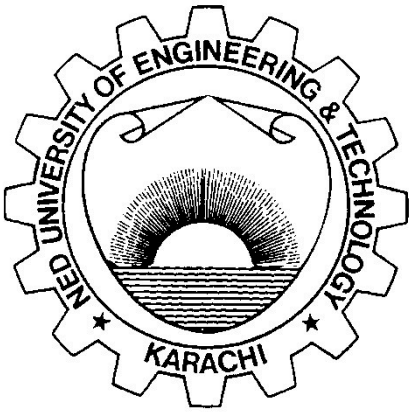
Practical Workbook

SE-312

SOFTWARE CONSTRUCTION & DEVELOPMENT



Name __SHEHNILA NAREJO_____

Year _____2022_____

Batch _____2022_____

Roll No _____

Department: SOFTWARE_____

Department of Software Engineering NED University of Engineering & Technology

CONTENTS

Lab Session No.	Title	Teacher's Signature	Date
1	Explore the usage of any documentation tool in Software Development Life Cycle (SDLC)		
2	Practice any project management tool to prepare a project plan		
3	Introduction and working of SOLID Principles in Software Construction		
4	Introduction and working of Design Patterns in Software Construction		
5	Practice function oriented UML diagram for the suggested system: Data Flow Diagrams		
6	Practice behavioral view UML diagrams for the suggested system: State Chart, Sequence and Collaboration Diagrams		
7	Code Quality Analysis		
8	Software Testing		
9	Virtual Machine Lab Environment		
10	Building a CI/CD Pipeline Using Jenkins		
11	Software Version Control with Git		
12	Open Ended Laboratory		

Lab Session 01

Explore the usage of any documentation tool in Software Development Life Cycle (SDLC)

Software documentation

All large software development projects, irrespective of application, generate a large amount of associated documentation. A high proportion of software process costs is incurred in producing this documentation. Furthermore, documentation errors and omissions can lead to errors by endusers and consequent system failures with their associated costs and disruption. Therefore, managers and software engineers should pay as much attention to documentation and its associated costs as to the development of the software itself. The documents associated with a software project and the system being developed, have a number of associated requirements:

1. They should act as a communication medium between members of the development team.
2. They should be a system information repository to be used by maintenance engineers.
3. They should provide information for management to help them plan, budget and schedule the software development process.
4. Some of the documents should tell users how to use and administer the system.

Types of Software Documentation

Generally, the software project documentation produced falls into two classes:

1. **Process documentation:** These documents record the process of development and maintenance. Plans, schedules, process quality documents and organizational and project standards are process documentation. The major characteristic of process documentation is that most of it becomes outdated. Plans may be drawn up on a weekly, fortnightly or monthly basis. Progress will normally be reported weekly. Although of interest to software historians, much of this process information is of little real use after it has gone out of date and there is not normally a need to preserve it after the system has been delivered.

2. **Product documentation:** This documentation describes the product that is being developed. System documentation describes the product from the point of view of the engineers developing and maintaining the system; user documentation provides a product description that is oriented towards system users. Unlike most process documentation, it has a relatively long life. It must evolve in step with the product, which it describes. Product documentation includes user documentation, which tells users how to use the software product and system documentation, which is principally intended for maintenance engineers. See Figure 2.1 for different types of User Documents.

System documentation consists of the following:

- Software Requirement Specification (SRS)
- Design and Architecture
- Source Code Documents
- Testing Documents
- Formal Technical Reviews

User documentation includes the following:

- End-users Documentation
- System-admin Documentation

Document structure

The document structure is the way in which the material in the document is organized into chapters and, within these chapters, into sections and subsections. Document structure has a major impact on readability and usability. As with software systems, you should design document structures so that the different parts are as independent as possible. The IEEE standard for user documentation proposes that the structure of a document should include the components shown in Figure 2.2.

Component	**Description**
Identification data	Data such as a title and identifier that uniquely identifies the document.
Table of contents	Chapter/section names and page numbers.
List of illustrations	Figure numbers and titles
Introduction	Defines the purpose of the document and a brief summary of the contents
Information for use of the documentation	Suggestions for different readers on how to use the documentation effectively.
Concept of operations	An explanation of the conceptual background to the use of the software.
Procedures	Directions on how to use the software to complete the tasks that it is designed to support.
Information on software commands	A description of each of the commands supported by the software.
Error messages and problem resolution	A description of the errors that can be reported and how to recover from these errors.
Glossary	Definitions of specialized terms used.
Related information sources	References or links to other documents that provide additional information
Navigational features	Features that allow readers to find their current location and move around the document.
Index	A list of key terms and the pages where these terms are referenced.
Search capability	In electronic documentation, a way of finding specific terms in the document.

Fig 1.1: Suggested components in a software user document

Document Preparation

Document preparation is the process of creating a document and formatting it for publication. Figure 2.3 shows the document preparation process as being split into 3 stages namely document creation, polishing and production. The three phases of preparation and associated support facilities are explained in the figure below:

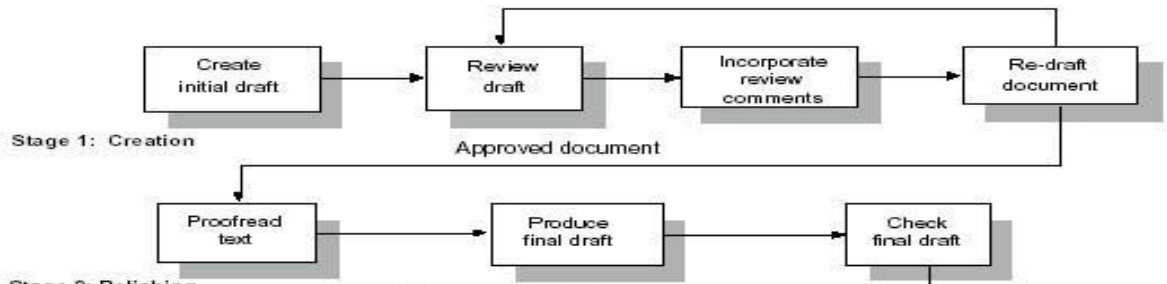


FIG: 1.2 Stagees of document preparation

LaTeX

LaTeX is a high-quality document preparation system; it includes features designed to produce technical and scientific documentation. It allows users to very quickly tackle the more complicated parts of typesetting, such as inputting mathematics, creating tables of contents, referencing and creating bibliographies, and having a consistent layout across all sections. It is based on the WYSIWYM (what you see is what you mean) idea, meaning you only have focus on the contents of your document and the computer will take care of the formatting.

LaTeX is available as free software. To set up a basic TeX/LaTeX system, download and run the Basic MiKTeX Installer. TeXworks is used as editor for writing commands. We start off with a basic .tex file which contains LaTeX code. LaTeX uses control statements, which define how your content should be formatted. LaTeX compiler takes .tex file as input and convert it into a .pdf file.

First Piece of LaTeX

```
\documentclass{article}

\begin{document}
First document. This is a simple example, with no  extra parameters or packages included.
\end{document}
```

The first line of code declares the type of document, known as the class. The class controls the overall appearance of the document. In this case, the class is article, the simplest and most common LATEX class. Other types of documents you may be working on may require different classes such as **book** or **report**. Then the contents of our document are written, enclosed inside the **\begin{document}** and **\end{document}** tags. This is known as the body of the document. You can start writing here and make changes to the text if you wish. To see the result of these changes in the PDF you have to compile the document.

The Preamble of a document

In the previous example the text was entered after the **\begin{document}** command. Everything in your **.tex** file before this point is called the **preamble**. In the preamble you define the type of document you are writing, the language you are writing in, the packages you would like to use (more on this later) and several other elements. For instance, a normal document preamble would look like this:

```
\documentclass[12pt, letterpaper]{article}
```

Below a detailed description of \documentclass:

\documentclass[12pt, letterpaper]{article}

As said before, this defines the type of document. Some additional parameters included in the square brackets brackets can be passed to the command. These parameters must be comma-separated. In the example, the extra parameters set the font size (**12pt**) and the paper size (**letterpaper**). Of course other font sizes (**9pt**, **11pt**, **12pt**) can be used, but if none is specified, the default size is **10pt**. As for the paper size other possible values are **a4paper** and **legalpaper.**

Adding a title, author and date

To add a title, author and date to our document, you must add three lines to the **preamble** (NOT the main body of the document). These lines are:

\title{First document}
\author{Hubert Farnsworth}
\thanks{funded by the Overleaf team}

This can be added after the name of the author, inside the braces of the **title** command. It will add a superscript and a footnote with the text inside the braces.

\date{September 2019}

Enter the date manually or use the command **\today** so the date will be updated automatically.

```
\documentclass[12pt, letterpaper, twoside] {article}

\title{First document}
\author{Hubert Farnsworth \thanks{funded by the Overleaf team}} \date{September 2019}
```

Now a title, author and date are provided to the document, we can print this information on the document with the **\maketitle** command. This should be included in the **body** of the document at the place you want the title to be printed.

```
\begin{document}

\maketitle
We have now added a title, author and date to our first \LaTeX{} document! \end{document}
```

Adding Comments

To make a comment in LATEX, simply write a **%** symbol at the beginning of the line as shown below:

```
\begin{document}

\maketitle
\pagenumbering {gobble}
We have now added a title, author and date to our first  \LaTeX{}
document!
 % This line here is a comment. It will not be printed in the
document.
 \end{document}
```

Basic Formatting

. Abstracts

In scientific documents it's a common practice to include a brief overview of the main subject of the paper. In LATEX there's the **abstract** environment for this. The **abstract** environment will put the text in a special format at the top of your document.

```
\begin{document} \newpage
\begin{abstract}
This is a simple paragraph at the beginning of the  document. A brief introduction about the main subject.
\end{abstract}
\end{document}
```

- Paragraphs and Newlines

When writing the contents of your document, if you need to start a new paragraph you must hit the "Enter" key twice (to insert a double blank line). Notice that LATEX automatically indents paragraphs. To start a new line without actually starting a new paragraph insert a break line point, this can be done by \\ (a double backslash as in the example) or the \newline command.

```
\begin{document}
\begin{abstract}
This is a simple paragraph at the beginning of the document. A brief introduction about the main subject.
\end{abstract}
Now that we have written our abstract, we can begin writing our first paragraph.

This line will start a second Paragraph.
\end{document}
```

- Chapters and Sections

Commands to organize a document vary depending on the document type, the simplest form of organization is the sectioning, available in all formats.

```
\chapter{First Chapter}

\section{Introduction}
```

```
This is the first section.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortisfacilisis sem. Nullam
nec mi et neque pharetra sollicitudin. Praesent imperdietmi nec ante. Donec ullamcorper, felis non
sodales...

\section{Second Section}

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisissem. Nullam nec mi et
neque pharetra sollicitudin. Praesent imperdiet mi necante...

\subsection{First Subsection}
Praesent imperdietmi nec ante. Donec ullamcorper, felis non sodales...

\section*{Unnumbered Section}
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Etiam lobortis facilisissem

The command \section{} marks the beginning of a new section, inside the braces is set the title. Section
numbering is automatic and can be disabled by including a * in the section command as \section*{}. We can
also have \subsection{}s, and indeed \subsubsection{}s. The basic levels of depth are listed below:
```

-1	**\part{part}**
0	**\chapter{chapter}**
1	**\section{section}**
2	**\subsection{subsection}**
3	**\subsubsection{subsubsection}**
4	**\paragraph{paragraph}**
5	**\subparagraph{subparagraph}**

Note that \part and \chapter are only available in report and book document classes.

• Bold, Italics and Underlining

Following are some simple text formatting commands.

- **Bold**: Bold text in LaTeX is written with the **\textbf{...}** command.
- Italics: Italicised text in LaTeX is written with the **\textit{...}** command.
- Underline: Underlined text in LaTeX is written with the **\underline{...}** command.

```
Some of the \textbf{greatest} discoveries in \underline{science}
were made by \textbf{\textit{accident}}.
```

• Unordered lists

Unordered lists are produced by the **itemize** environment. Each entry must be preceded by the control

sequence **\item** as shown below. By default the individual entries are indicated with a black dot, so-called bullet. The text in the entries may be of any length.

```
\begin{itemize}
  \item The individual entries are indicated with a black dot, a so-called bullet.
  \item The text in the entries may be of any length.
\end{itemize}
```

- Ordered lists

Ordered list have the same syntax inside a different environment. We make ordered lists using the enumerate environment:

```
\begin{enumerate}
  \item This is the first entry in our list
  \item The list numbers increase with each entry we add \end{enumerate}
```

EXERCISES

1. Construct a Software Requirement Specifications (SRS) document for a Software System of your choice using LaTeX. Attach the printout of the .tex file and .pdf file.

2. Develop a brief document describing the basic functionality of an IoT based Health Monitoring System. Also include any block diagram/ image for explaining the overall functionality of the system. Use LaTeX. Attach the printout of the .tex and .pdf files. (Feel free to you use the other available formatting options in LaTeX).

3. Suppose that you are developing a calculator application and gathering requirements for it. Your calculator is able to solve some basic linear equations as well. Prepare an initial draft of requirement specification document, highlighting various examples of equations your system is able to solve. Use LaTeX. Attach the printout of the .tex and .pdf files.

Lab Session 02

Practice any project management tool to prepare a project plan

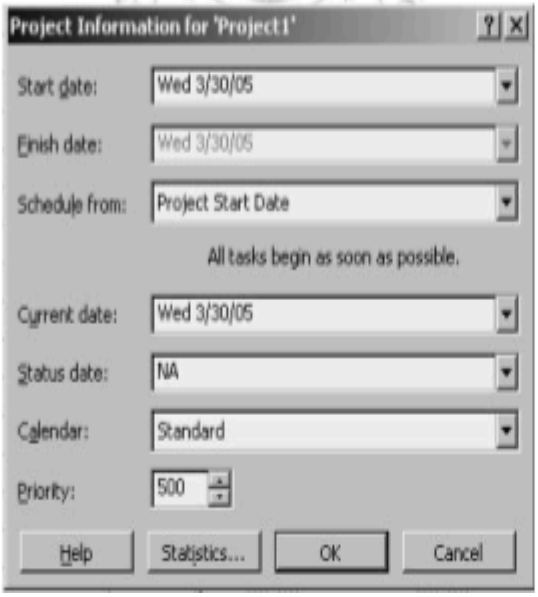
Project management

Project management is the process of planning, organizing, and managing tasks and resources to accomplish a defined objective, usually within constraints on time, resources, or cost. Sound planning is one key to project success, but sticking to the plan and keeping the project on track is no less critical. Microsoft Project features can help in monitoring progress and keep small slips from turning into landslides.

First let us understand some basic terms and definitions that will be used quite often:

Scope: of any project is a combination of all individual tasks and their goals.
Resources: can be people, equipment, materials or services that are needed to complete various tasks. The amount of resources affects the scope and time of any project.

You can create a Project from a Template File by choosing File > New from the menu. In the New File dialog box that opens, select the Project Templates tab and select the template that suits your project best and click OK. (You may choose a Blank Project Template and customize it)
Once a new Project page is opened, the Project Information dialog box opens.



Tasks: They are a division of all the work that needs to be completed in order to accomplish the project goals.

STARTING A NEW PROJECT

Fig 2.1: Project Information dialog box

Enter the start date or select an appropriate date by scrolling down the list. Click OK. Project automatically enters a default start time and stores it as part of the dates entered and the application window is displayed.

Fig 2.2: Main Window

Views allow you to examine y ıt information you want displayed at any given time. You can use a combination of views in the same window at the same time.

Project Views are categorized into two types:

- Task Views (5 types)
- Resource Views (3 types)

The Project worksheet is located in the main part of the application window and displays different information depending on the view you choose. The default view is the Gantt Chart View.

Tasks

Goals of any project need to be defined in terms of tasks. There are four major types of tasks:

1. Summary tasks - contain subtasks and their related properties
2. Subtasks - are smaller tasks that are a part of a summary task
3. Recurring tasks - are tasks that occur at regular intervals
4. Milestones - are tasks that are set to zero duration and are like interim goals in the project

• Entering Tasks and assigning task duration

Click in the first cell and type the task name. Press enter to move to the next row. By default, estimated duration of each task is a day. But, there are very few tasks that can be completed in a day's time. You can enter the task duration as and when you decide upon a suitable estimate.

Double-clicking a task or clicking on the Task Information button on the standard toolbar opens the Task Information box. You can fill in more detailed descriptions of tasks and attach documents related to the task in the options available in this box.

To enter a milestone, enter the name and set its duration to zero. Project represents it as a diamond shape instead of a bar in the Gantt chart.

To copy tasks and their contents, click on the task ID number at the left of the task and copy and paste as usual.

You can enter Recurring tasks by clicking on Insert > Recurring task and filling in the duration and recurrence pattern for the task.

Any action you perform on a summary task - deleting it, moving or copying it apply to its subtasks too.

• Outlining tasks

Once the summary tasks have been entered in a task table, you will need to insert subtasks in the blank rows and indent them under the summary task. This is accomplished with the help of the outlining tool.

Outlining is already active when you launch a project and its tools are found at the left end of the Formatting bar. To enter a subtask, enter the task in a blank cell in the Task Name column and click the Indent button on the Outlining tool bar. The Show feature in this toolbar is drop down tool that gives you an option of different Outline levels.

A summary task is outdented to the left cell border, is bold and has a Collapse (-) (Hide subtasks) button in front of it and its respective subtasks are indented with respect to it.

Advantages of Outlining:

- It creates multiple levels of subtasks that roll up into a summary task
- Collapse and expand summary tasks when necessary
- Apply a Work Breakdown structure
- Move, copy or delete entire groups of tasks

LINKS

Tasks are usually scheduled to start as soon as possible i.e. the first working day after the project start date.

Dependencies

Dependencies specify the manner in which two tasks are linked. Because Microsoft Project must maintain the manner in which tasks are linked, dependencies can affect the way a task is scheduled. In a scenario where there are two tasks, the following dependencies exist:

Finish to Start – Task 2 cannot start until task 1 finishes.
Start to Finish – Task 2 cannot finish until task 1 starts.
Start to Start – Task 2 cannot start until task 1 starts.
Finish to Finish – Task 2 cannot finish until task 1 finishes.

Tasks can be linked by following these steps:

1. Double-click a task and open the Task Information dialog box.
2. Click the predecessor tab.
3. Select the required predecessor from the drop down menu.
4. Select the type of dependency from drop down menu in the Type column.
5. Click OK.

The Split task button splits tasks that may be completed in parts at different times with breaks in their duration times.

CONSTRAINTS

Certain tasks need to be completed within a certain date. Intermediate deadlines may need to be specified. By assigning constraints to a task you can account for scheduling problems. There are about 8 types of constraints and they come under the flexible or inflexible category. They are:

- **As Late As Possible** – Sets the start date of your task as late in the Project as possible, without pushing out the Project finish date.

- **As Soon As Possible** – Sets the start date of your task as soon as possible without preceding the project start date.
- **Must Finish On** – Sets the finish date of your task to the specified date.
- **Must Start On** – Sets the start date of your task to the specified date.
- **Start No Earlier Than** – Sets the start date of your task to the specified date or later.
- **Start No Later Than** – Sets the start date of your task to the specified date or earlier.
- **Finish No Earlier Than** – Sets the finish date of your task to the specified date or later.
- **Finish No Later Than** – Sets the finish date of your task to the specified date or earlier.

Flexible constraints (demarcated by a red dot in Microsoft Project 2000) restrict scheduling to a great extent whereas flexible constraints (blue dot) allow Project to calculate the schedule and make appropriate adjustments based on the constraint applied.

Inflexible constraints can cause conflicts between successive and preceding tasks at times and you may need to remove such a constraint.

To apply a constraint:

1. Open the Task Information dialog box.
2. Click the Advanced tab and open the Constraint type list by clicking on the drop-down arrow and select it.
3. Select a date for the Constraint and click OK.

UPDATE COMPLETED TASKS

If you have tasks in your project that have been completed as they were scheduled, you can quickly update them to 100% complete all at once, up to a date you specify.

1. On the View menu, click Gantt Chart.
2. On the Tools menu, point to Tracking, and then click Update Project
3. Click Update work as complete through and then type or select the date through which you want progress updated.
4. If you don't specify a date, Microsoft Project uses the current or status date.
5. Click Set 0% or 100% complete only.
6. Click Entire Project

RESOURCES

Once you determine that you need to include resources into your project you will need to answer the following questions:

- What kind of resources do you need?

- How many of each resource do you need?
- Where will you get these resources?
- How do you determine what your project is going to cost?

Resources are of two types - work resources and material resources.

Work resources complete tasks by expending time on them. They are usually people and equipment that have been assigned to work on the project.

Material resources are supplies and stocks that are needed to complete a project.
A new feature in Microsoft Project 2000 is that it allows you to track material resources and assign them to tasks.

Entering Resource Information in Project

The Assign Resources dialog box is used to create list of names in the resource pool.

To enter resource lists:

1. Click the Assign Resources button on the Standard Tool bar or Tools > Resources > Assign resources.
2. Select a cell in the name column and type a response name. Press Enter
3. Repeat step 2 until you enter all the resource names.
4. Click OK.

Resource names cannot contain slash (/), brackets [ ] and commas (,). Another way of defining your resource list is through the Resource Sheet View.

1. Display Resource Sheet View by choosing View > Resource Sheet or click the Resource Sheet icon on the View bar.
2. Enter your information. (To enter material resource use the Material Label field)

To assign a resource to a task:

1. Open the Gantt Chart View and the Assign Resources Dialog box.
2. In the Entry Table select the tasks for which you want to assign resources.
3. In the Assign Resources Dialog box, select the resource (resources) you want to assign.
4. Click the Assign button.

EXERCISE

1. Construct a project plan for the project who's SRS is documented in Lab Session 01 (Exercise#1). Also attach its printout.

Lab Session 03
Introduction and working of SOLID Principles in Software Construction

Objectives:

Understand and implement the SOLID principles in software design.
Write cleaner, more maintainable, and flexible code using SOLID principles.
Refactor non-SOLID code to comply with the principles

Single Responsibility Principle (SRP)

A class should have only one reason to change, meaning that a class should only have one job or responsibility.

Real-Time Example of Single Responsibility Principle in C#: Banking System

Let's understand the real-time example of a Banking System using the Single Responsibility Principle. A banking system allows customers to maintain a bank account where they can deposit and withdraw money. Additionally, the bank wants to provide a facility to print a statement of the account's transaction history**.**

Violation of SRP:

The BankAccount class might combine transaction handling and statement printing in a simple design. Let us see how we can implement the above example without following the Single Responsibility Principle in C#:

```
using System;
using System.Collections.Generic;
namespace SRPDemo
{
public class BankAccount
{
public int AccountNumber { get; private set;
} public double Balance { get; private set; }
private List<string> Transactions = new List<string>();
public BankAccount(int accountNumber)
{
```

```
AccountNumber = accountNumber;
}
public void Deposit(double amount)
{
Balance += amount;
Transactions.Add($"Deposited ${amount}. New Balance: ${Balance}");
}
public void Withdraw(double amount)
{
Balance -= amount;
Transactions.Add($"Withdrew ${amount}. New Balance: ${Balance}");
}
public void PrintStatement()
{
Console.WriteLine($"Statement for Account: {AccountNumber}");
foreach (var transaction in Transactions)
{
Console.WriteLine(transaction);
}
}
}
}
```

Here, the BankAccount class handles:

Transaction operations.

Printing the transaction statement.

Following SRP:

A cleaner approach would separate transaction management from statement printing:

BankAccount manages the transactions of the account.

StatementPrinter handles printing the transaction statement.

Let us see how we can implement the above example following the Single Responsibility Principle in C#:

```
using System;
using System.Collections.Generic;
namespace SRPDemo
{
public class BankAccount
```

```
{
public int AccountNumber { get; private set; }
public double Balance { get; private set; }
public List<string> Transactions = new List<string>();
public BankAccount(int accountNumber)
{
AccountNumber = accountNumber;
}
public void Deposit(double amount)
{
Balance += amount;
Transactions.Add($"Deposited ${amount}. New Balance: ${Balance}");
}
public void Withdraw(double amount)
{
Balance -= amount;
Transactions.Add($"Withdrew ${amount}. New Balance: ${Balance}");
}
}
public class StatementPrinter
{
public void Print(BankAccount account)
{
Console.WriteLine($"Statement for Account: {account.AccountNumber}");
foreach (var transaction in account.Transactions)
{
Console.WriteLine(transaction);
}
}
}
//Testing the Single Responsibility Principle
public class Program
{
```

```
public static void Main()
{
BankAccount johnsAccount = new
BankAccount(123456); johnsAccount.Deposit(500);
johnsAccount.Withdraw(100);
StatementPrinter printer = new
StatementPrinter();
printer.Print(johnsAccount);
Console.ReadKey();
}
}
```

In this design, BankAccount focuses solely on transaction-related operations. Only this class is affected if there's a change in how transactions are managed. Conversely, StatementPrinter focuses on printing the statement. If the format or medium of the statement needs a change, only the StatementPrinter class requires modifications. This adherence to SRP makes the system modular and maintainable. When you run the above code, you will get the following output.

```
Statement for Account: 123456
Deposited $500. New Balance: $500
Withdrew $100. New Balance: $400
```

Real-Time Example of Single Responsibility Principle in C#: Education System

Let's see another real-time example Education System using the Single Responsibility Principle. In a university's digital system, students are enrolled in courses, and at the end of a semester, they receive grades for each course. The university wants to calculate each student's GPA (Grade Point Average) based on their grades. Additionally, the system should generate a transcript showing all courses, grades, and the calculated GPA.

Violation of SRP:

In an elementary design, the Student class might handle course enrollment, grade assignment, GPA calculation, and transcript generation. Let us see how we can implement the above example without following the Single Responsibility Principle in C#:

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace SRPDemo
{
public class Student
```

```
{
public string Name { get; set; }
private Dictionary<string, double> CoursesAndGrades = new Dictionary<string, double>(); public
void EnrollCourse(string courseName)
{
CoursesAndGrades[courseName] = 0; // default grade
}
public void AssignGrade(string courseName, double grade)
{
if (CoursesAndGrades.ContainsKey(courseName))
{
CoursesAndGrades[courseName] = grade;
}
}
public double CalculateGPA()
{
// Basic GPA calculation logic
return CoursesAndGrades.Values.Average();
}
public void PrintTranscript()
{
Console.WriteLine($"Transcript for {Name}");
foreach (var course in CoursesAndGrades)
{
Console.WriteLine($"{course.Key}: {course.Value}");
}
Console.WriteLine($"GPA: {CalculateGPA()}");
}
}
```

Here, the Student class has multiple responsibilities:

- Managing course enrollments.
- Handling grade assignments.
- Calculating GPA.
- Printing the transcript.

Following SRP:

A clearer approach would segregate these functionalities:

The student manages course enrollments and grade assignments.

GPACalculator calculates the GPA for a student.

TranscriptGenerator creates and prints the student's transcript.

Let us see how we can implement the above example following the Single Responsibility Principle in C#:

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace SRPDemo
{
public class Student
{
public string Name { get; set; }
public Dictionary<string, double> CoursesAndGrades = new Dictionary<string, double>();
public void EnrollCourse(string courseName)
{
CoursesAndGrades[courseName] = 0; // default grade
}
public void AssignGrade(string courseName, double grade)
{
if (CoursesAndGrades.ContainsKey(courseName))
{
CoursesAndGrades[courseName] = grade;
}
}
}
public class GPACalculator
{
public double CalculateGPA(Student student)
{
// Basic GPA calculation logic
return student.CoursesAndGrades.Values.Average();
}
```

```
}
public class TranscriptGenerator
{
private GPACalculator _gpaCalculator;
public TranscriptGenerator(GPACalculator gpaCalculator)
{
_gpaCalculator = gpaCalculator;
}
public void PrintTranscript(Student student)
{
Console.WriteLine($"Transcript for {student.Name}");
foreach (var course in student.CoursesAndGrades)
{
Console.WriteLine($"{course.Key}: {course.Value}");
}
Console.WriteLine($"GPA: {_gpaCalculator.CalculateGPA(student)}");
}
}
//Testing the Single Responsibility Principle
public class Program
{
public static void Main()
{
Student alice = new Student { Name = "Alice" };
alice.EnrollCourse("Mathematics");
alice.AssignGrade("Mathematics", 90);
GPACalculator gpaCalc = new GPACalculator();
TranscriptGenerator transcriptGen = new TranscriptGenerator(gpaCalc);
transcriptGen.PrintTranscript(alice);
Console.ReadKey();
}
}
```

By adhering to SRP, each class has a clear, singular duty. If the university decides to change the GPA calculation method, only the GPACalculator requires modification. Similarly, if the transcript format needs adjustments, only the TranscriptGenerator is affected. This modular design simplifies future changes and maintenance.

OPEN CLOSED PRINCIPLE:

The Open-Closed Principle (OCP) is one of the SOLID principles of object-oriented design. It states that software entities (like classes, modules, and functions) should be open for extension but closed for modification.

Real-Time Example of Open-Closed Principle in C#: E-commerce System that Provides Discounts
Let's understand the Open-Closed Principle (OCP) with a real-world example. Imagine you're building an e-commerce system that provides discounts to different types of customers.
Without OCP:
If you hard-code each discount type, you would end up with a switch or if-else chain, and each time you wanted to add a new customer type or discount rule, you'd modify the class. Let us see how we can implement the above example without following the Open-Closed Principle in C#: u'd modify the class.
Let us see how we can implement the above example without following the Open-Closed Principle in C#:

```
using System;
namespace OCPDemo
{
public enum CustomerType
{
Regular,
Premiu
m,
Newbie
}
public class DiscountCalculator
{
public double CalculateDiscount(double price, CustomerType customerType)
{
switch (customerType)
{
```

```
case CustomerType.Regular:
return price * 0.1; // 10% discount for regular customers
case CustomerType.Premium:
return price * 0.3; // 30% discount for premium customers
case CustomerType.Newbie:
return price * 0.05; // 5% discount for new customers
default:
throw new ArgumentOutOfRangeException();
}
}
}
}
```

With OCP:

We can define a strategy pattern to adhere to OCP. Each discount type will have its own class, and adding a new discount would mean adding a new class without modifying the existing ones. Let us see how we can implement the above example following the Open-Closed Principle in C#:

```
using System;
namespace OCPDemo
{
//Create an interface for the discount strategy
public interface IDiscountStrategy
{
double CalculateDiscount(double price);
}
//Implement this interface for each discount type
public class RegularDiscount : IDiscountStrategy
{
public double CalculateDiscount(double price)
{
return price * 0.1;
}
}
public class PremiumDiscount : IDiscountStrategy
```

```
{
public double CalculateDiscount(double price)
{
return price * 0.3;
}
}
public class NewbieDiscount : IDiscountStrategy
{
public double CalculateDiscount(double price)
{
return price * 0.05;
}
}
//Modify the DiscountCalculator class to accept an IDiscountStrategy
public class DiscountCalculator
{
private readonly IDiscountStrategy _discountStrategy;
public DiscountCalculator(IDiscountStrategy discountStrategy)
{
_discountStrategy = discountStrategy;
}
public double CalculateDiscount(double price)
{
return _discountStrategy.CalculateDiscount(price);
}
}
//Testing the Open-Closed Principle
public class Program
{
public static void Main()
{
var regularDiscount = new RegularDiscount();
var calculator = new DiscountCalculator(regularDiscount);
```

```
double discountedPrice = calculator.CalculateDiscount(100); // 10% discount applied
var premiumDiscount = new PremiumDiscount();
calculator = new DiscountCalculator(premiumDiscount);
discountedPrice = calculator.CalculateDiscount(100); // 30% discount applied
Console.ReadKey();
}
}
}
```

In this approach, if a new discount strategy needs to be introduced, you create a new class implementing the IDiscountStrategy without modifying the existing system. This aligns with OCP.

Real-Time Example of Open-Closed Principle in C#: Notification System

Let's take an example of a notification system where a user can be notified through various channels.

Without OCP:

Imagine if you hard-code each notification channel, you'd likely use a switch or if-else statement, and every time a new channel needs to be added, the NotificationSender class would be modified. Let us see how we can implement the above example without following the Open- Closed Principle in C#:

```
using System;
namespace OCPDemo
{
public enum NotificationChannel
{
Email
,
SMS
}
public class NotificationSender
{
public void SendNotification(NotificationChannel channel, string message)
{
switch (channel)
{
case NotificationChannel.Email:
Console.WriteLine($"Sending Email:
{message}"); break;
```

```
case NotificationChannel.SMS:
Console.WriteLine($"Sending SMS: {message}");
break;
default:
throw new ArgumentOutOfRangeException();
}
}
}
```

With OCP:
We'll use a strategy pattern for each notification channel to adhere to the Open-Closed Principle.

```
using System;
namespace OCPDemo
{
//Start with an interface
public interface INotificationChannel
{
void Send(string message);
}
//Implement this interface for each channel
public class EmailNotification : INotificationChannel
{
public void Send(string message)
{
Console.WriteLine($"Sending Email: {message}");
}
}
public class SMSNotification : INotificationChannel
{
public void Send(string message)
{
Console.WriteLine($"Sending SMS: {message}");
```

```
}
}
//Modify the NotificationSender class to accept any INotificationChannel
public class NotificationSender
{
private readonly INotificationChannel _channel;
public NotificationSender(INotificationChannel channel)
{
_channel = channel;
}
public void SendNotification(string message)
{
_channel.Send(message);
}
}
//Testing the Open-Closed Principle
public class Program
{
public static void Main()
{
var emailChannel = new EmailNotification();
var sender = new NotificationSender(emailChannel);
sender.SendNotification("Hello via Email!");
var smsChannel = new SMSNotification(); sender =
new NotificationSender(smsChannel);
sender.SendNotification("Hello via SMS!");
Console.ReadKey();
}
}
}
```

In the future, if you'd like to introduce a Slack channel, you'd create a SlackNotification class that implements INotificationChannel without modifying the NotificationSender class. This demonstrates how a system can be open for extensions (new channels) but closed for modifications. When you run the above code, you will get the following output.

```
Sending Email: Hello via Email!
Sending SMS: Hello via SMS!
```

Liskov Substitution Principle (LSP)

Theory:

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other words, subclasses should behave as expected when used in place of their parent class.

Real-Time Example of Liskov Substitution Principle in C#: Creating Different Shapes

Let's see one real-time example involving Rectangle and Square classes, one of the best examples to understand the Liskov Substitution Principle (LSP).

Violating LSP

Let us first see how we can implement the above example without following the Liskov Substitution Principle (LSP) in C#:

```
namespace LSPDemo
{
public class Rectangle
{
public virtual double Width { get; set; }
public virtual double Height { get; set; }
public double GetArea()
{
return Width * Height;
}
}
public class Square : Rectangle
{
public override double Width
{
get { return base.Width; }
set { base.Width = base.Height = value; }
}
public override double Height
```

```
{
get { return base.Height; }
set { base.Width = base.Height = value; }
}
}
}
```

As you can see in the above code, there is a class named Square that is a subclass of the Rectangle class and tries to enforce the rule to ensure that its width and height are equal. However, this approach violates the Liskov Substitution Principle (LSP) because if a method is intended to work with a Rectangle object, it may not function correctly when a Square instance is used as input. For a better understanding, please have a look at the following example.

```
using System;
namespace LSPDemo
{
public class Rectangle
{
public virtual double Width { get; set; }
public virtual double Height { get; set; }
public double GetArea()
{
return Width * Height;
}
public void ChangeDimensions(Rectangle rect, double width, double height)
{
rect.Width = width;
rect.Height =
height;
Console.WriteLine($"Area: {rect.GetArea()}");
}
}
public class Square : Rectangle
{
public override double Width
{
```

```
get { return base.Width; }
set { base.Width = base.Height = value; }
}
public override double Height
{
get { return base.Height; }
set { base.Width = base.Height = value; }
}
}
public class Program
{
public static void Main()
{
var rect = new Rectangle { Width = 2, Height =
3 }; rect.ChangeDimensions(rect, 4, 5); // This
works fine var square = new Square { Width = 2
};
rect.ChangeDimensions(square, 4, 5); // This behaves
unexpectedly! Console.ReadKey();
}
}
}
```

In the above example, when we try to set different values for the width and height of a square, the behavior is not what we expect because the area will not be 4 * 5. With LSP

Instead of trying to make Square a subclass of Rectangle, the classes can be refactored to follow the Liskov Substitution Principle in C#. Let us see how we can implement the above example following the Liskov Substitution Principle in C#:

```
using System;
namespace LSPDemo
{
public abstract class Shape
{
public abstract double GetArea();
}
public class Rectangle : Shape
```

```
{
public double Width { get; set; }
public double Height { get; set; }
public override double GetArea()
{
return Width * Height;
}
public void ChangeDimensions(Rectangle rect, double width, double height)
{
rect.Width = width;
rect.Height = height;
Console.WriteLine($"Area: {rect.GetArea()}");
}
}
public class Square : Shape
{
public double Side { get; set; }
public override double GetArea()
{
return Side * Side;
}
}
//Testing the Liskov Substitution Principle
public class Program
{
public static void Main()
{
var rect = new Rectangle { Width = 2, Height = 3 };
rect.GetArea(); // This works fine
rect.ChangeDimensions(rect, 4, 5); // This works fine var
square = new Square { Side = 2 };
square.GetArea(); // This also works fine
// This will not work
```

```
//square.ChangeDimensions(square,
4, 5); Console.ReadKey();
}
}
}
```

With this design, both the Rectangle and Square subclasses inherit from the common base Shape class. This guarantees that there are no unexpected behaviors or differences when calculating their respective areas and the Liskov Substitution Principle is followed. As a result, any method created for the Shape class can anticipate consistent behavior from both Rectangle and Square instances with no unexpected side effects.

Real-Time Example of Liskov Substitution Principle in C#: Bank Accounts

Let's see another real-time example of Bank Accounts to understand the Liskov Substitution Principle (LSP). Suppose you have two types of bank accounts: a RegularAccount and a FixedTermDepositAccount. The FixedTermDepositAccount doesn't allow withdrawals until the end of the term.

Violating LSP

Let us first see how we can implement the above example without following the Liskov Substitution Principle (LSP) in C#:

```
using System;
namespace LSPDemo
{
public class BankAccount
{
protected double balance;
public virtual void Deposit(double amount)
{
balance += amount;
}
public virtual void Withdraw(double amount)
{
```

```
if (balance >= amount)
{
balance -= amount;
}
else
{
throw new InvalidOperationException("Insufficient funds");
}
}
public double GetBalance()
{
return balance;
}
}
public class FixedTermDepositAccount : BankAccount
{
public override void Withdraw(double amount)
{
throw new InvalidOperationException("Cannot withdraw from a fixed term deposit
account until term ends");
}
}
}
```

If a FixedTermDepositAccount object is substituted for a BankAccount object in a context that expects withdrawals to be possible, the program will break, violating the Liskov Substitution Principle (LSP).

Following LSP:

Let's refactor the classes to follow the Liskov Substitution Principle (LSP). Let us see how we can implement the above example following the Liskov Substitution Principle in C#:

```
using System;
```

```
namespace LSPDemo
{
public abstract class BankAccount
{
protected double balance;
public virtual void Deposit(double amount)
{
balance += amount;
Console.WriteLine($"Deposit: {amount}, Total Amount: {balance}");
}
public abstract void Withdraw(double amount);
public double GetBalance()
{
return balance;
}
}
public class RegularAccount : BankAccount
{
public override void Withdraw(double amount)
{
if (balance >= amount)
{
balance -= amount;
Console.WriteLine($"Withdraw: {amount}, Balance: {balance}");
}
else
{
Console.WriteLine($"Trying to Withdraw: {amount}, Insufficient Funds, Available Funds:
{balance}");
}
}
}
public class FixedTermDepositAccount : BankAccount
```

```
{
private bool termEnded = false; // simplification for the example
public override void Withdraw(double amount)
{
if (!termEnded)
{
Console.WriteLine("Cannot withdraw from a fixed term deposit account until term ends");
}
else if (balance >= amount)
{
balance -= amount;
Console.WriteLine($"Withdraw: {amount}, Balance: {balance}");
}
else
{
Console.WriteLine($"Trying to Withdraw: {amount}, Insufficient Funds, Available Funds:
{balance}");
}
}
}
//Testing the Liskov Substitution Principle
public class Program
{
public static void Main()
{
Console.WriteLine("RegularAccount:");
var RegularBankAccount = new RegularAccount();
RegularBankAccount.Deposit(1000); RegularBankAccount.Deposit(500);
RegularBankAccount.Withdraw(900); RegularBankAccount.Withdraw(800);
Console.WriteLine("\nFixedTermDepositAccount:");
var FixedTermDepositBankAccount = new FixedTermDepositAccount();
```

```
FixedTermDepositBankAccount.Deposit(1000);
FixedTermDepositBankAccount.Withdraw(500);
Console.ReadKey();
}
}
}
```

When designing the BankAccount class, we make the Withdraw method abstract to enable derived types might have their own rules for withdrawal. This approach helps the client code understand that subclasses may have varying behaviors, and Liskov Substitution Principle (LSP) is not violated. When you run the above code, you will get the following output.

```
RegularAccount:
Deposit: 1000, Total Amount: 1000
Deposit: 500, Total Amount: 1500
Withdraw: 900, Balance: 600
Trying to Withdraw: 800, Insufficient Funds, Available Funds: 600

FixedTermDepositAccount:
Deposit: 1000, Total Amount: 1000
Cannot withdraw from a fixed term deposit account until term ends
```

Interface Segregation Principle

The Interface Segregation Principle (ISP) states that a class should not be forced to implement interfaces it does not use (a class should only implement interfaces it uses). In other words, interfaces should be designed to be specific to the needs of the implementing classes.

Real-Time Example of Interface Segregation Principle in C#: Library Management System

A library management system can perform multiple tasks, like borrowing books, returning books, and searching the catalog. Users, such as Members, Librarians, and Guests, have different permissions.

Members can borrow and return books and search the catalog.

Librarians can manage inventory (add or remove books) and also perform tasks members can.

Guests can only search the catalog.

Violating ISP:

Let us first see how we can implement the above example without following the Interface Segregation Principle (ISP) in C#. Here's an approach where a single interface tries to encompass all functionalities:

```
using System;
namespace ISPDemo
{
public class Book
{
public string Title { get; set; }
public string Author { get; set; }
public string ISBN { get; set; }
}
public interface IUser
{
void BorrowBook(string bookId);
void ReturnBook(string bookId);
void SearchCatalog(string searchTerm);
void AddBook(Book book);
void RemoveBook(string bookId);
}
public class Guest : IUser
{
public void BorrowBook(string bookId)
{
throw new NotImplementedException("Guests cannot borrow books.");
}
public void ReturnBook(string bookId)
{
throw new NotImplementedException("Guests cannot return books.");
}
public void SearchCatalog(string searchTerm)
{
// Implementation to search books.
}
public void AddBook(Book book)
{
```

```
throw new NotImplementedException("Guests cannot add books.");
}
public void RemoveBook(string bookId)
{
throw new NotImplementedException("Guests cannot remove books.");
}
}
}
```

In this approach, the Guest class is forced to implement irrelevant methods. With
to ISP:

Let us see how we can rewrite the above example following the Interface Segregation Principle (ISP) in C#. By segregating the interfaces based on functionality, we can make the system more focused and logical:

```
using System;
namespace ISPDemo
{
public class Book
{
public string BookId { get; set; }
public string Title { get; set; }
public string Author { get; set; }
public string ISBN { get; set; }
public override string
ToString()
{
return $"[BookId: {BookId}, Title: {Title}, Author:{Author}, ISBN:{ISBN}]";
}
}
public interface IBorrowReturn
{
void BorrowBook(string bookId);
void ReturnBook(string bookId);
}
public interface ISearchable
```

```
{
void SearchCatalog(string searchTerm);
}
public interface IManageInventory
{
void AddBook(Book book);
void RemoveBook(string bookId);
}
public class Member : IBorrowReturn, ISearchable
{
public void BorrowBook(string bookId)
{
// Implementation to borrow a book.
Console.WriteLine($"Member Borrow Book, BookId: {bookId}");
}
public void ReturnBook(string bookId)
{
// Implementation to return a book.
Console.WriteLine($"Member Return Book, BookId: {bookId}");
}
public void SearchCatalog(string searchTerm)
{
// Implementation to search books.
Console.WriteLine($"Member Search Book, Search Catalog: {searchTerm}");
}
}
public class Librarian : IBorrowReturn, ISearchable, IManageInventory
{
public void BorrowBook(string bookId)
{
// Implementation to borrow a book.
Console.WriteLine($"Librarian Borrow Book, BookId: {bookId}");
}
```

```
public void ReturnBook(string bookId)
{
// Implementation to return a book.
Console.WriteLine($"Librarian Return Book, BookId: {bookId}");
}
public void SearchCatalog(string searchTerm)
{
// Implementation to search books.
Console.WriteLine($"Librarian Search Book, Search Catalog: {searchTerm}");
}
public void AddBook(Book book)
{
// Implementation to add a book. Console.WriteLine($"Librarian
Add Book, {book}");
}
public void RemoveBook(string bookId)
{
// Implementation to remove a book.
Console.WriteLine($"Librarian Remove Book, BookId: {bookId}");
}
}
public class Guest : ISearchable
{
public void SearchCatalog(string searchTerm)
{
// Implementation to search books.
Console.WriteLine($"Guest Search Book, Search Catalog: {searchTerm}");
}
}
//Testing the Interface Segregation Principle
public class Program
{
public static void Main()
```

```
{
Console.WriteLine("Librarian:");
Librarian librarian = new Librarian();
Book book = new Book()
{
BookId = "BK-10001",
Title = "SOLID Principle using C#",
Author = "Pranaya Rout",
ISBN = "ISBN-Demo"
};
librarian.AddBook(book);
librarian.BorrowBook(book.BookId);
librarian.SearchCatalog("SOLID");
librarian.ReturnBook(book.BookId);
librarian.RemoveBook(book.BookId);
Console.WriteLine("\nMember:");
Member member = new Member();
//member.AddBook(book); //Compile Time Error
member.BorrowBook(book.BookId);
member.SearchCatalog("SOLID");
member.ReturnBook(book.BookId);
//member.RemoveBook(book.BookId); //Compile Time Error
Console.WriteLine("\nMember:");
Guest guest = new Guest();
//guest.AddBook(book); //Compile Time Error
//guest.BorrowBook(book.BookId); //Compile Time Error
guest.SearchCatalog("SOLID");
//guest.ReturnBook(book.BookId); //Compile Time Error
//guest.RemoveBook(book.BookId); //Compile Time Error
Console.ReadKey();
}
}
}
```

By adhering to the Interface Segregation Principle, we now have more fine-grained interfaces that only expose relevant methods to classes. Each user role only needs to implement the tasks that pertain to its responsibilities. When you run the above code, you will get the following output.

```
Librarian:
Librarian Add Book, [BookId: BK-10001, Title: SOLID Principle using C#, Author:Pranaya Rout, ISBN:ISBN-Demo]
Librarian Borrow Book, BookId: BK-10001
Librarian Search Book, Search Catalog: SOLID
Librarian Return Book, BookId: BK-10001
Librarian Remove Book, BookId: BK-10001

Member:
Member Borrow Book, BookId: BK-10001
Member Search Book, Search Catalog: SOLID
Member Return Book, BookId: BK-10001

Member:
Guest Search Book, Search Catalog: SOLID
```

Dependency Inversion Principle (DIP)

The main idea of DIP is to use abstractions instead of concrete implementations. By separating high-level modules with complex features from low-level modules that provide basic operations, DIP promotes the use of an abstraction layer to reduce coupling between the modules.

Real-Time Example of Dependency Inversion Principle in C#: Message Logger System Suppose we are building a system where we want to log messages. Initially, we might have a logging mechanism that writes messages to a console. However, in the future, we may want to switch to logging messages in a file or sending them over the network.

Without DIP:

Let us first see how we can implement the above example without following the Dependency Inversion Principle (DIP):

```
using System;
namespace DIPDemo
{
public class ConsoleLogger
{
public void LogMessage(string message)
{
Console.WriteLine(message);
}
}
public class NotificationService
{
private ConsoleLogger _logger = new ConsoleLogger();
public void Notify(string message)
```

```
{
// ... some notification logic ...
_logger.LogMessage(message);
}
}
}
```

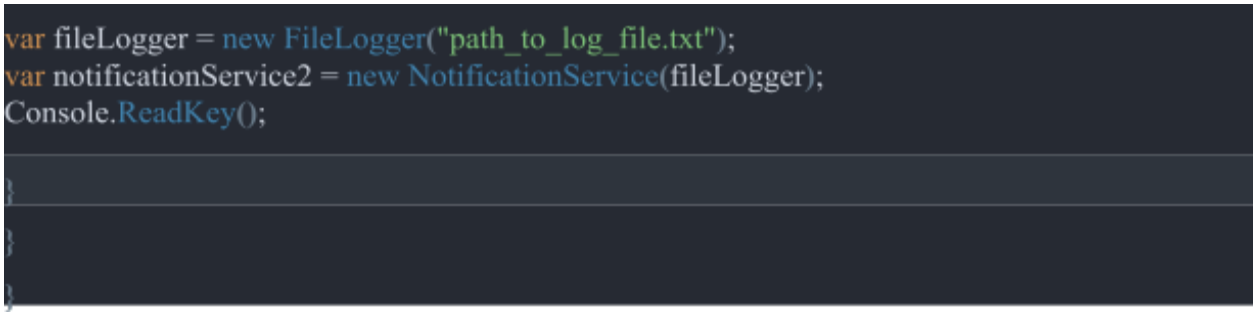
The problem here is that NotificationService is directly dependent on ConsoleLogger. What if we want to change our logging mechanism?

With DIP:

Let us see how we can rewrite the above example following the Dependency Inversion Principle (DIP). The following example code is self-explained, so please go through the comment lines for a better understanding.

```
using System;
using System.IO;
namespace DIPDemo
{
//Interface for logging
public interface ILogger
{
void LogMessage(string message);
}
//Concrete Loggers
public class ConsoleLogger : ILogger
{
public void LogMessage(string message)
{
Console.WriteLine(message);
}
}
public class FileLogger : ILogger
{
private string _filePath;
public FileLogger(string filePath)
```

```
{
_filePath = filePath;
}
public void LogMessage(string message)
{
// Just a simple example. In a real-world scenario, proper exception handling and file IO
management is needed.
File.AppendAllText(_filePath, message);
}
}
//Now, our NotificationService should depend on the abstraction
public class NotificationService
{
private ILogger _logger;
public NotificationService(ILogger logger)
{
_logger = logger;
}
public void Notify(string message)
{
// ... some notification logic ...
_logger.LogMessage(message);
}
}
//Testing the Dependency Inversion Principle
public class Program
{
public static void Main()
{
//Now, when initializing the NotificationService,
//we can decide which logger to use:
var consoleLogger = new ConsoleLogger();
var notificationService1 = new NotificationService(consoleLogger);
```



By applying the Dependency Inversion Principle (DIP), we have decoupled the NotificationService from the concrete implementation of the logger. This makes the system more flexible and open for extension.

Lab Session 04
Introduction and working of Design Patterns in Software Construction

Objective:

The objective of this lab is to understand and implement common design patterns to solve problems in software development. By the end of the lab, students will be able to:

- Recognize where design patterns can be applied.
- Implement design patterns in small programs.

Design Patterns to Implement:

1. **Singleton Pattern**
    - **Task**: Create a class that ensures only one instance can exist throughout the application.
    - **Example Scenario**: A configuration manager class that loads settings from a file and ensures the settings are available globally without creating multiple instances.

      Code:

```
using System;


public class ConfigurationManager
{
    private static ConfigurationManager _instance;


    // Private constructor to prevent instantiation
    private ConfigurationManager()
    {
        Console.WriteLine("Configuration Manager Created.");
    }


    // Public method to get the single instance
```

```
    public static ConfigurationManager GetInstance()
    {
        if (_instance == null)
        {
            _instance = new ConfigurationManager();
        }
        return _instance;
    }

    // Example method
    public string GetSetting(string key)
    {
        return key == "AppName" ? "My App" : "Unknown";
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Access the Singleton instance
        var config = ConfigurationManager.GetInstance();
        Console.WriteLine(config.GetSetting("AppName"));
```

```
        // Confirm that it is the same instance
        var anotherConfig = ConfigurationManager.GetInstance();
        Console.WriteLine(ReferenceEquals(config, anotherConfig)); // Output: True
    }
}
```

2. **Observer Pattern**
   - **Task**: Implement a simple notification system where multiple objects need to be updated when the state of another object changes.
   - **Example Scenario**: A weather monitoring system where multiple displays (e.g., temperature, humidity, pressure) are updated when the weather station data changes.

   Code:

```
using System;
using System.Collections.Generic;

// Subject (Weather Station)
public class WeatherStation
{
    private List<IObserver> observers = new List<IObserver>();
    private float temperature, humidity, pressure;

    // Subscribe an observer
    public void AddObserver(IObserver observer)
    {
```

```
    observers.Add(observer);
}

// Unsubscribe an observer
public void RemoveObserver(IObserver observer)
{
    observers.Remove(observer);
}

// Notify all observers
public void NotifyObservers()
{
    foreach (var observer in observers)
    {
        observer.Update(temperature, humidity, pressure);
    }
}

// Simulate changing weather data
public void SetMeasurements(float temp, float hum, float pres)
{
    temperature = temp;
    humidity = hum;
```

```
        pressure = pres;
        NotifyObservers();
    }
}

// Observer Interface public
interface IObserver
{
    void Update(float temp, float humidity, float pressure);
}

// Concrete Observers (Displays)
public class TemperatureDisplay : IObserver
{
    public void Update(float temp, float humidity, float pressure)
    {
            Console.WriteLine($"Temperature Display: {temp}°C");
    }
}

public class HumidityDisplay : IObserver
{
        public void Update(float temp, float humidity, float pressure)
```

```
    {
        Console.WriteLine($"Humidity Display: {humidity}%");
    }
}

public class PressureDisplay : IObserver
{
    public void Update(float temp, float humidity, float pressure)
    {
        Console.WriteLine($"Pressure Display: {pressure} hPa");
    }
}

// Main Program
class Program
{
    static void Main(string[] args)
    {
        WeatherStation station = new WeatherStation();

        // Create displays
        var tempDisplay = new TemperatureDisplay(); var
        humDisplay = new HumidityDisplay();
```

```
        var presDisplay = new PressureDisplay();

        // Register displays as observers
        station.AddObserver(tempDisplay);
        station.AddObserver(humDisplay);
        station.AddObserver(presDisplay);

        // Simulate new weather data
        station.SetMeasurements(25.3f, 65f, 1013.1f);
    }
}
```

3. **Factory Pattern**
   - **Task**: Develop a factory method that creates objects based on input parameters without exposing the instantiation logic.
   - **Example Scenario**: A shape creation program that can create circles, rectangles, and squares based on user input without exposing object creation details.

Code:

```
using System;

// Shape Interface public
interface IShape
{
    void Draw();
}
```

```
// Concrete Circle Class
public class Circle : IShape
{
    public void Draw() => Console.WriteLine("Drawing a Circle.");
}

// Concrete Rectangle Class
public class Rectangle : IShape
{
    public void Draw() => Console.WriteLine("Drawing a Rectangle.");
}

// Concrete Square Class
public class Square : IShape
{
    public void Draw() => Console.WriteLine("Drawing a Square.");
}

// Shape Factory
public class ShapeFactory
{
    public IShape GetShape(string shapeType)
```

```
    {
        return shapeType.ToLower() switch
        {
            "circle" => new Circle(),
            "rectangle" => new Rectangle(),
            "square" => new Square(),
            _ => null
        };
    }
}

// Main Program
class Program
{
    static void Main(string[] args)
    {
        ShapeFactory shapeFactory = new ShapeFactory();

        // Get user input for shape
        Console.WriteLine("Enter the shape to create (Circle, Rectangle, Square):"); string
        shapeType = Console.ReadLine();

        // Create shape using factory
```

```
            IShape shape = shapeFactory.GetShape(shapeType);


            if (shape != null)
            {
                shape.Draw();
            }
            else
            {
                Console.WriteLine("Invalid shape type.");
            }
        }
    }
```

4. **Strategy Pattern**
    - **Task**: Implement a system where different algorithms or behaviors can be selected at runtime.
    - **Example Scenario**: A payment system where different payment methods (credit card, PayPal, bank transfer) are selected dynamically based on user preference.

Code:

```
using System;


// Payment Strategy Interface public
interface IPaymentStrategy
{
    void Pay(double amount);
}
```

```
// Concrete Strategy for Credit Card Payment public
class CreditCardPayment : IPaymentStrategy
{
    public void Pay(double amount) => Console.WriteLine($"Paid {amount} using Credit Card.");
}

// Concrete Strategy for PayPal Payment
public class PayPalPayment : IPaymentStrategy
{
    public void Pay(double amount) => Console.WriteLine($"Paid {amount} using PayPal.");
}

// Concrete Strategy for Bank Transfer Payment public
class BankTransferPayment : IPaymentStrategy
{
    public void Pay(double amount) => Console.WriteLine($"Paid {amount} using Bank
Transfer.");
}

// Context Class
public class PaymentContext
{
    private IPaymentStrategy _paymentStrategy;

    // Set Payment Strategy at runtime
    public void SetPaymentMethod(IPaymentStrategy paymentStrategy)
```

```
    {
        _paymentStrategy = paymentStrategy;
    }

    // Execute payment
    public void Pay(double amount)
    {
        if (_paymentStrategy != null)
        {
            _paymentStrategy.Pay(amount);
        }
        else
        {
            Console.WriteLine("Payment method not selected.");
        }
    }
}

// Main Program
class Program
{
    static void Main(string[] args)
    {
        PaymentContext paymentContext = new PaymentContext();

        // Simulating user input for selecting payment method
        Console.WriteLine("Select payment method (CreditCard, PayPal, BankTransfer):");
```

```
    string method = Console.ReadLine().ToLower();

    // Dynamically set payment strategy based on user input
    switch (method)
    {
        case "creditcard":
            paymentContext.SetPaymentMethod(new CreditCardPayment());
            break;
        case "paypal":
            paymentContext.SetPaymentMethod(new PayPalPayment());
            break;
        case "banktransfer":
            paymentContext.SetPaymentMethod(new BankTransferPayment());
            break;
        default:
            Console.WriteLine("Invalid payment method.");
            return;
    }

    // Make the payment
    paymentContext.Pay(100.00);
}
```

Lab Session 05

Practice function oriented UML diagram for the suggested system: Data Flow Diagrams

Data flow diagram (DFD)

Data Flow Diagrams (DFD) are used to graphically represent the flow of data in a business information system. DFD describes the processes that are involved in a system to transfer data from the input to the file storage and reports generation.

DFD Symbols There are four basic symbols that are used to represent a data-flow diagram.

1. **Process:**

A process receives input data and produces output with a different content or form. Every process has a name that identifies the function it performs.

A rounded rectangle represents a process
Processes are given IDs for easy referencing

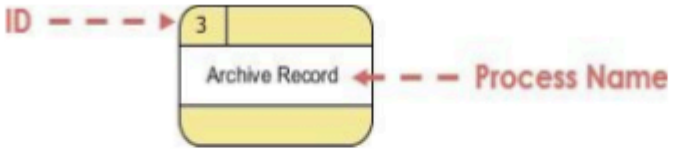
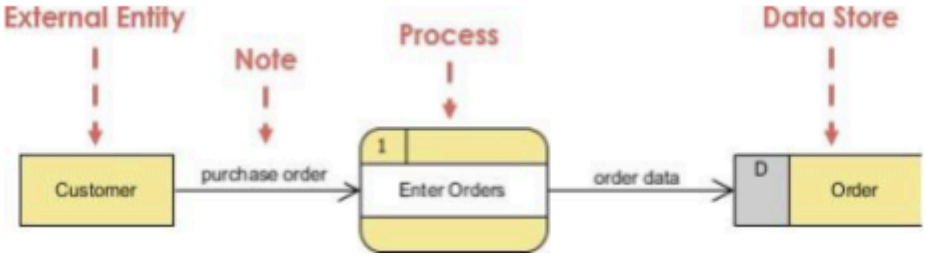


Fig 4.1 Process Notation



Notation

-
-

Process Example

Straight lines with incoming arrows are input data flow
Straight lines with outgoing arrows are output data flows

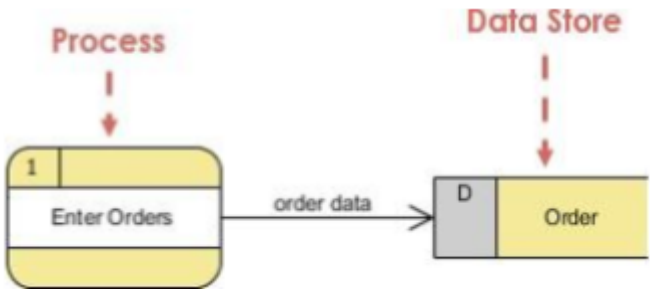


Fig 4.2 Process Example

2. **Data Store:**

A data store or data repository is used in a data-flow diagram to represent a situation when the system must retain data because one or more processes need to use the stored data in a later time.

Notation

- Data can be written into the data store, depicted by an outgoing arrow

Fig 4.3 Data Store Notation

- Data can be read from a data store, which is depicted by an incoming arrow.

Data Store Example

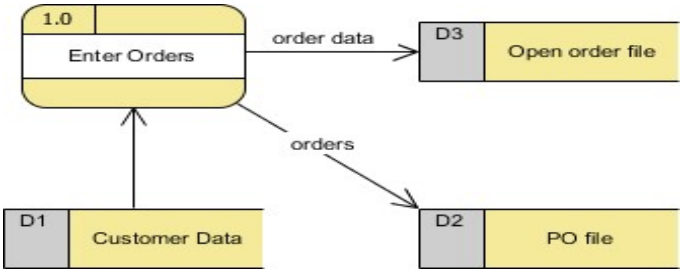


Fig 4.4 Data Store Example

3. **Data Flow**

A data-flow is a path for data to move from one part of the information system to another. A data-flow may represent a single data element such the Customer ID or it can represent a set of data element (or a data structure).

Notation

-
-

Data flow Example

Fig 4.4 Data Flow Example

4. **External Entity**

An external entity is a person, department, outside organization, or other information system that provides data to the system or receives outputs from the system. External entities are components outside of the boundaries of the information systems. They represent how the information system interacts with the outside world.

Notation

- A customer submitting an order and then receive a bill from the system
- A vendor issue an invoice

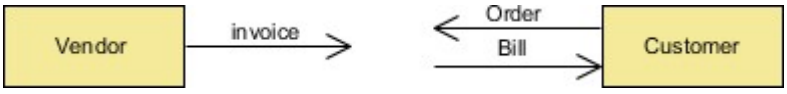


Fig 4.5 External Entity Notation

External Entity Example

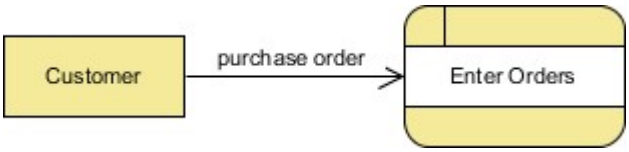


Fig 4.6 External Entity Example

Creating DFD in StarUML:

- Select **Model | Add Diagram | Data Flow Diagram** in Menu Bar or select **Add Diagram | Data Flow Diagram** in Context Menu.
- To create external entities, processes, data flow and data store, select the desired option from toolbox and drag on the main diagram. Upon double clicking, all the respective options will be made available.

The Food Ordering System

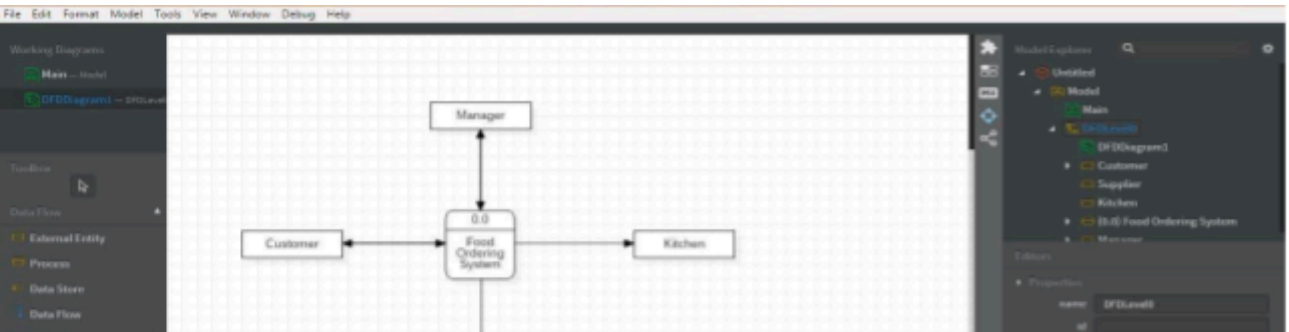
• Context/Level 0 DFD

A context diagram is a data flow diagram that only shows the top level, otherwise known as Level 0. Some of the benefits of a Context Diagram are:

- Shows the overview of the boundaries of a system
- No technical knowledge is required to understand with the simple notation
- Simple to draw, amend and elaborate as its limited notation

The figure below shows a context Data Flow Diagram that is drawn for a Food Ordering System. It contains a process that represents the system to model the "Food Ordering System". It also shows the participants who will interact with the system, called the external entities. In this example, Supplier, Kitchen, Manager and Customer are the entities who will interact with the system. In between the process and the external entities, there are data flow (connectors) that indicate the existence of information exchange between the entities and the system.

Based on the diagram, we know that a Customer can place an Order. The Order Food process receives the Order, forwards it to the Kitchen, store it in the Order data store, and store the updated Inventory details in the Inventory data store. The process also deliver a Bill to the Customer.



Manager can receive Reports through the Generate Reports process, which takes Inventory details and Orders as input from the Inventory and Order data store respectively.
Manager can also initiate the Order Inventory process by providing Inventory order. The process forwards the Inventory order to the Supplier and stores the updated Inventory details in the Inventory data store.

EXERCISES:

1. Construct Level 0 and Level 1 DFD for a "Super Market App" using StarUML. Identify the processes, data stores and external entities clearly. Attach printout.

2. In the "Super Market App", customers can view, search and shop all the required stuff easily. Construct a Level 2 DFD for "Search Item" process of Super Market App using StarUML. Attach printout.

3. Construct a Level 2 DFD for "2.0 Generate Reports" process of "Food Ordering System". Also attach printout.

Lab Session 06

Practice behavioral view UML diagrams for the suggested system: State Chart, Sequence and Collaboration Diagrams

State chart diagram

State machine diagram is a **behavior diagram** which shows discrete behavior of a part of designed system through finite state transitions. State machine diagrams can also be used to express the usage protocol of part of a system. The UML defines the Simple State, Composite State and Submachine state.

Simple State

A **simple state** is a state that does not have sub-states - it has no **regions** and it has no **submachine states**. Simple state is shown as a rectangle with rounded corners and the state name inside the rectangle. Simple state may have compartments. The compartments of the state are:

- name compartment
- internal activities compartment
- internal transitions compartment

Name compartment holds the (optional) name of the state, as a string. States without names are called **anonymous states** and are all considered distinct (different) states. Name compartments should not be used if a name tab is used and vice versa.

Internal activities compartment holds a list of internal actions or state (do) activities (behaviors) that are performed while the element is in the state. The activity label identifies the circumstances under which the behavior specified by the activity expression will be invoked. The behavior expression may use any attributes and association ends that are in the scope of the owning entity. For list items where the expression is empty, the slash separator is optional.
Several labels are reserved for special purposes and cannot be used as event names. The following are the reserved activity labels:

- o entry (behavior performed upon entry to the state)
- o do (ongoing behavior, performed as long as the element is in the state) o exit (behavior performed upon exit from the state)

Internal transition compartment contains a list of internal transitions, where each item has the form as described for trigger. Each event name may appear more than once per state if the guard conditions are different. The event parameters and the guard conditions are optional. If the event has parameters, they can be used in the expression through the current event variable.

Fig 5.1 Simple state with name and internal activities compartments

Creating State Chart Diagram in StarUML

To create a State-chart Diagram:

1. Select first an element where a new Statechart Diagram to be contained as a child.
2. **Model | Add Diagram | Statechart Diagram** in Menu Bar or select **Add Diagram | Statechart Diagram** in Context Menu.
3. Toolbox contains all the required options namely simple states, internal activities (entry, do and exit), transitions, initial and final states etc. to create the required state chart diagram.

Sequence diagram

Sequence diagram is the most common kind of **interaction diagram**, which focuses on the **message** interchange between a numbers of **lifelines**.
Sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines.

The following **fragment**, **interaction use**, **state invariant**, **continuation**, **destruction occurrence**

Major elements of the sequence diagram are shown on the picture below.

nodes and edges are typically drawn in a **UML sequence diagram**: **lifeline**, **execution specification**, **message**, **combined**

Fig 5.2 Major Elements of UML Sequence Diagram

Create Sequence Diagram

To create a Sequence Diagram:

1. Select first an element where a new Sequence Diagram to be contained as a child.
2. Select **Model | Add Diagram | Sequence Diagram** in Menu Bar or select **Add Diagram | Sequence Diagram** in Context Menu.
3. All the major elements of UML sequence diagram (shown above) are available in the toolbox.

Collaboration or communication diagram

Communication diagram (called **collaboration diagram** in UML 1.x) is a kind of UML **interaction diagram** which shows interactions between objects and/or **parts** (represented as **lifelines**) using sequenced messages in a free-form arrangement.
Communication diagram corresponds (i.e. could be converted to/from or replaced by) to a simple **sequence diagram** without structuring mechanisms such as interaction uses and combined fragments. It is also assumed that **message overtaking** (i.e., the order of the receptions are different from the order of sending of a given set of messages) will not take place or is irrelevant.

The following nodes and edges are drawn in a UML communication diagrams: **frame**, **lifeline**, **and message**. These major elements of the communication diagram are shown on the picture below:

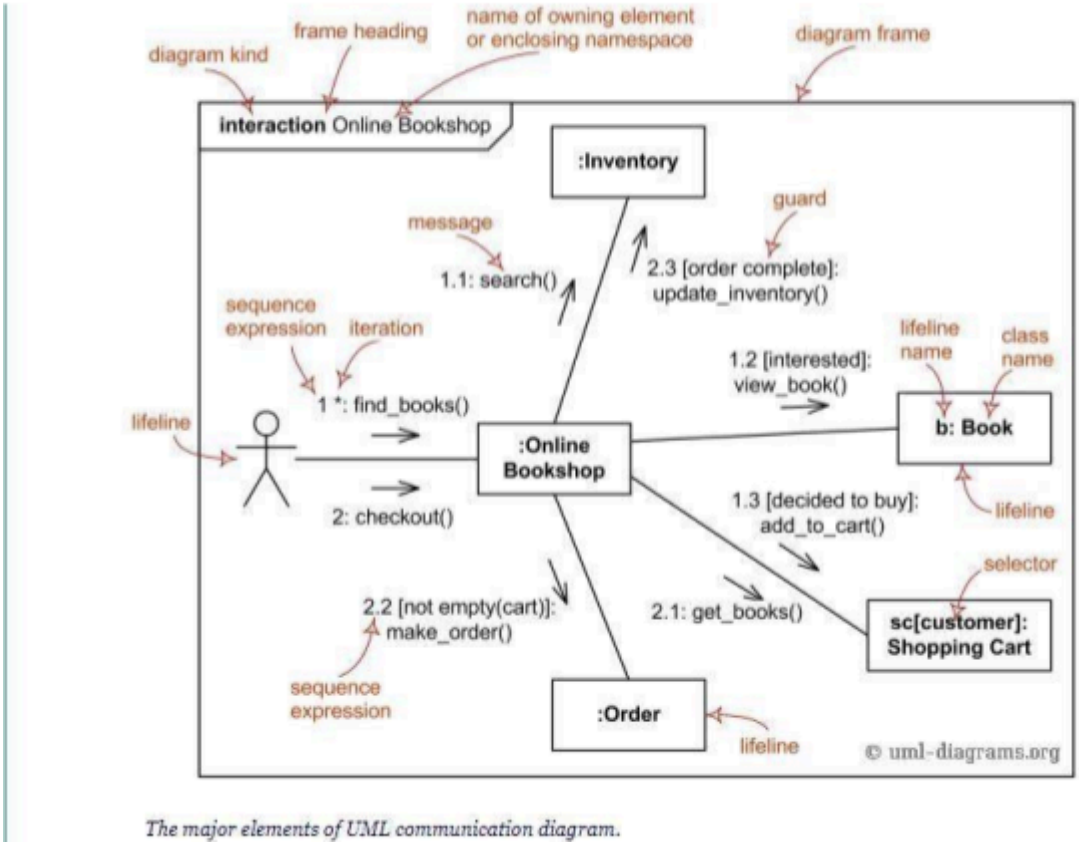


Fig 5.3 Major Elements of UML Communication Diagram

Create Communication/Collaboration Diagram

To create a Communication Diagram:

1. Select first an element where a new Communication Diagram to be contained as a child.
2. Select **Model | Add Diagram | Communication Diagram** in Menu Bar or select **Add Diagram | Communication Diagram** in Context Menu.
3. All the major elements of UML sequence diagram (shown above) are available in the toolbox.

Bank Automated Teller Machine (ATM) System

State Transition / State Chart Diagram

Purpose: An example of UML **behavioral state machine** diagram describing Bank Automated Teller Machine (ATM) top level state machine.
Summary: ATM is initially turned off. After the power is turned on, ATM performs startup action and enters **Self Test** state. If the test fails, ATM goes into **Out of Service** state, otherwise there is **triggerless transition** to the **Idle** state. In this state ATM waits for customer interaction.

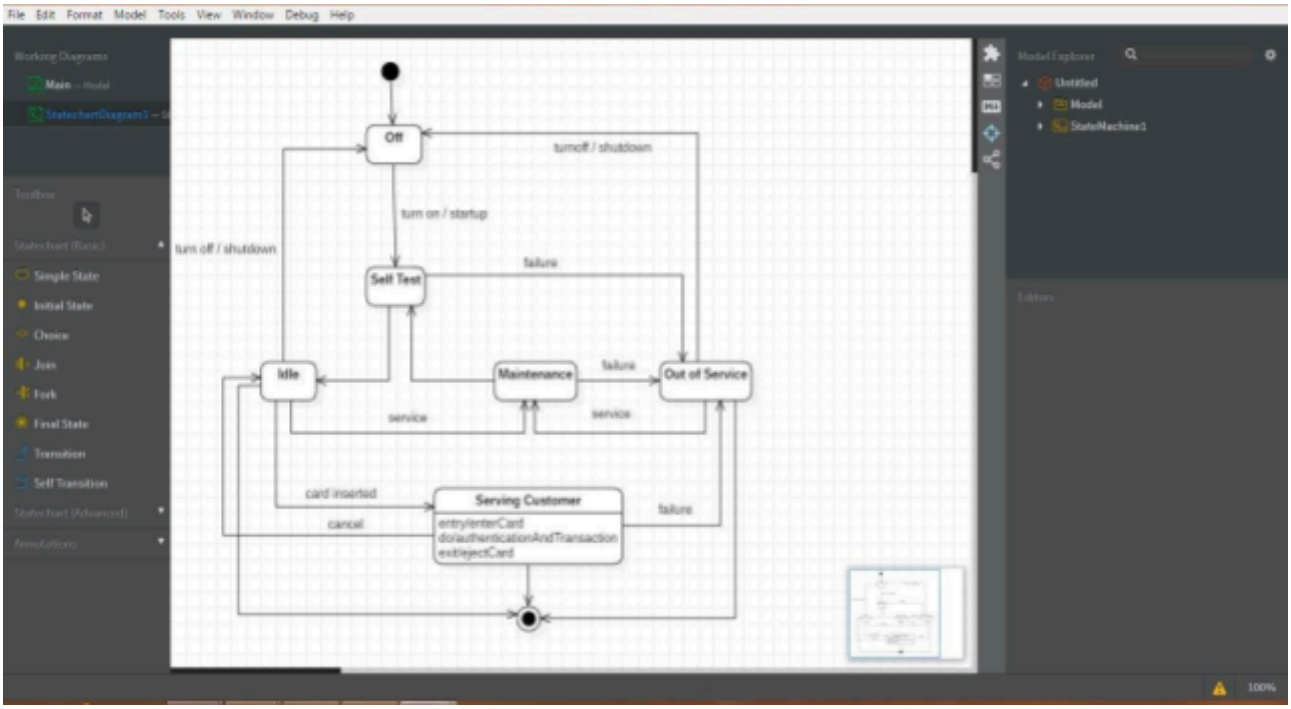


Fig 5.4 State Chart Diagram for ATM System

Sequence and Communication Diagram

Here, customer inserts his card in the ATM machine. The card is verified through bank and after authentication is done, customer is allowed to do transactions. He chooses to withdraw some cash. His request is processed and amount is debited from his account. The customer receives cash, card and receipt from the respective slots.

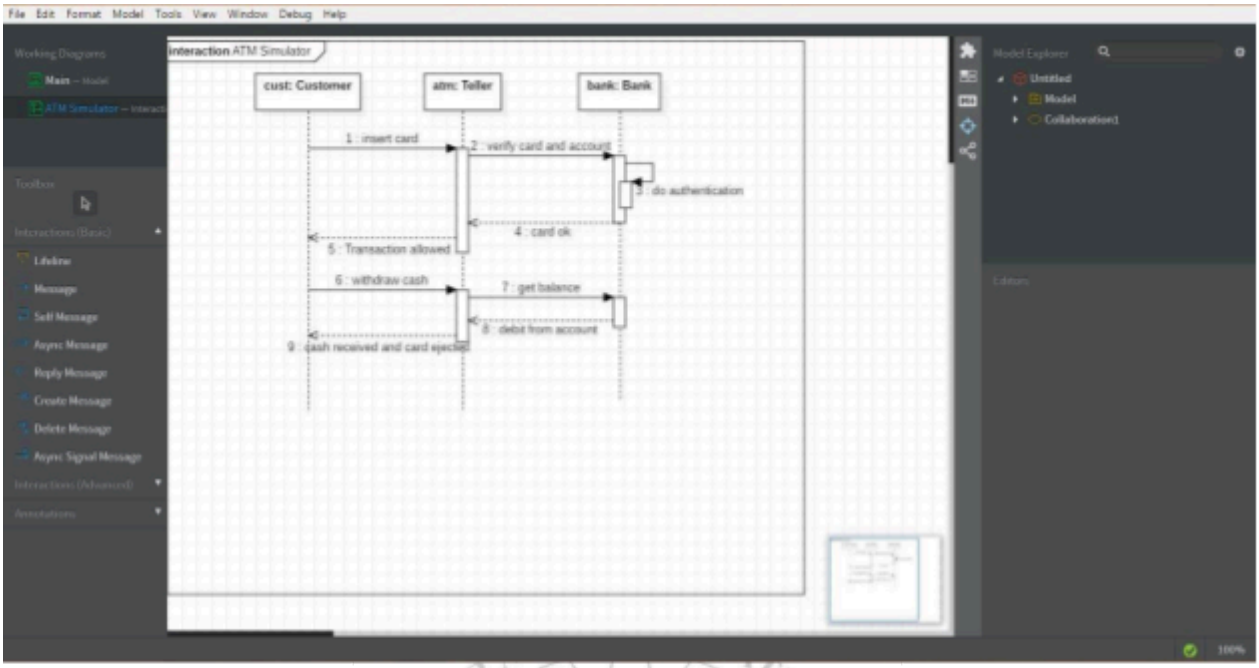


Fig 5.4 UML Sequence Diagram for ATM System

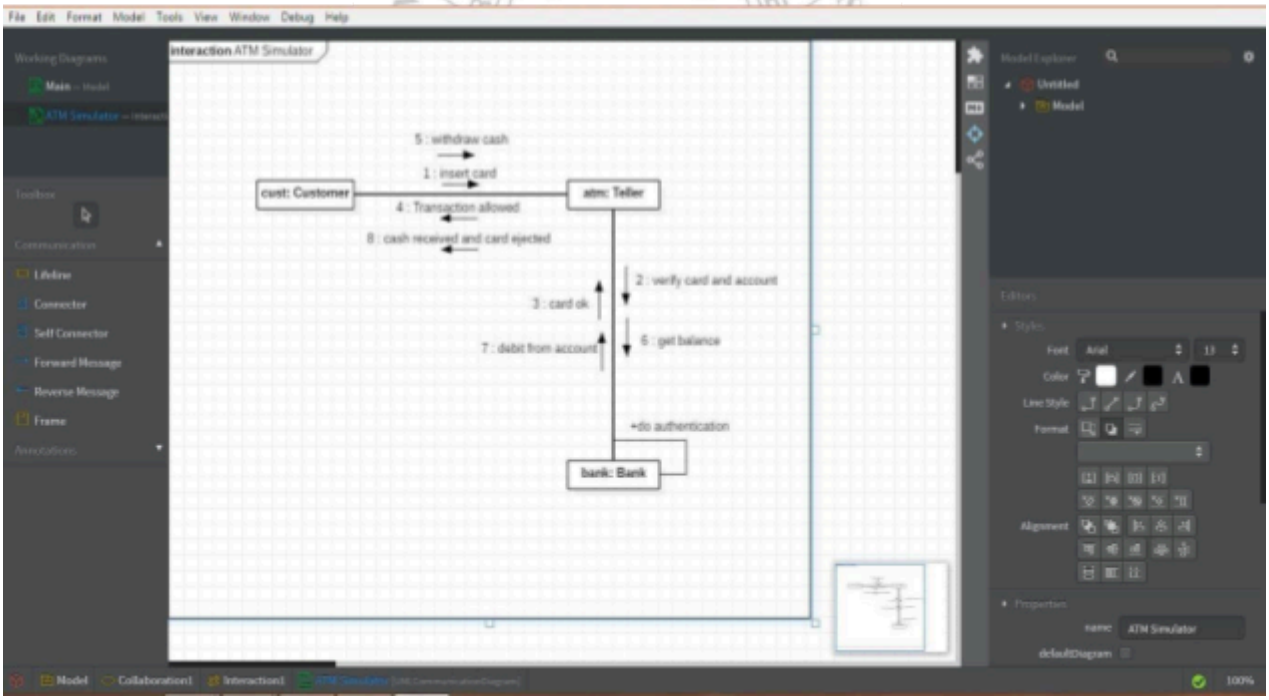


Fig 5.5 UML Collaboration Diagram for ATM System

EXERCISES

1. Create a State Chart Diagram for Microwave Oven Operation using StarUML. Also attach print out.

2. Create Sequence and Collaboration diagrams for Online Bookshop where the Web customer actor can search inventory, view and buy books online. Also attach the printout.

3. UML diagrams can also be created by means of variety of different online tools. Visual Paradigm is one of them. Create any three UML diagrams using this online tool for the 'Online Bookshop System'. Attach print out.

Lab Session 07

Code Quality Analysis

Code quality refers to the degree to which source code is written in a way that meets both functional and non-functional requirements, as well as best practices for readability, maintainability, efficiency, and reliability. High-quality code is:

1. **Readable:** Easy to understand for other developers, reducing time spent on code reviews and debugging.
2. **Maintainable:** Written in a way that changes can be easily made in the future without introducing bugs.
3. **Reliable:** It performs as expected with minimal risk of errors.
4. **Efficient:** It utilizes resources effectively, avoiding unnecessary computations.
5. **Consistent:** Adheres to established coding standards and guidelines.

Code Quality Tools

Several tools are available to help developers assess the quality of their code. Some common code quality tools include:

- **SonarQube**
- **ESLint** (for JavaScript/TypeScript)
- **Pylint** (for Python)
- **Checkstyle** (for Java)
- **Codacy**
- **PMD** (for Java)

These tools provide static analysis of the source code, detecting issues such as code smells, potential bugs, security vulnerabilities, and deviations from coding standards.

Using SonarQube for Code Quality

In this lab, we will use **SonarQube** to check and improve code quality. It will help identify issues early, enforce coding standards, and ensure that our software is reliable and maintainable. By integrating SonarQube into our CI/CD pipeline, we can perform automated code quality checks every time code is committed or merged, maintaining high standards throughout the development lifecycle.

What is SonarQube?

SonarQube is an open-source platform that continuously inspects the quality of

source code and provides comprehensive reports on its health. It helps teams write cleaner and safer code by identifying:

1. **Bugs:** Issues in the code that may lead to incorrect behavior.
2. **Vulnerabilities:** Potential security risks in the source code.
3. **Code Smells:** Poor code practices that can lead to maintainability problems.
4. **Duplications:** Repeated code blocks that increase the maintenance burden.
5. **Test Coverage:** The extent of the code covered by automated tests.

SonarQube uses **static code analysis** to detect issues without running the code. It assigns severity levels to issues, ranging from minor to blocker, and generates a **Code Quality Report** with detailed insights, helping developers prioritize and fix problems early in the development process.

Getting Started with SonarQube Cloud

SonarQube Cloud, also known as **SonarCloud**, is a cloud-based code quality and security tool that integrates with popular DevOps platforms like GitHub, GitLab, Bitbucket, and Azure DevOps. It offers a comprehensive analysis of your projects, providing real-time feedback on code quality, helping you identify bugs, vulnerabilities, and code smells, and enforcing clean code standards.

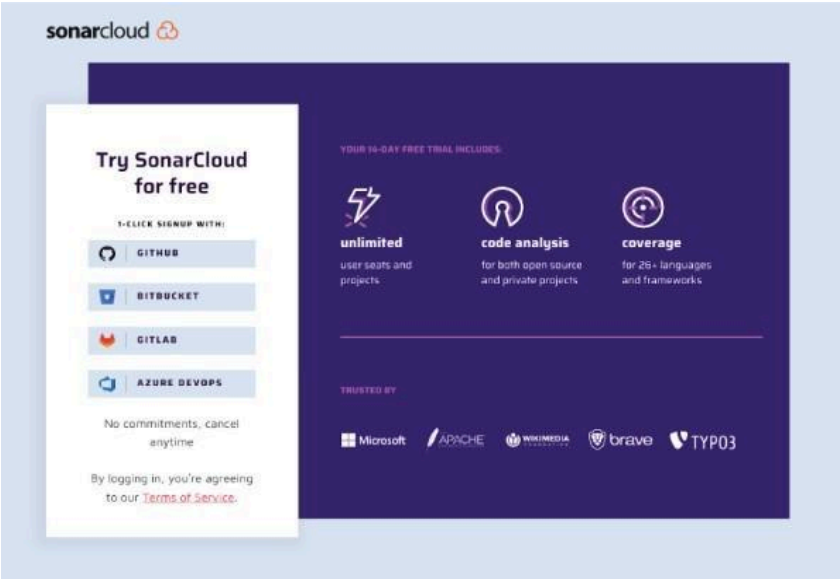
Here's a detailed step-by-step guide on how to get started:

Step 1: Sign Up and Integrate with Your Development Platform

To get started with SonarQube Cloud, visit the SonarCloud Sign Up page. Create an account using your preferred DevOps platform:

- **GitHub**
- **GitLab**
- **Bitbucket**
- **Azure DevOps**

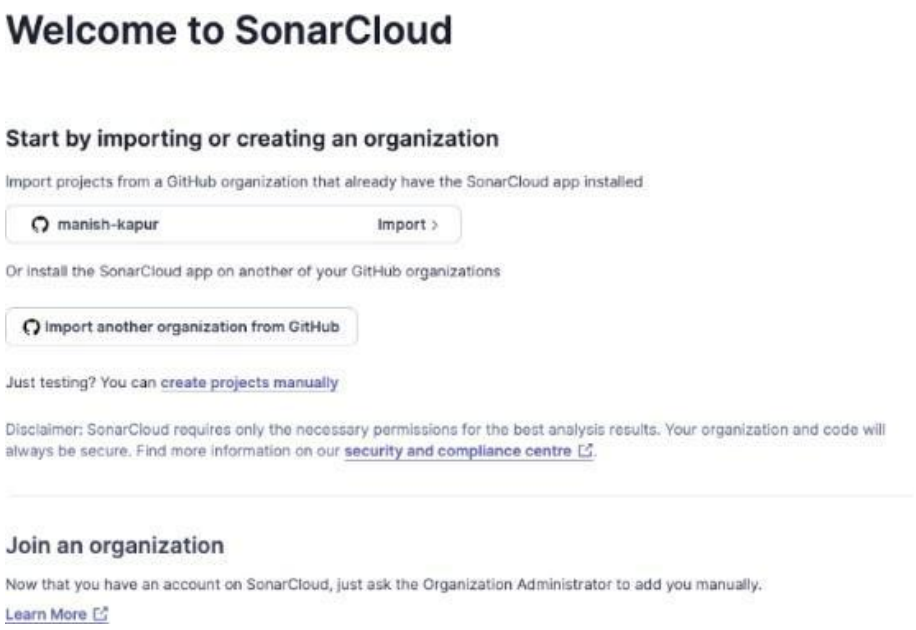
By connecting your DevOps account, SonarQube Cloud will link directly to your repositories, allowing seamless integration. You can choose your GitHub organization or personal account for setup.


sonarcloud
Try SonarCloud for free
1-CLICK SIGNUP WITH:
GITHUB
BITBUCKET
GITLAB
AZURE DEVOPS
No commitments, cancel anytime
By logging in, you're agreeing to our Terms of Service.
YOUR 14-DAY FREE TRIAL INCLUDES:
unlimited
user seats and projects
code analysis
for both open source and private projects
coverage
for 25+ languages and frameworks
TRUSTED BY
Microsoft
APACHE
WIKIMEDIA
brave
TYPO3

Step 2: Set Up Your Organization in SonarQube Cloud

Once you've connected your DevOps account, the next step is to create or select an organization in SonarQube Cloud:

- **Create a New Organization:** You can start fresh by setting up a new organization in SonarQube Cloud.
- **Import an Existing Organization:** If you already have an organization in your GitHub or GitLab account, you can import it directly. The imported organization will automatically include all members from your DevOps platform.

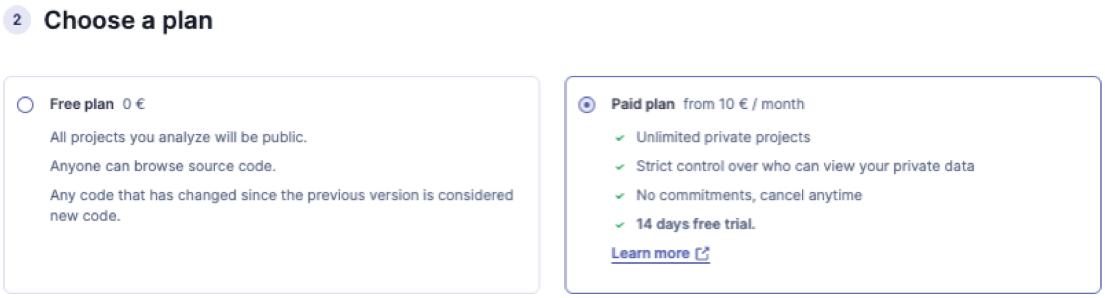


Step 3: Choose a Plan

SonarQube Cloud offers both free and paid plans:

- **Free Plan:** This is suitable for public repositories and includes basic code analysis features.
- **Paid Plan (14-Day Trial):** This plan is required for analyzing private repositories. You can start a no-commitment 14-day free trial. A credit card is required for the trial, but you won't be charged until the trial ends. Pricing is based on the number of Lines of Code (LOC) analyzed.

During the trial, you get access to all features, including advanced code analysis, Quality Gates, and detailed reporting.



Step 4: Import and Analyze Your First Repository

Once your organization is set up, it's time to import your projects:

1. Navigate to your organization's **Projects** tab in SonarQube Cloud.
2. Select the repositories you want to analyze from your GitHub, GitLab, Bitbucket, or Azure DevOps account.
3. Click **Set Up** to start the configuration.

Step 5: Run Your First Code Analysis

SonarQube Cloud offers two methods for running code analysis:

- **Automatic Analysis:** This method triggers analysis automatically when code is pushed to your repository.
- **CI-Based Analysis:** For greater control, you can integrate SonarQube Cloud with your CI/CD tool (e.g., GitHub Actions, GitLab CI/CD). This method allows you to run analysis during your build process using sonar-scanner.

Example GitHub Actions workflow:

```
name: Code Quality Analysis

on: [push, pull_request]

jobs:
  sonarQubeScan:
    runs-on:
    ubuntu-latest steps:
      - uses: actions/checkout@v3
      - name: Run
        SonarScanner run:
```

sonar-scanner

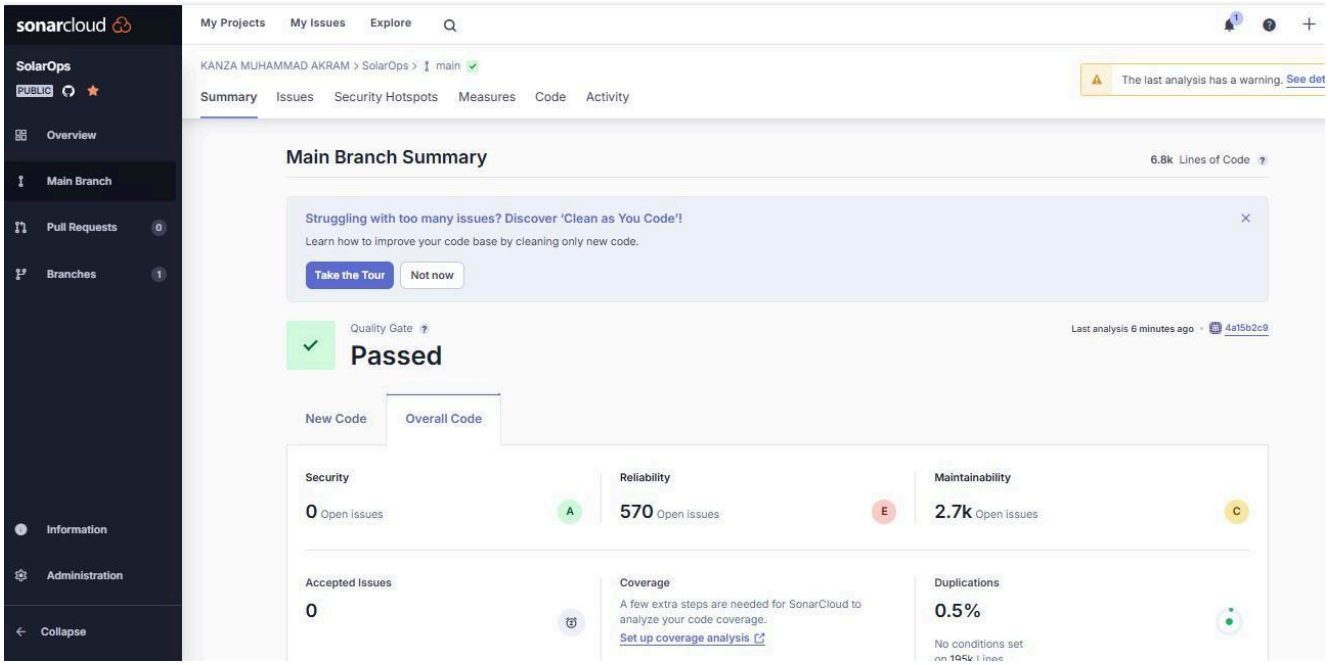
This setup ensures that every new pull request and push is analyzed automatically, providing immediate feedback.

Step 6: Explore the SonarQube Cloud Dashboard

After the first analysis is complete, visit the **SonarQube Cloud dashboard** to review the results. Here's what you can find:

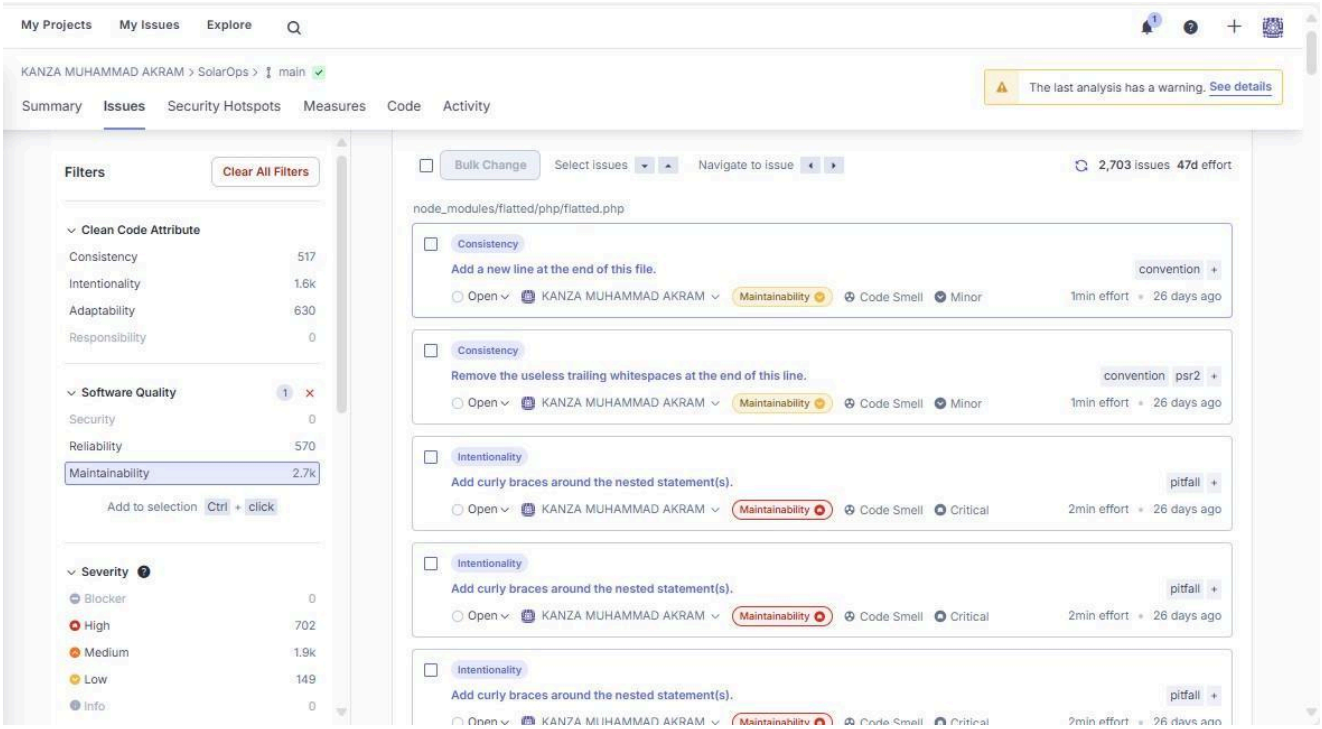
- **Bugs, Vulnerabilities, Code Smells:** See a list of issues detected in your code, along with recommendations for fixing them.
- **Code Coverage:** View the percentage of your code that is covered by tests, helping you identify areas that need more testing.
- **Quality Gate Status:** Check if your project passes the defined Quality Gate criteria, which include thresholds for code coverage, bugs, and vulnerabilities.

Spend time familiarizing yourself with these metrics to prioritize your code quality improvements effectively.



Step 7: Dive Into the Issues and PR Analysis

- When you create a pull request, SonarQube Cloud analyzes the changes and provides feedback right in the PR interface.
- Issues detected in the pull request are displayed, and the quality gate status is shown. If the code does not meet the defined criteria, the pull request can be blocked from merging.


My Projects
My Issues
Explore
KANZA MUHAMMAD AKRAM > SolarOps > main
Summary
Issues
Security Hotspots
Measures
Code
Activity
The last analysis has a warning. See details
Filters
Clear All Filters
Clean Code Attribute
Consistency 517
Intentionality 1.6k
Adaptability 630
Responsibility 0
Software Quality 1
Security 0
Reliability 570
Maintainability 2.7k
Add to selection Ctrl + click
Severity
Blocker 0
High 702
Medium 1.9k
Low 149
Info 0
Bulk Change
Select issues
Navigate to issue
2,703 issues 47d effort
node_modules/flatted/php/flatted.php
Consistency
Add a new line at the end of this file.
convention
Open
KANZA MUHAMMAD AKRAM
Maintainability
Code Smell
Minor
1min effort • 26 days ago
Consistency
Remove the useless trailing whitespaces at the end of this line.
convention psr2
Open
KANZA MUHAMMAD AKRAM
Maintainability
Code Smell
Minor
1min effort • 26 days ago
Intentionality
Add curly braces around the nested statement(s).
pitfall
Open
KANZA MUHAMMAD AKRAM
Maintainability
Code Smell
Critical
2min effort • 26 days ago
Intentionality
Add curly braces around the nested statement(s).
pitfall
Open
KANZA MUHAMMAD AKRAM
Maintainability
Code Smell
Critical
2min effort • 26 days ago
Intentionality
Add curly braces around the nested statement(s).
pitfall

Lab Session 08:

“SOFTWARE TESTING”

SOFTWARE TESTING:

Software testing is a crucial process in the software development lifecycle, aimed at evaluating and verifying that a software application functions as intended. It helps identify defects, ensures reliability, enhances performance, and confirms that the software meets specified
requirements. Software testing can take various forms, including manual testing and automation testing, each with its own methodologies, benefits, and challenges.

Automation testing:

Automation testing, on the other hand, uses scripts and tools to perform test cases, making it ideal for repetitive, high-volume, or complex tasks. Automation testing is efficient for regression testing, performance testing, and tasks that require consistent accuracy. Although it can require a significant initial investment in creating scripts, automation testing saves time and resources in the long run by reducing repetitive manual work and providing faster feedback on code changes.

Manual testing:

Manual testing involves human testers who execute test cases manually without using automated tools. This type of testing is highly effective for exploratory, usability, and ad-hoc testing, where human insight is essential to understand subtle issues in user interfaces or unique user scenarios. Manual testing can be time-consuming and prone to human error, but it provides valuable insights that automated testing might overlook.

Steps for Manual Testing:

Manual testing involves a structured approach to thoroughly validate software functionality.

1. Test Scenario

- Define the Test Scenario: "Verify the login functionality of Facebook login page." This scenario provides a high-level view of the feature being tested.

2. Test Case ID

- Assign a Unique Identifier: Label each test case with a unique identifier (e.g., TC-001) to ensure easy tracking and reference.

3. Test Case Description

- Description of Each Test Case: Provide a clear and concise description of what each test case aims to validate, such as "Verify that a user can log in with valid credentials."

4. Preconditions

- List Preconditions: Note any requirements that must be in place before testing begins, such as "User must have a registered Facebook account" or "Ensure internet connectivity."

5. Test Steps

- Define Each Step in the Test: Outline specific actions for the tester to follow, e.g.,

  - Open the Facebook login page.
  - Enter valid username in the username field.
  - Enter valid password in the password field.
  - Click the "Log In" button.

6. Expected Result

- State the Expected Outcome: Define what the expected result should be if the test passes, such as "User is redirected to the homepage after successful login."

7. Actual Result

- Record the Actual Outcome: After executing the test, note whether the actual result matches the expected result (e.g., “User successfully logged in and redirected to the homepage”).

8. Status (Pass/Fail)

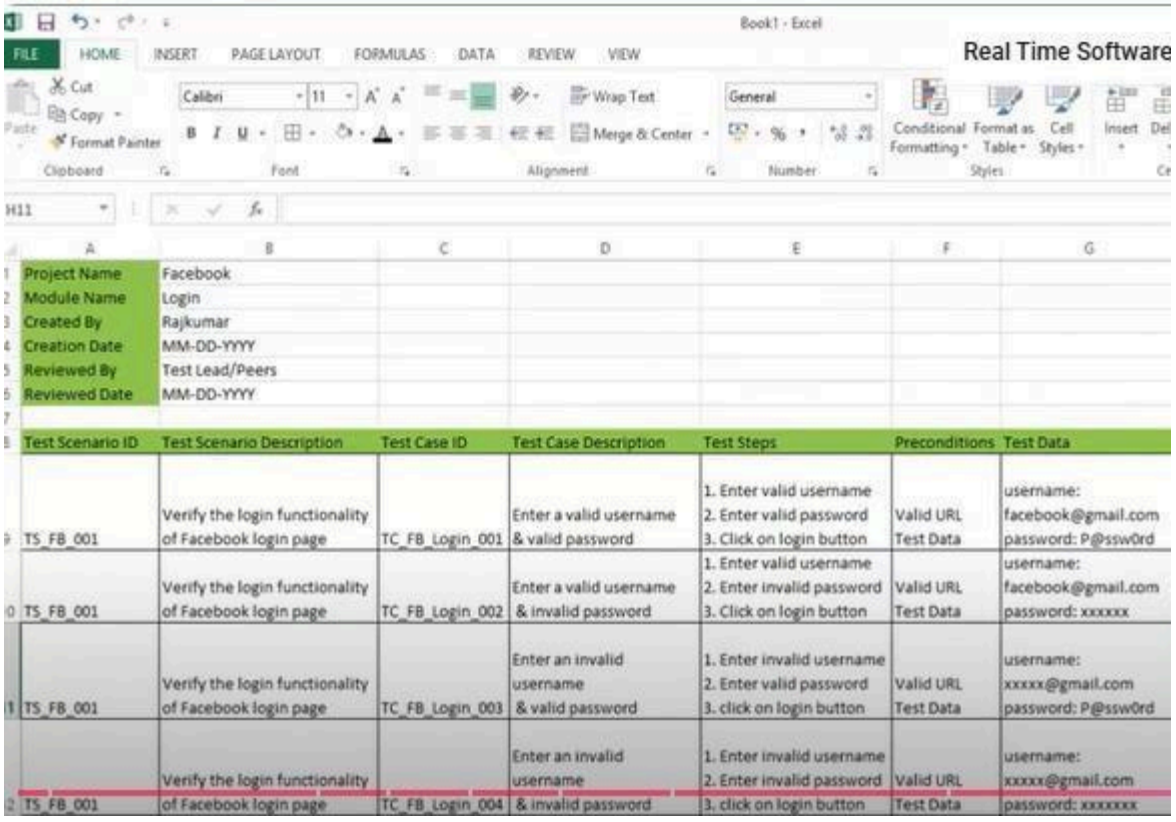
- Mark Test Status: Based on the comparison of actual versus expected results, mark the test as Pass or Fail.

9. Remarks

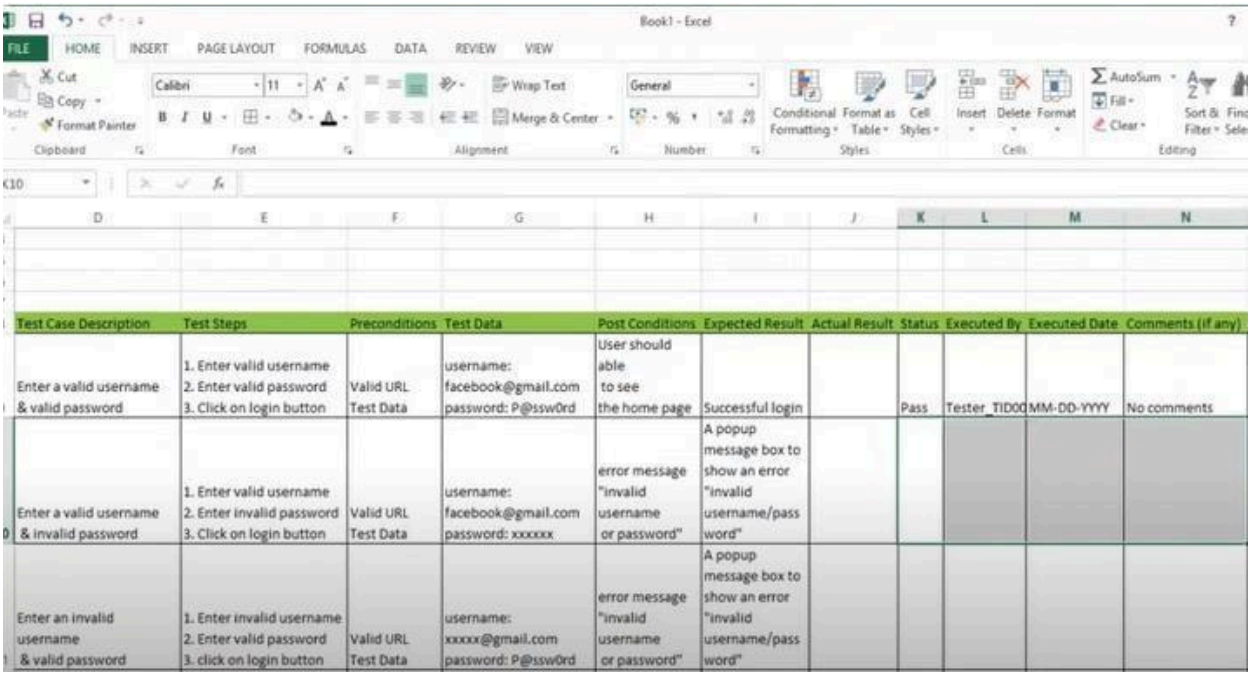
- Provide Additional Notes: Include any observations or comments related to the test, especially if it fails (e.g., “Login fails if password contains special characters”).

Example Test scenario:

Verify the login functionality of Facebook login page



	A	B	C	D	E	F	G
1	Project Name	Facebook					
2	Module Name	Login					
3	Created By	Rajkumar					
4	Creation Date	MM-DD-YYYY					
5	Reviewed By	Test Lead/Peers					
6	Reviewed Date	MM-DD-YYYY					
7							
8	Test Scenario ID	Test Scenario Description	Test Case ID	Test Case Description	Test Steps	Preconditions	Test Data
9	TS_FB_001	Verify the login functionality of Facebook login page	TC_FB_Login_001	Enter a valid username & valid password	1. Enter valid username 2. Enter valid password 3. Click on login button	Valid URL Test Data	username: facebook@gmail.com password: P@ssw0rd
10	TS_FB_001	Verify the login functionality of Facebook login page	TC_FB_Login_002	Enter a valid username & invalid password	1. Enter valid username 2. Enter invalid password 3. Click on login button	Valid URL Test Data	username: facebook@gmail.com password: xxxxxx
11	TS_FB_001	Verify the login functionality of Facebook login page	TC_FB_Login_003	Enter an invalid username & valid password	1. Enter invalid username 2. Enter valid password 3. click on login button	Valid URL Test Data	username: xxxxx@gmail.com password: P@ssw0rd
12	TS_FB_001	Verify the login functionality of Facebook login page	TC_FB_Login_004	Enter an invalid username & invalid password	1. Enter invalid username 2. Enter invalid password 3. click on login button	Valid URL Test Data	username: xxxxx@gmail.com password: xxxxxxx



	D	E	F	G	H	I	J	K	L	M	N
	Test Case Description	Test Steps	Preconditions	Test Data	Post Conditions	Expected Result	Actual Result	Status	Executed By	Executed Date	Comments (if any)
	Enter a valid username & valid password	1. Enter valid username 2. Enter valid password 3. Click on login button	Valid URL Test Data	username: facebook@gmail.com password: P@ssw0rd	User should able to see the home page	Successful login		Pass	Tester_TID00	MM-DD-YYYY	No comments
10	Enter a valid username & invalid password	1. Enter valid username 2. Enter invalid password 3. Click on login button	Valid URL Test Data	username: facebook@gmail.com password: xxxxxx	error message "invalid username or password"	A popup message box to show an error "invalid username/password"					
11	Enter an invalid username & valid password	1. Enter invalid username 2. Enter valid password 3. click on login button	Valid URL Test Data	username: xxxxx@gmail.com password: P@ssw0rd	error message "invalid username or password"	A popup message box to show an error "invalid username/password"					

Generic Field Validation Test Cases:

Generic Field Validation Test Cases fall under **manual testing** as they involve manually verifying that fields in the application accept valid data inputs, enforce constraints, and meet required specifications. These test cases are usually documented in advance, and testers execute them manually by entering different values into fields to confirm the application's behavior. These test cases are designed to ensure that fields like input boxes, dropdowns, and checkboxes meet specified requirements, such as:

- **Input Length Validation**: Verifying minimum and maximum character limits.
- **Format Validation**: Ensuring data entered meets required formats (e.g., email, phone number).
- **Boundary Testing**: Testing field limits to verify constraints.
- **Required Field Validation**: Checking that mandatory fields don’t accept blank inputs.
- **Special Character Handling**: Confirming fields appropriately handle or restrict special characters as required.

These tests ensure robust data handling and provide consistency in data entry fields across the application, helping to catch potential user entry errors early.

A	B	C	D	E	F
	Application Name	<mention application name here>	Version	<version of the application>	
	Testing Type	<manual/automation>	Testing Start Date	<testing start date>	
	Browser	<chrome/IE/safari/others>	Testing End Date	<testing end date>	
	Browser version	<version of the browser on which testing is done>	Cycle No.	<1/2/3>	
	FrontEnd Server Name	<frontend server name on which testing is done>	CR Status	<open/close/deferred/hold>	
	BackEnd Server Name	<backend server name on which testing is done>	Prepared By		
	DB Name	<DB name on which testing is done>	Reviewed By		
			Tested By		

Test Planning Checklist:

A **Test Planning Checklist** serves as a structured tool to ensure all aspects of test planning are addressed thoroughly used in both **manual** and **automation** testing. The checklist typically

includes items like defining scope, identifying resources, setting timelines, and establishing test criteria. This checklist is useful to ensure no critical areas are missed during planning, especially in comprehensive projects with ample time and resources. Following this checklist is ideal for
large, detailed projects where requirements are extensive and thorough documentation and planning are essential.

When to Follow the Test Planning Checklist:

- When the project has a significant scope and multiple modules that require detailed planning.
- When there are ample time and resources available to create a thorough plan.
- When the project demands high quality, such as in critical applications, and includes dedicated phases for test preparation.

	A	B	C	D	E
1	**Test Planning Checklist**				
2	**Review Date**				
3	**Reviewer**				
4	**Project Code**				
5	**Project Name**				
6	**PM Name**				
7	**Document Name**				
8	**Author**				
9	**Work Product Name , ID & Version Number**				
10					
11	**Sl.No**	**Description**	**Yes/No/NA**	**Defect ID**	**Remarks**
13	**1**	**Requirements based planning**			
14	1.1	Have you described how traceability of testing to requirements is to be demonstrated (e.g. references to the specified functions and requirements)?			
15	1.2	Do all test cases agree with the specification of the function or requirement to be tested?			
16	1.3	Have you defined comprehensive test objectives?			
17	1.4	Have you documented the purpose and the capability demonstrated by each test case? Alternatively have the test objectives been mapped to test cases?			
18	1.5	All the in scope activities and out scope are clearly defined?			
19	1.6	Is the entry, exit, test coverage, suspension and resumption criteria are clearly defined and adequate?			
20	1.7	Have all the Test environment, test data and database setup has been defined in the test strategy/test plan is same as specified by the customer?			
	1.8	Are all the clarifications tracked and closed? The same has			

	A	B	C	D	E
21		been translated in to the test plan wheverever applicable?			
22	**2 Time frames and project management issues**				
23	2.1	Have you identified Testing as a distinct phase in the SDLC planning of the project?			
24	2.2	Does the test planning start early enough during the project execution?			
25	2.3	Have you planned for an overall testing schedule and the personnel required, and associated training requirements?			
26	2.4	Have the test team members been given specific assignments?			
27	2.5	Are the testing schedule, time frames & resources adequate?			
28	2.6	Have you planned for post testing activities (e.g. Bug fixing) and multiple test cycles?			
29	2.7	Have you planned the SCM to manage test cases, test data and test results?			
30	**3 Testing Tools**				
31	3.1	Have you identified testing tools required for the purpose? (E.g. Test case generators, test data generators, test measurement tools, test harness etc.)			
32	3.2	Have identified testing tool(s) evaluated?			
33	3.3	Have technical support for identified testing tool addressed?			
34	3.4	Have you described the software configuration management tools needed to implement the designed test cases?			
35	3.5	Have you described the hardware configuration and resources needed to implement the designed test cases? (E.g. Golden Simulator)			

	A	B	C	D	E
36	3.6	Are the testing tools available with the team? Alternatively, is the procurement likely to be complete by the time testing starts in the project?			
37	3.7	Are the testing tools client supplied and have the risk factors for the availability of the testing tools identified?			
38	**4 Defect Management**				
39	4.1	Have you described the way in which test are to be monitored and recorded?			
40	4.2	Have you defined criteria for evaluating the test results?			
41	4.3	Have you considered requirements for regression testing?			
42	4.4	Have you determined the criteria on which the completion of the test will be judged?			
43	4.5	Is the bug tracking and closure mechanism clearly identified?			
44	4.6	Have the identified defect management tool(s) evaluated?			
45	**5 Testability, Verifiability & Reliability**				
46	5.1	Does requirement have specific validation criteria?			
47	5.2	Is the minimum and maximum configuration specified?			
48	5.3	Is a strategy for error detection specified?			
49	5.4	Is a strategy for correction specified?			
50	5.5	Are all simple and compound boundaries for testing each requirement identified?			
51	5.6	For each requirement is there a test(s) that can be executed to verify the requirement?			
52	5.7	Is there a requirement that may have potential problem in testing?			
53	5.8	Physical lab space for the test-setup and other test			

	A	B	C	D	E
53		equipments accessories are arranged, if required?			
54	5.9	Have you described the way in which test are to be monitored and recorded?			
55	**6 Types of tests and their coverage etc.**				
56	6.1	Have you established test plans and test procedures for module testing, integration testing, system testing, and acceptance Testing?			
57	6.2	Have you defined test cases for performance tests, boundary tests, and usability tests?			
58	6.3	Have you defined test cases for white-box-testing (structural tests)?			
59	6.4	Have you designed test cases for stress tests (intentional attempts to break system)?			
60	6.5	Have you designed test cases with special input values (e.g. empty files)?			
61	6.6	Have you designed test cases with default input values?			
62	6.7	Have you sufficiently considered error cases?			
63	6.8	Have you designed test cases for invalid and unexpected input conditions as well as valid conditions?			
64	6.9	Have you unambiguously provided test input data and expected test results or expected messages (exception / error messages (if any)) for each test case?			
65	6.10	Is test case description and expected result available for all test cases?			
66	6.11	Are the test data embedded into test cases?			
67	6.12	Are there enough steps to validate the various fields?			
68	6.13	Are there test cases to cover all the possible system outputs?			

	A	B	C	D	E
69	6.14	Is a separate test case included for multiple verifications of the application's behavior?			
70	6.15	Have you designed test cases with special input values (e.g. empty files)?			
71	6.16	Have you covered all the possible system outputs in the test cases?			
72	6.17	Have all the requirements in the FS/Requirement catalogue covered in the test conditions?			
73	6.18	Is there a traceability of the test conditions to the Requirements?			
74	6.19	Is the Test condition numbering proper?			
75	6.20	Are possible test conditions covered as per the test strategy/test plan document?			
76	6.21	Is there any clarification outstanding? If so, has reference been provided against related conditions?			
77	6.22	Has conditions included / excluded based on response to clarifications?			
78	6.23	Is the correct requirement ID mapped to the appropriate test condition?			
79	6.24	Is version control maintained?			
80	**7 Timing and Performance**				
81	7.1	Test case(s) for "processing times" identified and documented?			
82	7.2	Test case(s) for "Data transfer Rates" identified and documented?			
83	7.3	Test case(s) for "Throughput Rates" identified and documented?			
84	**8 Test Automation**				
85	8.1	Has the testing been automated?			
86	8.2	If not, are the constraints mentioned in test plan, significant enough to prevent automation of testing?			
	8.3	Any tools or s/w required for the testing automation process			
87		identified and arranged?			
88	**9 Test Measurement**				
89	9.1	Is it possible to meet and to measure all test objectives defined (e.g. test coverage)?			
90	9.2	Have you stated the level of coverage to be achieved?			

Lab Session 09
Installing the Virtual Machine Lab Environment

Objectives:

This lab guides you through the steps required to:

- Prepare your computer for virtualization.
- Navigate and explore the GUI of the DEVASC virtual machine (VM).
- Set up accounts needed for lab activities.
- Install and configure Webex Teams for collaboration.

Background / Scenario

In this lab, you will install the DEVASC virtual machine (DEVASC VM) using Oracle VirtualBox. After successfully setting up the environment, you will explore its GUI interface. Next, you will

create accounts for the required services used throughout the labs. Finally, you will install Webex Teams for class communication and future lab activities.

Required Resources

- A host computer with at least **4 GB of RAM** and **15 GB of free disk space**.
- **High-speed internet access** to download Oracle VirtualBox and the DEVASC VM.

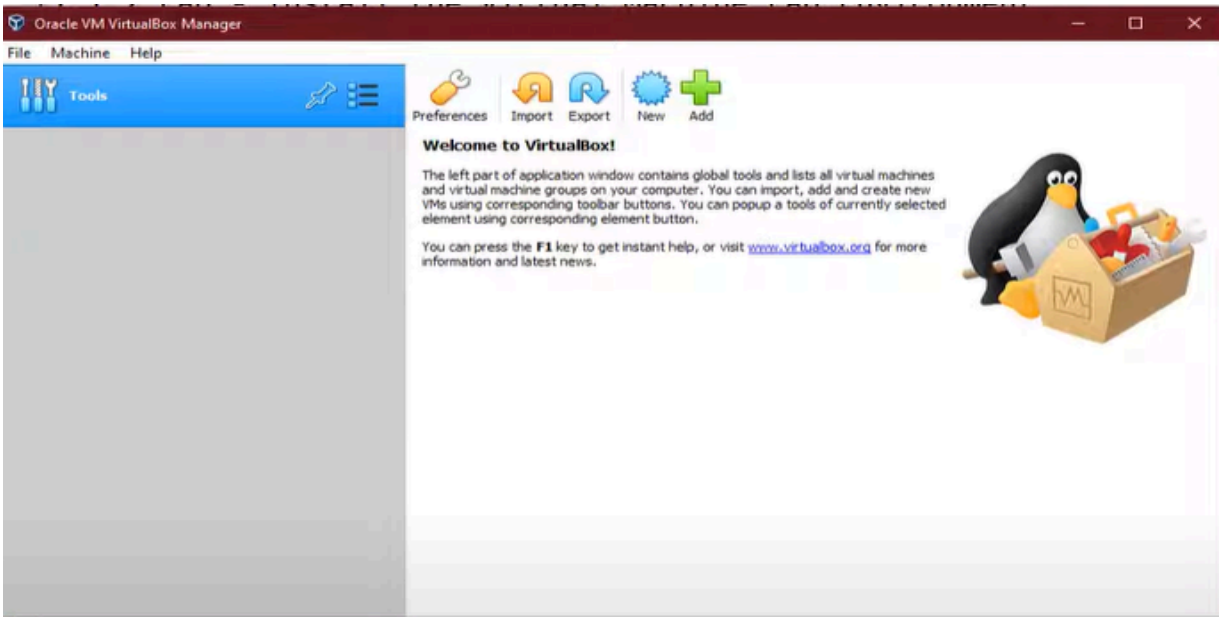
Instructions

Part 1: Prepare a Computer for Virtualization

Step 1: Download and Install VirtualBox

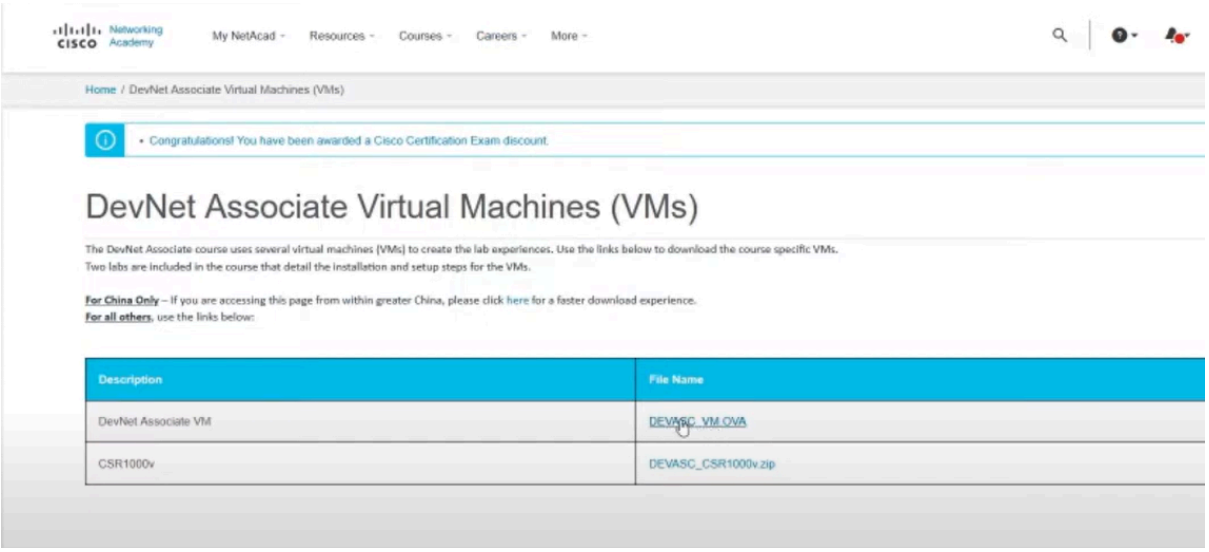
1. Navigate to VirtualBox and click the download link.
2. Select the appropriate VirtualBox installation file for your operating system.
3. Run the installer and accept the default installation settings.

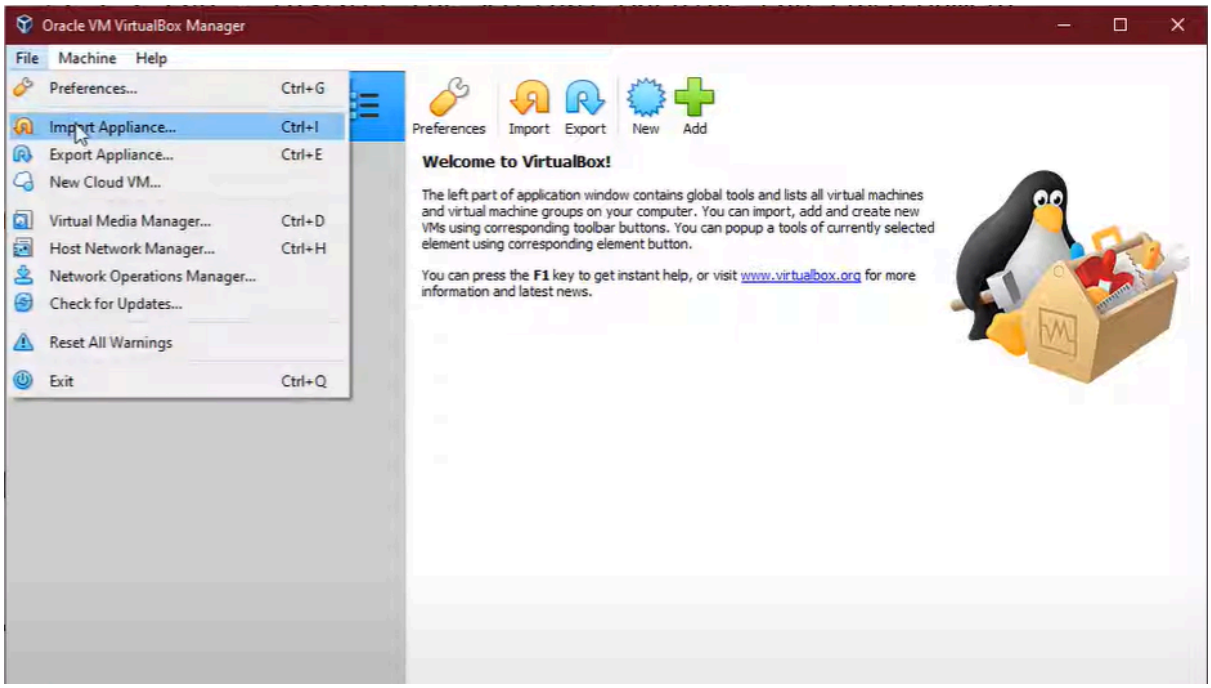
4. Open VirtualBox; it is now ready for use.

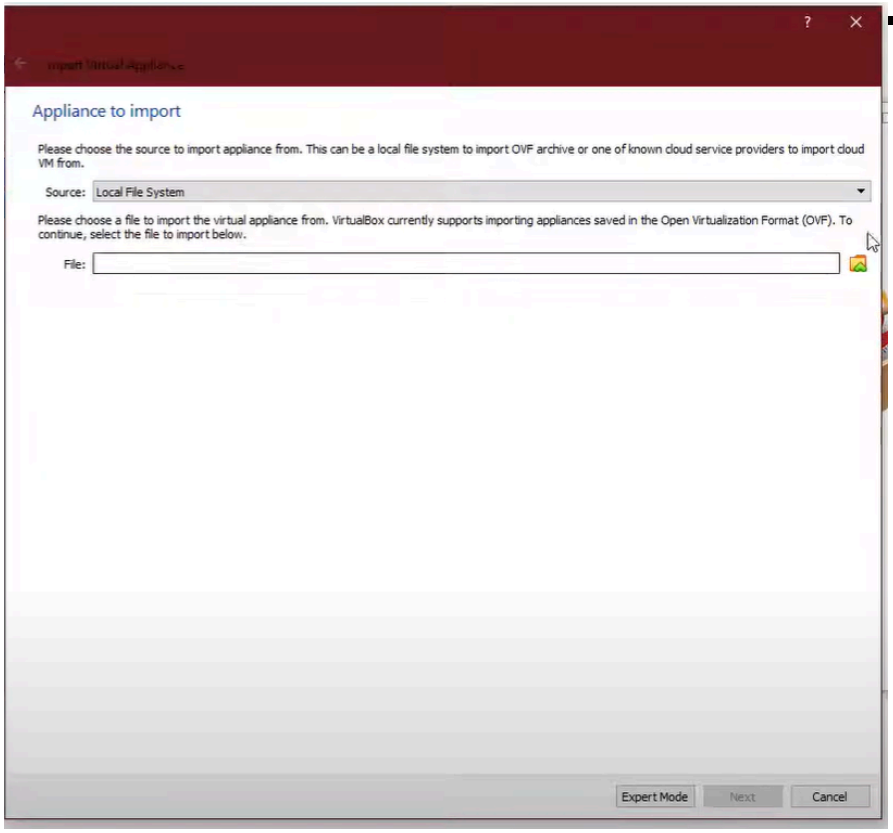


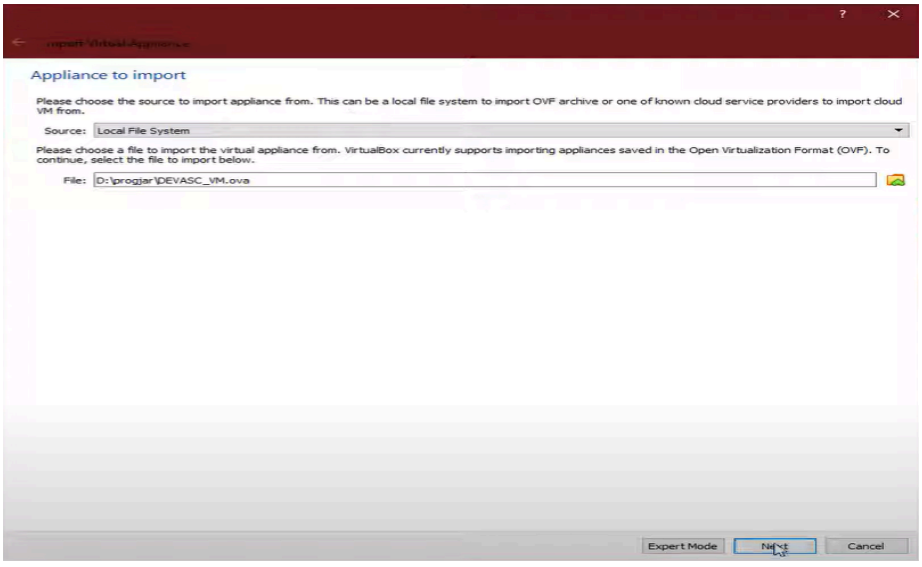
Step 2: Import the DEVASC VM

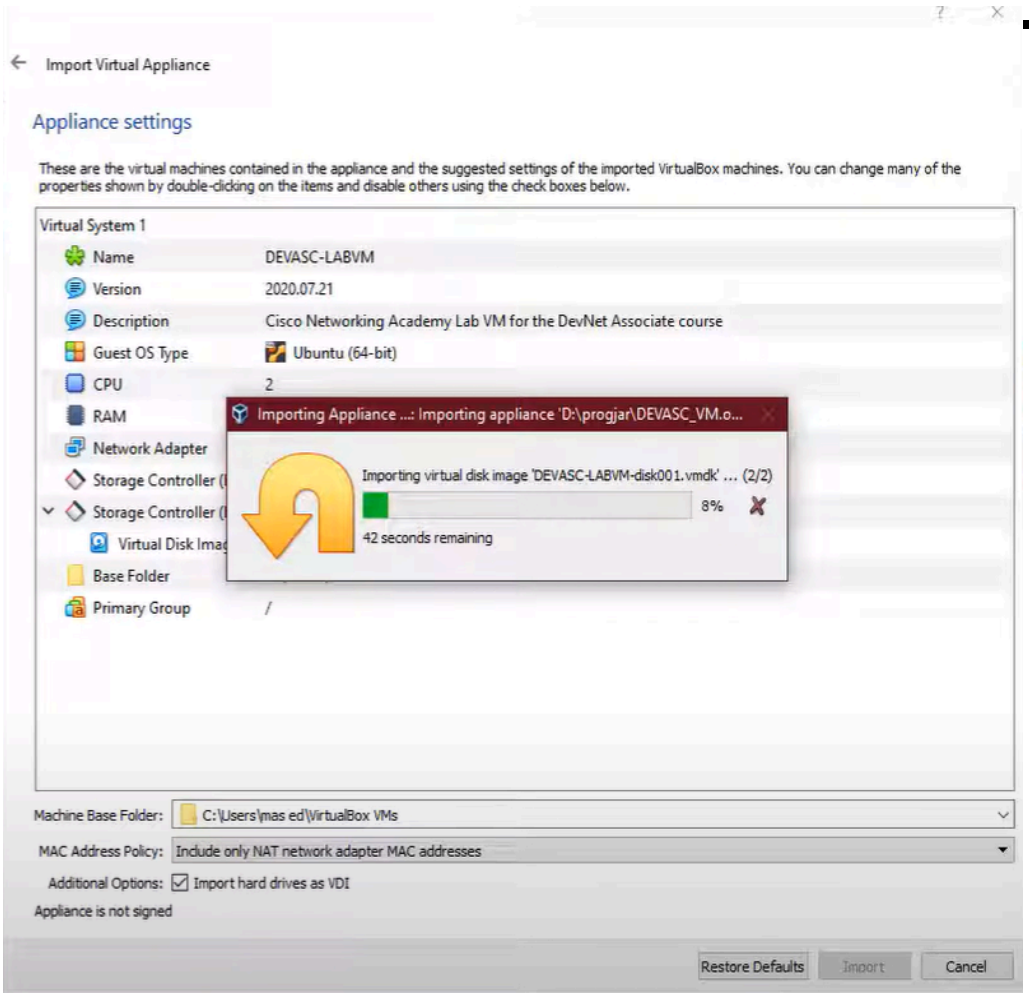
1. Download the DEVASC VM from the Cisco Networking Academy website.
2. Open VirtualBox, and go to **File > Import Appliance**.
3. Browse to the downloaded DEVASC VM file, select it, and click **Open**.
4. Click **Import** to begin the process, which may take a few minutes.

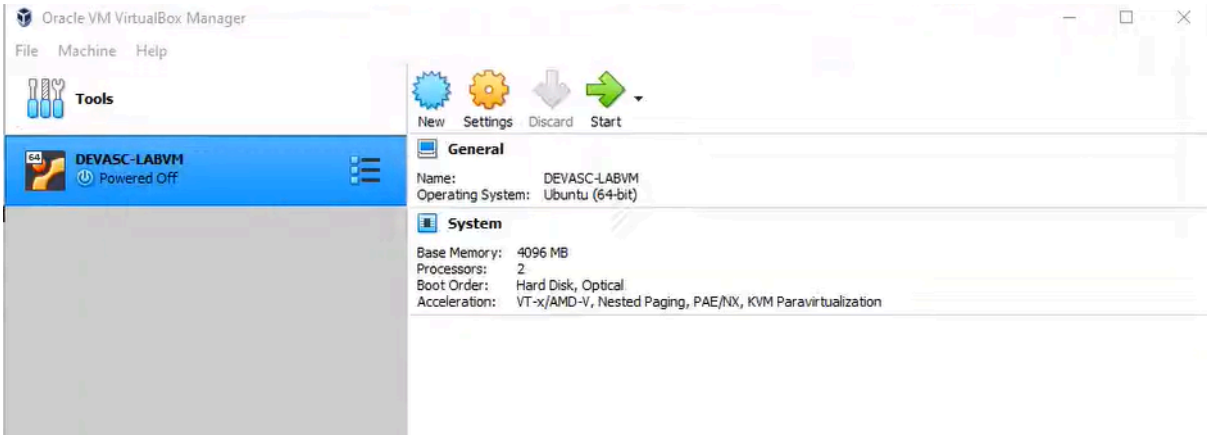





Appliance to import
Please choose the source to import appliance from. This can be a local file system to import OVF archive or one of known cloud service providers to import cloud VM from.
Source:
Local File System
Please choose a file to import the virtual appliance from. VirtualBox currently supports importing appliances saved in the Open Virtualization Format (OVF). To continue, select the file to import below.
File:
Expert Mode
Next
Cancel


Appliance to import
Please choose the source to import appliance from. This can be a local file system to import OVF archive or one of known cloud service providers to import cloud VM from.
Source:
Local File System
Please choose a file to import the virtual appliance from. VirtualBox currently supports importing appliances saved in the Open Virtualization Format (OVF). To continue, select the file to import below.
File:
D:\proggar\DEVASC_VM.ova
Expert Mode
Next
Cancel

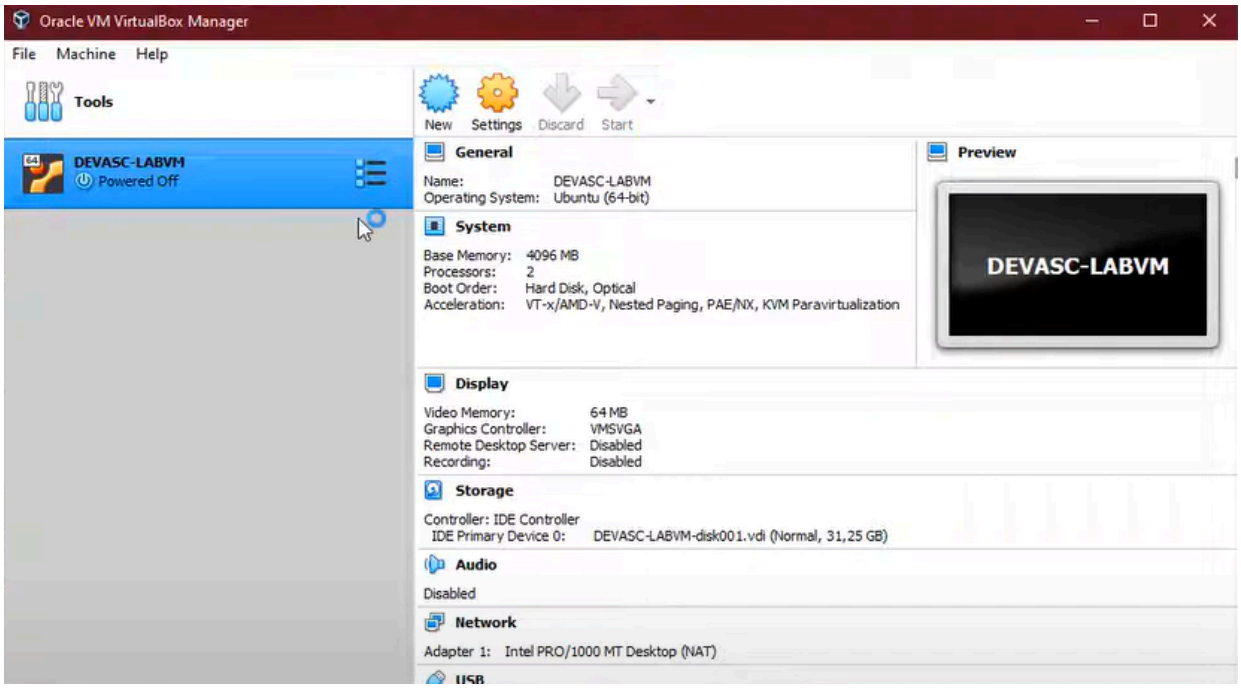


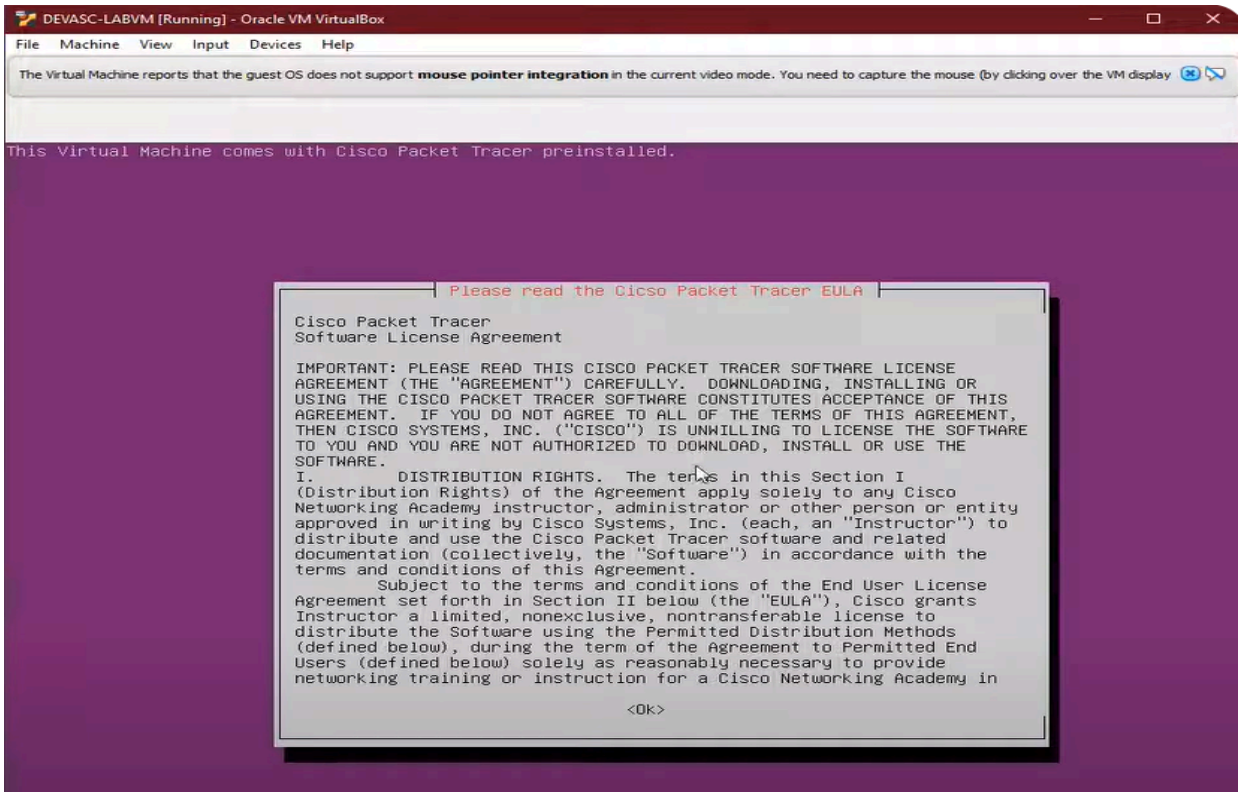


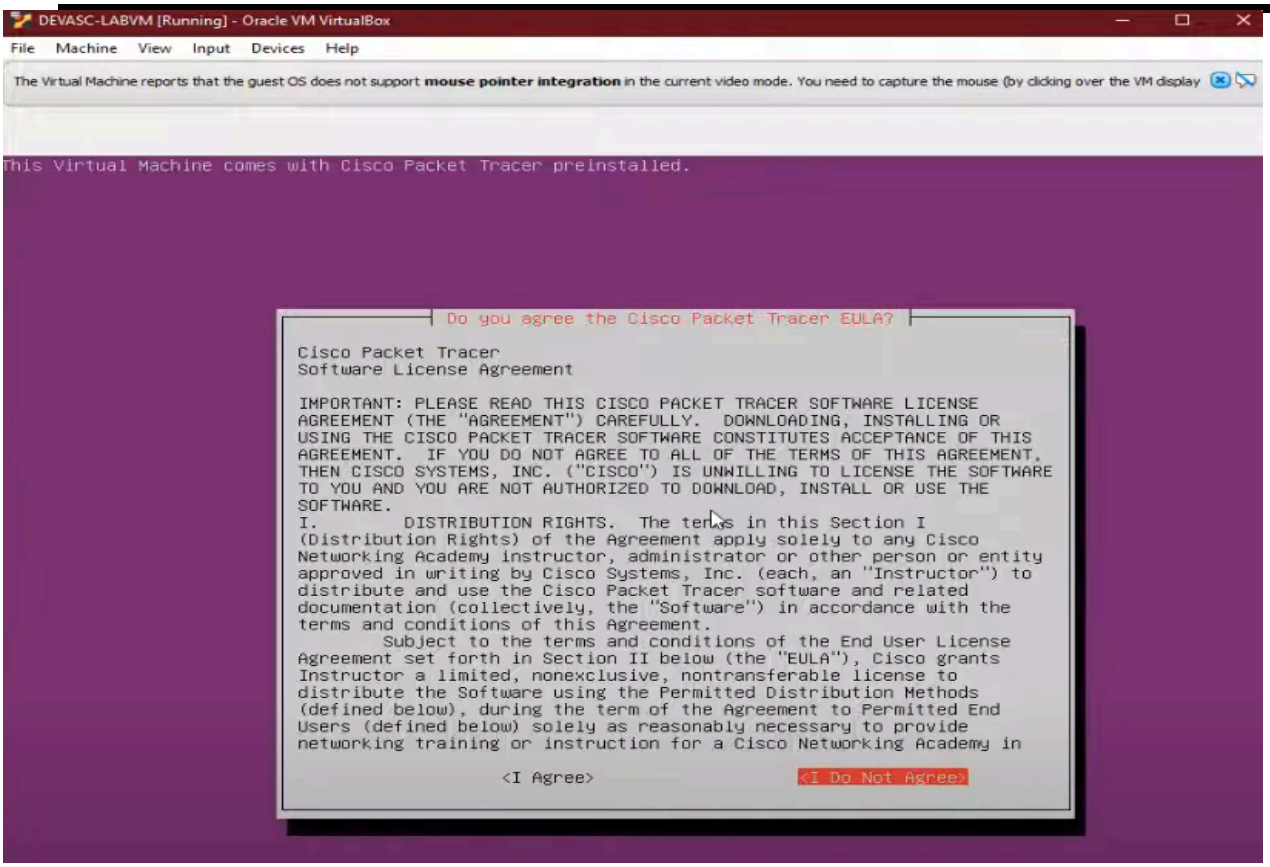
Part 2: Explore the DEVASC VM GUI

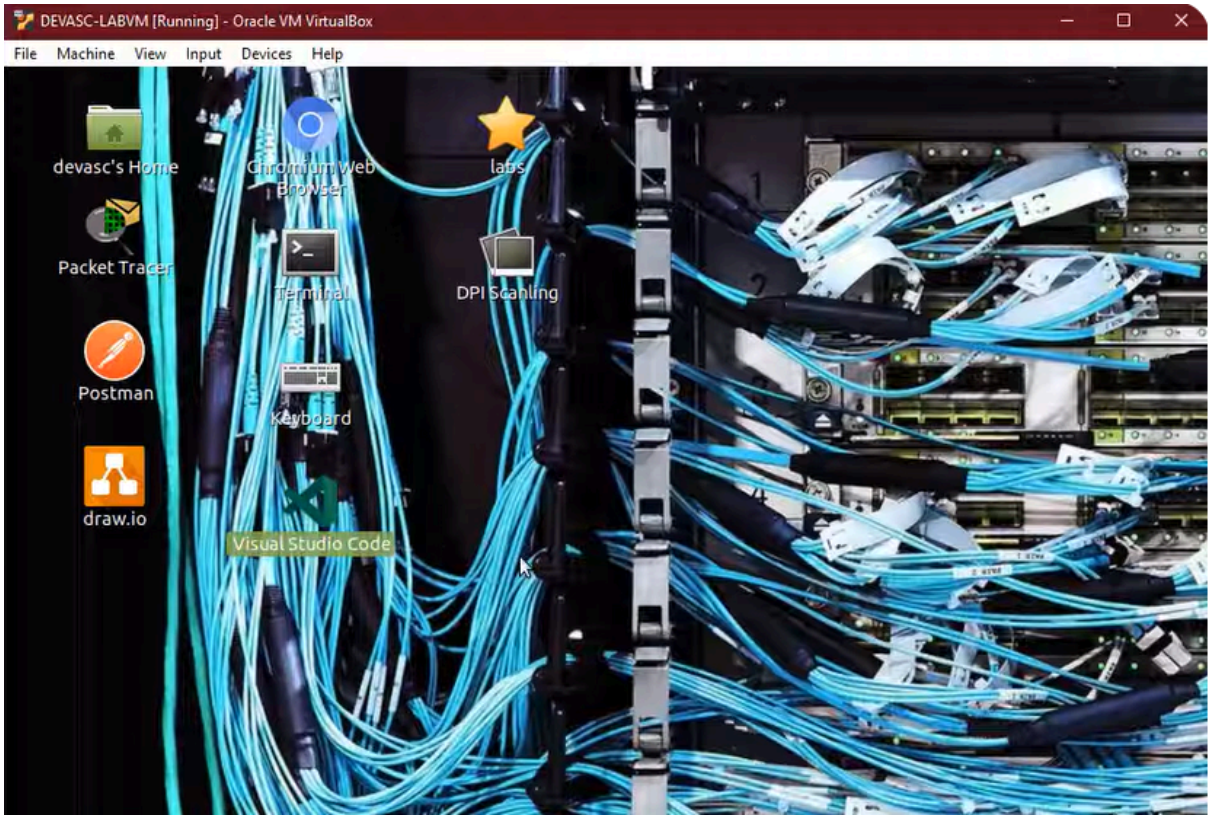
1. Start the DEVASC VM. Allow a few minutes for the Ubuntu image to boot.
2. Accept the Cisco Packet Tracer End User License Agreement (EULA) when prompted.
3. Explore the VM by:
    - Opening the terminal, VS Code, Packet Tracer, Chromium Browser, and Postman.
    - Clicking the **Menu** button and navigating through the **Places**, **System**, and **All Applications** menus.

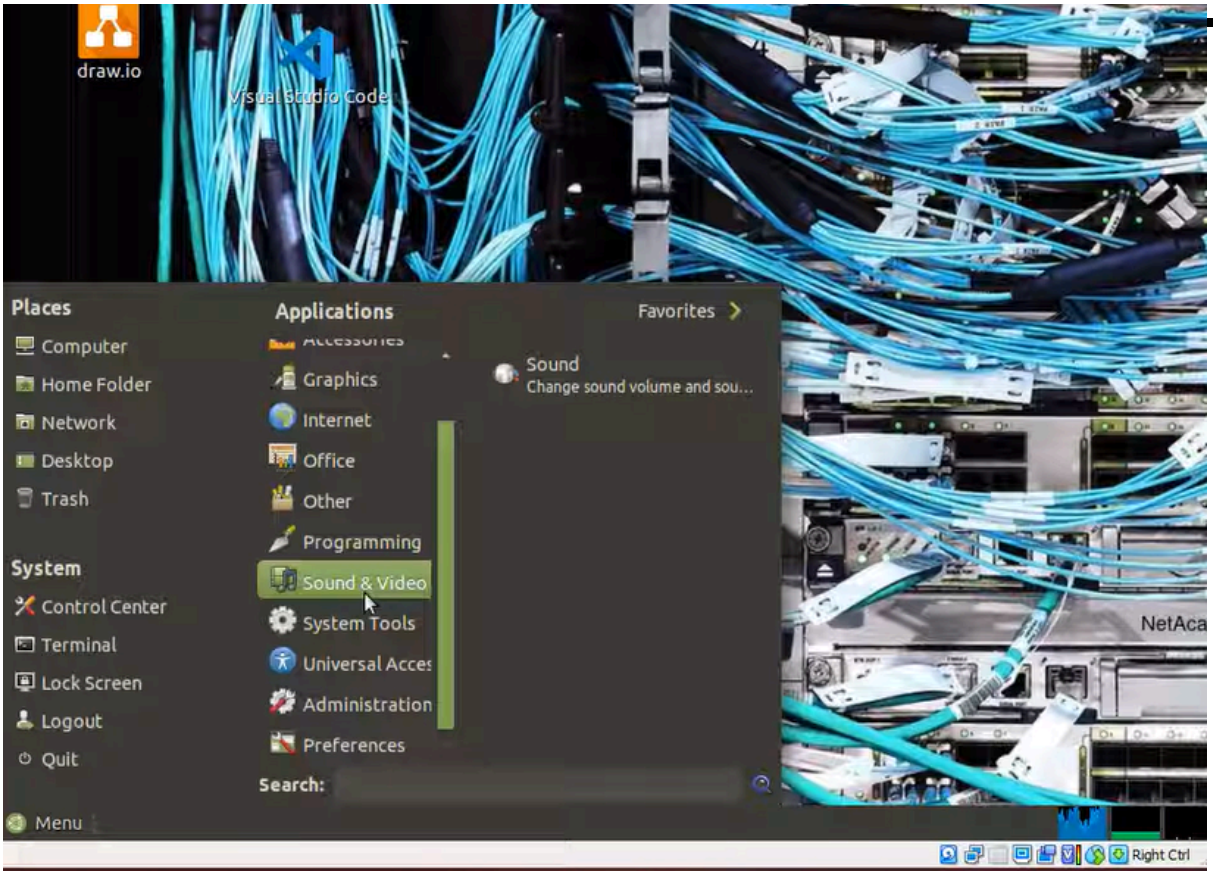
4. Remember that the VM is independent of your computer. Any changes can be reverted by re-importing the VM.






DEVASC-LABVM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
The Virtual Machine reports that the guest OS does not support mouse pointer integration in the current video mode. You need to capture the mouse (by clicking over the VM display
This Virtual Machine comes with Cisco Packet Tracer preinstalled.
Do you agree the Cisco Packet Tracer EULA?
Cisco Packet Tracer
Software License Agreement
IMPORTANT: PLEASE READ THIS CISCO PACKET TRACER SOFTWARE LICENSE AGREEMENT (THE "AGREEMENT") CAREFULLY. DOWNLOADING, INSTALLING OR USING THE CISCO PACKET TRACER SOFTWARE CONSTITUTES ACCEPTANCE OF THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, THEN CISCO SYSTEMS, INC. ("CISCO") IS UNWILLING TO LICENSE THE SOFTWARE TO YOU AND YOU ARE NOT AUTHORIZED TO DOWNLOAD, INSTALL OR USE THE SOFTWARE.
I. DISTRIBUTION RIGHTS. The terms in this Section I (Distribution Rights) of the Agreement apply solely to any Cisco Networking Academy instructor, administrator or other person or entity approved in writing by Cisco Systems, Inc. (each, an "Instructor") to distribute and use the Cisco Packet Tracer software and related documentation (collectively, the "Software") in accordance with the terms and conditions of this Agreement.
Subject to the terms and conditions of the End User License Agreement set forth in Section II below (the "EULA"), Cisco grants Instructor a limited, nonexclusive, nontransferable license to distribute the Software using the Permitted Distribution Methods (defined below), during the term of the Agreement to Permitted End Users (defined below) solely as reasonably necessary to provide networking training or instruction for a Cisco Networking Academy in
<I Agree>
<I Do Not Agree>

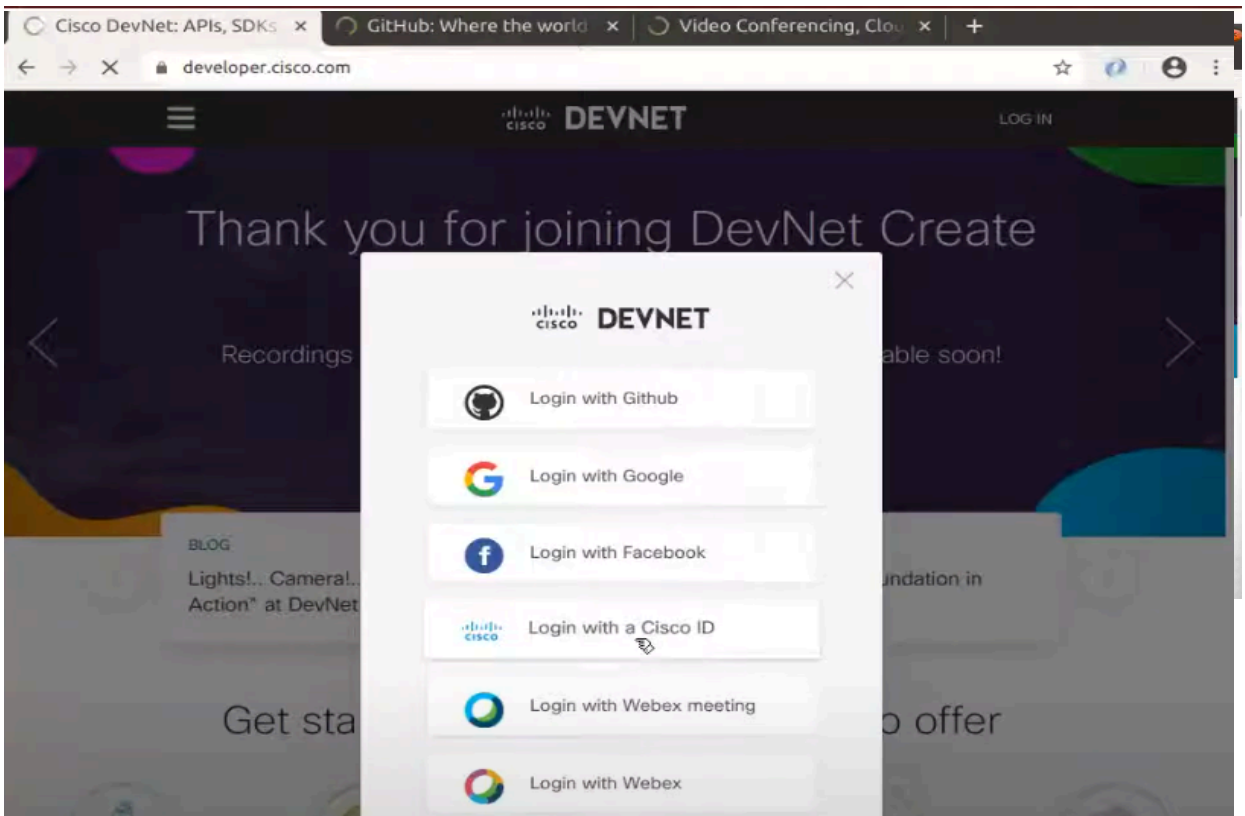

DEVASC-LABVM [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
devasc's Home
Chromium Web Browser
labs
Packet Tracer
Terminal
DPI Scanling
Postman
Keyboard
draw.io
Visual Studio Code



Part 3: Create Lab Environment Accounts

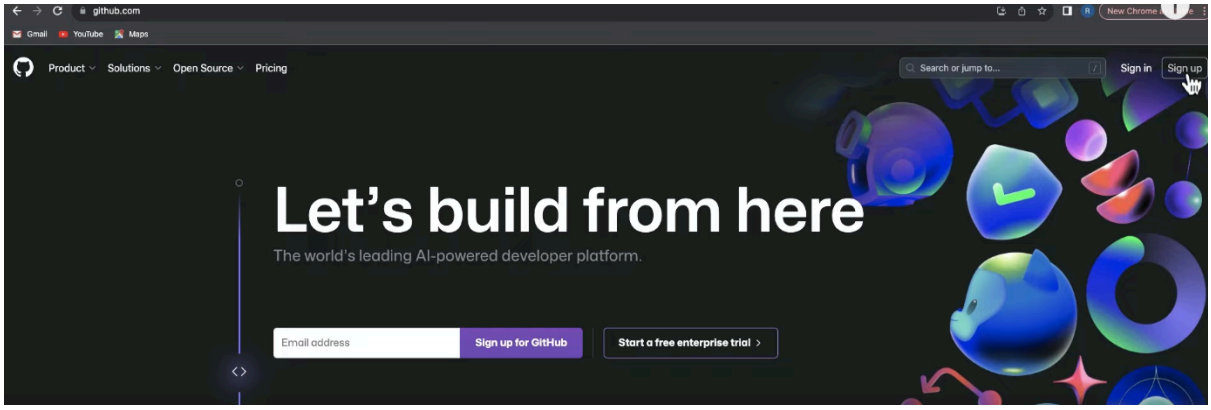
Create a DevNet Account

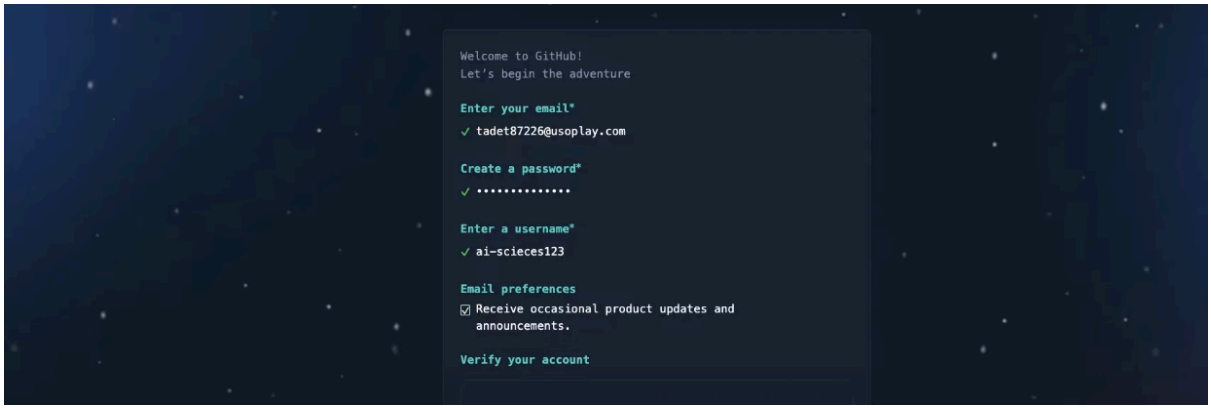
1. Open the Chromium browser in the DEVASC VM and navigate to developer.cisco.com.
2. Click **SIGN UP FREE** and complete the account creation process.

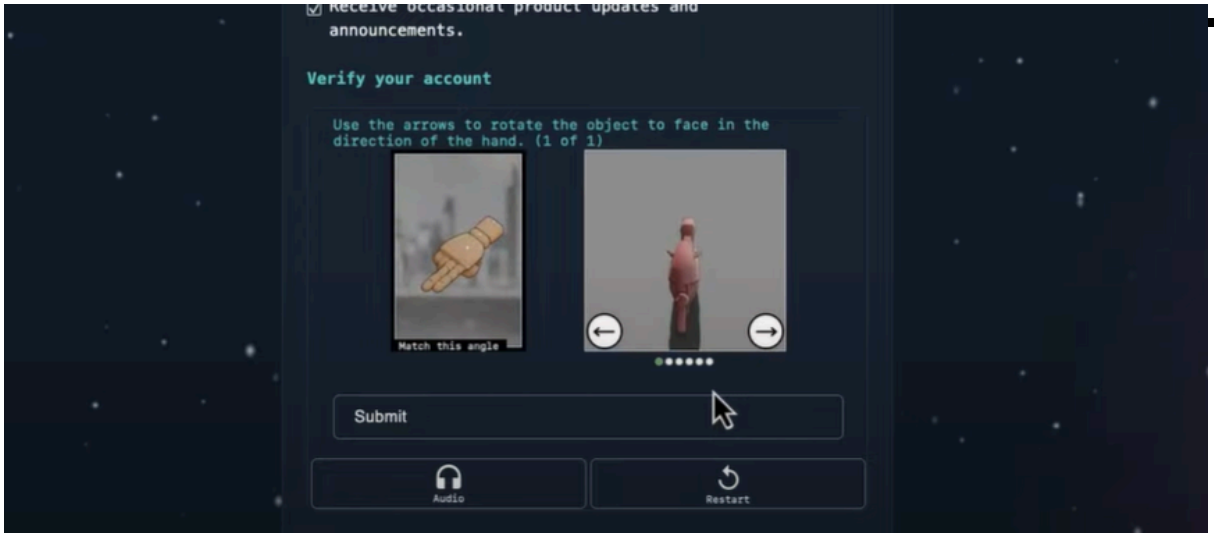


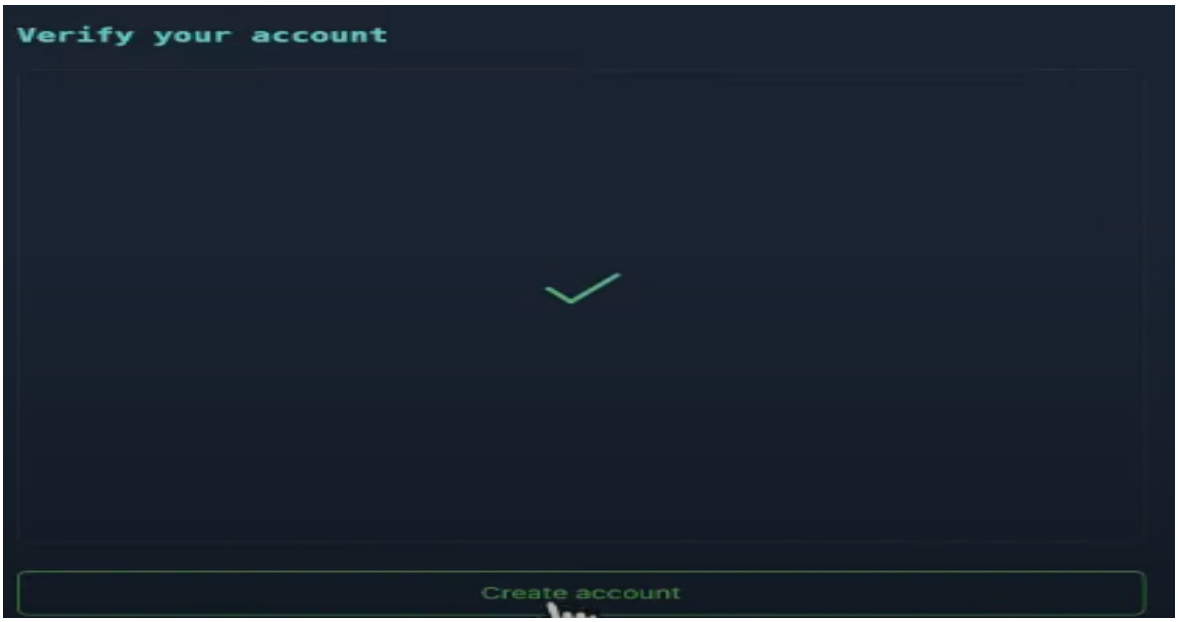
Create a GitHub Account

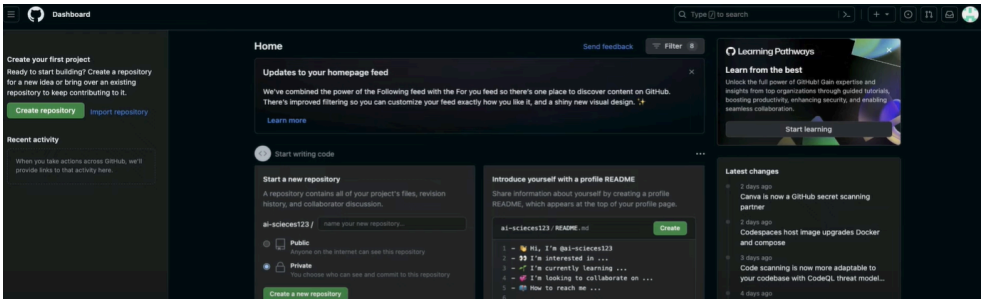
1. Navigate to GitHub.
2. Enter the required details (username, email, password) and click **Sign up for GitHub**.
3. Follow the verification and setup steps.





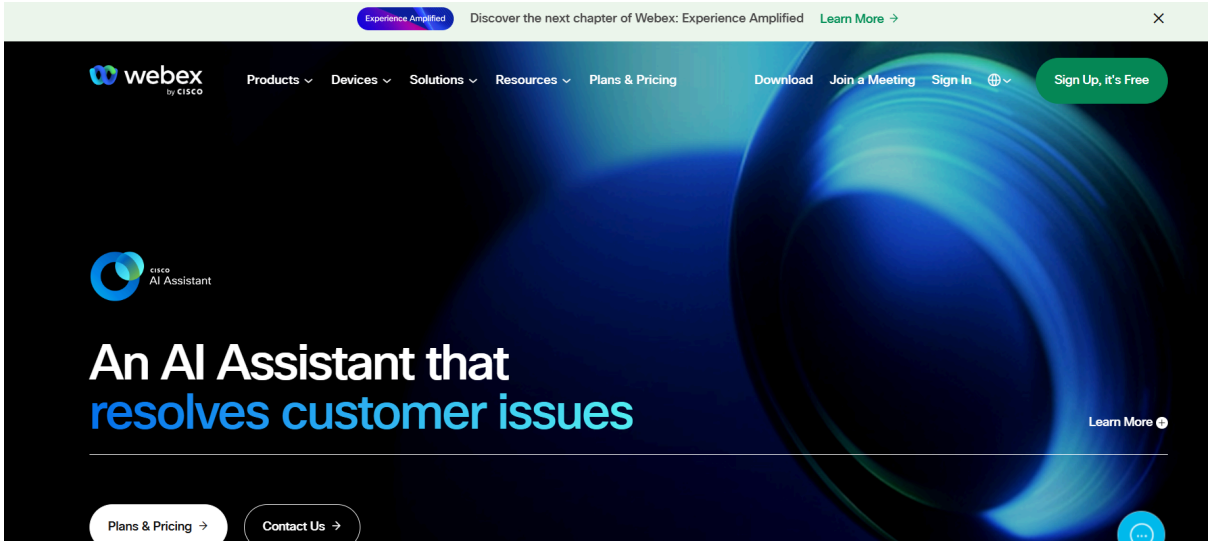






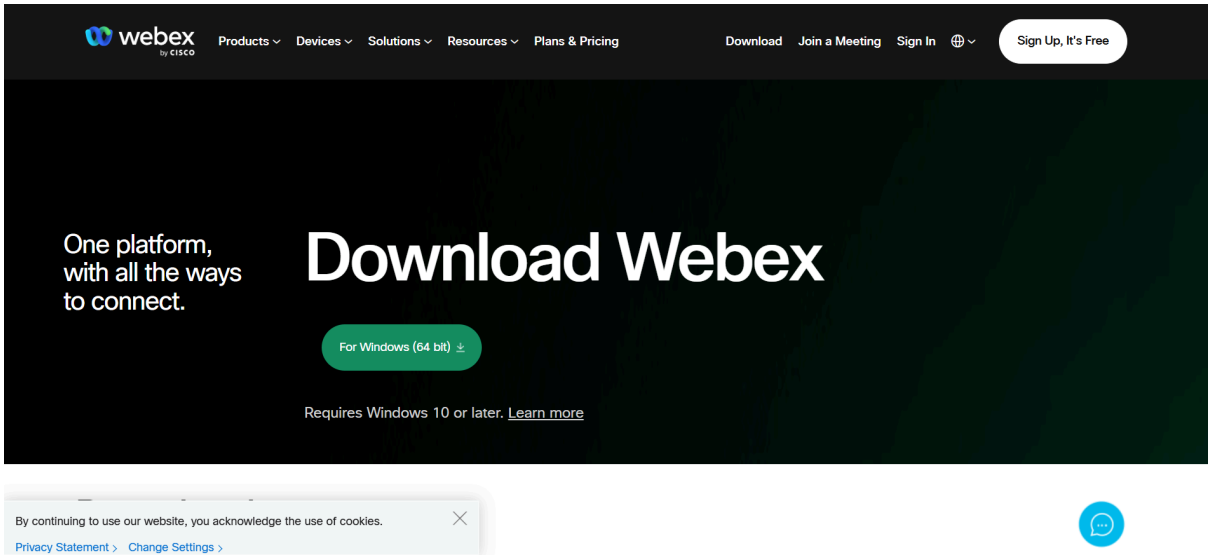
Create a Webex Account

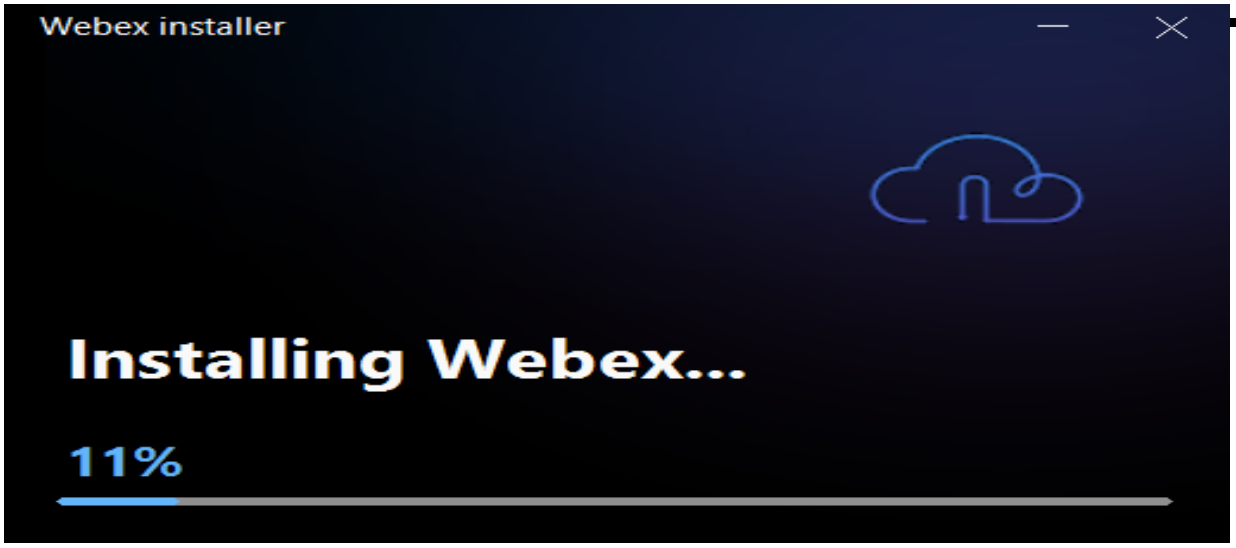
1. Navigate to Webex.
2. Click **Start for Free** and enter your email to sign up.
3. Complete the registration process.

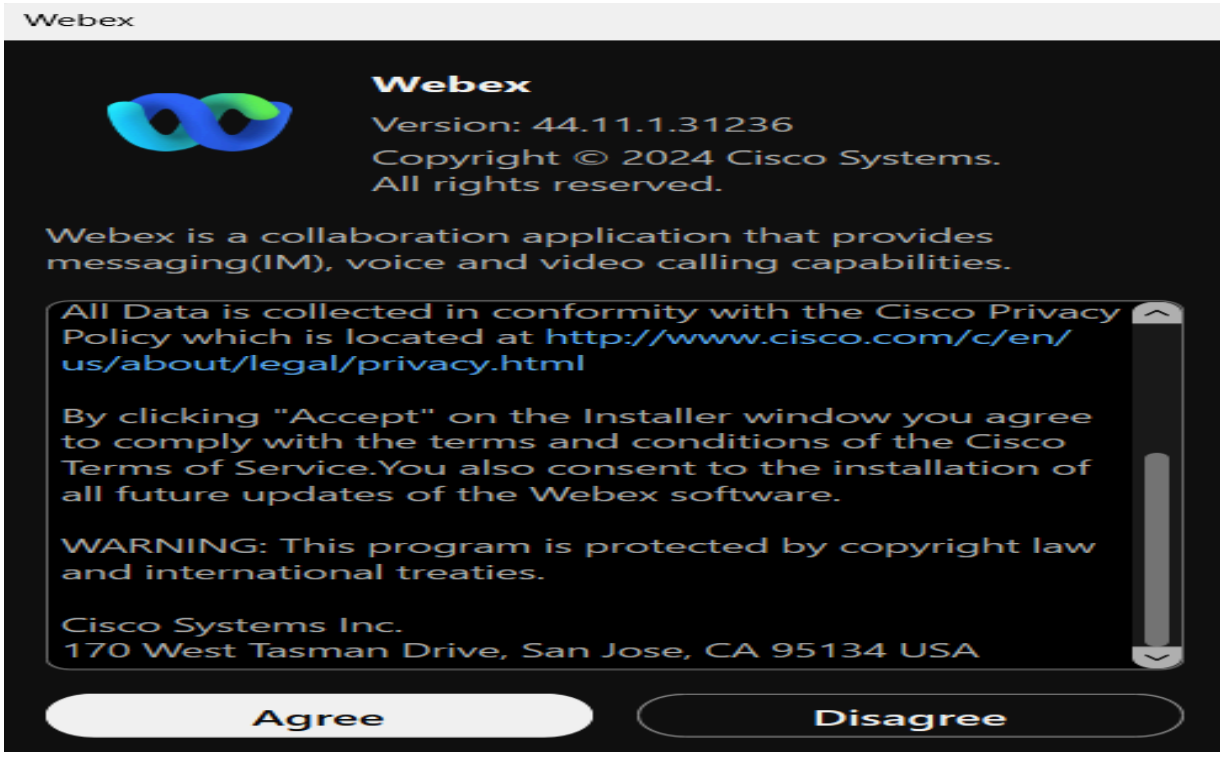


Part 4: Install Webex Teams on Your Device

1. Go to Webex Downloads.
2. Download the installation file for your device's operating system (Windows, macOS, or mobile).
3. Run the installer and follow the on-screen instructions.
4. Launch Webex Teams and join or create a team for collaboration.



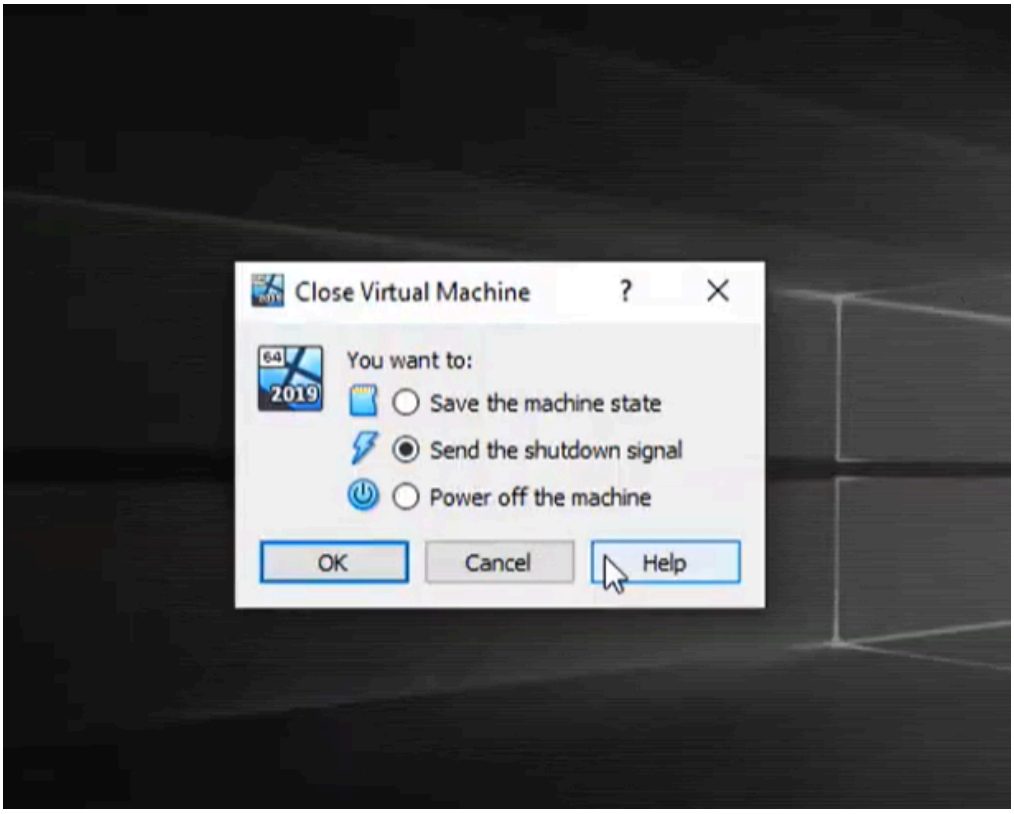

Webex installer
Installing Webex...
11%


Webex
Webex
Version: 44.11.1.31236
Copyright © 2024 Cisco Systems.
All rights reserved.
Webex is a collaboration application that provides messaging(IM), voice and video calling capabilities.
All Data is collected in conformity with the Cisco Privacy Policy which is located at http://www.cisco.com/c/en/us/about/legal/privacy.html
By clicking "Accept" on the Installer window you agree to comply with the terms and conditions of the Cisco Terms of Service.You also consent to the installation of all future updates of the Webex software.
WARNING: This program is protected by copyright law and international treaties.
Cisco Systems Inc.
170 West Tasman Drive, San Jose, CA 95134 USA
Agree
Disagree

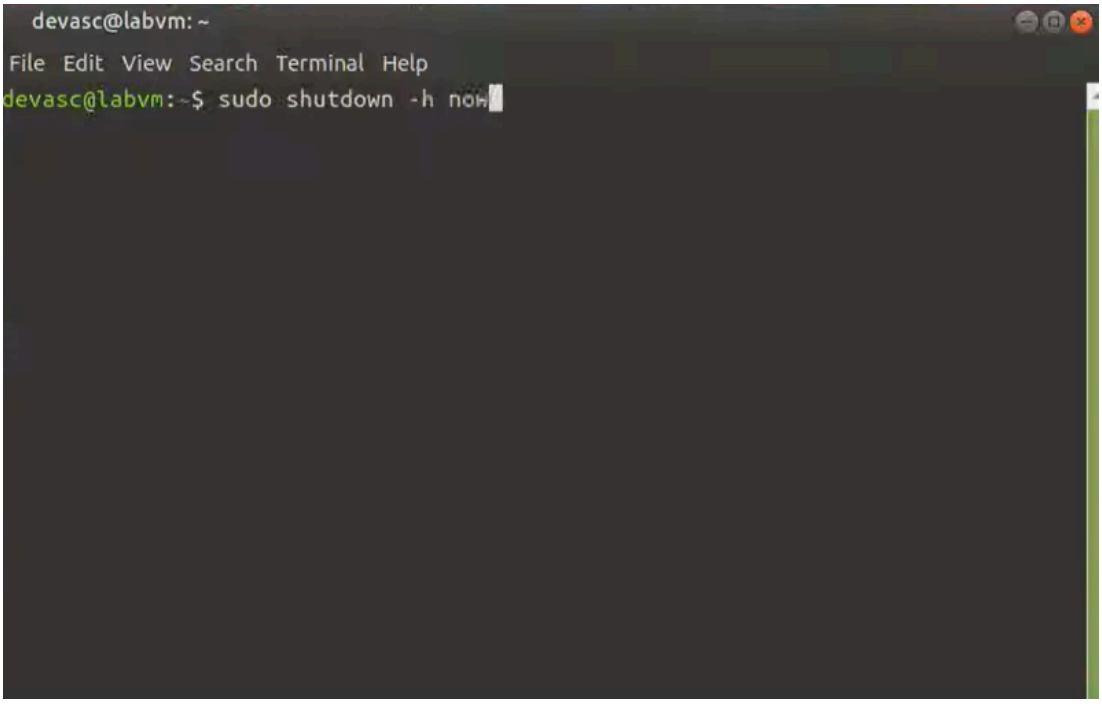

Webex
Sign in

Shutting Down and Managing the VM

- **Shut Down the VM:**
    - From the VirtualBox menu, select **Close > Save the Machine State** or **Send Shutdown Signal**.



    - Alternatively, in the VM, open the terminal and run: **sudo shutdown -h now.**



- Use the menu options or run: **sudo reboot**.

Lab Session 10:
Build a CI/CD Pipeline Using Jenkins

Objectives

Part 1: Launch the DEVASC VM

Part 2: Commit the Sample App to Git

Part 3: Modify the Sample App and Push

Changes to Git Part 4: Download and Run the

Jenkins Docker Image Part 5: Configure Jenkins

Part 6: Use Jenkins to Run a Build of

Your App Part 7: Use Jenkins to Test a

Build

Part 8: Create a Pipeline in Jenkins

Background / Scenario

In this lab, you will commit the Sample App code to a GitHub repository, modify the code locally, and then commit your changes. You will then install a Docker container that includes the latest version of Jenkins. You will configure Jenkins and then use Jenkins to download and run your Sample App program. Next, you will create a testing job inside Jenkins that will verify your Sample App program successfully runs each time you build it. Finally, you will integrate your Sample App and testing job into a Continuous Integration/Continuous Development pipeline that will verify your Sample App is ready to be deployed each time you change the code.

Required Resources

- 1 PC with operating system of your choice
- Virtual Box or VMWare
- DEVASC Virtual Machine

Instructions

Part 1: Launch the DEVASC VM

If you have not already completed the **Lab - Install the Virtual Machine Lab Environment**, do so now. If you have already completed that lab, launch the DEVASC VM now.

Part 2: Commit the Sample App to Git

In this part, you will create a GitHub repository to commit the sample-app files you created in a previous lab. You created a GitHub account in a previous lab. If you have not done so yet, visit github.com now and create an account.

Step 1: Login to GitHub and create a new repository.

a. Login at https://github.com/ with your credentials.

b. Select the "**New repository**" button or click on the "**+**" icon in the upper right corner and select "**New repository**".

c. Create a repository using the following information:

Repository name: **sample-app**

Description: **Explore CI/CD with GitHub and Jenkins**

Public/Private: **Private**

d. Select: **Create repository**

Step 2: Configure your Git credentials locally in the VM.

Open a terminal window **with VS Code** in the DEVASC VM. Use your name in place of "Sample User" for the name in quotes " ". Use @example.com for your email address.

devasc@labvm:~$ **git config --global user.name "Sample User"**

devasc@labvm:~$ **git config --global user.email sample@example.com**

Step 3: Initialize a directory as the Git repository.

You will use the sample-app files you created in a previous lab. However, those files are also stored for your convenience in the **/labs/devnet-src/jenkins/sample-app** directory. Navigate to the **jenkins/sample-app** directory and initialize it as a Git repository.

devasc@labvm:~$ **cd labs/devnet-src/jenkins/sample-app/**

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **git init**

Initialized empty Git repository in /home/devasc/labs/devnet-src/jenkins/sample-app/.git/

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$

Step 4: Point Git repository to GitHub repository.

Use the **git remote add** command to add a Git URL with a remote alias of "origin" and point to the newly created repository on GitHub. Using the URL of the Git repository you created in Step 1, you should only need to replace the **github-username** in the following command with your GitHub username.

Note: Your GitHub username is case-sensitive.

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **git remote add origin https://github.com/github-username/sample-app.git**

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$

Step 5: Stage, commit, and push the sample-app files to the GitHub repository.

a. Use the **git add** command to stage the files in the **jenkins/sample-app** directory. Use the asterisk (*) argument to stage all files in the current directory.

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **git add ***

b. Use the git status command to see the files and directories that are staged and ready to be committed to your GitHub repository.

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **git status**

On branch

master No

commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file: sample-app.sh
        new file: sample_app.py
        new file: static/style.css

```
        new file:   templates/index.html

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$
```

c. Use the **git commit** command to commit the staged files and start tracking changes. Add a message of your choice or use the one provided here.

```
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ git commit -m "Committing sample-app files."
[master 4030ab6] Committing sample-app files
 4 files changed, 46 insertions(+)
 create mode 100644 sample-app.sh
 create mode 100644 sample_app.py
 create mode 100644 static/style.css
 create mode 100644 templates/index.html
```

d. Use the **git push** command to push your local sample-app files to your GitHub repository.

```
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ git push origin master
Username for 'https://github.com': username
Password for 'https://AllJohns@github.com': password
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 2
threads Compressing objects: 100%
(5/5), done.
Writing objects: 100% (8/8), 1.05 KiB | 1.05 MiB/s, done. Total
8 (delta 0), reused 0 (delta 0)
To https://github.com/AllJohns/sample-app.git
   d0ee14a..4030ab6 master -> master
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$
```

Note: If, instead of a request for your username, you get a message from VS Code with the message, **The extension 'Git' wants to sign in using GitHub**, then you misconfigured either your GitHub credentials in Step 2 and/or the GitHub URL in Step 4. The URL must have the correct case-sensitive username and the name of the repository that you created in Step 1. To reverse your previous **git add** command, use the command **git remote rm origin**. Then return to Step 2 making sure to enter the correct credentials and, in Step 4, entering the correct URL.

Note: If, after entering your username and password, you get a fatal error stating repository is not found, you most likely submitted an incorrect URL. You will need to reverse your **git add** command with the **git remote rm origin** command.

Part 3: Modify the Sample App and Push Changes to Git

In Part 4, you will install a Jenkins Docker image that will use port 8080. Recall that your sample-app files are also specifying port 8080. The Flask server and Jenkins server cannot both use 8080 at the same time.

In this part, you will change the port number used by the sample-app files, run the sample-app again to verify it works on the new port, and then push your changes to your GitHub repository.

Step 1: Open the sample-app files.

Make sure you are still in the **~/labs/devnet-src/jenkins/sample-app** directory as these are the files that are associated with your GitHub repository. Open both **sample_app.py** and **sample-app.sh** for editing.

Step 2: Edit the sample-app files.

a. In sample_app.py, change the one instance of port 8080 to 5050 as shown below.

```
from flask import Flask
from flask import request
from flask import render_template
sample = Flask(_name_)
@sample.route("/")
def main():
    return render_template("index.html")

if_name_== "_main_":
    sample.run(host="0.0.0.0", port=5050)
```

b. In sample-app.sh, change the three instances of port 8080 to 5050 as shown below.

```
#!/bin/bash

mkdir tempdir
mkdir
tempdir/templates
mkdir tempdir/static

cp sample_app.py tempdir/.
cp -r templates/* tempdir/templates/.
cp -r static/* tempdir/static/.

echo "FROM python" >> tempdir/Dockerfile
echo "RUN pip install flask" >> tempdir/Dockerfile
echo "COPY ./static /home/myapp/static/" >> tempdir/Dockerfile
echo "COPY ./templates /home/myapp/templates/" >>
tempdir/Dockerfile echo "COPY sample_app.py /home/myapp/" >>
tempdir/Dockerfile
echo "EXPOSE 5050" >> tempdir/Dockerfile
echo "CMD python3 /home/myapp/sample_app.py" >>

tempdir/Dockerfile cd tempdir

docker build -t sampleapp .

docker run -t -d -p 5050:5050 --name samplerunning
sampleapp docker ps -a
```

Step 3: Build and verify the sample-app.

a. Enter the **bash** command to build your app using the new port 5050.

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **bash ./sample-app.sh**
Sending build context to Docker daemon
6.144kB Step 1/7 : FROM python
 ---> 4f7cd4269fa9
Step 2/7 : RUN pip install flask

---> Using cache

---> 57a74c0dff93

Step 3/7 : COPY ./static /home/myapp/static/

---> Using cache

---> e70310436097

Step 4/7 : COPY ./templates /home/myapp/templates/

---> Using cache

---> e41ed6d0f933

Step 5/7 : COPY sample_app.py /home/myapp/

---> 0a8d152f78fd

Step 6/7 : EXPOSE

5050

---> Running in d68f6bfbcffb

Removing intermediate container d68f6bfbcffb

---> 04fa04a1c3d7

Step 7/7 : CMD python3 /home/myapp/sample_app.py

---> Running in ed48fdbc031b

Removing intermediate container ed48fdbc031b

---> ec9f34fa98fe

Successfully built ec9f34fa98fe

Successfully tagged sampleapp:latest

d957a4094c1781ccd7d86977908f5419a32c05a2a1591943bb44eeb8271c02dc

CONTAINER ID IMAGE COMMAND
CREATED STATUS PORTS NAMES

d957a4094c17 sampleapp "/bin/sh -c 'python …" 1 second
ago Up Less than a second 0.0.0.0:5050->5050/tcp
samplerunning

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$

b. Open a browser tab and navigate to localhost:5050. You should see the message **You are calling me from 172.17.0.1**.

c. Shut down the server when you have verified that it is operating on port 5050. Return to the terminal window where the server is running and press CTRL+C to stop the server.

Step 4: Push your changes to GitHub.

a. Now you are ready to push your changes to your GitHub repository. Enter the following commands.

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **git add ***

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **git status**

On branch master

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified: sample-app.sh
        modified: sample_app.py
        new file: tempdir/Dockerfile
        new file: tempdir/sample_app.py
        new file: tempdir/static/style.css
        new file: tempdir/templates/index.html

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **git commit -m "Changed port from 8080 to 5050."**
[master 98d9b2f] Changed port from 8080 to 5050.

6 files changed, 33 insertions(+), 3 deletions(-)
create mode 100644 tempdir/Dockerfile
create mode 100644 tempdir/sample_app.py
create mode 100644 tempdir/static/style.css
create mode 100644
tempdir/templates/index.html
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **git push origin master**
Username for 'https://github.com': **username**
Password for 'https://AllJohns@github.com': **password**
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 2
threads Compressing objects: 100%
(6/6), done.
Writing objects: 100% (6/6), 748 bytes | 748.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/AllJohns/sample-app.git
a6b6b83..98d9b2f master -> master
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$

b. You can verify that your GitHub repository is updated by visiting **https://github.com/github- user/sample-app**. You should see your new message (Changed port from 8080 to 5050.) and that the latest commit timestamp has been updated.

Part 4: Download and Run the Jenkins Docker Image

In this part, you will download the Jenkins Docker image. You will then start an instance of the image and verify that the Jenkins server is running.

Step 1: Download the Jenkins Docker image.

The Jenkins Docker image is stored here: https://hub.docker.com/r/jenkins/jenkins. At the time of the writing of this lab, that site specifies that you use the **docker pull jenkins/jenkins** command to download the latest Jenkins container. You should get output similar to the following:

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **docker pull jenkins/jenkins:lts**
lts: Pulling from jenkins/jenkins
3192219afd04: Pulling fs layer
17c160265e75: Pulling fs layer
cc4fe40d0e61: Pulling fs layer

9d647f502a07: Pulling fs layer
d108b8c498aa: Pulling fs layer
1bfe918b8aa5: Pull complete
dafa1a7c0751: Pull complete
650a236d0150: Pull complete
cba44e30780e: Pull complete
52e2f7d12a4d: Pull complete
d642af5920ea: Pull complete
e65796f9919e: Pull complete
9138dabbc5cc: Pull complete
f6289c08656c: Pull complete
73d6b450f95c: Pull complete

a8f96fbec6a5: Pull complete 9b49ca1b4e3f: Pull complete d9c8f6503715: Pull complete 20fe25b7b8af: Pull complete
Digest: sha256:717dcbe5920753187a20ba43058ffd3d87647fa903d98cde64dda4f4c82c5c48 Status: Downloaded newer image for jenkins/jenkins:lts
docker.io/jenkins/jenkins:lts
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$

Step 2: Start the Jenkins Docker container.

Enter the following command on **one line**. You may need to copy it to a text editor if you are viewing a PDF version of this lab to avoid line breaks. This command will start the Jenkins Docker container and then allow Docker commands to be executed inside your Jenkins server.

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **docker run --rm -u root -p 8080:8080 -v jenkins-data:/var/jenkins_home -v $(which docker):/usr/bin/docker -v /var/run/docker.sock:/var/run/docker.sock -v "$HOME":/home --name jenkins_server jenkins/jenkins:lts**

The options used in this **docker run** command are as follows:

- **--rm** - This option automatically removes the Docker container when you stop running it.
- **-u** - This option specifies the user. You want this Docker container to run as root so that all Docker commands entered inside the Jenkins server are allowed.
- **-p** - This option specifies the port the Jenkins server will run on locally.
- **-v** - These options bind mount volumes needed for Jenkins and Docker. The first **-v** specifies where Jenkins data will be stored. The second **-v** specifies where to get Docker so that you can run Docker inside the Docker container that is running the Jenkins server. The third **-v** specifies the PATH variable for the home directory.

Step 3: Verify the Jenkins server is running.

The Jenkins server should now be running. Copy the admin password that displays in the output, as shown in the following.

Do not enter any commands in this server window. If you accidentally stop the Jenkins server, you will need to re-enter the **docker run** command from Step 2 above. After the initial install, the admin password is displayed as shown below.

<output omitted>
****
****

****

Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:

77dc402e31324c1b917f230af7bfebf2 <--Your password will be different

This may also be found at: /var/jenkins_home/secrets/initialAdminPassword

****

****

```
*************************************************************
<output omitted>
2020-05-12 16:34:29.608+0000 [id=19]  INFO    hudson.WebAppMain$3#run:
Jenkins is fully up and running
```

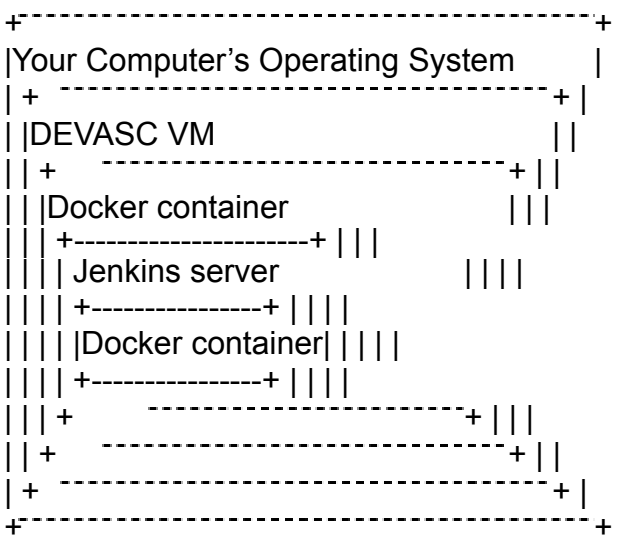
Note: If you lose the password, or it does not display as shown above, or you need to restart the Jenkins sever, you can always retrieve the password by accessing the command line of Jenkins Docker container. Create a second terminal window in VS Code and enter the following commands so that you do not stop the Jenkins server.:

```
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ docker exec -it
jenkins_server /bin/bash
root@19d2a847a54e:/# cat /var/jenkins_home/secrets/initialAdminPassword
77dc402e31324c1b917f230af7b
febf2 root@19d2a847a54e:/#
exit exit
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$
```

Note: Your container ID (19d2a847a54e highlighted above) and password will be different.

Step 4: Investigate the levels of abstraction currently running on your computer.

The following ASCII diagram shows the levels of abstraction in this Docker-inside-Docker (dind) implementation. This level of complexity is not unusual in today's networks and cloud infrastructures.



Part 5: Configure Jenkins

In this Part, you will complete the initial configuration of the Jenkins server.

Step 1: Open a web browser tab.

Navigate to http://localhost:8080/ and login in with your copied admin password.

Step 2: Install the recommended Jenkins plugins.

Click **Install suggested plugins** and wait for Jenkins to download and install the plugins. In the terminal window, you will see log messages as the installation proceeds. Be sure that you do not close this terminal window. You can open another terminal window for access to the command line.

Step 3: Skip creating a new admin user.

After the installation finishes, you are presented with the **Create First Admin User** window. For now, click
Skip and continue as admin at the bottom.

Step 4: Skip creating an instance configuration.

In the **Instance Configuration** window, do not change anything. Click **Save and Finish** at the bottom.

Step 5: Start using Jenkins.

In the next window, click **Start using Jenkins**. You should now be on the main dashboard with a **Welcome to Jenkins!** message.

Part 6: Use Jenkins to Run a Build of Your App

The fundamental unit of Jenkins is the job (also known as a project). You can create jobs that do a variety of tasks including the following:

- Retrieve code from a source code management repository such as GitHub.
- Build an application using a script or build tool.
- Package an application and run it on a server

In this part, you will create a simple Jenkins job that retrieves the latest version of your sample-app from GitHub and runs the build script. In Jenkins, you can then test your app (Part 7) and add it to a development pipeline (Part 8).

Step 1: Create a new job.

a. Click the **Create a job** link directly below the **Welcome to Jenkins!** message. Alternatively, you can click
New Item in the menu on the left.

b. In the **Enter an item name** field, fill in the name **BuildAppJob**.

c. Click **Freestyle project** as the job type. In the description, the SCM abbreviation stands for software configuration management, which is a classification of software that is responsible for tracking and controlling changes in software.

d. Scroll to the bottom and click **OK**.

Step 2: Configure the Jenkins BuildAppJob.

You are now in the configuration window where you can enter details about your job. The tabs across the top are just shortcuts to the sections below. Click through the tabs to explore the options you can configure. For this simple job, you only need to add a few configuration details.

a. Click the **General tab**, add a description for your job. For example, "**My first Jenkins job.**"

b. Click the **Source Code Management** tab and choose the **Git** radio button. In the Repository URL field, add your GitHub repository link for the sample-app taking care to enter your case-sensitive username. Be sure to add the .git extension at the end of your URL. For example:

https://github.com/github-username/sample-app.git

c. For **Credentials**, click the **Add** button and choose **Jenkins**.

d. In the **Add Credentials** dialog box, fill in your GitHub username and password, and then click **Add**.

   Note: You will receive an error message that the connection has failed. This is because you have not selected the credentials yet.

e. In the dropdown for **Credentials** where it currently says **None**, choose the credentials you just configured.

f. After you have added the correct URL and credentials, Jenkins tests access to the repository. You should have no error messages. If you do, verify your URL and credentials. You will need to **Add** them again as there is no way at this point to delete the ones you previously entered.

g. At the top of the **BuildAppJob** configuration window, click the **Build** tab.

h. For the **Add build step** dropdown, choose **Execute shell**.

i. In the **Command** field, enter the command you use to run the build for sample-app.sh script.

   bash ./sample-app.sh

j. Click the **Save** button. You are returned to the Jenkins dashboard with the **BuildAppJob** selected.

Step 3: Have Jenkins build the app.

On the left side, click **Build Now** to start the job. Jenkins will download your Git repository and execute the build command **bash ./sample-app.sh**. Your build should succeed because you have not changed anything in the code since Part 3 when you modified the code.

Step 4: Access the build details.

On the left, in the **Build History** section, click your build number which should be the **#1** unless you have built the app multiple times.

Step 5: View the console output.

On the left, click **Console Output**. You should see output similar to the following. Notice the success messages at the bottom as well as the output from the **docker ps -a** command. Two docker containers are running: one for your sample-app running on local port 5050 and one for Jenkins on local port 8080.

```
Started by user
admin Running as
SYSTEM
Building in workspace
/var/jenkins_home/workspace/BuildAppJob using credential
0cf684ea-48a1-4e8b-ba24-b2fa1c5aa3df Cloning the remote
Git repository
Cloning repository https://github.com/github-user/sample-app
 > git init /var/jenkins_home/workspace/BuildAppJob # timeout=10
Fetching upstream changes from
https://github.com/github-user/sample-app
 > git --version # timeout=10
using GIT_ASKPASS to set credentials
 > git fetch --tags --progress -- https://github.com/github-user/sample-app
+refs/heads/*:refs/remotes/origin/* # timeout=10
 > git config remote.origin.url https://github.com/github-user/sample-app # timeout=10
 > git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* #
timeout=10
```

> git config remote.origin.url https://github.com/github-user/sample-app # timeout=10

Fetching upstream changes from https://github.com/github-user/sample-app

using GIT_ASKPASS to set credentials

> git fetch --tags --progress -- https://github.com/github-user/sample-app +refs/heads/*:refs/remotes/origin/* # timeout=10

> git rev-parse refs/remotes/origin/master^{commit} # timeout=10

> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10

Checking out Revision 230ca953ce83b5d6bdb8f99f11829e3a963028bf (refs/remotes/origin/master)

> git config core.sparsecheckout # timeout=10

> git checkout -f 230ca953ce83b5d6bdb8f99f11829e3a963028bf # timeout=10 Commit message: "Changed port numbers from 8080 to 5050"

> git rev-list --no-walk 230ca953ce83b5d6bdb8f99f11829e3a963028bf # timeout=10 [BuildAppJob] $ /bin/sh -xe /tmp/jenkins1084219378602319752.sh

```
+ bash ./sample-app.sh
Sending build context to Docker daemon 6.144kB

Step 1/7 : FROM python
 ---> 4f7cd4269fa9
Step 2/7 : RUN pip install flask
 ---> Using cache
 ---> 57a74c0dff93
Step 3/7 : COPY ./static /home/myapp/static/
 ---> Using cache
 ---> aee4eb712490
Step 4/7 : COPY ./templates /home/myapp/templates/
 ---> Using cache
 ---> 594cdc822490
Step 5/7 : COPY sample_app.py /home/myapp/
 ---> Using cache
 ---> a001df90cf0c
Step 6/7 : EXPOSE
5050
 ---> Using cache
 ---> eae896e0a98c
Step 7/7 : CMD python3 /home/myapp/sample_app.py
 ---> Using cache
 ---> 272c61fddb45
Successfully built
272c61fddb45

CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                                NAMES
9c8594e62079        sampleapp           "/bin/sh -c 'python …"   Less than a second
Successfully tagged sampleapp:latest
9c8594e62079c069baf9a88a75c13c8c55a3aeaddde6fd6ef54010953c2d3fbb


ago    Up Less than a second  0.0.0.0:5050->5050/tcp                samplerunning
e25f233f9363        jenkins/jenkins:lts       "/sbin/tini -- /usr/…"   29 minutes
ago
Up 29 minutes               0.0.0.0:8080->8080/tcp, 50000/tcp
jenkins_server Finished: SUCCESS
```

Step 6: Open another web browser tab and verify sample app is running.

Type in the local address, **localhost:5050**. You should see the content of your index.html displayed in light steel blue background color with **You are calling me from 172.17.0.1** displayed in as H1.

Part 7: Use Jenkins to Test a Build

In this part, you will create a second job that tests the build to ensure that it is working properly.

Note: You need to stop and remove the **samplerunning** docker container.

devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **docker stop samplerunning**
samplerunning
devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ **docker rm samplerunning**
samplerunning

Step 1: Start a new job for testing your sample-app.

a. Return to the Jenkins web browser tab and click the **Jenkins** link in the top left corner to return to the main dashboard.
b. Click the **New Item** link to create a new job.
c. In the Enter an item name field, fill in the name **TestAppJob**.
d. Click **Freestyle project** as the job type.
e. Scroll to the bottom and click **OK**.

Step 2: Configure the Jenkins TestAppJob.

a. Add a description for your job. For example, "My first Jenkins test."
b. Leave **Source Code Management** set to **None**.
c. Click the **Build Triggers** tab and check the box, **Build after other projects are built**. For **Projects to watch**, fill in the name **BuildAppJob**.

Step 3: Write the test script that should run after a stable build of the BuildAppJob.

a. Click the **Build** tab.
b. Click **Add build step** and choose **Execute shell**.
c. Enter the following script. The **if** command should be all on one line including the **; then**. This command will **grep** the output returned from the cURL command to see if **You are calling me from 172.17.0.1** is returned. If true, the script exits with a code of 0, which means that there are no errors in the **BuildAppJob** build. If false, the script exits with a code of 1 which means the **BuildAppJob** failed.

```
if curl http://172.17.0.1:5050/ | grep "You are calling me from 172.17.0.1"; then exit
    0
else
    exit 1
fi
```

d. Click **Save** and then the **Back to Dashboard** link on the left side.

Step 4: Have Jenkins run the BuildAppJob job again.

a. Refresh the web page with the refresh button for your browser.
b. You should now see your two jobs listed in a table. For the **BuildAppJob** job, click the build button on the far right (a clock with an arrow).

Step 5: Verify both jobs completed.

If all goes well, you should see the timestamp for the **Last Success** column update for both **BuildAppJob** and **TestAppJob**. This means your code for both jobs ran without error. But you can also verify this for yourself.

Note: If timestamps do not update, make sure enable auto refresh is turned on by clicking the link in the top right corner.

a. Click the Link for **TestAppJob**. Under **Permalinks**, click the link for your last build, and then click
Console Output. You should see output similar to the following:

Started by upstream project "BuildAppJob" build number 13

originally caused by:

Started by user

admin Running as

SYSTEM

```
Building in workspace
/var/jenkins_home/workspace/TestAppJob [TestAppJob] $
/bin/sh -xe /tmp/jenkins1658055689664198619.sh
+ grep You are calling me from 172.17.0.1
+ curl http://172.17.0.1:5050/
  %  Total    %  Received %Xferd  Average Speed          Time    Time     Time  Current
                                   Dload  Upload          Total   Spent    Left  Speed
  0      0    0     0    0     0      0      0 --:--:--  --:--:--  --:--:--     0
 10    177  100   177    0     0   2977      0 --:--:--  --:--:--  --:--:--  3540
100                                  2                                       0
    <h1>You are calling me from 172.17.0.1</h1>
+ exit 0 Finished:
SUCCESS
```

b. It is not necessary to verify your sample app is running because the **TestAppJob** already did this for you. However, you can open a browser tab for **172.17.0.1:5050** to see that it is indeed running.

Part 8: Create a Pipeline in Jenkins

Although you can currently run your two jobs by simply clicking the Build Now button for the **BuildAppJob**, software development projects are typically much more complex. These projects can benefit greatly from automating builds for continuous integration of code changes and continuously creating development builds that are ready to deploy. This is the essence of CI/CD. A pipeline can be automated to run based on a variety of triggers including periodically, based on a GitHub poll for changes, or from a script run remotely. However, in this part you will script a pipeline in Jenkins to run your two apps whenever you click the pipeline **Build Now** button.

Step 1: Create a Pipeline job.

a. Click the **Jenkins** link in the top left, and then **New Item**.

b. In the **Enter an item name** field, type **SamplePipeline**.

c. Select **Pipeline** as the job type.

d. Scroll to the bottom and click **OK**.

Step 2: Configure the SamplePipeline job.

a. Along the top, click the tabs and investigate each section of the configuration page. Notice that there are a number of different ways to trigger a build. For the **SamplePipeline** job, you will trigger it manually.

b. In the **Pipeline** section, add the following script.

```
node {
    stage('Preparation') {
```

```
        catchError(buildResult: 'SUCCESS') {
            sh 'docker stop
            samplerunning' sh 'docker rm
            samplerunning'
        }
    }
    stage('Build') {
        build 'BuildAppJob'
    }
    stage('Results') { build
        'TestAppJob'
    }
```

}

This script does the following:

- It creates a single node build as opposed to a distributed or multi node. Distributed or multi node configurations are for larger pipelines than the one you are building in this lab and are beyond the scope of this course.
- In the **Preparation** stage, the **SamplePipeline** will first make sure that any previous instances of the **BuildAppJob** docker container are stopped and removed. But if there is not yet a running container you will get an error. Therefore, you use the **catchError** function to catch any errors and return a "SUCCESS" value. This will ensure that pipeline continues on to the next stage.
- In the **Build** stage, the **SamplePipeline** will build your **BuildAppJob**.
- In the **Results** stage, the **SamplePipeline** will build your **TestAppJob**.

c. Click **Save** and you will be returned to the Jenkins dashboard for the **SamplePipeline** job.

Step 3: Run the SamplePipeline.

On the left, click **Build Now** to run the **SamplePipeline** job. If you coded your Pipeline script without error, then the **Stage View** should show three green boxes with number of seconds each stage took to build. If not, click Configure on the left to return to the **SamplePipeline** configuration and check your Pipeline script.

Step 4: Verify the SamplePipeline output.

Click the latest build link under **Permalinks**, and then click **Console Output**. You should see output similar to the following:

```
Started by user admin
Running in Durability level:
MAX_SURVIVABILITY [Pipeline] Start of
Pipeline
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/SamplePipeline
[Pipeline] {
[Pipeline] stage [Pipeline]
{ (Preparation) [Pipeline]
catchError [Pipeline] {
[Pipeline] sh
+ docker stop
samplerunning
samplerunning
[Pipeline] sh
+ docker rm
```

samplerunning
samplerunning
[Pipeline] }
[Pipeline] // catchError
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] build (Building BuildAppJob)
Scheduling project: BuildAppJob
Starting building: BuildAppJob #15
[Pipeline] }
[Pipeline] // stage

[Pipeline] stage [Pipeline]
{ (Results)
[Pipeline] build (Building TestAppJob)
Scheduling project: TestAppJob
Starting building: TestAppJob #18
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

Lab Session 11

Software Version Control with Git

Objectives

Part 1: Launch the

DEVASC VM Part 2:

Initializing Git

Part 3: Staging and Committing a File in the Git

Repository Part 4: Managing the File and Tracking

Changes

Part 5: Branches and Merging

Part 6: Handling Merge

Conflicts Part 7: Integrating

Git with GitHub

Background / Scenario

In this lab, you will explore the fundamentals of the distributed version control system Git, including most of the features you need to know in order to collaborate on a software project. You will also integrate your local Git repository with the cloud-based GitHub repository.

Required Resources

- 1 PC with operating system of your choice
- Virtual Box or VMWare
- DEVASC Virtual Machine

Instructions

Part 1: Launch the DEVASC VM

If you have not already completed the **Lab - Install the Virtual Machine Lab Environment**, do so now. If you have already completed that lab, launch the DEVASC VM now.

Part 2: Initializing Git

In this part, you will initialize a Git repository.

Step 1: Open a terminal in the DEVASC-LABVM.

Double-click the Terminal Emulator icon on the desktop.

Step 2: Initialize a Git Repository

a. Use the **ls** command to display a listing of the current directory. Remember that commands are case- sensitive.

devasc@labvm:~$ **ls**

Desktop Downloads Music Public Templates
Documents labs Pictures snap Videos
devasc@labvm:~$

b. Next, configure user information to be used for this local repository. This will associate your information the work that you contribute to a local repository. Use your name in place of "Sample User" for the name in quotes " ". Use @example.com for your email address.

   Note: These settings can be anything you want at this point. However, when you reset these global values in Part 7, you will use the username for your GitHub account. If you wish, you can use your GitHub username now.

   devasc@labvm:~$ **git config --global user.name "SampleUser"**
   devasc@labvm:~$ **git config --global user.email sample@example.com**

c. At any time, you can review these setting with the **git config --list** command.

   devasc@labvm:~$ **git config --list**
   user.name=SampleUser
   user.email=sample@example.com
   devasc@labvm:~$

d. Use the **cd** command to navigate to the **devnet-src** folder:

   devasc@labvm:~$ **cd labs/devnet-src/**
   devasc@labvm:~/labs/devnet-src$

e. Make a directory **git-intro** and change directory into it:

   devasc@labvm:~/labs/devnet-src$ **mkdir git-intro**
   devasc@labvm:~/labs/devnet-src$ **cd git-intro**
   devasc@labvm:~/labs/devnet-src/git-intro$

f. Use the **git init** command to initialize the current directory (git-intro) as a Git repository. The message displayed indicates that you have created a local repository within your project contained in the hidden directory **.git**. This is where all of your change history is located. You can see it with the **ls -a** command.

   devasc@labvm:~/labs/devnet-src/git-intro$ **git init**
   Initialized empty Git repository in /home/devasc/labs/devnet-src/git-intro/.git/
   devasc@labvm:~/labs/devnet-src/git-intro$ **ls -a**
   . .. .git
   devasc@labvm:~/labs/devnet-src/git-intro$

g. As you work on your project, you will want to check to see which files have changed. This is helpful when you are committing files to the repo, and you don't want to commit all of them. The **git status** command displays modified files in working directory that are staged for your next commit.

   This message tells you:

   - That you are on branch master. (Branches are discussed later in this lab)
   - The commit message is Initial commit.
   - There is nothing changed to commit.

   You will see that the status of your repo will change once you add files and start making

changes.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
devasc@labvm:~/labs/devnet-src/git-intro$
```

Part 3: Staging and Committing a File in the Repository

In this part you will create a file, stage that file, and commit that file to the Git repository.

Step 1: Create a file.

a. The **git-intro** repository is created but empty. Using the **echo** command, create the file **DEVASC.txt** with the information contained in quotes.

devasc@labvm:~/labs/devnet-src/git-intro$ **echo "I am on my way to passing the Cisco DEVASC exam" > DEVASC.txt**

devasc@labvm:~/labs/devnet-src/git-intro$

b. Use the **ls -la** command terify the file, as well as the .git directory, are in the **git intro** directory. Then use
cat to display the contents of **DEVASC.txt**.

devasc@labvm:~/labs/devnet-src/git-intro$ **ls -la**

total 16

drwxrwxr-x 3 devasc devasc 4096 Apr 17 20:38 .

drwxrwxr-x 5 devasc devasc 4096 Apr 17 19:50 ..

-rw-rw-r-- 1 devasc devasc 48 Apr 17 20:38

DEVASC.txt drwxrwxr-x 7 devasc devasc 4096 Apr 17

19:57 .git evasc@labvm:~/src/git-intro$ **cat**

DEVASC.txt

I am on my way to passing the Cisco DEVASC exam

devasc@labvm:~/labs/devnet-src/git-intro$

Step 2: Examine the Repository Status.

Examine the repository status using **git status**. Notice that Git found the new file in the directory, and knows that it's not tracked.

devasc@labvm:~/labs/devnet-src/git-intro$ **git status**

On branch

master No

commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed) DEVASC.txt

nothing added to commit but untracked files present (use "git add" to track)

devasc@labvm:~/labs/devnet-src/git-intro$

Step 3: Staging the File.

a. Next, use the **git add** command to "stage" the **DEVASC.txt** file. Staging is an intermediate phase prior to committing a file to the repository with the **git commit** command. This command creates a snapshot of the contents of the file at the time this command is entered. Any changes to the file require another **git add** command prior to committing the file.

   devasc@labvm:~/labs/devnet-src/git-intro$ **git add DEVASC.txt**

b. Using the **git status** command again, notice the staged changes displayed as "new file: DEVASC.txt".

   devasc@labvm:~/labs/devnet-src/git-intro$ **git status**

   On branch master

```
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   DEVASC.txt

devasc@labvm:~/labs/devnet-src/git-intro$
```

Step 4: Committing a File.

Now that you have staged your changes, you will need to commit them in order to let Git know you want to start tracking those changes. Commit your staged content as a new commit snapshot by using the **git commit** command. The **-m message** switch enables you to add a message explaining the changes you've made. Note the number and letter combination highlighted in the output. This is the commit ID. Every commit is identified by a unique SHA1 hash. The commit ID is the first 7 characters of the full commit hash. Your commit ID will be different than the one displayed.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git commit -m "Committing
DEVASC.txt to begin tracking changes"
[master (root-commit) b510f8e] Committing DEVASC.txt to begin tracking changes
 1 file changed, 1 insertion(+)
 create mode 100644
 DEVASC.txt
devasc@labvm:~/labs/devnet-src/git-intro$
```

Step 5: Viewing the Commit History.

Use the **git log** command to show all commits in the current branch's history. By default, all commits are made to the master branch. (Branches will be discussed later.) The first line is the commit hash with the commit ID as first 7 characters. The file is committed to the master branch. This is followed by your name and email address, the date of the commit and the message you included with the commit.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git log
commit b510f8e5f9f63c97432d108a0413567552c07356 (HEAD
-> master) Author: Sample User <sample@example.com>
Date:   Sat Apr 18 18:03:28 2020 +0000

    Committing DEVASC.txt to begin tracking changes
devasc@labvm:~/labs/devnet-src/git-intro$
```

Part 4: Modifying the File and Tracking the Changes

In this part, you will modify a file, stage the file, commit the file, and verify changes in the repository.

Step 1: Modify the file.

a. Make a change to DEVASC.txt using the **echo** command. Be sure to use ">>" to append the existing file. The ">" will overwrite the existing file. Use the **cat** command to view the modified file.

devasc@labvm:~/labs/devnet-src/git-intro$ **echo "I am beginning to understand Git!" >> DEVASC.txt**

b. Use the **cat** command to view the modified file.

devasc@labvm:~/labs/devnet-src/git-intro$ **cat DEVASC.txt**
I am on my way to passing the Cisco DEVASC
exam I am beginning to understand Git!
devasc@labvm:~/labs/devnet-src/git-intro$

Step 2: Verify the change to the repository.

Verify the change in the repository using the **git status** command.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   DEVASC.txt

no changes added to commit (use "git add" and/or "git commit -a")
devasc@labvm:~/labs/devnet-src/git-intro$
```

Step 3: Stage the modified file.

The modified file will need to be staged again before it can be committed using the **git add** command again.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git add DEVASC.txt
```

Step 4: Commit the staged file.

Commit the staged file using the **git commit** command. Notice the new commit ID.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git commit -m "Added additional
line to file"
[master 9f5c4c5] Added additional line to file
 1 file changed, 1 insertion(+)
devasc@labvm:~/labs/devnet-src/git-intro$
```

Step 5: Verify the changes in the repository.

a. Use the **git log** command again to show all commits. Notice that the log contains the original commit entry along with the entry for the commit you just performed. The latest commit is shown first. The output highlights the commit ID (first 7 characters of the SHA1 hash), the date/time of the commit, and the message of the commit for each entry.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git log
commit 9f5c4c5d630e88abe2a873fe48144e25ebe7bd6a (HEAD
-> master) Author: Sample User <sample@example.com>
Date:   Sat Apr 18 19:17:50 2020 +0000

    Added additional line to file

commit b510f8e5f9f63c97432d108a0413567552c07356
Author: Sample User
```

<sample@example.com> Date:

Sat Apr 18 18:03:28 2020 +0000

Committing DEVASC.txt to begin tracking changes

devasc@labvm:~/labs/devnet-src/git-intro$

b. When you have multiple entries in the log, you can compare the two commits using the **git diff** command adding original commit ID first and the latest commit second: **git diff** <commit ID original> <commit ID latest>. You will need to use your commit IDs. The "+" sign at the end, followed by the text indicates the content that was appended to the file.

devasc@labvm:~/labs/devnet-src/git-intro$ **git diff b510f8e 9f5c4c5**

```
diff --git a/DEVASC.txt b/DEVASC.txt
index 93cd3fb..085273f 100644
--- a/DEVASC.txt
+++
b/DEVASC.txt
@@ -1 +1,2
@@
 I am on my way to passing the Cisco DEVASC exam
+I am beginning to understand Git!
devasc@labvm:~/labs/devnet-src/git-intro$
```

Part 5: Branches and Merging

When a repository is created, the files are automatically put in a branch called **master**. Whenever possible it is recommended to use branches rather than directly updating the master branch. Branching is used so that you can make changes in another area without affecting the master branch. This is done to help prevent accidental updates that might overwrite existing code.

In this part, you will create a new branch, checkout the branch, make changes in the branch, stage and commit the branch, merge the branch changes to the master branch, and then delete the branch.

Step 1: Create a new branch

Create a new branch called **feature** using the **git branch** <branch-name> command.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git branch feature
```

Step 2: Verify current branch

Use the **git branch** command without a branch-name to display all the branches for this repository. The "*" next to the master branch indicates that this is the current branch – the branch that is currently "checked out".

```
devasc@labvm:~/labs/devnet-src/git-intro$ git branch
  feature
* master
devasc@labvm:~/labs/devnet-src/git-intro$
```

Step 3: Checkout the new branch

Use the **git checkout** <branch-name> command to switch to the feature branch.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git checkout feature
```

Step 4: Verify current branch

a. Verify you have switched to the feature branch using the **git branch** command. Note the "*" next to the

feature branch. This is now the working branch.

devasc@labvm:~/labs/devnet-src/git-intro$ **git branch**

* feature

  master

devasc@labvm:~/labs/devnet-src/git-intro$

b. Append a new line of text to the DEVASC.txt file, again using the **echo** command with the ">>" signs.

   devasc@labvm:~/labs/devnet-src/git-intro$ **echo "This text was added originally while in the feature branch" >> DEVASC.txt**

c. Verify the line was appended to the file using the **cat** command.

   devasc@labvm:~/labs/devnet-src/git-intro$ **cat DEVASC.txt**

   I am on my way to passing the Cisco DEVASC exam

I am beginning to understand Git!
This text was added originally while in the feature branch
devasc@labvm:~/labs/devnet-src/git-intro$

Step 5: Stage the modified file in the feature branch.

a. Stage the updated file to the current feature branch.

devasc@labvm:~/labs/devnet-src/git-intro$ **git add DEVASC.txt**

b. Use the **git status** command and notice the modified file **DEVASC.txt** is staged in the feature branch.

devasc@labvm:~/labs/devnet-src/git-intro$ **git status**
On branch feature
Changes to be
committed:
(use "git restore --staged <file>..." to unstage)
modified: DEVASC.txt

devasc@labvm:~/labs/devnet-src/git-intro$

Step 6: Commit the staged file in the feature branch.

a. Commit the staged file using the **git commit** command. Notice the new commit ID and your message.

devasc@labvm:~/labs/devnet-src/git-intro$ **git commit -m "Added a third line in feature branch"**
[feature cd828a7] Added a third line in feature branch
1 file changed, 1 insertion(+)
devasc@labvm:~/labs/devnet-src/git-intro$

b. Use the **git log** command to show all commits including the commit you just did to the feature branch. The prior commit was done within the master branch.

devasc@labvm:~/labs/devnet-src/git-intro$ **git log**
commit cd828a73102cf308981d6290113c358cbd387620 (HEAD
-> feature) Author: Sample User <sample@example.com>
Date: Sat Apr 18 22:59:48 2020 +0000

Added a third line in feature branch

commit 9f5c4c5d630e88abe2a873fe48144e25ebe7bd6a (master)
Author: Sample User
<sample@example.com> Date:
Sat Apr 18 19:17:50 2020 +0000

Added additional line to file

commit b510f8e5f9f63c97432d108a0413567552c07356
Author: Sample User
<sample@example.com> Date:
Sat Apr 18 18:03:28 2020 +0000

Committing DEVASC.txt to begin tracking changes
devasc@labvm:~/labs/devnet-src/git-intro$

Step 7: Checkout the master branch.

Switch to the master branch using the **git checkout** master command and verify the current working branch using the **git branch** command.

devasc@labvm:~/labs/devnet-src/git-intro$ **git checkout master**

Switched to branch 'master'

devasc@labvm:~/labs/devnet-src/git-intro$ **git branch**

  feature

* master

devasc@labvm:~/labs/devnet-src/git-intro$

Step 8: Merge file contents from feature to master branch.

a. Branches are often used when implementing new features or fixes. They can be submitted for review by team members, and then once verified, can be pulled into the main codebase – the master branch.

   Merge the contents (known as the history) from the feature branch into the master branch using the **git merge** <branch-name> command. The branch-name is the branch that histories are pulled from into the current branch. The output displays that one file was changed with one line inserted.

   devasc@labvm:~/labs/devnet-src/git-intro$ **git merge feature**

   Updating

   9f5c4c5..cd828a7

   Fast-forward

   DEVASC.txt | 1 +

   1 file changed, 1 insertion(+)

   devasc@labvm:~/labs/devnet-src/git-intro$

b. Verify the appended content to the DEVASC.txt file in the master branch using the **cat** command.

   devasc@labvm:~/labs/devnet-src/git-intro$ **cat DEVASC.txt**

   I am on my way to passing the Cisco DEVASC

   exam I am beginning to understand Git!

   This text was added originally while in the feature branch

   devasc@labvm:~/labs/devnet-src/git-intro$

Step 9: Deleting a branch.

a. Verify the **feature** branch is still available using the **git branch** command.

   devasc@labvm:~/labs/devnet-src/git-intro$ **git branch**

     feature

   * master

   devasc@labvm:~/labs/devnet-src/git-intro$

b. Delete the **feature** branch using the **git branch -d** <branch-name> command. .

devasc@labvm:~/labs/devnet-src/git-intro$ **git branch -d feature**
Deleted branch feature (was cd828a7).
devasc@labvm:~/labs/devnet-src/git-intro$

c. Verify the feature branch is no longer available using the **git branch** command.

devasc@labvm:~/labs/devnet-src/git-intro$ **git branch**
* master
devasc@labvm:~/labs/devnet-src/git-intro$

Part 6: Handling Merge Conflicts

At times, you may experience a merge conflict. This is when you may have made overlapping changes to a file, and Git cannot automatically merge the changes.

In this Part, you will create a test branch, modify its content, stage and commit the test branch, switch to the master branch, modify the content again, stage and commit the master branch, attempt to merge branches, locate and resolve the conflict, stage and commit the master branch again, and verify your commit.

Step 1: Create a new branch test.

Create a new branch **test**.

devasc@labvm:~/labs/devnet-src/git-intro$ **git branch test**

Step 2: Checkout the branch test.

a. Checkout (switch to) the branch **test**.

devasc@labvm:~/labs/devnet-src/git-intro$ **git checkout test**
Switched to branch 'test'
devasc@labvm:~/labs/devnet-src/git-intro$

b. Verify the working branch is the **test** branch.

devasc@labvm:~/labs/devnet-src/git-intro$ **git branch**
  master
* test
devasc@labvm:~/labs/devnet-src/git-intro$

Step 3: Verify the current contents of DEVASC.txt.

Verify the current contents of the **DEVASC.txt** file. Notice the first line includes the word "Cisco".

devasc@labvm:~/labs/devnet-src/git-intro$ **cat DEVASC.txt**
I am on my way to passing the Cisco DEVASC
exam I am beginning to understand Git!
This text was added originally while in the feature branch
devasc@labvm:~/labs/devnet-src/git-intro$

Step 4: Modify the contents of DEVASC.txt in the test branch.

Use the **sed** command to change the word "Cisco" to "NetAcad" in the **DEVASC.txt** file.

devasc@labvm:~/labs/devnet-src/git-intro$ **sed -i 's/Cisco/NetAcad/' DEVASC.txt**

Step 5: Verify the contents of the modified DEVASC.txt in the test branch.

Verify the change to the **DEVASC.txt** file.

devasc@labvm:~/labs/devnet-src/git-intro$ **cat DEVASC.txt**

I am on my way to passing the NetAcad DEVASC
exam I am beginning to understand Git!
This text was added originally while in the feature branch
devasc@labvm:~/labs/devnet-src/git-intro$

Step 6: Stage and commit the test branch.

Stage and commit the file with a single **git commit -a** command. The **-a** option only affects files that have been modified and deleted. It does not affect new files.

devasc@labvm:~/labs/devnet-src/git-intro$ **git commit -a -m "Change Cisco to NetAcad"**
[test b6130a6] Change Cisco to NetAcad
1 file changed, 1 insertion(+), 1 deletion(-)
devasc@labvm:~/labs/devnet-src/git-intro$

Step 7: Checkout the master branch.

a. Checkout (switch to) the **master** branch.

devasc@labvm:~/labs/devnet-src/git-intro$ **git checkout master**
Switched to branch 'master'
devasc@labvm:~/labs/devnet-src/git-intro$

b. Verify that the master branch is your current working branch.

devasc@labvm:~/labs/devnet-src/git-intro$ **git branch**
*
maste
r test
devasc@labvm:~/labs/devnet-src/git-intro$

Step 8: Modify the contents of DEVASC.txt in the master branch.

Use the **sed** command to change the word "Cisco" to "DevNet" in the DEVASC.txt file.

devasc@labvm:~/labs/devnet-src/git-intro$ **sed -i 's/Cisco/DevNet/' DEVASC.txt**

Step 9: Verify the contents of the modified DEVASC.txt in the master branch.

Verify the change to the file.

devasc@labvm:~/labs/devnet-src/git-intro$ **cat DEVASC.txt**
I am on my way to passing the DevNet DEVASC
exam I am beginning to understand Git!
This text was added originally while in the feature branch
devasc@labvm:~/labs/devnet-src/git-intro$

Step 10: Stage and commit the master branch.

Stage and commit the file using the **git commit -a** command.

devasc@labvm:~/labs/devnet-src/git-intro$ **git commit -a -m "Changed Cisco to DevNet"**
[master 72996c0] Changed Cisco to DevNet
1 file changed, 1 insertion(+), 1 deletion(-)

devasc@labvm:~/labs/devnet-src/git-intro$

Step 11: Attempt to merge the test branch into the master branch.

Attempt to merge the test branch history into the master branch.

devasc@labvm:~/labs/devnet-src/git-intro$ **git merge test**
Auto-merging DEVASC.txt
CONFLICT (content): Merge conflict in DEVASC.txt

Automatic merge failed; fix conflicts and then commit the result.
devasc@labvm:~/labs/devnet-src/git-intro$

Step 12: Find the conflict.

a. Use the **git log** command to view the commits. Notice that the HEAD version is the master branch. This will be helpful in the next step.

devasc@labvm:~/labs/devnet-src/git-intro$ **git log**
commit 72996c09fa0ac5dd0b8ab9ec9f8530ae2c5c4eb6 (HEAD
-> master) Author: Sample User <sample@example.com>
Date: Sun Apr 19 00:36:05 2020 +0000

Changed Cisco to DevNet

<output omitted>

b. Use the **cat** command to view the contents of the DEVASC.txt file. The file now contains information to help you find the conflict. The HEAD version (master branch) containing the word "DevNet" is conflicting with the test branch version and the word "NetAcad".

devasc@labvm:~/labs/devnet-src/git-intro$ **cat DEVASC.txt**
<<<<<<< HEAD
I am on my way to passing the DevNet DEVASC exam
=======
I am on my way to passing the NetAcad DEVASC exam
>>>>>>> test
I am beginning to understand Git!
This text was added originally while in the feature branch
devasc@labvm:~/labs/devnet-src/git-intro$

Step 13: Manually edit the DEVASC.txt file to remove the conflicting text.

a. Use the **vim** command to edit the file.

devasc@labvm:~/labs/devnet-src/git-intro$ **vim DEVASC.txt**

b. Use the up and down arrow to select the proper line of text. Press **dd** (delete) on the following lines that are highlighted. **dd** will delete the line the cursor is on.

<<<<<<< HEAD
I am on my way to passing the DevNet DEVASC exam
=======
I am on my way to passing the NetAcad DEVASC exam
>>>>>>> test
I am beginning to understand Git!
This text was added originally while in the feature branch

c. Save your changes in vim by pressing **ESC** (the escape key) and then typing **:** (colon) followed by **wq**
and press enter.

ESC

:

wq

<Enter or Return>

Step 14: Verify your edits of DEVASC.txt in the master branch.

Verify you changes using the **cat** command.

```
devasc@labvm:~/labs/devnet-src/git-intro$ cat DEVASC.txt
I am on my way to passing the DevNet DEVASC
exam I am beginning to understand Git!
This text was added originally while in the feature branch
devasc@labvm:~/labs/devnet-src/git-intro$
```

Step 15: Stage and commit the master branch.

Stage and commit DEVASC.txt to the master branch using the **git commit -a** command.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git add DEVASC.txt
devasc@labvm:~/labs/devnet-src/git-intro$ git commit -a -m "Manually merged
from test branch"
[master 22d3da4] Manually merged from test branch
devasc@labvm:~/labs/devnet-src/git-intro$
```

Step 16: Verify the commit.

Use the **git log** command to verify the commit. If necessary, you can use **q** to quit out of the git log display.

```
devasc@labvm:~/labs/devnet-src/git-intro$ git log
commit 22d3da41e00549ce69dc145a84884af6a1697734 (HEAD
-> master) Merge: 72996c0 b6130a6
Author: Sample User <sample@example.com>
Date:   Sun Apr 19 01:09:53 2020 +0000

    manually merged from branch test
<output omitted>
```

Part 7: Integrating Git with GitHub

So far, all the changes you have made to your file have been stored on your local machine. Git runs locally and does not require any central file server or cloud-based hosting service. Git allows a user to locally store and manage files.

Although Git is useful for a single user, integrating the local Git repository with a cloud-based server like GitHub is helpful when working within a team. Each team member keeps a copy on the repository on their local machine and updates the central cloud-based repository to share any changes.

There are quite a few popular Git services, including GitHub, Stash from Atlassian, and GitLab. Because it is readily accessible, you will use GitHub in these examples.

Step 1: Create a GitHub Account.

If you have not so previously, go to github.com and create a GitHub account. If you have a

GitHub account go to step 2.

Step 2: Log into your GitHub Account Create a Repository.

Log into your GitHub account.

Step 3: Create a Repository.

a. Select the "**New repository**" button or click on the "**+**" icon in the upper right corner and select "**New repository**".

b. Create a repository using the following

information: Repository name:

devasc-study-team

Description: **Working together to pass the DEVASC exam**

Public/Private: **Private**

c. Select: **Create repository**

Step 4: Create a new directory devasc-study-team.

a. If you are not already in the **git-intro** directory, change to it now.

devasc@labvm:~$ **cd ~/labs/devnet-src/git-intro**

b. Make a new directory called devasc-study-team. The directory does not have to match the name as the repository.

devasc@labvm:~/labs/devnet-src/git-intro$ **mkdir devasc-study-team**

Step 5: Change directory to devasc-study-team.

Use the **cd** command to change directories to **devasc-study-team**.

devasc@labvm:~/labs/devnet-src/git-intro$ **cd devasc-study-team**
devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

Step 6: Copy the DEVASC file.

a. Use the **cp** command to copy the **DEVASC.txt** from **git-intro** parent directory to the **devasc-study-team** sub-directory. The two periods and a slash prior the file name indicates the parent directory. The space and period following the file name indicates to copy file in the current directory with the same file name.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **cp ../DEVASC.txt .**

b. Verify the file was copied with the **ls** command and the contents of the file with the **cat** command.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **ls**
DEVASC.txt
devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **cat DEVASC.txt**
I am on my way to passing the DevNet DEVASC
exam I am beginning to understand Git!
This text was added originally while in the feature branch
devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

Step 7: Initialize a new Git repository.

a. Use the **git init** command to initialize the current directory (devasc-study-team) as a Git repository. The message displayed indicates that you have created a local repository within your project contained in the hidden directory **.git**. This is where all of your change history is located.

   devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git init** Initialized empty Git repository in /home/devasc/src/git-intro/devasc-study-team/.git/

   devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

b. Next, check your global git variables with the **git config --list** command.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git config --list**

user.name=SampleUser

user.email=sample@example.c

om

core.repositoryformatversion=0

core.filemode=true

core.bare=false

core.logallrefupdates=true

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

c. If the user.name and user.email variables do not match your GitHub credentials, change them now.

devasc@labvm:~$ **git config --global user.name "**GitHub username**"**

devasc@labvm:~$ **git config --global user.email** GitHub-email-address

Step 8: Point Git repository to GitHub repository.

a. Use the **git remote add** command to add a Git URL as a remote alias. The value "origin" points to the newly created repository on GitHub. Use your GitHub username in the URL path for github-username.

Note: Your username is case-sensitive.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git remote add origin https://github.com/github-username/devasc-study-team.git**

b. Verify the **remote** is running on github.com.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git remote -- verbose**

origin https://github.com/username/devasc-study-team.git (fetch)

origin https://github.com/username/devasc-study-team.git (push)

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

c. View the **git log**. The error indicates that there are no commits.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git log**

fatal: your current branch 'master' does not have any commits yet

Step 9: Stage and Commit the DEVASC.txt file.

a. Use the **git add** command to stage the DEVASC.txt file.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git add DEVASC.txt**

b. Use **git commit** command to commit the DEVASC.txt file.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git commit -m "Add DEVASC.txt file to devasc-study-team"**

[master (root-commit) c60635f] Add DEVASC.txt file to devasc-study-team

1 file changed, 3 insertions(+)

create mode 100644
DEVASC.txt
devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

Step 10: Verify the commit.

a. Use the **git log** command to verify the commit.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git log**
commit c60635fe4a1f85667641afb9373e7f49a287bdd6 (HEAD
-> master) Author: username <user@example.com>

Date: Mon Apr 20 02:48:21 2020 +0000

Add DEVASC.txt file to devasc-study-team

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

b. Use the **git status** command to view status information. The phrase "working tree clean" means that Git has compared your file listing to what you've told Git, and it's a clean slate with nothing new to report.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git status**

On branch master

nothing to commit, working tree clean

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

Step 11: Send (push) the file from Git to GitHub.

Use the **git push origin master** command to send (push) the file to your GitHub repository. You will be prompted for a username and password, which will be the one you used to create your GitHub account.

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$ **git push origin master**

Username for 'https://github.com': **username**

Password for 'https://username@github.com': **password**

Enumerating objects: 3, done.

Counting objects: 100% (3/3), done.

Delta compression using up to 2

threads Compressing objects: 100%

(2/2), done.

Writing objects: 100% (3/3), 347 bytes | 347.00 KiB/s, done. Total

3 (delta 0), reused 0 (delta 0)

To https://github.com/username/devasc-study-team.git

* [new branch] master -> master

devasc@labvm:~/labs/devnet-src/git-intro/devasc-study-team$

Note: If, after entering your username and password, you get a fatal error stating repository is not found, you most likely submitted an incorrect URL. You will need to reverse your **git add** command with the **git remote rm origin** command.

Step 12: Verify file on GitHub.

a. Go to your GitHub account and under "Repositories" select **username/devasc-study-team**.

b. You should see that the DEVASC.txt file has been added to this GitHub repository. Click on the file to view the contents.

Lab Session 12

Course Code and Title: SE-312 Software Construction and Development

Select any one: **Open Ended Lab/~~Problem Based Learning / Design Project~~**

PROGRAMME LEARNING OUTCOMES COVERED	
PLO-2	Problem Analysis
PLO-7	Environment and Sustainability

COGNITIVE LEVELS COVERED	
C2	Comprehension
C3	Application

PROBLEM STATEMENT:
Design and develop a software system that addresses a real-world challenge related to sustainable development. The system should provide solutions to optimize the use, management, or distribution of critical resources such as water, energy, food, or transportation in a way that promotes sustainability, efficiency, and long-term impact**.** The problem should be framed based on one or more of the **UN** Sustainable Development Goals (SDGs)**.**

Requirements:

- Identify and define a specific problem related to sustainability that the software will solve.

- Perform a detailed problem analysis, identifying stakeholders, key requirements, and constraints.
- Propose a software solution, including high-level design, architecture, and technologies involved.
- Develop and implement the system following all phases of software construction, including design, coding, testing, and deployment.

Deliverables:

1. **Problem Definition Document (PDD)**:
    - Description of the specific problem related to sustainability.
    - Stakeholder analysis.
    - Functional and non-functional requirements.
    - Key assumptions, constraints, and risks.
2. **System Architecture and Design**:
    - High-level architectural diagrams.
    - Detailed design document including class diagrams, database schema, and user interfaces.
3. **Project Plan**:
    - Work breakdown structure (WBS).
    - Timeline and milestones for each phase of development.
    - Risk management plan.
4. **Software Implementation**:
    - Fully functional codebase of the developed software.
    - Demonstration of the working system.
5. **Testing Plan and Test Cases**:
    - Unit, integration, and system testing with detailed test cases.
    - Bug report and how issues were resolved.
6. **Deployment Strategy Document:**
    - Deployment environment and platform.
    - Deployment process (manual/automated), including any dependencies.
    - Backup, recovery, and security measures.
7. **Maintenance and Scalability Plan:**
    - Monitoring and logging strategies.
    - Bug tracking and issue resolution workflows.
    - Update process for new features, patches, or improvements.
    - Scalability strategy to handle increased load or additional features.

Rubrics for Evaluating Complex Engineering Activity
NED University of Engineering & Technology
Department of Software Engineering

Course Code and Title: SE-313 Software Construction and Development

Assessment Rubric for CEP/CEA

Batch: 2022 Class: TESE Semester: Fall 2024

Criterion	LevelofAttainment				
	Below Average (0)	**Average (2)**	**Satisfactory (3)**	**VeryGood (4)**	**Excellent (5)**
How well does the student understand and analyze the problem?	Lacks understanding; fails to analyze or grasp the core problem.	Partial understanding with superficial analysis; key aspects are misunderstood or missing.	Basic understanding; most aspects of the problem are covered but lacks complexity or key details.	Good understanding; problem addressed well but lacks some depth or critical insight.	Demonstrates profound understanding, providing deep insights and fully addresses all facets of the problem.
How well is the design and architecture structured and presented	No clear design presented; fails to meet functional requirements	Incomplete or inconsistent design; architecture does not address core needs and is poorly	Basic design that meets essential requirements but lacks innovation or scalability.	Solid design that meets most requirements; architecture is effective with minor improveme	Exceptional design with innovative solutions; architecture is highly efficient, scalable, and well-documented.

		structured.		nt areas	
How well is the code organized and maintainable?	Code is disorganized, unstructured, and mostly non-functional.	Code is poorly organized, lacks modularity, and has many errors; difficult to maintain.	Code is functional but lacks clarity, modularity, or structure; several practices are not followed.	Code is well-organized and functional but may have minor inefficiencies or formatting issues.	Code is exceptionally well-written, clean, modular, and fully adheres to best practices; highly maintainable.
How effectively is error handling and system resilience implemented?	No error handling present; system is unstable and fails frequently.	Minimal error handling; system is vulnerable to failures or crashes.	Basic error handling; only some edge cases considered; fails under certain conditions.	Good error handling; covers most scenarios but lacks coverage of extreme cases.	Comprehensive error handling with robust recovery mechanisms; system remains stable under all conditions.
How thorough is the student's testing and validation?	No testing performed; functionality and correctness are not validated.	Minimal testing with limited coverage; insufficient documentation of results and errors.	Basic testing with some major cases covered but inadequate validation of edge cases; documentation lacking.	Well-conducted testing, covering major cases with good documentation; minor gaps in edge scenarios..	Exhaustive testing with thorough documentation; all critical cases and edge scenarios are rigorously validated.
How well does the student demonstrate creativity and solve a technical challenge?	No innovation; relies on basic or copied solutions with no creativity.	Minimal innovation; solutions are overly simple or follow common patterns without challenge.	Solves the problem with standard solutions; moderate technical challenges attempted.	Shows creativity and tackles some complex problems; solid technical challenge is evident.	Demonstrates exceptional creativity and tackles significant technical challenges with advanced solutions.

How effectively is the deployment strategy and maintenance plan outlined?	No deployment strategy or maintenance plan provided.	Deployment strategy is unclear and manual; maintenance plan lacks coherence and detail.	Basic deployment plan with minimal automation; maintenance plan is vague and lacks structure.	Good deployment strategy but lacks some automation; maintenance plan exists but is not detailed	Comprehensive, automated deployment strategy with clear, detailed maintenance plan ensuring long-term success.
How well has the project aligned its objectives and outcomes with specific Sustainable Development Goals, assessing potential impacts?	Lack of evident mapping between project objectives/ outcomes and SDGs, entirely missing the connection or understanding of relevant sustainable development goals.	Incomplete or unclear mapping between project objectives/ outcomes and SDGs, lacking significant connection or understanding of relevant goals.	Partial mapping, evident gaps in connecting project objectives/ outcomes with SDGs, requiring improvements in aligning with relevant goals.	Adequate mapping with minor areas needing further connection or explanation between project outcomes and SDGs, mostly effective in aligning with relevant goals.	Thorough mapping of project objectives and outcomes with specific Sustainable Development Goals (SDGs), demonstrating comprehensive alignment and impact assessment.
How coherent and comprehensive is the documentation, ensuring clarity and organization in showcasing the requirements engineering process?	Inadequate documentation/presentation, entirely missing key aspects of the requirements engineering process, lacking clarity and	Incomplete or disorganized documentation/presentation, lacking clarity and organization, requiring significant improvements in overall presentatio	Incomplete or disorganized documentation, lacking clarity and organization, needing substantial improvements in overall presentati	Adequate documentation, some minor organizational or clarity issues, mostly effective in presenting the requirements engineering process.	Comprehensive, well-organized documentation of requirements engineering process, exhibiting exceptional clarity and organization.

	organizati on.	n.	on.		

Student's Name:_____RollNo.:_____

Total Score: _____ Instructor's Signature: _____