INFOPRESS

Practical DEVOPS and Cloud Engineering

Best Practices Workshop
Version Control & Microservices

คู่มือ Upskill & Reskill สำหรับคนที่อยากเป็น Cloud DevOps Engineer หนึ่งในอาชีพดาวรุ่งด้านเทคโนโลยีดิจิตอล

SOURCE CODE : SERAZU.COM

ผู้แต่ง อ.ดร.ณัฐโชติ พรหมฤทธิ์
บรรณาธิการ ภีรพล คชาเจริญ

Practical DevOps and Cloud Engineering

Writer	ดร.ณัฐโชติ พรหมฤทธิ์
Editor	ภีรพล คชาเจริญ
Graphic Designers	ชวนันท์ รัตนะ, กมลชนก ชัยสาครสมุทร
Page Layout	สิริลักษณ์ วาระเลิศ
Proofreaders	สุนทรี บรรลือศักดิ์, เกษรา พรวัฒนมงคล
Publishing Coordinators	วรพล ณธิกุล, สุพัตรา อาจปรุ, วัชรพงศ์ ยงปัญญาสกุล

บริษัท ไอดีซี พรีเมียร์ จำกัด จัดตั้งขึ้นเพื่อเผยแพร่ความรู้ที่มีคุณภาพสู่ผู้อ่านชาวไทย เรายินดีรับงานเขียนของนักวิชาการและนักเขียนทุกท่าน ท่านผู้สนใจกรุณาติดต่อผ่านทางอีเมลที่ infopress@idcpremier.com หรือทางโทรศัพท์หมายเลข 0-2962-1081 (อัตโนมัติ 10 คู่สาย) โทรสาร 0-2962-1084

สร้างสรรค์โดย

INFOPRESS

พิมพ์และจัดจำหน่ายโดย

IDC PREMIER

พิมพ์ครั้งที่ 1 ธันวาคม 2563

ข้อมูลทางบรรณานุกรม
ดร.ณัฐโชติ พรหมฤทธิ์
Practical DevOps
and Cloud Engineering
นนทบุรี : ไอดีซีฯ, 2563
432 หน้า

1. การเขียนโปรแกรมโดยใช้ภาษาโปรแกรมเฉพาะชนิด
I ชื่อเรื่อง
005.262
Barcode 885-916-101-081-4
ราคา 545 บาท

บริษัท ไอดีซี พรีเมียร์ จำกัด
200 หมู่ 4 ชั้น 19 ห้อง 1901
จัสมินอินเตอร์เนชั่นแนลทาวเวอร์
ถ.แจ้งวัฒนะ อ.ปากเกร็ด จ.นนทบุรี 11120
โทรศัพท์ 0-2962-1081 (อัตโนมัติ 10 คู่สาย)
โทรสาร 0-2962-1084

สมาชิกสัมพันธ์
โทรศัพท์ 0-2962-1081-3 ต่อ 121
โทรสาร 0-2962-1084

ร้านค้าและตัวแทนจำหน่าย
โทรศัพท์ 0-2962-1081-3 ต่อ 112-114
โทรสาร 0-2962-1084

PREFACE

การสร้างและดูแลรักษาซอฟต์แวร์ มีความแตกต่างจากการสร้างและดูแลรักษาผลิตภัณฑ์ประเภทอื่น เนื่องจากธรรมชาติของผลิตภัณฑ์ที่เป็นซอฟต์แวร์ มักมีความเปลี่ยนแปลงเกิดขึ้นบ่อย และต้องแข่งขันกับคู่แข่ง รวมทั้งมีการปรับเปลี่ยนไปตามเสียงสะท้อนของผู้ใช้งาน ซึ่งโดยส่วนใหญ่การปรับเปลี่ยนจะเป็นไปในลักษณะทั้งที่มีการแก้ไขปัญหาเก่าและเพิ่มเติมความสามารถใหม่ๆ โดยที่ยังคงให้บริการได้ในขณะที่มีการปรับเปลี่ยนเพื่อไม่ให้ผู้ใช้รู้สึกติดขัด

และเพื่อให้การทำงานของระบบหรือซอฟต์แวร์เป็นไปโดยราบรื่น สามารถรองรับการเปลี่ยนแปลงหรือเพิ่มเติมได้ตลอดเวลา จึงต้องอาศัยทั้งทักษะในแง่ของการพัฒนา (Development) และการดูแล (Operation) รวมทั้งต้องใช้เครื่องมือที่มีความทันสมัยในการบริหารจัดการ จึงทำให้บุคคลที่มีความรู้ความสามารถในเชิง DevOps หรือ DevOps Engineer เป็นกำลังคนที่ตลาดแรงงานมีความต้องการสูงในปัจจุบัน ซึ่งหน้าที่หลักๆ ของ DevOps Engineer คือ เป็นผู้เขียน CI/CD Pipeline Script, Dockerfile, docker-compose.yml วางสภาพแวดล้อมในการพัฒนาโปรแกรม รวมทั้งคอยดูแล Infrastructure และ Config Cluster (เช่น Docker Swarm และ Kubernetes) เป็นต้น

นอกจากนี้ DevOps Engineer ยังเป็นผู้ Monitor ระบบหรือซอฟต์แวร์ที่ Deploy บน Cloud Server ซึ่งเมื่อพบปัญหาจะต้องรีบแจ้ง Developer ให้หาทางแก้ไขอย่างทันท่วงที ดังนั้น หน้าที่ของ DevOps Engineer คือ ให้การสนับสนุนการทำงานของ Developer ในทำนองเดียวกันกับ System Admin แต่มีขอบเขตกว้างกว่าการทำหน้าที่ในงาน System Operation ตามรูปแบบเดิม

จากความต้องการกำลังคนในงานทางด้าน DevOps Engineer ดังกล่าว ผมจึงได้หารือกับ อ.ดร.สัจจาภรณ์ ไวจรรยา อดีต IT Development Department Manager บริษัท TARAD Dot Com Group Co., Ltd. และ Operational Readiness Team Lead (Manager) บริษัท Total Access Communication PLC (DTAC) และคุณชาญศิลป์ ชื้นประเสริฐ CTO Nipa.cloud ในการออกแบบเนื้อหาของหนังสือเล่มนี้ เพื่อให้ผู้อ่านได้มองเห็นภาพรวมของงานทางด้าน DevOps และหลังจากการฝึกปฏิบัติและทำ Workshop แล้ว หวังว่าผู้อ่านจะมีทักษะขั้นพื้นฐานที่สามารถนำไปใช้กับการทำงานในภาคอุตสาหกรรมดิจิตอลได้จริง

อ.ดร.ณัฐโชติ พรหมฤทธิ์
หัวหน้าภาควิชาคอมพิวเตอร์
คณะวิทยาศาสตร์ มหาวิทยาลัยศิลปากร

EDITOR'S NOTE

เมื่อเทคโนโลยีใหม่ๆ เกิดขึ้นตลอดเวลา ความรู้และทักษะที่ได้รับจากภาคการศึกษา จึงมักจะมีความล่าช้ากว่าในภาคอุตสาหกรรม ที่มีการปรับตัวให้ทันสมัยอยู่เสมอ โดยเฉพาะในภาคอุตสาหกรรมการพัฒนาผลิตภัณฑ์ซอฟต์แวร์ ซึ่งเป็นวงการหนึ่งที่มีการเปลี่ยนแปลงอย่างรวดเร็ว โดยเฉพาะซอฟต์แวร์ที่มีรูปแบบเป็น Platform เช่น E-Commerce Platform, Streaming Service Platform และ Social Network Platform

หนังสือ Practical DevOps and Cloud Engineering เล่มนี้ เป็นคู่มือพัฒนาทักษะทางด้าน DevOps and Cloud Engineering ภาคปฏิบัติ มีจุดประสงค์เพื่อพัฒนาทักษะ 2 ด้านคือ "Reskill" (การเสริมทักษะใหม่เพื่อรองรับอนาคต) และ "Upskill" (การพัฒนาทักษะที่มีอยู่ให้ดีขึ้นตามที่บริษัทต้องการ) โดยเน้นการพัฒนาซอฟต์แวร์ผ่าน Practice & Workshop ภายใต้แนวคิดใหม่อย่าง "Microservices Architecture" (Design-Develop-Deploy) เพื่อให้ผู้อ่านมีแนวทางที่จะนำความรู้และทักษะที่ได้จากหนังสือเล่มนี้ไปประยุกต์ใช้ได้ในการทำงานจริง

เพราะฉะนั้น หนังสือเล่มนี้จึงเหมาะสำหรับนักศึกษา โปรแกรมเมอร์ นักพัฒนาซอฟต์แวร์ และผู้ที่อยากข้ามสายงานมาทำงานทางด้าน "DevOps Engineer/Cloud DevOps Engineer" สายงานที่มีบทบาทสำคัญต่อการพัฒนาผลิตภัณฑ์ที่เป็นซอฟต์แวร์ และเป็นที่ต้องการมากที่สุดอาชีพหนึ่งในปัจจุบัน

หวังเป็นอย่างยิ่งว่า หนังสือเล่มนี้จะทำหน้าที่เป็นสะพานเชื่อม หรือลดช่องว่างระหว่างภาคการศึกษา ซึ่งเป็นผู้ผลิตบุคลากรป้อนตลาดแรงงาน กับภาคธุรกิจอุตสาหกรรมซอฟต์แวร์ที่ต้องการบุคลากรที่มีทักษะเหมาะกับงาน

ภีรพล คชาเจริญ
บรรณาธิการ

CONTENTS

CONTENTS

CONTENTS

บทนำ
(Introduction)

Who is A DevOps Engineer? และ What does a DevOps Engineer do? ผู้เขียนเชื่อว่านี่คือ 2 คำถามแรกที่ผู้อ่านคงอยากทราบคำตอบก่อนจะเริ่มศึกษาหนังสือเล่มนี้ ซึ่งหากดูตำแหน่งงานในปัจจุบันจาก Google และ JobsDB จะเห็นว่า DevOps Engineer เป็นตำแหน่งที่กำลังมาแรงมาก และมีค่าตอบแทนที่สูงมาก บทนี้เราจะมาหาคำตอบกันว่า DevOps Engineer คือใคร ทำหน้าที่อะไร และต้องมีทักษะด้านไหนบ้าง ตลอดจน Concept ของการจัดทำหนังสือเล่มนี้

DevOps Engineer คืออะไรกันแน่?

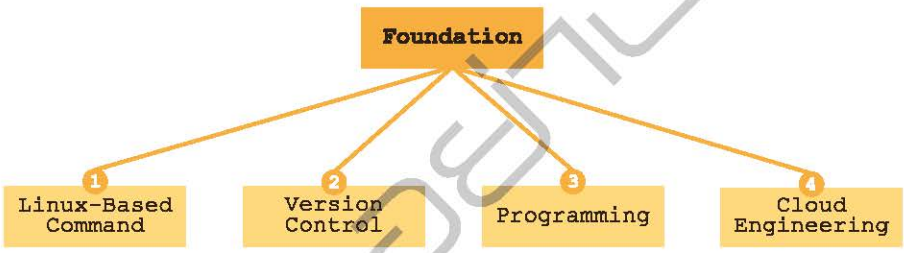
DevOps คือ วัฒนธรรมในการทำงานที่ทำให้นักพัฒนาซอฟต์แวร์ (Software Developer) และคนทำ Infrastructure (Operation) ทำงานร่วมกันโดยมีเป้าหมายเพื่อให้องค์กรมีความพร้อม และสามารถนำระบบงานขึ้น Production ได้อย่างรวดเร็ว ไม่ใช่แค่การมองเฉพาะเป้าหมายของแต่ละทีมเป็นหลัก เหมือนในสมัยก่อนที่ Dev Team จะมองในเรื่องความรวดเร็วของการพัฒนาซอฟต์แวร์ ที่ต้องส่งมอบให้ตรงเวลา และ Operation Team จะมองที่ความเสถียรของระบบ

ดังนั้น DevOps Engineer ก็คือ คนที่เข้ามาสนับสนุนและทำให้รูปแบบการทำงานดังที่กล่าวมาเป็นจริงแบบองค์รวม ทำให้เกิดทีมที่มีรูปแบบการทำงานที่มีการผสมผสานหน้าที่ระหว่าง Development และ Operation (DevOps Team) ที่สามารถดูแล Infrastructure สร้าง Environment และ Deploy ซอฟต์แวร์ รวมถึงสามารถ Monitor และแก้ปัญหาต่างๆ ที่เกิดขึ้นบนระบบงานจริงได้ด้วยตนเอง ซึ่งสอดคล้องกับวิถีการพัฒนาซอฟต์แวร์แบบ "อไจล์ (Agile)"

Basic Skill ที่ DevOps Engineer ต้องมี

เพื่อให้สามารถเอาระบบงานขึ้น Production ได้อย่างรวดเร็วในแบบอัตโนมัติ และลดความผิดพลาดในการทำงาน หลายองค์กรจึงเลือกที่จะพัฒนาซอฟต์แวร์บน Docker Container และเอาระบบงานขึ้นบน Cloud Server โดยใช้แนวทาง CI/CD Pipeline ซึ่ง Pipeline จะถูกกระตุ้นให้ทำงานเมื่อมีการ Push Source Code ไปยัง Version Control Server อย่างเช่น GitLab หรือ GitHub ฯลฯ

ดังนั้น DevOps Engineer จะต้องมีทักษะอย่างน้อยใน 4 ด้าน ได้แก่ Linux-based Command, Version Control, Programming และ Cloud Engineering



Book Concept

เพื่อให้ผู้สนใจได้เข้าใจแนวคิดของ DevOps อย่างลึกซึ้ง และเป็นการเตรียมความพร้อมสำหรับการเป็น DevOps Engineer ที่กำลังเป็นตำแหน่งงานที่เนื้อหอมที่สุดตำแหน่งหนึ่งในปัจจุบันได้ในเวลาที่ไม่นานนัก ผู้เขียนจึงได้จัดทำหนังสือเล่มนี้ขึ้นในลักษณะของ Practice & Workshop ที่มีการสอดแทรกแนวคิดสำคัญๆ ทั้งในเรื่องของ Version Control, Git, Docker Container, VPS และ Let's Encrypt, การพัฒนา Microservice, API Gateway, การพัฒนาระบบ OTP และ Session Server, การทำ Load Test, การทำ Unit Test, Docker Swarm และการคอนฟิก CI/CD Pipeline ฯลฯ โดยมีการแบ่งเนื้อหาออกเป็น 2 ส่วน ดังนี้

Part 1 บทที่ 1-10 จะเป็นการฝึกปฏิบัติ (Practice) การใช้งาน Git และ Docker ให้คล่องตัว

Part 2 บทที่ 11-16 จะเป็นการทำเวิร์คช็อป (Workshop) ผู้อ่านจะได้ทดลองนำระบบงานขึ้นบน Linux Cloud Server และเรียกใช้งานด้วย Domain Name Server จริง (ผู้เขียนเลือกใช้ Nipa.Cloud และ Google Domains เนื่องจากความสะดวกในการคอนฟิก) ในส่วนหลังนี้จะมีการพัฒนาระบบงานทะเบียนนักศึกษาบนสถาปัตยกรรมแบบ Microservices ด้วยภาษา Python ซึ่งเป็นภาษาเขียนโปรแกรม (Programming Language) ที่ได้รับความนิยมอย่างสูงในปัจจุบัน

สำหรับ Spec ของ Cloud Server จำนวน 3 VM ขณะที่มีการเขียนหนังสือเล่มนี้ ได้แก่

1) Ubuntu Server 18.04

2) VCPU 2 CPU

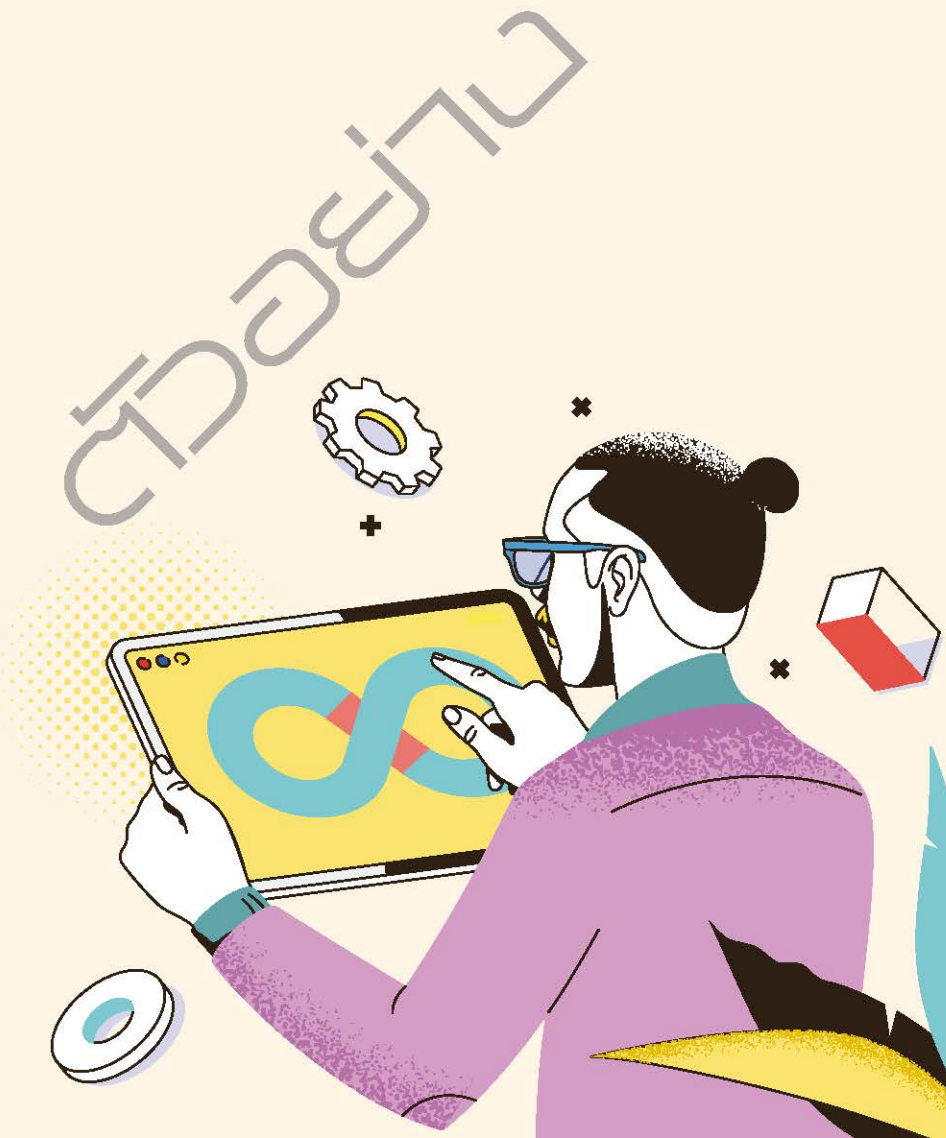
3) RAM 8 GB

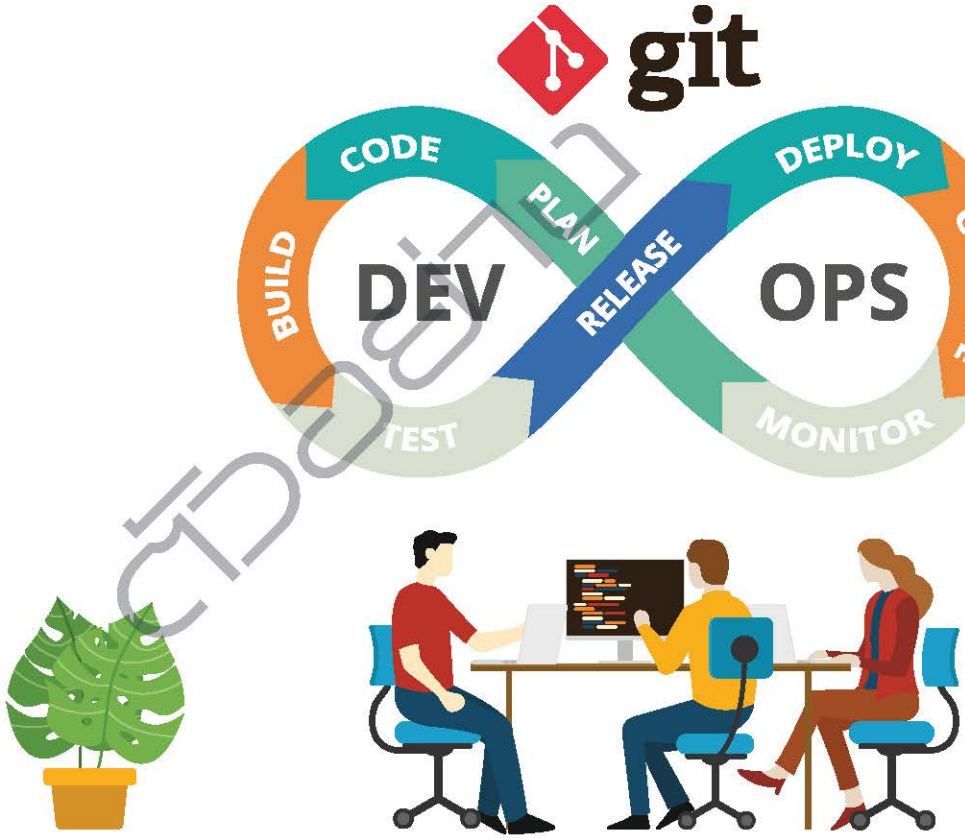
4) SSD 80 GB

อย่างไรก็ตาม ผู้อ่านสามารถทำ Workshop ได้โดยใช้ Linux Cloud Server เพียง 1 VM ก็เพียงพอแล้ว

เนื่องจากในการฝึกปฏิบัติและทำเวิร์คช็อปในหลายๆ บทของหนังสือเล่มนี้ จะมีการคอนฟิกระบบบน Docker Container และ Deploy งานขึ้น Linux Cloud Server ดังนั้น ผู้อ่านที่ใช้งาน Linux-Based Command ในระดับเบื้องต้นได้ ก็จะสามารถศึกษาทำความเข้าใจเนื้อหาในแต่ละบทได้อย่างราบรื่นขึ้น

อย่างไรก็ตาม สำหรับผู้อ่านที่ไม่มีพื้นฐานการใช้งาน Linux-Based Command มาก่อน ก็น่าจะสามารถพิมพ์ Code ตามตัวอย่างในหนังสือได้อย่างไม่ติดขัดครับ โดยท่านสามารถดาวน์โหลด Source Code ในแต่ละบทได้จากเว็บไซต์ serazu.com




git
CODE
PLAN
BUILD
DEV
RELEASE
DEPLOY
OPERATE
OPS
TEST
MONITOR

CHAPTER

01

แนวคิดการจัดเก็บเวอร์ชัน

(Version Control Concepts)

ในการพัฒนาซอฟต์แวร์ให้แล้วเสร็จ จนสามารถนำมาใช้งานได้อย่างสมบูรณ์นั้น จำเป็นจะต้องผ่านกระบวนการมากมายหลายขั้นตอน สิ่งที่พบได้ในระหว่างพัฒนาคือ การแก้ไขหรือปรับปรุงซอร์สโค้ดของทีมพัฒนาซอฟต์แวร์ ฉะนั้น การจัดเก็บประวัติเวอร์ชันของซอร์สโค้ดจึงเป็นเรื่องสำคัญ ทั้งนี้ เพื่อให้ทีมพัฒนาฯ สามารถติดตามความเปลี่ยนแปลงที่เกิดขึ้นได้ตลอดการทำงาน นับตั้งแต่เริ่มต้นไปจนถึงได้ซอฟต์แวร์ที่สมบูรณ์คือใช้งานได้จริง มีข้อบกพร่องน้อยที่สุด

การจัดเก็บเวอร์ชันในการพัฒนาซอฟต์แวร์

หากพูดถึง การจัดเก็บเวอร์ชัน (Version Control) โดยเฉพาะ Git หลายคนคงเคยได้ยินกันมาบ้าง และถ้าใครไม่อยากเชยก็น่าจะเรียนรู้ไว้ เพราะ Version Control ถือว่าเป็นสิ่งจำเป็นสำหรับการพัฒนาซอฟต์แวร์ (Software Development) เปรียบได้กับปัจจัย 4 ที่จำเป็นต่อการดำรงชีวิตของมนุษย์เลยทีเดียว

Version Control หรือ Source Control หมายถึง เครื่องมือช่วยติดตามการเปลี่ยนแปลงของซอร์สโค้ด (Source Code) โดยการเก็บประวัติการเปลี่ยนแปลงลงฐานข้อมูลชนิดพิเศษ ซึ่งจุดประสงค์ของการเก็บบันทึกทุกการเปลี่ยนแปลงก็คือ หากมีความผิดพลาดเกิดขึ้น ไม่ว่าจะเป็นความผิดพลาดเพียงเล็กน้อย ไปจนถึงความผิดพลาดขั้นร้ายแรงที่อาจทำให้ซอฟต์แวร์ที่กำลังพัฒนาอยู่นั้นพังทั้งโปรเจ็กต์ ซึ่งเหตุการณ์ทำนองนี้เป็นเรื่องปกติ การบันทึกประวัติเก็บไว้อย่างต่อเนื่อง จะช่วยให้นักพัฒนาสามารถย้อนกลับไปยังช่วงเวลาต่างๆ เพื่อเปรียบเทียบโค้ดใหม่กับโค้ดเวอร์ชันก่อนหน้า และตรวจสอบความผิดพลาดเพื่อแก้ไขให้มีผลกระทบต่อทีมน้อยที่สุด

นอกจากนี้ Version Control ยังเป็นเครื่องมือหลักของทีมพัฒนาแบบ DevOps ที่ทำให้เกิดความต่อเนื่องในการพัฒนาซอฟต์แวร์ ช่วยให้นักพัฒนาสามารถ Deploy Software วันละหลายครั้งได้จริงในทางปฏิบัติ เพียงการ Push ซอร์สโค้ดกลับไปยัง Git เซิร์ฟเวอร์

ถ้าจะเทียบให้เห็นภาพจากเรื่องใกล้ตัว Version Control ก็คงคล้ายกับคำสั่ง Undo/Redo ในโปรแกรม Microsoft Word ที่ช่วยเรียกคืนเวอร์ชันของเอกสารได้ แต่หากผู้ใช้สั่งบันทึกแล้วปิดโปรแกรม Microsoft Word เมื่อเรียกไฟล์ขึ้นมาแก้ไขอีกครั้ง ผู้ใช้จะไม่สามารถย้อนกลับไปยังเวอร์ชันก่อนหน้าได้อีก

แนวคิดการจัดเก็บเวอร์ชัน

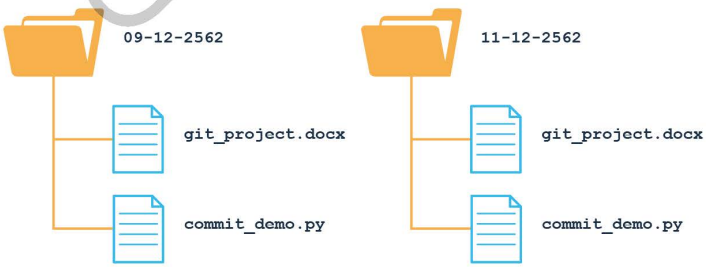
ในบทนี้ผู้อ่านจะได้เข้าใจแนวคิดและความเป็นมาของ Version Control ในยุคแรกๆ จนมาถึง Git ซึ่งเป็น Version Control ยอดนิยมในยุคนี้

การจัดเก็บเวอร์ชันด้วยวิธีก็อปปี้ (Copy File & Folder)

การก็อปปี้ไฟล์และโฟลเดอร์คือ วิธีการพื้นฐานแรกเริ่มที่เราสามารถจัดการได้ เป็นลักษณะของการบันทึกเอกสารแยกออกเป็นหลายๆ ไฟล์ แล้วตั้งชื่อไฟล์เพื่อระบุเวอร์ชันตามลำดับ เช่น

```
git_project.docx
git_project_v2.docx
git_project_back.docx
git_project_final.docx
git_project_lastest.docx
```

แต่จะเห็นว่า การตั้งชื่อไฟล์อาจทำให้ผู้ใช้สับสนในภายหลังว่า ไฟล์ไหนใหม่กว่ากัน รวมทั้งไม่สามารถบอกได้ว่า ไฟล์ไหนถูกสร้างต่อยอดมาจากไฟล์ไหนบ้าง เพราะฉะนั้น เพื่อไม่ให้เกิดความสับสนเรื่องเวอร์ชันของเอกสาร จึงมีการแก้ปัญหานี้โดยการจัดเก็บไฟล์แยกเป็นโฟลเดอร์ตามวันที่ เช่นตัวอย่างในรูปที่ 1.1

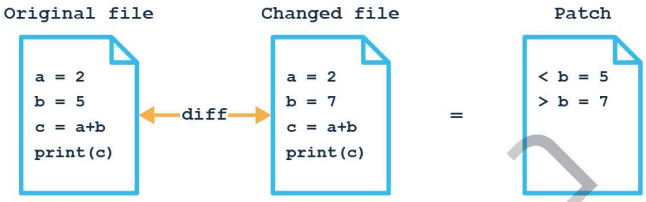


รูปที่ 1.1 ตัวอย่างโครงสร้างโฟลเดอร์ตามวันที่เพื่อช่วยจัดการเวอร์ชัน (Control Version)

แต่ข้อเสียของวิธีการนี้ก็คือ จะทำให้สิ้นเปลืองเนื้อที่การจัดเก็บข้อมูลตามจำนวนเวอร์ชันของเอกสารที่เกิดขึ้น แม้ว่าจะมีการเปลี่ยนแปลงในเอกสารเพียงไม่กี่บรรทัด ก็ต้องคัดลอกไฟล์ทั้งโฟลเดอร์เพื่อสร้างเอกสารเวอร์ชันใหม่ขึ้นมา และบอกไม่ได้ว่าเวอร์ชันนั้นๆ มีความสำคัญขนาดไหน

การจัดเก็บเวอร์ชันด้วยวิธีแพตช์ (Patch)

เพื่อหลีกเลี่ยงความสิ้นเปลืองพื้นที่ในการจัดเก็บของวิธี Copy File & Folder เราอาจเก็บเวอร์ชันของเอกสารเป็นแบบ Patch แทน เหมือนกับการเก็บเวอร์ชันของซอร์สโค้ดของระบบปฏิบัติการ Linux ในยุคแรกๆ ดังรูปที่ 1.2

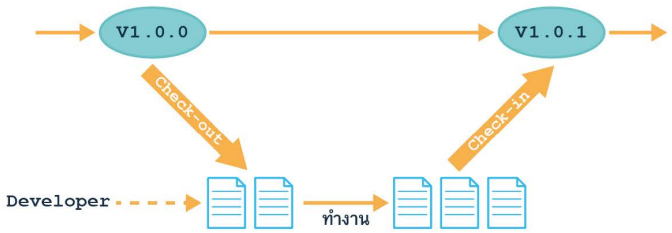


รูปที่ 1.2 ตัวอย่างการจัดเก็บเอกสารเป็น Patch

จากภาพด้านบนแสดงตัวอย่างของ Patch ที่เกิดจากการเปรียบเทียบไฟล์เก่า (Original file) และไฟล์ใหม่ (Changed file) Patch ที่สร้างจะแสดงถึงบรรทัดที่มีการเปลี่ยนแปลง โดยอาจใช้เครื่องหมาย < นำหน้าบรรทัดที่หายไป และ > นำหน้าบรรทัดที่เพิ่มมาแทน อย่างไรก็ตาม การเก็บเวอร์ชันในรูป Patch อาจมีความเสี่ยงต่อการทำ Patch บาง Patch หาย จนทำให้ไม่สามารถประกอบร่างซอร์สโค้ดเพื่อกลับไปยังเวอร์ชันต่างๆ ได้

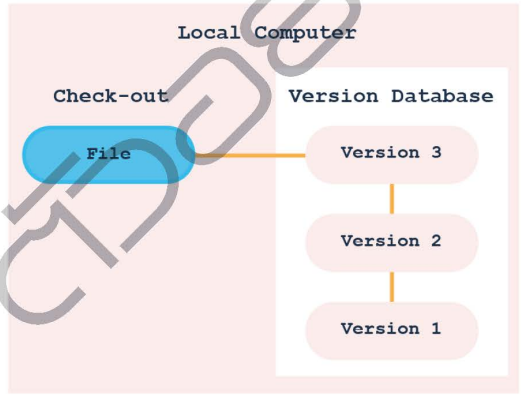
Local Version Control System

เพื่อแก้ปัญหา Patch หายจนไม่สามารถประกอบร่างซอร์สโค้ดได้ จึงมีการพัฒนา Version Control System (VCS) ที่มีฐานข้อมูลเฉพาะคอยจัดเก็บทุกการเปลี่ยนแปลงของซอร์สโค้ด โดยเราจะเรียกการจัดเก็บเวอร์ชันของซอร์สโค้ดลงฐานข้อมูลว่า "การ Check-in" และการเรียกคืนซอร์สโค้ดจากฐานข้อมูลเพื่อทำงานต่อว่า "การ Check-out" ดังรูปที่ 1.3



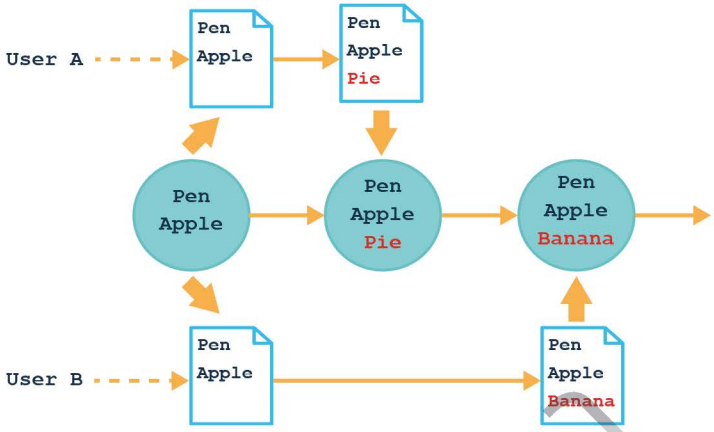
รูปที่ 1.3 ตัวอย่างโฟลว์ของการ Check-in และ Check-out

Local Version Control System เป็น Version Control ในยุคแรกๆ ที่มีความสามารถในการจัดเก็บเวอร์ชัน (Check-in) พร้อมทั้งข้อความช่วยจำ (Log Message) ลงฐานข้อมูล และเรียกคืนเวอร์ชันจากฐานข้อมูล (Check-out) กลับมายังพื้นที่ทำงาน เพื่อให้นักพัฒนาซอฟต์แวร์แก้ไขซอร์สโค้ดต่อไป โดยเราสามารถมอง Local Version Control System ได้อีกแบบ ดังรูปที่ 1.4



รูปที่ 1.4 Local Version Control System
(ที่มา : https://git-scm.com)

แต่การ Check-out จากผู้ใช้หลายคนพร้อมกัน อาจทำให้เกิดปัญหาขึ้น ดังรูปที่ 1.5



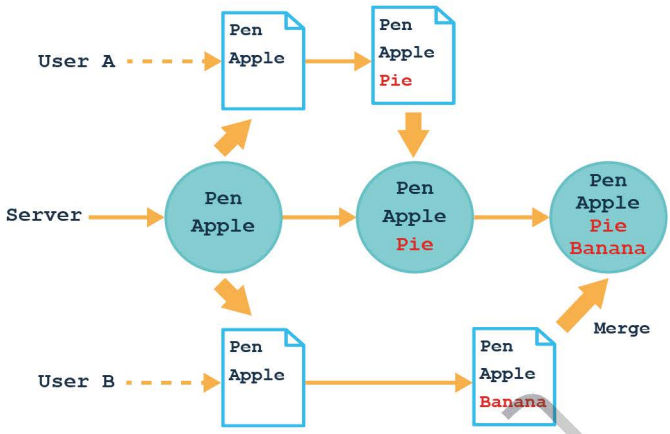
รูปที่ 1.5 ตัวอย่างปัญหาจากการ Check-out โดยผู้ใช้หลายคนพร้อมกัน

เมื่อ User A และ User B มีการ Check-out ซอร์สโค้ดออกจากฐานข้อมูล แต่ User A ทำงานเสร็จก่อนจึงได้ Check-in กลับเป็นคนแรก เมื่อ User B ทำงานของตนเองเสร็จจึง Check-in บ้าง ในกรณีนี้เนื้อหาที่เพิ่งถูกเพิ่มโดย User A ในเวอร์ชันก่อนหน้าจะถูกลบทิ้ง

เพื่อแก้ปัญหาการสูญเสียข้อมูลเมื่อมีการ Check-out จากผู้ใช้หลายคน จึงต้องมีการล็อคฐานข้อมูล (Lock Database) ไว้ ไม่ให้ผู้ใช้ที่ทำงานเสร็จช้ากว่าสามารถ Check-in ตามหลังได้ ซึ่งทำให้เป็นข้อจำกัดสำคัญของ Local Version Control System

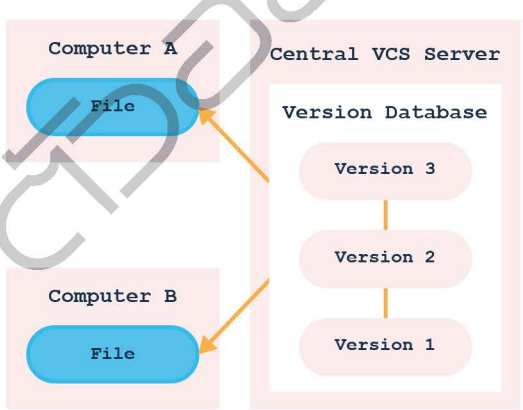
Centralized Version Control System

VCS แบบรวมศูนย์ เป็น Version Control ในยุคต่อมาที่พยายามแก้ปัญหากรณีมีการใช้งานหลายคน โดยการเก็บฐานข้อมูลไว้บนเซิร์ฟเวอร์ ซึ่งเมื่อผู้ใช้ Check-out งานเวอร์ชันเดียวกันแล้วกลับมา Check-in ระบบจะพยายามรวมเนื้อหาเข้าด้วยกัน (Merge) ถ้ารูปแบบการรวมเนื้อหามีความเรียบง่าย ระบบจะรวมเนื้อหาให้อัตโนมัติ แต่ถ้าการรวมมีความซับซ้อน ระบบก็จะแจ้งให้ผู้ใช้ตัดสินใจแทน แสดงดังรูปที่ 1.6



รูปที่ 1.6 การรวมเนื้อหาเข้าด้วยกัน (Merge) ของระบบแบบรวมศูนย์

เราสามารถมอง Centralized Version Control System (CVCS) ได้ ดังรูปที่ 1.7

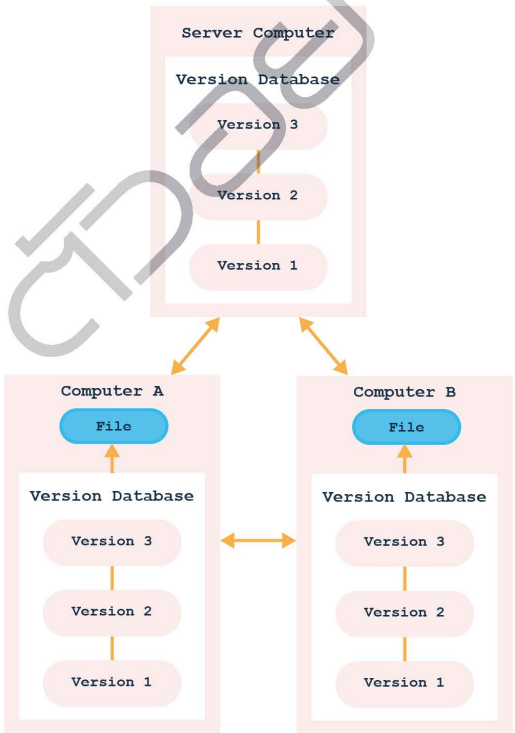


รูปที่ 1.7 Centralized Version Control System
(ที่มา : https://git-scm.com)

Centralized Version Control System ที่ได้รับความนิยมคือ Subversion (SVN) ซึ่งถูกพัฒนามาเมื่อปี ค.ศ. 2000 และจนถึงทุกวันนี้ก็ยังมีผู้ใช้งานอยู่บ้าง

Distributed Version Control System

อย่างไรก็ตาม ระบบ Centralized Version Control System ก็ยังมีข้อเสียสำคัญคือ มันเป็นการทำงานแบบรวมศูนย์ หากเซิร์ฟเวอร์ล่มไปชั่วขณะ ผู้ใช้งานจะไม่สามารถ Check-in หรือ Check-out ได้ และถ้าเซิร์ฟเวอร์เสียหายถาวร เวอร์ชันของซอร์สโค้ดทั้งหมดก็ได้รับผลกระทบไปด้วย เพราะการดึงงานจาก Centralized Version Control System จะเป็นการ Check-out เฉพาะเวอร์ชันแบบ Online ขณะที่ Distributed Version Control System (DVCS) จะแก้ปัญหาการทำงานแบบรวมศูนย์โดยการโคลนฐานข้อมูลมาทั้งหมด (เราจะเรียก Database ที่จัดเก็บเวอร์ชันของซอร์สโค้ดว่า "Repository") ซึ่งเมื่อ Clone มาแล้วเราสามารถ Check-in และ Check-out บน Local Host แบบ Offline ก่อนจะ Push ขึ้นสู่เซิร์ฟเวอร์ในภายหลัง โดยเราสามารถมอง Distributed Version Control System ได้ ดังรูปที่ 1.8



รูปที่ 1.8 Distributed Version Control
(ที่มา : https://git-scm.com)

ระบบ Distributed Version Control System ที่สำคัญตัวหนึ่งคือ BitKeeper โดยในปี ค.ศ. 2002 ได้ถูกนำมาใช้ในการเก็บซอร์สโค้ดของระบบปฏิบัติการ Linux ซึ่งในตอนนั้นอนุญาตให้นำไปใช้งานได้ฟรีๆ อย่างไรก็ตาม ในปี ค.ศ. 2005 BitKeeper ถูกเปลี่ยนมาเป็นซอฟต์แวร์ที่ไม่สามารถใช้งานได้ฟรีอีกต่อไป

ถ้าเราเป็น ไลนัส ทอร์วัลด์ส (Linus Torvalds) ผู้ก่อตั้ง Linux เราจะแก้ปัญหานี้อย่างไร?

สำหรับ Linus Torvalds แล้ว วิธีแก้ปัญหาของเขาคือ การพัฒนา Distributed Version Control System ขึ้นมาใช้งานเองภายใต้ชื่อว่า "Git" และจากจุดนี้เอง Git จึงถือกำเนิดขึ้นมาในปี ค.ศ. 2005 และได้รับอนุญาตให้นำไปใช้งานในแบบ Open Source และเป็น Free Software นับตั้งแต่นั้นเป็นต้นมา

สรุปท้ายบท

- Version Control คือ เครื่องมือที่ช่วยติดตามการเปลี่ยนแปลงของซอร์สโค้ด ซึ่งคล้ายกับแนวคิดของการทำ Undo/Redo ใน Microsoft Word ที่ช่วยเรียกคืนเวอร์ชันของเอกสาร
- ก่อนจะมีระบบ Version Control นั้น นักพัฒนาซอฟต์แวร์ต้องแยกเวอร์ชันด้วยการ Copy File และ Folder แต่เพื่อหลีกเลี่ยงความสิ้นเปลืองพื้นที่จัดเก็บซอร์สโค้ดของระบบปฏิบัติการ Linux ในยุคแรกๆ จึงมีการเก็บเวอร์ชันเป็นแบบ Patch
- เพื่อหลีกเลี่ยงการทำ Patch บาง Patch หาย จนทำให้ไม่สามารถประกอบร่างซอร์สโค้ดกลับไปยังเวอร์ชันต่างๆ ได้ จึงมีการพัฒนา Version Control System แบบ Local Version Control System ขึ้นมาแทน
- เพื่อแก้ปัญหาการใช้งานหลายคน จึงมีการออกแบบ Centralized Version Control System ที่เก็บฐานข้อมูลไว้บนเซิร์ฟเวอร์ และมีระบบรวมเนื้อหาจากผู้ใช้หลายคนเข้าด้วยกัน
- Distributed Version Control System จะแก้ปัญหาการทำงานแบบรวมศูนย์ของ Centralized Version Control System โดยการโคลนฐานข้อมูลมาที่ Local Host ทั้งหมด ทำให้สามารถ Check-in และ Check-out บน Local Host แบบ Offline ก่อนจะ Push ขึ้นเซิร์ฟเวอร์ในภายหลัง
- Linus Torvalds พัฒนา Distributed Version Control System ขึ้นมาเองสำหรับเก็บซอร์สโค้ดของระบบปฏิบัติการ Linux ที่ชื่อว่า "Git" และเปิดให้ใช้งานแบบ Open Source และ Free Software


git

CHAPTER

02

หลักการพื้นฐานของ Git

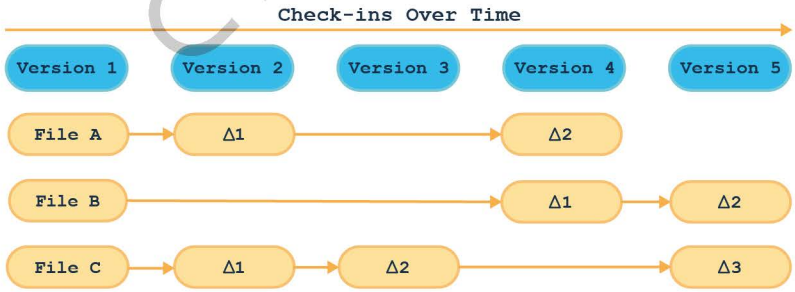
(Basic Principles of Git)

นอกจาก Linus Torvalds จะเปิดให้ใช้งาน Git แบบ Open Source และ Free Software แล้ว สิ่งที่ทำให้ Git เป็น Version Control ยอดนิยมในยุคนี้ก็คือ เรื่องของความเร็ว บทนี้ผู้อ่านจะได้เข้าใจวิธีการจัดเก็บ Version ของ Git ซึ่งเป็นปัจจัยหนึ่งที่ทำให้เราสามารถย้อนกลับไปยังเวอร์ชันก่อนหน้าได้รวดเร็ว เปรียบเหมือนกับการทำ Undo/Redo ใน Microsoft Word นอกจากนี้ เพื่อให้สามารถใช้งาน Git แบบ Command Line ได้อย่างคล่องแคล่ว ผมจะแนะนำขั้นตอนการทำงานของ Git (Git Workflow) ให้ผู้อ่านได้เข้าใจกันก่อน

เปรียบเทียบ Git กับ Version Control System อื่นๆ

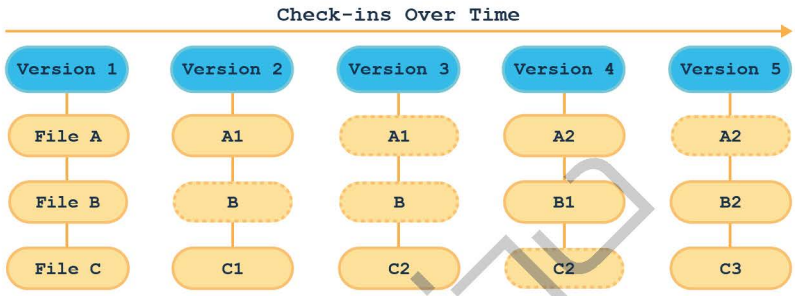
Git เป็น Distributed Version Control System (DVCS) ซึ่งต้องมีการก็อปปี้เวอร์ชันของซอร์สโค้ดมาเก็บไว้ที่เครื่องของเราก่อน (Local Host) ทำให้ผู้ใช้สามารถแก้ไขโปรเจ็กต์ได้ทุกที่แบบ Offline และ Check-in เพื่อจัดเก็บความเปลี่ยนแปลงของซอร์สโค้ดลงในฐานข้อมูลภายในเครื่องของเรา (Local Repository) โดยไม่จำเป็นต้องติดต่อกับ Git Repository บนเซิร์ฟเวอร์ และพอแก้ไขเสร็จแล้วจึงค่อยสั่ง Sync เพื่อให้ Version ฝั่ง Local และ Server อัพเดตเหมือนกันในภายหลัง (ด้วยการ Pull/Merge/Push)

ดังนั้น เมื่อเราสามารถทำทุกอย่างบน Local Host ได้ มันจึงเร็วกว่าระบบแบบรวมศูนย์ เช่น Subversion (SVN) ที่เป็น Centralized Version Control System อย่างมาก นอกจากนี้ Git ยังมีความแตกต่างกับ Version Control System (VCS) อื่นๆ ในเรื่องของวิธีการจัดเก็บข้อมูลเวอร์ชัน โดย Version Control System ก่อนหน้านี้จะเก็บเป็นลิงค์ (Link) ของความเปลี่ยนแปลง หรือส่วนต่างระหว่างเวอร์ชัน ดังรูปที่ 2.1



รูปที่ 2.1 Delta-based Version Control
(ที่มา : https://git-scm.com)

แทนที่จะเก็บส่วนต่างระหว่างเวอร์ชัน แต่ **Git จะเก็บเวอร์ชันของซอร์สโค้ดเป็น Snapshot ซึ่งเป็นเหมือนการถ่ายรูปไฟล์** และอ้างถึงไฟล์ที่ไม่มีการเปลี่ยนแปลงในเวอร์ชันก่อนหน้า เมื่อมีการ Check-in มันจะถ่ายรูปไฟล์ของเราว่ามีหน้าตาเป็นอย่างไร โดย Git จะมองข้อมูลที่จัดเก็บเป็นเหมือนกับระบบแฟ้ม (File System) เล็กๆ ระบบหนึ่ง ดังรูปที่ 2.2



รูปที่ 2.2 Snapshot ซึ่งเป็น Version Control ในแบบของ Git
(ที่มา : https://git-scm.com)

ก่อนที่จะมีการบันทึก ทุกอย่างในเวอร์ชันจะถูกนำไปหา Checksum ซึ่งเป็นการตรวจสอบความถูกต้องของข้อมูล โดยใช้ SHA-1 Hash ดังนั้น จึงไม่มีทางที่เราจะเปลี่ยนแปลงข้อมูลในไฟล์และโฟลเดอร์โดย Git จะไม่รู้ เราจะใช้ค่า Checksum ในการอ้างถึงเวอร์ชันของซอร์สโค้ดในหลายๆ งาน ซึ่ง SHA-1 Hash มีหน้าตาดังตัวอย่าง

ตัวอย่างของ Checksum

```
e70cdec636912065839fed45238db9d4f898f199
```

SHA-1 Hash คืออะไร

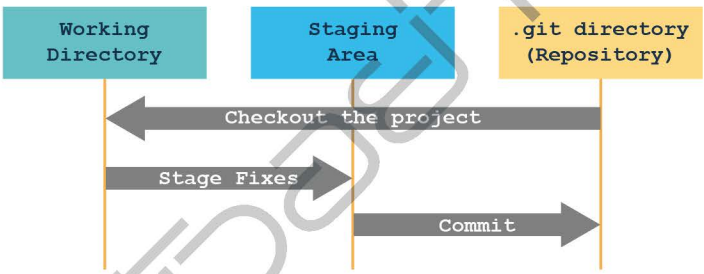
เป็นอัลกอริทึมที่อาศัยการคำนวณทางคณิตศาสตร์ในการเข้ารหัสทางเดียว โดยผลจากการเข้ารหัสเวอร์ชันแบบ SHA-1 จะได้เลขฐาน 16 ขนาด 40 ตัวอักษร

เข้าใจการทำงานของ Git (Git Workflow)

หลักการทำงานของ Git ที่ต้องเข้าใจก่อนใช้งานก็คือ Git แบ่งสถานะของไฟล์ที่มันติดตาม (Tracked) ออกเป็น 3 สถานะ ได้แก่ Modified, Staged และ Committed

- **Modified** หมายถึง เราได้แก้ไขไฟล์ไปแล้ว แต่ยังไม่เริ่มกระบวนการจัดเก็บลง Repository
- **Staged** หมายถึง เราได้ทำเครื่องหมาย File ที่ได้ถูกแก้ไขเพื่อจะบันทึกในเวอร์ชันหน้า
- **Committed** หมายถึง ข้อมูลถูกบันทึกอย่างปลอดภัยใน Repository บน Local Host

ซึ่งจะแสดงการเปลี่ยนสถานะของไฟล์ในส่วนต่างๆ ดังรูปที่ 2.3



รูปที่ 2.3 Git Workflow : Working Directory, Staging Area และ .git directory (ที่มา : https://git-scm.com)

จากรูปที่ 2.3 Workflow ของ Git มีกระบวนการดังนี้

- เราจะแก้ไขไฟล์ใน Working Directory (ไฟล์ที่แก้ไขจะอยู่ในสถานะ Modified)
- รวบรวมไฟล์ที่ถูกแก้ไขลงใน Staging Area เพื่อจะ Commit ใน Version ถัดไป (ไฟล์ที่ถูกรวบรวมจะอยู่ในสถานะ Staged)
- Commit เพื่อนำไฟล์จาก Staging Area ไปจัดเก็บอย่างถาวรใน Local Repository (ไฟล์ที่ถูกบันทึกลง Repository อยู่ในสถานะ Committed)

สรุปท้ายบท

- ปัจจัยที่ทำให้ Git มีความเร็วกว่า VCS หลายตัวก็คือ มันเป็น Distributed Version Control System ซึ่งมีการก็อปปี้เวอร์ชันของซอร์สโค้ดมาเก็บไว้ที่ Local Host ก่อน ทำให้ผู้ใช้สามารถแก้ไขโปรเจ็กต์ได้แบบ Offline และ Check-in ซอร์สโค้ดบน Local Repository โดยไม่ต้องติดต่อกับ Repository บนเซิร์ฟเวอร์
- วิธีการจัดเก็บเวอร์ชันแบบ Snapshot ก็สามารถเพิ่มความเร็วให้ Git ได้เช่นกัน
- Git จะใช้ค่า Checksum เพื่ออ้างถึงเวอร์ชันของซอร์สโค้ด
- Git แบ่งสถานะของไฟล์ที่มันติดตาม (Tracked) ออกเป็น 3 สถานะคือ Modified, Staged และ Committed ขึ้นอยู่กับว่ามันจะถูกเก็บไว้ที่ไหนใน 3 ที่ ได้แก่ Working Directory, Staging Area หรือ Repository

GitLab

CHAPTER

03

ฝึกการใช้งาน Git ขั้นพื้นฐาน

(Practicing Git Basics)

เพื่อให้ผู้อ่านได้เห็นขั้นตอนการทำงานของ Git (Git Workflow) จากการปฏิบัติจริง บทนี้เราจะเริ่มต้นด้วยการสร้าง Project บน GitLab Server แล้วเชื่อมโยงกับ Local Repository เนื่องจาก Git นั้นเป็น Distributed Version Control System เราจึงแก้ไข Project แบบ Offline แล้ว Check-in ซอร์สโค้ดบน Local Repository หลังจากนั้นจึง Sync ข้อมูลกับ GitLab Server อีกที

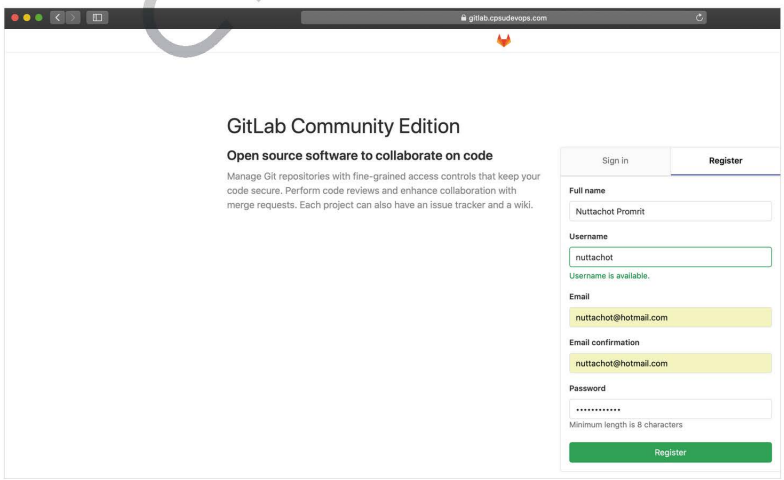
เริ่มต้นใช้งาน Git ครั้งแรกกับ GitLab

สำหรับบทที่ 3 นี้ ผู้อ่านจะได้เห็นขั้นตอนการทำงานของ Git (Git Workflow) โดยสามารถฝึกปฏิบัติตามได้ จะเริ่มต้นด้วยการสร้าง Project บน GitLab Server แล้วเชื่อมโยงเข้ากับ Local Repository และเพราะเหตุว่า Git นั้นเป็น Distributed Version Control System เราจึงสามารถแก้ไข Project แบบ Offline แล้ว Check-in ซอร์สโค้ดบน Local Repository หลังจากนั้นจึง Sync เพื่ออัพเดตข้อมูลให้ตรงกันกับ GitLab Server อีกที

เริ่มต้นนับหนึ่งกับ Git Version Control

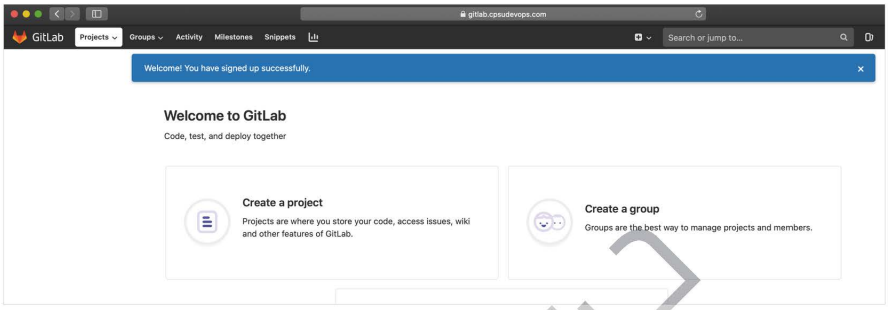
การ Register และ Sign in ใน GitLab

ในบทที่ 3 และบทต่อไป เราสามารถฝึกการใช้งาน Git ได้จากเว็บไซต์ gitlab.cpsudevops.com หรือ https://gitlab.com ซึ่งในขั้นแรกให้เรา Register โดยป้อนข้อมูลต่างๆ ลงในแบบฟอร์มให้ครบ ได้แก่ Full name, Username, Email และ Password แล้วคลิกปุ่ม Register เพื่อลงทะเบียนอย่างเป็นทางการ ดังรูปที่ 3.1



รูปที่ 3.1 หน้าเพจ Sign in ของ GitLab เพื่อ Register เข้าใช้งานครั้งแรก

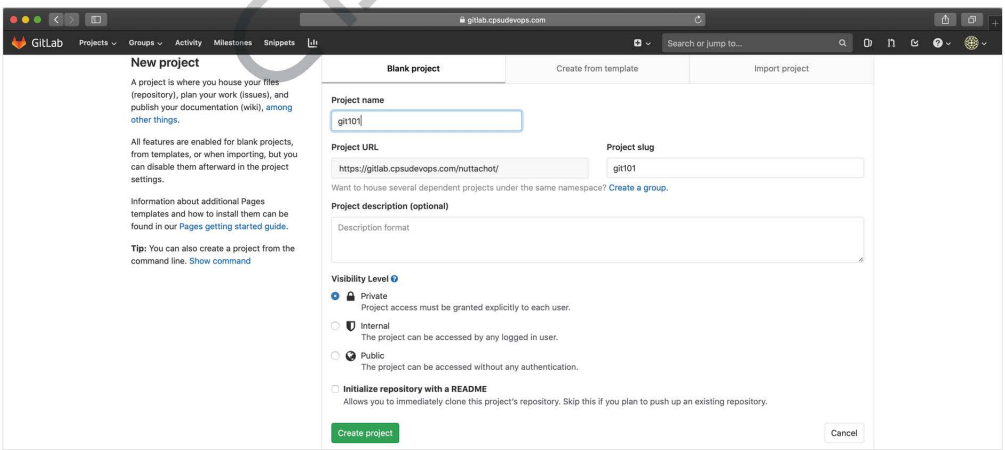
เมื่อ Sign in เรียบร้อยแล้ว เราจะเห็นหน้าตาของ GitLab ดังรูปที่ 3.2 โดย GitLab จะเรียก "Repository" สำหรับเก็บเวอร์ชันของซอร์สโค้ดว่า "Project" ซึ่งในที่นี้เราจะทดลองสร้าง Project ชื่อ "git101"



รูปที่ 3.2 หน้าแรกของ GitLab

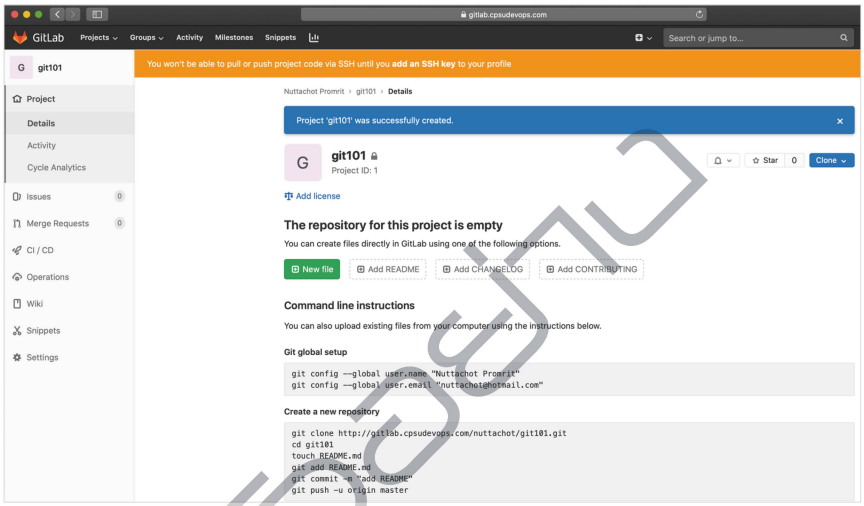
การสร้าง Project บน GitLab

สำหรับ GitLab จะมีค่าเริ่มต้นของ Repository เป็นส่วนตัว (Private) ซึ่งเหมาะสำหรับการพัฒนา Project ที่ยังไม่ต้องการเผยแพร่ซอร์สโค้ดเป็นสาธารณะ (Public) โดยเราสามารถสร้าง Project บน GitLab แบบ Private ก่อนได้ โดยการกำหนดที่หัวข้อ Visibility Level ดังรูปที่ 3.3



รูปที่ 3.3 การกำหนด Visibility Level ของ Project ใน GitLab

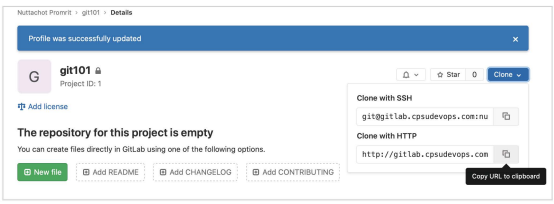
เมื่อ Project ใหม่ถูกสร้างขึ้น มันจะยังไม่มีการเก็บเวอร์ชันใดๆ ลงฐานข้อมูล จะยังเป็นเพียง Empty Project ซึ่งเราสามารถเพิ่มไฟล์ README.md เพื่ออธิบายรายละเอียดเกี่ยวกับ Project ได้จากหน้า Project Details ที่ปุ่ม Add README ดังรูปที่ 3.4 หรือจะเพิ่มบน Local Host ในเครื่องของเราก่อนแล้วจึงค่อย Sync กับ Remote Repository ไปยัง GitLab Server ในภายหลังก็ได้ (ช่วงเริ่มต้นใช้งานการเพิ่ม README.md บน Local Host น่าจะสะดวกกว่า)



รูปที่ 3.4 หน้าจอ Project Details โดยเมนูจะอยู่ด้านซ้าย

การสร้าง Remote Repository

เพื่อจะใช้งาน Distributed Version Control System อย่าง Git Local Host เราจะต้องสร้างการเชื่อมโยงเพื่อติดต่อกับ Remote Repository โดยใช้ URL จาก Clone with HTTP ดังรูปที่ 3.5 ซึ่ง URL ที่ได้เป็นเหมือนเส้นทางติดต่อระหว่างเครื่องของเรากับ GitLab Server



รูปที่ 3.5 การก็อปปี้ URL เพื่อใช้ในการเชื่อมโยงกับ Remote Repository

การติดตั้ง Git Client และการเรียกใช้งาน

ก่อนจะใช้งาน Git บนระบบปฏิบัติ Windows หรือ Mac จำเป็นจะต้องมีการติดตั้ง Git Client จาก https://git-scm.com/downloads ก่อน อย่างไรก็ตาม ถ้าผู้อ่านที่ใช้ Windows ต้องการใช้งาน Git ในรูปแบบ Unix Command Line Based เหมือนกับในหนังสือเล่มนี้ ผู้เขียนแนะนำให้ติดตั้ง Windows Subsystem for Linux (WSL) และ Ubuntu Distribution จาก Microsoft Store แทน โดยทำตามขั้นตอนดังนี้

1. เปิด PowerShell แบบ Administrator แล้ว Enable WSL ด้วยคำสั่งดังนี้

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-
Linux /all /norestart
```

2. Enable Virtual Machine Feature

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /
norestart
```

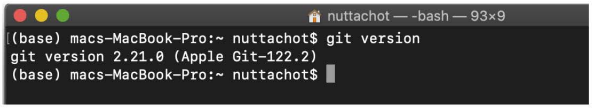
3. รีสตาร์ทคอมพิวเตอร์
4. ดาวน์โหลดและติดตั้ง Linux Kernel Update Package จาก https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi
5. กำหนด WSL เป็น version 2

```
wsl --set-default-version 2
```

6. ติดตั้ง Ubuntu Distribution (Ubuntu 18.04 LTS) จาก Microsoft Store (https://www.microsoft.com/store/apps/9N9TNGVNDL3Q)
7. ในครั้งแรกที่ใช้งาน เราจะต้องสร้าง Account ใหม่สำหรับ Ubuntu Distribution โดยป้อน Username และ Password หลังจากติดตั้งเสร็จแล้ว เราสามารถดูเวอร์ชันของ Git ได้จาก Command Line ของ Ubuntu Distribution โดยใช้คำสั่ง git version ดังนี้

```
git version
```

ผลลัพธ์ที่ได้จากการเรียกดูเวอร์ชัน ดังรูปที่ 3.6



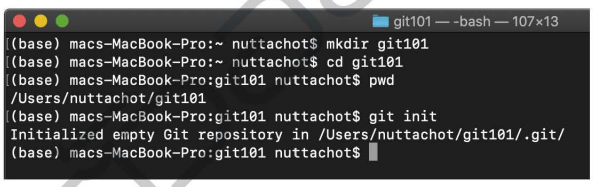
รูปที่ 3.6 ผลลัพธ์ที่ได้จากการเรียกดูเวอร์ชัน

การคอนฟิก Git ให้พร้อมใช้งาน

ในการใช้งานครั้งแรก เราจะต้องคอนฟิก Git ให้จำ Username และ Email ด้วยคำสั่ง git config

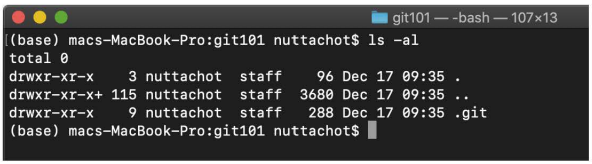
```
git config --global user.name "nuttachot"
git config --global user.email nuttachot@hotmail.com
```

จากนั้นจึงสร้างโฟลเดอร์ของ Project ชื่อ "git101" บน Local Host และใช้คำสั่ง git init เพื่อเริ่มต้นการทำงานจะได้ผลลัพธ์ ดังรูปที่ 3.7



รูปที่ 3.7 ผลลัพธ์ที่ได้จากคำสั่ง git init

จากรูปที่ 3.7 Git จะสร้างโฟลเดอร์ .git ไว้ภายในโฟลเดอร์ Project (git101) สำหรับจัดเก็บเวอร์ชันในเครื่องของเรา (Local Repository) ให้เอง ซึ่งเราจะมองไม่เห็นโฟลเดอร์ที่มีการตั้งชื่อด้วย "." นำหน้า สำหรับบน Unix Command Line Based จะใช้คำสั่ง ls -a หรือ ls -al เพื่อดูโฟลเดอร์ .git ดังรูปที่ 3.8



รูปที่ 3.8 โฟลเดอร์ .git ที่สร้างขึ้น

หลังจากนั้นเราจะเชื่อมโยง Local Repository บนเครื่องของเรากับ Remote Repository บน GitLab Server โดยนำ URL จาก Clone with HTTP ที่ได้ก่อนหน้านี้มาใช้ผ่านคำสั่ง git remote ดังนี้

```
git remote add origin http://gitlab.cpsudevops.com/nuttachot/git101.git
```

เป็นอันเสร็จสิ้นสำหรับการคอนฟิก Git เพื่อให้พร้อมใช้งาน

การ Check-in Source Code

จุดประสงค์ที่สำคัญของการใช้งาน Version Control System คือ เพื่อให้เราสามารถย้อนกลับไปยังเวอร์ชันก่อนหน้าได้เมื่อพบปัญหาในระหว่างการเขียนโปรแกรม การ Check-in เพื่อส่งการเปลี่ยนแปลงซอร์สโค้ดไปยังระบบจัดเก็บเวอร์ชัน (Version Control System) จึงเป็นเรื่องที่ต้องทำเป็นประจำ เพราะถ้าคนเขียนโปรแกรมคนใดคนหนึ่งในทีมพัฒนาไม่ Check-in เป็นประจำ แล้วพบปัญหาขึ้นกับซอฟต์แวร์ที่อยู่ระหว่างพัฒนา ก็อาจก่อให้เกิดปัญหาใหญ่ตามมาคือไม่สามารถย้อนกลับไปยังเวอร์ชันต่างๆ ก่อนหน้าได้ หรืออาจต้องยุ่งยากหรือใช้เวลานานกว่าจะย้อนกลับไปได้

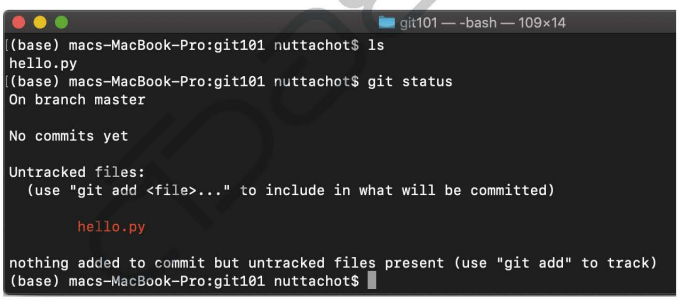
ต่อไปจะเป็นตัวอย่างการ Check-in Source Code กับ Local Repository (ระบบจัดเก็บเวอร์ชันภายในเครื่องของเรา) และวิธีการ Sync ไปยัง GitLap Server ผ่านทาง URL: gitlab.cpsudevops.com

วิธี Check-in กับ Local Repository

ให้เราสร้างไฟล์ hello.py โดยใช้โปรแกรม Text Editor ที่ผู้อ่านมีอยู่แล้วในเครื่อง ดังตัวอย่างรูปที่ 3.9

รูปที่ 3.9 ไฟล์ hello.py

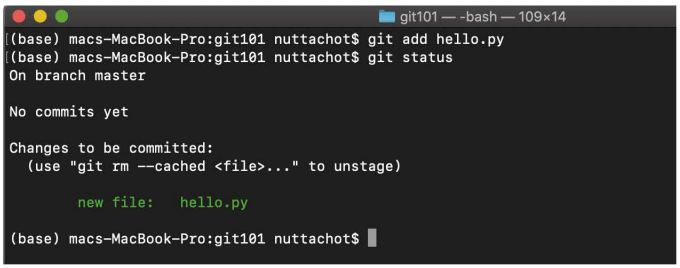
ไฟล์ที่สร้างใหม่จะยังไม่ถูกติดตามความเปลี่ยนแปลง (Track) จากระบบ Git ซึ่งเราสามารถดูสถานะของไฟล์ใน Project โดยใช้คำสั่ง git status ดังรูปที่ 3.10 สังเกตว่าไฟล์ที่มีสถานะ Modified หรือไฟล์ที่ยังไม่ถูก Track จะแสดงด้วยสีแดง



รูปที่ 3.10 ไฟล์ hello.py ที่ยังไม่ถูก Track

เพื่อจะ Track ไฟล์ hello.py และส่งมันไปยัง Staging Area เราจะใช้คำสั่ง git add ซึ่งไฟล์ที่มีสถานะ Staged จะแสดงด้วยสีเขียว ดังรูปที่ 3.11

```
git add hello.py
```

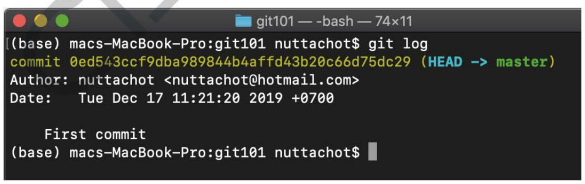


รูปที่ 3.11 ไฟล์ hello.py ที่อยู่ในสถานะ Staged

เราจะใช้คำสั่ง git commit เพื่อจัดเก็บไฟล์อย่างถาวรด้วยการสร้างเป็นเวอร์ชันของซอร์สโค้ด และใช้พารามิเตอร์ (Parameter) -m สำหรับใส่ข้อความช่วยจำ (Message Log) เพื่ออธิบายการ Commit แต่ละเวอร์ชัน

```
git commit -m 'First commit'
```

Git จะเรียกเวอร์ชันต่างๆ ของ Project ว่า "History" ซึ่งในการเรียกดู History ของ Project เราจะใช้คำสั่ง git log โดยรูปที่ 3.12 แสดงถึงผลการใช้คำสั่ง git log เมื่อเราได้ Commit ไปแล้ว 1 เวอร์ชัน ซึ่ง Git จะใช้ค่า SHA-1 Hash เป็น Commit ID

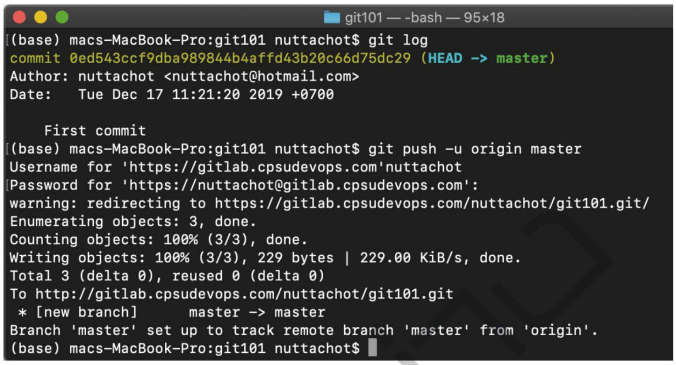


รูปที่ 3.12 ผลลัพธ์ของคำสั่ง git log

จากรูปที่ 3.12 History ทุกคนในทีมจะสามารถเห็นว่าใครเป็นคน Commit และ Commit ที่ Branch ไหน (สำหรับเรื่องของ Branch เราจะพูดถึงอย่างละเอียดในบทที่ 6) โดย Branch Default ที่เรา Check-in คือ Master Branch

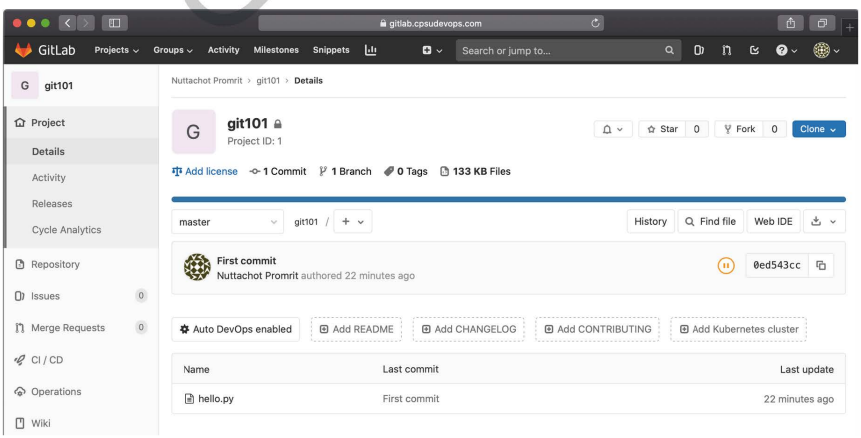
วิธี Sync History กับ GitLab Server

เราจะ Sync History กับ Remote Git Repository โดยใช้คำสั่ง git push -u origin master และยืนยันตัวตนกับ GitLab Server ด้วย Username และ Password ดังรูปที่ 3.13



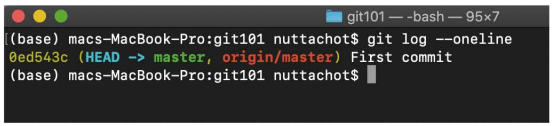
รูปที่ 3.13 ผลลัพธ์ของคำสั่ง git push -u origin master

เมื่อเข้าไปดูที่เว็บไซต์ http://gitlab.cpsudevops.com/nuttachot/git101.git ดังรูปที่ 3.14 พบว่ามีการ Sync ข้อมูลกับ Remote Repository เรียบร้อยแล้ว ในครั้งต่อไปเราสามารถ Push โดยใช้เพียงคำสั่ง git push สั้นๆ เพราะเราได้ใช้พารามิเตอร์ -u เพื่อให้มันจดจำว่าจะ Push ไปที่ Remote และ Branch ไหน



รูปที่ 3.14 หน้าจอ Project Details ที่มีการ Sync ข้อมูลกับ Remote Repository แล้ว

ในการดู History อย่างย่อจะใช้คำสั่ง git log --oneline ซึ่งเราจะเห็นว่าขณะนี้มีการ Check-out อยู่ที่ Master Branch และ Commit ID 0ed543c จาก HEAD Pointer ดังรูปที่ 3.15



รูปที่ 3.15 ผลลัพธ์ของคำสั่ง git log --oneline

สรุปท้ายบท

- เริ่มต้นใช้งาน Git ครั้งแรกด้วยการสร้าง Project บน GitLab Server ที่ https://gitlab.com หรือ https://gitlab.cpsudevops.com
- ในการใช้งานครั้งแรก เราต้อง Config ให้ Git จำ Username และ Email ของผู้ใช้ ด้วยคำสั่ง git config บน Local Host
- จากนั้นจึงสร้าง Folder ของ Project บน Local Host และใช้คำสั่ง git init เพื่อเริ่มต้นการทำงาน
- เพื่อใช้งาน Git เราจะต้องเชื่อมโยง Local Repository กับ Remote Repository ด้วย URL ที่ได้ตอนสร้าง Project บน GitLab Server
- เราจะดูสถานะของไฟล์ใน Project ด้วยคำสั่ง git status
- เพื่อจะ Track ไฟล์และส่งมันไปยัง Staging Area เราจะใช้คำสั่ง git add
- เราสามารถ Check-in ซอร์สโค้ดบน Local Repository ด้วยคำสั่ง git commit
- หลังจากนั้นจึง Sync ข้อมูลกับ GitLab Server ด้วยคำสั่ง git push


jupyter
GitLab