

11 Why manual deployment is painful , Algorithmic thinking , And importance of git, and configuration of GIT

Project configuration

create a server installed programming language downloaded code/application created directories, users installed dependencies start application

project configuration

Certainly! Here's a breakdown of the project configuration using the steps you've provided:

1. **Create a Server:** This is typically the first step in setting up a project. A server can be a physical machine or a virtual instance hosted on a cloud platform like AWS, Google Cloud, or Azure. The server provides the infrastructure where your application will run.
2. **Installed Programming Language:** Choose and install the programming language(s) you'll use for developing your application. This could be languages like Python, JavaScript (Node.js), Java, C#, etc. Ensure that the necessary runtime environments and compilers are set up properly on your server.
3. **Downloaded Code/Application:** Obtain the source code or application files for your project. This could involve cloning a Git repository, downloading a ZIP file from a website, or any other method of obtaining the necessary codebase.
4. **Created Directories, Users:** Organize your project files by creating directories (folders) for different components of your application. This might include directories for source code, configuration files, static assets, etc. Additionally, create user accounts with appropriate permissions for accessing and managing the project files.
5. **Installed Dependencies:** Install any libraries, frameworks, or dependencies required by your application. This could involve using package managers like npm (for Node.js), pip (for Python), Maven (for Java), or NuGet (for .NET). Ensuring all dependencies are installed correctly is crucial for the proper functioning of your application.
6. **Start Application:** Once everything is set up, start the application to test its functionality. This might involve running a command to start a server, executing a script, or launching an executable file. Monitor the application to ensure it starts without errors and behaves as expected.

This configuration process ensures that your project is set up correctly with the necessary infrastructure, dependencies, and resources to run your application smoothly. It's an essential part of the development lifecycle that lays the foundation for building and deploying your project.

deployment of application / pushing an update

Sure, here's a step-by-step explanation of how the deployment of a new application version over an existing one is typically done:

1. **Deployment/Release:** The new version of the application is prepared for deployment by the development team. This includes implementing new features, fixing bugs, and making any necessary changes to the codebase.
2. **Developer Pushes New App/Code:** Once the new version is ready, the developer pushes the updated code to the version control system, such as Git. This ensures that the latest changes are accessible to all team members and can be deployed to the production environment.
3. **Stop the Server:** Before deploying the new version, the server running the existing application is stopped. This ensures that there are no conflicts or issues arising from simultaneous access to the application during the deployment process. Users may experience a brief interruption in service during this step.
4. **Remove/Backup the Old Code:** The old codebase is either removed or backed up to ensure that it can be restored in case of any unforeseen issues with the deployment of the new version. This step helps to maintain data integrity and rollback capabilities.
5. **Download New Code:** The updated code for the new version is downloaded onto the server. This could involve pulling the latest changes from the version control system or copying files from a designated deployment package.
6. **Start the Server:** Once the new code is in place, the server is restarted to launch the updated application. This step brings the new version online and makes it accessible to users. The server configuration may need to be adjusted to accommodate any changes introduced in the new version.
7. **Post-Deployment Checks:** After the server is restarted, various checks and tests are performed to ensure that the deployment was successful. This may include smoke testing, regression testing, and monitoring system logs for any errors or abnormalities.
8. **Rollback Plan:** In case any issues are detected post-deployment, there should be a rollback plan in place to revert to the previous version quickly. This could involve restoring the backed-up code or rolling back changes in the version control system.

By following these steps, the deployment of a new application version over an existing one can be carried out smoothly and efficiently, minimizing downtime and ensuring a seamless transition for users.

Manual deployment is a tedious and error-prone process. It is also time-consuming and costly.

Manual deployment of software

wood choppin anology

Certainly! Let's illustrate the manual deployment process using the analogy of wood chopping, incorporating the points you've mentioned:

Manual Deployment Process: Wood Chopping

In the old days, before modern machinery, wood chopping was a manual process that involved significant effort and time. Let's compare it to the manual deployment of software:

1. **Downtime:** Just as chopping wood manually required time, manual deployment often results in downtime for the application. This downtime means users may not have access to the service during the deployment process, leading to potential frustration and loss of productivity.
2. **Time Consuming:** Manual deployment is time-consuming. It can take up to 30 minutes or more to complete the deployment process manually, especially if there are multiple steps involved, such as stopping the server, removing old code, and deploying the new version.
3. **Manual Mistakes:** Similar to how manual wood chopping can lead to inconsistent wood sizes or uneven cuts, manual deployment can result in mistakes. Human error during the deployment process, such as misconfigurations or incorrect commands, can lead to issues with the application's functionality.
4. **Troubleshooting:** When issues arise during manual deployment, troubleshooting becomes necessary. Just like trying to figure out why a wood chopping axe isn't working properly, troubleshooting deployment errors can be time-consuming and complex, requiring expertise to identify and resolve.
5. **Identifying Mistakes:** It's often challenging to pinpoint who made mistakes during manual deployment. Without proper documentation or tracking, it's difficult to hold individuals accountable for errors that occur during the process.
6. **Repeated Work:** Manual deployment often involves repetitive tasks, such as copying files or executing commands on multiple servers. This repetition can lead to inefficiencies and wasted effort, as the same steps need to be performed each time a new deployment is initiated.

Now, with modern technology, the deployment process has evolved. Just as the introduction of machinery like lorries streamlined the transportation of wood, automation and DevOps practices have revolutionized software deployment. By leveraging tools and scripting, manual tasks can be automated, reducing downtime, minimizing errors, and improving overall efficiency in the deployment process. However, transitioning to automated deployment methods may require a learning curve, akin to learning how to drive a lorry, but the benefits in terms of speed, reliability, and repeatability are significant.

Manual deployment of software issues

Certainly! Let's illustrate the manual deployment process using the analogy of wood chopping, incorporating the points you've mentioned:

Manual Deployment Process: Wood Chopping

In the manual deployment process, imagine deploying software is akin to chopping wood by hand:

1. **Downtime:** Just as chopping wood manually requires time and effort, manual deployment results in downtime for the application. During deployment, the application may need to be taken offline, causing interruptions to users' access.
2. **Time Consuming:** Manual deployment is time-consuming. It can take up to 30 minutes or more to complete all the necessary steps, including stopping the server, removing old code, deploying the new version, and restarting the server.
3. **Manual Mistakes:** Similar to the potential for mistakes while chopping wood manually, manual deployment can lead to errors. Human error, such as misconfigurations or typos in commands, can occur during the deployment process, leading to issues with the application's functionality.

4. **Troubleshooting:** When errors occur during manual deployment, troubleshooting becomes necessary. Just as you might need to troubleshoot a dull axe or a snagged saw while chopping wood, troubleshooting deployment errors involves identifying the source of the problem and rectifying it, which can be time-consuming and complex.
5. **Identifying Mistakes:** It's often challenging to determine who made mistakes during manual deployment. Without proper documentation or tracking, it's difficult to attribute errors to specific individuals, making it challenging to address accountability and prevent recurring mistakes.
6. **Repeated Work:** Manual deployment often involves repetitive tasks, such as copying files or executing commands on multiple servers. This repetition can lead to inefficiencies and wasted effort, as the same steps need to be performed each time a new deployment is initiated.

In summary, manual deployment, much like chopping wood by hand, is a labor-intensive process prone to downtime, errors, and inefficiencies. Transitioning to automated deployment methods can help streamline the process, reduce downtime, minimize errors, and improve overall efficiency in software deployment.

coding --> learn scripting

programming --> java, .net, python scripting --> shell, python, ansible, terraform, kubernetes, etc.

coding --> patterns and methods

variables data types conditions functions loops

algorithm thinking

1. 10 public sessions, 11th session private
2. first we need to get today's session no
3. if today's session is greater than 10, we need to say it is private
4. less than 10 public

today session = 5

if (today session) > 10 print("private") else print("public")

install git through scripting

1. check user have sudo access or not
2. if not sudo, we will throw error,
3. if sudo proceed
4. install package git
5. we need to check explicitly installed or not
6. if error installation throw the error to user, otherwise say success.

coding vs scripting

Here's a breakdown of notes and a detailed response, combining the best aspects of clarity, structure, and additional insights for your coding learning journey.

Notes

- **Coding vs. Scripting:**
 - **Coding:** Building software from scratch using programming languages like Java, .NET, Python. Focus on patterns, methods, data structures, and algorithms.
 - **Scripting:** Automating tasks within existing systems, using languages like Shell, Python, Ansible, Terraform, Kubernetes.
- **Coding Fundamentals:**
 - **Variables:** Placeholders for storing data.
 - **Data Types:** Classify data (e.g., numbers, text, booleans).
 - **Conditions:** Control code execution based on true/false evaluations (e.g., `if`, `else`).
 - **Functions:** Reusable blocks of code.
 - **Loops:** Repeat actions multiple times (e.g., `for`, `while`).
- **Algorithmic Thinking:**
 - Problem-solving approach in programming
 - Break down problems into smaller, sequential steps.

Long Detailed Answer

Understanding Coding, Scripting, and Algorithms

Coding and scripting are both essential tools for software development but serve different purposes.

- **Coding** is the foundation of software development. It involves using programming languages with well-defined structures to build applications from the ground up. You focus on good design patterns, efficient algorithms, and creating a solid software architecture.
- **Scripting** simplifies things by automating tasks and integrating different systems. Scripting languages often have a less rigid structure, making them easier to learn and use for quick solutions. Think of them as the glue that connects the parts of your software system.

Coding Fundamentals in Action: YouTube Example

Your YouTube playlist example demonstrates core coding concepts:

- **Variables:** `todaySession` stores the current session number.
- **Conditions:** The `if` statement checks if `todaySession` exceeds the public limit.
- **Output:** Using `print` to display "public" or "private."

Algorithmic Thinking for Installing Git

The git installation steps outline an algorithm, a step-by-step problem-solving process computers excel at:

1. **Check Permissions:** Ensure the user has the necessary privileges to install software.
2. **Install Package:** Use the system's package manager to install git.

3. **Verify Installation:** Confirm success to avoid issues later.
4. **Handle Errors:** Provide clear feedback to the user if anything goes wrong.

Why Machines Lack Common Sense

Computers are brilliant at following explicit instructions (algorithms), but they struggle with common sense. Here's why:

- **Context:** Humans understand the world through context and nuanced meanings. Machines rely on literal data without awareness of broader implications.
- **Ambiguity:** Language is imprecise. Computers need extremely clear definitions, while humans can interpret things flexibly.
- **Experience:** Common sense arises from a vast store of real-world experiences and knowledge that machines don't naturally possess.

Key Takeaways

- Coding and scripting are complementary skills that power modern software.
- Master the coding fundamentals for building robust applications.
- Think algorithmically to break down any problem for a computer to solve.
- Understand a computer's limitations to design effective solutions.

Let me know if you'd like to explore any of these topics in more depth or have other coding questions!

scripts are stored in git

shell scripting

where to keep the scripts???

10 devops engineers working on same project

one central location to do and share the things

Git --> global information tracking system

version control --> 10 days

after 10 days --> I wanted to see what was script at 2nd day..

we should always go back to previous version if something goes wrong

track the changes --> through git we can understand who did that and why

review the changes --> our team members should review

backup and security

branching --> create a branch from master, do the changes, if everything is good then proceed

collaboration

GIT is a popular decentralised version control system

decentralised/distributed vs centralised

git repo --> store the code

repository --> some storage just like folder with some extra capabilities

Notes: Shell Scripting

- **Where to Keep the Scripts?**

- Consider one central location for storing and sharing scripts.

- **Collaboration**

- With 10 DevOps engineers working on the same project, centralizing scripts ensures coordination and collaboration.

- **Git: Global Information Tracking System**

- Git facilitates version control, crucial for tracking changes and reverting to previous versions if necessary.

- **Version Control**

- Essential for tracking changes over time; after 10 days, you can review scripts from any point in the project's history.

- **Review and Collaboration**

- Git enables team members to review changes, fostering collaboration and ensuring quality.

- **Backup and Security**

- Git provides backup and security features, safeguarding scripts and project data.

- **Branching**

- Utilize branching in Git: create a branch from the master, make changes, and merge back if successful.

- **Decentralized vs. Centralized**

- Git is a decentralized version control system, allowing flexibility and efficiency in collaboration.

- **Git Repository**

- Acts as a storage space for code, akin to a folder with added capabilities for version control and collaboration.

how this works in realtime

Certainly! Let's explain why Git is useful based on the provided context:

Shell scripting is a powerful tool for automating tasks, but it raises questions about where to store and manage the scripts. This becomes especially crucial when multiple DevOps engineers are working on the same

project.

With 10 DevOps engineers collaborating on a project, having a central location to create and share scripts becomes essential. Git serves as a global information tracking system, providing version control capabilities.

Version control ensures that changes made to scripts are tracked over time. For instance, if 10 days have passed and you want to refer back to a script from the 2nd day, Git allows you to easily access previous versions. This ability to revert to earlier versions is crucial for maintaining stability and reliability, particularly if something goes wrong with the current version.

Moreover, Git enables tracking of changes at a granular level, allowing you to understand who made specific changes and why. This transparency promotes accountability and facilitates collaboration among team members. Additionally, it allows team members to review each other's changes, ensuring code quality and consistency.

Backup and security are also significant advantages of using Git. By storing scripts in a Git repository, you automatically create backups of your codebase, reducing the risk of data loss. Git's branching feature allows you to create separate branches from the main codebase (e.g., master branch), enabling parallel development and experimentation without affecting the main codebase. Once changes are tested and validated in a branch, they can be merged back into the main codebase.

In summary, Git provides a robust platform for managing shell scripts in a collaborative environment. It offers version control, change tracking, accountability, backup, security, branching, and collaboration features, making it an indispensable tool for DevOps teams working on automation projects.

create git account 29:00 42:47

~ --> your home folder

git process

git init --> initialises a folder as git repo

git status --> we can know what is the status of our files

git add -->

100 scripts

signup, login, forgot password

temp area --> select what is completed and push to git

signup --> add to it temp area and push to git

git config --global user.name "your-user-name" git config --global user.email "your-email-id"

git commit -m "as part of bug fix" --> till here local...

Notes: Staging and Git Commands

- **Staging Area in Git**

- Utilize the staging area in Git to select and prepare files for commit.

- **Git Init**

- Initialize a folder as a Git repository using `git init`.

- **Git Status**

- Determine the status of files in the repository using `git status`.

- **Git Add**

- Add files to the staging area using `git add`, allowing selective inclusion of changes.

- **Committing Changes**

- After staging changes, commit them to the repository using `git commit -m "commit message"`. This records the changes locally.

- **User Configuration**

- Configure Git with user information using:

```
git config --global user.name "your-user-name"  
git config --global user.email "your-email-id"
```

- **Staging Scripts**

- Staging allows for organized management of scripts. For instance, with 100 scripts, only completed features like signup, login, and forgot password can be staged and pushed to Git.

- **Local Committing**

- After staging, commit changes locally with a descriptive message, such as "as part of bug fix", using `git commit -m "message"`. This step preserves the changes within the local repository.

git completed at 56:16

assignment

setup git and configure along with vscode

01:08:00 end of class Q and A