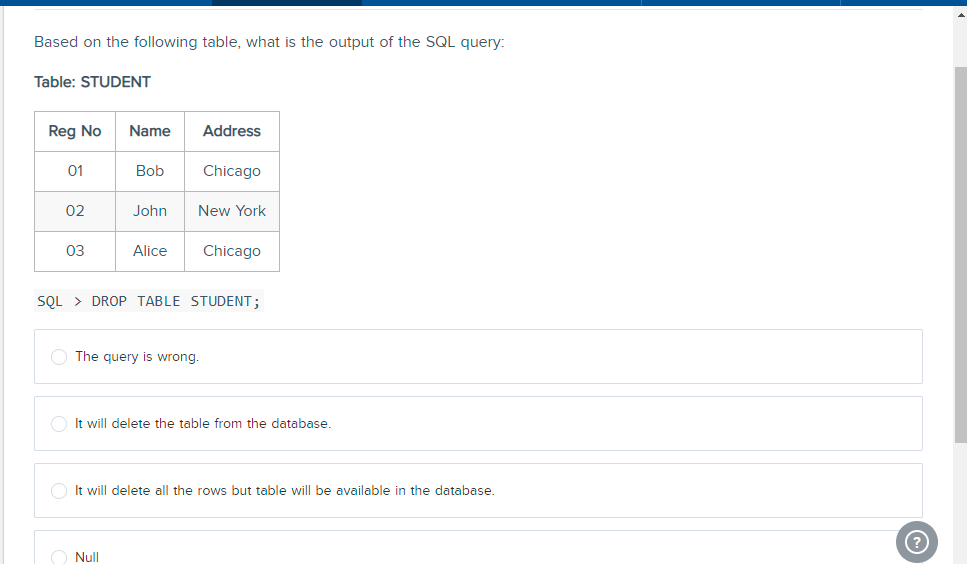
### Links:

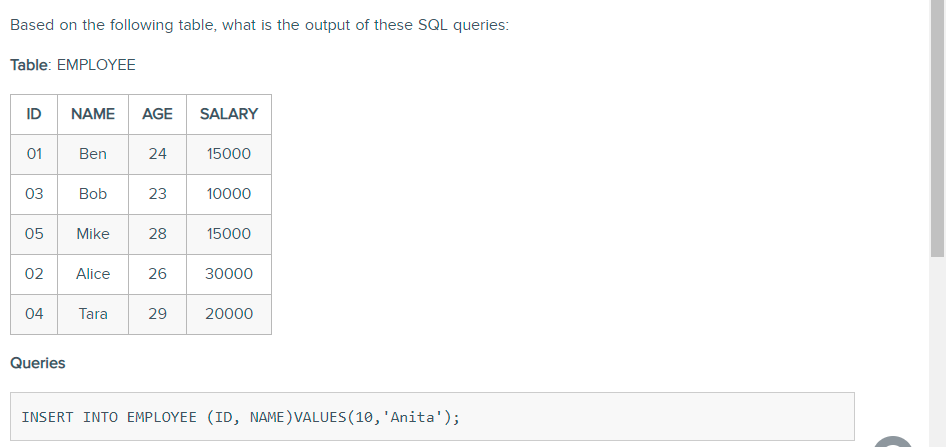
https://www.guru99.com/microservices-interview-questions.html

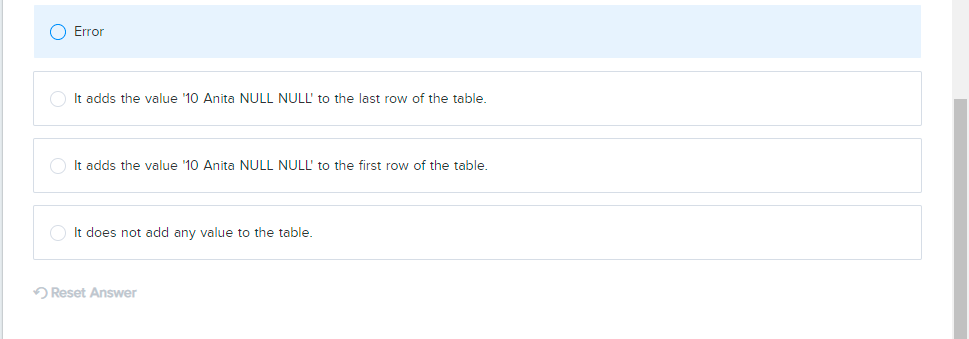
# Apisero Test Practice:

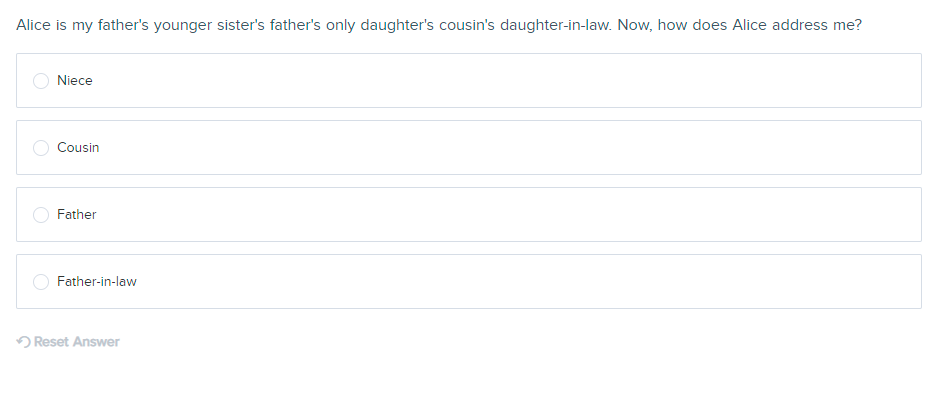












## Main Test Questions:

Saas, PaaS, IaaS

Cloud