College of Engineering & Applied Sciences

CSPB 3308

Software Devlopment Methods And Tools

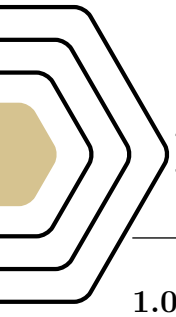
Class Notes

University Of Colorado

2024

Software Devlopment Methods And Tools - Class Notes

Life In The Shell



Life In The Shell

1.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- **The Linux Command Line - Chapter 1 - What Is The Shell?**
- **The Linux Command Line - Chapter 2 - Navigation**
- **The Linux Command Line - Chapter 3 - Exploring The System**
- **The Linux Command Line - Chapter 4 - Manipulating Files And Directories**
- **The Linux Command Line - Chapter 5 - Working With Commands**
- **The Linux Command Line - Chapter 6 - Redirection**
- **The Linux Command Line - Chapter 11 - The Environment**
- **The Linux Command Line - Chapter 12 - A Gentle Introduction To vi**

1.0.2 Lectures

The lectures for this week are:

- Overview Of Course By Dr. Knox ≈ 10 **min**.
- Introduction To Software Development ≈ 16 **min**.
- Introduction To Linux ≈ 13 **min**.
- The Bash Shell ≈ 17 **min**.
- Vim Terminal Editor ≈ 13 **min**.
- How To Learn New Tools ≈ 17 **min**.
- Setting Up GitHub Classroom ≈ 28 **min**.
- Why We Use GitHub Classroom ≈ 8 **min**.
- Example Of Accepting An Assignment ≈ 7 **min**.
- Lab 1 Introduction ≈ 20 **min**.

The lecture notes for this week are:

- Explain Shell Tool

1.0.3 Assignments

The assignment(s) for this week are:

- Lab 1 - Using Command Line Interface

1.0.4 Quiz

The quiz for this week is:

- Quiz 1A - TLCL Chapters 1 - 4
- Quiz 1B - TLCL Chapters 5, 6, 11

1.0.5 Project

The assignment(s) for the project this week is:

- Availability And Skills Assessment

1.0.6 Chapter Summary

The first chapter that is covered this week is **Chapter 1: What Is The Shell?**.

Chapter 1: What Is the Shell?

Overview

This chapter introduces the shell, a program that takes keyboard commands and passes them to the operating system to execute. The most commonly used shell in Linux is `bash`, which stands for "Bourne Again SHell," an enhanced version of the original Unix shell `sh` created by Steve Bourne.

Terminal Emulators

In graphical user interfaces (GUI), a terminal emulator is needed to interact with the shell. Common terminal emulators include `konsole` for KDE and `gnome-terminal` for GNOME. These emulators provide access to the shell and may have different features and preferences.

Making Your First Keystrokes

Upon launching a terminal emulator, a shell prompt appears, typically displaying the format `username@machinename:directo` This prompt indicates the shell is ready to accept input. If the prompt ends with a `#` instead of a `$`, it signifies superuser (root) privileges.

To test the shell, users can type gibberish which results in an error message indicating the command was not found. The shell also supports command history, allowing users to recall previous commands using the up-arrow and down-arrow keys.

Cursor Movement

The left and right-arrow keys allow users to move the cursor within the command line for easy editing.

A Few Words About Mice and Focus

The terminal emulator supports mouse interactions, such as copying and pasting text using the middle mouse button. Traditional X Window System behavior allows "focus follows mouse," where the window under the mouse pointer receives input focus.

Try Some Simple Commands

Simple commands include:

Simple Commands

- `date`: Displays the current date and time.
- `cal`: Displays a calendar of the current month.
- `df`: Shows disk space usage.
- `free`: Displays memory usage.

Ending a Terminal Session

To end a terminal session, users can close the terminal emulator window, enter the `exit` command, or press `Ctrl-d`.

The Console Behind the Curtain

Linux systems support virtual terminals, accessible via `Ctrl-Alt-F1` through `Ctrl-Alt-F6`. Each provides a login prompt and can be switched using `Alt-F1` to `Alt-F6`, with `Alt-F7` typically returning to the graphical desktop.

Summing Up

This chapter introduces the basics of the Linux shell and terminal usage, covering command input, simple commands, and session management. The next chapter will explore more commands and navigation within the Linux file system.

Further Reading

- Steve Bourne - Father of the Bourne Shell.
- Brian Fox - Original author of `bash`.
- Shell (computing) - Concept of shells in computing.

The next chapter that is covered this week is **Chapter 2: Navigation**.

Chapter 2: Navigation

Overview

This chapter focuses on navigating the file system in a Linux environment. It introduces essential commands for directory navigation, including `pwd` (print working directory), `cd` (change directory), and `ls` (list directory contents).

Understanding the File System Tree

Linux organizes its files in a hierarchical directory structure, similar to a tree. The root directory is the starting point, and it contains files and subdirectories. Unlike Windows, which has separate file system trees for each storage device, Linux maintains a single file system tree regardless of the number of attached drives or storage devices.

The Current Working Directory

The current working directory is the directory in which a user is currently located. To display it, the `pwd` command is used.

Example: Print Working Directory

```
[me@linuxbox ~]$ pwd
/home/me

```

When a terminal session starts, the current working directory is set to the user's home directory.

Listing the Contents of a Directory

The `ls` command lists the files and directories in the current working directory. It can also list the contents of any specified directory.

Example: List Directory Contents

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos

```

Changing the Current Working Directory

To change the working directory, the `cd` command is used followed by the pathname of the desired directory. Pathnames can be absolute or relative.

Absolute Pathnames An absolute pathname starts from the root directory and follows the tree structure.

Example: Absolute Pathname

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin

```

Relative Pathnames A relative pathname starts from the current working directory and uses special notations `.` (current directory) and `..` (parent directory).

Example: Relative Pathname

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr

```

Some Helpful Shortcuts

The `cd` command has several shortcuts:

- `cd` - Changes to the home directory.
- `cd -` - Changes to the previous working directory.
- `cd ~user_name` - Changes to the home directory of `user_name`.

Important Facts About Filenames

Filenames in Linux have several key characteristics:

- Filenames starting with a period are hidden.
- Filenames are case-sensitive.
- Linux does not rely on file extensions to determine file types.
- Limit punctuation in filenames to periods, dashes, and underscores. Avoid spaces.

Summing Up

This chapter covered the basics of navigating the Linux file system using commands like `pwd`, `cd`, and `ls`. Understanding absolute and relative pathnames, as well as useful shortcuts, helps in efficiently managing directories. The next chapter will build on this knowledge to explore a modern Linux system.

The next chapter that is covered this week is **Chapter 3: Exploring the System**.

Chapter 3: Exploring the System

Overview

This chapter provides a guided tour of a Linux system, introducing additional commands to explore and understand the file system. The key commands covered include `ls` (list directory contents), `file` (determine file type), and `less` (view file contents).

Having More Fun with `ls`

The `ls` command is versatile and essential for viewing directory contents and file attributes. It can be used with various options to enhance its output:

- `ls -l` for long format
- `ls -lt` to sort by modification time
- `ls -lt -reverse` to reverse the sort order

Common `ls` Options

- `-a`, `-all`: List all files, including hidden files.
- `-A`, `-almost-all`: List all files except `.` and `..`
- `-d`, `-directory`: List directory details instead of contents.
- `-F`, `-classify`: Append indicator characters to file names.
- `-h`, `-human-readable`: Display file sizes in human-readable format.
- `-l`: Display results in long format.
- `-r`, `-reverse`: Display results in reverse order.
- `-S`: Sort results by file size.
- `-t`: Sort by modification time.

Determining a File's Type with `file`

The `file` command is used to identify the type of a file. It examines the file and provides a brief description of its contents.

Example: Using `file`

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01

```

Viewing File Contents with `less`

The `less` command allows viewing the contents of text files in a scrollable manner. It is useful for examining configuration files and scripts.

Common `less` Commands

- `Page Up` or `b`: Scroll back one page.
- `Page Down` or `space`: Scroll forward one page.
- `Up arrow`: Scroll up one line.
- `Down arrow`: Scroll down one line.
- `G`: Move to the end of the text file.
- `1G` or `g`: Move to the beginning of the text file.
- `/characters`: Search forward for characters.
- `n`: Search for the next occurrence of the previous search.
- `h`: Display help screen.
- `q`: Quit `less`.

Taking a Guided Tour

The file system layout on a Linux system adheres to the Linux Filesystem Hierarchy Standard. Key directories and their contents include:

Key Directories

- `/`: The root directory.
- `/bin`: Essential system binaries.
- `/boot`: Contains the Linux kernel and boot loader.
- `/dev`: Contains device nodes.
- `/etc`: System-wide configuration files.
- `/home`: User home directories.
- `/lib`: Shared libraries for core system programs.
- `/media`: Mount points for removable media.
- `/mnt`: Mount points for manually mounted devices.
- `/opt`: Optional software.
- `/proc`: Virtual file system for kernel information.
- `/root`: Home directory for the root account.
- `/sbin`: System binaries for superuser.
- `/tmp`: Temporary files.
- `/usr`: Programs and support files for users.
- `/var`: Variable data like logs and databases.

Symbolic Links

Symbolic links (symlinks) are special files that point to other files. They are useful for version management and easier access to frequently used files.

Hard Links

Hard links are another type of link that allow multiple filenames to refer to the same file data.

Summing Up

This chapter explored the Linux file system, providing insight into its structure and the tools available to navigate and understand it. Key takeaways include the use of `ls`, `file`, and `less` commands, as well as an overview of important directories and links.

Further Reading

- Linux Filesystem Hierarchy Standard
- Unix directory structure
- ASCII text format

The next chapter that is covered this week is **Chapter 4: Manipulating Files and Directories**.

Chapter 4: Manipulating Files and Directories

Overview

This chapter introduces five fundamental commands for manipulating files and directories in Linux: `cp` (copy), `mv` (move/rename), `mkdir` (create directories), `rm` (remove), and `ln` (create links). These commands provide powerful and flexible ways to manage files and directories, often more effectively than using a graphical file manager.

Wildcards

Wildcards, also known as globbing, allow users to specify groups of filenames based on patterns. Common wildcards include:

- `*`: Matches any characters
- `?`: Matches any single character
- `[characters]`: Matches any character in the set
- `[!characters]`: Matches any character not in the set
- `[[:class:]]`: Matches any character in the specified class (e.g., `[:alnum:]`, `[:alpha:]`, `[:digit:]`, `[:lower:]`, `[:upper:]`)

`mkdir` - Create Directories

The `mkdir` command creates directories. Multiple directories can be created simultaneously by listing them as arguments.

Example: Create Directories

```
mkdir dir1 dir2 dir3

```

`cp` - Copy Files and Directories

The `cp` command copies files or directories. It can be used in two ways:

- `cp item1 item2`: Copies `item1` to `item2`.
- `cp item... directory`: Copies multiple items into a directory.

Common `cp` Options

- `-a`, `-archive`: Copy files and directories with all attributes.
- `-i`, `-interactive`: Prompt before overwriting files.
- `-r`, `-recursive`: Recursively copy directories.
- `-u`, `-update`: Copy only newer or non-existing files.
- `-v`, `-verbose`: Display informative messages during copy.

`mv` - Move and Rename Files

The `mv` command moves or renames files and directories. Usage is similar to `cp`:

- `mv item1 item2`: Moves/renames `item1` to `item2`.
- `mv item... directory`: Moves multiple items into a directory.

Common `mv` Options

- `-i`, `-interactive`: Prompt before overwriting files.
- `-u`, `-update`: Move only newer or non-existing files.
- `-v`, `-verbose`: Display informative messages during move.

`rm` - Remove Files and Directories

The `rm` command deletes files and directories.

Common `rm` Options

- `-i`, `-interactive`: Prompt before deleting files.
- `-r`, `-recursive`: Recursively delete directories.
- `-f`, `-force`: Ignore nonexistent files and do not prompt.
- `-v`, `-verbose`: Display informative messages during deletion.

`ln` - Create Links

The `ln` command creates hard or symbolic links.

Hard Links Hard links reference the same data as the original file. Limitations include:

- Cannot reference files outside their file system.
- Cannot reference directories.

Example: Create Hard Link

```
ln file link

```

Symbolic Links Symbolic links contain a text pointer to the target file or directory, overcoming the limitations of hard links.

Example: Create Symbolic Link

```
ln -s item link

```

Let's Build a Playground

To practice file manipulation, a safe directory (playground) can be created in the home directory. Commands to create and manipulate files and directories in the playground include:

- `mkdir playground`
- `cp /etc/passwd playground/`
- `mv playground/passwd playground/fun`
- `ln playground/fun playground/fun-hard`
- `ln -s playground/fun playground/fun-sym`
- `rm playground/fun`

Summing Up

This chapter covered essential commands for file and directory manipulation, including creating, copying, moving, renaming, linking, and deleting files and directories. Practicing these commands in a safe environment helps in mastering them for effective use.

Further Reading

- Discussion of symbolic links

The next chapter that is covered this week is **Chapter 5: Working with Commands**.

Chapter 5: Working with Commands

Overview

This chapter demystifies various commands, options, and arguments used in Linux. It introduces the following commands to help understand and create commands:

- `type` - Indicate how a command name is interpreted
- `which` - Display which executable program will be executed
- `help` - Get help for shell builtins
- `man` - Display a command's manual page
- `apropos` - Display a list of appropriate commands
- `info` - Display a command's info entry
- `whatis` - Display one-line manual page descriptions
- `alias` - Create an alias for a command

What Exactly Are Commands?

A command can be one of four types:

- An executable program (e.g., compiled binaries or scripts)
- A command built into the shell (e.g., `cd`)
- A shell function (miniature shell scripts in the environment)
- An alias (user-defined commands built from other commands)

Identifying Commands

`type` - Display a Command's Type The `type` command reveals the kind of command the shell will execute.

Example: `type`

```
[me@linuxbox ~]$ type type
type is a shell builtin

[me@linuxbox ~]$ type ls
ls is aliased to 'ls --color=tty'

[me@linuxbox ~]$ type cp
cp is /bin/cp

```

`which` - Display an Executable's Location The `which` command shows the location of an executable program.

Example: `which`

```
[me@linuxbox ~]$ which ls
/bin/ls

```

Getting a Command's Documentation

`help` - Get Help for Shell Builtins The `help` command provides information about shell builtins.

Example: help

```
[me@linuxbox ~]$ help cd

```

`-help` - Display Usage Information Many executables support the `-help` option to display usage information.

Example: -help

```
[me@linuxbox ~]$ mkdir --help

```

`man` - Display a Program's Manual Page The `man` command displays a program's manual page.

Example: man

```
[me@linuxbox ~]$ man ls

```

Man pages are organized into sections:

- 1: User commands
- 2: Programming interfaces for kernel system calls
- 3: Programming interfaces to the C library
- 4: Special files (e.g., device nodes, drivers)
- 5: File formats
- 6: Games and amusements
- 7: Miscellaneous
- 8: System administration commands

`apropos` - Display Appropriate Commands The `apropos` command searches man pages based on a search term.

Example: apropos

```
[me@linuxbox ~]$ apropos partition

```

`whatis` - Display One-line Manual Page Descriptions The `whatis` command displays a brief description of a command.

Example: whatis

```
[me@linuxbox ~]$ whatis ls
ls (1)  - list directory contents

```

`info` - Display a Program's Info Entry The `info` command displays a program's info page, an alternative to man pages provided by the GNU Project.

Example: info

```
[me@linuxbox ~]$ info coreutils

```

README and Other Program Documentation Files Many software packages have documentation in the `/usr/share/doc` directory. These can often be viewed with `less` or a web browser.

Creating Our Own Commands with `alias`

The `alias` command allows users to create their own commands.

Example: `alias`

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'

```

To remove an alias, use the `unalias` command.

Example: `unalias`

```
[me@linuxbox ~]$ unalias foo

```

Summing Up

This chapter covered various commands for identifying, using, and documenting other commands, as well as creating custom commands with `alias`. Practicing these commands will enhance understanding and efficiency in using the Linux command line.

Further Reading

- Bash Reference Manual
- Bash FAQ
- GNU Project Documentation
- Wikipedia: Man Page

The next chapter that is covered this week is **Chapter 6: Redirection**

Chapter 6: Redirection

Overview

This chapter introduces I/O redirection, a powerful feature of the command line that allows the redirection of input and output of commands to and from files, as well as connecting multiple commands into pipelines. The key commands covered include `cat`, `sort`, `uniq`, `grep`, `wc`, `head`, `tail`, and `tee`.

Standard Input, Output, and Error

Unix programs often produce two types of output: program results (standard output or `stdout`) and status/error messages (standard error or `stderr`). By default, both are displayed on the screen, and input is taken from the keyboard (standard input or `stdin`).

Redirecting Standard Output

Standard output can be redirected to a file using the `>` operator.

Example: Redirecting Standard Output

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

To append output to a file instead of overwriting, use the `>>` operator.

Example: Appending Output

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Redirecting Standard Error

Standard error can be redirected using the file descriptor `2`.

Example: Redirecting Standard Error

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

Redirecting Both Standard Output and Standard Error

Both outputs can be redirected to a single file.

Example: Redirecting Both Outputs

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Modern `bash` allows a more concise way.

Example: Concise Redirection

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

Suppressing Output

Unwanted output can be redirected to `/dev/null`.

Example: Suppressing Error Messages

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

Redirecting Standard Input

Commands can take input from a file instead of the keyboard using the `<` operator.

Example: Redirecting Standard Input

```
[me@linuxbox ~]$ cat < lazy_dog.txt
```

Pipelines

Pipelines use the | operator to send the output of one command as input to another.

Example: Pipeline

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

Filters

Filters process data in a pipeline. Examples include `sort`, `uniq`, `wc`, `grep`, `head`, `tail`, and `tee`.

Examples: Filters

- `sort`: Sorts lines of text.
- `uniq`: Removes duplicate lines.
- `wc`: Counts lines, words, and bytes.
- `grep`: Finds lines matching a pattern.
- `head`: Outputs the first part of a file.
- `tail`: Outputs the last part of a file.
- `tee`: Reads from stdin and writes to stdout and files.

Summing Up

This chapter covered the basics of I/O redirection and introduced commands to manipulate input and output effectively. Understanding these concepts is crucial for efficient command line usage.

Further Reading

- Bash Reference Manual
- Wikipedia: /dev/null

The next chapter that is covered this week is

Chapter 11: The Environment

Overview

This chapter explores the shell environment, which stores information used by programs to determine system configuration. We will learn how to customize our shell experience using environment variables, shell variables, and configuration scripts. Key commands include `printenv`, `set`, `export`, and `alias`.

What is Stored in the Environment?

The environment consists of:

- **Environment variables**: Used by programs to adjust their behavior.
- **Shell variables**: Data specific to the shell session.
- **Aliases and shell functions**: Custom commands and functions.

Examining the Environment

`printenv` - Print Environment Variables The `printenv` command displays environment variables.

Example: `printenv`

```
[me@linuxbox ~]$ printenv | less

```

`set` - Display Shell and Environment Variables The `set` command shows both shell and environment variables.

Example: `set`

```
[me@linuxbox ~]$ set | less

```

`echo` - Display Variable Contents The `echo` command can be used to view the contents of a variable.

Example: `echo`

```
[me@linuxbox ~]$ echo \$HOME
/home/me

```

`alias` - Display Aliases The `alias` command shows all defined aliases.

Example: `alias`

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'

```

Some Interesting Variables

Common environment variables include:

- `DISPLAY`: Name of the display in graphical environments.
- `EDITOR`: Preferred text editor.
- `SHELL`: User's default shell program.
- `HOME`: Path to the home directory.
- `LANG`: Character set and collation order.
- `OLDPWD`: Previous working directory.
- `PAGER`: Program used for paging output.
- `PATH`: Directories to search for executable programs.
- `PS1`: Prompt string definition.
- `PWD`: Current working directory.
- `TERM`: Terminal type.
- `TZ`: Time zone.
- `USER`: Username.

How is the Environment Established?

The shell reads startup files to establish the environment. The sequence depends on whether the shell session is a login or non-login session.

Login Shell Sessions Login shells read the following files in order:

- `/etc/profile`
- `~/.bash_profile`
- `~/.bash_login`
- `~/.profile`

Non-Login Shell Sessions Non-login shells read the following files:

- `/etc/bash.bashrc`
- `~/.bashrc`

Modifying the Environment

Modifications to the environment can be made by editing the appropriate startup files. Changes to `PATH` or additional environment variables should be placed in `.bash_profile`. Other customizations should be placed in `.bashrc`.

Using a Text Editor Text editors like `nano`, `vi`, and `gedit` can be used to edit startup files. It is recommended to create a backup before making changes.

Example: Editing `.bashrc`

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
[me@linuxbox ~]$ nano .bashrc

```

Example Modifications

Adding the following lines to `.bashrc` can customize the environment:

Example: Modifications

```
# Change umask to make directory sharing easier
umask 0002

# Ignore duplicates in command history and increase history size to 1000 lines
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Add some helpful aliases
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'

```

Activating Changes

Changes can be activated by restarting the terminal session or by sourcing the `.bashrc` file.

Example: Sourcing `.bashrc`

```
[me@linuxbox ~]$ source \~{}/.bashrc

```

Summing Up

This chapter covered the basics of the shell environment and how to customize it using environment variables, shell variables, and aliases. It also introduced editing configuration files using text editors.

Further Reading

- Bash Reference Manual: INVOCATION section

The last chapter that is going to be covered this week is

Chapter 12: A Gentle Introduction to vi

Overview

This chapter introduces the `vi` text editor, a core program in the Unix tradition known for its efficiency and minimalism. Learning `vi` is essential due to its ubiquity, speed, and compliance with POSIX standards. This chapter covers basic usage and commands to help users become familiar with `vi`.

Why We Should Learn `vi`

- `vi` is almost always available, making it crucial for systems without a graphical interface or with a broken X configuration.
- `vi` is lightweight and fast, designed for speed and efficiency without needing to leave the keyboard.
- Mastery of `vi` demonstrates proficiency and confidence in using Unix-based systems.

Starting and Stopping `vi`

To start `vi`, enter:

Example: Starting `vi`

```
[me@linuxbox ~]$ vi

```

To exit `vi`, use:

Example: Exiting `vi`

```
:q

```

If changes are unsaved, force quit with:

Example: Forcing Quit

```
:q!

```

Editing Modes

`vi` operates in different modes. Command mode is the default, where most keys are commands. To enter insert mode, press:

Example: Insert Mode

```
i

```

Press `Esc` to return to command mode.

Saving Work

To save changes, use:

Example: Saving Work

```
:w
```

Basic Editing

Moving the Cursor Movement commands include:

- `l` or right arrow: Right one character.
- `h` or left arrow: Left one character.
- `j` or down arrow: Down one line.
- `k` or up arrow: Up one line.
- `0` (zero): To the beginning of the current line.
- `$`: To the end of the current line.
- `w`: To the beginning of the next word.
- `b`: To the beginning of the previous word.
- `Ctrl-f` or Page Down: Down one page.
- `Ctrl-b` or Page Up: Up one page.
- `numberG`: To line number.
- `G`: To the last line of the file.

Inserting and Appending Text

- `i`: Insert before the cursor.
- `a`: Append after the cursor.
- `A`: Append at the end of the line.
- `o`: Open a new line below the current line.
- `O`: Open a new line above the current line.

Deleting Text

- `x`: Delete the character under the cursor.
- `dd`: Delete the current line.
- `d$`: Delete to the end of the line.
- `d0`: Delete to the beginning of the line.

Copying and Pasting Text

- `yy`: Yank (copy) the current line.
- `p`: Paste after the cursor.
- `P`: Paste before the cursor.

Search-and-Replace

Searching Use `/` followed by the search term to search the file. Repeat search with `n`.

Example: Search

```
/search_term

```

Replacing Use `:s` for search and replace. For example, to replace "foo" with "bar" globally:

Example: Search-and-Replace

```
:%s/foo/bar/g

```

Editing Multiple Files

Open multiple files by listing them in the command:

Example: Multiple Files

```
vi file1 file2

```

Switch between files with:

Example: Switch Files

```
:bn
:bp

```

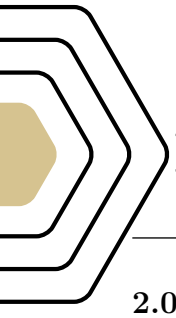
Summing Up

This chapter covered the basics of `vi`, including starting, stopping, editing modes, basic commands, search-and-replace, and working with multiple files. Mastering `vi` requires practice, but it offers powerful capabilities for efficient text editing.

Further Reading

- Vim, with Vigor
- Learning The vi Editor - Wikibook
- The Vim Book
- Bill Joy - Wikipedia
- Bram Moolenaar - Wikipedia

Regular Expressions And Scripting



Regular Expressions And Scripting

2.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- **The Linux Command Line - Chapter 7 - Seeing The World As The Shell Sees It**
- **The Linux Command Line - Chapter 9 - Permissions**
- **The Linux Command Line - Chapter 13 - Customizing The Prompt**
- **The Linux Command Line - Chapter 19 - Regular Expressions**
- **The Linux Command Line - Chapter 24 - Writing Your First Script**
- **The Linux Command Line - Chapter 26 - Top Down Design**
- **The Linux Command Line - Chapter 27 - Flow Control - Branching With if**
- **The Linux Command Line - Chapter 32 - Positional Parameters**

2.0.2 Lectures

The lectures for this week are:

- Definition Of Regular Expressions $\approx$ 25 **min**.
- Development And Evaluation Of Regular Expressions $\approx$ 12 **min**.
- Bash Scripting Using The Shell $\approx$ 20 **min**.
- Python Scripting $\approx$ 21 **min**.
- Lab 2 Introduction $\approx$ 40 **min**.

The lecture notes for this week are:

- Regex Golf
- Bash Scripting Cheatsheet

2.0.3 Assignments

The assignment(s) for this week are:

- Lab 2 - Using Command Line Interface

2.0.4 Quiz

The quiz for this week is:

- Quiz 2 - Regular Expressions And Scripting

2.0.5 Project

The assignment(s) for the project this week is:

- Introduction To The Team Project - Guidelines
- Project Milestone 1: Project Proposal

2.0.6 Chapter Summary

The first chapter that is covered this week is **Chapter 7: Seeing The World As The Shell Sees It**.

Chapter 7: Seeing The World As The Shell Sees It

Overview

This chapter explores how the shell processes commands by performing expansions before execution. We will cover various types of expansions including pathname, tilde, arithmetic, brace, parameter, and command substitution. Understanding these expansions is crucial for effective use of the shell.

Expansion

Expansion refers to the shell's process of substituting text in a command before execution. For example, the `echo` command prints its arguments to standard output, but special characters like `*` can be expanded to match filenames.

Expansion

Expansions transform simple text into more complex expressions before the shell executes commands.

- The `echo` command can display text or expanded results.
- The `*` wildcard is expanded to match filenames in the current directory.

Pathname Expansion

Pathname expansion, also known as globbing, uses wildcards to match filenames. Characters like `*`, `?`, and `[]` create patterns that match multiple files or directories.

Pathname Expansion

Wildcards expand to match filenames and directories.

- `D*` matches all filenames starting with "D".
- `*s` matches all filenames ending with "s".
- Patterns can include character classes like `[[:upper:]]*` to match uppercase filenames.

Tilde Expansion

The tilde `~` expands to the home directory of the current user or another specified user.

Arithmetic Expansion

Arithmetic expansion allows the shell to perform calculations within commands using the syntax `$((expression))`.

Arithmetic Expansion

Perform calculations within commands using arithmetic expansion.

- `$((2 + 2))` evaluates to 4.
- Supports operations like addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), modulo (`%`), and exponentiation (`**`).

Brace Expansion

Brace expansion generates multiple text strings from a pattern containing braces `{}`, which can include comma-separated lists or ranges of characters or integers.

Brace Expansion

Create multiple text strings from a single pattern.

- `Front-{A,B,C}-Back` expands to `Front-A-Back`, `Front-B-Back`, `Front-C-Back`.
- `{1..5}` expands to `1 2 3 4 5`.
- Patterns can be nested for more complex expansions.

Parameter Expansion

Parameter expansion retrieves and manipulates the values of variables using the syntax `${parameter}`.

Parameter Expansion

Retrieve and manipulate variable values.

- `$USER` expands to the current user's name.
- `$HOME` expands to the current user's home directory.

Command Substitution

Command substitution captures the output of a command and uses it as an argument in another command, using `$(command)` or backticks `` `command` ``.

Command Substitution

Use the output of one command as an argument in another.

- `echo $(ls)` prints the output of the `ls` command.
- Backticks (`` `command` ``) can also be used for command substitution.

Quoting

Quoting controls how the shell interprets special characters. Double quotes `"` suppress most expansions except for parameter, arithmetic, and command substitution. Single quotes `'` suppress all expansions. The backslash `\` escapes a single character.

Quoting

Control the shell's interpretation of special characters.

- Double quotes (`"`) suppress most expansions.
- Single quotes (`'`) suppress all expansions.
- The backslash (`\`) escapes individual characters.

Escaping Characters

The backslash `\` is used to escape a single character, preventing its special meaning.

Escaping Characters

Escape individual characters to prevent their special meaning.

- `echo "The balance for user $USER is: $5.00"` prints `The balance for user $USER is:`

`$5.00`.

- Use double backslashes (\\) to include a literal backslash.

Summary of Key Concepts

The chapter covers essential shell features such as expansions and quoting, which are fundamental for efficient command-line usage and script writing.

Summary of Key Concepts

Understand how the shell interprets and expands commands.

- **Expansion**: Transforms simple text into more complex expressions.
- **Pathname Expansion**: Uses wildcards to match filenames.
- **Tilde Expansion**: Expands to the user's home directory.
- **Arithmetic Expansion**: Performs calculations within commands.
- **Brace Expansion**: Generates multiple text strings from a pattern.
- **Parameter Expansion**: Retrieves and manipulates variable values.
- **Command Substitution**: Uses the output of one command in another.
- **Quoting**: Controls how the shell interprets special characters.

These concepts are crucial for mastering the shell and utilizing its full potential.

The next chapter that is covered this week is **Chapter 9: Permissions**.

Chapter 9: Permissions

Overview

This chapter explores the Unix-like permission system in Linux, which supports multiuser and multitasking environments. It introduces key commands for managing permissions, ownership, and user identity, providing a foundation for understanding system security and access control.

Owners, Group Members, and Everybody Else

In Unix-like systems, each file and directory has an owner and a group. Permissions are assigned to three categories: the owner, the group, and others (everyone else). The `id` command can be used to display user identity information.

Owners, Group Members, and Everybody Else

The Unix permission model assigns permissions to the owner, group, and others.

- `id` command displays user identity, including user ID (uid), group ID (gid), and group memberships.
- Files and directories have an owner and a group, controlling access through permissions.

Reading, Writing, and Executing

Permissions are categorized into read (r), write (w), and execute (x) for files and directories. The `ls -l` command reveals these permissions in the file attributes.

Reading, Writing, and Executing

Permissions control the ability to read, write, and execute files and directories.

- Read (r): Allows reading the contents of a file or listing a directory.
- Write (w): Allows modifying a file or altering the contents of a directory.
- Execute (x): Allows running a file as a program or entering a directory.

chmod - Change File Mode

The `chmod` command is used to change the mode (permissions) of a file or directory. It supports octal and symbolic representations for specifying permissions.

chmod - Change File Mode

Change file or directory permissions using `chmod`.

- Octal notation: Uses numbers (e.g., `chmod 600 file.txt`) to set permissions.
- Symbolic notation: Uses characters (e.g., `chmod u+x file.txt`) to modify specific permissions.

umask - Set Default Permissions

The `umask` command sets the default permissions for newly created files and directories by defining a mask that removes permissions.

umask - Set Default Permissions

Control default permissions for new files and directories using `umask`.

- `umask 0002`: Sets the mask to remove write permissions for others.
- Default umask values vary, commonly `0022` or `0002`.

Changing Identities

The `su` and `sudo` commands allow users to assume the identity of another user, typically the superuser, to perform administrative tasks.

Changing Identities

Assume another user's identity to perform tasks requiring different permissions.

- `su`: Start a new shell as another user, typically the superuser.
- `sudo`: Execute a command as another user, with permissions configured in `/etc/sudoers`.

chown - Change File Owner and Group

The `chown` command changes the owner and group of a file or directory. Superuser privileges are required to use this command.

chown - Change File Owner and Group

Change the ownership of files and directories using `chown`.

- `chown user:group file.txt`: Changes the owner to `user` and the group to `group`.
- Useful for managing access to shared files and directories.

chgrp - Change Group Ownership

The `chgrp` command changes the group ownership of a file or directory. It is similar to `chown`, but specifically for groups.

chgrp - Change Group Ownership

Change the group ownership of files and directories using `chgrp`.

- `chgrp group file.txt`: Changes the group ownership to `group`.
- Simplifies management of group permissions.

passwd - Change a User's Password

The `passwd` command changes a user's password. Superusers can change passwords for other users.

passwd - Change a User's Password

Change user passwords using `passwd`.

- `passwd`: Changes the current user's password.
- `passwd user`: Changes the password for `user` (requires superuser privileges).

Summary of Key Concepts

This chapter covers essential commands and concepts for managing permissions and ownership in Unix-like systems.

- **Owners, Group Members, and Everybody Else**: Understand user and group-based permissions.
- **Reading, Writing, and Executing**: Learn the significance of read, write, and execute permissions.
- **chmod**: Change file and directory permissions.
- **umask**: Set default permissions for new files and directories.
- **Changing Identities**: Use `su` and `sudo` for administrative tasks.
- **chown**: Change file and directory ownership.
- **chgrp**: Change group ownership of files and directories.
- **passwd**: Change user passwords.

These tools and concepts are fundamental for effective system administration and security management in Unix-like environments.

The next chapter that is covered this week is **Chapter 13: Customizing The Prompt**.

Chapter 13: Customizing The Prompt

Overview

This chapter explores customizing the shell prompt in Linux, revealing some of the inner workings of the shell and the terminal emulator program. By learning how to control the prompt, users can enhance their command-line experience.

Anatomy of a Prompt

The default shell prompt is defined by the environment variable `PS1`. This variable contains special backslash-escaped characters that represent various pieces of information.

Anatomy of a Prompt

The shell prompt is configured using the `PS1` variable.

- `PS1` contains special characters, like `$`, `~`, and , which represent the current user, hostname, and more.
- Example: `[me@linuxbox ~]$` is defined by `[\u@\h \W]\$`.

Trying Some Alternative Prompt Designs

Users can experiment with different prompt designs by modifying the `PS1` variable.

Alternative Prompt Designs

Change the prompt by modifying `PS1`.

- `PS1="$ "` sets a minimal prompt.
- `PS1="$ "` includes the time and hostname.
- `PS1="<\u@\h>$ "` mimics the default prompt with angle brackets.

Adding Color

ANSI escape codes can be used to add color to the shell prompt.

Adding Color

Use ANSI escape codes to color the prompt.

- `\033[0;31m` sets text color to red.
- `\033[0m` resets text color to default.
- Example: `PS1="\[\033[0;31m\]<\u@\h>$\[\033[0m\] "`

Moving the Cursor

Escape codes can also control cursor position, enabling more complex prompt designs.

Moving the Cursor

Position the cursor using escape codes.

- `\033[l;cH` moves the cursor to line `l` and column `c`.
- `\033[2J` clears the screen.
- Example: `PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\033[0m\033[u\]<\u@\h>$ "`

Saving the Prompt

To make the prompt permanent, add the configuration to the `.bashrc` file.

Saving the Prompt

Persist the prompt configuration by adding it to `.bashrc`.

- Add the `PS1` assignment to `.bashrc`.
- Example: `PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\033[0m\033[u\]<\u@\h>$ "`
- `export PS1`

Summary of Key Concepts

This chapter covers how to customize the shell prompt for a more personalized and informative command-line experience.

- **Anatomy of a Prompt**: Understand the components of the `PS1` variable.
- **Alternative Prompt Designs**: Experiment with different prompt styles.
- **Adding Color**: Use ANSI escape codes to add colors to the prompt.
- **Moving the Cursor**: Control cursor positioning with escape codes.
- **Saving the Prompt**: Make prompt changes permanent by updating `.bashrc`.

Customizing the prompt enhances the usability and aesthetics of the command-line interface.

The next chapter that is covered this week is **Chapter 19: Regular Expressions**.

Chapter 19: Regular Expressions

Overview

This chapter delves into regular expressions, a powerful text manipulation technology essential for sophisticated uses of various command-line tools in Unix-like systems. Regular expressions are symbolic notations used to identify patterns in text, similar to shell wildcards but more advanced.

What are Regular Expressions?

Regular expressions (regex) are sequences of characters defining search patterns. They are supported by many command-line tools and programming languages for text manipulation.

What are Regular Expressions?

Regular expressions identify patterns in text.

- Regex can match literal text and complex patterns.
- Supported by command-line tools like `grep` and many programming languages.

grep

The `grep` command searches text files for lines matching a specified regex and prints the matching lines.

grep

`grep` uses regular expressions to search text files.

- `grep [options] regex [file...]` syntax.
- Options include `-i` (ignore case), `-v` (invert match), `-c` (count matches), and `-n` (show line numbers).

Metacharacters and Literals

Regex consists of literals (ordinary characters) and metacharacters (special characters with specific meanings).

Metacharacters and Literals

Metacharacters and literals form the basis of regex patterns.

- Metacharacters: `^`, `$`, `.`, `[ ]`, `{ }`, `-`, `?`, `*`, `+`, `( )`, `|`, `\`.

- Literals match themselves, while metacharacters have special functions.

The Any Character

The dot (`.`) metacharacter matches any single character.

The Any Character

The dot (`.`) matches any single character.

- Example: `grep -h '.zip' dirlist*.txt` matches any character followed by "zip".

Anchors

Anchors like `^` and `$` match the start and end of a line, respectively.

Bracket Expressions and Character Classes

Bracket expressions match any single character from a set. Character classes provide predefined sets of characters.

Bracket Expressions and Character Classes

Bracket expressions and character classes match specific sets of characters.

- Example: `grep -h '[bg]zip' dirlist*.txt` matches "bzip" or "gzip".
- POSIX character classes: `[:alnum:]`, `[:alpha:]`, `[:digit:]`, etc.

Negation

Negation in bracket expressions is specified by placing a caret (`^`) at the beginning.

Negation

Negate a set of characters using `^` in a bracket expression.

- Example: `grep -h '[^bg]zip' dirlist*.txt` matches "zip" preceded by any character except "b" or "g".

Traditional Character Ranges

Character ranges can be specified within brackets to match characters in a specific range.

Traditional Character Ranges

Specify ranges of characters within brackets.

- Example: `grep -h '^[A-Z]' dirlist*.txt` matches lines starting with an uppercase letter.

POSIX Character Classes

POSIX character classes provide predefined sets of characters, useful for matching ranges like alphanumeric characters.

POSIX Character Classes

POSIX character classes match predefined sets of characters.

- Examples: `[:alnum:]`, `[:alpha:]`, `[:digit:]`, `[:upper:]`, `[:lower:]`.

POSIX Basic vs. Extended Regular Expressions

POSIX defines two types of regex: Basic (BRE) and Extended (ERE). ERE includes additional metacharacters.

POSIX Basic vs. Extended Regular Expressions

Basic and extended regex have different sets of metacharacters.

- BRE metacharacters: `^`, `$`, `.`, `[ ]`, `*`.
- ERE adds: `( )`, `{ }`, `?`, `+`, `|`.
- Use `grep -E` or `egrep` for ERE.

Alternation

Alternation, signified by the vertical bar (`|`), allows matching from multiple expressions.

Alternation

Use the vertical bar (`|`) to match multiple patterns.

- Example: `grep -E 'AAA|BBB'` matches lines containing "AAA" or "BBB".

Quantifiers

Quantifiers specify the number of times an element is matched.

Quantifiers

Quantifiers control the number of matches.

- `?`: Matches zero or one time.
- `*`: Matches zero or more times.
- `+`: Matches one or more times.
- `{n}`: Matches exactly `n` times.
- `{n,m}`: Matches between `n` and `m` times.
- Example: `[0-9]{3}` matches exactly three digits.

Summary of Key Concepts

Regular expressions provide powerful tools for text manipulation and pattern matching, essential for efficient command-line usage.

Summary of Key Concepts

Key concepts of regular expressions:

- **What are Regular Expressions?**: Patterns to identify text.
- **grep**: Command to search text using regex.
- **Metacharacters and Literals**: Building blocks of regex.
- **The Any Character**: Dot (`.`) matches any character.
- **Anchors**: `^` and `$` match start and end of lines.
- **Bracket Expressions and Character Classes**: Match sets of characters.

- **Negation**: Exclude characters in a set.
- **Traditional Character Ranges**: Specify character ranges.
- **POSIX Character Classes**: Predefined character sets.
- **POSIX Basic vs. Extended Regular Expressions**: Different sets of metacharacters.
- **Alternation**: Match from multiple patterns.
- **Quantifiers**: Control the number of matches.

Understanding regular expressions enhances text manipulation capabilities on the command line.

The next chapter that is covered this week is **Chapter 24: Writing Your First Script**.

Chapter 24: Writing Your First Script

Overview

This chapter introduces shell scripting in Linux, allowing users to automate complex sequences of tasks. By writing shell scripts, we can make the shell perform repetitive tasks efficiently.

What are Shell Scripts?

A shell script is a file containing a series of commands. The shell reads this file and executes the commands as if they were entered directly on the command line.

What are Shell Scripts?

Shell scripts automate the execution of a series of commands.

- Shell scripts are text files containing shell commands.
- They enable the automation of tasks by grouping commands together.

How to Write a Shell Script

To create and run a shell script, follow these steps: write the script, make it executable, and place it in a directory included in the PATH.

How to Write a Shell Script

Steps to create and run a shell script:

- **Write a script**: Use a text editor to create a script file.
- **Make the script executable**: Change the file permissions to allow execution.
- **Put the script in the PATH**: Place the script in a directory included in the PATH.

Script File Format

The basic format of a script includes the shebang line, comments, and commands. For example, a "Hello World" script:

Script File Format

Basic elements of a shell script:

- Shebang line: `#!/bin/bash` indicates the script interpreter.

- Comments: Lines starting with `#` are ignored by the shell.
- Commands: The actual commands to be executed.
- Example script:

```
#!/bin/bash
# This is our first script.
echo 'Hello World!'

```

Executable Permissions

Scripts need executable permissions to run. Use the `chmod` command to set these permissions.

Executable Permissions

Set the script to be executable using `chmod`.

- Example: `chmod 755 hello_world`
- `755` for scripts everyone can execute; `700` for scripts only the owner can execute.

Script File Location

Place scripts in a directory included in the PATH to execute them without specifying the path. Common locations include `/bin` for personal scripts and `/usr/local/bin` for system-wide scripts.

Script File Location

Ensure the script is in a directory included in the PATH.

- `/bin` for personal scripts.
- `/usr/local/bin` for system-wide scripts.
- Add `/bin` to the PATH if not already included:

```
export PATH=~/bin:"$PATH"

```

More Formatting Tricks

Formatting techniques like long option names, indentation, and line continuation enhance the readability and maintainability of scripts.

More Formatting Tricks

Improve script readability and maintainability.

- **Long option names**: Use for clarity.
- **Indentation and line continuation**: Use backslashes for line continuation and indent for readability.
- Example:

```
find playground \
    \( \
    -type f \
    -not -perm 0600 \
    -exec chmod 0600 '{}' ';' \
    \) \
    -or \
    \( \
    -type d \
    -not -perm 0700 \
    -exec chmod 0700 '{}' ';' \
    \)

```

Configuring vim for Script Writing

The `vim` text editor can be configured to assist in script writing with syntax highlighting, auto-indentation, and other features.

Configuring vim for Script Writing

Configure `vim` for better script writing experience.

- `:syntax on`: Enables syntax highlighting.
- `:set hlsearch`: Highlights search results.
- `:set tabstop=4`: Sets tab width to 4 spaces.
- `:set autoindent`: Enables automatic indentation.

Summary of Key Concepts

Key concepts for writing and executing shell scripts:

- **What are Shell Scripts?**: Automate tasks with scripts.
- **How to Write a Shell Script**: Write, make executable, and place in PATH.
- **Script File Format**: Use shebang, comments, and commands.
- **Executable Permissions**: Set executable permissions.
- **Script File Location**: Place scripts in PATH directories.
- **More Formatting Tricks**: Enhance readability with formatting.
- **Configuring vim for Script Writing**: Use `vim` settings for better scripting.

These techniques ensure that scripts are effective, readable, and maintainable.

The next chapter that is covered this week is **Chapter 26: Top Down Design**.

Chapter 26: Top Down Design

Overview

This chapter introduces top-down design, a method of breaking down large, complex tasks into smaller, manageable ones. This approach is particularly useful for shell programming, enabling the creation of maintainable and understandable scripts.

Top-Down Design

Top-down design involves identifying the top-level steps of a task and developing increasingly detailed steps for each. This method helps in organizing complex processes into simpler, executable tasks.

Top-Down Design

Top-down design breaks large tasks into smaller, detailed steps.

- Example: Describing a process like going to the market can be broken down into steps such as driving to the market, parking the car, and purchasing food.
- Each step can be further detailed to ensure clarity and completeness.

Shell Functions

Shell functions are "mini-scripts" within a script, acting as autonomous programs. They help in organizing code and reusing common tasks.

Shell Functions

Shell functions encapsulate a series of commands to perform specific tasks.

- Defined using the syntax:

```
function_name() {
    commands
}

```

- Functions can be called within the script to execute the encapsulated commands.

Implementing Shell Functions

Implementing shell functions in a script involves defining the function and calling it within the main program. This helps in structuring the script logically.

Implementing Shell Functions

Define and use shell functions to organize script logic.

- Example script:

```
#!/bin/bash

report_uptime() {
    echo "Function report_uptime executed."
}

report_disk_space() {
    echo "Function report_disk_space executed."
}

report_home_space() {
    echo "Function report_home_space executed."
}

report_uptime
report_disk_space
report_home_space

```

Local Variables

Local variables are defined within functions and only accessible within the scope of those functions. This prevents conflicts with global variables.

Local Variables

Use local variables to avoid conflicts and ensure function independence.

- Example:

```
#!/bin/bash

foo=0  # Global variable

function1() {
    local foo=1  # Local variable
    echo "function1: foo = $foo"
}

function2() {
    local foo=2  # Local variable
    echo "function2: foo = $foo"
}

echo "global: foo = $foo"
function1
echo "global: foo = $foo"
function2
echo "global: foo = $foo"

```

Keeping Scripts Running

To ensure scripts remain functional during development, it is useful to frequently test and keep them in a runnable state. This helps in identifying issues early.

Keeping Scripts Running

Regular testing and maintaining runnable scripts help in early error detection.

- Use stubs (minimal function definitions) to verify script logic.
- Example:

```
report_uptime() {
    echo "Function report_uptime executed."
}

report_disk_space() {
    echo "Function report_disk_space executed."
}

report_home_space() {
    echo "Function report_home_space executed."
}

```

Fleshing Out Functions

With the function framework in place, we can add the actual commands to each function to perform the desired tasks.

Fleshing Out Functions

Add the actual commands to each function to complete the tasks.

- Example: The function to report system uptime.

```
report_uptime() {
    cat <<- _EOF_
        <h2>System Uptime</h2>
        <pre>$(uptime)</pre>
    _EOF_
}

```

Summary of Key Concepts

Top-down design and shell functions are crucial for writing organized, maintainable scripts. They enable the breakdown of complex tasks and promote code reuse.

Summary of Key Concepts

Key concepts of top-down design and shell functions:

- **Top-Down Design**: Break down tasks into manageable steps.
- **Shell Functions**: Encapsulate commands for reuse and organization.
- **Local Variables**: Use within functions to avoid conflicts.
- **Keeping Scripts Running**: Regular testing and maintaining runnable states.
- **Fleshing Out Functions**: Complete functions with actual commands.

These practices ensure that scripts are effective, readable, and maintainable.

The next chapter that is covered this week is **Chapter 27: Flow Control - Branching With if**.

Chapter 27: Flow Control - Branching With if

Overview

This chapter introduces flow control in shell scripting, focusing on the ‘if‘ statement. Flow control allows scripts to make decisions and execute different commands based on conditions. This enables scripts to adapt to different situations, such as varying user privileges.

if Statement

The ‘if‘ statement allows branching based on the success or failure of commands. It evaluates a condition and executes commands based on whether the condition is true or false.

if Statement

The ‘if‘ statement evaluates conditions to control the flow of the script.

- Syntax:

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi

```

- Executes the commands in the ‘then‘ block if the condition is true; otherwise, it executes the commands in the ‘else‘ block.

Exit Status

Commands return an exit status upon completion. An exit status of 0 indicates success, while any other value indicates failure. The special variable ‘$?’ holds the exit status of the last executed command.

Exit Status

Commands return an exit status to indicate success or failure.

- Exit status 0: Success.
- Exit status non-zero: Failure.
- Example:

```
ls -d /usr/bin
echo $?
# Output: 0

ls -d /bin/usr
echo $?
# Output: 2

```

test Command

The ‘test‘ command performs various checks and comparisons. It can be used with the ‘if‘ statement to evaluate conditions.

test Command

Use the ‘test‘ command to perform checks and comparisons.

- Two forms:

```
test expression
[ expression ]

```

- Returns an exit status of 0 if the expression is true, and 1 if the expression is false.

File Expressions

File expressions in the ‘test‘ command evaluate the status of files.

File Expressions

Evaluate file status using ‘test‘ expressions.

- Examples:
  - ‘-e file‘: File exists.
  - ‘-f file‘: File is a regular file.
  - ‘-d file‘: File is a directory.

String Expressions

String expressions compare and evaluate strings.

String Expressions

Evaluate string values and comparisons.

- Examples:
  - ‘-z string‘: String is empty.
  - ‘string1 = string2‘: Strings are equal.
  - ‘string1 != string2‘: Strings are not equal.

Integer Expressions

Integer expressions compare integer values.

Integer Expressions

Compare integer values using ‘test‘.

- Examples:
  - ‘integer1 -eq integer2‘: Integers are equal.
  - ‘integer1 -ne integer2‘: Integers are not equal.
  - ‘integer1 -lt integer2‘: Integer1 is less than integer2.

Combining Expressions

Expressions can be combined using logical operators to create more complex conditions.

Combining Expressions

Combine expressions using logical operators.

- Logical operators:
  - ‘-a‘ (AND)
  - ‘-o‘ (OR)
  - ‘¡ (NOT)
- Example:

```
if [ "$INT" -ge 1 -a "$INT" -le 100 ]; then
    echo "$INT is within range."
else
    echo "$INT is out of range."
fi

```

Modern Alternatives to test

The ‘[[...]]‘ syntax offers an enhanced replacement for ‘test‘, supporting additional features such as regex matching and improved readability.

Modern Alternatives to test

Use ‘[[...]]‘ for enhanced functionality.

- Syntax:

```
[[ expression ]]

```

- Supports regex matching:

```
[[ string =~ regex ]]

```

- Example:

```
if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    echo "$INT is a valid integer."
fi

```

Arithmetic Evaluation with ((...))

The ‘((...))‘ syntax performs arithmetic evaluations and returns a success status for non-zero results.

Arithmetic Evaluation with ((...))

Perform arithmetic evaluations using ‘((...))‘.

- Syntax:

```
(( expression ))

```

- Example:

```
if (( INT < 0 )); then
    echo "INT is negative."
fi

```

Control Operators

Control operators ‘‘ and ‘||‘ provide additional methods for branching by executing commands based on the success or failure of preceding commands.

Control Operators

Use control operators for branching.

- ‘command1 command2‘: Executes ‘command2‘ if ‘command1‘ is successful.
- ‘command1 || command2‘: Executes ‘command2‘ if ‘command1‘ fails.
- Example:

```
mkdir temp && cd temp

```

Summary of Key Concepts

Key concepts for flow control using the ‘if‘ statement:

- **if Statement**: Evaluate conditions to control script flow.

- **Exit Status**: Determine success or failure of commands.
- **test Command**: Perform checks and comparisons.
- **File Expressions**: Evaluate file status.
- **String Expressions**: Evaluate string values.
- **Integer Expressions**: Compare integer values.
- **Combining Expressions**: Create complex conditions with logical operators.
- **Modern Alternatives to test**: Use '[[...]]' for enhanced functionality.
- **Arithmetic Evaluation with ((...))**: Perform arithmetic evaluations.
- **Control Operators**: Use '' and '||' for branching.

Understanding these concepts allows for more dynamic and adaptable scripts.

The last chapter that is covered this week is **Chapter 32: Positional Parameters**.

Chapter 32: Positional Parameters

Overview

This chapter explores positional parameters in shell scripting, which allow scripts to accept and process command-line options and arguments. This feature enables scripts to become more flexible and interactive.

Accessing the Command Line

The shell provides a set of variables called positional parameters that contain the individual words on the command line. These variables are named `0` through `9`.

Accessing the Command Line

Positional parameters store command-line arguments.

- `$0` contains the name of the script.
- `$1, $2, ..., $9` contain the first, second, ..., ninth arguments.
- Example script:

```
#!/bin/bash
# posit-param: script to view command line parameters
echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"

```

Determining the Number of Arguments

The special variable `$` contains the number of arguments provided on the command line.

Determining the Number of Arguments

Use `$` to determine the number of command-line arguments.

- Example:

```
#!/bin/bash
# posit-param: script to view command line parameters
echo "
Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"

```

shift – Getting Access to Many Arguments

The `shift` command shifts positional parameters to the left, allowing access to arguments beyond the ninth.

shift – Getting Access to Many Arguments

Use `shift` to access more than nine command-line arguments.

- Example script:

```
#!/bin/bash
# posit-param2: script to display all arguments
count=1
while [[ $# -gt 0 ]]; do
    echo "Argument $count = $1"
    count=$((count + 1))
    shift
done

```

Simple Applications

Positional parameters can be used to create useful applications without needing `shift`.

Simple Applications

Create simple applications using positional parameters.

- Example script:

```
#!/bin/bash
# file-info: simple file information program
PROGNAME="$(basename "$0")"
if [[ -e "$1" ]]; then
    echo -e "\nFile Type:"
    file "$1"
    echo -e "\nFile Status:"
    stat "$1"
else
    echo "$PROGNAME: usage: $PROGNAME file" >&2
    exit 1
fi

```

Using Positional Parameters with Shell Functions

Positional parameters can be passed to shell functions in the same way as scripts.

Using Positional Parameters with Shell Functions

Pass arguments to shell functions using positional parameters.

- Example function:

```
file_info () {
    if [[ -e "$1" ]]; then
        echo -e "\nFile Type:"
        file "$1"
        echo -e "\nFile Status:"
        stat "$1"
    else
        echo "$FUNCNAME: usage: $FUNCNAME file" >&2
        return 1
    fi
}

```

Handling Positional Parameters en Masse

The special parameters `$*` and `$@` expand into all positional parameters, but they differ in behavior when quoted.

Handling Positional Parameters en Masse

Use `$*` and `$@` to handle all positional parameters.

- `$*`: Expands into all positional parameters as a single word.
- `$@`: Expands each positional parameter into a separate word.
- Example script:

```
#!/bin/bash
# posit-params3: script to demonstrate $* and $@
print_params () {
    echo "\$1 = $1"
    echo "\$2 = $2"
    echo "\$3 = $3"
    echo "\$4 = $4"
}
pass_params () {
    echo -e "\n\$* :";   print_params $*
    echo -e "\n\"$*\" :"; print_params "$*"
    echo -e "\n\$@ :";   print_params $@
    echo -e "\n\"$@\" :"; print_params "$@"
}
pass_params "word" "words with spaces"

```

A More Complete Application

An example of a more complex application using positional parameters to handle command-line options.

A More Complete Application

Implement a script with command-line options using positional parameters.

- Example script:

```
#!/bin/bash
# sys_info_page: program to output a system information page

PROGNAME="$(basename "$0")"
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    cat <<- _EOF_
    <h2>System Uptime</h2>
    <pre>$(uptime)</pre>
    _EOF_
    return
}

report_disk_space () {
    cat <<- _EOF_
    <h2>Disk Space Utilization</h2>
    <pre>$(df -h)</pre>
    _EOF_
    return
}

report_home_space () {
    if [[ "$(id -u)" -eq 0 ]]; then
        cat <<- _EOF_
```

```
        <h2>Home Space Utilization (All Users)</h2>
        <pre>$(du -sh /home/*)</pre>
        _EOF_
    else
        cat <<- _EOF_
        <h2>Home Space Utilization ($USER)</h2>
        <pre>$(du -sh "$HOME")</pre>
        _EOF_
    fi
    return
}

usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

write_html_page () {
    cat <<- _EOF_
    <html>
    <head>
    <title>$TITLE</title>
    </head>
    <body>
    <h1>$TITLE</h1>
    <p>$TIMESTAMP</p>
    $(report_uptime)
    $(report_disk_space)
    $(report_home_space)
    </body>
    </html>
    _EOF_
    return
}

# process command line options
interactive=
filename=

while [[ -n "$1" ]]; do
    case "$1" in
    -f | --file) shift
        filename="$1"
        ;;
    -i | --interactive) interactive=1
        ;;
    -h | --help) usage
        exit
        ;;
    *) usage >&2
        exit 1
        ;;
    esac
    shift
done

# interactive mode
if [[ -n "$interactive" ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e "$filename" ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case "$REPLY" in
            Y|y) break
                ;;
            Q|q) echo "Program terminated."
                exit
                ;;
            *) continue
                ;;
            esac
        elif [[ -z "$filename" ]]; then
            continue
        else
            break
        fi
    done
fi

# output html page
if [[ -n "$filename" ]]; then
    if touch "$filename" && [[ -f "$filename" ]]; then
        write_html_page > "$filename"
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page
fi

```

Summary of Key Concepts

Key concepts for using positional parameters in shell scripts:

- **Accessing the Command Line**: Use positional parameters to access command-line arguments.
- **Determining the Number of Arguments**: Use `$` to count arguments.
- **shift**: Access arguments beyond the ninth using `shift`.
- **Simple Applications**: Create useful scripts with positional parameters.
- **Using Positional Parameters with Shell Functions**: Pass arguments to shell functions.
- **Handling Positional Parameters en Masse**: Use `$*` and `$@` to handle all arguments.
- **A More Complete Application**: Implement a script with command-line options.

Understanding positional parameters enhances script flexibility and interactivity.

Agile Devlopment

Agile Development

3.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- **The Linux Command Line - Chapter 29 - Flow Control - Looping With while & until**
- **The Linux Command Line - Chapter 31 - Flow Control - Branching With case**
- **Agile For Dummies - Chapter 1 - Getting The ABCs Of Agile**
- **Agile For Dummies - Chapter 2 - Understanding Agile Roles**
- **Agile For Dummies - Chapter 3 - Getting Started With Agile**
- **Agile For Dummies - Chapter 4 - Choosing An Agile Approach**
- **Agile For Dummies - Chapter 10 - Ten Myths About Agile**
- **Engineering Software As A Service - Chapter 7 - Requirements - Behavior-Driven Design And User Stories**
- **Engineering Software As A Service - Chapter 10 - Project Management - Scrum, Pair Programming, And Version Control Systems**
- Scrum Roles

3.0.2 Lectures

The lectures for this week are:

- Project Management Overview ≈ 6 **min**.
- Waterfall Vs. Agile ≈ 13 **min**.
- Scrum ≈ 19 **min**.
- Agile Team Roles ≈ 5 **min**.
- Agile In Practice: Pair Programming ≈ 3 **min**.
- Agile In Practice: Big Visible Charts ≈ 4 **min**.
- Agile In Practice: Planning Poker ≈ 4 **min**.
- Agile In Practice: Prioritisation Using MoSCoW ≈ 4 **min**.
- Agile In Practice: Sustainable Pace ≈ 3 **min**.
- Agile In Practice: Stand-Ups ≈ 3 **min**.
- Agile In Practice: Showcases ≈ 3 **min**.
- Agile In Practice: Test Driven Development ≈ 4 **min**.
- Agile In Practice: Automated Testing ≈ 3 **min**.
- Agile In Practice: Continuous Integration ≈ 3 **min**.

The lecture notes for this week are:

- Essential Scrum

3.0.3 Assignments

The assignment(s) for this week are:

- Lab 3 - Scripting

3.0.4 Quiz

The quiz for this week is:

- Quiz 3 - Agile Development

3.0.5 Chapter Summary

The first chapter that is being covered this week is **Chapter 29: Flow Control - Looping With while & until** from **The Linux Command Line**.

Chapter 29: Flow Control - Looping With while & until

Overview

This chapter explores looping constructs in shell scripting, focusing on the "while" and "until" commands. These constructs allow portions of programs to repeat until a specified condition is met, enhancing script usability and control flow.

Looping

Looping is a fundamental programming concept where a sequence of instructions is executed repeatedly. This section introduces the concept using a real-world analogy and then applies it to shell scripting.

Looping

Looping repeats a sequence of steps until a condition is met.

- Real-world example: Slicing a carrot involves repeating steps until the entire carrot is sliced.
- Pseudocode example:

```
1. get cutting board
2. get knife
3. place carrot on cutting board
4. lift knife
5. advance carrot
6. slice carrot
7. if entire carrot sliced, then quit; else go to step 4
```

while Loop

The "while" loop repeats commands as long as a specified condition returns a zero exit status.

while Loop

The "while" loop executes commands repeatedly as long as a condition is true.

- Syntax: `while commands; do commands; done`
- Example script:

```
#!/bin/bash
# while-count: display a series of numbers
count=1
while [[ "$count" -le 5 ]]; do
    echo "$count"
    count=$((count + 1))
done
echo "Finished."
```

Improving the Menu Program

The "while" loop can be used to enhance the menu-driven program by allowing repeated selections until the user chooses to exit.

Improving the Menu Program

Use a "while" loop to repeat menu display and selection.

- Example script:

```
#!/bin/bash
# while-menu: a menu driven system information program
DELAY=3
while [[ "$REPLY" != 0 ]]; do
    clear
    cat <<- _EOF_
    Please Select:
    1. Display System Information
    2. Display Disk Space
    3. Display Home Space Utilization
    0. Quit
    _EOF_
    read -p "Enter selection [0-3] > "
    if [[ "$REPLY" =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            sleep "$DELAY"
        elif [[ "$REPLY" == 2 ]]; then
            df -h
            sleep "$DELAY"
        elif [[ "$REPLY" == 3 ]]; then
            if [[ "$(id -u)" -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh "$HOME"
            fi
            sleep "$DELAY"
        fi
    else
        echo "Invalid entry."
        sleep "$DELAY"
    fi
done
echo "Program terminated."

```

Breaking Out of a Loop

The "break" command terminates a loop, while the "continue" command skips to the next iteration.

Breaking Out of a Loop

Use "break" to exit a loop and "continue" to skip to the next iteration.

- Example script:

```
#!/bin/bash
# while-menu2: a menu driven system information program
DELAY=3
while true; do
    clear
    cat <<- _EOF_
    Please Select:
    1. Display System Information
    2. Display Disk Space
    3. Display Home Space Utilization
    0. Quit
    _EOF_
    read -p "Enter selection [0-3] > "
    if [[ "$REPLY" =~ ^[0-3]$ ]]; then
        if [[ "$REPLY" == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            sleep "$DELAY"
            continue
        elif [[ "$REPLY" == 2 ]]; then
            df -h
            sleep "$DELAY"
            continue
        elif [[ "$REPLY" == 3 ]]; then
            if [[ "$(id -u)" -eq 0 ]]; then
```

```
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh "$HOME"
            fi
            sleep "$DELAY"
            continue
        elif [[ "$REPLY" == 0 ]]; then
            break
        fi
    else
        echo "Invalid entry."
        sleep "$DELAY"
    fi
done
echo "Program terminated."

```

until Loop

The "until" loop is similar to the "while" loop but continues until a condition returns a zero exit status.

until Loop

The "until" loop executes commands until a condition is true.

- Syntax: `until commands; do commands; done`
- Example script:

```
#!/bin/bash
# until-count: display a series of numbers
count=1
until [[ "$count" -gt 5 ]]; do
    echo "$count"
    count=$((count + 1))
done
echo "Finished."

```

Reading Files with Loops

The "while" and "until" loops can process standard input, allowing file processing within loops.

Reading Files with Loops

Process files using "while" and "until" loops.

- Example script with file redirection:

```
#!/bin/bash
# while-read: read lines from a file
while read distro version release; do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
    "$distro" \
    "$version" \
    "$release"
done < distros.txt

```

- Example script with piped input:

```
#!/bin/bash
# while-read2: read lines from a file
sort -k 1,1 -k 2n distros.txt | while read distro version release; do
    printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
    "$distro" \
    "$version" \
    "$release"
done

```

Summary of Key Concepts

Key concepts for looping in shell scripts:

- **Looping**: Repeat a sequence of steps until a condition is met.
- **while Loop**: Execute commands repeatedly as long as a condition is true.
- **Improving the Menu Program**: Use "while" loops to repeat menu selections.
- **Breaking Out of a Loop**: Use "break" to exit a loop and "continue" to skip to the next iteration.
- **until Loop**: Execute commands until a condition is true.
- **Reading Files with Loops**: Process files using "while" and "until" loops.

Understanding these looping constructs allows for more dynamic and adaptable scripts.

The last chapter that is being covered from **The Linux Command Line** this week is **Chapter 31: Flow Control - Branching With case**.

Chapter 31: Flow Control - Branching With case

Overview

This chapter continues the discussion of flow control by introducing the `case` command in bash. The `case` command simplifies the handling of multiple-choice decisions, replacing a series of `if` statements with a more readable structure.

case Command

The `case` command evaluates a word and matches it against a list of patterns, executing the corresponding commands when a match is found.

case Command

The `case` command matches a word against patterns and executes commands based on the match.

- Syntax:

```
case word in
    [pattern [| pattern]...) commands ;;
esac

```

- Example replacing multiple `if` statements with `case`:

```
#!/bin/bash
# case-menu: a menu driven system information program
clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "
case "$REPLY" in
    0) echo "Program terminated."
       exit ;;
    1) echo "Hostname: $HOSTNAME"
       uptime ;;
    2) df -h ;;
    3) if [[ "$(id -u)" -eq 0 ]]; then
           echo "Home Space Utilization (All Users)"
           du -sh /home/*
       else
           echo "Home Space Utilization ($USER)"
```

```
            du -sh "$HOME"
        fi ;;
    *) echo "Invalid entry" >&2
        exit 1 ;;
esac
```

Patterns

Patterns in the `case` command use the same syntax as pathname expansion and are terminated with a `)` character.

Patterns

Patterns in `case` match specific values or ranges of values.

- Examples of patterns:
- `a)` matches if the word equals "a".
- `[[:alpha:]])` matches if the word is a single alphabetic character.
- `???` matches if the word is exactly three characters long.
- `*.txt)` matches if the word ends with ".txt".
- `*)` matches any value (used as a default case).
- Example demonstrating pattern matching:

```
#!/bin/bash
read -p "enter word > "
case "$REPLY" in
    [[:alpha:]]) echo "is a single alphabetic character." ;;
    [ABC][0-9]) echo "is A, B, or C followed by a digit." ;;
    ???) echo "is three characters long." ;;
    *.txt) echo "is a word ending in '.txt'" ;;
    *) echo "is something else." ;;
esac
```

Combining Patterns

Multiple patterns can be combined using the vertical bar character `|`, creating an "or" conditional pattern.

Combining Patterns

Combine patterns with `|` to match multiple conditions.

- Example using combined patterns:

```
#!/bin/bash
# case-menu: a menu driven system information program
clear
echo "
Please Select:
A. Display System Information
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "
case "$REPLY" in
    q|Q) echo "Program terminated."
        exit ;;
    a|A) echo "Hostname: $HOSTNAME"
        uptime ;;
    b|B) df -h ;;
    c|C) if [[ "$(id -u)" -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/*
        else
            echo "Home Space Utilization ($USER)"
            du -sh "$HOME"
        fi ;;
    *) echo "Invalid entry" >&2
        exit 1 ;;
esac
```

Performing Multiple Actions

Modern versions of bash support the `;;` notation, allowing multiple actions to be performed for a single match.

Performing Multiple Actions

Use `;;` to continue matching after a successful match.

- Example script performing multiple actions:

```
#!/bin/bash
# case4-2: test a character
read -n 1 -p "Type a character > "
echo
case "$REPLY" in
    [[:upper:]]) echo "'$REPLY' is upper case." ;;&
    [[:lower:]]) echo "'$REPLY' is lower case." ;;&
    [[:alpha:]]) echo "'$REPLY' is alphabetic." ;;&
    [[:digit:]]) echo "'$REPLY' is a digit." ;;&
    [[:graph:]]) echo "'$REPLY' is a visible character." ;;&
    [[:punct:]]) echo "'$REPLY' is a punctuation symbol." ;;&
    [[:space:]]) echo "'$REPLY' is a whitespace character." ;;&
    [[:xdigit:]]) echo "'$REPLY' is a hexadecimal digit." ;;&
esac

```

Summary of Key Concepts

Key concepts for using the `case` command in shell scripts:

- **case Command**: Match a word against patterns and execute corresponding commands.
- **Patterns**: Use pathname expansion syntax to define match patterns.
- **Combining Patterns**: Combine patterns with `|` for multiple conditions.
- **Performing Multiple Actions**: Use `;;` to perform multiple actions for a match.

The `case` command enhances script readability and simplifies handling multiple-choice decisions.

The next set of chapters that is being covered this week is from **Agile For Dummies** and the first chapter that is covered is **Chapter 1: Getting The ABCs Of Agile**.

Chapter 1: Getting The ABCs Of Agile

Overview

This chapter provides an introduction to Agile, explaining its evolution, the Agile Manifesto, and its core principles. Agile is an incremental and iterative approach to software development that emphasizes flexibility, collaboration, and delivering high-quality software through frequent iterations.

Looking Back at Software Development Approaches

To understand Agile, it's important to review previous software development methodologies, including Code-and-Fix, Waterfall, and the Spiral model.

Looking Back at Software Development Approaches

Software development has evolved through various methodologies, each with its benefits and drawbacks.

- **Code-and-Fix/Big Bang**: Early approach where software was written and fixed in a single, large release. This method was risky and often resulted in significant errors.
- **Waterfall**: Introduced in the mid-1950s, this model follows a sequential, stage-based approach with phases such as Requirements, Design, Development, Integration, Testing, and Deployment. Despite

improvements, it faces issues like schedule risk, limited flexibility, and reduced customer involvement.

- **Spiral Model**: Emerged in the mid-1980s, combining iterative and incremental approaches to manage risk through prototyping and regular feedback.

Introducing the Agile Manifesto

The Agile Manifesto was created in 2001 by a group of developers to promote lightweight development methodologies. It emphasizes flexibility, efficiency, and teamwork.

Introducing the Agile Manifesto

The Agile Manifesto consists of 68 words that highlight four key values.

- **Individuals and interactions over processes and tools**: Emphasizes the importance of collaboration and communication among team members.
- **Working software over comprehensive documentation**: Prioritizes delivering functional software over creating extensive documentation.
- **Customer collaboration over contract negotiation**: Focuses on continuous collaboration with customers to meet their needs.
- **Responding to change over following a plan**: Values flexibility and adaptability to changing requirements.

The 12 Principles that Drive the Agile Manifesto

The Agile Manifesto is supported by 12 principles that further elaborate on its values.

The 12 Principles that Drive the Agile Manifesto

The Agile principles provide detailed guidance for agile software development.

- Satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development.
- Deliver working software frequently, with a preference for shorter timescales.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals, providing them the necessary support and trust.
- Face-to-face conversation is the most efficient and effective method of conveying information.
- Working software is the primary measure of progress.
- Promote sustainable development with a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- Reflect regularly on how to become more effective and adjust behavior accordingly.

Redefining Today's Agile

Agile has grown in popularity and scalability, extending to larger organizations and more complex projects.

Redefining Today's Agile

Agile is widely accepted and applied across various industries, demonstrating its flexibility and scalability.

- **Growing popularity**: Agile is used in numerous fields, including web-based applications, mobile applications, business intelligence systems, life-critical systems, and embedded software. It is adopted by financial companies, retailers, healthcare organizations, manufacturers, and government agencies.

- **Growing scalability**: Teams are successfully scaling agile practices to larger and more complex projects, integrating core agile principles into the entire software life cycle.

Summary of Key Concepts

Key concepts from Chapter 1 on Agile:

- **Evolution of Software Development**: Understanding the history and limitations of previous methodologies helps appreciate Agile's benefits.
- **Agile Manifesto**: Highlights four core values that prioritize individuals, working software, customer collaboration, and responsiveness to change.
- **Agile Principles**: Twelve principles provide detailed guidance on implementing Agile practices effectively.
- **Modern Agile**: Agile's widespread adoption and scalability demonstrate its applicability to various industries and project sizes.

Understanding Agile's core values and principles is crucial for its successful implementation in software development projects.

The next chapter that is being covered from **Agile For Dummies** is **Chapter 2: Understanding Agile Roles**.

Chapter 2: Understanding Agile Roles

Overview

This chapter explores the various roles within Agile teams. It emphasizes that roles are flexible and can evolve over time, and it distinguishes roles from positions. Agile teams focus on collaboration and delivering solutions, with all members contributing regardless of their specific job titles.

Being a Stakeholder

A stakeholder is anyone financially impacted by the project's outcome. This role includes more than just end-users and can encompass various positions.

Being a Stakeholder

Stakeholders are those financially impacted by the project's outcome.

- Roles include:
  - Direct or indirect users
  - Managers of users
  - Senior managers
  - Operations or IT staff
  - Project funders
  - Auditors
  - Program/portfolio managers
  - Developers of interacting systems
  - Maintenance professionals

Representing Stakeholders: The Product Owner

The product owner represents the stakeholder community and ensures the team's work aligns with stakeholder needs. This role involves maintaining a prioritized list of work items and answering team questions.

Representing Stakeholders: The Product Owner

The product owner is the voice of the customer and manages the backlog of work items.

- Responsibilities include:
  - Communicating project status to stakeholders
  - Developing project strategy and goals
  - Understanding and conveying customer needs
  - Gathering and prioritizing requirements
  - Managing the product's budget and profitability
  - Choosing release dates
  - Accepting or rejecting completed work
  - Presenting team accomplishments

Being a Team Member

Team members are responsible for producing the actual solution. They perform a variety of tasks including testing, analysis, design, programming, and more.

Being a Team Member

Team members contribute to all aspects of the project to deliver the solution.

- Activities include:
  - Testing
  - Analysis
  - Architecture
  - Design
  - Programming
  - Planning
  - Estimation
- Team members continuously develop their skills and take responsibility for task completion.

Assuming the Team Lead

The team lead is a servant-leader who guides the team in management activities. This role involves facilitating communication, empowering the team, and ensuring resources are available.

Assuming the Team Lead

The team lead supports the team's success and acts as an agile coach.

- Responsibilities include:
  - Facilitating communication
  - Empowering the team to optimize processes
  - Ensuring the team has necessary resources
  - Managing issue resolution
- An experienced team lead may also act as a mentor for new teams.

Acting As the Architecture Owner

The architecture owner is responsible for mitigating project risk by managing architecture decisions and facilitating the creation and evolution of the solution's design.

Acting As the Architecture Owner

The architecture owner ensures the team mitigates architecture-related risks.

- Responsibilities include:
  - Owning architecture decisions
  - Facilitating the creation and evolution of the solution's design
- On small teams, the team lead may also fulfill this role.

Stepping Up As an Agile Mentor

The agile mentor, or agile coach, provides guidance and feedback to help teams implement Agile practices effectively. This role is typically filled by someone outside the team with significant Agile experience.

Stepping Up As an Agile Mentor

Agile mentors guide teams in adopting and improving Agile practices.

- Characteristics of an agile mentor:
  - Acts as a coach, not part of the team
  - Often external to the organization for objective guidance
  - Experienced in implementing Agile techniques

Looking at Agile Secondary Roles

In addition to primary roles, Agile projects may include secondary roles to address specific needs.

Looking at Agile Secondary Roles

Secondary roles support the team with specialized expertise.

- Possible secondary roles include:
  - Domain expert: Provides deep business/domain knowledge.
  - Specialist: Addresses specific needs, such as business analysis or program management.
  - Technical expert: Helps overcome technical challenges and transfers skills to the team.
  - Independent tester: Provides parallel validation of work.
  - Integrator: Manages system integration in complex environments.

Summary of Key Concepts

Key concepts from Chapter 2 on Agile roles:

- **Stakeholders**: Financially impacted individuals involved in the project.
- **Product Owner**: Represents stakeholders and manages the backlog.
- **Team Members**: Perform various tasks to deliver the solution.
- **Team Lead**: Guides and supports the team as a servant-leader.
- **Architecture Owner**: Manages architecture decisions to mitigate risks.
- **Agile Mentor**: Provides guidance and feedback to help teams adopt Agile practices.
- **Secondary Roles**: Include domain experts, specialists, technical experts, independent testers, and integrators.

Understanding these roles helps in effectively implementing Agile methodologies in projects.

The next chapter that is being covered from **Agile For Dummies** is **Chapter 3: Getting Started With Agile**.

Chapter 3: Getting Started With Agile

Overview

This chapter provides guidance on how to get started with Agile, including planning practices, managing and tracking progress, and reflecting for future improvement. Agile breaks down stakeholders' needs into small chunks, ranked and worked on in priority order over short iterations, reviewed for approval, and delivered to production.

Agile Planning

Agile teams typically divide their release schedule into a series of fixed-length development iterations, usually two to four weeks. Planning involves scheduling work to be done during an iteration or release and assigning tasks to team members.

Agile Planning

Agile planning occurs at three levels: release planning, iteration planning, and daily planning.

- **Release planning**: The product owner creates a release plan at the start of each release, containing a release schedule for a specific set of features.
- **Iteration planning**: At the beginning of each iteration (or sprint in Scrum), the team identifies the work to be done, a process known as self-organization.
- **Daily planning**: Development teams hold daily standup meetings to plan the day, usually lasting 5 to 15 minutes.

Attending the Daily Coordination Meeting

Agile teams start each workday with a brief daily coordination meeting to note completed items, identify impediments, and plan the day's tasks.

Attending the Daily Coordination Meeting

The daily coordination meeting, or daily standup, is crucial for tracking progress and addressing issues.

- Each team member states:
  - Yesterday, I completed [state items completed].
  - Today, I'm going to take on [state task].
  - My impediments are [state impediments, if any].

Creating User Stories

User stories capture brief descriptions of product requirements, focusing on what the requirement must accomplish for whom.

Creating User Stories

User stories are used to define what the software will do and what services it will provide to its users.

- A user story should include:
  - Title: A name for the user story.
  - As a <user or persona>, I want to <take this action>, so that <I get this benefit>.
  - Validation steps: When I <take this action>, this happens <description of action>.

- Additional elements may include:
  - ID: A unique identifier for the user story.
  - Value and effort estimate: Value indicates the benefit to the organization, and effort represents the complexity of the task.
  - Author: The person who created the user story.

Estimating Your Work

Agile teams estimate their work in points, which represent the size and complexity of tasks rather than time.

Estimating Your Work

Points are used to estimate the size and complexity of work items, helping teams manage their workload.

- Points are assigned in whole numbers (e.g., 1, 2, 3) and represent relative sizes and complexity of tasks.
- Teams should avoid equating points to hours to maintain consistency.

Tracking Velocity

Velocity is the total number of story points completed in an iteration, used to measure a team's work output.

Tracking Velocity

Velocity helps teams understand their productivity and forecast project timelines.

- Velocity is calculated as the total number of completed story points divided by the number of iterations.
- Example: If a team completes 15, 19, 21, and 25 points in four iterations, their average velocity is 20 points per iteration.

Measuring Progress with Burndown Reports

Burndown reports track the number of points completed over time, helping monitor progress and remaining work.

Measuring Progress with Burndown Reports

Burndown reports show the progress of completing tasks and clearing the backlog.

- The X-axis represents iterations, while the Y-axis represents the total number of points.
- Burndown reports help visualize the team's progress and velocity.

Test-Driven Development

Test-Driven Development (TDD) involves writing tests before writing the code they validate, leading to higher-quality software.

Test-Driven Development

TDD ensures that code is written to pass predefined tests, enhancing code quality and efficiency.

- Write a small test before writing the corresponding code.
- Run the test to ensure it fails, then write the code to make the test pass.
- Use automated unit tests to quickly identify and address issues.

Continuous Integration and Deployment

Continuous Integration (CI) and Continuous Deployment (CD) involve regularly integrating and testing code, ensuring high-quality software.

Continuous Integration and Deployment

CI and CD help maintain high-quality software by frequently integrating and deploying changes.

- CI involves regularly integrating and testing changes, ideally multiple times a day.
- CD extends CI by automatically deploying successful builds to various environments.

Presenting Results at the Iteration Review

The iteration review, or sprint review, is a meeting to demonstrate completed user stories and gather feedback from stakeholders.

Presenting Results at the Iteration Review

The iteration review showcases the team's accomplishments and gathers stakeholder feedback.

- The review is informal but organized, focusing on demonstrating completed work.
- Stakeholders provide feedback, and new user stories may be created based on this feedback.

Collecting Feedback in the Iteration Review Meeting

Feedback from the iteration review is used to improve the product and inform future iterations.

Collecting Feedback in the Iteration Review Meeting

Collect feedback during the iteration review to refine and prioritize future work.

- The product owner adds new user stories to the backlog and re-prioritizes existing ones based on feedback.

Learning and Improving at the Iteration Retrospective

The iteration retrospective is a meeting to discuss the iteration's successes and areas for improvement, aiming for continuous process enhancement.

Learning and Improving at the Iteration Retrospective

The iteration retrospective focuses on continuous improvement and process optimization.

- Team members discuss what went well, what didn't, and how to improve future iterations.
- The goal is to enhance team morale, efficiency, and velocity.

Summary of Key Concepts

Key concepts from Chapter 3 on getting started with Agile:

- **Agile Planning**: Involves release, iteration, and daily planning to manage and schedule work.
- **Daily Coordination Meeting**: Brief meetings to track progress and address issues.
- **Creating User Stories**: Captures product requirements in a simple, descriptive format.
- **Estimating Work**: Uses points to estimate task size and complexity.
- **Tracking Velocity**: Measures work output and helps forecast project timelines.
- **Burndown Reports**: Visualize progress and remaining work.
- **Test-Driven Development**: Ensures high-quality code through pre-written tests.
- **Continuous Integration and Deployment**: Maintains high-quality software through frequent integration and deployment.
- **Iteration Review**: Demonstrates completed work and gathers feedback.
- **Iteration Retrospective**: Focuses on continuous improvement and process enhancement.

These concepts are crucial for effectively implementing Agile practices and achieving successful project outcomes.

The next chapter that is being covered from **Agile For Dummies** is **Chapter 4: Choosing An Agile Approach**.

Chapter 4: Choosing An Agile Approach

Overview

This chapter explores various Agile methodologies, providing insights into their strengths and weaknesses. Understanding these approaches helps tailor an Agile strategy to meet the unique needs of different situations.

Scrum: Organizing Construction

Scrum is the most popular Agile approach, emphasizing adjustments based on experience rather than theory. It focuses on delivering shippable functionality through iterations called sprints.

Scrum: Organizing Construction

Scrum emphasizes experience-based adjustments and iterative development.

- Key deliverables:
  - **Product backlog**: Full list of requirements.
  - **Sprint backlog**: Requirements and tasks for a sprint.
  - **Burndown charts**: Visual progress representations.
  - **Shippable functionality**: Usable product meeting business goals.
- Key practices:
  - Release planning
  - Sprint planning
  - Daily scrum meeting
  - Sprint review meeting
  - Sprint retrospective

XP: Putting the Customer First

Extreme Programming (XP) focuses on customer satisfaction, adapting to new requests as they arise. XP teams work to solve problems efficiently and ensure continuous integration and testing.

XP: Putting the Customer First

XP emphasizes customer satisfaction through adaptive planning and continuous feedback.

- Key practices:
  - **Coding standard**: Follow established coding guidelines.
  - **Collective ownership**: All team members can view and edit code.
  - **Continuous integration**: Frequent code integration and testing.
  - **Test-Driven Development (TDD)**: Write tests before code.
  - **Customer tests**: Capture detailed requirements as acceptance tests.
  - **Refactoring**: Improve design incrementally.
  - **Pair programming**: Two programmers work together on the same code.

  - **Planning game**: High-level and detailed planning.
  - **Simple design**: Implement the simplest solution.
  - **Small releases**: Frequent deployment of working software.
  - **Sustainable pace**: Maintain a consistent and manageable work pace.
  - **Whole team**: Team has all skills required to deliver the solution.

Lean Programming: Producing JIT

Lean programming, originating from manufacturing, focuses on just-in-time (JIT) processes, reducing waste and optimizing production efficiency.

Lean Programming: Producing JIT

Lean principles optimize the whole IT value stream, reducing waste and increasing efficiency.

- Key principles:
  - Eliminate waste
  - Build in quality
  - Create knowledge
  - Defer commitment
  - Deliver quickly
  - Respect people
  - Optimize the whole

Kanban: Improving on Existing Systems

Kanban is a lean methodology focused on visualizing workflow and limiting work in progress (WIP) to improve efficiency and quality.

Kanban: Improving on Existing Systems

Kanban emphasizes visual workflow and WIP limitations to enhance productivity.

- Key principles:
  - **Visualizing workflow**: Use a Kanban board to track work stages.
  - **Limit work in progress (WIP)**: Reduce lead time and increase productivity by limiting WIP.

Agile Modeling

Agile Modeling (AM) is a collection of values, principles, and practices for effective and lightweight software modeling.

Agile Modeling

Agile Modeling focuses on lightweight, just-in-time (JIT) modeling to support development.

- Key practices:
  - Active stakeholder participation
  - Architecture envisioning
  - Document continuously and late
  - Executable specifications
  - Iteration modeling
  - Just barely good enough artifacts
  - Look-ahead modeling
  - Model storming

  - Multiple models
  - Prioritized requirements
  - Requirements envisioning
  - Single-source information
  - TDD

Unified Process (UP)

The Unified Process (UP) is an iterative and incremental methodology emphasizing collaboration and adaptability, structured into four phases.

Unified Process (UP)

UP combines iterative and incremental approaches within a structured lifecycle.

- Phases:
  - **Inception**: Define the project scope and objectives.
  - **Elaboration**: Develop the project's architecture and resolve high-risk elements.
  - **Construction**: Build the product through multiple iterations.
  - **Transition**: Deploy the product to users.
- UP supports self-organizing teams and continuous integration.

Summary of Key Concepts

Key concepts from Chapter 4 on choosing an Agile approach:

- **Scrum**: Focuses on iterative development and experience-based adjustments.
- **XP**: Emphasizes customer satisfaction and adaptive planning.
- **Lean Programming**: Reduces waste and optimizes efficiency through JIT processes.
- **Kanban**: Enhances productivity by visualizing workflow and limiting WIP.
- **Agile Modeling**: Supports lightweight, JIT modeling practices.
- **Unified Process (UP)**: Combines iterative and incremental development within a structured lifecycle.

Understanding these Agile methodologies helps tailor an approach to meet specific project needs.

The last chapter that is being covered from **Agile For Dummies** is **Chapter 10: Ten Myths About Agile**.

Chapter 10: Ten Myths About Agile

Overview

This chapter addresses common misconceptions about Agile, providing clarity on its principles and practices. It aims to dispel myths and demonstrate how Agile can benefit organizations by improving project success rates.

Agile Is a Fad

Agile is not a passing trend but a well-established approach that has been formalized through the Agile Manifesto. It has been in use for decades because it consistently produces successful projects.

Agile Is a Fad

Agile is a well-established, enduring approach that has proven effective over time.

- Agile's formalization through the Agile Manifesto has strengthened its credibility.
- It continues to be preferred over traditional project management due to its success rates.

Agile Isn't Disciplined

Despite its collaborative nature, Agile requires significant discipline. It demands rapid responses, incremental delivery, close stakeholder collaboration, and adherence to specific practices.

Agile Isn't Disciplined

Agile requires a high level of discipline and coordination.

- Teams must reduce feedback cycles, deliver incrementally, and engage closely with stakeholders.
- Individual practices, like Test-Driven Development (TDD), necessitate discipline.

Agile Means "We Don't Plan"

Agile involves incremental and evolutionary planning, rather than planning everything upfront. This method has proven more successful than traditional approaches.

Agile Means "We Don't Plan"

Agile employs continuous, incremental planning to adapt to changes effectively.

- Planning is ongoing and evolves throughout the project.
- This approach allows for flexibility and responsiveness to changing requirements.

Agile Means "No Documentation"

While Agile keeps documentation lightweight, it does not eliminate it. Agile teams document continuously and use executable specifications to ensure clarity and accountability.

Agile Means "No Documentation"

Agile promotes lightweight but essential documentation practices.

- Documentation is created continuously throughout the project.
- Strategies like executable specifications ensure necessary documentation is maintained.

Agile Is Only Effective for Collocated Teams

Agile can work effectively for distributed teams with the right practices and tools. Team cohesion is crucial, and proper tools can facilitate effective collaboration regardless of physical location.

Agile Is Only Effective for Collocated Teams

Agile can be successful with distributed teams using appropriate tools and practices.

- Key is to adopt tools and methods that build team cohesion.
- Successful examples include remote collaboration platforms and regular virtual meetings.

Agile Doesn't Scale

Agile can scale to large teams and complex projects with proper organization and tools. Large-scale Agile practices involve more structured approaches and tools to manage complexity.

Agile Doesn't Scale

Agile can scale effectively with the right strategies and tools.

- Large teams use structured tools like IBM Rational Requirements Composer for modeling.
- Agile practices are adapted to handle the complexity of large-scale projects.

Agile Is Unsuitable for Regulated Environments

Agile can be applied in regulated environments, providing faster delivery and higher quality outputs. Agile's iterative nature helps ensure compliance with regulatory mandates.

Agile Is Unsuitable for Regulated Environments

Agile is effective in regulated environments, ensuring compliance and quality.

- Iterative processes allow for regular reviews and adjustments to meet regulations.
- Examples include medical, finance, and government sectors.

Agile Means We Don't Know What Will Be Delivered

Agile provides greater control over building the right product through iterative feedback and stakeholder collaboration. It ensures that requirements are continuously refined and agreed upon.

Agile Means We Don't Know What Will Be Delivered

Agile ensures clarity and control over project deliverables through iterative processes.

- Regular iterations and stakeholder feedback guide the development.
- Disciplined Agile Delivery (DAD) explicitly explores high-level requirements early on.

Agile Won't Work at My Company

Agile can work in any company with the right cultural shift and support. It requires buy-in from all levels of the organization and a willingness to embrace frequent feedback and transparency.

Agile Won't Work at My Company

Agile can succeed in any organization with the right cultural adaptation.

- Requires support from executives and all team members.
- Emphasizes frequent feedback and transparency to foster trust and collaboration.

It's Enough for My Development Team to Be Agile

For Agile to be fully effective, the entire organization needs to adopt Agile practices. This includes all teams involved in the project, such as testing and operations.

It's Enough for My Development Team to Be Agile

Agile requires organization-wide adoption to be truly effective.

- All teams, including testing and operations, must embrace Agile practices.
- The effectiveness of Agile is limited by the slowest group in the process.

Agile Is a Silver Bullet

Agile is not a one-size-fits-all solution. While it excels in projects undergoing development or rapid changes, it may not be necessary for stable, maintenance-mode projects.

Agile Is a Silver Bullet

Agile is not suitable for every project or team.

- Best suited for projects in development or with frequent changes.
- May not be required for stable projects with minimal changes.

Summary of Key Concepts

Key concepts from Chapter 10 on myths about Agile:

- **Agile Is a Fad**: Agile is a proven, enduring approach.
- **Agile Isn't Disciplined**: Agile requires significant discipline and collaboration.
- **Agile Means "We Don't Plan"**: Planning is continuous and adaptive.
- **Agile Means "No Documentation"**: Documentation is kept lightweight but essential.
- **Agile Is Only Effective for Collocated Teams**: Distributed teams can succeed with the right tools and practices.
- **Agile Doesn't Scale**: Agile scales effectively with appropriate tools and strategies.
- **Agile Is Unsuitable for Regulated Environments**: Agile ensures compliance and quality in regulated sectors.
- **Agile Means We Don't Know What Will Be Delivered**: Agile provides better control over deliverables.
- **Agile Won't Work at My Company**: Agile can succeed with cultural adaptation and support.
- **It's Enough for My Development Team to Be Agile**: Organization-wide adoption is necessary for full effectiveness.
- **Agile Is a Silver Bullet**: Agile is not a cure-all; it's best for development and rapidly changing projects.

Understanding these myths and truths about Agile helps organizations implement it effectively and realize its benefits.

The next set of chapters that is being covered this week is from **Engineering Software As A Service** and the first chapter that is covered is **Chapter 7: Requirements - Behavior-Driven Design And User Stories**

Chapter 7: Requirements - Behavior-Driven Design And User Stories

---

Overview

This chapter discusses Behavior-Driven Design (BDD) and its use in Agile software development, focusing on creating user stories to define requirements. BDD emphasizes collaboration with stakeholders to ensure that the software meets their needs and evolves based on continuous feedback.

Behavior-Driven Design and User Stories

BDD involves writing user stories that describe how the application is expected to be used. These stories are used to plan and prioritize development, ensuring that the resulting software meets stakeholders' desires.

Behavior-Driven Design and User Stories

BDD is an Agile approach that focuses on specifying the behavior of an application through user stories.

- User stories are lightweight requirements written in everyday language.
- They help stakeholders and developers collaborate and avoid misunderstandings.
- Example user story format:

```
Feature: Add a movie to RottenPotatoes
As a movie fan
So that I can share a movie with other movie fans
I want to add a movie to RottenPotatoes database

```

Points, Velocity, and Pivotal Tracker

Agile teams use points to estimate the effort required for user stories, and velocity to measure progress. Tools like Pivotal Tracker help manage these aspects.

Points, Velocity, and Pivotal Tracker

Points and velocity are key metrics for Agile teams.

- Points quantify the effort required for user stories.
- Velocity measures the average points completed per iteration.
- Pivotal Tracker is a tool for tracking user stories and project progress.
- Example of using points and velocity:

```
# Assign points to user stories and track completion
Story: Add movie feature
Points: 3
Velocity: 5 points per iteration

```

SMART User Stories

Good user stories should be Specific, Measurable, Achievable, Relevant, and Timeboxed (SMART).

SMART User Stories

SMART criteria ensure that user stories are effective and actionable.

- **Specific**: Clearly define what is needed.
- **Measurable**: Include criteria to verify success.
- **Achievable**: Can be completed within one iteration.
- **Relevant**: Provide value to stakeholders.
- **Timeboxed**: Can be completed within a set time frame.
- Example of a SMART user story:

```
Feature: User can search for a movie by title
As a user
I want to search for movies by title
So that I can find movies quickly

```

Lo-Fi User Interface Sketches and Storyboards

Lo-Fi sketches and storyboards are used to design user interfaces (UIs) in a way that encourages feedback and iteration.

Lo-Fi User Interface Sketches and Storyboards

Lo-Fi sketches and storyboards help visualize and refine UI design.

- Sketches are simple drawings of UIs.
- Storyboards show the flow of interactions over time.
- Example storyboard:

```
1. User sees login page.
2. User enters credentials.
3. User is taken to the dashboard.

```

Agile Cost Estimation

Agile teams estimate costs based on the effort required for user stories, often using time and materials contracts.

Agile Cost Estimation

Cost estimation in Agile focuses on advising clients and adjusting team size for efficiency.

- Estimates are based on points and velocity.
- Time and materials contracts are used.
- Example cost estimation process:

```
# Estimate time and cost for user stories
Story: Add movie feature
Estimated time: 3 weeks
Cost: \$15,000

```

Introducing Cucumber and Capybara

Cucumber and Capybara are tools that turn user stories into automated acceptance and integration tests.

Introducing Cucumber and Capybara

Cucumber and Capybara automate testing based on user stories.

- Cucumber uses plain language scenarios for testing.
- Capybara simulates user interactions in a web browser.
- Example Cucumber scenario:

```
Feature: Add a movie to RottenPotatoes
Scenario: Add a new movie
Given I am on the RottenPotatoes home page
When I follow "Add new movie"
Then I should be on the Create New Movie page

```

Summary of Key Concepts

Key concepts from this chapter include:

- **Behavior-Driven Design**: Collaborating with stakeholders to define application behavior.
- **Points and Velocity**: Measuring effort and progress in Agile projects.
- **SMART User Stories**: Ensuring user stories are specific, measurable, achievable, relevant, and timeboxed.
- **Lo-Fi UI Sketches and Storyboards**: Designing user interfaces using simple sketches and storyboards.
- **Agile Cost Estimation**: Estimating project costs based on points, velocity, and team size.

- **Cucumber and Capybara**: Tools for automating acceptance and integration tests based on user stories.

The last chapter that is being covered from **Engineering Software As A Service** is **Chapter 10: Project Management - Scrum, Pair Programming, And Version Control Systems**.

Chapter 10: Project Management - Scrum, Pair Programming, And Version Control Systems

---

Overview

This chapter explores project management techniques in Agile, focusing on Scrum, pair programming, and version control systems. These practices are essential for enhancing collaboration, improving code quality, and managing changes in software development projects.

It Takes a Team: Two-Pizza and Scrum

Scrum is an Agile framework that emphasizes team collaboration and iterative progress. It uses small teams, often called "two-pizza" teams, which can be fed with two pizzas.

It Takes a Team: Two-Pizza and Scrum

Scrum promotes teamwork and iterative development through structured roles and ceremonies.

- Key roles:
  - **Team**: Delivers the software.
  - **ScrumMaster**: Removes impediments and keeps the team focused.
  - **Product Owner**: Represents the customer and prioritizes user stories.
- Key practices:
  - Daily stand-up meetings
  - Sprint planning
  - Sprint reviews
  - Sprint retrospectives

Pair Programming

Pair programming is an Agile practice where two developers work together at one workstation. It enhances code quality and knowledge sharing.

Pair Programming

Pair programming involves two developers working together, improving code quality and collaboration.

- Roles:
  - **Driver**: Writes the code.
  - **Observer/Navigator**: Reviews each line of code and thinks strategically.
- Benefits:
  - Improved code quality
  - Enhanced knowledge sharing
  - Reduced development time

Agile Design and Code Reviews

Agile design and code reviews involve continuous feedback and collaboration, often making formal reviews unnecessary.

Agile Design and Code Reviews

Continuous feedback in Agile often eliminates the need for formal design and code reviews.

- Tools like pull requests allow for ongoing mini-reviews.
- Benefits include constant code review and immediate feedback.

Version Control for the Two-Pizza Team: Merge Conflicts

Effective version control is crucial for managing changes and resolving conflicts in a team setting.

Version Control for the Two-Pizza Team: Merge Conflicts

Version control systems help manage changes and resolve conflicts in team projects.

- Use a shared-repository model.
- Always commit before merging changes.
- Key commands:
  - `git pull`: Fetches and merges changes from the remote repository.
  - `git push`: Updates the remote repository with local commits.

Using Branches Effectively

Branches allow teams to work on multiple features simultaneously without interfering with each other's work.

Using Branches Effectively

Branches enable parallel development of features without disrupting the main codebase.

- Types of branches:
  - **Feature branches**: For developing new features.
  - **Release branches**: For stabilizing code before a release.
- Best practices:
  - Frequently merge changes to avoid conflicts.
  - Use Git commands to manage branches efficiently.

Reporting and Fixing Bugs: The Five R's

Managing the lifecycle of a bug involves several stages, ensuring thorough resolution and prevention of future issues.

Reporting and Fixing Bugs: The Five R's

The lifecycle of a bug involves reporting, reproducing, creating regression tests, repairing, and releasing fixes.

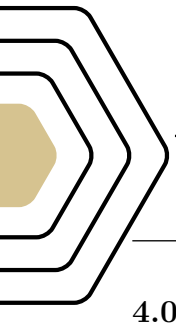
- Stages:
  - Reporting
  - Reproducing
  - Regression testing
  - Repairing
  - Releasing

Summary of Key Concepts

Key concepts from Chapter 10 on project management:

- **Two-Pizza and Scrum**: Emphasize small, collaborative teams with defined roles and iterative progress.
- **Pair Programming**: Enhances code quality and knowledge sharing through collaboration.
- **Agile Design and Code Reviews**: Utilize continuous feedback to maintain code quality.
- **Version Control**: Manage changes and resolve conflicts efficiently in team settings.
- **Using Branches**: Support parallel development and maintain a stable main codebase.
- **Bug Lifecycle**: Ensure thorough resolution of bugs with structured management practices.

Version Control



Version Control

4.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- **Pro Git - Chapter 1 - Getting Started**
- **Pro Git - Chapter 2 - Git Basics**
- **Pro Git - Chapter 3 - Git Branching**

4.0.2 Lectures

The lectures for this week are:

- Overview Of Version Control ≈ 13 **min**.
- Introduction To Git ≈ 25 **min**.
- Git Merging And Branching ≈ 21 **min**.
- AT&T Archives: The UNIX Operating System ≈ 27 **min**.
- Where GREP Came From - Computerphile ≈ 10 **min**.
- UNIX Pipeline (Brian Kernighan) - Computerphile ≈ 5 **min**.

4.0.3 Assignments

The assignment(s) for this week are:

- Lab 4 - Version Control

4.0.4 Quiz

The quiz for this week is:

- Quiz 4 - Version Control

4.0.5 Project

The assignment(s) for the project this week is:

- Project Milestone 2: Agile Meeting Recording

4.0.6 Chapter Summary

The first chapter that is being covered this week is **Chapter 1: Getting Started**.

Chapter 1: Getting Started

Overview

This chapter introduces Git, covering its history, setup, and essential concepts. Git is a distributed version control system that helps manage changes to files and coordinate work among multiple people.

About Version Control

Version control systems (VCS) record changes to files over time, enabling you to recall specific versions later. This chapter discusses local, centralized, and distributed version control systems.

About Version Control

VCS records changes to files over time, allowing specific versions to be recalled.

- **Local VCS**: Keeps changes in a simple database.
- **Centralized VCS**: Uses a single server to store all changes and file versions.
- **Distributed VCS**: Each client mirrors the entire repository.

A Short History of Git

Git was created in 2005 by Linus Torvalds for the Linux kernel development, emphasizing speed, simple design, and support for non-linear development.

A Short History of Git

Git was developed in response to the limitations of previous systems, focusing on speed and non-linear development.

- Developed by Linus Torvalds in 2005.
- Inspired by the challenges faced with BitKeeper.
- Key goals: Speed, simplicity, strong support for branching, and distributed development.

What is Git?

Git is a distributed VCS that uses snapshots instead of differences, providing a more efficient way to manage changes.

What is Git?

Git uses snapshots to store data, making it efficient and reliable.

- **Snapshots**: Git records the state of the project at each commit.
- **Local operations**: Most Git operations are local, making them fast.
- **Integrity**: Everything in Git is checksummed with a SHA-1 hash.
- **Data addition**: Nearly all actions in Git add data to the repository.

The Three States

Files in Git can be in one of three states: modified, staged, or committed. Understanding these states is crucial for using Git effectively.

The Three States

Git files can be modified, staged, or committed.

- **Modified**: Changes have been made but not committed.
- **Staged**: Marked to be included in the next commit.
- **Committed**: Data is stored in the local database.

The Command Line

Using Git on the command line provides access to all Git commands and is essential for mastering Git.

The Command Line

The command line provides full access to Git's functionality.

- Essential for using all Git commands.
- Examples include `git init`, `git add`, `git commit`.

Installing Git

Installing Git varies by operating system. The chapter covers installation on Linux, macOS, and Windows.

Installing Git

Install Git to start using it on your system.

- **Linux**: Use package managers like `dnf` or `apt`.
- **macOS**: Use Xcode Command Line Tools or a binary installer.
- **Windows**: Download from the Git website or use the Chocolatey package manager.

First-Time Git Setup

Configure Git with your personal information and preferred settings.

First-Time Git Setup

Set up Git with your user name, email, and default text editor.

- Configure user identity:

```
\$ git config --global user.name "John Doe"
\$ git config --global user.email johndoe@example.com

```

- Set default text editor:

```
\$ git config --global core.editor emacs

```

Summary of Key Concepts

Key concepts from Chapter 1 on getting started with Git:

- **Version Control Systems**: Manage changes to files over time.
- **History of Git**: Developed for the Linux kernel with a focus on speed and efficiency.
- **Snapshots**: Git uses snapshots to track project changes.
- **Three States**: Modified, staged, and committed states for files.
- **Command Line**: Provides full access to Git functionality.
- **Installing Git**: Procedures for Linux, macOS, and Windows.
- **First-Time Setup**: Configure user identity and settings.

The next chapter that is being covered this week is **Chapter 2: Git Basics**.

Chapter 2: Git Basics

Overview

This chapter provides an introduction to Git basics, covering the fundamental commands and concepts necessary to start using Git effectively. By the end of this chapter, you will be able to configure and initialize a repository, track files, stage and commit changes, ignore files, undo mistakes, view project history, and interact with remote repositories.

Getting a Git Repository

There are two main ways to obtain a Git repository: initializing a new repository in an existing directory or cloning an existing repository.

Getting a Git Repository

Initialize a new repository or clone an existing one.

- **Initialize a repository**:

```
\$ cd /path/to/project
\$ git init

```

- **Clone a repository**:

```
\$ git clone <url>

```

Recording Changes to the Repository

Changes to files are recorded in the repository through a cycle of modifying, staging, and committing files.

Recording Changes to the Repository

Track, stage, and commit changes to your files.

- **Track new files**:

```
\$ git add <file>

```

- **Stage modified files**:

```
\$ git add <file>

```

- **Commit changes**:

```
\$ git commit -m "commit message"

```

Viewing the Commit History

The commit history can be viewed to understand the changes made over time.

Viewing the Commit History

Use the `git log` command to view commit history.

- **Basic log**:

```
\$ git log

```

- **Detailed log with patch**:

```
\$ git log -p
```

- **Short log format**:

```
\$ git log --oneline
```

Undoing Things

Git provides several commands to undo changes at various stages.

Undoing Things

Undo changes using Git commands to revert, reset, and clean.

- **Unstage a file**:

```
\$ git reset HEAD <file>
```

- **Discard changes in a file**:

```
\$ git checkout -- <file>
```

- **Amend a commit**:

```
\$ git commit --amend
```

Working with Remotes

Remote repositories are essential for collaboration. You can fetch, pull, and push changes to and from remotes.

Working with Remotes

Interact with remote repositories for collaboration.

- **Add a remote**:

```
\$ git remote add <name> <url>
```

- **Fetch changes**:

```
\$ git fetch <remote>
```

- **Push changes**:

```
\$ git push <remote> <branch>
```

Summary of Key Concepts

Key concepts from Chapter 2 on Git basics:

- **Getting a Repository**: Initialize a new repository or clone an existing one.
- **Recording Changes**: Track, stage, and commit changes to files.
- **Viewing History**: Use git log to view the commit history.
- **Undoing Changes**: Commands to unstage files, discard changes, and amend commits.
- **Working with Remotes**: Add, fetch from, and push to remote repositories.

The last chapter that is being covered this week is **Chapter 3: Git Branching**.

Chapter 3: Git Branching

Overview

This chapter delves into Git branching, a core feature that distinguishes Git from other version control systems (VCS). Branching in Git is efficient and lightweight, facilitating multiple workflows and frequent branching and merging.

Branches in a Nutshell

Branches in Git allow for divergent lines of development within the same project. This section explains how Git's branching model operates, focusing on its efficiency and ease of use.

Branches in a Nutshell

Git branches are lightweight and encourage frequent branching and merging.

- Branches are pointers to commits.
- The default branch is `master`.
- Creating a branch:

```
\$ git branch <branchname>
```

- Switching branches:

```
\$ git checkout <branchname>
```

Basic Branching and Merging

This section covers basic branching and merging operations, illustrating common workflows.

Basic Branching and Merging

Branching and merging allow for isolated development and integration of changes.

- Create and switch to a new branch:

```
\$ git checkout -b <branchname>
```

- Merge changes from one branch into another:

```
\$ git checkout master
\$ git merge <branchname>
```

- Example workflow for a hotfix:

```
\$ git checkout master
\$ git checkout -b hotfix
# make changes
\$ git commit -a -m "Fix issue"
\$ git checkout master
\$ git merge hotfix
```

Branch Management

Effective branch management involves creating, deleting, and listing branches, as well as understanding merged and unmerged branches.

Branch Management

Manage branches to keep the repository organized and up-to-date.

- List branches:

```
\$ git branch
```

- Delete a branch:

```
\$ git branch -d <branchname>
```

- List merged branches:

```
\$ git branch --merged
```

Branching Workflows

Different workflows suit various project needs, from simple feature branches to complex multi-branch strategies.

Branching Workflows

Choose a branching workflow that fits your project's complexity and size.

- **Feature branches**: For developing new features.
- **Long-running branches**: Maintain stable and development branches.
- Example of creating and merging a feature branch:

```
\$ git checkout -b new-feature
# work on feature
\$ git commit -a -m "Add new feature"
\$ git checkout master
\$ git merge new-feature
\$ git branch -d new-feature
```

Remote Branches

Working with remote branches involves fetching, pulling, and pushing changes to and from remote repositories.

Remote Branches

Remote branches track the state of branches in remote repositories.

- Fetch updates from a remote:

```
\$ git fetch <remote>
```

- Push a local branch to a remote:

```
\$ git push <remote> <branchname>
```

- Track a remote branch:

```
\$ git checkout --track <remote>/<branchname>
```

Rebasing

Rebasing replays commits from one branch onto another, offering a cleaner project history compared to merging.

Rebasing

Rebase to create a linear commit history and simplify branch integration.

- Basic rebase:

```
\$ git checkout <branchname>
\$ git rebase master

```

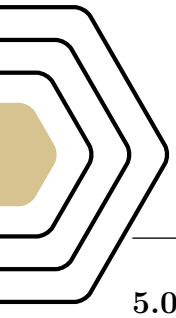
- Avoid rebasing shared branches to prevent history conflicts.

Summary of Key Concepts

Key concepts from Chapter 3 on Git branching:

- **Branches**: Lightweight pointers to commits.
- **Branching and Merging**: Essential for isolated development and integration.
- **Branch Management**: Tools for organizing and maintaining branches.
- **Branching Workflows**: Tailor workflows to project needs.
- **Remote Branches**: Track and manage branches in remote repositories.
- **Rebasing**: Create a linear commit history and streamline integration.

Testing



Testing

5.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- **Engineering Software As A Service - Chapter 8 - Software Testing - Test-Driven Development**

5.0.2 Lectures

The lectures for this week are:

- Introduction To Unit Testing In Python $\approx$ 51 **min**.
- Introduction To Lab 5 $\approx$ 28 **min**.

5.0.3 Assignments

The assignment(s) for this week are:

- Lab 5 - Automated Unit Test

5.0.4 Chapter Summary

The reading for this week is from **Engineering Software As A Service**. The chapter that is being covered this week is **Chapter 8: Software Testing - Test-Driven Development**.

Chapter 8: Software Testing - Test-Driven Development

Overview

This chapter explores Test-Driven Development (TDD) in the context of software testing. It introduces the principles of TDD, the process of writing tests before code, and the benefits of this methodology. By the end of this chapter, you will understand how to apply TDD to create robust, maintainable, and well-documented software.

Background: A RESTful API and a Ruby Gem

A RESTful API, such as TMDb, allows for self-contained HTTP requests to interact with external services. The TMDb gem simplifies the use of this API by constructing the necessary URIs and handling responses.

Background: A RESTful API and a Ruby Gem

Using the TMDb gem to interact with the TMDb API for movie information.

- Construct RESTful URIs with the API key.
- Use the TMDb gem to handle API requests and responses.

FIRST, TDD, and Red-Green-Refactor

The FIRST principles (Fast, Independent, Repeatable, Self-checking, Timely) guide good test creation. TDD involves writing a failing test first, then writing code to pass the test, and finally refactoring the code.

FIRST, TDD, and Red-Green-Refactor

TDD ensures code quality and maintainability through iterative testing and refactoring.

- **Fast**: Quick tests avoid disrupting workflow.
- **Independent**: Tests should not depend on each other.
- **Repeatable**: Consistent results regardless of external factors.
- **Self-checking**: Tests automatically verify results.
- **Timely**: Write tests concurrently with code development.

Seams and Doubles

Seams allow altering program behavior without changing source code. Doubles, such as mocks and stubs, help isolate tests by mimicking the behavior of real objects.

Seams and Doubles

Use seams and doubles to isolate the code under test.

- Seams change program behavior during tests without altering the source code.
- Doubles (mocks and stubs) simulate interactions with other objects.

Expectations, Mocks, Stubs, Setup

RSpec allows setting expectations for method calls and return values, using mocks and stubs to isolate test behavior.

Expectations, Mocks, Stubs, Setup

Set expectations for method interactions and use mocks/stubs to control test behavior.

- **Mocks**: Verify method calls.
- **Stubs**: Control method return values.
- Use `before` blocks for common test setup.

Fixtures and Factories

Fixtures provide a fixed state for tests, while factories create objects dynamically, reducing interdependencies between tests.

Fixtures and Factories

Use fixtures for predefined states and factories for dynamic object creation.

- **Fixtures**: Load predefined objects for testing.
- **Factories**: Create objects with specific attributes as needed.

Implicit Requirements and Stubbing the Internet

Testing external services requires handling implicit requirements and stubbing out service calls to ensure tests are fast and reliable.

Implicit Requirements and Stubbing the Internet

Handle implicit requirements and stub external service calls for reliable testing.

- Identify and test implicit requirements.
- Use stubs to simulate external service responses.

Summary of Key Concepts

Key concepts from Chapter 8 on Test-Driven Development:

- **RESTful API and Ruby Gem**: Simplify API interactions.
- **FIRST Principles**: Guide good test creation.
- **TDD and Red-Green-Refactor**: Iterative testing and refactoring.
- **Seams and Doubles**: Isolate code for testing.
- **Expectations, Mocks, Stubs, Setup**: Control and verify test behavior.
- **Fixtures and Factories**: Manage test objects and dependencies.
- **Implicit Requirements and Stubbing**: Ensure fast, reliable tests for external services.

Flask Web Framework And More Python

Flask Web Framework And More Python

6.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- **Engineering Software As A Service - Chapter 2 - The Architecture Of SaaS Applications**

6.0.2 Lectures

The lectures for this week are:

- Python Decorators ≈ 8 **min**.
- Python Virtual Environments ≈ 10 **min**.
- Flask Quickstart ≈ 12 **min**.
- Creating Tables In HTML ≈ 10 **min**.

The lecture notes for this week are:

- Routes Lecture Notes

6.0.3 Assignments

The assignment(s) for this week are:

- Assignment 6 - Flask Tutorial

6.0.4 Project

The assignment(s) for the project this week is:

- Project Milestone 3: Weekly Status

6.0.5 Chapter Summary

The reading for this week is from **Engineering Software As A Service**. The chapter that is being covered this week is **Chapter 2: The Architecture Of SaaS Applications**.

Chapter 2: The Architecture Of SaaS Applications

Overview

This chapter discusses the architecture of SaaS (Software as a Service) applications, outlining various levels of detail from high-level client-server architecture to specific implementation patterns. Understanding these architectures helps in designing scalable and efficient SaaS applications.

100,000 Feet: Client-Server Architecture

The client-server architecture is a fundamental pattern in SaaS applications, where clients request services and servers respond to these requests.

Client-Server Architecture

Clients and servers communicate over the network, with clients initiating requests and servers responding.

- Clients, such as web browsers, request services.
- Servers, such as WEBrick, respond to client requests.
- Example: Firefox requests a page from WEBrick, which serves content from the RottenPotatoes app.

50,000 Feet: Communication—HTTP and URIs

HTTP and URIs are the foundation of web communication, enabling resource identification and data exchange.

Communication—HTTP and URIs

HTTP (HyperText Transfer Protocol) and URIs (Uniform Resource Identifiers) facilitate client-server communication.

- HTTP is a stateless protocol used for web communication.
- URIs identify resources on the web.
- Example: A browser uses an HTTP GET request to retrieve a web page via its URI.

10,000 Feet: Representation—HTML and CSS

HTML and CSS define the structure and style of web content, respectively, providing a clear separation between content and presentation.

Representation—HTML and CSS

HTML (HyperText Markup Language) and CSS (Cascading Style Sheets) structure and style web content.

- HTML elements structure web pages.
- CSS styles HTML elements.
- Example: An HTML document with embedded CSS defines the layout and appearance of a web page.

5,000 Feet: 3-Tier Architecture Horizontal Scaling

The 3-tier architecture separates concerns into presentation, logic, and persistence tiers, enabling scalability.

3-Tier Architecture Horizontal Scaling

The 3-tier architecture divides applications into presentation, logic, and persistence tiers.

- Presentation tier handles user interfaces.
- Logic tier processes application functionality.
- Persistence tier manages data storage.
- Horizontal scaling allows adding more servers to each tier as needed.

1,000 Feet: Model-View-Controller Architecture

The MVC pattern separates applications into models, views, and controllers, each with distinct responsibilities.

Model-View-Controller Architecture

MVC (Model-View-Controller) pattern divides application logic, user interface, and user input handling.

- Models manage data and business logic.
- Views display data and user interfaces.
- Controllers handle user input and interactions.

500 Feet: Active Record for Models

Active Record is a pattern used to map objects to database records, facilitating data persistence.

Active Record for Models

Active Record maps objects to database tables, enabling CRUD operations.

- Models correspond to database tables.
- CRUD operations: Create, Read, Update, Delete.
- Example: A Movie model maps to a movies table in the database.

500 Feet: Routes, Controllers, and REST

Routes map HTTP requests to controller actions, and RESTful principles ensure stateless and self-descriptive interactions.

Routes, Controllers, and REST

Routes map URIs and HTTP methods to controller actions; REST (Representational State Transfer) organizes resource interactions.

- Routes define URL patterns and corresponding actions.
- RESTful design ensures each request contains all necessary information.
- Example: GET /movies maps to the index action in the Movies controller.

500 Feet: Template Views

Template Views render dynamic content by combining static templates with dynamic data.

Template Views

Template Views generate dynamic content by integrating data into HTML templates.

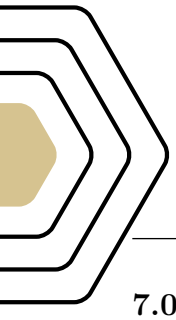
- Templates contain static HTML with placeholders for dynamic data.
- Example: Haml template renders a list of movies dynamically.

Summary of Key Concepts

Key concepts from Chapter 2 on the architecture of SaaS applications:

- **Client-Server Architecture**: Separates clients and servers for specialized roles.
- **Communication—HTTP and URIs**: Facilitates web communication and resource identification.
- **Representation—HTML and CSS**: Structures and styles web content.
- **3-Tier Architecture Horizontal Scaling**: Divides applications into scalable tiers.
- **Model-View-Controller Architecture**: Separates data, presentation, and input handling.
- **Active Record for Models**: Maps objects to database records for data persistence.
- **Routes, Controllers, and REST**: Maps requests to actions and ensures stateless interactions.
- **Template Views**: Generates dynamic content from static templates.

SQL



SQL

7.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- Introduction To Structured Query Language
- NoSQL

7.0.2 Lectures

The lectures for this week are:

- Overview Of Relational Databases ≈ 27 **min**.
- Introduction To SQL ≈ 33 **min**.
- Database SQL API ≈ 16 **min**.
- Introduction To SQL ≈ 140 **min**.
- Single Table Query ≈ 17 **min**.
- Introduction To SQL ≈ 13 **min**.
- Lab 7 - Test Driven Development ≈ 13 **min**.
- Lab 7 - Using Modules In Python ≈ 17 **min**.

7.0.3 Assignments

The assignment(s) for this week are:

- Assignment 7 - SQL And Unit Testing

7.0.4 Project

The assignment(s) for the project this week is:

- Project Milestone 4: Web Pages Design

7.0.5 Chapter Summary

The topic that is being covered this week is **SQL**.

SQL

Overview

SQL (Structured Query Language) is a powerful tool used for managing and manipulating relational databases. It enables users to perform a variety of operations such as creating databases, querying data, updating records, and controlling access to data. Understanding SQL is crucial for anyone working with databases, as it provides the foundation for database management and data analysis.

Basic SQL Commands

SQL commands are the building blocks for interacting with databases. They allow users to create structures within the database, manipulate data, and retrieve information efficiently.

Basic SQL Commands

SQL commands enable the creation, modification, and retrieval of data within databases. These commands are essential for managing database systems effectively.

- **Creating a Database**: This command initializes a new database, which serves as a container for tables and other database objects.

```
CREATE DATABASE my_database;

```

- **Creating a Table**: Tables are the core structures within a database, consisting of rows and columns. This command defines the schema of a table, specifying the data types and constraints for each column.

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    position VARCHAR(50),
    salary DECIMAL(10, 2),
    hire_date DATE
);

```

- **Inserting Data**: This command adds new rows to a table. Each value corresponds to a column in the table.

```
INSERT INTO employees (id, name, position, salary, hire_date)
VALUES (1, 'Alice Smith', 'Developer', 75000.00, '2021-06-15');

```

- **Querying Data**: SQL queries retrieve data from tables based on specified criteria. The `SELECT` statement is used to specify the columns to return and any conditions that must be met.

```
SELECT * FROM employees;

```

- **Updating Data**: This command modifies existing data within a table. The `SET` clause specifies the new values, and the `WHERE` clause defines the conditions for the update.

```
UPDATE employees
SET salary = 80000.00
WHERE id = 1;

```

- **Deleting Data**: This command removes rows from a table based on specified conditions.

```
DELETE FROM employees
WHERE id = 1;

```

Advanced SQL Concepts

As you become more proficient with SQL, you can utilize advanced features to perform complex data manipulations and analyses. These features include joins, aggregations, and subqueries, which enhance your ability to work with multiple tables and datasets.

Advanced SQL Concepts

Advanced SQL techniques allow for sophisticated data manipulation and analysis, enabling users to draw insights from their data efficiently.

- **Joins**: Joins are used to combine rows from two or more tables based on related columns. This is essential when working with normalized databases where data is split across multiple tables.

```
SELECT e.name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.id;

```

- **Aggregations**: Aggregation functions compute a single result from a set of input values. Common functions include `AVG`, `COUNT`, `SUM`, `MIN`, and `MAX`.

```
SELECT AVG(salary) AS average_salary
FROM employees;

```

- **Subqueries**: Subqueries are nested queries within a larger query. They allow you to perform operations in stages, using the result of one query as input to another.

```
SELECT name
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

```

SQL Data Types and Constraints

Data types define the nature of data that can be stored in a column, while constraints enforce rules on the data, ensuring integrity and validity.

SQL Data Types and Constraints

Understanding data types and constraints is crucial for defining robust database schemas that maintain data integrity.

- **Data Types**: SQL provides various data types to store different kinds of data, such as numbers, strings, and dates.
  - **INT**: Integer numbers.
  - **VARCHAR(n)**: Variable-length strings up to `n` characters.
  - **DECIMAL(p, s)**: Fixed-point numbers with precision `p` and scale `s`.
  - **DATE**: Date values.
  - **BOOLEAN**: True/False values.
- **Constraints**: Constraints enforce rules on the data within columns, helping maintain data integrity and prevent errors.
  - **PRIMARY KEY**: Uniquely identifies each row in a table.
  - **FOREIGN KEY**: Enforces referential integrity between tables.
  - **NOT NULL**: Ensures that a column cannot contain NULL values.
  - **UNIQUE**: Ensures all values in a column are distinct.
  - **CHECK**: Validates that values in a column meet a specified condition.

Transactions

Transactions group multiple SQL operations into a single unit of work, ensuring that all operations are completed successfully before changes are committed to the database. This maintains data integrity, especially in scenarios involving multiple, interdependent operations.

Transactions

Transactions ensure that a sequence of operations is completed successfully, maintaining data integrity and consistency.

- **Transaction Example**: In this example, a salary update is applied to all employees in a specific department. The changes are only saved if all operations within the transaction complete successfully.

```
BEGIN;
UPDATE employees SET salary = salary * 1.10 WHERE department_id = 2;
COMMIT;

```

Indexing

Indexes improve the speed of data retrieval operations by creating data structures that allow the database to find rows efficiently. They are particularly useful in queries involving large datasets.

Indexing

Indexes enhance query performance by allowing faster data retrieval, especially in large tables.

- **Creating an Index**: This example creates an index on the `salary` column of the `employees` table, speeding up queries that filter or sort by salary.

```
CREATE INDEX idx_salary ON employees (salary);

```

Comprehensive Query Example

Combining various SQL concepts in a single query allows for powerful data analysis and retrieval, demonstrating the flexibility and capabilities of SQL.

Comprehensive Query Example

This query combines multiple SQL concepts, such as joins, subqueries, and aggregations, to retrieve detailed information from the database.

- **Example**:

```
SELECT e.name, e.position, d.department_name, e.salary
FROM employees e
JOIN departments d ON e.department_id = d.id
WHERE e.salary > (SELECT AVG(salary) FROM employees)
ORDER BY e.salary DESC;

```

This query retrieves the names, positions, department names, and salaries of employees whose salaries are above the average, ordered by salary in descending order.

Best Practices

Adhering to best practices in SQL helps ensure efficient, secure, and maintainable database systems.

Best Practices

Following best practices in SQL development ensures code quality, efficiency, and maintainability.

- Use meaningful and descriptive names for tables and columns.
- Normalize your database to reduce redundancy and improve data integrity.
- Use transactions to ensure that multiple operations succeed together.
- Optimize queries with indexes to enhance performance.
- Regularly back up your database to prevent data loss.

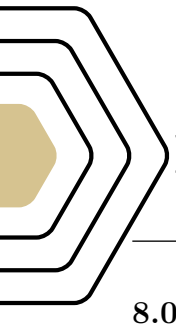
Summary of Key Concepts

Key concepts from the overview of SQL:

- **Basic Commands**: Core operations such as `CREATE`, `SELECT`, `INSERT`, `UPDATE`, and `DELETE` for managing data.
- **Advanced Concepts**: Use of joins, aggregations, and subqueries to handle complex queries and data manipulations.
- **Data Types and Constraints**: Different data types like `INT`, `VARCHAR`, and constraints such as `PRIMARY KEY`, `FOREIGN KEY`, and `NOT NULL` to enforce data integrity.
- **Transactions**: Grouping multiple SQL operations to ensure atomicity and consistency.
- **Indexing**: Enhancing query performance by creating indexes on frequently accessed columns.

- **Best Practices**: Recommendations for meaningful naming conventions, database normalization, using transactions, optimizing queries, and regular backups.

HTML And CSS



HTML And CSS

8.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- Mozilla Web Developer Tutorials
- Full Stack Development
- HTML Tutorial
- CSS Tutorial

8.0.2 Lectures

The lectures for this week are:

- HTML Introduction ≈ 16 **min**.
- Wireframes And Layout ≈ 12 **min**.
- CSS Introduction ≈ 12 **min**.
- Web Forms ≈ 12 **min**.
- Look And Feel For HTML-CSS Menu Page For Lab ≈ 2 **min**.

The lecture notes for this week are:

- CSS Lecture Notes

8.0.3 Assignments

The assignment(s) for this week are:

- Assignment 8 - HTML & CSS

8.0.4 Quiz

The quiz for this week is:

- Quiz 5 - HTML And CSS

8.0.5 Exam

The exam for this week is:

- Exam 1 Notes
- Exam 1

8.0.6 Chapter Summary

The first topic that is being covered this week is **HTML**.

HTML

Overview

HTML (HyperText Markup Language) is the standard markup language used to create web pages. It provides the structure of a web page, allowing developers to define headings, paragraphs, links, images, and other elements. HTML is a foundational technology, alongside CSS and JavaScript, that is used to build modern web applications.

Basic HTML Elements

HTML elements are the building blocks of web pages. Each element is defined by tags, which can contain attributes to provide additional information.

Basic HTML Elements

HTML elements structure web content and provide semantic meaning.

- **Paragraphs and Headings**: Used to define blocks of text.

```
<p>This is a paragraph.</p>
<h1>This is a heading</h1>

```

- **Links**: Used to create hyperlinks to other pages or resources.

```
<a href="https://www.example.com">Visit Example</a>

```

- **Images**: Used to embed images in a web page.

```
<img src="image.jpg" alt="Description of image">

```

HTML Document Structure

An HTML document has a standard structure that includes a doctype declaration, a head section, and a body section.

HTML Document Structure

The structure of an HTML document includes metadata and content.

- **Doctype Declaration**: Defines the document type and version of HTML.

```
<!DOCTYPE html>

```

- **Head Section**: Contains metadata, title, and links to stylesheets or scripts.

```
<head>
    <title>My Web Page</title>
    <link rel="stylesheet" href="styles.css">
</head>

```

- **Body Section**: Contains the content of the web page.

```
<body>
    <h1>Welcome to my website</h1>
    <p>This is a sample web page.</p>
</body>

```

HTML Attributes

Attributes provide additional information about HTML elements, such as classes, IDs, or styles.

HTML Attributes

Attributes add metadata to HTML elements and affect their behavior or presentation.

- **Class Attribute**: Used to assign one or more class names for CSS styling.

```
<p class="intro">This is an introductory paragraph.</p>

```

- **ID Attribute**: Provides a unique identifier for an element.

```
<div id="main-content">Main content here</div>

```

- **Style Attribute**: Allows inline CSS styling.

```
<p style="color: red;">This text is red.</p>

```

HTML Forms

HTML forms are used to collect user input and submit it to a server for processing.

HTML Forms

Forms allow for user interaction and data submission.

- **Form Element**: Encloses form controls and specifies the action URL.

```
<form action="/submit-form" method="post">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name">
    <input type="submit" value="Submit">
</form>

```

- **Input Elements**: Various types of inputs for user data.

```
<input type="text" name="username" placeholder="Enter your name">
<input type="password" name="password" placeholder="Enter your password">
<input type="email" name="email" placeholder="Enter your email">

```

Multimedia in HTML

HTML supports embedding multimedia content such as images, audio, and video.

Multimedia in HTML

Embed multimedia elements to enhance web pages.

- **Image Element**: Embeds an image file.

```
<img src="image.jpg" alt="An example image">

```

- **Audio Element**: Embeds audio content.

```
<audio controls>
    <source src="audio.mp3" type="audio/mpeg">
    Your browser does not support the audio element.
</audio>

```

- **Video Element**: Embeds video content.

```
<video controls>
    <source src="video.mp4" type="video/mp4">
    Your browser does not support the video element.
</video>

```

HTML Tables

HTML tables are used to display tabular data in a structured format.

HTML Tables

Tables organize data into rows and columns for easy readability.

- **Table Element**: Defines the table structure.

```
<table>
    <tr>
        <th>Header 1</th>
        <th>Header 2</th>
    </tr>
    <tr>
        <td>Data 1</td>
        <td>Data 2</td>
    </tr>
</table>

```

Best Practices

Following best practices in HTML ensures that web pages are accessible, maintainable, and performant.

Best Practices

Adhering to best practices improves the quality and usability of web pages.

- Use semantic HTML to improve accessibility and SEO.
- Keep the HTML structure clean and well-organized.
- Use external CSS and JavaScript files to separate content and presentation.
- Validate HTML code to ensure compatibility across different browsers.

Summary of Key Concepts

Key concepts from HTML overview:

- **Basic Elements**: Fundamental HTML tags for structuring content.
- **Document Structure**: Standard layout of an HTML document.
- **Attributes**: Metadata added to elements to modify behavior or presentation.
- **Forms**: Collecting and submitting user input.
- **Multimedia**: Embedding images, audio, and video in web pages.
- **Tables**: Organizing data into rows and columns.
- **Best Practices**: Writing clean, accessible, and maintainable HTML code.

The next topic that is being covered this week is **CSS**.

CSS

Overview

CSS (Cascading Style Sheets) is a stylesheet language used to describe the presentation of a document written in HTML or XML. It enables the separation of document content from document presentation, including layout, colors, and fonts. CSS is essential for creating visually engaging web pages and enhancing the user experience.

Basic CSS Syntax and Selectors

CSS is a rule-based language where rules specify how HTML elements are styled. Each rule consists of a selector and a declaration block.

Basic CSS Syntax and Selectors

CSS rules define how HTML elements should be styled.

- **Selectors**: Determine which HTML elements the styles apply to.
  - **Element Selector**: Targets all instances of an element.

```
p {
    color: red;
}

```

  - **Class Selector**: Targets elements with a specific class attribute.

```
.intro {
    font-size: 16px;
}

```

  - **ID Selector**: Targets a unique element with a specific ID.

```
#main-title {
    text-align: center;
}

```

- **Declaration Block**: Contains one or more declarations separated by semicolons.
  - **Property and Value**: Each declaration includes a CSS property and a value.

```
p {
    color: red;
    font-size: 14px;
}

```

CSS Box Model

The CSS box model is a fundamental concept that defines the rectangular boxes generated for elements in the document tree and governs their dimensions and spacing.

CSS Box Model

The box model includes the element's content, padding, border, and margin.

- **Content**: The actual content of the element.
- **Padding**: Space between the content and the border.

```
.box {
    padding: 20px;
}

```

- **Border**: Surrounds the padding (if any) and content.

```
.box {
    border: 1px solid black;
}

```

- **Margin**: Space outside the border.

```
.box {
    margin: 10px;
}

```

CSS Layout Techniques

CSS provides several layout techniques to control the positioning and alignment of elements on a web page, such as Flexbox, Grid, and positioning properties.

CSS Layout Techniques

Effective layout techniques improve the structure and usability of web pages.

- **Flexbox**: A one-dimensional layout method for arranging items in rows or columns.

```
.container {
    display: flex;
    justify-content: space-between;
}

```

- **Grid**: A two-dimensional layout system for complex layouts.

```
.grid-container {
    display: grid;
    grid-template-columns: auto auto auto;
}

```

- **Positioning**: Controls the position of elements using properties like `static`, `relative`, `absolute`, and `fixed`.

```
.relative {
    position: relative;
    top: 10px;
    left: 20px;
}

```

Styling Text and Fonts

CSS allows for extensive control over text and font properties to enhance the readability and aesthetics of web content.

Styling Text and Fonts

Text and font styling improve the readability and visual appeal of web content.

- **Font Family**: Specifies the font of an element.

```
p {
    font-family: Arial, sans-serif;
}

```

- **Font Size**: Sets the size of the font.

```
        h1 {
            font-size: 2em;
        }

```

- **Text Alignment**: Aligns text horizontally.

```
.center-text {
    text-align: center;
}

```

CSS Animations and Transitions

CSS animations and transitions allow for creating dynamic effects and animations, enhancing user interaction.

CSS Animations and Transitions

Use CSS to add dynamic and interactive elements to web pages.

- **Transitions**: Smoothly change a property value over time.

```
.button {
    transition: background-color 0.3s;
}
.button:hover {
    background-color: blue;
}

```

- **Animations**: Define keyframes for more complex animations.

```
@keyframes example {
    from {background-color: red;}
    to {background-color: yellow;}
}
.animated {
    animation: example 5s infinite;
}

```

Responsive Design with Media Queries

Media queries are used to create responsive web designs that adapt to different screen sizes and devices.

Responsive Design with Media Queries

Media queries allow web pages to adapt to various screen sizes.

- **Media Query Example**: Apply different styles based on screen width.

```
@media (max-width: 600px) {
    .container {
        flex-direction: column;
    }
}

```

Summary of Key Concepts

Key concepts from CSS overview:

- **Basic Syntax and Selectors**: Define how HTML elements are styled.
- **Box Model**: Manage element dimensions and spacing.
- **Layout Techniques**: Arrange elements using Flexbox, Grid, and positioning.
- **Text and Fonts**: Style text for better readability and aesthetics.
- **Animations and Transitions**: Create dynamic and interactive web elements.
- **Responsive Design**: Ensure web pages adapt to different devices using media queries.

The next topic that is being covered this week is **Mozilla Web Developer**.

Mozilla Web Developer

Overview

The MDN Web Docs, formerly known as Mozilla Developer Network, is a comprehensive resource for web developers, providing documentation, tutorials, and guides on HTML, CSS, JavaScript, and various web APIs. It

serves as a valuable tool for both beginners and experienced developers to learn and enhance their web development skills.

Getting Started with Web Development

MDN Web Docs offers a structured approach to learning web development, starting with the basics of HTML and CSS, and progressing to more advanced topics such as JavaScript and server-side programming.

Getting Started with Web Development

MDN Web Docs provides a step-by-step learning path for beginners, helping them build a strong foundation in web development.

- **HTML and CSS**: Learn the basics of structuring web content with HTML and styling it with CSS.

```
<!DOCTYPE html>
<html>
    <head>
        <title>My First Web Page</title>
    </head>
    <body>
        <h1>Hello, world!</h1>
        <p>This is my first web page.</p>
    </body>
</html>

```

- **JavaScript**: Understand the fundamentals of JavaScript to add interactivity to web pages.

```
document.getElementById("demo").innerHTML = "Hello, JavaScript!";

```

Core Topics Covered

MDN Web Docs covers a wide range of core topics essential for web development.

Core Topics Covered

The core topics include detailed guides and references on HTML, CSS, and JavaScript, as well as best practices for web development.

- **HTML**: Learn about HTML elements, attributes, and best practices for structuring web content.
- **CSS**: Explore CSS properties, selectors, and techniques for styling web pages.
- **JavaScript**: Dive into JavaScript syntax, functions, and events to create dynamic web applications.

Advanced Topics and Tools

For those looking to deepen their knowledge, MDN Web Docs provides resources on advanced web development topics and tools.

Advanced Topics and Tools

Advanced topics and tools help developers create sophisticated and efficient web applications.

- **Web APIs**: Learn about various Web APIs for handling multimedia, geolocation, and more.
- **Performance Optimization**: Techniques to improve the performance and responsiveness of web applications.
- **Progressive Web Apps (PWAs)**: Build web applications that provide a native app-like experience.

Practical Applications

MDN Web Docs is not just about theory; it includes practical examples and tutorials to apply what you've learned.

Practical Applications

Practical tutorials and examples enable developers to apply their knowledge in real-world scenarios.

- **Building a Simple Web Page**: Start with a basic HTML structure and progressively enhance it with CSS and JavaScript.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Simple Web Page</title>
        <style>
            body { font-family: Arial, sans-serif; }
            h1 { color: blue; }
        </style>
    </head>
    <body>
        <h1>Welcome to My Website</h1>
        <p>This is a simple web page.</p>
    </body>
</html>

```

- **Creating a To-Do List App**: Use HTML, CSS, and JavaScript to create an interactive to-do list application.

```
<!DOCTYPE html>
<html>
    <head>
        <title>To-Do List</title>
        <style>
            ul { list-style-type: none; padding: 0; }
            li { margin: 5px 0; }
        </style>
    </head>
    <body>
        <h1>To-Do List</h1>
        <input type="text" id="taskInput" placeholder="New task...">
        <button onclick="addTask()">Add</button>
        <ul id="taskList"></ul>

        <script>
            function addTask() {
                var taskInput = document.getElementById('taskInput');
                var taskList = document.getElementById('taskList');
                var newTask = document.createElement('li');
                newTask.textContent = taskInput.value;
                taskList.appendChild(newTask);
                taskInput.value = '';
            }
        </script>
    </body>
</html>

```

Best Practices

MDN Web Docs emphasizes best practices in web development to ensure code quality, performance, and accessibility.

Best Practices

Following best practices helps create high-quality, performant, and accessible web applications.

- Use semantic HTML to improve accessibility and SEO.
- Optimize CSS for performance and maintainability.
- Write clean and efficient JavaScript code.
- Implement responsive design to ensure compatibility across devices.

Summary of Key Concepts

Key concepts from MDN Web Docs for utilizing web development resources:

- **Getting Started**: Fundamental skills in HTML, CSS, and JavaScript for beginners.
- **Core Topics**: Detailed guides on essential web technologies.

- **Advanced Topics**: In-depth resources on Web APIs, performance optimization, and PWAs.
- **Practical Applications**: Tutorials and examples for real-world projects.
- **Best Practices**: Guidelines for writing high-quality, efficient, and accessible code.

The last topic that is being covered this week is **Full Stack Development**.

Full Stack Development

Overview

Full Stack Development refers to the practice of developing both the front-end (client-side) and back-end (server-side) portions of a web application. Full stack developers possess a broad skill set that allows them to manage and develop all parts of a software application, from the user interface to the database and server infrastructure. This comprehensive approach to development ensures a seamless and cohesive user experience while maintaining robust server-side operations.

Front-End Development

The front-end, or client-side, is where users interact with the application. Full stack developers use various technologies to build engaging and responsive user interfaces.

Front-End Development

Full stack developers utilize HTML, CSS, and JavaScript to create the client-facing part of web applications.

- **HTML (Hypertext Markup Language)**: Structures the content on the web page.

```html
<!DOCTYPE html>
<html>
    <head>
        <title>My Web Page</title>
    </head>
    <body>
        <h1>Welcome to my website</h1>
        <p>This is an example of HTML.</p>
    </body>
</html>

```

- **CSS (Cascading Style Sheets)**: Styles the HTML content to make it visually appealing.

```css
body {
    font-family: Arial, sans-serif;
}
h1 {
    color: blue;
}

```

- **JavaScript**: Adds interactivity and dynamic behavior to web pages.

```javascript
document.getElementById("demo").innerHTML = "Hello, JavaScript!";

```

Back-End Development

The back-end, or server-side, handles the application logic, database interactions, user authentication, and server configuration.

Back-End Development

Full stack developers use server-side languages and frameworks to build the backbone of web applications.

- **Server-Side Languages**: Commonly used languages include Python, Java, PHP, and Node.js.
  - **Python Example**:

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, Flask!"

if __name__ == '__main__':
    app.run(debug=True)

```

- **Database Management**: Managing and interacting with databases using SQL or NoSQL databases like MySQL, PostgreSQL, or MongoDB.
  - **SQL Example**:

```sql
CREATE TABLE users (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100)
);

INSERT INTO users (id, name, email)
VALUES (1, 'John Doe', 'john@example.com');

```

Full Stack Development Frameworks

Frameworks simplify the development process by providing reusable components and tools for both front-end and back-end development.

Full Stack Development Frameworks

Popular frameworks provide integrated tools to streamline full stack development.

- **Ruby on Rails**: A full-stack framework that uses Ruby for building both the front-end and back-end.
- **Django**: A high-level Python framework that encourages rapid development and clean design.
- **Spring Boot**: A Java framework for building production-ready applications quickly.
- **Laravel**: A PHP framework known for its elegant syntax and robust set of tools.

Popular Stacks in Full Stack Development

A stack is a set of technologies used together to build a full application. Each stack includes an operating system, web server, database, and programming language.

Popular Stacks in Full Stack Development

Different technology stacks are used for various application requirements and developer preferences.

- **LAMP Stack**: Linux, Apache, MySQL, PHP
- **MEAN Stack**: MongoDB, Express.js, Angular, Node.js
- **MERN Stack**: MongoDB, Express.js, React, Node.js

Best Practices

Following best practices in full stack development ensures code quality, maintainability, and performance.

Best Practices

Adhering to best practices helps create efficient and scalable web applications.

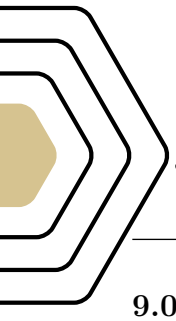
- Use version control systems like Git for code management.
- Write clean, modular, and reusable code.
- Implement automated testing to ensure code reliability.
- Optimize for performance and scalability.
- Maintain thorough documentation for all parts of the application.

Summary of Key Concepts

Key concepts from Full Stack Development:

- **Front-End Development**: Building user interfaces with HTML, CSS, and JavaScript.
- **Back-End Development**: Handling server-side logic, database interactions, and application workflows.
- **Frameworks**: Utilizing frameworks like Ruby on Rails, Django, Spring Boot, and Laravel to streamline development.
- **Technology Stacks**: Choosing the right stack (e.g., LAMP, MEAN, MERN) based on project needs.
- **Best Practices**: Ensuring code quality, performance, and maintainability through established best practices.

JavaScript



JavaScript

9.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- JavaScript Tutorial Point

9.0.2 Lectures

The lectures for this week are:

- Introduction To JavaScript ≈ 16 **min**.
- JavaScript Syntax And Practice ≈ 14 **min**.

9.0.3 Assignments

The assignment(s) for this week are:

- Assignment 9 - JavaScript

9.0.4 Quiz

The quiz for this week is:

- Quiz 6 - JavaScript

9.0.5 Project

The assignment(s) for the project this week is:

- Project Milestone 5: SQL Design

9.0.6 Chapter Summary

The topic that is being covered this week is **JavaScript**.

JavaScript

Overview

JavaScript is a versatile and powerful programming language commonly used in web development to create dynamic and interactive user experiences. It allows developers to implement complex features on web pages, such as real-time updates, interactive maps, animations, and much more. JavaScript is a core technology of the web, alongside HTML and CSS.

Basic JavaScript Syntax and Functions

JavaScript syntax is influenced by Java and C, making it familiar to developers with experience in those languages. It supports various programming paradigms, including procedural, object-oriented, and functional programming.

Basic JavaScript Syntax and Functions

JavaScript syntax forms the foundation for writing scripts to control web page behavior.

- **Variables**: Used to store data values.

```
let name = "Alice";
const pi = 3.14159;

```

- **Functions**: Blocks of code designed to perform a particular task.

```
function greet() {
    console.log("Hello, world!");
}
greet();

```

DOM Manipulation

The Document Object Model (DOM) is an API for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content.

DOM Manipulation

JavaScript interacts with the DOM to dynamically update web content.

- **Selecting Elements**: Access elements in the DOM.

```
const element = document.getElementById("myElement");

```

- **Changing Content**: Modify the content of elements.

```
element.textContent = "New content";

```

- **Event Listeners**: Respond to user actions.

```
element.addEventListener("click", function() {
    alert("Element clicked!");
});

```

JavaScript Objects and Prototypes

JavaScript is an object-oriented language. Objects are collections of related data and functionality, and prototypes allow for the inheritance of properties and methods.

JavaScript Objects and Prototypes

Objects and prototypes are key concepts for organizing and reusing code.

- **Creating Objects**: Define objects using literal notation.

```
let person = {
    name: "Alice",
    age: 25,
    greet: function() {
        console.log("Hello, " + this.name);
    }
};
person.greet();

```

- **Prototypes**: Share properties and methods across instances.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.greet = function() {
    console.log("Hello, " + this.name);
};
```

```
let bob = new Person("Bob", 30);
bob.greet();

```

Asynchronous JavaScript

Asynchronous programming is essential in JavaScript for handling operations that take time to complete, such as data fetching or file reading, without blocking the execution of other code.

Asynchronous JavaScript

Asynchronous techniques improve the responsiveness of web applications.

- **Callbacks**: Functions passed as arguments to other functions.

```
function fetchData(callback) {
    setTimeout(() => {
        callback("Data received");
    }, 1000);
}
fetchData(data => {
    console.log(data);
});

```

- **Promises**: Objects representing the eventual completion or failure of an asynchronous operation.

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Data received");
    }, 1000);
});
promise.then(data => {
    console.log(data);
});

```

- **Async/Await**: Syntactic sugar over promises for writing asynchronous code.

```
async function fetchData() {
    let data = await new Promise((resolve) => {
        setTimeout(() => {
            resolve("Data received");
        }, 1000);
    });
    console.log(data);
}
fetchData();

```

JavaScript Modules

Modules allow you to break up your code into reusable pieces. They are essential for maintaining clean and manageable codebases.

JavaScript Modules

Use modules to organize and encapsulate code.

- **Exporting and Importing**: Share functionality between files.

```
// In math.js
export function add(a, b) {
    return a + b;
}
// In main.js
import { add } from './math.js';
console.log(add(2, 3));

```

Summary of Key Concepts

Key concepts from JavaScript overview:

- **Syntax and Functions**: Basic building blocks of JavaScript.
- **DOM Manipulation**: Interacting with and modifying the document structure.
- **Objects and Prototypes**: Organizing code and enabling inheritance.
- **Asynchronous Programming**: Handling time-consuming tasks without blocking the main thread.
- **Modules**: Structuring code into reusable components.

Hosting A Public Web Site

Hosting A Public Web Site

10.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- **N/A**

10.0.2 Lectures

The lectures for this week are:

- Introduction To Cloud Computing ≈ 25 **min**.
- How To Deploy A Flask App And Postgres Database To Render ≈ 5 **min**.
- How To Deploy A Flask App And Postgres Database To Render (Full Video) ≈ 17 **min**.

10.0.3 Assignments

The assignment(s) for this week are:

- Assignment 10 - Website Hosting Tutorial

10.0.4 Chapter Summary

The topic that is being covered this week is **Cloud Computing**.

Cloud Computing

Overview

Cloud Computing refers to the delivery of computing services over the internet, enabling users to access and manage data and applications remotely. This technology provides on-demand availability of computing resources like servers, storage, databases, and networking, without the need for direct management by the user. Cloud computing is essential for businesses of all sizes, offering scalability, cost-efficiency, and flexibility.

Types of Cloud Computing Services

Cloud computing services are generally categorized into three main types: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

Types of Cloud Computing Services

Different types of cloud services cater to varying levels of control and management, suitable for diverse business needs.

- **Infrastructure as a Service (IaaS)**: Provides virtualized computing resources over the internet. IaaS includes essential services such as virtual machines, storage, and networking. It offers high flexibility and control over IT resources, allowing businesses to scale as needed.

```
        Use case: Hosting a website or a large-scale application that requires flexible
    resource allocation.

```

- **Platform as a Service (PaaS)**: Offers a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the underlying infrastructure. PaaS includes services like development tools, database management, and business analytics.

```
1           Use case: Developing a web application with pre-configured environments for testing and
    deployment.
2
```

- **Software as a Service (SaaS)**: Delivers software applications over the internet, on a subscription basis. SaaS providers manage the infrastructure and platforms that run the applications, which are accessible via a web browser.

```
1           Use case: Using CRM software or email services.
2
```

Benefits of Cloud Computing

Cloud computing offers numerous benefits that have revolutionized how businesses and individuals interact with technology.

Benefits of Cloud Computing

The advantages of cloud computing include:

- **Cost Efficiency**: Reduces the need for physical infrastructure and maintenance, converting capital expenses to operational expenses. Users pay only for the resources they use, which can lead to significant cost savings.
- **Scalability and Flexibility**: Allows businesses to scale resources up or down based on demand, ensuring optimal resource utilization and avoiding over-provisioning.
- **Global Reach**: Deploy applications globally within minutes, allowing companies to reach customers around the world and improve latency and user experience.
- **Security and Compliance**: Leading cloud providers offer robust security measures and compliance certifications, helping businesses meet regulatory requirements and protect data integrity.
- **Innovation and Agility**: Cloud services enable rapid development and deployment of new applications and services, fostering innovation and reducing time-to-market.

Key Cloud Computing Models

Cloud computing models vary based on deployment and service delivery, offering different levels of management and control.

Key Cloud Computing Models

The primary models include:

- **Public Cloud**: Resources are owned and operated by a third-party cloud service provider and delivered over the internet. Public clouds are ideal for workloads with varying or unpredictable demand.
- **Private Cloud**: Dedicated infrastructure for a single organization, offering more control and security. It can be hosted on-premises or by a third-party provider.
- **Hybrid Cloud**: Combines public and private clouds, allowing data and applications to move between them. This model offers greater flexibility and optimization of existing infrastructure, security, and compliance requirements.

Applications of Cloud Computing

Cloud computing supports a wide range of applications across various industries, enhancing efficiency and innovation.

Applications of Cloud Computing

Key applications include:

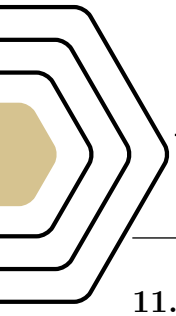
- **Big Data Analytics**: Processing large datasets to derive business insights and improve decision-making.
- **Web Hosting**: Hosting websites and applications with scalable and reliable infrastructure.
- **Disaster Recovery and Backup**: Protecting data and applications from disruptions and ensuring business continuity.
- **AI and Machine Learning**: Running complex AI models and algorithms with scalable compute resources.
- **IoT (Internet of Things)**: Connecting and managing devices and sensors, processing data in real-time.

Summary of Key Concepts

Key concepts from Cloud Computing:

- **Types of Services**: IaaS, PaaS, and SaaS each offer different levels of control and management.
- **Benefits**: Includes cost efficiency, scalability, global reach, security, and innovation.
- **Cloud Models**: Public, private, and hybrid clouds cater to different organizational needs.
- **Applications**: Encompasses big data analytics, web hosting, disaster recovery, AI, and IoT.

Web Services



Web Services

11.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- Cloud Computing Introduction

11.0.2 Lectures

The lectures for this week are:

- Web API And Protocols ≈ 24 **min**.
- JSON And XML ≈ 18 **min**.

11.0.3 Assignments

The assignment(s) for this week are:

- Assignment 11 - REST Weather Map

11.0.4 Chapter Summary

The first topic that is being covered this week is **Web API And Protocols**.

Web API And Protocols

Overview

Web APIs (Application Programming Interfaces) are a set of defined protocols and tools that allow different software applications to communicate over the internet. They provide a way for developers to interact with web services and integrate various functionalities into their applications. Web APIs are integral to modern web development, enabling the connection and interaction of disparate systems and services.

Types of APIs and Protocols

APIs can be categorized based on their architecture and the protocols they use. The most common types include REST, SOAP, GraphQL, and WebSocket.

Types of APIs and Protocols

Different API protocols offer various advantages and are suitable for different use cases.

- **REST (Representational State Transfer)**: REST APIs use standard HTTP methods (GET, POST, PUT, DELETE) and are widely used for their simplicity and scalability. They allow clients to interact with resources via URIs and standard HTTP status codes.

```
GET /users/123

```

- **SOAP (Simple Object Access Protocol)**: SOAP is a protocol that uses XML to encode its messages and is often used in enterprise environments. It supports a wide range of transport protocols, including HTTP and SMTP.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <GetUser>
            <UserID>123</UserID>
        </GetUser>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

- **GraphQL**: A query language for APIs, GraphQL allows clients to request only the data they need, reducing the number of requests required. It uses a single endpoint for all operations.

```
{
    user(id: "123") {
    name
    email
    }
}

```

- **WebSocket**: This protocol provides full-duplex communication channels over a single TCP connection, enabling real-time interaction between a client and a server.

```
const socket = new WebSocket("wss://example.com/socket");
socket.onmessage = function(event) {
    console.log("Received data: " + event.data);
};

```

Benefits of Web APIs

Web APIs offer several key benefits, including ease of integration, enhanced functionality, and support for a wide range of devices and platforms.

Benefits of Web APIs

Web APIs provide numerous advantages that enhance application development and user experience.

- **Integration**: APIs enable the integration of new applications with existing systems, facilitating faster development and deployment of new features.
- **Innovation**: By leveraging APIs, businesses can quickly adapt to new technologies and trends, fostering innovation without needing to overhaul entire systems.
- **Scalability**: APIs can support a wide range of devices and platforms, making it easier to scale applications across different environments.
- **Ease of Maintenance**: Using APIs allows for modular code, where individual components can be updated or replaced without affecting the entire system.

Common Use Cases of Web APIs

APIs are used in a variety of applications, from integrating third-party services to enhancing user experiences with interactive features.

Common Use Cases of Web APIs

Web APIs support a broad range of applications across different domains.

- **Integrating with Third-Party Services**: Examples include payment processing, social media integration, and map services.
- **Enhancing User Interaction**: APIs enable features like real-time notifications, live data feeds, and user authentication.
- **Data Exchange and Automation**: APIs facilitate the automated transfer of data between systems, reducing manual work and improving efficiency.

Security and API Management

Security is a critical aspect of API management, ensuring that data is protected and access is controlled.

Security and API Management

Proper API security and management are essential for protecting data and ensuring reliable service.

- **Authentication and Authorization**: Use of tokens and API keys to verify the identity of users and restrict access to certain resources.
- **Monitoring and Logging**: Tracking API usage and performance to detect and respond to potential issues.
- **Rate Limiting**: Implementing limits on the number of API requests to prevent abuse and ensure fair usage.

Summary of Key Concepts

Key concepts from Web APIs and Protocols:

- **Types of APIs and Protocols**: REST, SOAP, GraphQL, and WebSocket, each with unique characteristics and use cases.
- **Benefits of APIs**: Including integration, innovation, scalability, and ease of maintenance.
- **Common Use Cases**: Integration with third-party services, enhancing user interaction, and data automation.
- **Security**: Critical measures like authentication, monitoring, and rate limiting.

The last topic that is being covered this week is **JSON And XML**.

JSON And XML

Overview

JSON (JavaScript Object Notation) and XML (eXtensible Markup Language) are two popular data interchange formats used in web development and other applications. Both are text-based and designed to facilitate data exchange between systems. However, they have different structures and use cases, making them suitable for various scenarios depending on the requirements.

JSON

JSON is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is based on a subset of the JavaScript programming language and is language-independent.

JSON

JSON structures data in a collection of name/value pairs and arrays, making it an ideal format for transmitting data between a server and a web application.

- **Data Structures**: JSON supports simple data structures such as objects (name/value pairs) and arrays.

```
{
    "name": "John Doe",
    "age": 30,
    "isStudent": false,
    "courses": ["Math", "Science", "History"]
}

```

- **Human Readability**: JSON's structure is easy to understand and write, which simplifies debugging and development.
- **Compatibility**: JSON is compatible with most programming languages, making it versatile for various applications.

XML

XML is a markup language that defines rules for encoding documents in a format that is both human-readable and machine-readable. It is more verbose than JSON but provides greater flexibility through its use of custom tags.

XML

XML is designed to store and transport data, emphasizing the self-descriptive nature of the data and the flexibility of defining custom tags.

- **Structure**: XML documents are structured with a root element, child elements, attributes, and text content.

```
<person>
    <name>John Doe</name>
    <age>30</age>
    <isStudent>false</isStudent>
    <courses>
        <course>Math</course>
        <course>Science</course>
        <course>History</course>
    </courses>
</person>

```

- **Extensibility**: XML allows the definition of custom tags, making it highly extensible and suitable for complex data structures.
- **Use Cases**: XML is widely used in systems where data needs to be validated against a schema (XSD) or when the data interchange requires rich data descriptions.

Comparison and Use Cases

Both JSON and XML are widely used for data interchange, but they have distinct advantages and are suited for different scenarios.

Comparison and Use Cases

When choosing between JSON and XML, consider the specific requirements of your project, including data complexity, readability, and interoperability needs.

- **JSON Advantages**: Simplicity, ease of use, faster parsing, and native support in JavaScript.
- **XML Advantages**: Extensive support for complex data structures, schema validation, and data transformation capabilities (XSLT).
- **Typical Use Cases**: JSON is commonly used in web APIs and data transmission in web applications, while XML is often used in document storage, configuration files, and data interchange in enterprise applications.

Summary of Key Concepts

Key concepts from JSON and XML overview:

- **JSON**: Lightweight, easy to read/write, ideal for data interchange in web applications.
- **XML**: Flexible, supports complex data structures, used in applications requiring rich data descriptions and validation.
- **Comparison**: JSON is simpler and more efficient for most web applications, while XML offers more extensive capabilities for document-based data.

Documentation And Requirements

Documentation And Requirements

12.0.1 Assigned Reading

The reading for this week is from, Agile For Dummies, Engineering Software As A Service - An Agile Approach Using Cloud Computing, Pro Git, and The Linux Command Line.

- Documentation Types

12.0.2 Lectures

The lectures for this week are:

- Documentation ≈ 33 **min**.
- Software Requirements ≈ 16 **min**.

12.0.3 Assignments

The assignment(s) for this week are:

- Assignment 12 - Documentation

12.0.4 Project

The assignment(s) for the project this week is:

- Project Milestone 6: Asynchronous Interview
- Project Milestone 7: Presentation

12.0.5 Exam

The exam for this week is:

- Exam 2 Notes
- Exam 2

12.0.6 Chapter Summary

The topic that is being covered this week is **Documentation**.

Documentation

Overview

Documentation in software development refers to the comprehensive set of materials that explain how software operates or how to use it. Effective documentation is crucial for the success of any software project, ensuring that both developers and end-users can understand and effectively utilize the software. It encompasses a range of documents, including technical manuals, user guides, API references, and more, each serving different purposes and audiences.

Types of Software Documentation

There are several types of software documentation, each tailored to a specific aspect of the software development lifecycle.

Types of Software Documentation

Software documentation is categorized based on its audience and purpose.

- **System Documentation**: Includes architectural diagrams and technical specifications that describe the overall structure and design of the software system. It is essential for developers to understand how the system is built and maintained.
- **API Documentation**: Provides detailed information on how to interact with software components via APIs, including methods, parameters, and example calls.

```
1           \texttt{GET /users/\{id\}} - Retrieves the user with the specified ID.
2
```

- **User Documentation**: Guides end-users on how to install, configure, and use the software. This can include user manuals, tutorials, FAQs, and how-to guides.
- **Source Code Documentation**: Embedded within the source code, this documentation explains the purpose, logic, and usage of specific code sections. It is invaluable for ongoing maintenance and future development.

Best Practices for Creating Documentation

Creating high-quality software documentation involves several best practices that ensure clarity, accuracy, and accessibility.

Best Practices for Creating Documentation

Adhering to best practices in documentation enhances its effectiveness and usability.

- **Understand the Audience**: Tailor the documentation to the needs and skill levels of the intended audience, whether they are developers, end-users, or stakeholders.
- **Keep It Simple and Clear**: Use plain language, avoid unnecessary jargon, and organize the content logically to make it easily understandable.
- **Maintain Consistency**: Ensure that the documentation is consistent in style, terminology, and format across all sections.
- **Update Regularly**: Documentation should be updated continuously to reflect changes in the software, such as new features or bug fixes.
- **Incorporate Visual Aids**: Use diagrams, screenshots, and examples to enhance understanding and break down complex concepts.

Common Challenges in Documentation

Despite its importance, creating and maintaining software documentation can present several challenges.

Common Challenges in Documentation

Overcoming documentation challenges is crucial to maintaining effective communication and ensuring software usability.

- **Time Constraints**: Documentation often competes with development tasks for time and resources, leading to incomplete or outdated documentation.
- **Keeping It Updated**: As software evolves, ensuring that documentation remains accurate and up-to-date can be difficult.
- **Ensuring Accessibility**: Documentation needs to be easy to find and use, which can be a challenge when multiple documents are scattered across different platforms.

- **Engaging the Right Expertise**: High-quality documentation requires input from developers, technical writers, and subject matter experts, which can be hard to coordinate.

Tools for Creating Documentation

Several tools can aid in the creation and management of software documentation, providing features that enhance collaboration, organization, and accessibility.

Tools for Creating Documentation

Utilizing the right tools can streamline the documentation process and improve its quality.

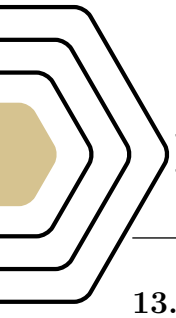
- **Markdown and HTML Support**: Tools like GitHub Pages and ReadMe.io allow for the easy creation and publication of documentation using Markdown and HTML.
- **API Documentation Tools**: Tools such as Swagger and Postman help automate the creation of API documentation, ensuring accuracy and ease of use.
- **Version Control Integration**: Integrating documentation with version control systems like Git ensures that documentation evolves alongside the codebase.
- **Collaboration Platforms**: Tools like Confluence and Notion facilitate team collaboration on documentation projects, allowing multiple contributors to work simultaneously.

Summary of Key Concepts

Key concepts from software documentation:

- **Types of Documentation**: Includes system, API, user, and source code documentation, each serving different audiences and purposes.
- **Best Practices**: Emphasizes clarity, consistency, regular updates, and understanding the audience.
- **Challenges**: Involves time constraints, keeping documentation updated, ensuring accessibility, and engaging the right expertise.
- **Tools**: Markdown, HTML, API documentation tools, version control, and collaboration platforms enhance the documentation process.

Finals



Finals

13.0.1 Project

The assignment(s) for the project this week is:

- Project Milestone 8: Final Report Submission
- Project Milestone 9: Peer Evaluation And Project Reflection
- Project Milestone 10: Team Participation And Contribution