Version Control

## จุดประสงค์

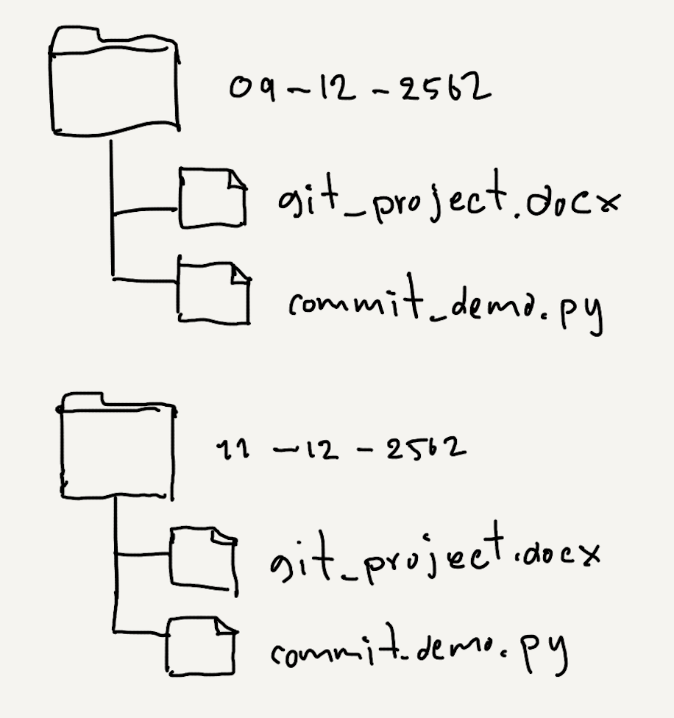
* เรียนรู้และเข้าใจการทำงานของระบบ Version Control System
* เข้าใจหลักการทำงานของ Git และ Git Branch
* เรียนรู้ถึงระบบการทำงานที่เหมาะสมใน Git ด้วย Git Flow
* สามารถใช้งาน GitLab ได้

## [1] Version Control System (VCS)

Version Control คือเครื่องมือช่วยติดตามการเปลี่ยนแปลงของ Source Code โดยการเก็บประวัติการเปลี่ยนแปลงลง Database ชนิดพิเศษ เมื่อมีความผิดพลาดเกิดขึ้น Software Developers จะสามารถย้อนกลับไปยังช่วงเวลาต่าง ๆ และเปรียบเทียบ Code ใหม่กับ Code ใน Version ก่อนหน้า เพื่อแก้ไขความผิดพลาดให้มีผลกระทบต่อทีมน้อยที่สุด

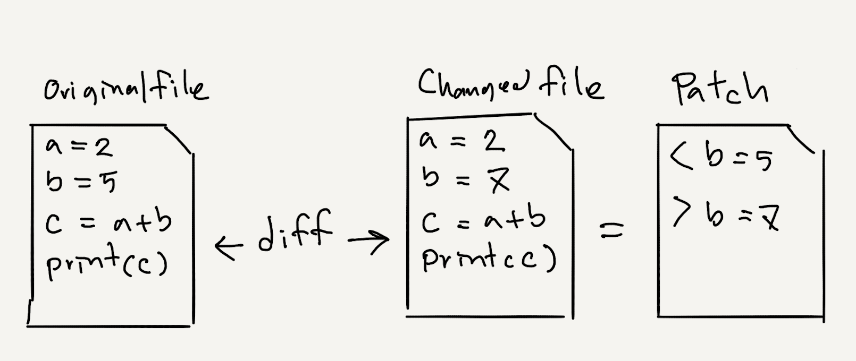
ถ้าจะเทียบให้เห็นภาพจากเรื่องใกล้ตัว Version Control ก็คล้ายกับคำสั่ง Undo/Redo ในโปรแกรม Microsoft Word ที่ช่วยเรียกคืน Version ของเอกสารได้ แต่หากผู้ใช้สั่งบันทึกแล้วปิดโปรแกรม Microsoft Word เมื่อเรียกไฟล์ขึ้นมาแก้ไขอีกครั้ง ผู้ใช้จะไม่สามารถย้อนกลับไปยัง Version ก่อนหน้าได้อีก ดังนั้นวิธีการหนึ่งที่เราสามารถจัดการได้ คือ การบันทึกเอกสารแยกเป็นไฟล์หลาย ๆ ไฟล์ เช่น git\_project.docx, git\_project\_v2.docx, git\_project\_back.docx, git\_project.\_final.docx และ git\_project\_lastest.docx เป็นต้น

แต่จะเห็นว่าการเก็บไฟล์ด้วยการตั้งชื่ออาจทำให้ผู้ใช้สับสนในภายหลังว่าไฟล์ไหนใหม่กว่ากัน รวมทั้งไม่สามารถบอกได้ว่าไฟล์ไหนถูกสร้างต่อยยอดมาจากไฟล์ไหนบ้าง เพื่อไม่ให้เกิดความสับสนเรื่อง Version ของเอกสาร เราอาจแก้ปัญหาโดยการจัดเก็บไฟล์แยกเป็น Folder ตามวันที่ เช่น



รูปที่ . การจัดเก็บงานแยกตามวันที่

แต่วิธีการนี้จะทำให้เราสิ้นเปลืองเนื้อที่เก็บข้อมูลตามจำนวน Version ของเอกสารที่เกิดขึ้น แม้ว่าจะมีการเปลี่ยนแปลงในเอกสารเพียงไม่กี่บรรทัด ก็ต้องคัดลอกไฟล์ทั้ง Folder เพื่อสร้างเอกสาร Version ใหม่ขึ้นมา และบอกไม่ได้ว่าไฟล์ Version นั้น ๆ มีความสำคัญขนาดไหน เพื่อหลีกเลี่ยงความสิ้นเปลืองพื้นที่จัดเก็บ เราอาจเก็บ Version เอกสารเป็น Patch เหมือนกับการเก็บ Version ของ Source Code ของระบบปฏิบัติการ Linux ในยุคแรก ๆ ดังรูปที่ 4.2

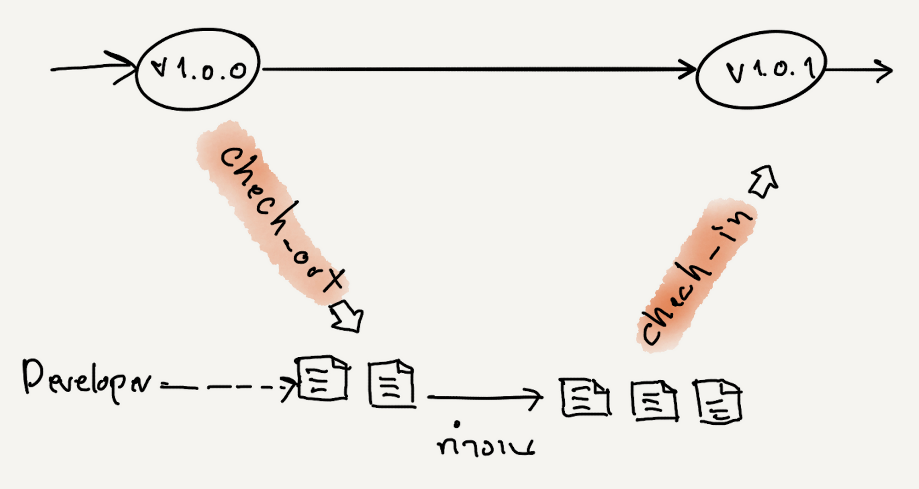


รูปที่ . การใช้ Path จัดเก็บความเปลี่ยนแปลง

จากภาพด้านบนแสดงตัวอย่างของ Patch ที่เกิดจากการเปรียบเทียบไฟล์เก่า (Original file) และ ไฟล์ใหม่ (Changed File) Patch ที่สร้างจะแสดงถึงบรรทัดที่มีการเปลี่ยนแปลง โดยใช้เครื่องหมาย < นำหน้าบรรทัดที่หายไป และ > นำหน้าบรรทัดที่เพิ่มมาแทน อย่างไรก็ตามการเก็บ Version ในรูป Patch อาจมีความเสี่ยงต่อการทำ Patch บาง Patch หาย จนทำให้ไม่สามารถประกอบร่าง Source Code เพื่อกลับไปยัง Version ต่าง ๆ ได้

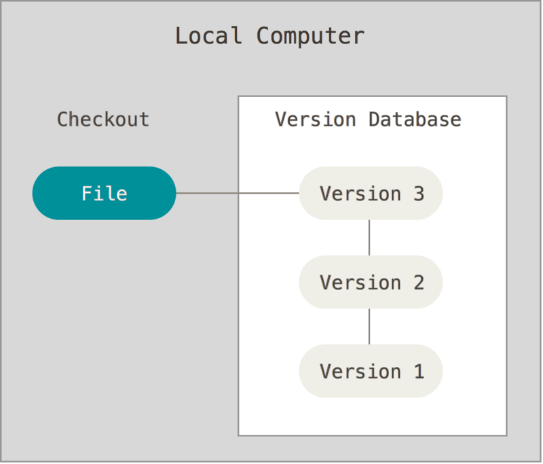
### Local Version Control System

เพื่อแก้ปัญหาดังที่กล่าวมา จึงมีการพัฒนา Version Control System ที่มี Database เฉพาะคอยจัดเก็บทุกการเปลี่ยนแปลงของ Source Code โดยเราจะเรียกการจัดเก็บ Version ของ Source Code ลง Database ว่าการ "Check-In" และการเรียกคืน Source Code จาก Database เพื่อทำงานต่อว่าการ "Check-Out"



รูปที่ . การ Check-In และ Check-/out

Local Version Control System เป็น Version Control ในยุคแรก ๆ ที่มีความสามารถในการจัดเก็บ Version (Check-In) พร้อมทั้งข้อความช่วยจำ (Log Message) ลง Database และเรียกคืน Version จาก Database (Check-Out) กลับมายังพื้นที่ทำงาน (Working Directory) เพื่อให้ Software developers แก้ไข Source Code ต่อไป โดยเราสามารถมอง Local Version Control System ได้อีกแบบดังภาพด้านล่าง

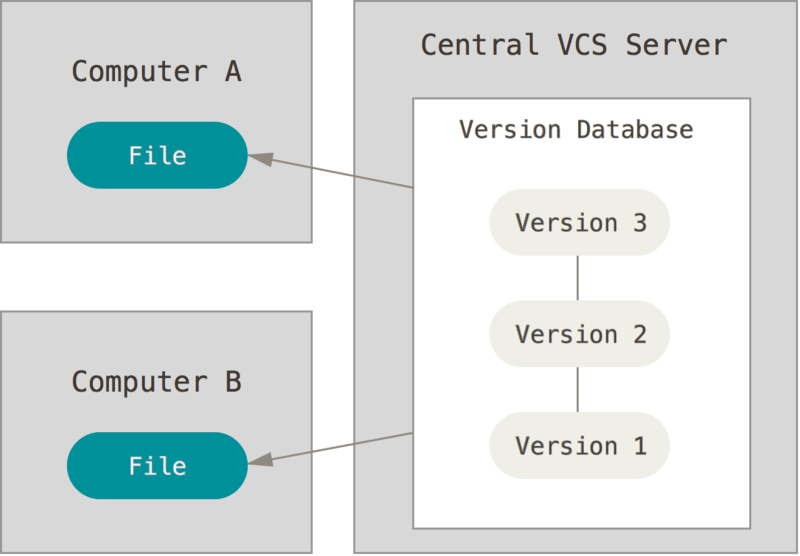


รูปที่ . Local Version Control System

แต่การ Check-Out จากผู้ใช้หลายคนพร้อมกันอาจทำให้เกิดปัญหาขึ้นในกรณีที่ A และ B ได้ Check-Out โค้ดออกจากฐานข้อมูลของ Local Version Control System จากนั้นเมื่อ A แก้ไขงานของตัวเองเสร็จแล้วจึงทำการ Check-In โค้ดกลับเข้าไป ต่อมาภายหลัง B แก้งานเสร็จเช่นกันและทำการ Check-In โค้ดกลับเข้าไปตามหลัง A จะทำให้โค้ดที่ A ได้แก้ไขไว้ถูกเขียนทับจากโค้ดของ B ที่ได้ Check-In ทับเข้าไปนั่นเอง ดังนั้นเพื่อแก้ปัญหาการสูญเสียข้อมูลเมื่อมีการ Check-Out จากผู้ใช้หลายคนจึงต้องมีการ Lock Database ไว้ไม่ให้ผู้ใช้ที่ทำงานเสร็จช้ากว่า สามารถ Check-In ตามหลังได้ ซึ่งทำให้เป็นข้อจำกัดสำคัญของ Local Version Control System

### Centralized Version Control System

Version Control ในยุคต่อมาจึงพยายามแก้ปัญหาการใช้งานหลายคน โดยเก็บ Database ไว้บน Server เมื่อผู้ใช้ Check-Out งาน version เดียวกัน แล้วกลับมา Check-In ระบบจะพยายามรวมเนื้อหาเข้าด้วยกัน (Merge) ถ้ารูปแบบการรวมเนื้อหามีความเรียบง่าย ระบบจะ Merge ให้อัตโนมัติ แต่ถ้าการรวมมีความซับซ้อน ระบบจะแจ้งให้ผู้ใช้ผู้ตัดสินใจแทนโดยเราสามารถมอง Centralized Version Control System ได้ดังภาพด้านล่าง



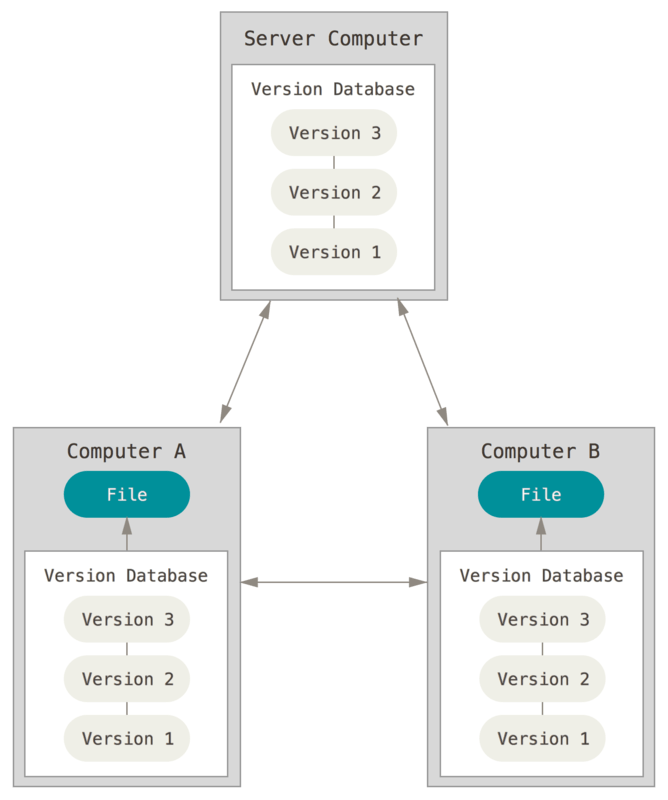
รูปที่ . Centralized Version Control System

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

### Distributed Version Control System

อย่างไรก็ตาม Centralized Version Control System มีข้อเสียสำคัญคือมันเป็นการทำงานแบบรวมศูนย์ หาก Server ล่มไปชั่วขณะ ผู้ใช้งานจะไม่สามารถ Check-In หรือ Check-Out งานได้ ถ้า Server เสียหายถาวร Version ของ Source Code ทั้งหมดก็ได้รับผลกระทบด้วย เพราะการดึงงานจาก Centralized Version Control System จะเป็นการ Check-Out เฉพาะเวอร์ชัน แบบ Online ขณะที่ Distributed Version Control System จะแก้ปัญหาการทำงานแบบรวมศูนย์โดยการ Clone Database มาทั้งหมด (โดยเราจะเรียก Database ที่จัดเก็บ Version ของ Source ว่า Repository)

ซึ่งเมื่อ Clone มาแล้วเราสามารถ Check-In และ Check-Out ใน Local Host แบบ Offline ก่อนจะ Push ขึ้นสู่ Server ในภายหลังโดยเราสามารถมอง Distributed Version Control System ได้ดังภาพด้านล่าง



รูปที่ . Distributed version control

Distributed Version Control System ที่สำคัญตัวหนึ่ง คือ BitKeeper ซึ่งเป็น Open Source โดย BitKeeper ได้ถูกนำมาใช้ในการเก็บ Source Code ของระบบปฏิบัติการ Linux ในปี 2002 แบบให้ใช้ฟรี อย่างไรก็ตามในปี 2005 BitKeeper ไม่ฟรีอีกต่อไป ดังนั้น Linus Torvalds ผู้ก่อตั้ง Linux จึงแก้ปัญหาโดยการพัฒนา Distributed Version Control System ขึ้นมาเองในชื่อว่า Git และจากจุดนี้เอง Git จึงถือกำเนิดขึ้นมาในปี 2005 และเปิดให้ใช้งานแบบ Open Source และ Free Software ตั้งแต่นั้นเป็นต้นมา

## [2] Git

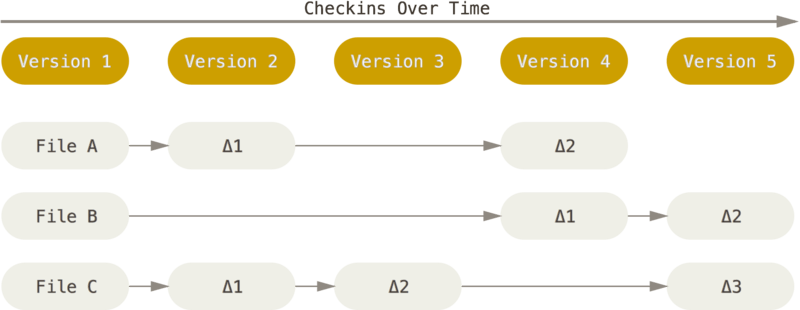
Git คือ เครื่องมือที่เอาไว้บันทึกเวอร์ชันการเปลี่ยนแปลงของไฟล์ในโปรเจคเรา ทำให้เราสามารถที่จะย้อนกลับไปดูเวอร์ชันต่าง ๆ ของโปรเจคได้ทุกเมื่อ หรือจะดูว่าใครเป็นคนแก้ไขไฟล์นั้น ๆ ได้ ซึ่ง Git นั้นจัดว่าเป็น Version Control แบบ Decentralized/Distributed คือไม่ต้องมีเซิร์ฟเวอร์เก็บโค้ดไว้ตรงกลาง ซึ่งจะต่างกับพวก Subversion หรือ CVS ที่ต้องมีเซิร์ฟเวอร์เก็บโค้ดรวมไว้ตรงกลาง (Centralized) และที่สำคัญ Git นั้นทำงานได้เร็วมากกว่าด้วย



รูปที่ . Git logo

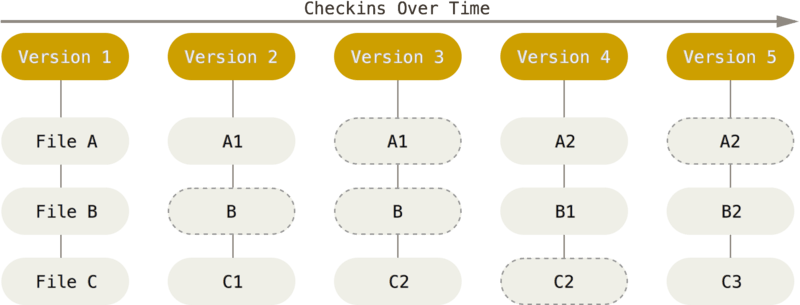
จากที่ได้กล่าวไว้แล้วว่า Git เป็น Distributed Version Control System ซึ่งต้องมีการ Copy Version ของ Source Code มาเก็บไว้ที่ Local host ก่อน ทำให้ผู้ใช้สามารถแก้ไข Project ได้ทุกที่แบบ Offline และ Check-In Source Code บน Local Repository โดยไม่ต้องติดต่อกับ Repository บน Server (Remote Repository) พอแก้ไขเสร็จแล้วก็ค่อยสั่ง Sync ข้อมูล (Pull/Merge/Push) ทีหลัง มันจึงทำงานได้เร็วกว่า SVN ที่เป็น Centralized Version Control System อย่างมาก

นอกจากนี้ Git ยังมีความแตกต่างกับ Version Control System อื่น ๆ ในเรื่องของการจัดเก็บข้อมูล Version โดย Version Control System ทั่วไปจะเก็บเป็น Link ของความเปลี่ยนแปลง หรือส่วนต่างระหว่าง Version ดังภาพด้านล่าง



รูปที่ . การจัดเก็บข้อมูลเวอร์ชันของ Git

แทนที่จะเก็บส่วนต่างระหว่าง Version, Git เก็บ Version ของ Source Code เป็น Snapshot และอ้างถึง File ที่ไม่มีการเปลี่ยนแปลงใน Version ก่อนหน้า ทุกครั้งที่มีการ Check-In มันจะถ่ายรูปไฟล์ของเราว่ามีหน้าตาเป็นอย่างไร (Snapshot) โดย Git มองข้อมูลที่จัดเก็บเป็นเหมือน File System เล็ก ๆ หนึ่ง ดังภาพด้านล่าง



รูปที่ . snapshots version control

จากในรูปส่วนที่เป็นเส้นประคือ snapshots version control ของ Git ก่อนที่จะถูก Save ทุกอย่างใน Version จะถูกหา Checksum โดยใช้ SHA-1 hash ดังนั้นจึงไม่มีทางที่เราจะเปลี่ยนแปลงข้อมูลใน File และ Folder โดย Git จะไม่รู้ เราจะใช้ค่า Checksum ในการอ้างถึง Version ของ Source Code ในหลายๆ งาน SHA-1 hash มีหน้าตาดังนี้

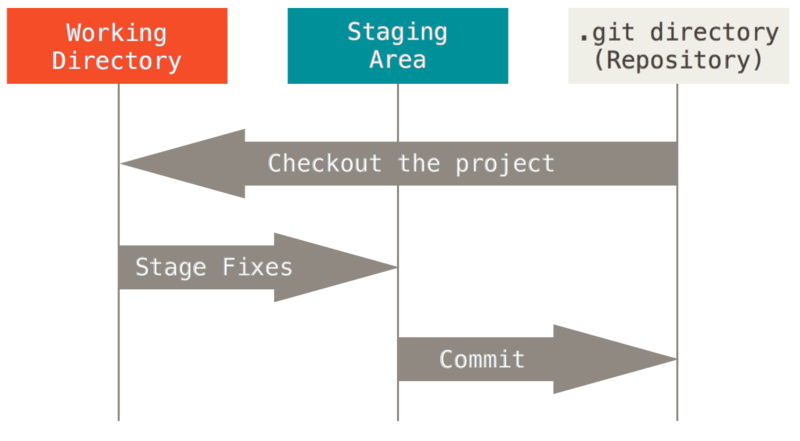
e70cdec636912065839fed45238db9d4f898f199

### Git Work Flow

แก่นของ Git ที่ต้องเข้าใจก่อนใช้งาน คือ Git แบ่งสถานะของไฟล์ที่มันจะติดตาม (Tracked) เป็น 3 สถานะ คือ Unstaged หรือ Modified, Staged และ Committed โดยมีรายละเอียดดังนี้

* สถานะ Unstaged หรือ Modified หมายถึงเราได้แก้ไขไฟล์ไปแล้ว แต่ยังไม่เริ่มกระบวนการจัดเก็บลง Repository
* สถานะ Staged หมายถึงเราได้ทำเครื่องหมาย File ที่ได้ถูกแก้ไข เพื่อจะบันทึกใน เวอร์ชันหน้า
* สถานะ Committed หมายถึงข้อมูลถูกบันทึกอย่างปลอดภัยใน Repository บน Local Host

ซึ่งจะแสดงการเปลี่ยนสถานะของไฟล์ในส่วนต่าง ๆ ดังรูปที่ 4.10



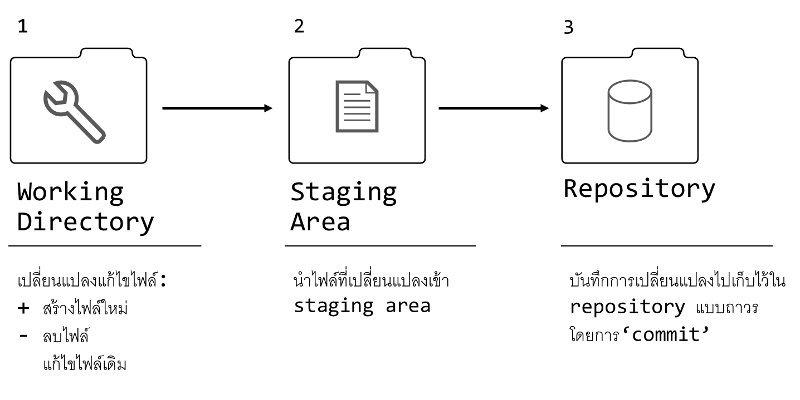
รูปที่ . Git Workflow

จากรูปจะเห็นได้ว่าใน Git project ใน Local Host จะประกอบด้วย 3 ส่วน

1. Working Directory เป็นที่เก็บไฟล์ที่เราทำงานทุกอย่าง ทั้งสร้างใหม่ แก้ไข และลบทิ้ง ไฟล์ที่แก้ไขจะอยู่ในสถานะ Unstaged หรือ Modified

2. Staging Area คือ ที่รวบรวมรายการที่แก้ไขไปทั้งหมดจาก working directory ที่จะนำเข้า repository รวบรวมไฟล์ที่ถูกแก้ไขลงใน Staging area เพื่อจะ Commit ในเวอร์ชันถัดไป ไฟล์ที่ถูกรวบรวมจะอยู่ในสถานะ Staged

3. Repository เป็นที่จัดเก็บไฟล์ที่มีการเปลี่ยนแปลงอย่างถาวร แยกเป็นเวอร์ชัน โดยการนำไฟล์จาก Staging Area ไปจัดเก็บอย่างถาวรใน Local Repository ไฟล์ที่ถูกบันทึกลง Repository อยู่ในสถานะ Committed



รูปที่ . Basic Git Workflow

ดังนั้นเราจึงกล่าวได้ว่าลำดับการทำงานของ Git จะประกอบด้วยการแก้ไขไฟล์ใน working directory แล้วนำไฟล์เหล่านั้นเข้า staging area สุดท้ายบันทึกการแก้ไขเข้า Git repository ใน Local Host นั่นเอง

### คำศัพท์ต่าง ๆ ที่จะต้องรู้จักเมื่อใช้งาน Git เบื้องต้น

เนื่องจากในการใช้งาน Git จะมีศัพท์เฉพาะตัวค่อนข้างมากและผู้ให้บริการ Git Server แต่ละเจ้าจะมีการเรียกใช้คำแตกต่างกันอีก ซึ่งขออธิบายไว้ดังนี้

#### Repository

เมื่อเราพัฒนาโปรแกรมเราจะสร้างสิ่งที่เรียกว่า Project ซึ่งการสร้าง Project สำหรับใช้งาน Git จะเรียกว่า Repository ซึ่ง Repository ของ Git คือโฟลเดอร์ที่ใช้เก็บข้อมูลนั่นเอง ทั้งนี้ในความเป็นจริง 1 Repository สามารถเก็บ Project เท่าไรก็ได้ตามที่ต้องการ แต่ส่วนใหญ่ก็นิยมเก็บ Project 1 ตัวต่อ 1 Repository

#### Clone

เวลาที่เรามี Repository อยู่บน Remote Server ซักแห่งอยู่แล้วและต้องการ Sync มาลงเครื่องของเรา เราจะต้องทำสิ่งที่เรียกว่า Clone Repository หรือก็คือการคัดลอก Repository จาก Remote มาลงเครื่องนั่นเอง

#### Commit

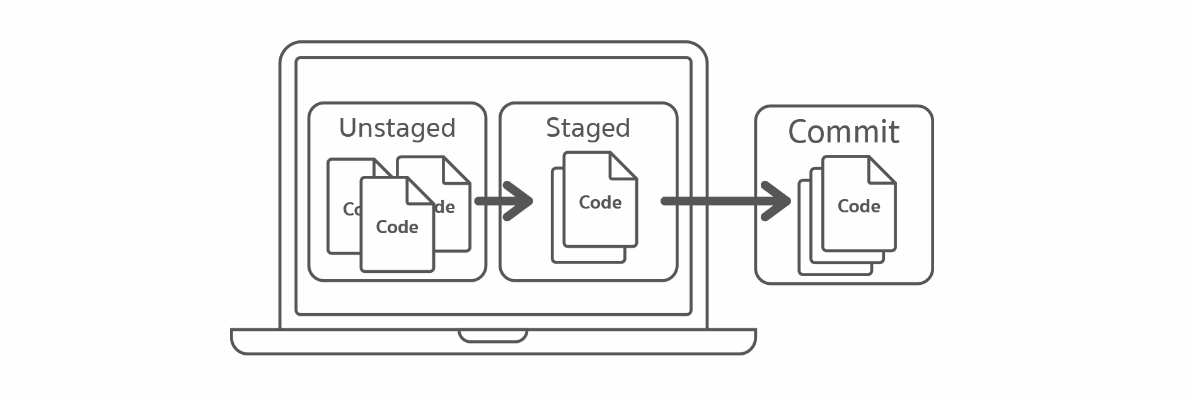
เวลาที่มีข้อมูลที่แก้ไขเสร็จแล้ว (โค้ดที่เขียนคำสั่งบางอย่างเสร็จแล้ว) แล้วอยากจะทำการ Backup เก็บไว้ใน VCS จะเรียกกันว่า Commit ซึ่งการ Commit จะสามารถเลือกได้ว่าจะเอาไฟล์ไหนบ้าง (ไม่จำเป็นต้องเลือกทุกไฟล์)

ซึ่งเบื้องหลังของ Commit ก็คือในแต่ละครั้งที่ทำการ Commit มันจะจำแค่ว่ามีตรงไหนของข้อมูลที่ถูกเปลี่ยนแปลงไปเรื่อย ๆ ดังนั้นในแต่ละ Commit มันจะไม่มีไฟล์ข้อมูลฉบับเต็ม แต่มันสามารถย้อนดู History ได้ว่ามีการแก้ไขอะไรบ้าง ทำให้รู้ว่าใน Commit นั้น ๆ แต่ละไฟล์มีข้อมูลเป็นอย่างไร

ในการ Commit แต่ละครั้ง จะต้องใส่ Commit Message ด้วย เพื่ออธิบายรายละเอียดของข้อมูลใน Commit นั้น ๆว่าเราทำอะไรไปบ้าง เพื่อที่มาดูในภายหลังจะได้อ่านจาก Commit Message ได้เลย ไม่ต้องไปนั่งกดดูเองทีละไฟล์ ดังนั้นการ Commit ที่ดีจึงควรใส่ใจกับ Commit Message ด้วย

#### Unstaged และ Staged

เมื่อเราทำการแก้ไขโค้ดหรือแก้ไขข้อมูล ไฟล์ที่ถูกแก้ไขจะอยู่ในสถานะ Unstaged หรือ Modified และเวลาที่เราทำอะไรเสร็จเรียบร้อย แล้วอยากจะ Commit เก็บไว้ จะต้องเลือกไฟล์ที่ต้องการเพื่อย้ายเข้าสู่ในสถานะ Staged ก่อนถึงจะทำการ Commit ได้ ซึ่งสถานะ Unstaged และ Staged ทำให้เราสามารถเลือกเฉพาะบางไฟล์สำหรับ Commit ได้นั่นเอง จะได้ Commit เฉพาะไฟล์ที่เราเขียนเสร็จแล้ว

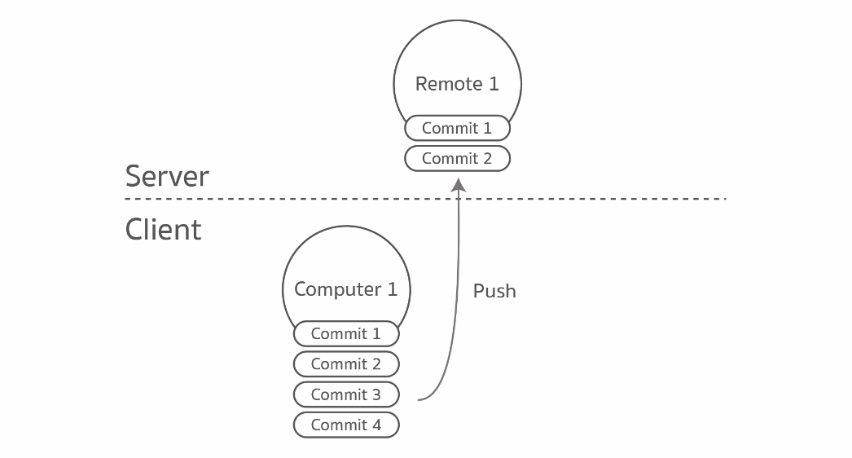


รูปที่ . สถานะ Unstaged และ Staged ใน Git

เมื่อเราทำการแก้ไขโค้ดไฟล์นั้นจะอยู่ในสถานะ Unstaged โดยอัตโนมัติ หากเราต้องการแบคอัพไฟล์นั้นใน Git บนเครื่องของเรา ต้องเลือกไฟล์ที่แก้ไขนั้นเข้ามาใน Staged เสียก่อนแล้วจึงทำการ Commit ได้ สำหรับ GitLab จะใช้คำสั่ง git add เพื่อทำการเลือกไฟล์เข้ามาใน Staged

#### Push

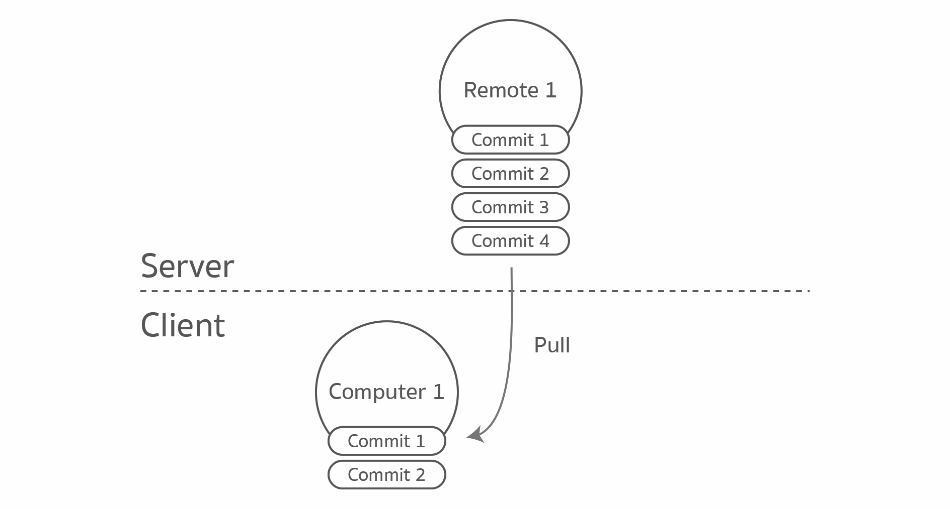
เวลาที่มี Commit อยู่ในเครื่องและต้องการจะ Sync ขึ้นไปเก็บไว้ใน Remote จะเรียกขั้นตอนนี้ว่า Push



รูปที่ . การ Sync ระหว่าง Local และ Remote

#### Pull

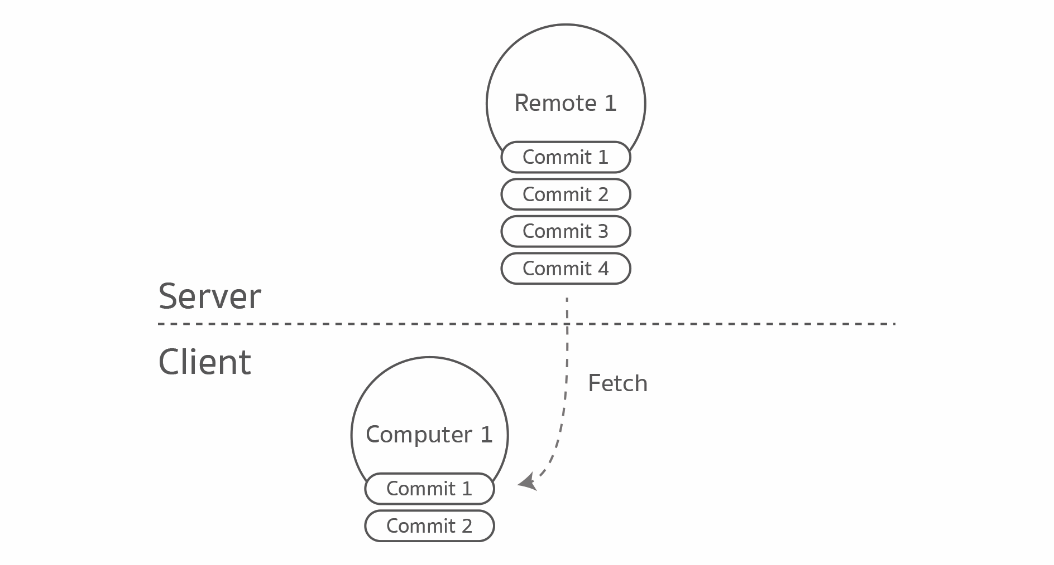
เวลา Sync จาก Remote เพื่อดึงข้อมูล Commit ใหม่ ๆ ลงมาเก็บไว้ในเครื่องจะเรียกขั้นตอนนี้ว่า Pull



รูปที่ . การดึงข้อมูลจาก Remote มายัง Local

#### Fetch

ในบางครั้งเราอาจจะไม่ต้องการ Pull ข้อมูลลงมาเก็บไว้ในเครื่องทันที แค่อยากเช็คสถานะของ Remote เฉย ๆ ว่ามีใคร Push ข้อมูลใหม่ขึ้นไปที่ Remote หรือไม่เราเรียกวิธีนี้ว่า Fetch ซึ่งการ Fetch จะทำให้เราสามารถอัพเดทและดู History ทั้งหมดที่อยู่บน Remote ได้ โดยไม่ต้องดึงข้อมูลลงมา

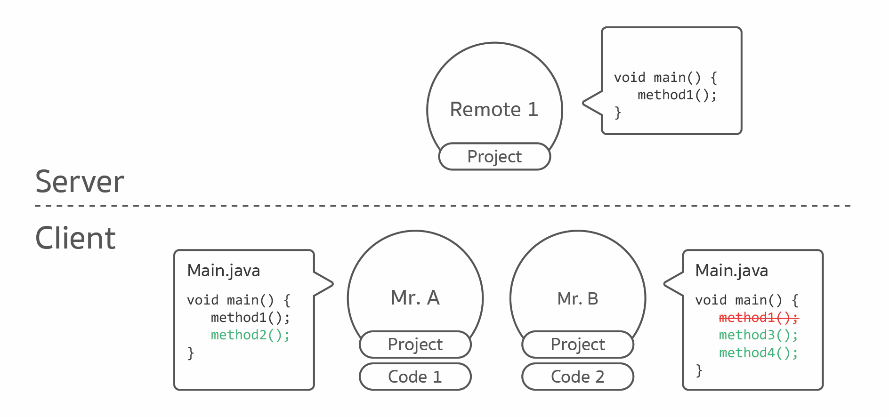


รูปที่ . การ Fetch เพื่อเรียกดูข้อมูลจาก Remote

จริง ๆ แล้วขั้นตอนการ Fetch จะถูกเรียกทุกครั้งที่ทำการ Pull ดังนั้นเราจึงสามารถเลือกได้ว่าจะ Pull เลยหรือจะ Fetch ดูก่อนว่ามี Commit อะไรเพิ่มเข้ามา แล้วค่อย Pull ลงมา

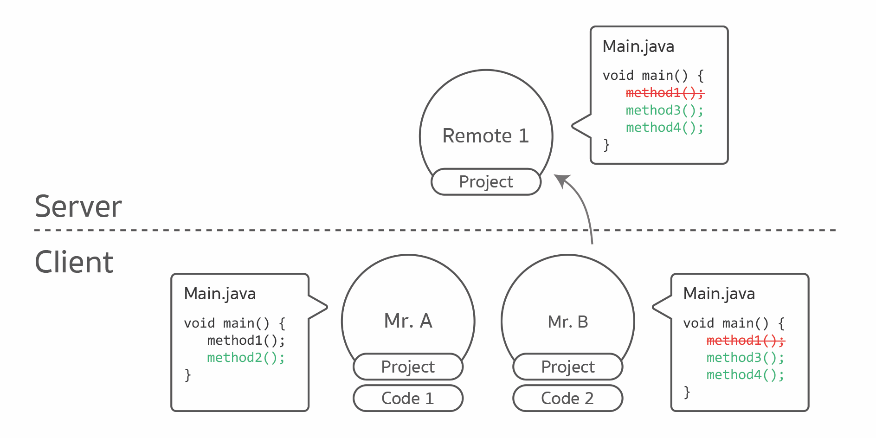
#### Merge Commit

สมมติว่านาย A กับนาย B เขียนโค้ดด้วยกันอยู่ และทั้งคู่ก็เขียนโค้ดที่อยู่ในไฟล์เดียวกัน



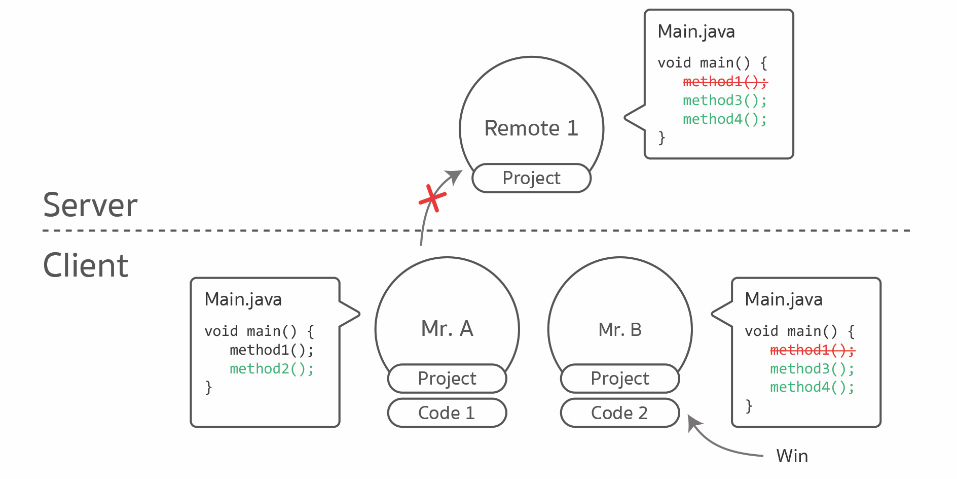
รูปที่ . ผู้ใช้จัดการไฟล์ไฟล์เดียวกัน

นาย A เพิ่ม method2 ต่อท้าย method1 ส่วนนาย B ลบ method1 ออก แล้วเพิ่ม method3 กับ method4 เข้าไป นาย B เขียนโค้ดเสร็จแล้ว ก็เลย Commit ไฟล์ที่ตัวเองเขียนเสร็จแล้ว Push ขึ้น Remote



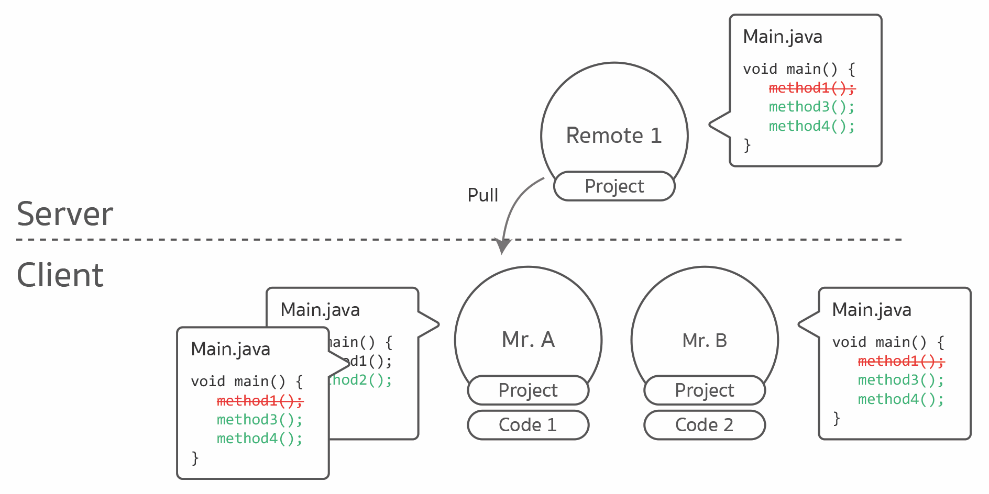
รูปที่ . นาย B ทำการ Push ข้อมูลขึ้น Remote

นาย A เขียนโค้ดเสร็จทีหลัง และจะ Push ขึ้น Remote แต่พบว่านาย B ได้ Push ขึ้นไปก่อนแล้ว จึง Push ขึ้นไปทันทีไม่ได้



รูปที่ . ปัญหาที่เกิดขึ้นทำให้นาย A ไม่สามารถ Push ไฟล์ได้

ดังนั้นสิ่งที่นาย A ต้องทำก่อนที่จะ Push ของตัวเองขึ้นไปได้ ก็คือจะต้อง Pull จาก Remote ลงมาใหม่ก่อนเพื่ออัพเดต Commit ที่นาย B ได้ Push ขึ้นไป ซึ่งเราเรียกขั้นตอนนี้ว่า Merge Commit นั่นเอง



รูปที่ . โค้ดของนาย A และ B แก้ที่บรรทัดเดียวกัน แต่โค้ดไม่เหมือนกัน

ในกรณีที่โค้ดสามารถ Merge กันได้ปกติ ยกตัวอย่างเช่น

// โค้ดเดิม

void main() {

method1();

}

// โค้ดที่นาย A แก้ไข

void main() {

method2();

method1();

}

// โค้ดที่นาย B แก้ไข

void main() {

method1();

method3();

}

เวลา Merge เข้าด้วยกันจะได้ผลลัพธ์ออกมาเป็นแบบนี้

void main() {

method2(); // โค้ดของนาย A

method1();

method3(); // โค้ดของนาย B

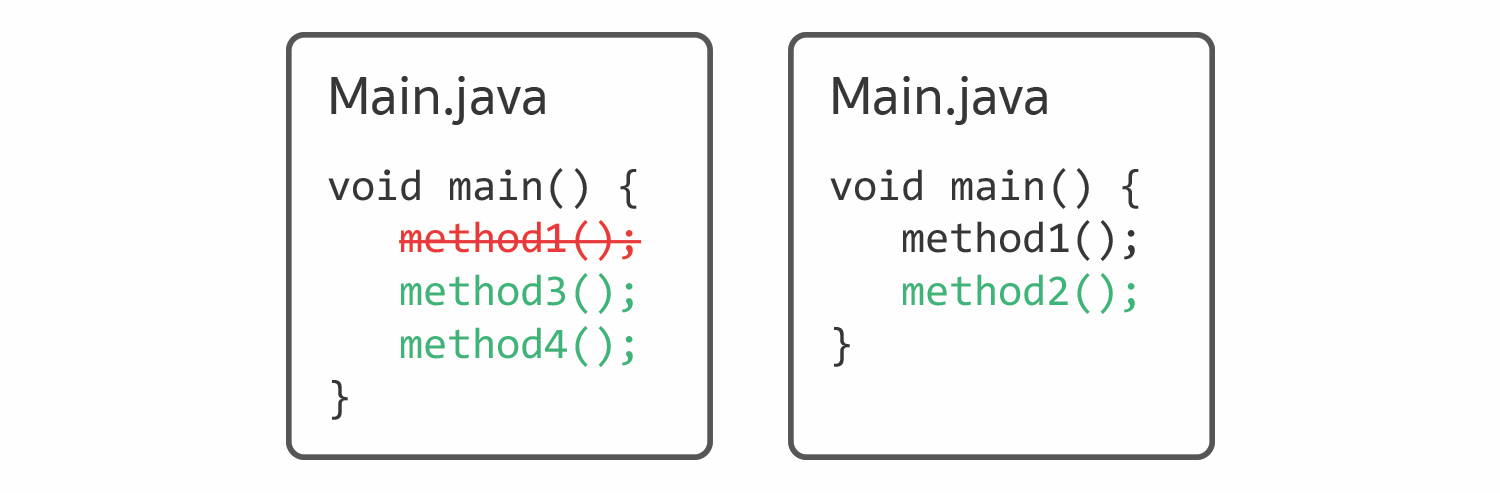
}

ซึ่งนี่คือกรณีที่โค้ดสามารถ Merge รวมกันได้โดยไม่มีปัญหาอะไร เมื่อ Merge เสร็จแล้ว มันก็จะกลายเป็น Commit ตัวหนึ่งที่ให้เราเก็บไว้และ Push ขึ้น Remote

ทั้งนี้ Git สามารถ Auto Merge ได้เลย แต่แนะนำให้ทำ Manual Merge จะดีกว่า เพื่อที่จะได้ Review Code ไปในตัวด้วย

#### Conflict

ในขณะที่กำลังจะ Merge อยู่นั้น จะเห็นว่าโค้ดของนาย A และนาย B มันชนกัน ไปแก้โค้ดที่เดียวกัน ดังนั้น Git จะแจ้งว่าเกิด Conflict หรือก็คือโค้ดที่ทับซ้อนกันนั่นเอง



รูปที่ . การเกิด Conflict ของไฟล์ที่ต้องการ Merge

ซึ่งนาย A ก็ต้องแก้ Conflict นี้ให้เรียบร้อยก่อนในตัวอย่างนี้นาย A จะทำการลบ method1() ของตัว A ทิ้งไปก่อนถึงจะ Merge Commit แล้ว Push ขึ้น Remote ได้ (นาย B ต้อง Pull Commit ตัวนี้ทีหลัง และถ้าเผลอไปแก้ไขอะไรที่เดิมซ้ำก็อาจจะเกิด Conflict ได้)

Conflict ถือว่าเป็นเหตุการณ์ที่เกิดขึ้นได้เป็นปกติ โดยเฉพาะอย่างยิ่งโปรเจคใหญ่ ๆ ที่มี Developer หลาย ๆ คนช่วยกันรุมเขียนโค้ด ดังนั้นการแก้ Conflict จึงเป็นหนึ่งในพื้นฐานของการใช้งาน Git ที่นักพัฒนาต้องเข้าใจและจัดการกับมันได้ ไม่เช่นนั้นจะเกิดปัญหาอย่างเช่น เผลอไปลบโค้ดของเพื่อน โดยไม่สนใจอะไร เพื่อให้ Conflict หายไป เป็นต้น

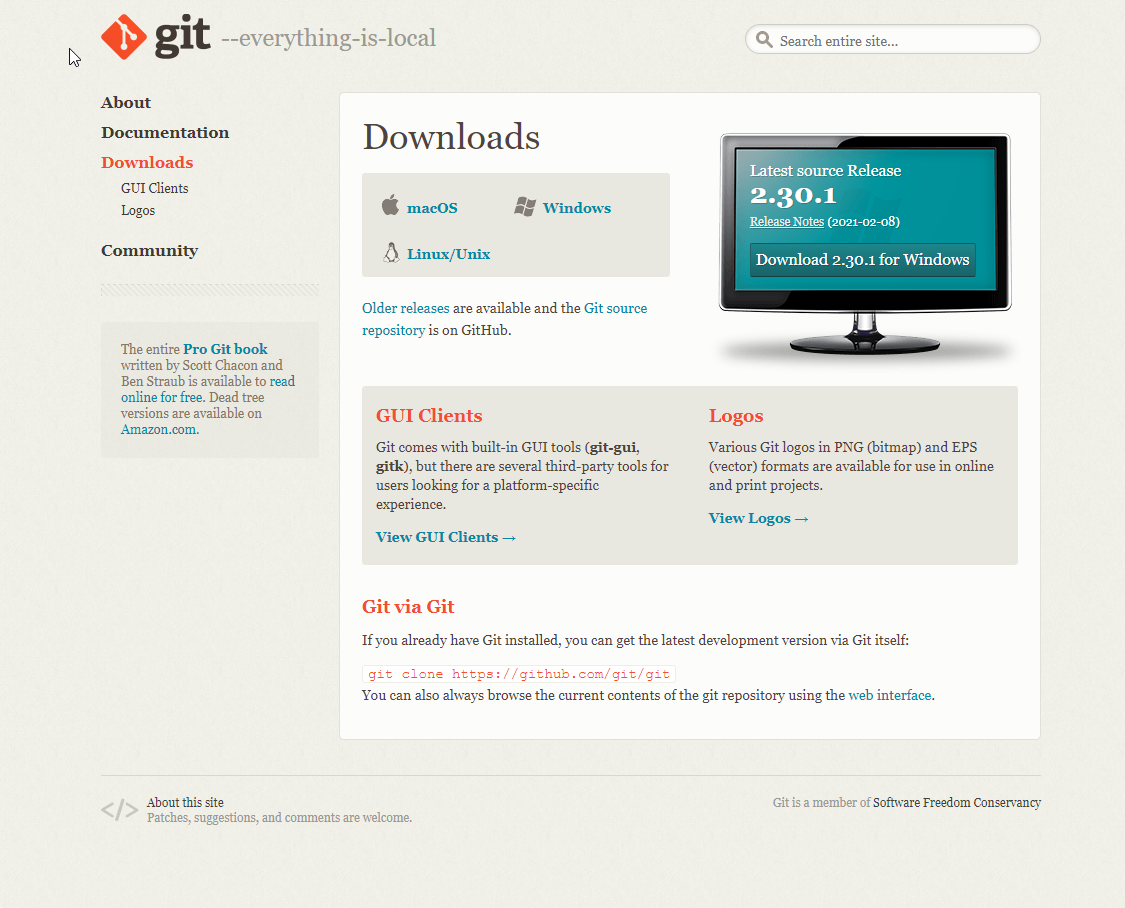
### การติดตั้ง Git

ก่อนจะใช้งาน Git บน Windows หรือ Mac OS เราจะต้องมีการติดตั้ง Git Client จากตัวเลือกต่อไปนี้

Windows ดาวน์โหลดตัวติดตั้งได้จาก https://git-scm.com/downloads

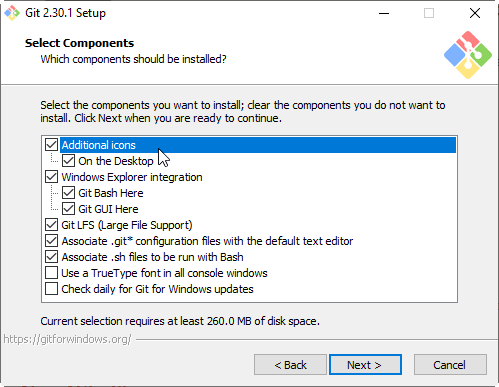
Mac OS X Yosemite or later ติดตั่งผ่าน Terminal ให้พิมพ์ $ git version

Linux ติดตั่งผ่าน Terminal ให้พิมพ์ $ sudo apt-get install git



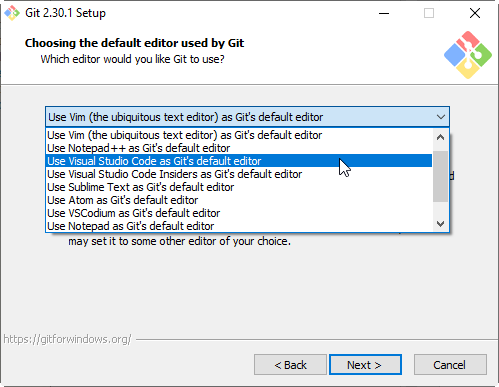
รูปที่ . เว็บไซต์ Git

Git มีให้เลือกการใช้งานด้วยกันสองแบบคือแบบ Command line และแบบที่ใช้ GUI ในกรณีที่ต้องการใช้งานแบบ GUI Client ให้เลือกเมนู Download > GUI Clients ซึ่งในเนื้อหาของบทนี้จะแสดงตัวอย่างแบบ Command line เท่านั้น เพื่อต้องการให้เกิดความใจว่า Git นั้นทำงานอย่างไร แต่ละขั้นตอนใช้คำสั่งอะไรบ้าง เมื่อเปลี่ยนไปใช้ Git แบบ GUI เช่น SourceTree หรือ GitHub Desktop จะได้จะรู้ว่าการทำงานของมันมีที่มาที่ไปอย่างไร แต่สุดท้ายเราจะเลือกใช้งานแบบ Command Line หรือ GUI ก็แล้วความถนัดของเรา



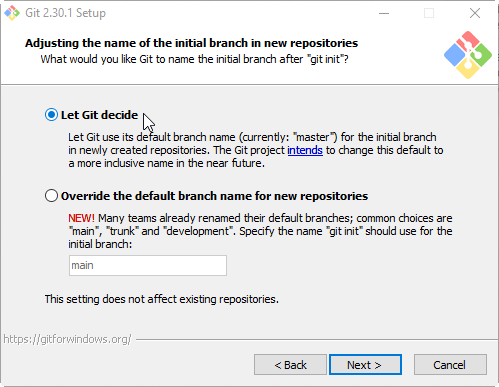
รูปที่ . เลือก Additional icons

ให้ทำการเลือกตัวเลือก Additional icons แบะ On the Desktop เพิ่มเติมเพื่อให้แสดงไอคอนของ Git Clients บนหน้าเดสก์ทอปของเครื่องคอมพิวเตอร์



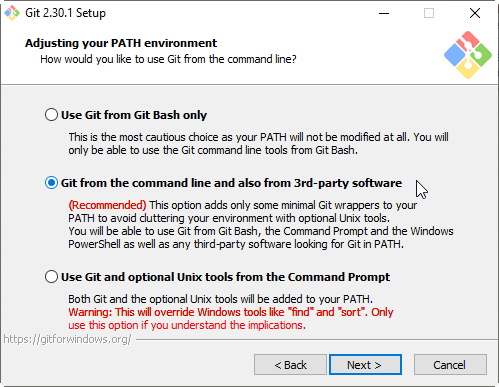
รูปที่ . เลือก Editor สำหรับจัดการคำสั่งของ Git

ในที่นี้ให้เลือก Editor ที่ถนัดใช้งานที่มีอยู่ภายในเครื่อง ในที่นี้เลือกใช้ VSCode



รูปที่ . กำหนดชื่อ branch หลักของ Git

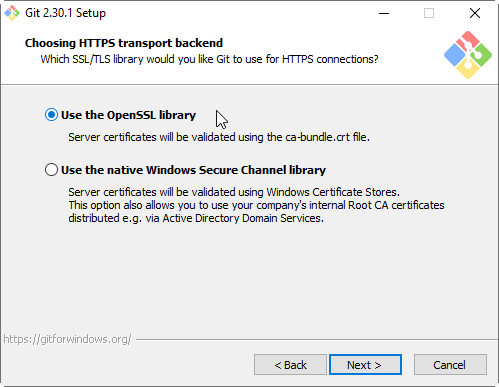
โดยปรกติแล้ว Git จะกำหนดชื่อ branch หลักในการทำงานเป็น master แต่เนื่องจากเรื่องการเหยียดสีผิวในอเมริกาใน ค.ศ. 2020 ซึ่งเป็นเรื่องที่อ่อนไหวในอเมริกาอย่างมาก ทาง Git จึงแก้ไขให้ผู้ใช้งานสามารถเปลี่ยนคำว่า master เป็นคำอื่นได้เนื่องจากมองว่าเป็นคำที่ส่อไปในทางการค้าทาสสมัยก่อน หากต้องการใช้คำว่า master ตามเดิมให้เลือกตัวเลือก Let Git decide หากต้องการกำหนดเป็นชื่ออื่นให้เลือกตัวเลือกด้านล่างและกำหนดชื่อใหม่ เช่น main เป็นต้น



รูปที่ . เลือกโปรแกรมที่ใช้เรียกคำสั่ง Git ทางคอมมานด์ไลน์

โดยทั่วไปเราจะใช้ Git Bash ในการรับคำสั่งทางคอมมานด์ไลน์ของ Git ซึ่งตัวเลือกที่สองที่แนะนำโดย Git ก็สามารถใช้งาน Git Bash ได้เช่นกันดังนั้นจึงแนะนำให้ใช้ตัวเลือกที่ Git แนะนำมา โดยมีรายละเอียดแต่ละตัวเลือกดังต่อไปนี้

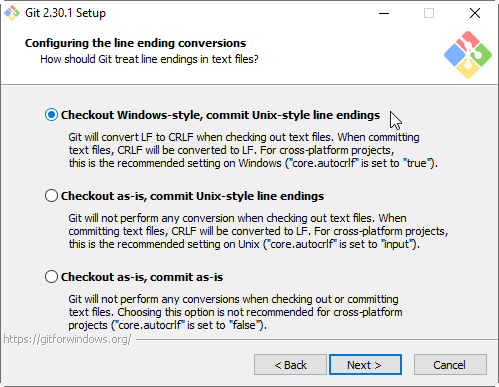
* Use Git from Git Bash only: ใช้งาน Git ผ่าน Command line ต้องเปิดจาก Git Bash เท่านั้น
* Git from the Command line and also from 3rd-party software: ตัวเลือกนี้ สามารถใช้คำสั่ง Git ผ่านทาง Command Prompt ของ Windows ได้เลยหรือจะใช้ Git Bash ที่จะถูกติดตั้งมาพร้อมกับ Git Clients ก็ได้
* Use Git and optional Unix tools from the Command Prompt: ตัวเลือกนี้ จะเหมือนตัวเลือกที่สอง แต่จะสามารถใช้คำสั่งของ Unix ได้ด้วย ซึ่งอาจมีบางคำสั่ง ที่จะไปทับคำสั่งเดิมของ Command Prompt ได้ (ไม่แนะนำตัวเลือกนี้ เพราะอาจไปกวนการทำงานของ Windows หรือโปรแกรมอื่น ๆ บางโปรแกรม)



รูปที่ . เลือก HTTPS transport ที่จะให้ใช้งาน

รายละเอียดสำหรับแต่ละตัวเลือกจะเป็นดังนี้

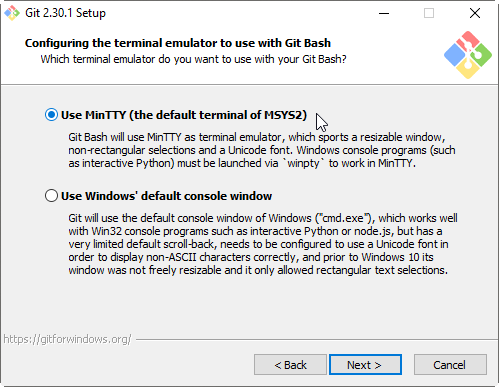
* Use the OpenSSL library: ใช้การตรวจสอบ Server certificates โดยไฟล์ ca-bundle.crt ในที่นี้ให้เลือกตัวเลือกนี้เนื่องจากเราไม่ได้ใช้งานในนามบริษัท
* Use the native Windows Secure Channel library: ใช้การตรวจสอบ Server certificates โดย Windows Certificate Stores ซึ่งสามารถใช้ใบรับรองของบริษัทได้



รูปที่ . เลือกรูปแบบการเข้ารหัสในการจบบรรทัด

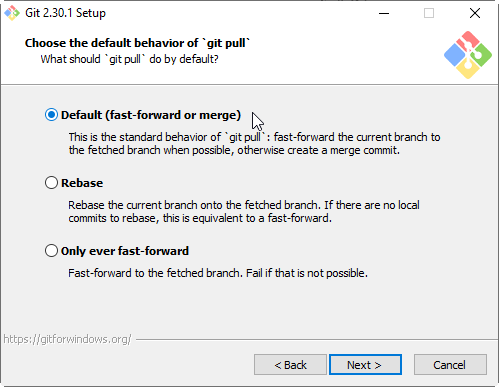
ในระบบปฏิบัติการแต่ละตัวจะมีรูปแบบในการเข้ารหัสไฟล์สำหรับการจบบรรทัดและขึ้นบรรทัดใหม่แตกต่างกันดังนั้นตัวเลือกนี้จึงต้องเลือกให้สอดคล้องกับระบบปฏิบัติการและเซอร์เวอร์ที่ใช้งานอยู่ โดยมีรายละเอียดดังต่อไปนี้

* Checkout Windows-style, commit Unix-style line endings: เมื่อเอา Source Code ออกมาจาก Git จะได้เป็น CRLF ซึ่งเป็นรูปแบบที่อยู่ในระบบปฏิบัติการวินโดวส์ แต่เมื่อให้ Git เก็บ Source Code จะเก็บเป็น LF ซึ่งเป็นรูปแบบการเข้ารหัสที่ใช้ในระบบปฏิบัติการยูนิกซ์และลีนุกซ์
* Checkout as-is, commit Unix-style line endings : เมื่อเอา Source Code ออกมาจาก Git จะไม่มีการแปลงใด ๆ แต่เมื่อให้ Git เก็บ Source Code จะเก็บเป็น LF
* Checkout as-is, commit as-is: เมื่อเอา Source Code เข้าและออก จะไม่มีการแปลงใด ๆ



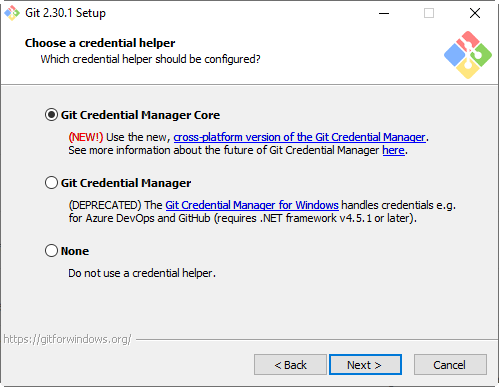
รูปที่ . เลือกตัวจำลอง terminal ที่จะใช้ใน Git Bash

ให้เลือกตัวเลือก Use MinTTY (the default terminal of MSYS2) เนื่องจากตัวเลือก Use Windows' default console window﻿ มีข้อจำกัดในการแสดงผล



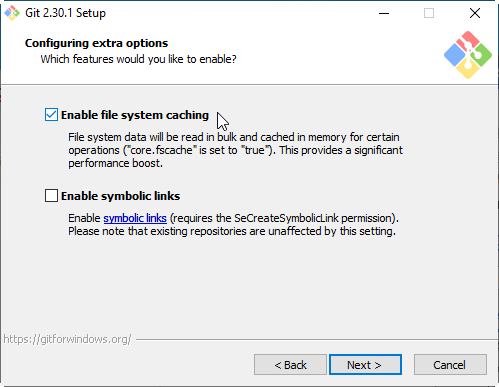
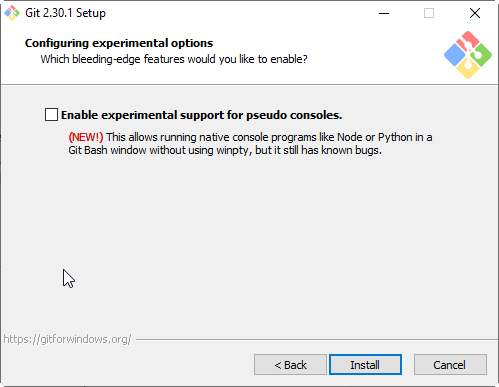
รูปที่ . รูปแบบการดึงโค้ดจาก Git

โดยปรกติหากต้องการดึงโค้ดจาก Git โดยทำการดึงโค้ดใหม่ที่เพิ่มเติม และการรวมโค้ดใหม่เข้ากับโค้ดเก่า (Merge) ที่มีอยู่เดิมจะใช้ตัวเลือกแรก แต่หากต้องการสร้างสายงานหรือ Branch ใหม่ซึ่งจะเป็นการเปลี่ยน master branch ใหม่ทุกครั้งให้เลือกตัวเลือกที่สอง ส่วนตัวเลือกที่สามจะคล้ายกับตัวเลือกแรกเพียงแต่ไม่มีการรวมโค้ดเข้าด้วยกัน



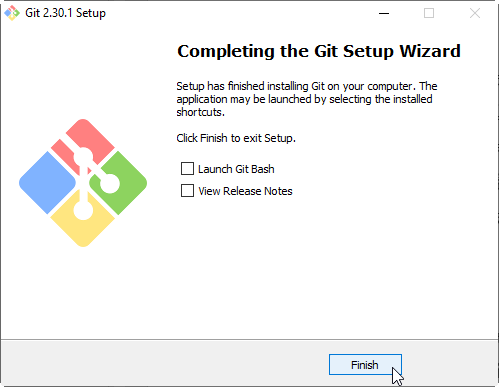
รูปที่ . ตัวจัดการข้อมูลรับรองการเข้าใช้งาน

Credential Helper จะเป็นระบบที่ติดมากับ Git โดยจะเก็บ Username และรหัสผ่านเอาไว้ ทำให้ไม่ต้องล็อกอินทุกครั้งที่ใช้งานโปรโตคอล https ทั้งนี้การเก็บข้อมูลของ Username และรหัสผ่านจะขึ้นอยู่กับแต่ละระบบปฏิบัติการที่ใช้ ในที่นี้ให้เลือกตัวเลือกแรก

รูปที่ . เลือกเปิดใช้งานแคชสำหรับไฟล์ System

ข้อมูลของระบบไฟล์ต่าง ๆ ใน Git จะถูกอ่านเป็นกลุ่ม ๆ และพักข้อมูลเหล่านี้ไว้ในหน่วยความจำเพื่อนำไปใช้งานต่อไปในภายหลัง ซึ่งจะเพิ่มประสิทธิภาพในการทำงานอย่างมาก สำหรับตัวเลือกที่สองเป็นการเปิดการใช้งาน Symbolic Link ซึ่งเป็นการสร้างตัวอ้างอิงจากไฟล์ที่มีอยู่แล้ว ทำให้เมื่อไฟล์ต้นฉบับถูกลบ ข้อมูลในส่วนนั้นก็จะไม่สามารถเข้าถึงได้จาก Link ที่สร้างไว้ได้ทั้งนี้การสร้าง Symbolic Link สามารถสร้างบนระบบไฟล์ที่แตกต่างกันได้ การใช้งานความสามารถนี้ระบบปฏิบัติการต้องเปิดสิทธิ์การใช้งานไว้ก่อนแล้ว สำหรับตัวเลือกที่สามที่อยู่ในหน้าถัดไปเป็นการเปิดใช้งานหากต้องการใช้งาน Python หรือ Node



รูปที่ . การติดตั้ง Git Clients เสร็จสมบูรณ์

## [3] Git Branch

Version Control System เกือบทุกตัวจะมี Branch Feature ให้ใช้งานในรูปหนึ่งรูปแบบใด แต่การแตก Branch ใน Version Control System เหล่านั้นค่อนข้างมีค่าใช้จ่ายที่สูงเหมือนกับการทำสำเนาซอร์สโค้ดทั้ง Folder ยิ่ง Project มีขนาดใหญ่ยิ่งใช้เวลาในการสร้าง Branch หลายวินาที หรือเป็นนาที แต่ Branch ใน Git เป็นอะไรที่แตกต่างกัน การสร้าง Branch ของ Git นั้นรวดเร็วอย่างไม่น่าเชื่อ และใช้ทรัพยากรน้อยกว่า

### Branch

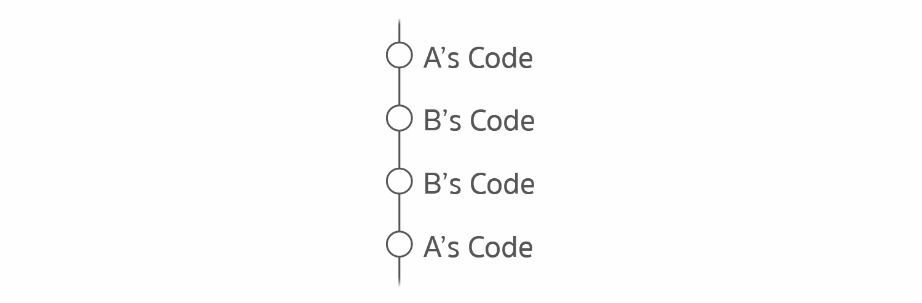
เวลาที่เรา Commit เรื่อย ๆ จนกลายเป็น History มันจะมีหน้าตาดังที่แสดงไว้ในรูปที่ 4.33



รูปที่ . History ใน Git

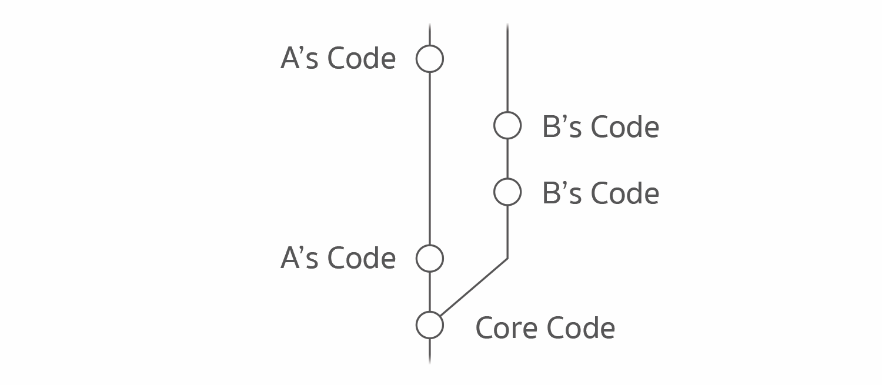
แต่ละ Commit จะเรียงต่อกันเป็น Node จนกลายเป็นเส้นยาว ๆ การเรียงกันของ Commit จนเป็นเส้นยาว ๆ แบบนี้จะเรียกว่า Branch และในการใช้งาน Git จริง ๆ จะไม่ได้มีแค่ Branch เดียวเท่านั้น

จะเกิดอะไรขึ้นถ้านาย A และนาย B ทำโปรเจคเดียวกัน แต่ว่าทำคนละ Feature ถ้าเราทำงานกันแบบ Branch เดียว เราก็จะพบว่านาย A กับนาย B ต้องเสียเวลา Pull ข้อมูลของกันและกันบ่อยมาก ทั้ง ๆ ที่โค้ดของทั้งสองไม่ได้เกี่ยวข้องกันซักนิดดังที่แสดงไว้ในรูปที่ 4.34



รูปที่ . โค้ดไม่ได้เกี่ยวข้องกัน แต่ต้องมาอยู่บน Branch เดียวกัน

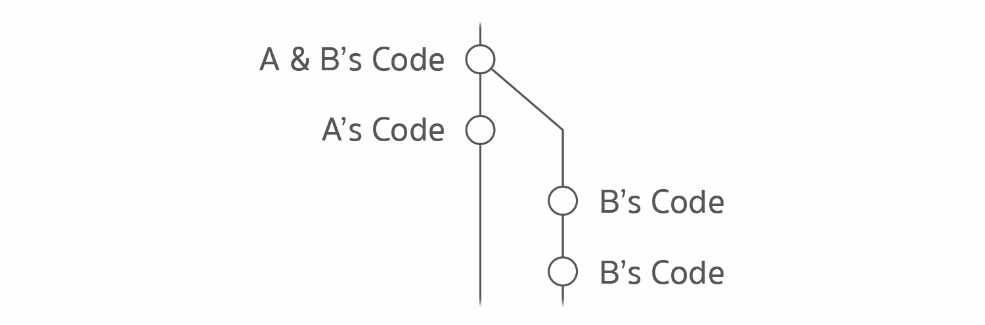
ซึ่ง Git ออกแบบมาให้สร้าง Branch แยกออกมาได้หลายเส้น ดังนั้นในกรณีดังกล่าวเราจึงควรแยก Branch ออกมาสำหรับนาย A และอีก Branch สำหรับนาย B



รูปที่ . แยก Branch สำหรับการทำงานคนละ Feature

#### Merge Branch

เมื่อคนในทีมเขียนโค้ดสำหรับแต่ละ Feature เสร็จลำดับต่อไปจะถึงเวลาของการรวมกันของ Branch สองตัวที่ได้แยกกันไปพัฒนา

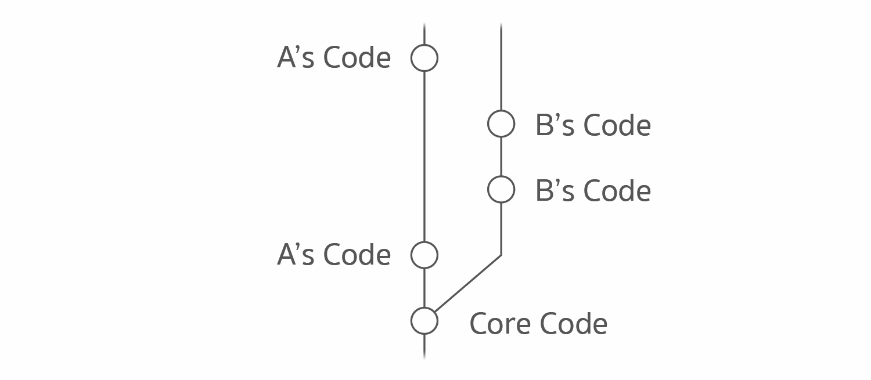


รูปที่ . ผลลัพธ์ที่ได้จากการ Merge Branch ของ A และ B

ในระหว่างการ Merge Branch อาจจะมี Conflict เกิดขึ้น แต่เนื่องจากการแยก Branch มักจะใช้สำหรับกรณีที่เขียนโค้ดคนละส่วนกันอยู่แล้ว ดังนั้นตำแหน่งของโค้ดที่เกิด Conflict ก็จะมีเพียงบางจุดเท่านั้น และเมื่อแก้ไข Conflict จนเสร็จแล้ว Merge Branch ทั้งสองเข้าด้วยกันได้แล้ว มันก็จะได้ออกมาเป็น Merge Commit นั่นเอง

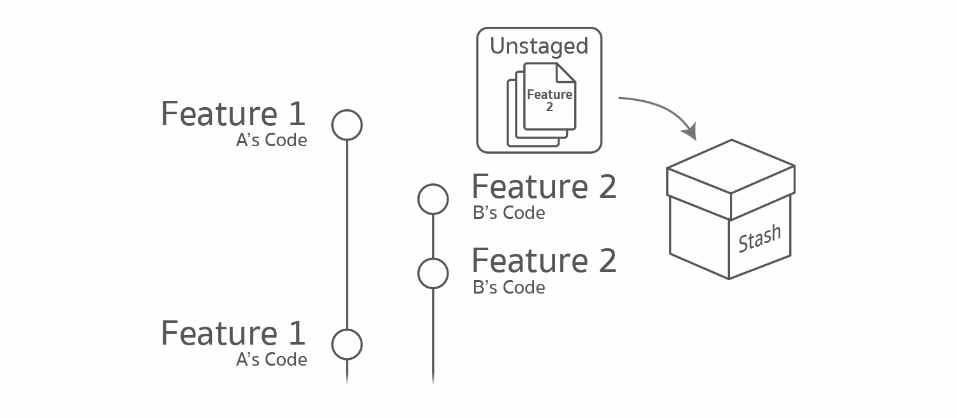
#### Stash และ Unstash

สมมติว่าโปรเจคตัวหนึ่งถูกแยกออกมาเป็น 2 Branch ด้วยกัน



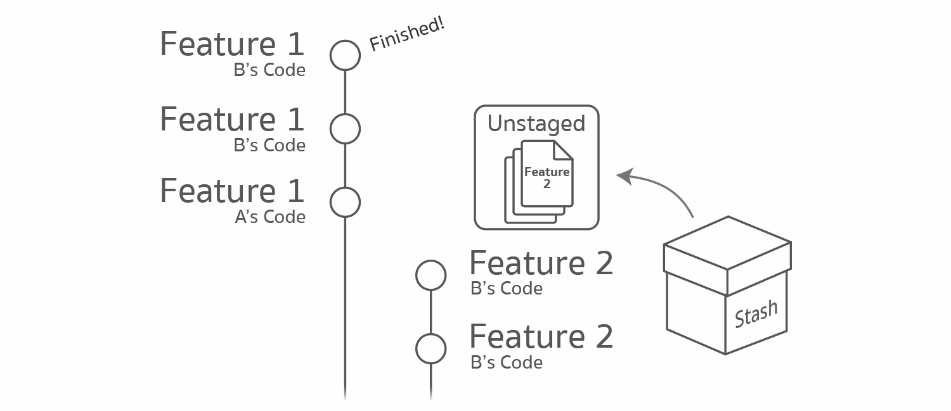
รูปที่ . Branch ที่แยกกันพัฒนาของนาย A และ B

แต่อยู่ดี ๆ ก็เกิดเหตุด่วน เพราะนาย A ดันเขียนโค้ดไม่ทัน แล้ว Feature ของนาย A ต้องส่งให้ลูกค้าในวันพรุ่งนี้ จึงทำให้นาย B ที่เขียนโค้ดใน Feature ของตัวเองอยู่ ก็ต้องหยุดชะงักกลางคันเพื่อสลับไปช่วยนาย A เขียนให้เสร็จ แต่มีบางไฟล์ที่นาย B เขียนค้างไว้ยังไม่เสร็จ แถมจะ Commit เลยก็ทำไม่ได้ อีกทั้งยังต้องสลับไป Branch ของนาย A ด่วนด้วย ดังนั้นนาย B จะต้องเอาไฟล์ที่อยู่ใน Unstaged ไปเก็บไว้ใน Stash ชั่วคราวก่อน จะได้สลับ Branch ไปช่วยนาย A เขียนโค้ดให้เสร็จทันเวลา



รูปที่ . เก็บโค้ดที่เขียนยังไม่เสร็จลงใน Stash ไว้ชั่วคราวก่อน

พอนาย B ไปช่วยนาย A เขียนจนเสร็จแล้ว ก็กลับมา Branch ของตัวเองแล้ว Unstash เพื่อเอาไฟล์ที่เคยเขียนค้างไว้กลับมาทำต่อให้เสร็จ



รูปที่ . Unstash ของเก่ามาเขียนต่อ

นอกจากนี้ Stash ยังสามารถใช้ย้ายไฟล์ข้าม Branch ได้อีกด้วย

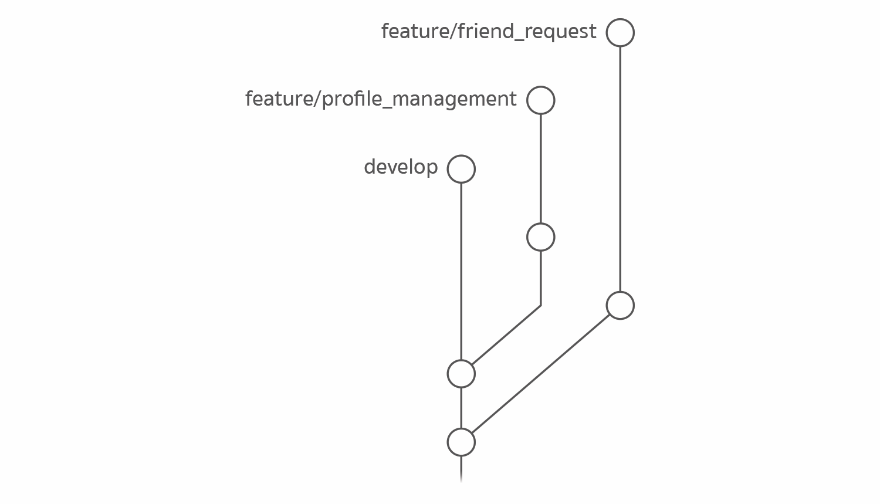
## [4] Git Flow

ในขั้นตอนการพัฒนาซอฟต์แวร์โดยทั่วไปจะเป็นการพัฒนาหลาย Feature พร้อม ๆ กัน แต่เวลาขึ้น Production จะหยิบเฉพาะบาง Feature เท่านั้น ส่วน Feature ที่เหลือก็อาจจะพัฒนาต่อหรือปล่อยทิ้งไว้รอขึ้นในรอบหน้า รวมไปถึงขั้นตอนก่อนที่จะขึ้น Production ที่จะต้องให้ QA/Tester ทดสอบก่อน หรือการทำ Hotfix เพราะเจอบั๊กตัวใหญ่ใน Production ที่ต้องแก้ด่วน ดังนั้นเราจึงสามารถใช้ความสามารถในการแยก Branch ของ Git มาช่วยแก้ปัญหานี้ได้ และเมื่อวิธีแบบนี้นิยมมากขึ้นเรื่อย ๆ ก็ทำให้เกิดสิ่งที่เรียกว่า Git Flow ขึ้นมา

Git Flow คือหนึ่งในมาตรฐานที่กำหนดรูปแบบของการแตก Branch ให้เป็นระบบระเบียบเพื่อตอบโจทย์ใน Development Process โดยแบ่ง Branch ออกเป็น 5 กลุ่มใหญ่ ๆ ดังนี้

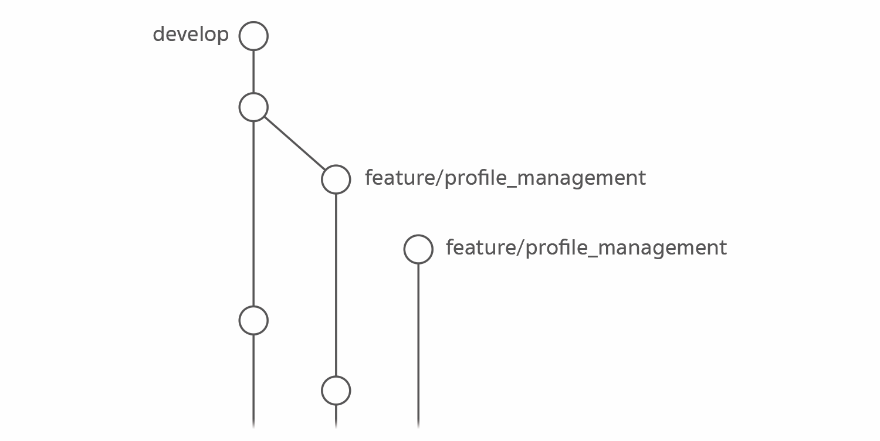
* Master
* Develop
* Feature
* Release
* hotfix

ขั้นตอนการเขียนโค้ดทั้งหมดจะอยู่ใน develop และ feature อยู่เสมอ โดยตอนขึ้นโครงโปรเจคในตอนเริ่มต้นอาจจะทำใน develop แต่เมื่อโค้ดพื้นฐานพร้อมแล้ว ก็จะแยก Branch ตาม Feature เพื่อเขียนโค้ด และนั่นก็คือ branch ที่ชื่อว่า feature นั่นเอง



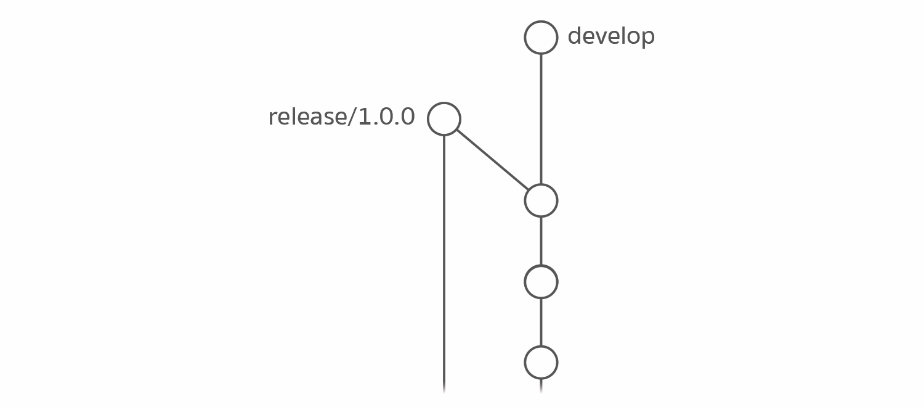
รูปที่ . develop มีตัวเดียว แต่ feature จะแยกเป็น Sub Branch โดยใช้ชื่อตาม Feature นั้น ๆ

เมื่อพัฒนา Feature ตัวไหนเสร็จ และพร้อมจะ Release ก็จะ Merge เข้าไปใน develop เลย ส่วนตัวไหนที่ยังไม่พร้อม Release ก็เก็บการทำงานส่วนนั้นไว้ก่อนยังไม่ต้อง Merge



รูปที่ . เลือก Merge เฉพาะ Branch ที่พร้อมใช้งานเท่านั้น

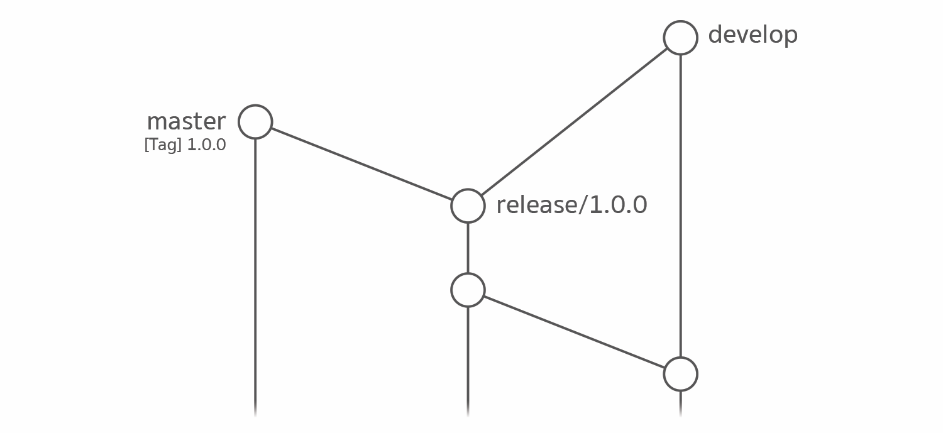
Merge Branch แบบนี้ ใน Git Flow จะเรียกว่า Finish Branch และเมื่อถึงเวลาอันสมควรที่จะ Release เพื่อเตรียมขึ้น Production แล้ว ก็จะแตก Branch ออกมาเป็น release โดยมี Sub Branch เป็นชื่อเวอร์ชันที่ต้องการ



รูปที่ . Release Branch

ซึ่งการแตกเข้าสู่ release นี้มีไว้เพื่อให้ QA/Tester ทำการทดสอบโปรแกรมของเราให้ผ่านก่อนที่จะขึ้น Production นั่นเอง ดังนั้นถ้ามี Issue เกิดขึ้นระหว่างนี้ก็ต้องแก้ไขในนี้ให้เรียบร้อย

เมื่อผ่านขั้นตอนของ QA/Tester เรียบร้อยแล้ว ก็ถึงเวลาพร้อมที่จะเข้าสู่ master

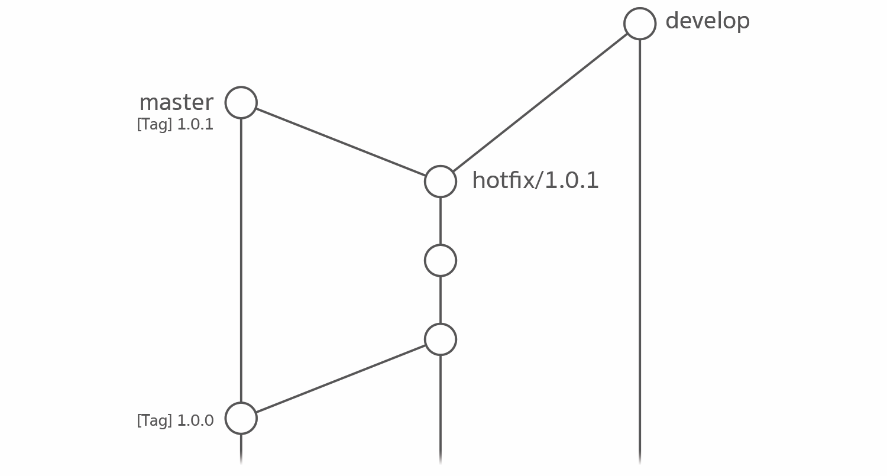


รูปที่ . การ Merge Release Branch กลับเข้าสู่ Master Branch

สำหรับการ Finish Branch ของ release จะต้อง Merge เข้า master และ develop ทุกครั้ง สาเหตุที่ต้อง Merge เข้า develop ด้วยก็เพราะว่ามันมีโค้ดที่แก้ Issue ในระหว่างการทดสอบของ QA/Tester เพิ่มเข้ามา

จะเห็นว่าที่ master นั้นจะเป็นโค้ดที่สมบูรณ์ที่สุด (ของเวอร์ชันนั้น ๆ) และจะไม่มีการ Commit เข้าไปตรง ๆ เลย แต่จะมาจากการ Merge ของ release เท่านั้น ดังนั้น master จึงมีแต่โค้ดที่เสร็จสมบูรณ์ เวลาจะ Rollback แบบกระทันหันก็สามารถทำได้โดยไม่ต้องกังวลว่าจะมีปัญหาอะไรหรือป่าว

ในกรณีที่ตัว Production ตัวล่าสุดเกิด Critical Issue ขึ้นมา แต่ไม่ถึงกับต้อง Rollback ก็จะมี hotfix เพื่อเข้ามาแก้ปัญหานี้ให้โดยเฉพาะ



รูปที่ . สร้าง Branch Hotfix เพื่อแก้ไขบั๊ก

หน้าที่ของ hotfix มีไว้แก้บั๊กบน Production แบบด่วน โดย hotfix จะมี Sub Branch เป็นชื่อเวอร์ชัน (นิยมเปลี่ยนเฉพาะเลขของ Patch Version) เมื่อแก้ไขเสร็จแล้วก็จะ Merge เข้าสู่ master และ develop เหมือนกับ release

นี่คือรูปแบบทั้งหมดของ Git Flow ที่ถูกออกแบบมาให้ตอบโจทย์ในการพัฒนาโปรเจคใหญ่ ๆ โดยจะเห็นว่า Git Flow จะช่วยให้เขียนโค้ดไม่ซับซ้อนวุ่นวาย บริหารและจัดการ Feature ต่าง ๆ ได้ง่าย รองรับ Process ที่ต้องมีการผ่าน QA/Tester ก่อนจะขึ้นสู่ Production ด้วย และตอบโจทย์สำหรับการรับมือกับ Issue ต่าง ๆ ที่สามารถเกิดขึ้นได้ตลอดเวลา

### Fork

โดยปกติแล้ว คนที่สามารถเข้าไป Commit และ Push ใน Repository ได้จะต้องเป็นคนที่ได้รับอนุญาตเท่านั้น ถึงแม้ว่าจะเป็น Public Repository ก็ตาม

สมมติว่าเราไปใช้ Library ซักตัวหนึ่งที่มี Repository อยู่บน GitHub แล้ว Library ตัวนั้นดันมีบั๊กอยู่ แต่คนดูแลไม่ว่างมาแก้ไขให้ซักที ส่วนตัวเราอยากจะใช้งานทันทีเพื่อให้งานเสร็จ ดังนั้นเราจึงต้องทำการ Fork Repository ตัวนั้นมาไว้ที่ GitHub ของเราแล้วทำการแก้ไขให้เรียบร้อย ซึ่งการ Fork ก็คือการคัดลอก Repository (ของคนอื่นที่อนุญาตให้เราเข้าไปดูได้) มาเก็บไว้เป็น Repository ของเรา ซึ่งทุก ๆ อย่างจะเหมือนกันทั้งหมด ต่างกันแค่ว่าเราเป็นเจ้าของ Repository ตัวที่แยกออกมาเท่านั้นเอง ดังนั้นเราจึงสามารถแก้ไขบั๊กตัวนั้นจากใน Repository ของเราได้ทันที

### Pull Request

เมื่อเราแก้บั๊กใน Repository ที่ Fork มาจาก Library ของคนอื่นแล้ว แต่พบว่าคนอื่นที่เอา Library ของคน ๆ นั้นไปก็ยังเจอบั๊กเหมือนเราอยู่ดี ก็เลยอยากจะช่วยแก้บั๊กนี้ใน Library ของต้นฉบับ ดังนั้นเราจึงส่ง Pull Request ไปให้ต้นทางเพื่อให้เจ้าของ Library เห็น Commit ของเราที่แก้บั๊กตัวนั้น เจ้าของ Library ที่เห็น Commit ของเราจะมีทางเลือกอยู่สองทาง คือ ปล่อยทิ้งไว้ให้มีบั๊กแบบเดิม หรือรับ Pull Request ของเราเพื่อรวม Commit ที่แก้บั๊กแล้ว เข้าไปใน Repository ของเจ้าของต้นฉบับ ซึ่งจะทำให้เรากลายหนึ่งใน Contributor ของ Library ตัวนี้ไปด้วยและนี่ก็คือโลกของ Open Source นั่นเอง

## [5] การเลือกใช้ผู้ให้บริการ Git

การใช้ Git ให้เกิดประโยชน์ที่สุดก็คงต้องมีเซิร์ฟเวอร์ที่ทำหน้าที่เป็น VCS ด้วย ซึ่งในปัจจุบันมีทางเลือกมากมายไม่ว่าจะใช้บริการจากเว็ปต่าง ๆ หรือตั้งเซิร์ฟเวอร์ขึ้นมาเอง ในกรณีเริ่มต้นศึกษาใช้งานแนะนำให้ใช้ Server ที่ให้บริการ Git โดยเว็บไซต์ที่นิยมจะมีดังนี้

* GitHub
* Gitlab
* Bitbucket

ซึ่งเว็บไซต์เหล่านี้จะมีให้บริการทั้งแบบ Public (ใคร ๆ ก็เข้ามาดูโค้ดของเราได้) และ Private (เฉพาะคนที่เราอนุญาตเท่านั้น) หากอยากทำเป็น Open Source อยู่แล้วแนะนำให้ใช้ GitHub แต่หากต้องการใช้งานระบบ CI/CD ด้วยแนะนำให้ใช้ GitLab เพื่อให้มองเห็นถึงภาพผู้ให้บริการทั้งหมดที่มีอยู่ในท้องตลาดปัจจุบันจึงของอธิบายถึงผู้ให้บริการอีกหนึ่งรายได้แก่ Bitbucket ซึ่งมีจุดขายว่าเป็นมากกว่าแค่ Git code management เนื่องจาก Bitbucket เป็นสถานที่เอาไว้ plan project, collaborate code สามารถ test และ deploy ได้ โดยทุก Project สามารถตั้ง Private Git repositories ได้โดยไม่ต้องเสียเงินสำหรับ 5 Users (GitHub ก็ทำได้แล้วแต่เป็นแบบฟรีสำหรับ 3 Users )

ตัว Bitbucket เป็นหนึ่งในผลิตภัณฑ์ของ Atlassian ซึ่งเป็นเจ้าของเดียวกันกับ Jira และ Trello ซึ่งนิยมนำมาใช้งานมากสำหรับระบบติดตามงานในปัจจุบัน ดังนั้นจึงง่ายกับการนำมาทำ Integration, built-in CI/CD มีบริษัทจำนวนมากที่นำ Bitbucket มาใช้ เช่น PayPal, Salesforce, CicleCI, Starbucks, Tesla Motors, Bitbucket, Pandora

ตารางที่ 4.3 เปรียบเทียบระหว่าง GitHub, GitLab และ Bitbucket แบบฟรี

| **Smallest Pack** | **GitHub** | **GitLab** | **Bitbucket** |
| --- | --- | --- | --- |
| Public repositories | Unlimited | Unlimited | Unlimited |
| Private repositories | Unlimited | Unlimited | Unlimited |
| Collaborators public repositories | Unlimited | Unlimited | 5 Users |
| Collaborators private repositories | 3 Users | Unlimited | 5 Users |
| Host | Cloud only | Cloud or Pay/Users | Cloud or on-premise |
| File Storage | 1GB/project | 10GB/project | 1GB/project |
| Branch Compare | Yes | Yes | Yes |
| Syntax Highligh | Yes | Yes | Yes |
| Pull/Merge Request | Yes | Yes | Yes |
| Webhook | Yes | Yes | Yes |
| Boards | Project boards | Issue Boards | Jira / Tello |
| CI/CD | N/A | Built-in CI/CD | Bitbucket Pipelines |
| Pipeline | N/A | Yes | Yes |

### ความเสถียร

GitHub มีล่มให้เห็นอยู่บ้าง แต่ก็จะเป็นเวลาสั้น ๆ อาจมีบ้างที่ล่มเป็นวันแต่ไม่บ่อยมากนัก แล้วก็มี Degraded Performance ให้เห็นเรื่อย ๆ อย่างไรก็ตามโดยรวมถือว่าเสถียรมาก

GitLab ไม่ค่อยล่ม ถึงจะเคยมีเคสใหญ่ที่ Database พังจนข้อมูลพวกคอมเมนต์ Pull Request ฯลฯ หายไป 6 ชั่วโมง (แต่ Git ไม่พัง) ก็ไม่ใช่ปัญหาอะไร แต่ปัญหาหลัก ๆ คือมีบั๊กโผล่มาให้ปวดหัวอยู่เรื่อย ๆ

Bitbucket ล่มเรื่อย ๆ โดยเฉลี่ยสองสามเดือนครั้ง ล่มที 3–4 ชั่วโมง

### การ Host Server เอง

บริษัทใหญ่ ๆ บางเจ้าก็ค่อนข้างหวง Source Code และไม่อยากเอาขึ้นไว้บน Cloud แต่อยากจะ Self-Hosted เอง อันนี้ก็ทำได้ โดยรายละเอียดตามนี้

GitHub: ทำไม่ได้

GitLab: เป็น Open Source สามารถติดตั้งบน Server ตัวเองได้ฟรีไม่มีค่าใช้จ่าย

Bitbucket: จ่ายราคาแบบครั้งเดียวตามจำนวน User เริ่มต้น $10 ได้ 10 Users ไปจนถึง $60,700 ได้ 2,000 Users

### สรุป ความเหมาะสม

GitHub หากใช้เป็น Open Source ก็ GitHub ได้เลยเนื่องจาก Community ของ GitHub มีความแข็งแกร่งมากและมีนักพัฒนาเป็นจำนวนมาก

GitLab ถ้าสร้าง Repo แบบ Private แนะนำให้ใช้ GitLab ไม่ก็ Bitbucket ถ้าทำงานเป็นทีมในแบบ Private แนะนำ GitLab เพราะค่าใช้จ่ายถูกกว่า Bitbucket

Bitbucket ถ้าใช้บริการของ Atlassian อื่น ๆ อยู่ เช่น JIRA ก็ใช้ Bitbucket เลยจะทำงานร่วมกันสมบูรณ์กว่า

## [6] การใช้งาน GitLab

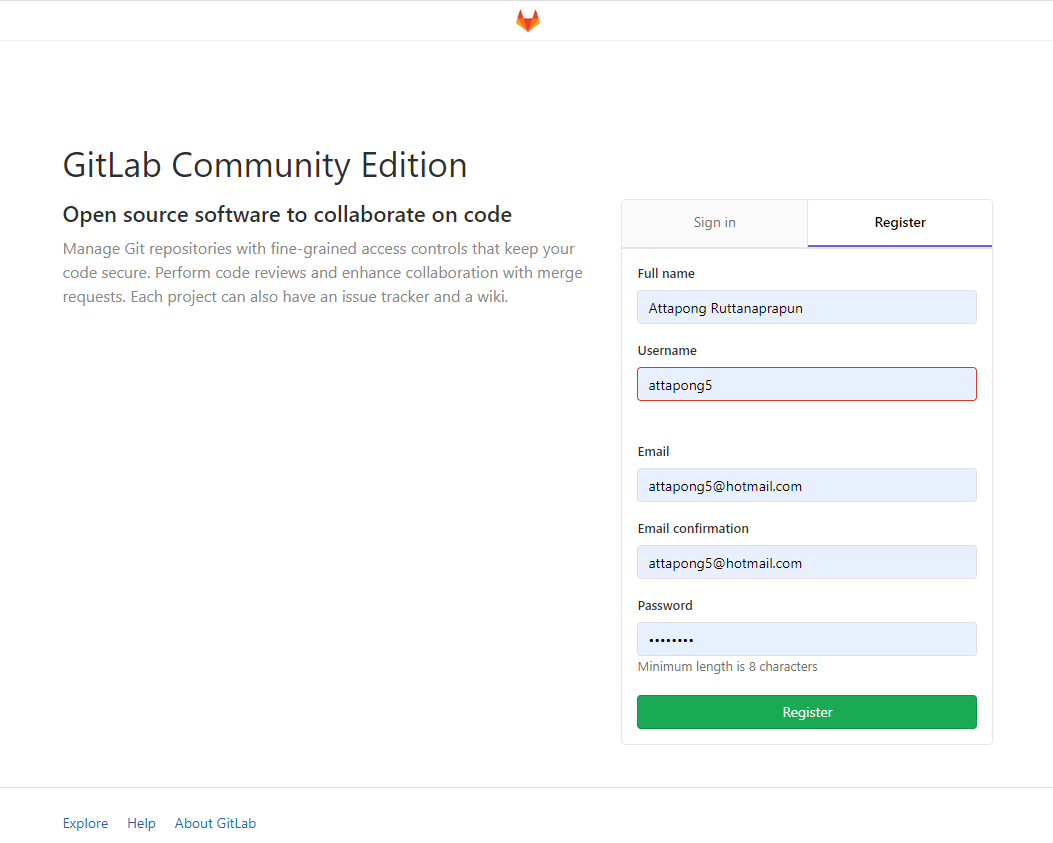
GitLab เป็น Git repository management และเป็น Open-Source เป็นเครื่องมือหนึ่งที่ได้รับความนิยม สามารถทำ Code reviews, Issue tracking, activity feeds และ wikis ได้ ทั้งนี้แค่ Single GitLab server ก็สามารถรองรับ users ได้มากกว่า 25,000 คน แล้วก็ยังสามารถรองรับได้สูงกว่านี้อีกถ้าหากใช้เครื่องที่มีคุณภาพสูงขึ้น หรือเพิ่มเครื่องขึ้นมา โดยเน้นไปทางด้าน DevOps lifecycle มากขึ้น เพื่อที่ต้องการให้ผู้ที่ใช้ GitLab สามารถทำงานร่วมกันได้ดีกว่า และส่งมอบงานให้ลูกค้าได้รวดเร็วกว่า

จุดเด่นที่เด่นมาก ๆ ของ GitLab ได้แก่ Built-in CI/CD และ มี issue board มาตั้งแต่ Free repository มีบริษัทประมาณ 100,000 บริษัท ที่นำ GitLab ทำ Organization เช่น Nasa, ING, SONY, EA, Citrix, SIEMENS

GitLab มี License ด้วยกันสองแบบ คือ Gitlab CE และ Gitlab EE

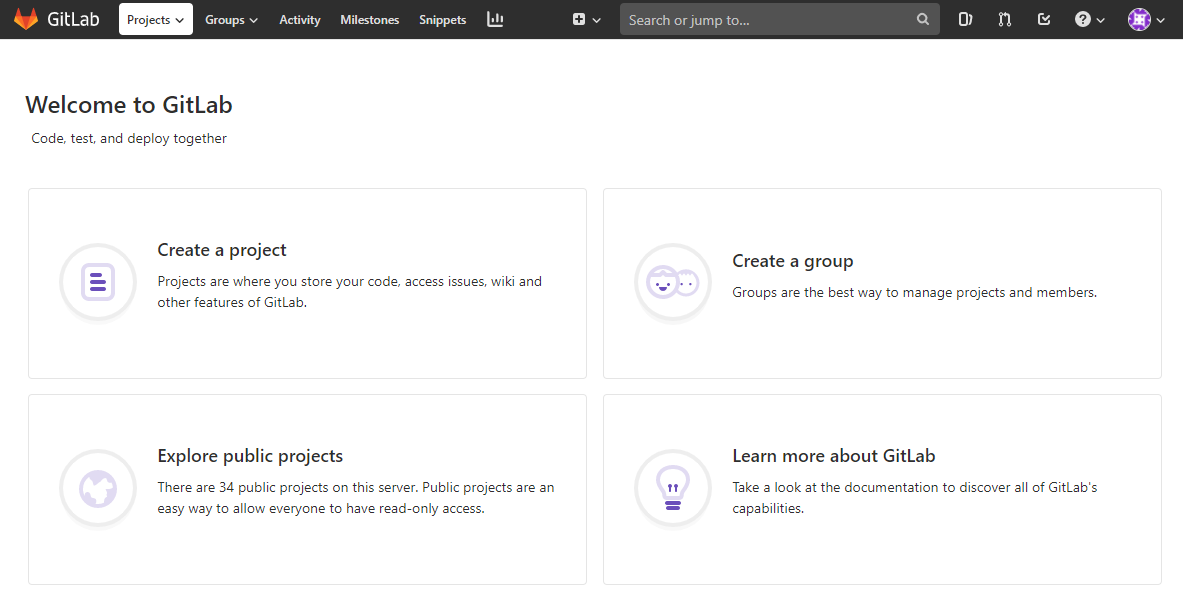
* Gitlab CE หรือ Community Edition เป็น open-source พร้อม MIT License ฟรี
* Gitlab EE หรือ Enterprise Edition สร้างขึ้นจาก Community Edition ใช้แกนเดียวกัน แต่เพิ่มฟีเจอร์และฟังก์ชันเพิ่มเติมนอกเหนือจากนั้น และไม่ติด License และมีค่าใช้จ่าย

เราสามารถฝึกการใช้ Git ได้จากเว็บไซต์ของ GitLab หากต้องการใช้งานภายในองค์กรซึ่งมีความสามารถในระดับการค้าให้ใช้ https://gitlab.com หรือในกรณีที่ต้องการใช้งานสำหรับศึกษาการเรียนรู้การใช้งานและไม่มีการคิดค่าใช้จ่ายเรียกว่าเวอร์ชัน Community ให้เข้าไปที่เว็บไซต์ https://gitlab.cpsudevops.com ซึ่งในปฏิบัติการจะใช้เวอร์ชันนี้ ในขั้นแรกให้เรา Register โดยใส่ Full name, Username, Email และ Password แล้วกดลงทะเบียนดังภาพ



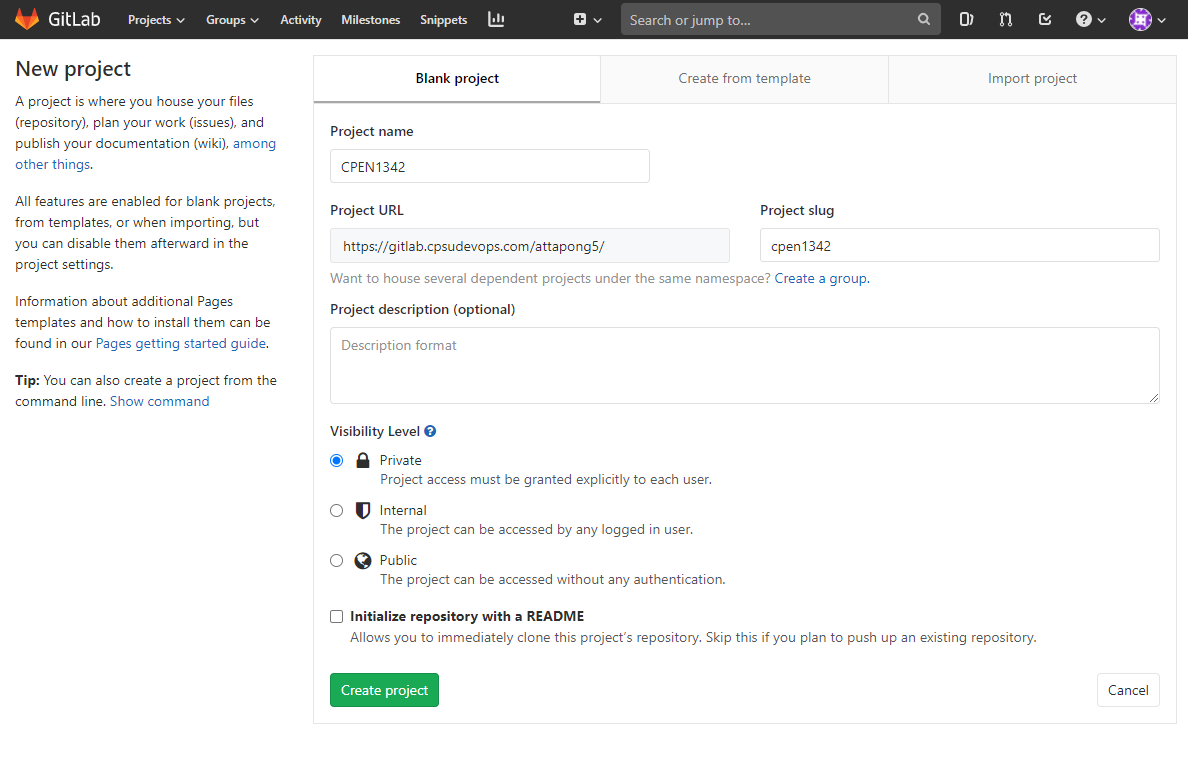
รูปที่ . การสมัครใช้งาน GitLab

เมื่อ Login เราจะเห็นหน้าตาของ GitLab ดังภาพด้านล่าง โดย GitLab เรียก Repository สำหรับเก็บ Version ของ Source Code ว่า Project ซึ่งเราจะทดลองสร้าง Project ชื่อ CPEN1342



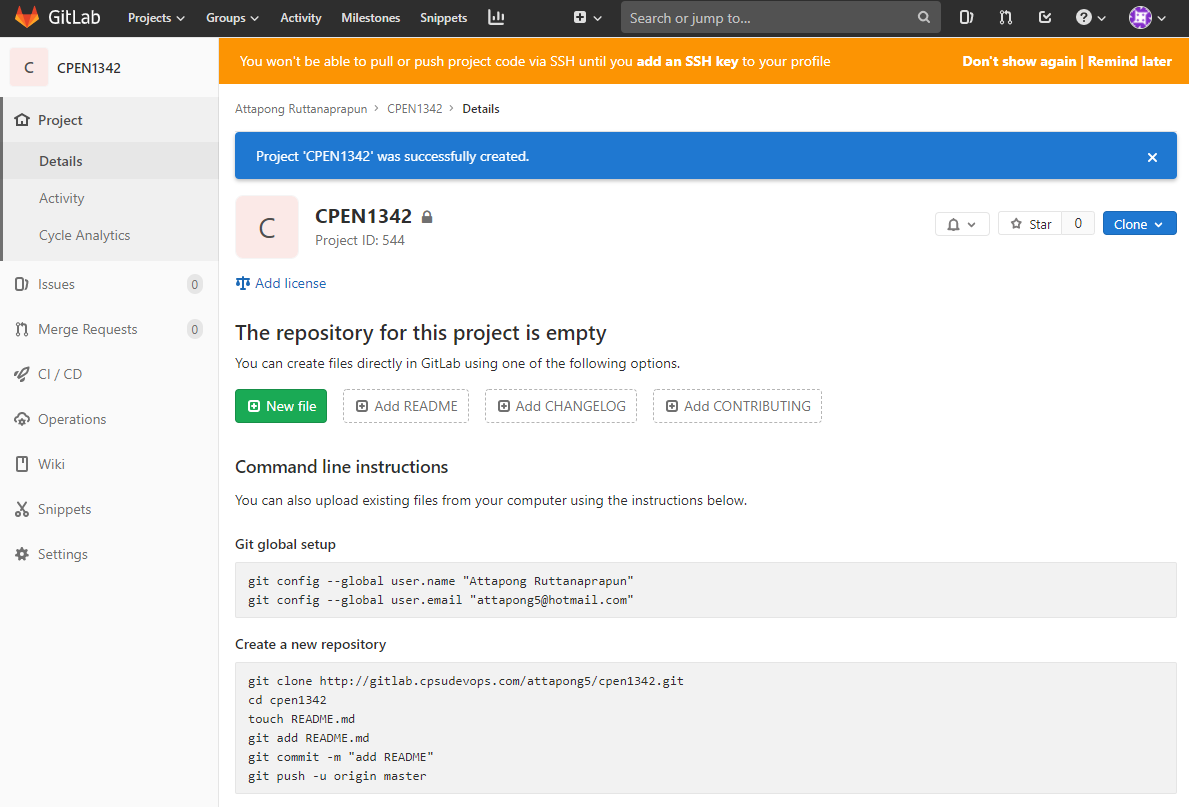
รูปที่ . หน้าต้อนรับการใช้งาน GitLab เมื่อเริ่มใช้งานครั้งแรก

เมื่อเข้าใช้งาน GitLab ครั้งแรกจะไม่พบโปรเจกต์ใด ๆ ใน GitLab จะมีหน้าต่างแสดงรายละเอียดดังรูป จากนั้นให้ทำการกดปุ่ม Create a project เพื่อเข้าสู่หน้าต่างสร้างโปรเจกต์ใน GitLab



รูปที่ . การสร้างโปรเจกต์ใหม่ใน GitLab

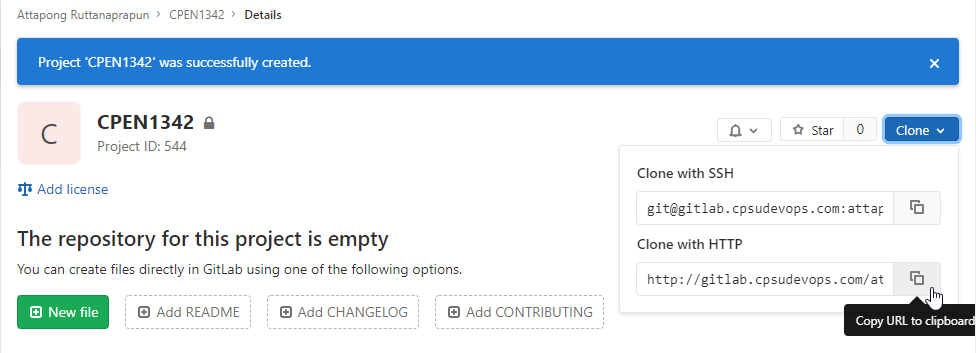
ให้ทำการตั้งชื่อโปรเจกต์เป็น CPEN1342 จากนั้นให้ทำการกดปุ่ม Create Project สำหรับ GitLab จะมีค่า Default ของ Repository เป็น Private ซึ่งเหมาะสำหรับการพัฒนา Project ที่ยังไม่ต้องการเผยแพร่ Source Code เป็นสาธารณะ ขณะที่ GitHub จะมีค่า Default ของ Repository เป็น Public



รูปที่ . หน้าโปรเจกต์ที่ถูกสร้างในครั้งแรก

Project ที่ถูกสร้างขึ้นจะยังไม่มีการเก็บ Version ใด ๆ ซึ่งเราสามารเพิ่มไฟล์ README.md เพื่ออธิบาย Project ได้จากตรงนี้ หรือจะเพิ่มบน Local Host แล้วค่อย Sync กับ Remote Repository ก็ได้ (สำหรับการเริ่มต้นใช้งาน การเพิ่ม README.md บน Local Host จะสะดวกกว่า) ส่วนด้านล่างจะเป็นการแสดงรายละเอียดคำสั่งทางคอมมานด์ไลน์ของ Git ให้ใช้งานผ่านทางโปรแกรม Git Bash

เพื่อจะใช้งาน Distributed Version Control System อย่าง Git บน Local Host เราจะต้องเชื่อมโยงกับ Remote Repository โดยการกดที่ปุ่ม Clone จากนั้นให้ใช้ URL ตามภาพด้านล่าง



รูปที่ . คัดลอก URL สำหรับ Git ของโปรเจกต์ที่สร้างขึ้น

### ทดสอบการใช้งาน Git เบื้องต้น

สำหรับระบบปฏิบัติการ Windows หากต้องการใช้งาน Git แบบ Unix Command Line Based แนะนำให้ทำการติดตั้ง Windows Subsystem for Linux (WSL) และ Ubuntu Distribution จาก Microsoft Store แทน สำหรับนักศึกษาที่ได้ทำปฏิบัติการ Docker ในปฏิบัติการที่ผ่านมาสามารถข้ามขั้นตอนนี้ไปได้เนื่องจากได้ทำการติดตั้งไว้เรียบร้อยแล้ว หากเป็นการติดตั้งใหม่ให้ทำตามขั้นตอนต่อไปนี้

เปิด PowerShell แบบ Administrator แล้วติดตั้ง WSL ด้วยคำสั่งต่อไปนี้

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

Update WSL เป็น Version 2

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

wsl --set-default-version 2

ติดตั้ง Ubuntu Distribution (Ubuntu 18.04 LTS) จาก Microsoft Store ผ่านทางลิงค์

https://www.microsoft.com/store/apps/9N9TNGVNDL3Q

ก่อนจะเริ่มใช้งาน Git ได้นั้น จำเป็นต้องตั้งค่าการใช้งานเบื้องต้นก่อน ในที่นี้จะใช้โปรแกรม Git Bash ที่ติดตั้งมาพร้อม Git for Windows



รูปที่ . Git Bash

หลังจากติดตั้งแล้ว เราสามารถดู Version ของ Git ได้จาก Command Line ของ Ubuntu Distribution โดยใช้คำสั่งดังนี้

git version

ในการใช้งานครั้งแรก เราจะต้อง Config Git ให้จำ Username และ Email ที่ได้ใช้ใน GitLab ด้วยคำสั่ง git config โดยมีรายละเอียดดังนี้

ตั้งชื่อผู้ใช้งาน “user.name”

git config --global user.name "ชื่อ นามสกุล"

ตั้งอีเมลของผู้ใช้งาน “user.email”

git config --global user.email "อีเมล"

ตรวจสอบว่าที่ตั้งค่าถูกต้องหรือไม่

git config --global --list

หรือ

cat ~/.gitconfig

ในกรณีที่ต้องการเคลียร์หน้าจอ Terminal ของ Git Bash ให้ใช้คำสั่ง clear

clear

### สร้างโฟลเดอร์บน Localhost

ในที่นี้เราจะทำการย้ายการทำงานไปไว้ที่ไดรฟ์ D: ด้วยการพิมพ์คำสั่ง cd D: ดังนี้

cd D:

จากนั้นให้ทำการสร้างโฟลเดอร์ CPEN1342\_GIT ด้วยคำสั่ง mkdir ซึ่งโฟลเดอร์ที่สร้างนี้จะเป็น working directory ของ GIT นั่นเอง

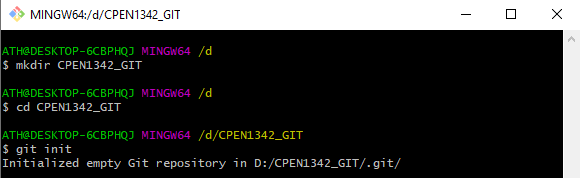
mkdir CPEN1342\_GIT

ย้ายการทำงานของ Git Bash ไปยังโฟลเดอร์ CPEN1342\_GIT ด้วยคำสั่ง cd ดังนี้

cd CPEN1342\_GIT

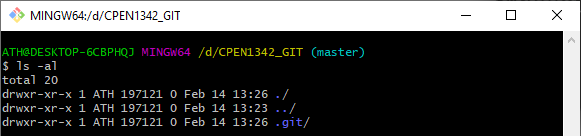
ใช้คำสั่ง git init เพื่อเริ่มต้นสร้าง Git repository

git init



รูปที่ . การสร้างโฟลเดอร์และเริ่มให้ Git ทำงานบน Local Host

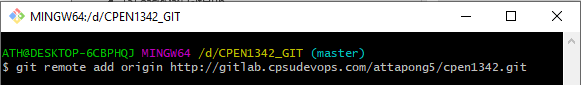
จากภาพด้านบน Git จะสร้าง Folder .git ภายใน Folder Project (CPEN1342\_GIT) สำหรับเก็บ Local Repository ให้เอง ซึ่งเราจะมองไม่เห็น Folder ที่มีการตั้งชื่อด้วย "." นำหน้าอยู่แล้ว สำหรับบน Unix Command Line Based จะต้องใช้คำสั่ง ls -a หรือ ls -al เพื่อดู Folder .git ดังภาพด้านล่าง



รูปที่ . รายละเอียดของโฟลเดอร์ของโปรเจกต์ใน Local Host

หลังจากนั้นเราจะเชื่อมโยงกับ Remote Repository ซึ่งในที่นี้จะหมายถึงตัว GitLab Server นั่นเองโดยนำ URL มาจาก GitLab Server ที่ได้ก่อนหน้านี้ โดยใช้คำสั่ง git remote add origin ดังรายละเอียดด้านล่าง

git remote add origin URL ที่คัดลอกมาจาก GitLab Server



รูปที่ . การเชื่อมโยงระหว่าง Local Host Project กับ Remote Repository

เป็นอันเสร็จสิ้นสำหรับการ Config Git เพื่อการใช้งาน

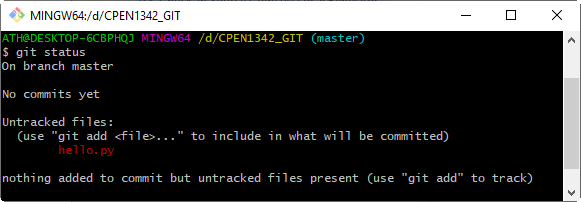
### การ Check-In Source Code กับ Local Repository

ให้ทำการสร้างไฟล์ hello.py และบันทึกไว้ในโฟลเดอร์ D:\CPEN1342\_GIT โดยใช้ Editor ที่ถนัดโดยเขียนโค้ดภายในดังนี้

print('Hello git')

ไฟล์ที่สร้างใหม่จะยังไม่ถูก Track ในระบบ Git เราสามารถดูสถานะของไฟล์ใน Project โดยใช้คำสั่ง git status ดังภาพด้านล่าง ไฟล์ที่มีสถานะ Modified หรือยังไม่ถูก Track จะแสดงด้วยสีแดง

git status



รูปที่ . สถาณะของ hello.py ที่ยังไม่ถูก Track เข้าไปในระบบ Git

สถานะของไฟล์ที่อยู่ภายใน GIT แบ่งออกได้ดังนี้

Staged คือสถานะที่ไฟล์พร้อมจะ commit

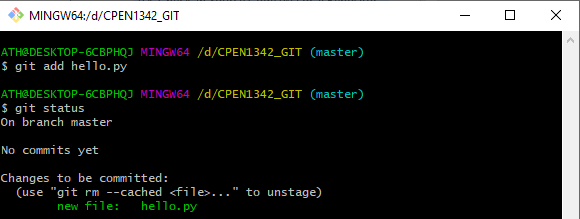
Unstaged คือไฟล์ที่ถูกแก้ไขแต่ว่ายังไม่ได้เตรียม commit

untracked คือไฟล์ที่ยังไม่ถูก track โดย Git (ส่วนมากจะเป็นไฟล์ที่เพิ่งสร้างใหม่)

deleted ไฟล์ที่ถูกลบไปแล้ว

วิธีการเริ่มต้น tracking ไฟล์ hello.py นั้น คือต้องเพิ่มไฟล์นี้ลงไปยัง Staging Area ก่อนเราจะใช้คำสั่ง git add ซึ่งไฟล์ที่มีสถานะ Staged จะแสดงด้วยสีเขียว

git add hello.py



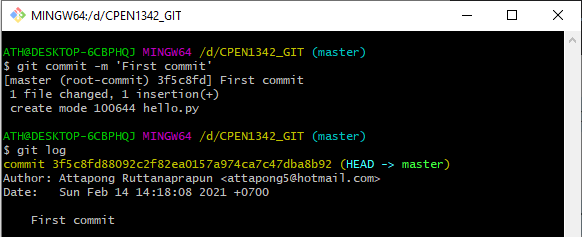
รูปที่ . การ Track file ด้วยคำสั่ง git add

เมื่อทำการ track ไฟล์เรียบร้อยแล้วเราจะใช้คำสั่ง git commit เพื่อจัดเก็บไฟล์อย่างถาวร ด้วยการสร้างเป็น Version ของ Source Code และใช้ parameter -m สำหรับใส่ Commit Message Log อธิบายการ Commit แต่ละ Version ภายในสัญลักษณ์ ‘ ’ โดย Commit Message Log นั้นข้อความต้องเป็น present tense และไม่ควรเกิน 50 ตัวอักษร ดังตัวอย่างการใช้งานคำสั่งด้านล่าง

git commit -m 'First commit'

Git จะเรียก Version ต่าง ๆ ของ Project ว่า History เพื่อดู History ของ Project เราจะใช้คำสั่ง git log จากรูปที่ 4.56 จะเห็นได้ว่าเราได้ทำการ Commit ไปแล้ว 1 Version ซึ่ง Git จะใช้ค่าจาก SHA-1 Hash เป็น Commit ID

git log



รูปที่ . การ Commit และเรียกดู History ของโปรเจกต์

Output ที่ได้ มีดังนี้

* เลขการ ‘commit’ ออกมา 40 ตัวอักษร เรียกว่า SHA ตัวอักษรสีเหลือง แต่เวลาอ้างอิงใช้แค่ 7 ตัวอักษรแรก
* ชื่อคน ‘commit’
* วันที่ และเวลาที่ ‘commit’
* commit message

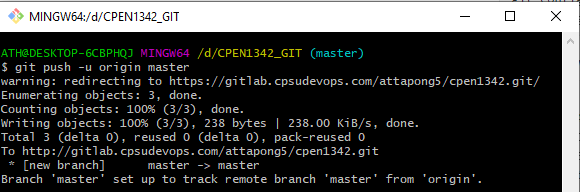
จากภาพ History ด้านบน ทุกคนในทีมจะสามารถเห็นว่าใครเป็นคน Commit และ Commit ที่ Branch โดย Branch Default ที่เรา Check-In คือ Master Branch

เราสามารถบอก Git ได้ว่าไม่ต้องการให้ Git เก็บไฟล์ไหนบ้าง โดยสร้างไฟล์ชื่อ .gitignore ไว้ที่ไดเรกทอรี Git project ภายในก็กำหนดค่าเป็นชื่อไฟล์ หรือชื่อไดเรกทอรี หรือจะกำหนดเป็น \*.log คือไม่ต้องเก็บไฟล์ .log ทั้งหมด เป็นต้น

### การ Sync ไปยัง Remote Repository

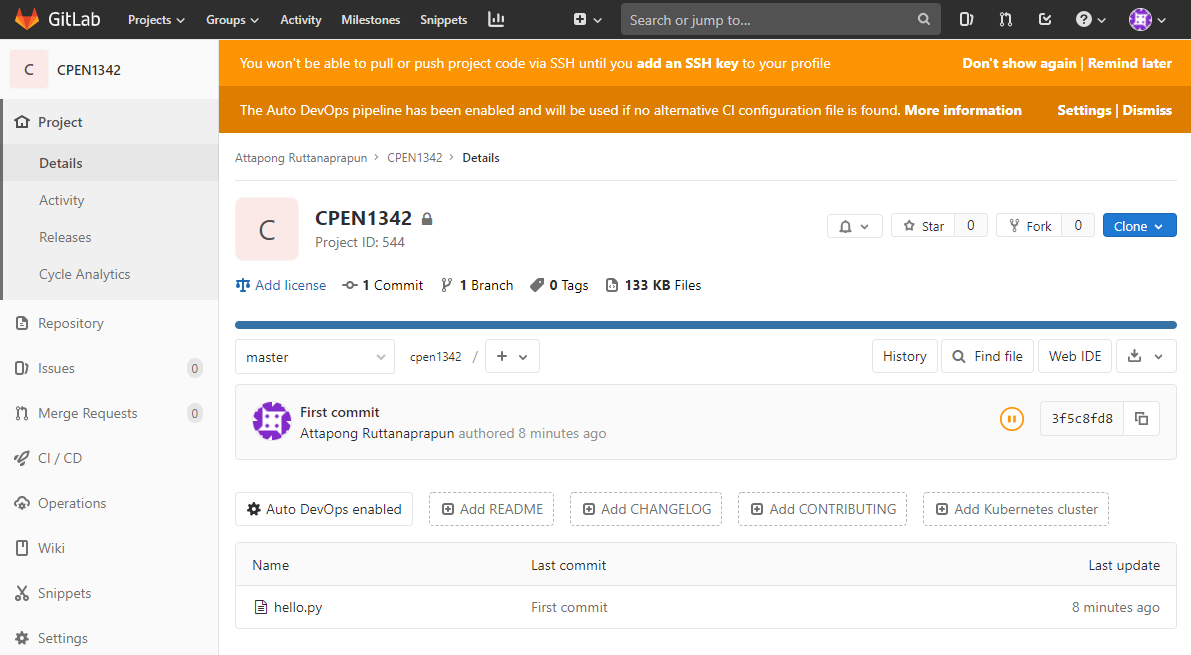
เพื่อ Sync History กับ Remote Repository เราจะใช้คำสั่ง git push -u origin master และยืนยันตัวตนกับ GitLab Server ด้วย Username และ Password ที่ได้สมัครไว้ในขั้นตอนก่อนหน้า

git push -u origin master



รูปที่ . การ Sync History กับ Remote Repository

เมื่อทำการรีเฟรชหน้าเว็บไซต์ของ GitLab ใหม่จะพบว่าในส่วน Details ของ Project จะเปลี่ยนรายละเอียดจากเดิมไปเป็นดังรูปที่ 4.58

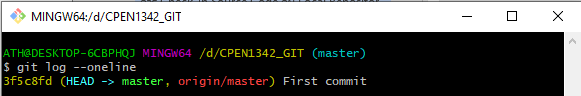


รูปที่ . ผลที่ได้จากการ push ขึ้น GitLab Server

ในหน้าเว็บไซต์ของ GitLab จะพบว่ามีการ Sync ข้อมูลกับ Remote Repository เรียบร้อยแล้ว ในครั้งต่อไปเราสามารถ Push โดยใช้เพียงคำสั่ง git push สั้น ๆ เพราะเราได้ใช้ parameter -u เพื่อจำว่าจะ Push ไปที่ Remote ตัวไหน และ Branch ไหน

ทั้งนี้ให้ทำการดู History ของ Git อีกครั้งซึ่งในครั้งนี้จะเป็นการดู History อย่างย่อ โดยใช้คำสั่ง git log --oneline ซึ่งเราจะเห็นว่าขณะนี้เรา Check-Out อยู่ที่ Master Branch และ Commit ID 3f5c8fd จาก HEAD Pointer

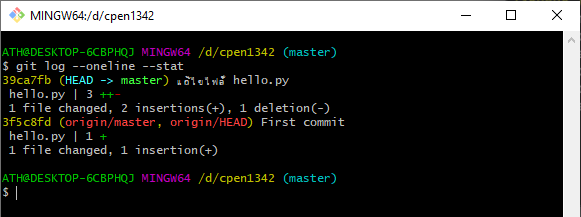
git log --oneline



รูปที่ . การดู History ของ Git อย่างย่อ

ดังที่ได้กล่าวไว้แล้วว่าเราสามารถดู Commit ทั้งหมดได้โดยใช้คำสั่ง git log --oneline แต่ถ้าต้องการเห็นว่ามีความเปลี่ยนแปลงที่ไฟล์ใดและมีการเปลี่ยนแปลงอย่างไรบ้าง ก็ให้เพิ่ม parameter --stat เพิ่มเข้าไป

git log –oneline –stat



รูปที่ . การดูรายละเอียดการเปลี่ยนแปลงของแต่ละไฟล์

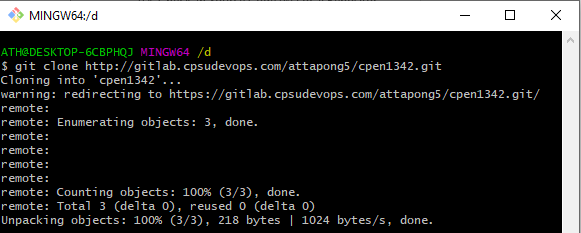
### การ Clone Project

ก่อนจะ Clone หากชื่อโปรเจกต์ตรงกับชื่อโฟลเดอร์ที่อยู่ใน Local Host ให้ลบ Folder เดิม ออก ในกรณีที่ชื่อโปรเจกต์ไม่ตรงกับชื่อโฟลเดอร์ใน Local Host ให้ข้ามการทำงานส่วนนี้ไป จากนั้นให้ย้ายการทำงานของ Git Bash ที่ไดรฟ์ D: เสียก่อนด้วยคำสั่ง cd D: เพื่อไม่ให้โคลนโปรเจกต์ไปไว้ในโฟลเดอร์โปรเจกต์เดิม (D:\CPEN1342\_GIT) ที่เราได้ทำงานไว้ก่อนหน้านี้

cd D:

จากนั้นให้ใช้คำสั่ง git clone โดยใช้ URL ที่ได้มาจากเว็บไซต์ของ GitLab ดังที่เคยแสดงไว้ใน รูปที่ 4.49

git clone URL ที่คัดลอกมาจาก GitLab



รูปที่ . การโคลนโปรเจกต์จากเซอร์เวอร์ของ GitLab

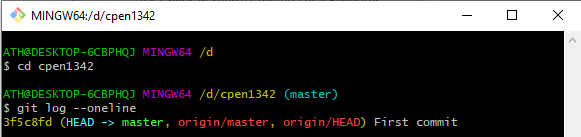
โฟลเดอร์ที่โคลนมาจะถูกสร้างไว้ในโฟลเดอร์ D:\cpen1342 ซึ่งเป็นชื่อเดียวกันกับชื่อโปรเจกต์ที่ตั้งไว้ใน GitLab นั่นเอง จากรูปที่ 4.61 Git จะคัดลอก Source Code ทุก Version จาก Remote Project และสร้าง Folder ใหม่ (cpen1342) บน Local Host

เราสามารถใช้คำสั่ง git log --oneline ดู History ของ Branch ปัจจุบัน (Master Branch) ที่กำลัง Check-Out ได้แต่ต้องย้ายการทำงานเข้าไปยังโฟลเดอร์ cpen1342 เสียก่อน

cd cpen1342

จากนั้นใช้คำสั่ง git log –oneline เพื่อดู History ของ Branch

git log –oneline



รูปที่ . การดู History ของโปรเจกต์ที่โคลนมา

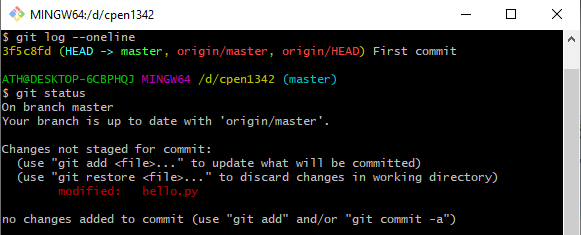
ขั้นตอนต่อมาให้เปิดไฟล์ hello.py ในโฟลเดอร์ cpen1342 โดยใช้ Editor ตัวใดก็ได้ แล้วเพิ่ม Comment ที่บรรทัดแรก และคำสั่ง print ในบรรทัดที่ 2 แล้วกดบันทึก โดยในโค้ดมีรายละเอียดต่อไปนี้

print('Hello git') #First Comment

print('Add new line')

ขณะนี้เราจะมีไฟล์ hello.py อยู่ 2 Version โดย Version หนึ่งถูก Check-In ด้วย Commit ID 3f5c8fd อีก Version เป็น Code ที่มีการแก้ไขจาก Commit เดิม ซึ่งทำให้ไฟล์ hello.py มีในสถานะเป็น Modified ซึ่งตรวจสอบได้ด้วยคำสั่ง git status

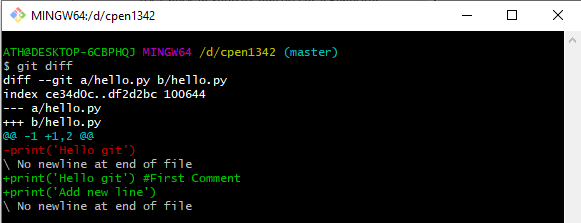
git status



รูปที่ . สถานะของ hello.py ที่ถูกเปลี่ยนแปลง

เราจะใช้คำสั่ง git diff เปรียบเทียบ Source Code ของไฟล์ 2 เวอร์ชันบรรทัดต่อบรรทัด โดย Git จะแสดงเครื่องหมาย "-" และตัวอักษรสีแดงในบรรทัดเดิมก่อนถูกแก้ไขหรือก่อนถูกลบ และแสดงเครื่องหมาย "+" และโค้ดใหม่ที่ถูกแก้ไขหรือเพิ่มใหม่ด้วยตัวอักษรสีเขียว ดังรูปที่ 4.64

git diff



รูปที่ . ความแตกต่างระหว่างสองเวอร์ชัน

การออกจาก diff mode ให้กด q จากนั้นให้ทำการ Check-In เพื่อจัดเก็บ Source Code อย่างถาวรโดยใช้คำสั่ง git add hello.py และ git commit -m 'Edit hello.py' ตามลำดับ แล้วดู History ของ Source Code ด้วยคำสั่ง git log –oneline ตามขั้นตอนต่อไปนี้

ทำการ Check-In โค้ดเข้าไปใน Git

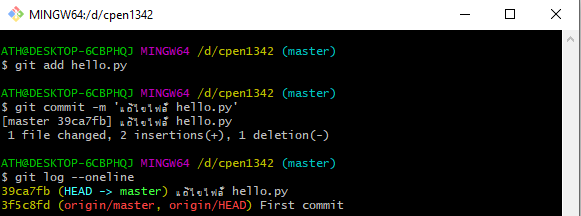
git add hello.py

ทำการ Commit เพื่อจัดเก็บไฟล์อย่างถาวร

git commit -m 'แก้ไขไฟล์ hello.py'

เรียกตรวจดู History ของ Git อีกครั้ง

git log –oneline



รูปที่ . ตรวจดูสถาณะการ Commit ของไฟล์ hello.py

ซึ่งจะเห็น History ของเรามี Commit เพิ่มอีก 1 Commit คือ Commit ID 39ca7fb รวมทั้ง Local Head Pointer ขยับมาชี้ที่ Commit ใหม่ แต่ Remote Head Pointer ยังคงอยู่ที่ Commit ID เดิมเหมือนตอนที่ Clone Project มาใหม่ Commit แต่ละอันนี้เราจะเรียกว่า Branch หรือสายงาน

### การ Check-Out

เมื่อเราดึงโค้ดจากเซิร์ฟเวอร์มาแก้ไขแล้วพัง และอยากจะ Reset กลับไปยัง Version เดิมใน Commit ล่าสุด เราจะใช้คำสั่ง git checkout เพื่อนำไฟล์จากใน Commit ล่าสุดกลับมาแทนที่ไฟล์ที่พังใน Working Directory ของ Local Host เช่น

git checkout hello.py

### คำสั่งเพิ่มเติมในการใช้งาน Git

เราสามารถใช้คำสั่ง git add . เพื่อทำการส่งไฟล์ทุกไฟล์ไปยัง Staging Area เพื่อเตรียมการ Commit ได้

git add .

ในกรณีที่เราเพิ่มไฟล์บางไฟล์ลงใน Staging Area โดยไม่ตั้งใจก็สามารถเอามันออกมาโดยใช้คำสั่ง git reset เช่น git reset hello.py

git reset hello.py

บางครั้งเราอาจเขียน Commit Message ผิด เราสามารแก้ไขมันได้โดยการเพิ่ม parameter --amend ตามหลังคำสั่ง git commit ซึ่งจะทำให้ Hash ID ใน Commit เดิมเปลี่ยนไปด้วย

ในกรณีที่ต้องการรีเซต Git ทั้งชุดสามารถทำได้เช่นกันโดยให้ทำการระบุ Commit ID ที่ต้องการรีเซต ซึ่งการทำงานเช่นนี้ค่อนข้างอันตรายพอสมควร ทั้งนี้คำสั่ง git reset มี parameter อยู่ 3 แบบ คือ --soft, --mixed และ --hard ซึ่งแต่ละแบบมีความรุนแรงในการแก้ปัญหาต่างกันโดยมีรายละเอียดดังนี้

git reset --soft fdd69ae

จะลบ Commit ทุก Commit หลัง Commit ID fdd69ae แล้วนำไฟล์ที่เคยอยู่ใน Commit เหล่านั้นกลับมายัง Staging Area

git reset --mixed fdd69ae หรือ git reset fdd69ae

จะลบ Commit ทุก Commit หลัง Commit ID fdd69ae แล้วนำไฟล์ที่เคยอยู่ใน Commit เหล่านั้นกลับมายัง Working Directory

git reset --hard fdd69ae

จะลบ Commit ทุก Commit หลัง Commit ID fdd69ae และจะทำลายไฟล์ที่เคยอยู่ใน Commit เหล่านั้นซะ

git commit --amend -m 'edit add function'

ในกรณีที่เราสร้างไฟล์ใหม่แต่ต้องการเพิ่มไฟล์นั้นใน Commit เดิมก็สามารถใช้พารามิเตอร์ --amend ได้เช่นกัน

git add .

git commit --amend --no-edit

## [7] การใช้งาน Git Branch ใน GitLab

Git สนับสนุนให้ Software Developers สร้าง Branch/สลับ Branch ไปมา และ Merge Branch บ่อย ๆ เท่าที่จะทำได้ในแต่ละวัน บางคนบอกว่า Branch ของ Git คือ Killer Feature ดังนั้นการใช้งาน Branch ได้คล่องแคล่วจะทำให้ Software Developers มีเครื่องมือที่ทรงพลัง ซึ่งจะทำให้มีการเปลี่ยนแปลงวิธีการพัฒนา Software ไปจากเดิมอย่างสิ้นเชิง

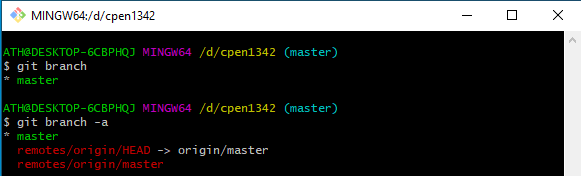
เพื่อให้เห็นภาพการทำงานกับ Branch จะขอเปรียบเทียบ Branch เป็นดังจุดบนสุดของ Timeline ในจักรวาลคู่ขนานที่มี Timeline มากมาย เมื่อเราใช้คำสั่ง git log เราจะเห็น History หรือลำดับของ Commit หรือจุดต่าง ๆ บน Timeline ซึ่งเราสามารถเดินทางไปยังตามจุดเหล่านั้นได้ในพริบตาเพียงแค่ใช้คำสั่ง git checkout ซึ่ง ณ จุดที่เราไปถึงก็จะเป็นเวลาปัจจุบันที่เราจะพบเจอเหตุการณ์ ผู้คน และสิ่งต่าง ๆ ใน Version ตามที่เราเคย Commit ไว้ดังที่แสดงไว้ในรูปที่ 4.65 เราสามารถเดินทางสลับไปมายังจุดบนสุดของแต่ละ Timeline (สลับไปมายัง Branch ต่าง ๆ ที่สร้างไว้) แล้วทำการแก้ไขโค้ดที่ต้องการและบันทึกมันเก็บไว้เป็น Snapshots โดยการสร้าง Commit ใหม่เป็น Commit ล่าสุดบน Timeline

เราสามารถดู Branch บน Local Host โดยใช้คำสั่ง git branch

git branch

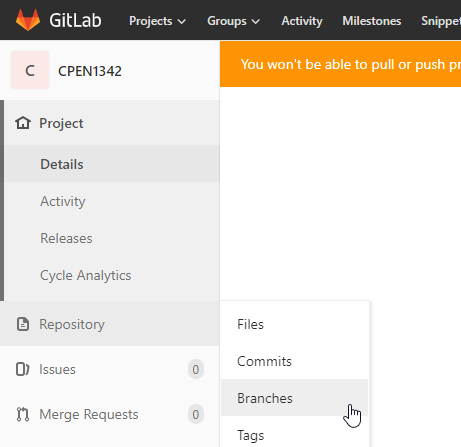
เพื่อแสดงชื่อ Branch ทั้งบน Local Project และ Remote Project เราจะพิมพ์คำสั่ง git branch -a

git branch -a



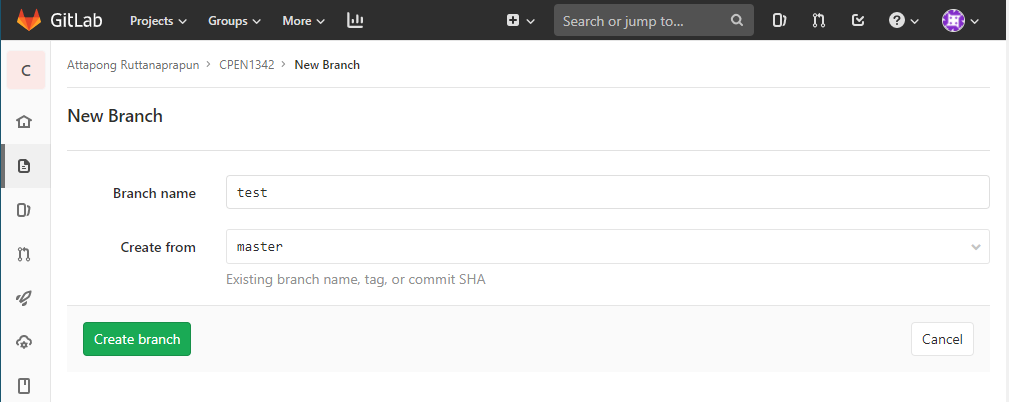
รูปที่ . การเรียกดู Branch ของ Local Project และ Remote Project

ในที่นี้เราจะทำการสร้าง branch ใหม่จากหน้าเว็บไซต์ของ GitLab โดยไปที่เมนู Repository > Branches



รูปที่ . เลือกเมนู Branches บนหน้าเว็บไซต์ของ GitLab

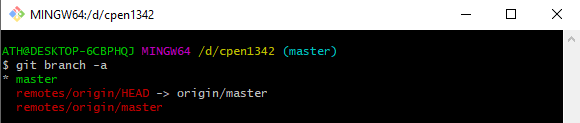
เมื่อเข้าสู่หน้าจัดการ Branches แล้วให้กดปุ่ม New branch โดยกำหนดชื่อ branch ใหม่ที่ต้องการสร้างเป็น test



รูปที่ . การสร้าง Branch ใหม่บนหน้าเว็บไซต์ของ GitLab

เมื่อกำหนดค่าต่าง ๆ เรียบร้อยแล้วให้กดปุ่ม Create branch จากนั้นให้ตรวจสอบ branch ใหม่ด้วยคำสั่ง git branch -a

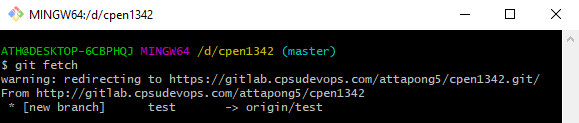
git branch -a



รูปที่ . ข้อมูลของ Branch ในฝั่ง Local และ Remote Project

จะเห็นได้ว่าข้อมูลฝั่ง Remote ไม่ปรากฏ branch ใหม่ที่สร้างขึ้นมาซึ่งแสดงว่ายังไม่มีการอัปเดตข้อมูล ทั้งนี้สามารถปรับปรุง Remote Branch Pointer ให้อัปเดตเป็นข้อมูลล่าสุดได้ตลอดเวลา เนื่องจากมีผู้ใช้งานคนอื่นได้ทำการ commit โค้ดขึ้นไปหรือทำการสร้าง branch ใหม่ ซึ่งสามารถทำได้โดยการใช้คำสั่ง git fetch

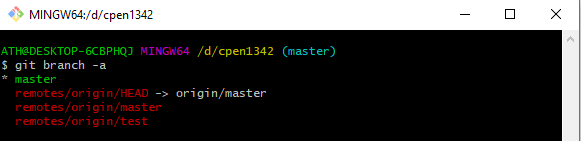
git fetch



รูปที่ . การใช้คำสั่ง fetch เพื่อดึงข้อมูลล่าสุดจาก Remote Project

จากนั้นให้ให้ตรวจสอบ branch ใหม่ด้วยคำสั่ง git branch -a อีกครั้ง

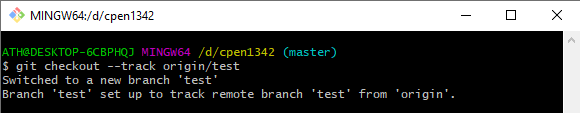
git branch -a



รูปที่ . ตรวจสอบ branch หลังจากใช้คำสั่ง git fetch

จะเห็นได้ว่าข้อมูลของ Branch บนฝั่ง Remote Project ได้ทำการอัปเดตข้อมูลแล้วอย่างไรก็ตาม Git จะไม่สร้าง Test Branch บน Local Host จนกว่าจะใช้คำสั่ง git checkout --track เพื่อเชื่อมโยงกับ Test Branch บน Remote Project ด้วยคำสั่งต่อไปนี้

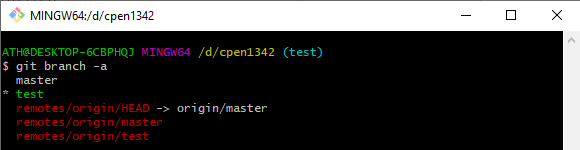
git checkout --track origin/test



รูปที่ . การดึง Test Branch มายัง Local Host

จากนั้นใช้คำสั่ง git branch -a อีกครั้ง

git branch -a



รูปที่ . ตรวจสอบ branch หลังจากใช้คำสั่ง git checkout –track

สมมติเรามีไฟล์อยู่ 3 ไฟล์ คือ README test.rb และ LICENSE ใน Git Folder เมื่อเรา Check-In ทั้ง 3 ไฟล์ลง Database ด้วยคำสั่ง git add และ git commit แล้ว Git จะสร้าง Object ที่ชื่อว่า Blob ทั้งหมด 3 Blobs ซึ่งเก็บ Content ของไฟล์ 3 ไฟล์ และสร้าง Object ที่ชื่อว่า Tree 1 Tree เก็บรายการของ Content ทั้งหมด รวมทั้งสร้าง Object ชื่อว่า Commit 1 Commit เก็บข้อมูลการ Commit และ Pointer ที่ชี้ไปยัง Tree ดังกล่าว

git add README test.rb LICENSE

git commit -m 'The initial commit of my project'



รูปที่ . Blob สำหรับเก็บรายละเอียดของไฟล์

เมื่อเราแก้ไขโค้ดแล้ว Check-In อีกครั้ง Commit ใหม่ที่เกิดขึ้นมาก็จะเก็บ Pointer ที่ชี้ไปยัง Commit ก่อนหน้า ดังภาพด้านล่าง



รูปที่ . Snapshot

Branch คือ Pointer ที่สามารถเคลื่อนที่ไปยัง Commit บนสุดของ Timeline โดย Git จะสร้าง Branch ชื่อ Master เป็น Branch Default เมื่อมีการ Commit ในครั้งแรก ทุกครั้งที่เราสร้าง Commit ขึ้นมาใหม่ Master Branch Pointer จะเคลื่อนไปข้างหน้า (Move Forward) โดยอัตโนมัติ ซึ่งจะทำให้มันชี้ไปยัง Commit บนสุดของ Timeline เสมอ นอกจากนี้มันยังมี HEAD Pointer ที่ชี้ไปยัง Master Branch ที่เรากำลัง Check-Out ด้วย



รูปที่ . Head Pointer

### การสร้าง Branch ใหม่

สิ่งที่เกิดขึ้นเมื่อเราสร้าง Branch ใหม่ชื่อ Testing ก็คือ Git จะสร้าง Testing Pointer ที่ชี้ไปยัง Commit เดิมที่ Master Pointer ชี้อยู่ โดยคำสั่งด้านล่างคือคำสั่งในการสร้าง Branch

git branch testing



รูปที่ . Pointer ใหม่ที่เกิดขึ้นจากคำสั่ง git branch

ซึ่งขณะนี้แต่ละ Branch Pointer ยังคงอยู่บน Timeline เดียวกัน และ Branch ปัจจุบันยังคงเป็น Master Branch เพราะ HEAD Pointer ยังคงอยู่ที่นี่



รูปที่ . Head ยังคงชี้ที่ master branch

### การสลับ Branch

การสลับ branch จะใช้คำสั่ง git checkout เมื่อเราใช้คำสั่ง git checkout จะทำให้ HEAD pointer เคลื่อนตัวไปยัง Testing Branch ดังภาพด้านล่าง

git checkout testing



รูปที่ . การสลับ Branch ทำให้เกิดการย้ายตำแหน่งของ HEAD Pointer

สมมติเราแก้ไขโค้ดในไฟล์ test.rb แล้ว Check-In สิ่งที่เกิดขึ้นก็คือ Testing Pointer และ HEAD Pointer จะเคลื่อนตัวไปข้างหน้ายัง Commit ใหม่ ขณะที่ Master Pointer ยังอยู่ที่เดิม

vi test.rb

git commit -a -m 'made a change'



รูปที่ . การเลื่อนของ HEADER เมื่อทำการ commit โค้ดใหม่

การสลับไปมาระหว่าง Branch มายัง Master Branch เป็นเรื่องที่ทำได้ง่าย ๆ โดยใช้คำสั่ง git checkout ซึ่งจะทำให้ HEAD Pointer เคลื่อนตัวกลับมายัง Master Pointer

git checkout master



รูปที่ . การสลับกลับมาทำงานยัง master branch

การสลับมายัง Master Branch นอกจากจะทำให้ HEAD Pointer เคลื่อนที่แล้ว ไฟล์และ Folder ต่าง ๆ ใน Working Directory จะถูกเปลี่ยนกลับมาเป็น Version ที่เรา Commit ไว้ด้วย

ทีนี้ถ้าเราแก้ไข Code ในไฟล์ test.rb แล้ว Check-In ที่ Master Branch ทั้ง HEAD Pointer และ Master Pointer จะเคลื่อนตัวไปข้างหน้ายัง Commit ล่าสุด ซึ่งจะทำให้เกิดการแยกตัวออกเป็น 2 Timeline ดัง



รูปที่ . การเกิด Branch ใหม่

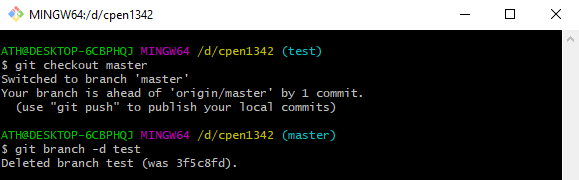
ตอนนี้เรามี Source Code อยู่ 2 Timeline การ Merge Branch จึงเป็นสิ่งสำคัญเมื่อมีการทำงานร่วมกันเป็นทีมเพื่อส่ง Source Code ที่แก้ไขไปยัง Branch หลัก การสร้าง Branch และลบ Branch นั้นมีค่าใช้จ่ายที่ถูกมากเพราะความจริงแล้ว Branch จะเก็บข้อมูลอักขระเพียง 40 ตัวอักษร ที่เป็น SHA-1 Checksum ของ Commit ที่มันชี้อยู่นั่นเอง

### การลบ Branch

ลบ Branch บน Local Host ที่ไม่ได้ใช้งานแล้ว เช่น Test Branch จะต้อง Check-Out ไปที่ Branch อื่นก่อน แล้วใช้สำสั่ง git branch -d

git checkout master

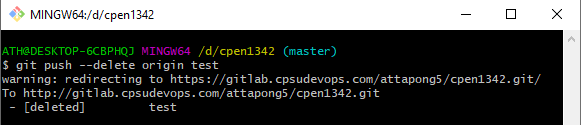
git branch -d test



รูปที่ . การลบ Branch บน Local Project

ในกรณีที่ต้องการลบ Branch บน Remote Project จะใช้คำสั่งด้านล่าง

git push --delete origin test



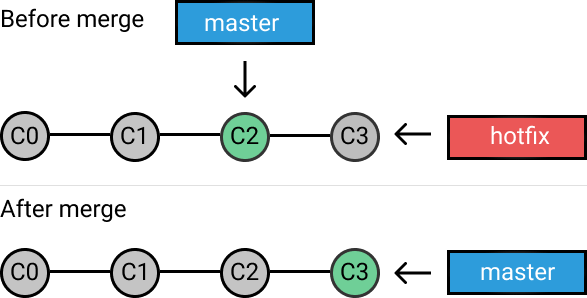
รูปที่ . การลบ Branch บน Remote Project

หลังจากลบ Branch แล้วหากมีการพบไฟล์ .swp ใน Git Branch Folder ให้ลบทิ้งด้วยคำสั่ง rm

rm .test.swp

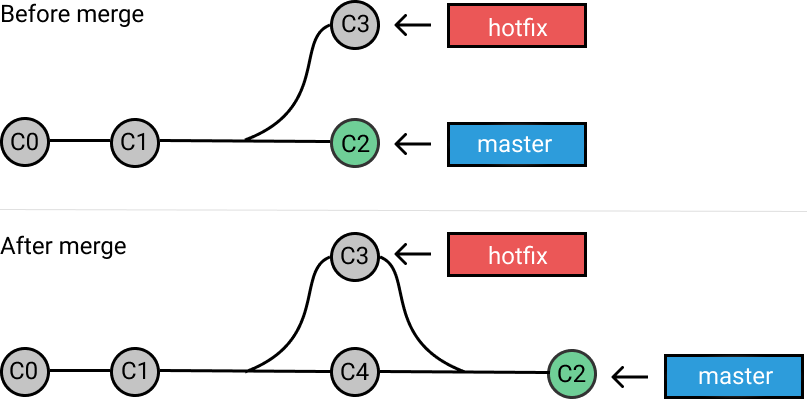
### Git Merge

การ Merge Branch เป็นหน้าที่ประการหนึ่งของ DevOps Engineer การ merge โดยปกติจะเรียกว่า Fast Forward Merge ถ้าไม่มีการเปลี่ยนแปลงหรือข้อแตกต่างกับ branch หลักแล้วจะทำการย้าย pointer ของของ branch หลักไปยัง commit ใหม่ ทำให้เกิดการเรียง commit ในรูปแบบเส้นตรงซึ่งเป็นเหตุการณ์ที่เกิดขึ้นน้อยมาก ๆ



รูปที่ . การเรียง Commit ในรูปแบบเส้นตรง

แต่ถ้า branch หลักแตกต่างจาก branch ของเราสิ่งที่เกิดขึ้นคือ Three-way merge สิ่งที่เกิดขึ้นจากการ merge คือ ทำการสร้าง commit ใหม่ขึ้นมา 1 commit ซึ่งจะขึ้นต้นด้วยคำว่า Merge branch … of เพื่อผูกหรือเชื่อมโยงของ branch ทั้งสองเข้าด้วยกัน pointer ในแต่ละ branch ไม่ได้เปลี่ยนแปลงเลยเกิดเป็น 3 ทางดังรูป



รูปที่ . Three-way merge

ผลที่ตามมาคือ การ merge จะเกิด conflict หรือไม่นั่นเอง ถ้าไม่มี conflict ก็รอดไป แต่ถ้ามีจะเราสามารถใช้คำสั่ง git status เพื่อดูปัญหาได้

git status

# On branch master

# You have unmerged paths.

# (fix conflicts and run "git commit")

# Unmerged paths:

# (use "git add ..." to mark resolution)

# both modified: xxx.txt

# no changes added to commit (use "git add" and/or "git commit -a")

จากตัวอย่างเราลองเปิดไฟล์ xxx.txt มาดู จะเจอบรรทัดที่เกิด conflict จะมีตัว maker ใส่ไว้ดังนี้

<<<<<<< คือ จุดเริ่มต้นของ Conflict

======= คือ เส้นแบ่งระหว่างการเปลี่ยนแปลงจาก branch หลัก กับ branch ของเรา

>>>>>>> คือ จุดสิ้นสุดของ Conflict

แสดงดังนี้

<<<<<<< HEAD(master)

conflicted text from HEAD(master)

=======

conflicted text from hotfix

>>>>>>> hotfix

จากนั้นเราก็ต้องตัดสินใจว่าส่วนไหนจะอยู่หรือจะไป หรือต้องรวมกัน ตรงนี้นี่เอง ที่นักพัฒนาส่วนใหญ่จะถามหาเครื่องมือในการ merge code ดังนั้นเพื่อลดความเสี่ยงของการ merge แล้วเกิด conflict จำเป็นต้องแก้ไขที่ต้นเหตุ คือ การคุยกัน และหนึ่งในการคุยกันที่ดีคือ commit-> pull-> push บ่อย ๆ นั่นเอง ส่วนเครื่องมือมันจะตามมาทีหลังเมื่อทุกอย่างเรียบร้อยก็ทำการ add, commit ต่อไป

#### ทดสอบการ Merge Branch

ก่อนที่จะ merge ทำการตรวจสอบโค้ดที่ local Host หรือเครื่องของเราว่ามีสถานะล่าสุดตรงกับ Remote หรือบน server หรือไม่ด้วยคำสั่ง git fetch

git fetch

ให้ทำการสร้าง Branch ใหม่ชื่อ Dev บน Local Host

git branch dev

เพื่อสลับไปยัง Dev Branch เราต้องใช้คำสั่ง git checkout เพื่อเคลื่อนย้าย HEAD Pointer

git checkout dev

เพื่อแสดงชื่อ Branch ทั้งบน Local Project และ Remote Project เราจะพิมพ์คำสั่ง git branch -a

git branch -a



รูปที่ . การสร้าง Branch ใหม่และตรวจสอบสถาณะ Branch บน Remote

หลังจากนั้นให้ทำการแก้ไขโค้ดใหม่ใน hello.py โดยพิมพ์คำสั่งต่อไปนี้เพิ่มจากไฟล์เดิม

def hello(name):

print('Hello %s' % name)

def count\_vowel(str):

vowel = 0

for c in str:

if c in ('A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u'):

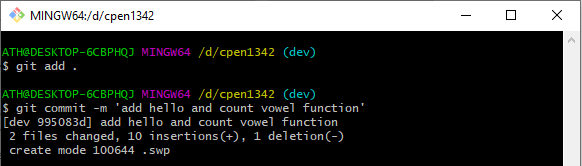
vowel = vowel + 1

return vowel

ลำดับต่อมาให้ทำการ Check-In โค้ดทั้งหมดบน Git และ Commit เพื่อเตรียมพร้อมจะส่งข้อมูลไปบน Remote Server ด้วยคำสั่ง

git add .

git commit -m 'add hello and count vowel function'



รูปที่ . การ Check-In และ Commit บน Remote Server

จากนั้นทำการเพิ่มโค้ดใน hello.py ดังนี้

# calling functions

hello('Danny')

hello('Mateo')

print('Vowel in string = %d' % count\_vowel('git merge'))

print('Vowel in string = %d' % count\_vowel('Python'))

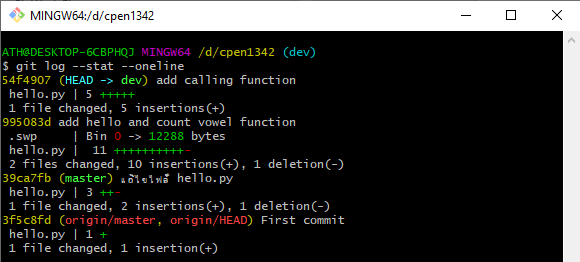
เมื่อทำการบันทึกไฟล์แล้วให้ทำการ Check-In โค้ดทั้งหมดบน Git และ Commit เช่นเดิม

git add .

git commit -m 'add calling function'

จากนั้นให้ลองใช้คำสั่ง git –log เพื่อดู History ของ Dev Branch จะมีหน้าตาดังภาพด้านล่าง

git log --stat --oneline



รูปที่ . รายละเอียดของ History ของ Git

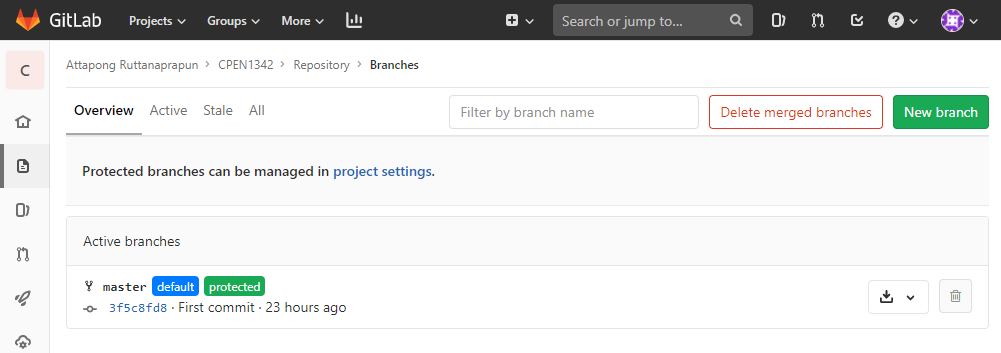
ลำดับการทำงานต่อไปจะเป็นการ Sync History กับ Remote Project ในครั้งแรกของการ Sync เราต้องสร้างการเชื่อมโยง ด้วยคำสั่ง git push --set-upstream ดังนี้

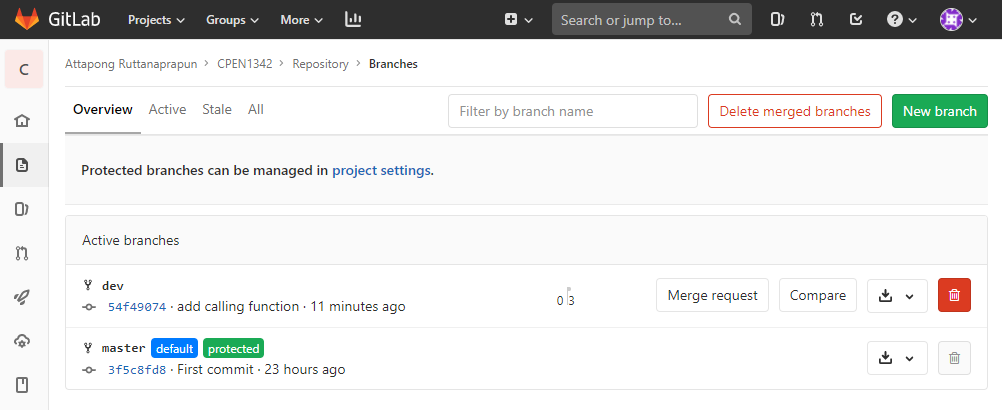
git push --set-upstream origin dev



รูปที่ . ผลที่ได้จากคำสั่ง git push

เมื่อเข้าไปยังหน้าเว็บไซต์ของ GitLab จะพบความเปลี่ยนแปลงดังนี้

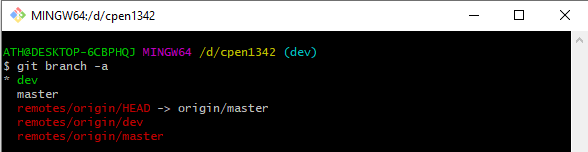




รูปที่ . ผลที่เกิดขึ้นบน Remote Server

เมื่อเราทำสอบด้วยคำสั่ง git branch -a จะพบว่า History ของ Dev Branch ที่ Local Project ถูก Sync กับ Remote Project เรียบร้อยแล้ว

git branch -a



รูปที่ . History ของ Dev Branch ที่ Local Project ถูก Sync กับ Remote Project

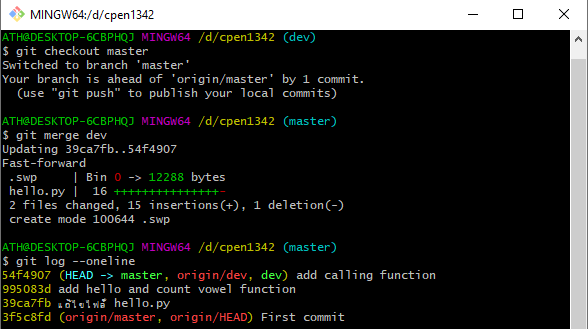
ขั้นตอนต่อไปจะทำการ Merge Dev Branch เข้ากับ Master Branch โดยในลำดับแรกให้ทำการ Check-Out ที่ Master Branch ก่อนแล้วจึงใช้คำสั่ง git merge ดังนี้

git checkout master

git merge dev

จากนั้นตรวจสอบสถานะของ Git จากคำสั่ง git log ดังนี้

git log --oneline



รูปที่ . การ Merge แบบ Fast-forward

จากภาพจะเห็นการ Merge แบบ Fast-forward ซึ่งมีการเคลื่อนย้าย Master Branch Pointer ไปข้างหน้ายัง Commit บนสุดของ Timeline ที่ตำแหน่งเดียวกับ Dev Branch Pointer

ให้เราย้ายกลับไปยัง Dev Branch ด้วยคำสั่ง git checkout ซึ่งทำให้มีการเคลื่อนย้าย HEAD Pointer ไปที่ Dev Branch

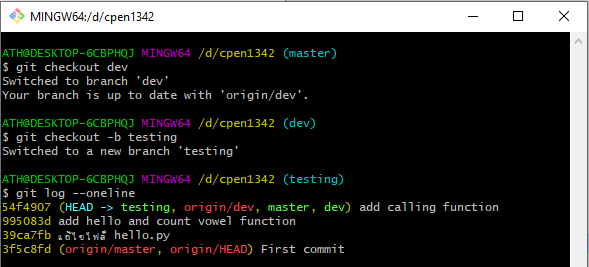
git checkout dev

สร้าง Branch ใหม่ชื่อ Testing แล้ว Check-Out ที่ Testing Branch ในคราวเดียวกัน ด้วยคำสั่ง git checkout -b

git checkout -b testing

จากนั้นทำการตรวจสอบสถานะของ Git ใหม่ด้วยคำสั่ง git log ดังนี้

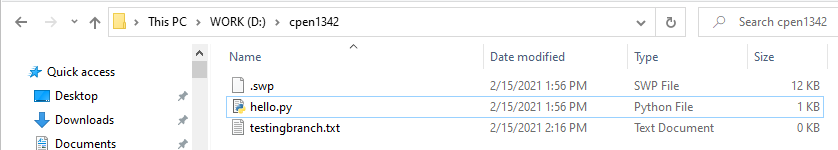
git log --oneline



รูปที่ . สร้าง branch ใหม่ที่ชื่อ testing

ผลลัพธ์สุดท้ายจะได้ Testing Branch Pointer ที่ชี้ไปยัง Commit เดียวกับ Dev Branch Pointer ต่อมาเราจะแตก History ออกเป็น 2 Timeline ที่เกิดจาก Testing Branch และ Master Branch เพื่อทำ Three Way Merge ดังตัวอย่างต่อไปนี้

ทำการเพิ่มไฟล์ชื่อ testingbranch.txt จาก Editor ที่ถนัดโดยไม่ต้องเพิ่มรายละเอียดใด ๆ ในไฟล์แล้วบันทึกไว้ใน Work Directory ในที่นี้คือ d:\cpen1342 สำหรับ Testing Branch

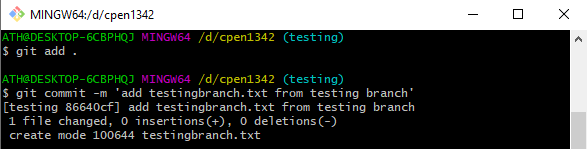


รูปที่ . การเพิ่มไฟล์ testingbranch.txt

จากนั้นจึง Check-In ไฟล์ testingbranch.txt ที่ Testing Branch เมื่อบันทึกไฟล์แล้ว Check-In ด้วยคำสั่ง git add และ Commit ด้วยคำสั่งต่อไปนี้

git add .

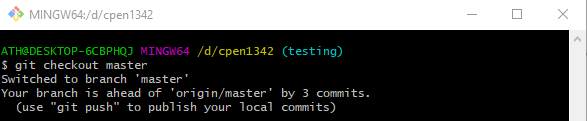
git commit -m 'add testingbranch.txt from testing branch'



รูปที่ . การ Add และ Commit ไฟล์บน Testing Branch

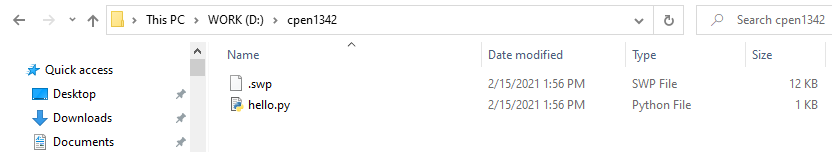
จากนั้นให้ย้ายไปที่ Master Branch ด้วยคำสั่ง checkout

git checkout master



รูปที่ . ย้ายการทำงานไปยัง Master Branch ด้วยคำสั่ง git checkout

จากนั้นให้ใช้ Windows Explorer ย้อนกลับไปดูข้อมูลในโฟลเดอร์ D:\cpen1342 จะพบข้อมูลในโฟลเดอร์ดังรูปที่ 4.98



รูปที่ . ข้อมูลในโฟลเดอร์ cpen1342 เมื่อกลับมาที่ Master Branch

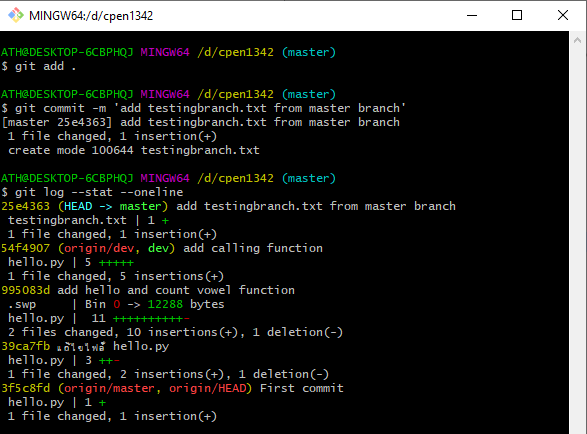
เราจะเห็นว่าไฟล์ testingbranch.txt ได้หายไปเนื่องจากตอนนี้เราย้ายการทำงานมายัง Master Branch นั่นเอง จากนั้นให้ทำการเพิ่มไฟล์ testingbranch.txt เช่นเดิมแต่ให้เพิ่มข้อความเข้าไปในไฟล์ตามต้องการเช่น “cpen1342 git test” เป็นต้นหลังจากนั้นให้กดบันทึกไฟล์แล้วทำการ Check-In และ Commit ด้วยคำสั่งต่อไปนี้

git add .

git commit -m 'add testingbranch.txt from master branch'

ตอนนี้เราจะมี Source Code อยู่ 2 Timeline แต่เมื่อใช้คำสั่ง git log เราจะเห็น History เฉพาะใน Master Branch เท่านั้น

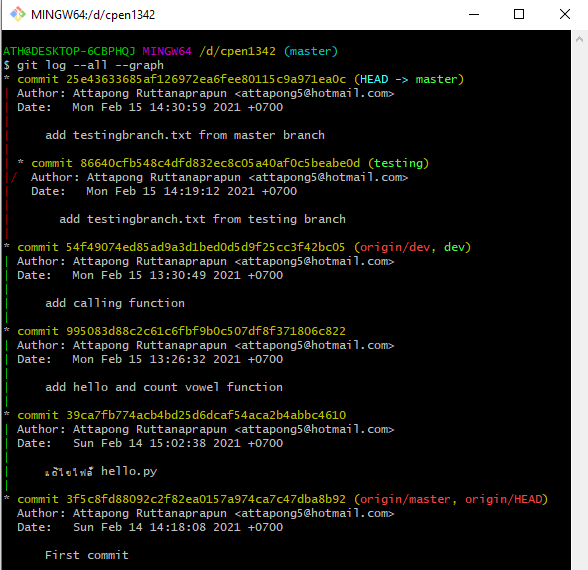
git log --stat --oneline



รูปที่ . git log จะมองเห็นเฉพาะ Branch ที่ทำงานอยู่เท่านั้น

ทั้งนี้เราสามารถดู Source Code ทั้ง 2 Timeline พร้อมกัน ด้วยคำสั่ง git log --all –graph

git log --all --graph



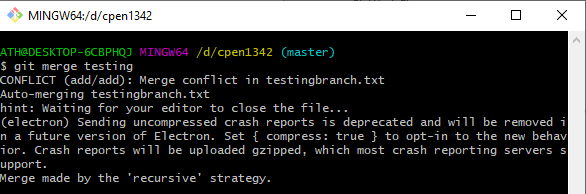
รูปที่ . การดู Git Log ของทุก ๆ Branch

เมื่อ Software Developers แก้ไข Source Code เสร็จแล้วจึง Merge Branch ที่ Stable น้อยกว่ากลับมายัง Branch หลักเช่น Master Branch ทั้งนี้ให้เราใช้คำสั่ง git branch เพื่อให้แน่ใจว่าเราอยู่ใน Branch ที่ถูกต้องหรือไม่

ในกรณีตัวอย่างนี้เราอยู่ที่ Master Branch ซึ่งเราต้องการให้เป็น Branch หลักแล้วดังนั้นเราจึงทำการรวม Branch ด้วยคำสั่ง git merge ดังนี้

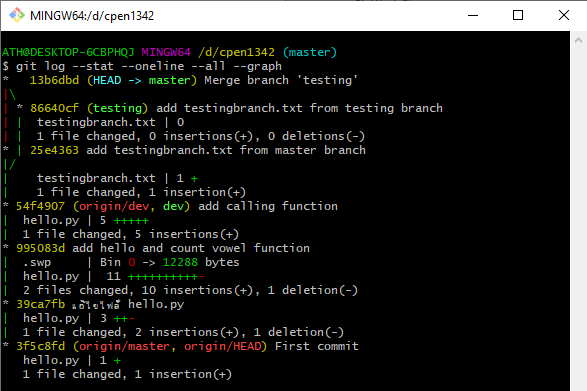
git merge testing

เมื่อ Merge Testing Branch จาก Timeline หนึ่งกลับมายัง Master Branch ในอีก Timeline หนึ่ง ด้วยคำสั่ง git merge แล้ว Git พบความแตกต่างระหว่าง Version ของ Code ทั้งสองในไฟล์ testingbranch.txt ซึ่งเป็นไฟล์ชื่อเดียวกันแต่เนื้อหาภายในแตกต่างกันทำให้เกิดข้อขัดแย้งหรือ Conflict นั่นเอง ตัว Git จะทำการเปิดไฟล์ดังกล่าวเพื่อให้ทำการแก้ไขไฟล์เสียใหม่ในการ Merge และการ Merge จะเสร็จสิ้นเมื่อทำการปิด Editor แล้วเท่านั้น



รูปที่ . การแจ้งเตือนจาก Git เมื่อพบ Conflict

เมื่อรวม Branch ด้วยคำสั่ง git merge แล้ว Git จะสร้าง Version ของ Code ที่เป็นผลลัพธ์จากการเปรียบเทียบแบบ 3 ทาง คือระหว่าง Commit ที่เป็น Root ของทั้ง 2 Timeline ได้แก่ Commit ID 25e4363 (Base), Commit สุดท้ายของ Testing Branch ได้แก่ Commit ID 86640cf (Source) และ Commit สุดท้ายของ Master Branch ได้แก่ Commit ID 13b6dbd (Target)



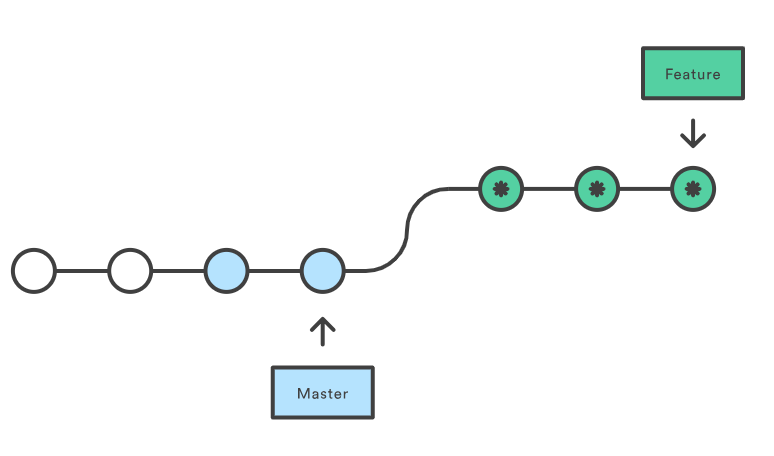
รูปที่ . ผลลัพธ์ที่ได้จากการ Merge

หลังจากที่เราเลือกโค้ดจาก Source เพื่อแก้ไข Conflict ด้วยมือดังแล้วให้ทำการกด Save แล้ว Check-In กระบวนการเลือกโค้ดนี้เป็นส่วนหนึ่งของการ Review Code ใน CI/CD นั่นเอง

### Git Rebase

Rebase เป็นอีกวิธีในการ Integrate การเปลี่ยนแปลงจาก Branch หนึ่งไปอีก Branch หนึ่ง การ Rebase จะทำการรวบรวมการเปลี่ยนแปลงทั้งหมดให้อยู่ใน "Patch" เดียว จากนั้นมันจะ Integrate Patch ไปยัง Target Branch

Rebase แตกต่างจาก Merge ตรงที่การ Rebase จะทำให้ History ดูไม่ซับซ้อน เนื่องจากมันจะย้าย Work ที่เสร็จแล้วจาก Branch หนึ่งไปอีก Branch หนึ่ง ในกระบวนการนี้ History ที่ไม่ต้องการจะถูกกำจัดออกไป



รูปที่ . Git Rebase

การทำ Git Rebase จะมีข้อดีดังต่อไปนี้

* ปรับปรุง History ที่ซับซ้อนให้มีประสิทธิภาพมากขึ้น
* จัดการกับ Single Commit ได้ง่าย (เช่น การคืนค่า)
* ช่วย Clean ตัว Intermediate Commits ด้วยการทำให้มันเป็น Single Commit ซึ่งจะเป็นประโยชน์สำหรับทีม DevOps

ข้อด้อยที่ตามมา

* ด้วยเหตุที่มันลดจำนวน Commit ให้เหลือน้อยลง อาจเป็นการซ่อน Context บางอย่างได้
* การ Rebase ตัว Public Repositories อาจสร้างความเสียหายได้หากมีการทำงานกันเป็นทีม
* มี Process การทำงานมากขึ้นเพราะเมื่อใช้ Rebase จะมีการ Update Feature Branch ของคุณตลอดเวลา
* การ Rebase ด้วย Remote Branches นั้น มันจำเป็นต้อง Force Push ซึ่งปัญหาใหญ่ที่ต้องเจอ คือ มีการ Force Push โดยที่ยังไม่ได้ตั้งค่า default ของ git push ส่งผลให้มีการ Update ทุก Branch ที่มีชื่อเหมือนกันทั้งใน Local และที่เป็น Remote ซึ่งถือเป็นสิ่งที่น่ากังวลอย่างมาก

หากเราทำ Rebase อย่างไม่ถูกต้อง และทำการ Rewrite History โดยไม่ได้ตั้งใจ มันอาจนำไปสู่ปัญหาร้ายแรงได้ ดังนั้น เราต้องรู้ว่ากำลังทำอะไรอยู่ตลอดเวลา

ทำการ Rebase ตัว Feature Branch ไปยัง Master Branch โดยใช้คำสั่งต่อไปนี้

git checkout feature

git rebase master

สิ่งนี้จะย้าย Feature Branch ทั้งหมดไปอยู่บน Master Branch ซึ่งทำได้โดยการ Re-write ตัว History ของ Project ด้วยการสร้างCommit ใหม่สำหรับแต่ละ Commit ใน Original (Feature) Branch

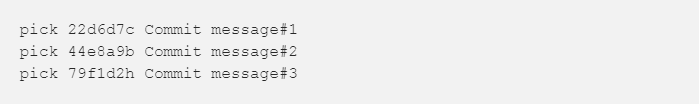
#### Interactive Rebasing

วิธีนี้มีประสิทธิภาพกว่าการ Rebase แบบอัตโนมัติเนื่องจากสามารถควบคุม Commit History ของ Branch ได้เป็นอย่างดี โดยทั่วไปแล้วใช้วิธีนี้เพื่อ Clean ตัว History (ที่มีอยู่เป็นจำนวนมาก) ก่อนที่จะรวม Feature Branch เข้ากับ Master Branch

git checkout feature

git rebase -i master

นี่จะเป็นการเปิด Editor ขึ้นมาและจะแสดงรายการ Commit ทั้งหมดที่กำลังจะถูกย้าย



รูปที่ . รายละเอียดที่แสดงไว้ในตัว Editor

หลังจากนี้เราสามารถใช้ตัว Editor เข้าไปแก้ไขเพื่อกำหนดว่า Branch จะมีลักษณะอย่างไรหลังจากการ Rebase แล้ว ด้วยการจัดลำดับใหม่ เราสามารถทำให้ History เป็นไปตามที่ต้องการ อย่างเช่น คุณสามารถใช้คำสั่งอย่าง fixup, squash, edit เป็นต้นแทนการใช้ pick

### การเลือกใช้ Git Merge และ Git Rebase

เป็นการยากที่จะตัดสินใจเลือกว่าจะใช้แบบไหนดีระหว่าง Git Rebase กับ Git Merge เนื่องจากแต่ละทีมก็มีความแตกต่างกัน คนในทีมต้องพิจารณาคำถามหลายข้อขณะที่ตั้ง Policy ของ Git Rebase กับ Git Merge ซึ่งทุกอย่างล้วนขึ้นอยู่กับแต่ละทีม โดยให้พิจารณาที่ความชำนาญของการ Rebase กับทักษะในเรื่อง Git ของทีมงาน กำหนดระดับความสำคัญที่คุณให้กับความยาก-ง่ายของการ Rebase เทียบกับเรื่อง การตรวจสอบย้อนกลับ (Traceability) และ History ของการ Merge ที่สุดแล้ว การตัดสินใจว่าจะใช้ Merge หรือ Rebase ควรพิจารณาไปถึงบริบทที่ชัดเจนของ Branching Strategy

#### ข้อแนะนำ

เมื่อทีมเติบโตขึ้น มันจะกลายเป็นเรื่องยากที่จะจัดการหรือติดตามการเปลี่ยนแปลงของการ Develop ด้วย Merge Policy ตลอดเวลา หากต้องการมี Commit History ที่ชัดเจนและสามาถเข้าใจได้ง่าย การใช้ Rebase ดูจะสมเหตุสมผลและมีประสิทธิภาพ และหากลองพิจารณาถึงสถานการณ์และแนวทางต่อไปนี้เราน่าจะได้ประโยชน์อย่างมากจากการใช้ Rebase:

* เป็นการพัฒนาด้วยคนคนเดียว: หากเราไม่จำเป็นต้อง Share งานกับคนอื่น ในจุดนี้ควรเลือกใช้ Rebase มากกว่า Merge เพื่อรักษาความเป็นระเบียบของ History หากมีที่เก็บงานส่วนตัวแยกออกมา ซึ่งไม่ได้ Share กับ Developer คนอื่น เราจะปลอดภัยที่จะใช้ Rebase แม้หลังจากที่เรา Push Branch ไปแล้วก็ตาม
* โค้ดของเราพร้อมสำหรับการ Review แล้ว: เราสร้าง Pull Request แล้ว แต่มีคนอื่นกำลัง Review งานของเราอยู่ ขณะเดียวกันอาจกำลังนำงานของเราไป Review ในเครื่องของพวกเขา ในจุดนี้ เราไม่ควร Rebase งานของเรา เราควรสร้าง "Rework" Commits และทำการ Update Feature Branch ของเรา ซึ่งจะช่วยในการตรวจสอบย้อนกลับในการ Pull Request และป้องกันปัญหาที่อาจเกิดขึ้นกับ History อีกด้วย
* การ Review เสร็จแล้ว และพร้อมที่จะ Integrate เข้ากับ Target Branch: ถึงตอนนี้ เรากำลังจะลบ Feature Branch ของคุณ ทำให้ Developer คนอื่นไม่สามารถมาดึงข้อมูลการเปลี่ยนแปลงเหล่านี้ได้อีก นี่เป็นโอกาสของเราที่จะทำให้ History ของเราเป็นระเบียบ ในจุดนี้เราสามารถ Rewrite History และเก็บ Original Commits และ ‘PR Rework’ ที่น่ารำคาญและ ‘Merge’ Commits เข้ากับกลุ่มของ Commit ที่เรา Focus อยู่ การสร้าง Merge สำหรับ Commits เหล่านี้ถือเป็นทางเลือก แต่มันมีประโยชน์มาก โดยมันจะ Record เมื่อ Feature ไปอยู่ที่ Master แล้ว

## [8] การจัดการกลุ่มและสมาชิกใน Gitlab

การจัดการสมาชิกของแต่ละ Project จะมีสิทธิ์การใช้งานของ Project และ Group เข้ามาเกี่ยวข้องด้วยเสมอซึ่งมีรายละเอียดดังต่อไปนี้

ตารางที่ . Project Permissions

| Action | User Type | | | | |
| --- | --- | --- | --- | --- | --- |
| Guest | Reporter | Developer | Master | Owner |
| Create new issue | ✓ | ✓ | ✓ | ✓ | ✓ |
| Create confidential issue | ✓ | ✓ | ✓ | ✓ | ✓ |
| View confidential issues | ✓ | ✓ | ✓ | ✓ | ✓ |
| Leave comments | ✓ | ✓ | ✓ | ✓ | ✓ |
| See a list of jobs | ✓ | ✓ | ✓ | ✓ | ✓ |
| See a job log | ✓ | ✓ | ✓ | ✓ | ✓ |
| Download and browse job artifacts | ✓ | ✓ | ✓ | ✓ | ✓ |
| View wiki pages | ✓ | ✓ | ✓ | ✓ | ✓ |
| Pull project code |  | ✓ | ✓ | ✓ | ✓ |
| Download project |  | ✓ | ✓ | ✓ | ✓ |
| Create code snippets |  | ✓ | ✓ | ✓ | ✓ |
| Manage issue tracker |  | ✓ | ✓ | ✓ | ✓ |
| Manage labels |  | ✓ | ✓ | ✓ | ✓ |
| See a commit status |  | ✓ | ✓ | ✓ | ✓ |
| See a container registry |  | ✓ | ✓ | ✓ | ✓ |
| See environments |  | ✓ | ✓ | ✓ | ✓ |
| Create new environments |  |  | ✓ | ✓ | ✓ |
| Use environment terminals |  |  |  | ✓ | ✓ |
| Stop environments |  |  | ✓ | ✓ | ✓ |
| See a list of merge requests |  | ✓ | ✓ | ✓ | ✓ |
| Manage/Accept merge requests |  |  | ✓ | ✓ | ✓ |
| Create new merge request |  |  | ✓ | ✓ | ✓ |
| Create new branches |  |  | ✓ | ✓ | ✓ |
| Push to non-protected branches |  |  | ✓ | ✓ | ✓ |
| Force push to non-protected branches |  |  | ✓ | ✓ | ✓ |
| Remove non-protected branches |  |  | ✓ | ✓ | ✓ |
| Add tags |  |  | ✓ | ✓ | ✓ |
| Write a wiki |  |  | ✓ | ✓ | ✓ |
| Cancel and retry jobs |  |  | ✓ | ✓ | ✓ |
| Create or update commit status |  |  | ✓ | ✓ | ✓ |
| Update a container registry |  |  | ✓ | ✓ | ✓ |
| Remove a container registry image |  |  | ✓ | ✓ | ✓ |
| Create new milestones |  |  |  | ✓ | ✓ |
| Add new team members |  |  |  | ✓ | ✓ |
| Push to protected branches |  |  |  | ✓ | ✓ |
| Enable/disable branch protection |  |  |  | ✓ | ✓ |
| Turn on/off protected branch push for devs |  |  |  | ✓ | ✓ |
| Enable/disable tag protections |  |  |  | ✓ | ✓ |
| Rewrite/remove Git tags |  |  |  | ✓ | ✓ |
| Edit project |  |  |  | ✓ | ✓ |
| Add deploy keys to project |  |  |  | ✓ | ✓ |
| Configure project hooks |  |  |  | ✓ | ✓ |
| Manage runners |  |  |  | ✓ | ✓ |
| Manage job triggers |  |  |  | ✓ | ✓ |
| Manage variables |  |  |  | ✓ | ✓ |
| Manage pages |  |  |  | ✓ | ✓ |
| Manage pages domains and certificates |  |  |  | ✓ | ✓ |
| Switch visibility level |  |  |  |  | ✓ |
| Transfer project to another namespace |  |  |  |  | ✓ |
| Remove project |  |  |  |  | ✓ |
| Force push to protected branches |  |  |  |  |  |
| Remove protected branches |  |  |  |  |  |
| Remove pages |  |  |  |  | ✓ |

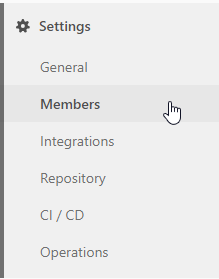
ตารางที่ . Group Permissions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Action | User Type | | | | |
| Guest | Reporter | Developer | Master | Owner |
| Browse group | ✓ | ✓ | ✓ | ✓ | ✓ |
| Edit group |  |  |  |  | ✓ |
| Create subgroup |  |  |  |  | ✓ |
| Create project in group |  |  |  | ✓ | ✓ |
| Remove project in group |  |  |  |  | ✓ |
| Manage group members |  |  |  |  | ✓ |
| Remove group |  |  |  |  | ✓ |
| Manage group labels |  | ✓ | ✓ | ✓ | ✓ |

### Manage users in Project

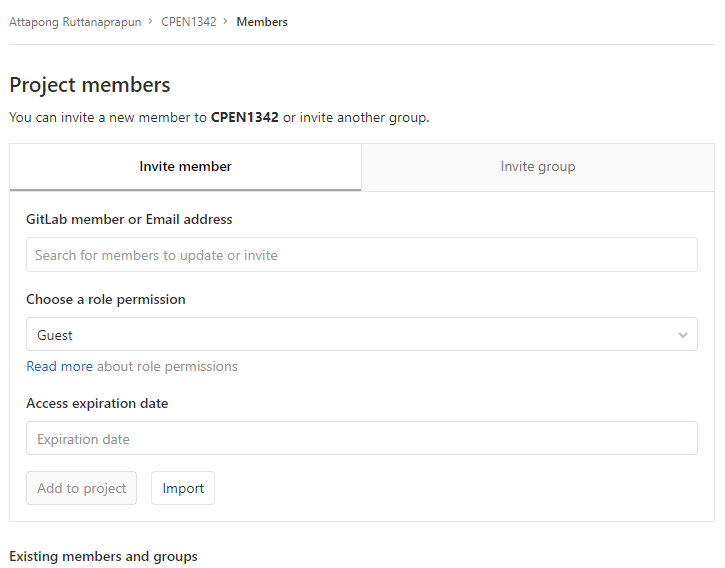
ในการเพิ่มสมาชิกเข้ามาในโปรเจกต์ให้ทำดังนี้

1. ไปที่เมนู Settings > Members แล้วเลือกโปรเจกต์ที่ต้องการจัดการสมาชิก



รูปที่ . เลือกส่วนของการจัดการสมาชิก

2. เลือกแถบ Invite member จากนั้นให้พิมพ์ชื่อหรือ Email ของสมาชิก Gitlab ที่ต้องการ ในช่อง GitLab member or Email address



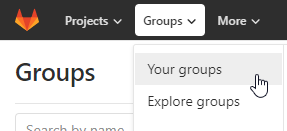
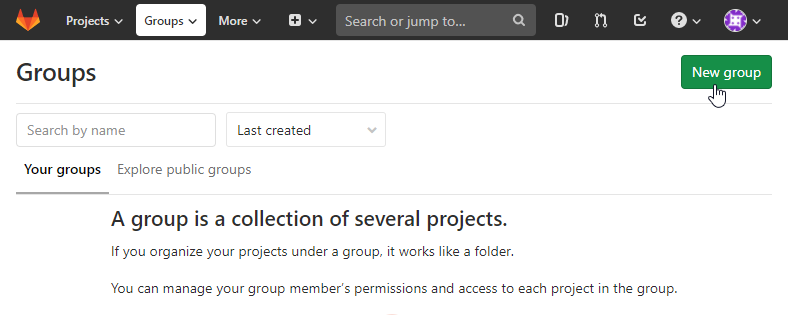
รูปที่ . การเพิ่มสมาชิกในโปรเจกต์

ให้ทำการเพิ่มสมาชิกพร้อมทั้งกำหนดตำแหน่งตามต้องการ

### การสร้าง Group ใหม่

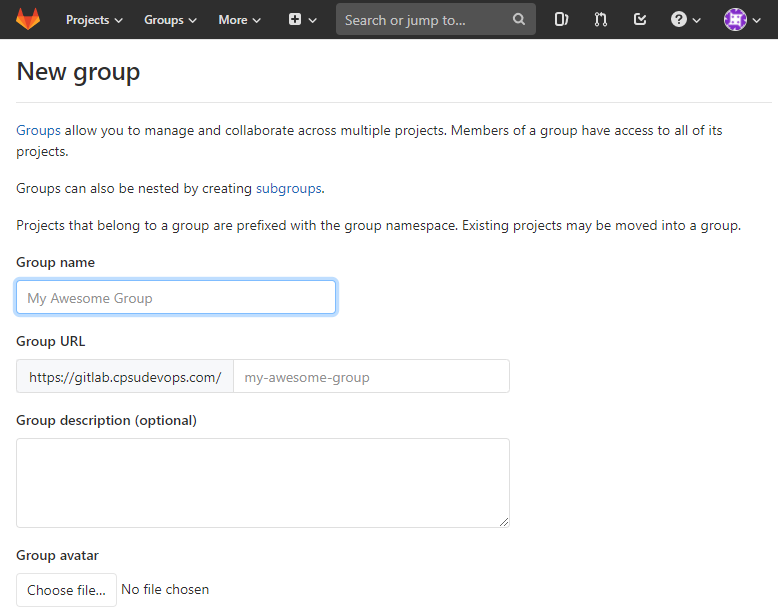
สำหรับ Group ใน Gitlab จะช่วยให้สมาชิกในกลุ่มสามารถจัดการและทำงานร่วมกันในหลายโครงการที่กลุ่มนั้นเข้าถึงได้ เมื่อเริ่มใช้งาน Gitlab เราจะไม่มีกลุ่มจนกว่าจะมีผู้ใช้งานอื่นเชิญมาเข้ากลุ่มหรือสร้างกลุ่มใหม่ขึ้นมาเอง ในการสร้าง Group ใหม่ให้ดำเนินการตามขั้นตอนต่อไปนี้

1. เลือกเมนู Groups ด้านบนจากนั้นเลือก Your groups

รูปที่ . การสร้าง Group ใหม่

2. เมื่อเข้าสู่หน้าจัดการกลุ่มแล้วให้ทำการกดปุ่ม New group จะเข้าสู่หน้ากรอกรายละเอียดของกลุ่มให้กลอกตามความต้องการ



รูปที่ . กรอกรายละเอียดสำหรับกลุ่มใหม่

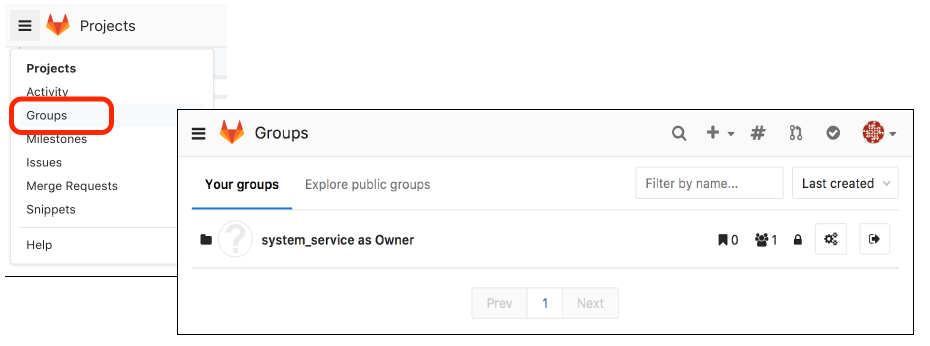
เมื่อกรอกรายละเอียดต่าง ๆ ตามต้องการแล้วให้กดปุ่ม Create group เป็นอันเสร็จสิ้น

### Manage Subgroup

Admin ของระบบจะสร้าง Group และ Assign User ในระบบเข้าไปและให้ตำแหน่งเป็น Owner เพื่อทำหน้าที่บริหารจัดการภายใน Group ซึ่ง User ที่ได้รับตำแหน่งเป็น Owner คนแรก จะได้รับสิทธิเพิ่มเติมคือ สิทธิในการสร้าง Group เพื่อให้สามารถสร้าง Subgroup ได้ ซึ่งเมื่อมีสิทธิสร้าง Group ก็จะสามารถสร้าง Group ใหม่ได้เหมือนกัน (เนื่องจาก GitLab ไม่ได้แยกสิทธิการสร้าง Group และ Subgroup ออกจากกัน) แต่เพื่อความเป็นระเบียบหากมีการสร้าง Group ใหม่ที่ไม่ใช่ Subgroup เกิดขึ้นโดยไม่มีการแจ้งต่อ Admin ก็จะอาจจะมีการพิจารณาลบ Group นั้นออกจากระบบ

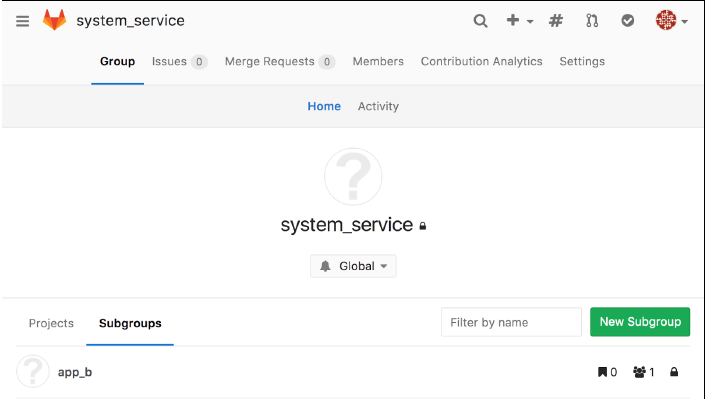
ขั้นตอนการสร้าง Subgroup มีดังนี้

1. ที่เมนูมุมขวาบนเลือก Groups จากนั้นจะแสดงรายชื่อ Group ที่เป็น Member แล้วกดที่ชื่อ Group ที่ต้องการเพิ่ม Subgroup



รูปที่ . การเข้าจัดการ Group

2. กดปุ่ม Subgroups แล้วจะแสดง Subgroup ที่มีใน Group จากนั้นกดปุ่ม New Subgroup สีเขียวทางขวาเพื่อสร้าง Subgroup

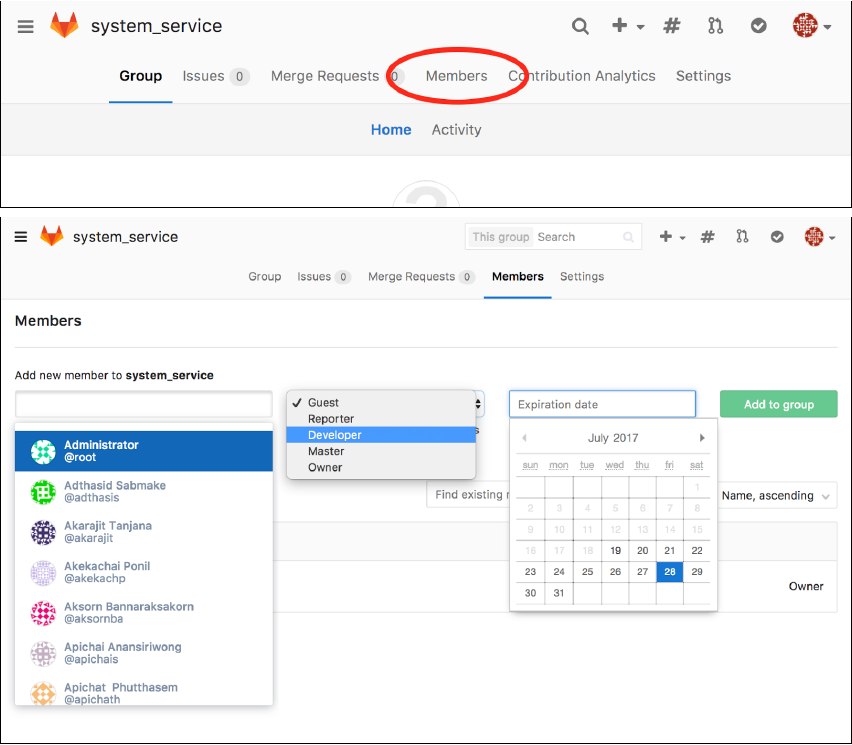


รูปที่ . การเพิ่ม Subgroup

### Manage users in Group

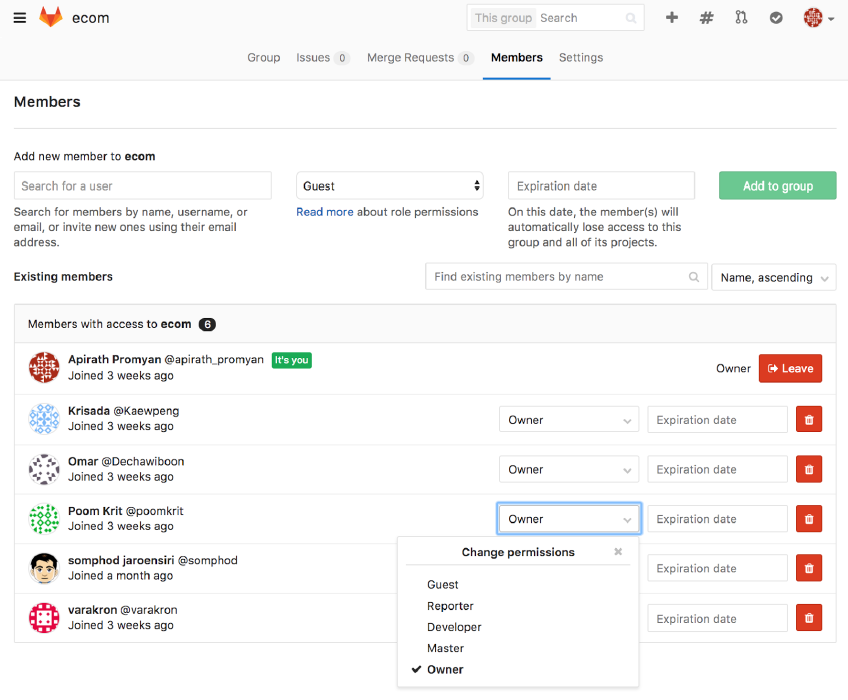
การเพิ่มหรือเชิญผู้ใช้งานอื่นเข้าใน Group มีขั้นตอนดังนี้

1. การเพิ่ม/เชิญ User ในระบบเข้า Group (Invite Users to Group) กดที่เมนูมุมขวาบนเลือก Groups จากนั้น กดที่ชื่อ Group แล้วกดแท็บ Members พิมพ์ชื่อหรืออีเมล์ ของ User ที่ลงทะเบียนในระบบเรียบร้อยแล้ว กำหนดตำแหน่ง วันหมดอายุจะกำหนดหรือไม่กำหนดก็ได้ และกด Add to group



รูปที่ . เลือก Members ใน Group

2. ภายในกลุ่มสมาชิกแต่ละตำแหน่งจะมีสิทธิ์การใช้งานต่างกันดังที่ได้แสดงไว้ในตารางที่ 4.1 และ ตารางที่ 4.2



รูปที่ . การเปลี่ยนตำแหน่งของสมาชิกในกลุ่ม

### กระบวนการจัดการทำงานด้วย Git

จากการใช้งาน GitLab ที่ผ่านมาเราสามารถสรุปกระบวนการการใช้งาน Git ได้ดังนี้

#### ขั้นตอนการพัฒนา

1. เขียนโค้ด

2. เลือกไฟล์ที่ต้องการเข้า Staged

3. Commit

4. เขียนโค้ดต่อ

5. เลือกไฟล์ที่ต้องการเข้า Staged

6. Commit

7. วนลูปไปเรื่อย ๆ จนกว่างานจะเสร็จ

#### ขั้นตอนการ Sync ข้อมูลไปที่ Remote Server

1. Fetch เพื่อเช็คว่ามีใคร Push อะไรขึ้น Remote หรือป่าว (ไม่จำเป็นต้องทำก็ได้)

2. ถ้ามีก็จะต้อง Pull โค้ดก่อน แล้ว Merge Commit ให้เรียบร้อย

3. Push ข้อมูลขึ้น Remote

#### การแก้ปัญหาการเกิด Conflict ในการ Merge Branch

1. Fetch เพื่อเช็คว่ามีใคร Push อะไรขึ้น Remote หรือเปล่า (ไม่จำเป็นต้องทำก็ได้)

2. Pull เพื่อดึงข้อมูลมาไว้ในเครื่อง

3. เกิด Conflict

4. แก้ไขโค้ดให้เหมาะสม

5. Merge Commit

6. Push ข้อมูลขึ้น Remote

### เขียนโค้ดไปจนถึงเมื่อไรถึงจะต้อง Commit

ในการ Commit แต่ละครั้งถึงแม้ว่าจะไม่ได้มีใครมากำหนดมาตรฐาน แต่การ Commit ที่ดีควรทำให้สามารถไล่โค้ดทีหลังได้ง่าย สามารถทำงานร่วมกับคนอื่นได้ และต้องทำเหมือนเดิมทุกครั้งจนเป็นนิสัย ไม่ใช่ว่านึกออกเมื่อไรก็ค่อย Commit

การ Commit ที่ดีต้องกระชับและเล็กมากพอที่จะไล่ดูใน History ได้ง่าย ๆ เวลามีโปรเจคตัวหนึ่ง เราจะแยกสิ่งที่ต้องทำก็จะแบ่งออกมาเป็น Feature แล้วก็แบ่งย่อยลงไปอีกให้เป็นระดับ Function ถึงจะเริ่มเขียนโค้ด เมื่อทำในแต่ละ Function เสร็จก็จะ Commit ให้เรียบร้อย แล้วก็ทำ Function ต่อไป และวนแบบนี้ไปเรื่อย ๆ จน Feature นั้นเสร็จ

ทั้งนี้ไม่แนะนำให้ทำในระดับ Feature เสร็จแล้วค่อย Commit เพราะการทำแบบนั้นเราจะใช้ประโยชน์ Git ได้แค่การ Backup เท่านั้น แต่จะไล่ดู History เวลาแก้ไขโค้ดได้ยาก และไม่สามารถย้อนข้อมูลกลับได้เลย เพราะมันจะย้อนกลับไปทั้งหมดเลย โค้ดที่เขียนไว้ทั้งหมดก็จะหายไปพร้อม ๆ กัน และไม่แนะนำให้ Commit โค้ดที่ยังเขียนไม่เสร็จเช่นกัน โค้ดที่เขียนไม่เสร็จ ณ ที่นี้หมายถึงโค้ดที่ไม่สามารถกด Run เพื่อทดสอบโปรแกรมได้ เพราะถ้าทำงานร่วมกับคนอื่น แล้วคนอื่น Pull ข้อมูลไป กลายเป็นว่าคน ๆ นั้นก็ Run โปรแกรมไม่ได้เช่นกัน ต้องเสียเวลานั่งแก้โค้ดของเราอีก

อย่าลืมใส่ใจกับ Commit Message และไม่ควรใส่อะไรที่ไม่สื่อความหมายในการ Commit เช่น “Edit code”, “Commit” หรือ “Refactor” เป็นต้น เพราะเราจะไม่มีทางรู้เลยว่า Commit นั้นทำอะไรลงไป

### ต้อง Push เมื่อไร

เนื่องจากการ Push นั้น ในบางครั้งก็ต้อง Pull แล้ว Merge Commit ให้เรียบร้อยก่อนถึงจะทำได้ ดังนั้นไม่แนะนำให้ Push ก่อนปิดคอมพิวเตอร์เพื่อเลิกงาน เนื่องจากถ้ามีคน Push ก่อนหน้า แล้วเมื่อเรา Pull มาก็พบว่ามัน Conflict กันสุดท้ายแล้วเราต้องมานั่งแก้ Conflict ชุดใหญ่จนไม่ได้กลับบ้านนั่นเอง ดังนั้นทางที่ดีควร Push ทุกครั้งที่มีโอกาสดีกว่า อย่างน้อยถ้ามันจะ Conflict เราจะได้แก้ตั้งแต่เนิ่น ๆ และไม่เยอะมากนัก ที่สำคัญมันจะช่วยให้ทุกคนในทีมรู้ Progress ด้วยกันได้จากสถานะการ Push ด้วย

นอกจากนี้ ในบางที่ที่มีการทำ Continuous Integration (CI) ที่จะดึง Commit จาก Remote ไปทำ Testing ถ้าเรา Push บ่อย ๆ แล้ว Test เกิด Failed ขึ้นมาก็จะทำให้รู้ตัวได้ไวและแก้ไขได้ทันที

## [9] การใช้งาน GitHub

GitHub เป็น development platform ที่เอาไว้แบ่งปัน source-code ของเราให้กับชาวโลก ไม่ว่าจะเป็น co-workers, classmates, teammates หรือไม่ว่าใครก็ตาม รองรับทั้ง open-source และ business เราสามารถ host, review code, manage projects และสามารถ build software ได้แม้ตัวมันเองจะไม่มี build-in เหมือนเจ้าอื่น ๆ แต่ก็มีทูลอีกมากที่ทำมารองรับ มีผู้ใช้ GitHub ราว ๆ 31 ล้านคน มี Repository มากกว่า 100,000,000 repository โดยปัจจุบันเป็นของ Microsoft

มีบริษัทประมาณ 2,000,000 บริษัท ที่นำ GitHub ไปทำสำหรับ Business และ Organization ทั้ง Enterprise Cloud และ Enterprise Server เช่น airbnb, NetFlix, Medium, IBM, SAP, PayPal, Spotify, Bloomberg, StackShare

### สมัครใช้งาน GitHub

ไปที่ https://github.com/ และทำการ Sign up ให้เรียบร้อย

เชื่อมต่อ Github ด้วย SSH

1. ตรวจสอบว่ามี SSH keys แล้วหรือไม่ ผ่านโปรแกรม Git Bash

$ ls -al ~/.ssh

2. ถ้ายังไม่มี ให้สร้างใหม่โดยใช้ GitHub email โปรแกรมจะถามชื่อไฟล์ และรหัสผ่าน ถ้าไม่ต้องการเปลี่ยนแปลงก็กด Enter ไปได้เลย

$ ssh-keygen -t rsa -b 4096 -C "your\_email@example.com"

3. เพิ่ม SSH keys ที่สร้างมาเข้าใน ssh-agent

$ eval $(ssh-agent -s)

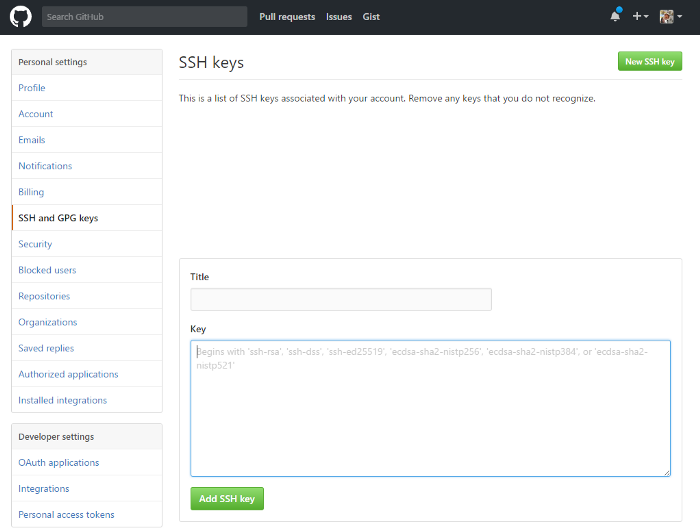
Agent pid 59566

$ ssh-add ~/.ssh/id\_rsa

4. เพิ่ม SSH keys ในบัญชี GitHub ของเรา โดยให้คัดลอก SSH key ก่อน

$ clip < ~/.ssh/id\_rsa.pub

5. ล็อกอินเข้า https://github.com เลือกที่รูปโปรไฟล์ > เลือก Setting > เลือกเมนู SSH and GPG keys > คลิก New SSH Key > ให้ตั้งชื่อเครื่องของเราที่ “Title” และวาง SSH key ลงไปที่ “Key” > สุดท้ายกด Add SSH Key



รูปที่ . การเพิ่ม SSH Key

### ทดสอบการใช้งาน

เริ่มจากสร้าง working directory ชื่อ git-demo และสร้างไฟล์ greeting.txt ไว้ข้างใน

$ mkdir git-demo

$ cd git-demo

$ echo "me: Hello Git" > greeting.txt

สร้าง Git repository (.git/) ด้วยคำสั่ง init

$ git init

ตรวจสอบสถานะไฟล์ในโปรเจค git-demo จะพบข้อความเตือนสีแดงใต้ untracked files ซึ่งหมายความว่า Git มองเห็นว่ามีไฟล์อยู่ แต่ไฟล์นี้ยังไม่ได้เริ่ม tracking การเปลี่ยนแปลงของไฟล์นี้เลย

$ git status

วิธีการเริ่มต้น tracking ไฟล์ greeting.txt นั้น คือต้องเพิ่มไฟล์นี้ลงไฟใน staging area ก่อน โดยใช้คำสั่ง git add filename1 filename2 filename3 แต่ถ้าต้องการเพิ่มทุกไฟล์ที่มีการแก้ไข/เปลี่ยนแปลงที่อยู่ใน working directory ใช้ git add .

$ git add greeting.txt

เมื่อลองตรวจสอบสถานะใหม่อีกครั้ง greeting.txt จะเปลี่ยนเป็นสีเขียวแล้ว พร้อมที่จะ commit

$ git status

คราวนี้ให้แก้ไขไฟล์ greeting.txt โดยเพิ่มข้อความบรรทัดใหม่ว่า “git: Hey, Good to see you.” และเมื่อบันทึกเสร็จให้ลองใช้คำสั่ง git diff filename เพื่อดูความแตกต่างของไฟล์ระหว่าง working directory กับ staging area เสร็จแล้วให้เอาไฟล์นี้เอา staging area ด้วย

$ echo "git: Hey, Good to see you." >> greeting.txt

$ git diff greeting.txt

$ git add greeting.txt

ขั้นตอนถัดไปได้แก่การ ‘commit’ ไฟล์ที่อยู่ใน staging area เข้าไปเก็บใน Git repository แบบถาวร

$ git commit -m "My first commit"

ถ้าจะดูประวัติการ ‘commit’ ใน repository ทำได้โดยใช้คำสั่ง git log

$ git log

เนื่องจากรูปแบบการแสดงผลปกติของ git log นั้นดูยากไปหน่อยถ้ามีประวัติการ ‘commit’ หลาย ๆ ครั้ง ส่วนตัวชอบใช้วิธีให้แสดงประวัติแต่ละครั้งในบรรทัดเดียว ตามนี้

$ git log --oneline --graph --decorate --all

ถ้าความขี้เกียจพิมพ์คำสั่งยาว ๆ เราสามารถใช้ Git alias สร้างคำสั่งของเราขึ้นมาเองได้ โดยจะเอาการแสดง log ข้างบน มาสร้างเป็นคำสั่ง git hist

$ git config --global alias.hist "log --oneline --graph --decorate --all"

ลองทดสอบคำสั่งใหม่ดู

$ git hist

\* 6dd2383 (HEAD -> master) My first commit

สรุปคำสั่ง

* git init ไว้สร้าง Git repository
* git status ตรวจสอบสถานะไฟล์ของ working directory และ staging area
* git add เพิ่มไฟล์ the working directory เข้าสู่ staging area
* git diff แสดงความแตกต่างของไฟล์ระหว่าง working directory กับ staging area
* git commitเก็บประวัติการแก้ไขแบบถาวรจาก staging area ไว้ใน repository
* git log แสดงรายการที่ commit มาทั้งหมด

### การกู้คืนไฟล์

บางครั้งในการทำงาน เช่น การแก้ซอร์สโค้ด เราอาจจะแก้ซอร์สโค้ดนั้นจนไม่สามารถรันได้อีก และจำไม่ได้ว่าเราแก้ไขอะไรไปแล้วบ้าง ซึ่ง Git นั้นมีฟีเจอร์ง่าย ๆ ในการกู้คืนไฟล์ เพื่อแก้ไขความผิดพลาดที่เกิดขึ้นใน Git project ของเรา

สมมุติว่าเผลอลบข้อความในไฟล์ greeting.txt ออกทั้งหมด และบันทึกทับไปแล้ว

$ echo "" > greeting.txt

ลองพิมพ์คำสั่ง git show HEAD ซึ่งจะแสดงข้อมูลทุกอย่างจากคำสั่ง git log จาก HEAD commit และต่อท้ายด้วยสิ่งที่แก้ไขเพิ่มเข้ามาจากการ ‘commit’ ครั้งนั้น

HEAD: คือ pointer ที่เก็บทุกๆ commit ของเรา โดยปกติแล้ว HEAD จะชี้ไปที่ commit ล่าสุด reference ของ HEAD จะอยู่ในรูปแบบ SHA

ถ้าเราต้องการย้อนกลับแก้ไขไฟล์ greeting.txt กลับไปยังการ HEAD : commit ล่าสุด ให้ใช้คำสั่ง git checkout HEAD filename ก็จะได้ไฟล์กลับมาเหมือนเดิม

$ git checkout HEAD greeting.txt

$ cat greeting.txt

me: Hello Git

git: Hey, Good to see you.

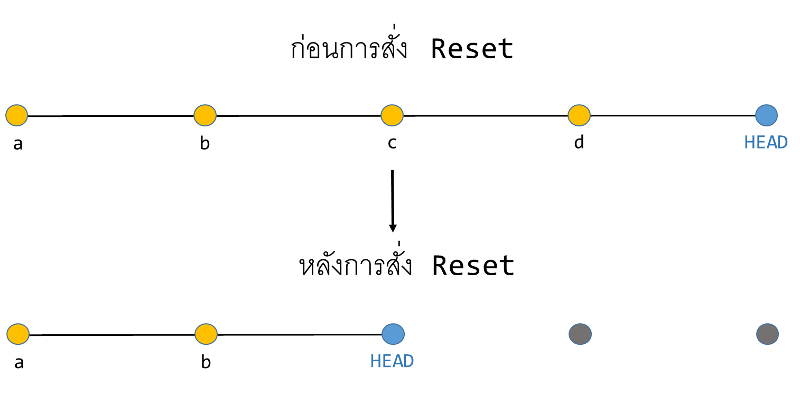
แต่ถ้าเราเผลอไปใช้ git add จะทำให้ไฟล์ถูกเปลี่ยนสถานะเป็น staged คือพร้อมที่จะ commit แต่เราไม่ต้องการ commit ไฟล์นั้นก่อน ดังนั้นต้องทำให้ไฟล์นั้นกลับมาเป็นสถานะ unstaged เสียก่อน โดยใช้คำสั่ง git reset HEAD filename แต่ไฟล์ไม่ได้ย้อนกลับการแก้ไขให้เหมือนการ checkout

$ git reset HEAD greeting.txt

Unstaged changes after reset:

M greeting.txt

กรณีที่เราต้องการย้อนกลับไปยัง commit ใด ๆ ใช้คำสั่ง git reset SHA (SHA ใช้แค่ 7 ตัวอักษรแรก) ซึ่งเมื่อย้อนกลับไปแล้ว HEAD จะถูกย้ายกลับไปยัง commit นั้นๆ ด้วย ดูได้จากรูปข้างล่าง และตรง commit สีเทานั้น ก็จะหายไปจาก Git project ด้วย



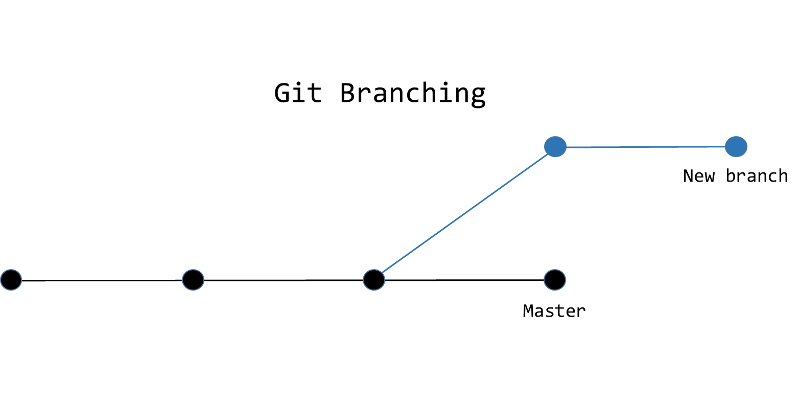
รูปที่ . การกู้คืนไฟล์

สรุปคำสั่ง

* git checkout HEAD filename: ยกเลิกการแก้ไขใน working directory
* git reset HEAD filename: ยกเลิกสถานะกลับเป็น Unstages ใน staging area
* git reset SHA: ยกเลิกการแก้ไขกลับไปยัง commit ก่อนหน้าตามที่เลือก

### Branching and Merging

โดยปกติแล้วการจัดการไฟล์ทั้งหมดจะถูกทำอยู่บน Git branch ที่ชื่อว่า master แต่ถ้าเราต้องการจะแยกไฟล์ทั้งหมดออกไปเป็นอีกเวอร์ชันนึง ก็สามารถทำได้โดยการสร้าง branch ใหม่ขึ้น ซึ่งการแก้ไขอะไรก็ตาม จะไม่กระทบเวอร์ชันเดิมใน branch master เมื่อแก้ไขเสร็จแล้ว และต้องการจะเอากลับมารวมกันก็สามารถทำได้



รูปที่ . Git Branch

New branch จะมีไฟล์ทั้งหมดที่ commit มาจาก Master และมีการ commit ใน New branch จะไม่ส่งผลกับ Master

วิธีการดูว่าตอนนี้เรามี branch อะไรบ้าง และจะมี \* (asterisk) อยู่หน้าชื่อ branch ที่กำลังทำงานอยู่ ทำได้โดย

$ git branch

\* master

สร้าง branch ใหม่ ชื่อ goodbye โดยใช้คำสั่ง git branch new\_branch\_name

$ git branch goodbye

สลับไปใช้งาน branch ใหม่ โดย git checkout branch\_name

$ git checkout goodbye

M greeting.txt

Switched to branch 'goodbye'

$ git branch

\* goodbye

master

เราสามารถสั่ง git checkout -b branch\_name เพื่อสร้าง branch ใหม่ พร้อมเปลี่ยนไปใช้งานได้เลย

ทดสอบสร้างไฟล์ใหม่ และ commit ดู

$ echo "me: See ya later." > goodbye.txt

$ git add goodbye.txt

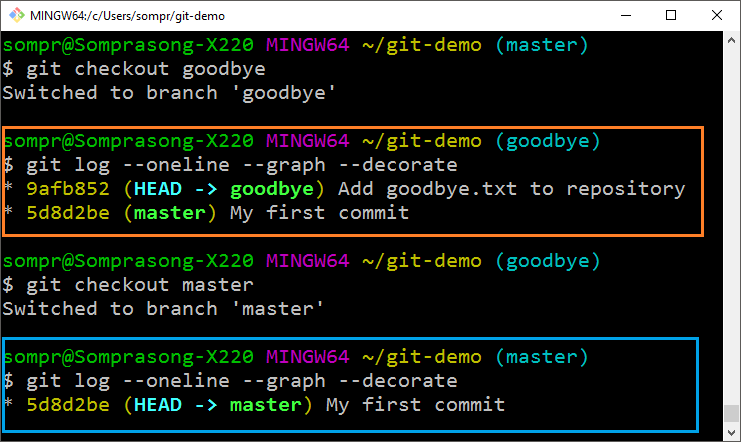
$ git commit -m "Add goodbye.txt to repository"

[goodbye f215b31] Add goodbye.txt to repository

1 file changed, 1 insertion(+)

create mode 100644 goodbye.txt

ลองดูจาก log เทียบกันระหว่าง 2 branch ดู จะพบว่า goodbye.txt จะมีอยู่แต่ใน branch ชื่อ goodbye เท่านั้น ส่วนใน master ไม่มีอะไรเปลี่ยนแปลง



รูปที่ . ผลที่ได้จากการทดสอบ

แต่ถ้าต้องการรวม branch goodbye เข้าไปใน master ให้สลับกลับมาอยู่ master ก่อน จากนั้นใช้คำสั่ง git merge branch\_name

$ git checkout master

Switched to branch 'master'

$ git merge goodbye

Updating 5d8d2be..9afb852

Fast-forward

goodbye.txt | 0

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 goodbye.txt

$ git log --oneline --graph --decorate

\* 9afb852 (HEAD -> master, goodbye) Add goodbye.txt to repository

\* 5d8d2be My first commit

เมื่อลองดูจากลอง commit 9afb852 ของ branch goodbye จะถูกรวมเข้ากับ master เรียบร้อยแล้ว แต่ถ้ามีการแก้ไขไฟล์จาก master และได้ทำการ commit ไปก่อนที่จะทำการ merge และใน goodbye branch ก็มีการแก้ไขไฟล์เดียวกันด้วย เมื่อสั่ง merge Git จะไม่ทราบว่าเราจะเอาเวอร์ชันไหนกันแน่ สิ่งนี้เรียกว่า Merge Conflict

# add new line with "Bye from master version"

$ npp goodbye.txt

$ git add .

$ git commit -m "Add respone from master branch"

[master f93dc63] Add respone from master branch

1 file changed, 1 insertion(+)

$ git checkout goodbye

Switched to branch 'goodbye'

# add new line with "Bye from goodbye version"

$ npp goodbye.txt

$ git add .

$ git commit -m "Add response from goodbye branch"

[goodbye f44d780] Add response from goodbye branch

1 file changed, 1 insertion(+)

$ git checkout master

Switched to branch 'master'

$ git merge goodbye

Auto-merging goodbye.txt

CONFLICT (content): Merge conflict in goodbye.txt

Automatic merge failed; fix conflicts and then commit the result.

เมื่อลองเปิดไฟล์ดูข้างใน notepad++ goodbye.txt จะข้อความแปลก ๆ แบบนี้

me: See ya later

<<<<<<< HEAD

git: Bye from master version

=======

git: Bye from goodbye version

>>>>>>> goodbye

คือ Git จะไม่รู้ว่าเราต้องการเก็บข้อมูลชุดไหนไว้กันแน่ โดยจะมี Git’s special markings บอกเอาไว้ว่าข้อความของ master branch จะอยู่ระหว่าง <<<<<<< HEAD กับ ======= ส่วนของ goodbye branch อยู่ระหว่าง ======= กับ >>>>>> ซึ่งเราต้องแก้ไขเองว่าจะเอาข้อความไหน และต้องลบ Git’s special markings ออกด้วย

เมื่อแก้ไข conflict เสร็จแล้วก็สั่ง add เข้า staging area และ commit เก็บไว้ สุดท้ายเมื่อเรา merge branch ที่เราแยกออกไปเสร็จแล้ว หรือไม่ต้องการ branch นั้นๆ แล้ว ก็สามารถลบทิ้งได้ โดยใช้คำสั่ง git branch -d branch\_name

$ git branch -d goodbye

Deleted branch goodbye (was f44d780).

สรุปคำสั่ง

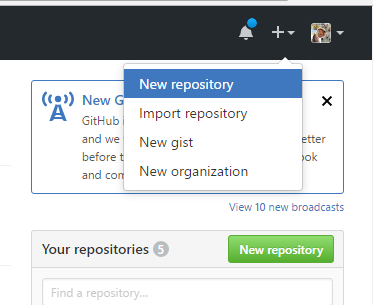
* git branch: ใช้แสดงรายชื่อ branch ทั้งหมด
* git branch branch\_name: สร้าง branch ใหม่
* git checkout branch\_name: สลับไปใช้งาน branch ที่ระบุ
* git merge branch\_name: ใช้รวมไฟล์จาก branch ที่ระบุ มายัง branch ปัจจุบัน
* git branch -d branch\_name: ลบ branch ที่ระบุ

### Git Remote Repository

จากที่ศึกษาวิธีการใช้ Git มาจากข้างต้นนั้นจะเป็นการใช้งานแบบ single user บน local repository ที่อยู่บนเครื่องใครเครื่องมัน แต่ Git นั้นจัดเป็น collaboration tool ตัวนึงที่ทำให้เราทำงานร่วมกับผู้อื่นในโปรเจคได้ง่ายขึ้น โดยผ่าน remote repository เช่น GitHub หรือ Bitbucket ซึ่งจะเป็นตัวกลางในการทำ pulling หรือ pushing งานในโปรเจคที่ทำร่วมกัน

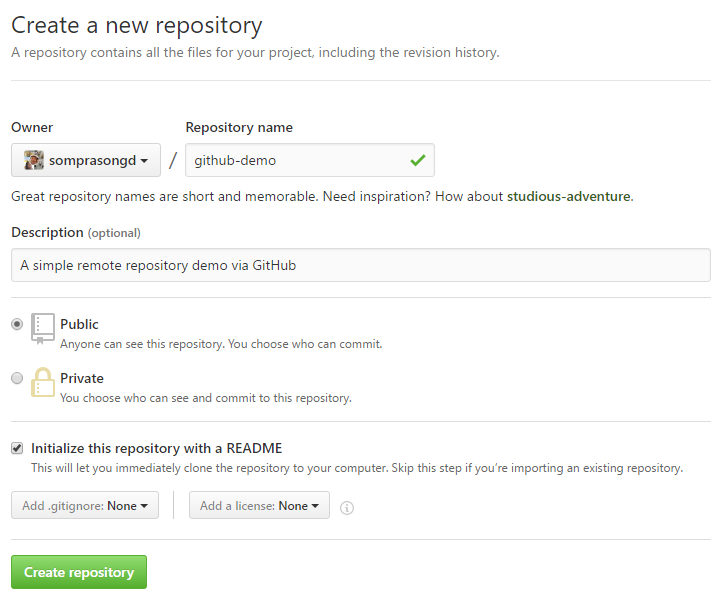
ตัวอย่างการสร้าง remote repository อยู่บน GitHub ชื่อ github-demo

ล็อกอินเข้า GitHub เลือกเครื่องหมาย + ข้างๆ รูปโปรไฟล์ แล้วเลือกเมนู New repository



รูปที่ . สร้าง remote repository บน GitHub

ตั้งชื่อ Repository name และเลือก “ Initialize this repository with a README” เพื่อสร้างไฟล์ README.md กดปุ่ม Create repository



รูปที่ . ตั้งชื่อ Repository name

เมื่อได้ remote repository มาแล้ว ก็ให้ ‘clone’ มาไว้ที่เครื่องของเรา โดยใช้คำสั่ง git clone remote\_location clone\_name ซึ่ง remote\_location คือที่อยู่ของ remote repository และ clone\_name คือชื่อไดเรกทอรีที่จะให้บันทึก แต่ถ้าไม่ระบุ Git จะสร้างไดเรกทอรีชื่อเดียวกับชื่อของ repository ให้อัตโนมัติ

$ git clone git@github.com:somprasongd/github-demo.git

Cloning into 'github-demo'...

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

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

$ cd github-demo

$ ls -a

./ ../ .git/ README.md

สิ่งที่ Git ทำอยู่เบื้องหลังเมื่อเรา clone โปรเจคมา คือ Git จะสร้าง remote address ขึ้นมาชื่อ origin เพื่อความสะดวกในการอ้างอิงถึง remote repository โดยใช้เพียงแค่ชื่อ ถ้าต้องการจะดูว่าใน Git project ของเรามีรายการ remotes อะไรบ้าง ใช้คำสั่ง git remote -v จากในไดเรกทอรีของ Git project

$ git remote -v

origin git@github.com:somprasongd/github-demo.git (fetch)

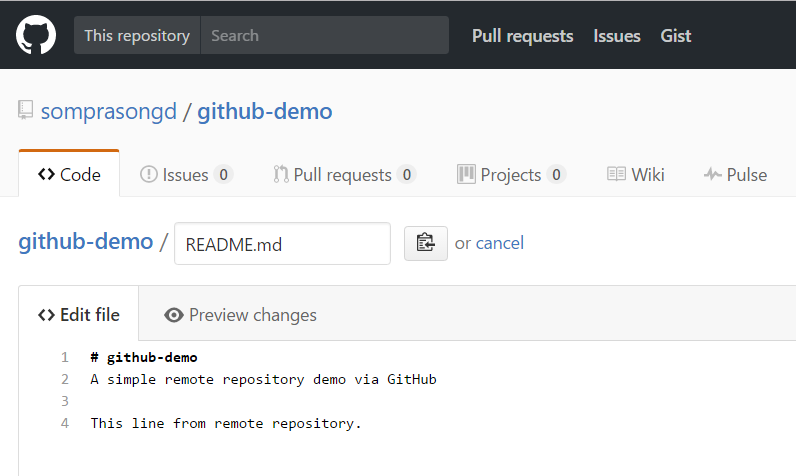
origin git@github.com:somprasongd/github-demo.git (push)

fetch คือ remote address ที่จะให้ไปดึงจากฝั่ง remote มายังฝั่ง local

push คือ remote address ที่จะให้เอาจากฝั่ง local ไปรวมที่ฝั่ง remote ที่ไหน

หลังจากที่เรา clone โปรเจคมานานแล้ว และไม่ทราบว่าทางฝั่ง remote นั้นมีการเปลี่ยนแปลงอะไรเกิดขึ้นแล้วบ้าง เราสามารถตรวจสอบได้โดยใช้คำสั่ง git fetch

แก้ไขไฟล์ README.md ที่ GitHub โดยเพิ่มข้อความในบรรทัดใหม่ว่า “This line from remote repository.”



รูปที่ .

ดึงข้อมูลจากฝั่ง remote ว่ามีการเปลี่ยนแปลงอะไรเกิดขึ้นบ้าง

$ git fetch

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

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

From github.com:somprasongd/github-demo

4fae075..9595d50 master -> origin/master

ตรวจสอบด้วย git status จะได้ข้อความดังนี้

$ git status

On branch master

Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

(use "git pull" to update your local branch)

nothing to commit, working tree clean

หมายความว่าบน local ( branch master)นั้นตามหลังบน remote ( branch origin/master) อยู่ 1 commit ซึ่งเราสามารถรวมโค้ดจาก remote มายัง local ได้ด้วยวิธีเดียวกันกับการรวมโค้ดจาก branch อื่น โดยใช้คำสั่ง git merge โดยระบุชื่อ branch เป็น “origin/master”

$ git merge origin/master

Updating 4fae075..9595d50

Fast-forward

README.md | 2 ++

1 file changed, 2 insertions(+)

$ cat README.md

# github-demo

A simple remote repository demo via GitHub

This line from remote repository.

git pull คือการรวมโค้ดจาก remote มายัง local เลยโดยไม่สนใจว่าระหว่าง remote กับ local ต่างกันอย่างไรบ้าง ซึ่งจริงแล้ว git pull คือการทำ git merge แล้วต่อด้วย git pull ให้แบบอัตโนมัติ

ต่อมาเมื่อเรามีการแก้ไขจากฝั่ง local และต้องการจะส่งไปเก็บไว้ที่ฝั่ง remote ด้วย ทำได้โดยใช้คำสั่ง git push remote\_alias\_name branch\_name เช่น git push -u origin master

-u : เอาไว้จำ parameter origin master ครั้งต่อๆ ไปก็พิมพ์แค่ git push

origin : คือ ชื่อ alias ของ remote (github)

master : คือ ชื่อ branch ที่เราต้องการ push ขึ้นไป

$ echo -e "\nThis line from local repository" >> README.md

$ git add .

$ git commit -m "Update README.md from local"

[master 395d3f6] Update README.md from local

1 file changed, 2 insertions(+)

$ git push -u origin master

Counting objects: 3, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 320 bytes | 0 bytes/s, done.

Total 3 (delta 1), reused 0 (delta 0)

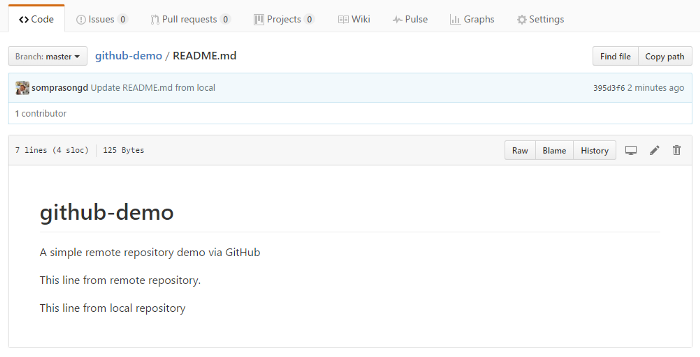
remote: Resolving deltas: 100% (1/1), completed with 1 local objects.

To github.com:somprasongd/github-demo.git

9595d50..395d3f6 master -> master

Branch master set up to track remote branch master from origin.

เมื่อลองไปดูจาก GitHub จะเห็นว่าไฟล์ README.md จะถูกอัพเดทแล้ว



รูปที่ .

สรุปคำสั่ง

* git clone: สร้าง local repository จาก remote repository
* git remote -v: แสดงรายการ remote address ทั้งหมด
* git fetch: ดึงข้อมูลทั้งหมดจากฝั่ง remote มายัง local
* git merge origin/master: สั่งรวมไฟล์จาก origin/master มายัง local branch
* git pull: สั่งรวมไฟล์จาก ฝั่ง remote มายัง local (git fetch + git merge)
* git push origin <branch\_name>: ส่งข้อมูลจาก local branch ไปรวมกับ originremote.

### Stashing

คือ การบันทึกโปรเจคของเรา และซ่อนเอาไว้ เมื่อจะใช้ก็สั่ง restore กลับมาได้ตลอดเวลา เช่น กรณีที่เรากำลังแก้ไขไฟล์อยู่ และจำเป็นต้องสั่ง commit โปรเจคนี้ แต่ไฟล์นี้ยังแก้ไขไม่เสร็จเลยไม่ต้องการ commit ในครั้งนี้ด้วย ก็ใช้ stash เพื่อซ่อนไฟล์นี้เอาไว้ก่อน เมื่อได้ commit เสร็จแล้ว ก็สั่ง restore ไฟล์ที่ซ้อนไว้กลับมา แก้ไขจนเสร็จ แล้วค่อย commit ใหม่

git stash: สั่งซ่อนไฟล์

git stash list: แสดงรายการที่ซ้อนไว้ โดยรายการล่าสุดจะอยู่ที่ stash@{0}

git stash pop: สั่ง restore รายการล่าสุดกลับมา

# edit README.md file

$ echo -e "\nGit stashing demo line." >> README.md

# show git status, see README.md modified.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

no changes added to commit (use "git add" and/or "git commit -a")

# use git stash to hide modified README.md

$ git stash

Saved working directory and index state WIP on master: 395d3f6 Update README.md from local

HEAD is now at 395d3f6 Update README.md from local

# show list of stash

$ git stash list

stash@{0}: WIP on master: 395d3f6 Update README.md from local

# show git status, see nothing to commit.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean

# create new text file

$ touch somefile.txt

# add text file to staging area

$ git add somefile.txt

# commit new text file, not include modified README.md

$ git commit -m "Add some text file"

[master 6fe9c68] Add some text file

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 somefile.txt

# show git status, see nothing to commit.

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

nothing to commit, working tree clean

# restore README.md and remove it from stashing

$ git stash pop

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

no changes added to commit (use "git add" and/or "git commit -a")

Dropped refs/stash@{0} (5c131b45b0be6e0af3b06b47282149d7dd904678)

# show list of stash again, see nothing

$ git stash list

# view data in README.md

$ cat README.md

# github-demo

A simple remote repository demo via GitHub

This line from remote repository.

This line from local repository

Git stashing demo line.

# add modified README.md to staging area

$ git add README.md

# commit README.md

$ git commit -m "Update README.me with stash demo"

[master b1e16ac] Update README.me with stash demo

1 file changed, 2 insertions(+)

### Tagging

ก่อนที่จะทำการ release ระบบออกไปนั้น แนะนำให้ทำการสร้าง tag หรือ ป้ายชื่อ เพื่อบอกให้รู้ว่าเรากำลังทำอะไร อยู่ตรงไหน

git tag tag\_name SHA: สร้าง tag ขึ้นมาใน commit ที่ระบุ ถ้าไม่ระบุจะเป็น HEAD commit

git tag --list: แสดงรายการชื่อ tag ทั้งหมด

git tag -d tag\_name: สั่งลบจากชื่อ tag ที่ระบุ

git show tag\_name: แสดงรายละเอียด commit จากชื่อ tag ที่ระบุ

git tag -a tag\_name -m "message": สร้าง tag แบบระบุข้อความบอกว่า tag นี้คืออะไร ซึ่งเมื่อใส่ git show tag\_name จะแสดงข้อความจาก -m ขึ้นมาให้ด้วย

# create new tag

$ git tag my\_tag

# show git log, see my\_tag at HEAD commit

$ git hist

\* b1e16ac (HEAD -> master, tag: my\_tag, origin/master, origin/HEAD) Update README.me with stash demo

\* 6fe9c68 Add some text file

\* 395d3f6 Update README.md from local

\* 9595d50 Update README.md

\* 4fae075 Initial commit

# show commit detail via tag name

$ git show my\_tag

commit b1e16acec97bc597ecca1cefe3d227c7664541a6

Author: Somprasong Damyos <somprasong@git.com>

Date: Fri Feb 17 16:59:36 2017 +0700

Update README.me wiht stash demo

diff --git a/README.md b/README.md

index 6cc2788..442636f 100644

--- a/README.md

+++ b/README.md

@@ -4,3 +4,5 @@ A simple remote repository demo via GitHub

This line from remote repository.

This line from local repository

+

+git stash demo line.

# show all tag name

$ git tag --list

my\_tag

# delete tag

$ git tag -d my\_tag

Deleted tag 'my\_tag' (was b1e16ac)

# show all tag name, see nothing

$ git tag --list

# create new tag with information use -a flag

$ git tag -a v-1.0 -m "Release version 1.0"

# show commit detail via tag name and include tag information

$ git show v-1.0

tag v-1.0

Tagger: Somprasong Damyos <somprasong@git.com>

Date: Fri Feb 17 17:58:50 2017 +0700

Release version 1.0

commit b1e16acec97bc597ecca1cefe3d227c7664541a6

Author: Somprasong Damyos <somprasong@git.com>

Date: Fri Feb 17 16:59:36 2017 +0700

Update README.me wiht stash demo

diff --git a/README.md b/README.md

index 6cc2788..442636f 100644

--- a/README.md

+++ b/README.md

@@ -4,3 +4,5 @@ A simple remote repository demo via GitHub

This line from remote repository.

This line from local repository

+

+git stash demo line.

### Remove

git rm filename: สั่งลบไฟล์ และสั่งให้ Git ทำการ untracked ไฟล์ด้วย

git rm --cached filename: สั่งลบไฟล์ออกจาก Git repository เฉยๆ ไม่ได้ลบไฟล์จริงใน working directory ออกด้วย

### คำสั่งเกี่ยวกับ Git ที่ใช้บ่อย ๆ

git clone

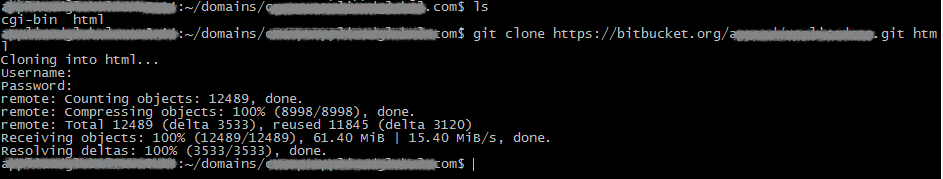
git clone <repo> <directory>

ทำการคัดลอก Repository (repo) ที่อยู่บน Remote Server มาวางไว้ที่โฟลเดอร์ในเครื่อง Local Host (directory)

การ Clone git ขึ้น Server ผ่าน SSH ก่อนอื่นให้ทำการ ล็อกอินเข้า SSH ก่อน

ssh username@example.com

แล้วตามด้วยโค้ดตามฟอร์แมตด้านล่าง Link ของ Git ที่จะทำการ Clone หากต้องการให้คนภายในทีมช่วยกัน Submit ให้เอา Username ที่อยู่ใน Link ออก (กรณีโปรเจคนี้มีการทำงานหลายคน)



Clone ลงเครื่องเรา (localhost) ให้ cd เข้าไปที่โฟลเดอร์ที่เราต้องการแล้วตามด้วยคำสั่งด้านล่าง

$ git clone URL ของผู้ให้บริการ Git

อัปโหลดไฟล์จาก git ขึ้น FTP ด้วย git-ftp สำหรับ Hosting ที่ไม่มี SSH หรือ git support ก่อนอื่นให้ติดตั้ง git-ftp ตาม URL นี้ https://github.com/git-ftp/git-ftp

หากเอาขึ้น FTP ครั้งแรกให้ใช้

$ git ftp init -u 'username' -P ftp://host.example.com/public\_html

ครั้งต่อ ๆ ไปใช้

$ git ftp push -u 'username' -P ftp://host.example.com/public\_html

ตัวอย่าง



รูปที่ . การอัปโหลดไฟล์จาก git ขึ้น FTP

git status ตรวจสอบสถานะของ Git อาจจะตรวจสอบว่า มีโค้ดที่มีการอัพเดท ที่ต้อง Pull หรือ Push หรือเปล่า

$ git status

git add เลือก file สำหรับ Commit

$ git add <file>

เลือก directory ที่มีการเปลี่ยนแปลง

git add <directory>

เลือกไฟล์ทั้งหมดที่มีการเปลี่ยนแปลง

git add .

git commit สำหรับ Commit ไฟล์ หลังจากใช้ git add เลือกไฟล์สำหรับ commit แล้วให้ใช้คำสั่ง git commit เพื่อที่จะทำการ commit ไฟล์ โดย -m “<message>” คือ ใส่คำอธิบายในการ commit ครั้งนี้

git commit -m "<message>"

git push หลังจาก Commit เสร็จก็ทำการ push ไฟล์ขึ้น git หลังจาก commit เสร็จ ก็ทำการ push ไฟล์ขึ้น git -> ใส่รหัสผ่าน -> ไฟล์ก็จะถูกอัพเดทขึ้น git

git push

git pull ดึงลงมาอัพเดทในเครื่องเรา หากมีไฟล์ที่อยู่บน git อัพเดทกว่าในเครื่องเรา (มีคนในทีมอัพเดท) ก็ใช้คำสั่ง pull ดึงลงมาอัพเดทในเครื่องเรา

git pull

git log ดูประวัติการ Commit กด q เพื่อออกจากการดู log

git log