A Workshop on Software Development Life Cycle:From Git to CI and Containerization

short line

Assist. Prof. Dr. Warodom Werapun

Mr. Kullawat Chaowanawatee

Mr. Sakarin Kammanee

May 19, 2017

# Table of Contents

[**Table of Contents**](#_u7e6yylqtmef) **1**

[**Git**](#_vh4w002d6tn6) **3**

[Why Git ?](#_mk1k48bo9vmf) 3

[Understanding Git](#_e4gp78v408vb) 3

[Working Directory, Repository and The Thing Between](#_izqr0436gz2w) 3

[Untracked, Unmodified, Modified and Staged](#_3ivk1wmkmz73) 4

[Ignoring List](#_lr8tyur3azfr) 4

[Best Practices](#_hk909l6sza5t) 5

[Preparing Git Environment](#_kp21a4t9bw6k) 7

[Generating SSH Key Pairs and Setting Keys on GitHub](#_lane3boep527) 7

[Installing Git](#_sc1sifo6z8b2) 8

[Git, The First Time](#_jzbbjby5gpci) 8

[Repository Management](#_rgzf4l9ehqfn) 9

[Repository Creation](#_q7psq8syvj76) 9

[Staging, Committing, and Pushing](#_985l3lszh2r2) 11

[Fetching, Pulling, and Traversal](#_mplcq7khemaj) 12

[Branching and Merging](#_22r1lyuynmqf) 12

[Merge Conflict Resolution](#_3chua4bk713a) 13

[Tagging](#_cu64t79x3k24) 14

[Issues and Milestones](#_3cpol47eraof) 14

[Advanced Topics](#_35yztlk71rxo) 17

[Stashing](#_b6vtww2nrfnq) 17

[Bugs Searching](#_z3dzmir1nae1) 18

[Rebasing and Cherry Picking](#_24wi4m5cv9kq) 20

[Git Flow Model](#_uehewjony009) 22

[Scrum Board using Waffle.io (for GitHub)](#_ei72ov9duzxg) 23

[Git Cheatsheet](#_29s66hb4p0dx) 23

[Suggested Playground and References](#_4t2iy3iqnjm1) 23

[**Test Driven Development**](#_1dj80gf82y1f) **24**

[Why TDD?](#_92yuqluxaouc) 24

[Unit testing in PHP](#_grje8uf54t31) 24

[เริ่มต้นที่การทำ API test](#_p25qw14ozg5d) 24

[สร้าง controller FizzBuzz](#_c4k0il6qc7wr) 25

[ทดลอง run test](#_nkhxwhw6gpyg) 25

[Web-UI Testing](#_v7g4fhpirrul) 26

[เริ่มเขียน GUI](#_7xx5u7d2gvav) 26

[เขียน Web Test](#_b1gxf78uwihi) 27

[ทดสอบกรณีอื่น ๆ เพิ่มเติม](#_k98lsgdzfnaa) 28

[PHP Code Coverage](#_6562vjiavoif) 30

[Configuration](#_72p2zqvujqa1) 30

[Require Xdebug library](#_9m133kmwycko) 30

[ทดลอง run test](#_m96ft254nrcj) 31

[**Continuous Integration / Continuous Delivery (CI/CD)**](#_5cz9jp50serk) **33**

[Jenkins](#_82w2v2nqx6yz) 33

[Installation: Jenkins 2.39 on Debian 8.6](#_45f5nnii06xv) 33

[Jenkins GitHub Service Integration (CI)](#_6183iuiwwpl7) 34

[Jenkins permission และ Local user](#_bo9crgctqmno) 37

[Github Setting for Jenkins Hook URL](#_21x5583r3c1r) 37

[ทดสอบการทำงาน](#_6x2nlwevdy51) 38

[Jenkins User/Port Configuration](#_hnpqy76zy0jo) 40

[Jenkins ไม่ทำงาน เพราะ RAM หมด](#_gl4kha52gg) 40

[**Moby Containerize**](#_vw5i98yqlgy4) **41**

[Moby](#_m4vum7yc8nyx) 41

[Why Moby?](#_kiz9vejc2w3) 41

[Installation: Moby on Debian](#_79p9p8rvdxpq) 41

[Jenkins & Docker Configuration](#_b07yterqerh5) 41

[เพิ่ม User jenkins ไปยัง docker group](#_m1cicukoqrwn) 41

[เพิ่ม Credential ของ Docker Registry ใน Jenkins](#_giupk6fghswb) 41

[เตรียม Repository บน Docker Registry](#_8s3ucu2qboem) 42

[สร้าง Jenkins job](#_wa6np76fmg9p) 42

[ทดสอบการทำงาน](#_9r747tyusa4c) 42

# Git

## Why Git ?

(ยาวไปไม่อ่าน) การใช้ git ดีกว่าการไม่ใช้ git

Git เป็น version control system (VCS) สำหรับการติดตามการเปลี่ยนแปลงของไฟล์ และใช้งานไฟล์ร่วมกันหลายคน ปัจจุบันถูกใช้ในสายการพัฒนาซอฟต์แวร์ Git ถูกออกแบบให้เป็น VCS แบบกระจาย (distributed) จึงเน้นที่ความเร็ว ความถูกต้องของข้อมูล สนับสนุนการทำงานแบบกระจาย (ไม่รวมศูนย์)

Git เกิดขึ้นจาก Project Linux Kernel ของ kernel.org ซึ่งเดิมใช้ BitKeeper ซึ่งเป็น proprietary DVCS (Distributed VCS) ที่มี free use แต่ภายหลังถูกถอด free use ออก เนื่องจากมีการพบหลักฐานการ reverse engineer protocol ของ BitKeeper แล้วสร้างใหม่เป็น Mercurial (หรือรู้จักกันในอีกชื่อคือ hg ซึ่งเป็นสัญลักษณ์ทางเคมีของ Mercury) ด้วยเหตุนี้ ทีมพัฒนา Linux Kernel จึงถอดใจจาก BitKeeper และเป็นจุดเริ่มต้นการพัฒนา Git โดย Linus Torvalds

## Understanding Git

### Working Directory, Repository and The Thing Between

ระบบของ git ถูกแบ่งออกเป็น 3 ส่วนหลัก ๆ คือ Working Directory, Staging Area และ Repository เราสามารถเคลื่อนย้ายงานไปตามแต่ละส่วนได้ โดยมีรายละเอียดดังนี้



* Staging คือการนำงานใน Working Directory เข้าสู่ Staging Area
* Committing คือการนำไฟล์ใน Staging Area เข้าสู่ Repository \*
* Checkout คือการนำไฟล์ที่เคย commit ใน Repository กลับมายัง Working Directory \*

\* หมายเหตุ การ commit และ checkout ทำให้ HEAD pointer ของ Repository เปลี่ยนแปลง และการบังคับ checkout อาจทำให้ไฟล์ที่ยังไม่ได้ commit ใน working directory เสียหาย

### Untracked, Unmodified, Modified and Staged

ไฟล์ในระบบ Git จะมีสถานะ 4 สถานะ ได้แก่

1. Untracked ในสถานะนี้ repository ยังไม่มีไฟล์ดังกล่าวอยู่ ดังนั้นหากต้องการนำไฟล์เข้าสู่ repository จะต้องใช้คำสั่ง add
2. Unmodified คือสถานะที่ไฟล์มีข้อมูลเหมือนกับ repository ทุกประการ หากแก้ไขไฟล์ ไฟล์จะเปลี่ยนเป็นสถานะ Modified แต่หากลบไฟล์ (คำสั่ง rm) จะเปลี่ยนเป็นสถานะ Untracked
3. Modified คือสถานะที่ไฟล์ถูกแก้ไข จึงทำให้ข้อมูลไม่ตรงกับ repository หากต้องการจะนำไฟล์เข้าสู่ staging area จะต้องใช้คำสั่ง add สถานะจะเปลี่ยนเป็น Staged
4. Staged คือสถานะที่ไฟล์ถูกหมายว่าจะเปลี่ยนแปลงใน repository เมื่อ staging ครบทุกไฟล์แล้ว สามารถนำไฟล์ใหม่และไฟล์ที่แก้ไขเข้าสู่ repository ได้โดยใช้คำสั่ง commit สถานะจะเปลี่ยนเป็น Unmodified (หมายความว่าข้อมูลตรงกันกับ repository แล้ว)



### Ignoring List

Ignoring List คือรายการไฟล์ที่ถูกระบุใน .girignore ไฟล์ตามรายการดังกล่าวจะไม่ถูกพิจารณา เมื่อเรานำไฟล์เข้าสู่ repository โดยปกติเรามักใส่ object file, log file, swap file และอื่น ๆ ไว้ใน ignoring list เพื่อป้องกันไม่ให้นำไฟล์อื่นนอกจาก source code และ assets ที่จำเป็น เข้าสู่ repository

### Best Practices

1. **Commit บ่อย ๆ**

หาก commit น้อยครั้งเกินไปจะทำให้แต่ละ commit มีขนาดใหญ่ เกิด merge conflict ได้ง่ายและอาจเกิดจำนวนหลายแห่ง ดังนั้นการ commit บ่อยครั้ง จะช่วยให้ commit มีขนาดเล็กลง ลดการขัดแย้งใน code ขณะ merge และหากเกิด merge conflict ก็สามารถแก้ไขได้ง่าย

1. **Commit ให้สัมพันธ์กับงานที่ทำ**

งานแต่ละงานควรมีขนาดเล็ก จัดให้มีหนึ่ง commit ต่อหนึ่งงาน จะช่วยให้เกิดการ commit บ่อย ๆ และช่วยให้สามารถย้อนกลับได้ง่ายหากงานที่ทำเกิดปัญหา เช่น มี bug 2 ตัว ก็แบ่งเป็น 2 commits

1. **Test ก่อน Commit เสมอ**

ก่อน commit ต้องทดสอบก่อนเสมอว่า code จะทำงานได้ถูกต้องตามที่กำหนดใน test suite ทั้งนี้ก็เพราะหากยังมีคนอื่นในทีมที่ต้องใช้ git ร่วมกันกับเราแล้ว หากเขา fetch / pull commit ที่เรายังไม่ได้ test ไปจะทำให้เขา test งานของเขาลำบากด้วย

1. **อย่า Stage ไฟล์ที่ไม่จำเป็นต่อการ build**

คุณไม่ควร commit ไฟล์จำพวก object file, compiled bitcode / bytecode, cache, auxiliary build, log file, empty file หรือ executable file เป็นต้น เนื่องจากเป็นผลจากการ build หรือเกิดขณะ run

ไฟล์ที่ไม่จำเป็นเหล่านี้ ในบางครั้งอาจทำให้พลาดและติดมาใน commit ได้ ควรระบุรายการไฟล์ใน .gitignore เพื่อเพิกเฉยต่อไฟล์เหล่านั้น และจะช่วยลดความซับซ้อนในกระบวนการ staging ด้วย

ทั้งนี้ยกเว้นตัวอย่าง configuration file แต่ควรระวังไม่ให้มี API Key หรือข้อมูลสำคัญติดมา ในกรณีที่ไฟล์มีขนาดใหญ่เช่น video, audio หรือ dataset ควรเลี่ยงไปใช้ Git LFS

1. **อย่า Commit งานที่ยังทำไม่เสร็จ**

หากงานที่ทำยังไม่เสร็จดี หรือทำได้เพียงครึ่ง ๆ กลาง ๆ เช่น feature ใหม่ที่พัฒนาไม่ทันถึงไหน แต่ถึงเวลาเลิกงานเสียก่อน ก็เลยต้อง commit แบบนี้ไม่ดี ในกรณีที่ต้องเก็บพักงานไว้ให้ใช้ stash แทน

1. **เขียนอธิบาย Commit ดี ๆ**

คำอธิบาย commit หรือ commit message ที่ดีควรเป็นวลีสรุปสั้น ๆ ความยาว 2 - 15 คำ เมื่ออ่านแล้วต้องรู้ได้ทันทีว่า commit นี้ทำเพื่ออะไร และจุดที่แก้ไขต่างจากเดิมอย่างไรบ้าง แนะนำให้ใช้ present tense เพื่อให้อ่านลื่น (ทั้งนี้ เพราะข้อความอัตโนมัติที่เกิดจากการ merge ก็เป็น present tense) ตัวอย่างเช่น “revise knapsack code to DP algorithm instead of exhaustive trials” หรือ “add a contributor to README.md”

อย่างไรก็ตาม หากต้องการอธิบายยาว ๆ ว่าเกิดอะไรขึ้นบ้าง ให้เขียนเพิ่มใน CHANGELOG แทน

1. **ใช้ Branch**

การแยกการพัฒนาออกเป็น branch ควรจะทำตั้งแต่เนิ่น ๆ โดยแบ่งสายการพัฒนาต่าง ๆ เช่น release, feature, bug, hot fix หรือ beta ออกเป็น branch แยกกันออกไป เช่นนี้จะช่วยให้จัดการได้ง่าย แต่อย่างไรก็ตาม จะต้องตกลงแผนการทำงานหรือ workflow เพื่อใช้ร่วมกันในทีม

1. **ตกลงใช้ workflow ร่วมกัน**

การใช้ workflow ร่วมกันในทีมจะช่วยให้สายงาน branch และ commit ใน repository ดูเป็นระบบ จัดระเบียบได้ง่าย แบ่งทีมพัฒนาได้ แต่ทั้งทีมพัฒนาควรหารือและตกลงกันว่าจะใช้ workflow แบบใด เนื่องจากขึ้นอยู่กับความชอบของคนในทีม และลักษณะของ project นั้น ๆ ตามหลัก software engineer ด้วย

1. **Git ไม่ใช่เครื่องมือ backup**

Git เป็น version control system ดังนั้นในกรณีที่ต้องการ backup คุณควรใช้ git-annex, rsync หรือ cloud storage

## 

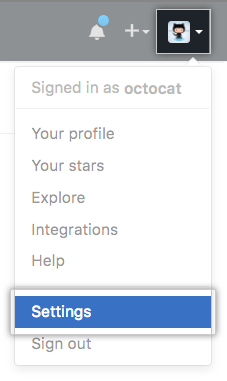
## Preparing Git Environment

### Generating SSH Key Pairs and Setting Keys on GitHub

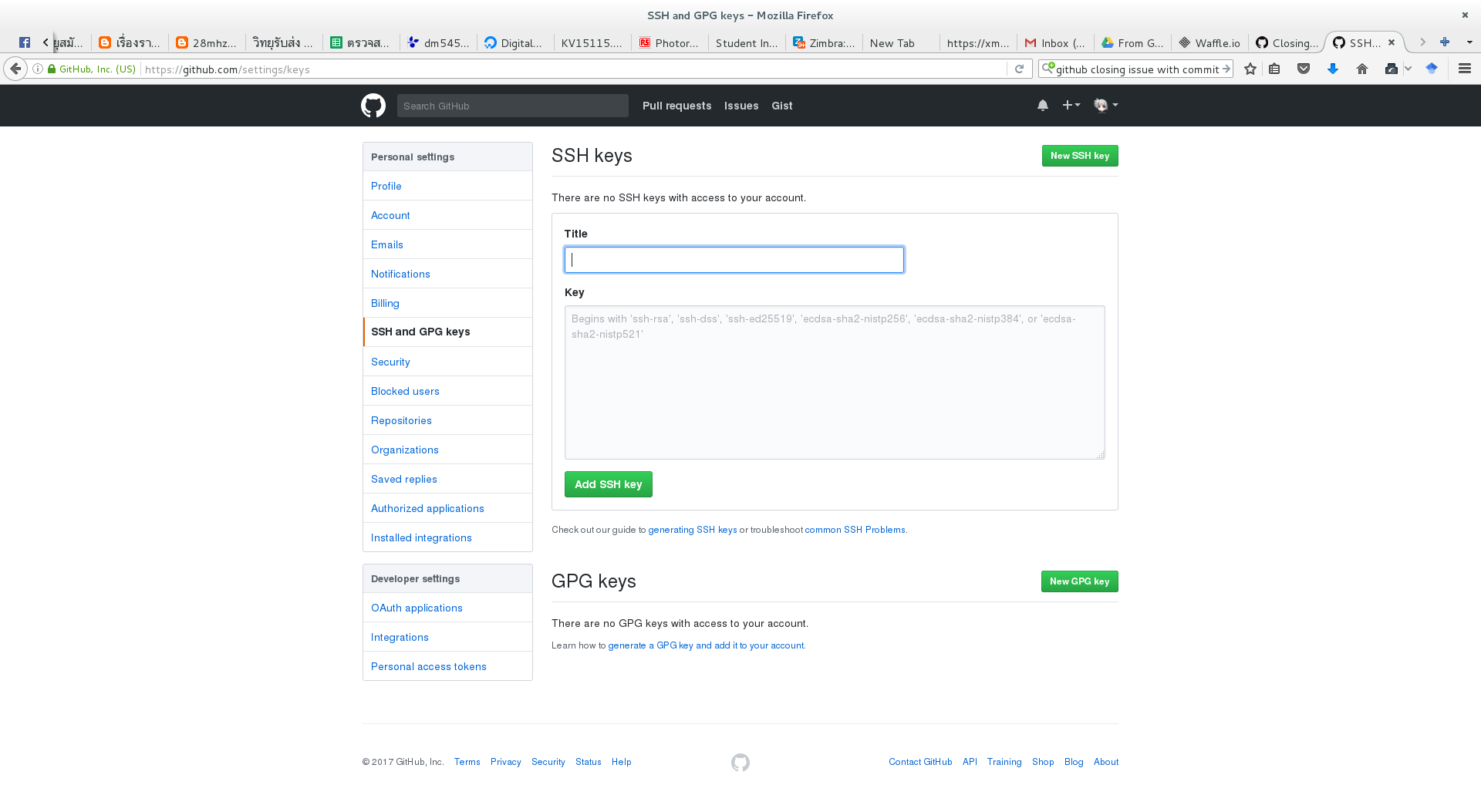
1. ใช้คำสั่ง ssh-keygen เพื่อสร้าง Key Pairs ขึ้นมา 1 คู่ แนะนำให้ใช้ RSA 4096 bits หรือสูงกว่า

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

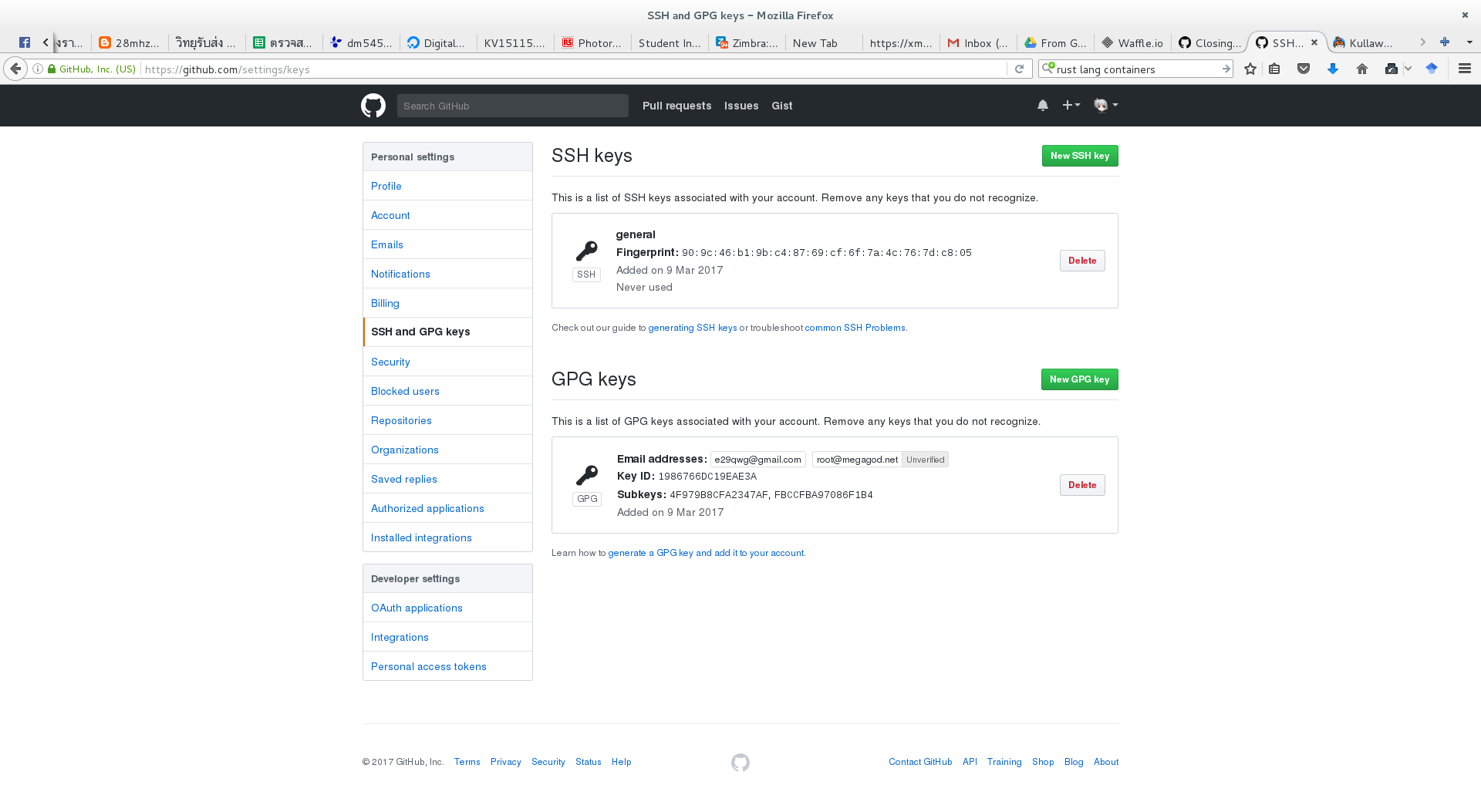
1. เข้าสู่ Account Settings โดยคลิกที่รูป profile แล้วเลือกเมนู Settings



1. เข้าสู่หมวด SSH and GPG keys แล้วกด New SSH key



1. ป้อนชื่อ Key ในช่อง Title และนำข้อมูลจากไฟล์ ~/.ssh/id\_rsa.pub มาใส่ในช่อง Key
2. กดปุ่ม Add SSH key จะได้ผลลัพธ์ดังภาพ



1. ทดสอบการเชื่อมต่อโดยใช้ SSH key โดยการเข้าสู่ระบบผ่าน ssh โดยใช้คำสั่ง

$ ssh -T git@github.com

1. สังเกตว่าจะต้องได้รับข้อความตอบกลับเป็น You've successfully authenticated, but GitHub does not provide shell access. หากไม่พบข้อความเช่นนี้แสดงว่ายังติดตั้ง SSH Key Pairs ไม่สำเร็จ

### Installing Git

**สำหรับ Linux ที่เป็น Debian derivatives (เช่น Debian, Devaun, Ubuntu, Mint, และอื่น ๆ)**

$ sudo apt install git

**สำหรับ Linux ที่เป็น Red Hat derivatives (เช่น RHEL, Fedora, Cent OS, และอื่น ๆ)**

$ sudo yum install git หรือ

$ sudo dnf install git สำหรับ Fedora 22 หรือสูงกว่า

**สำหรับ Linux สาย distro อื่น ๆ รวมถึง Windows และ Mac OS**

สามารถดาวน์โหลดได้จาก<https://git-scm.com/downloads>

### Git, The First Time

$ git config --global user.name "John Doe"

$ git config --global user.email "john@example.com"

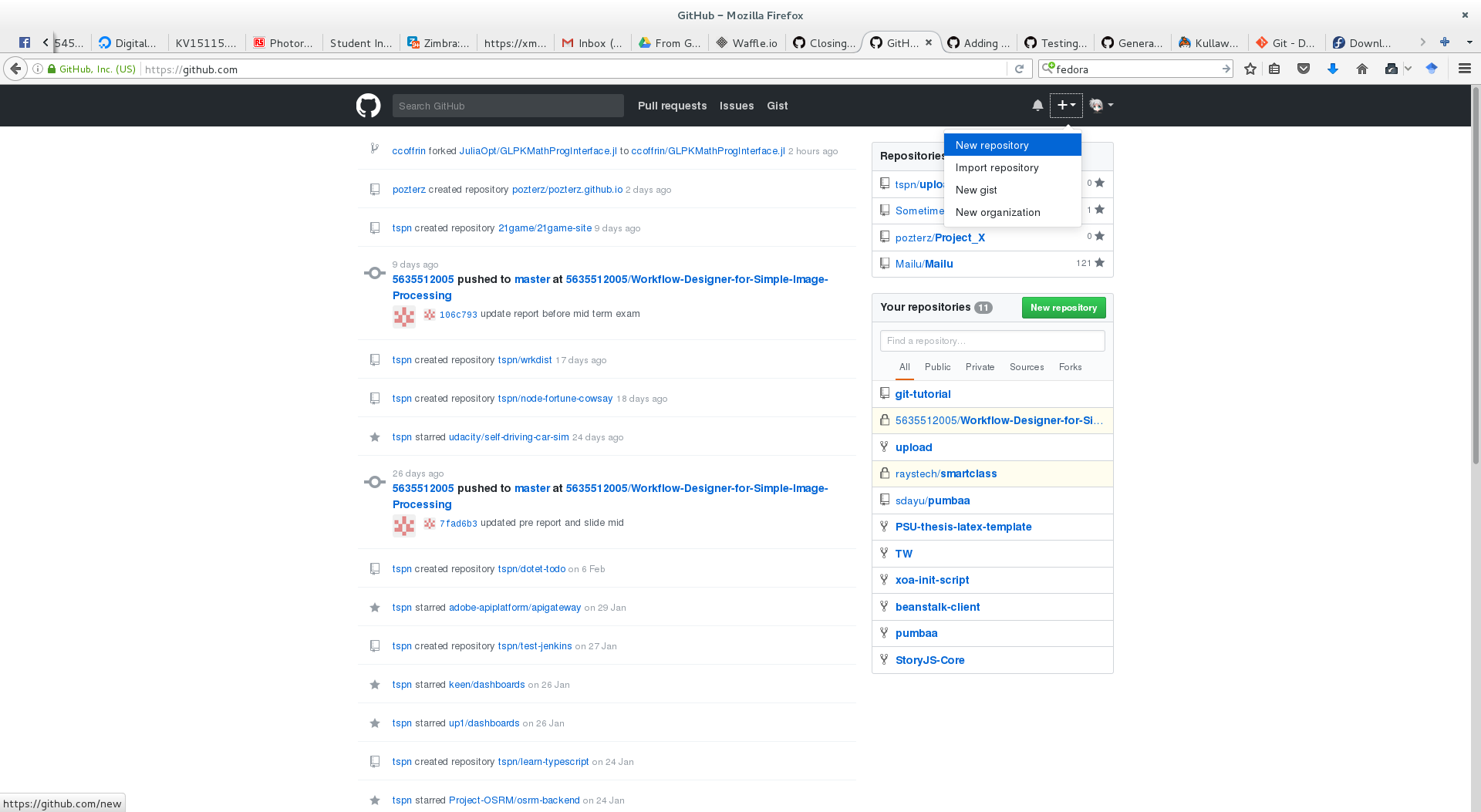
$ git config --global push.default simple

$ git config --global core.editor vim

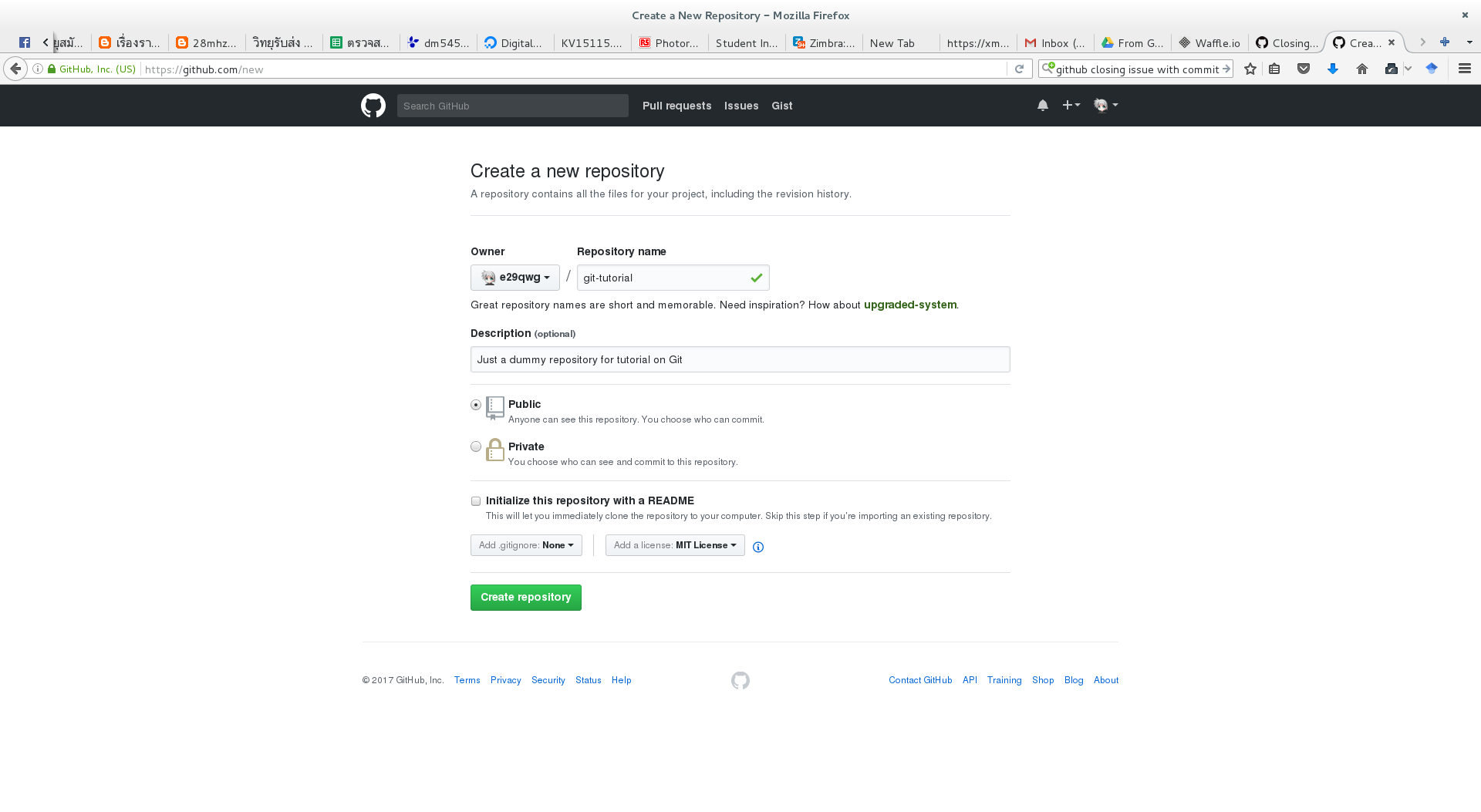
## Repository Management

### Repository Creation

1. กดปุ่ม + บริเวณ user navigation บน header แล้วเลือก New repository ดังภาพ

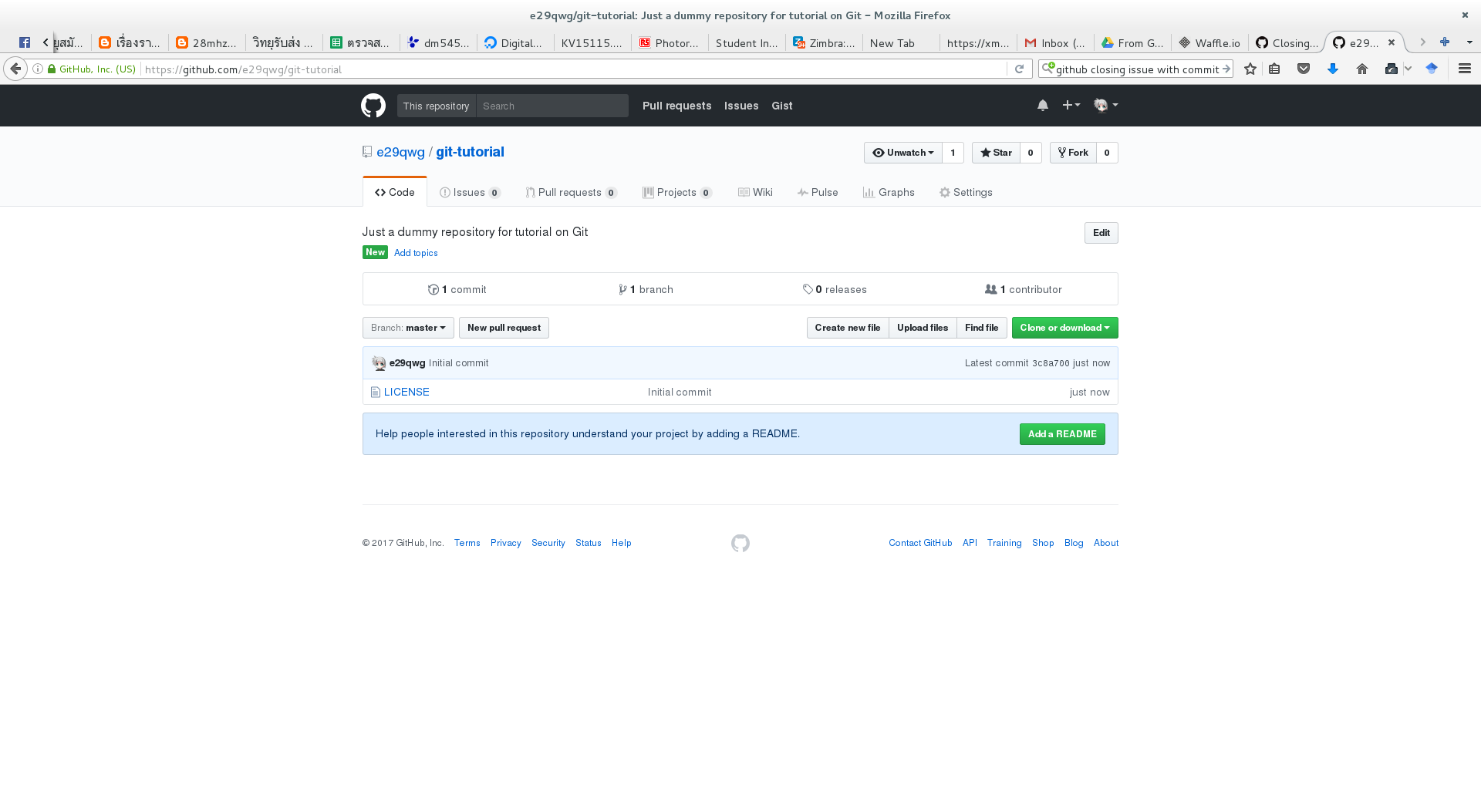


1. จากนั้นกรอกรายละเอียด repository ลงในฟอร์มที่ปรากฎ

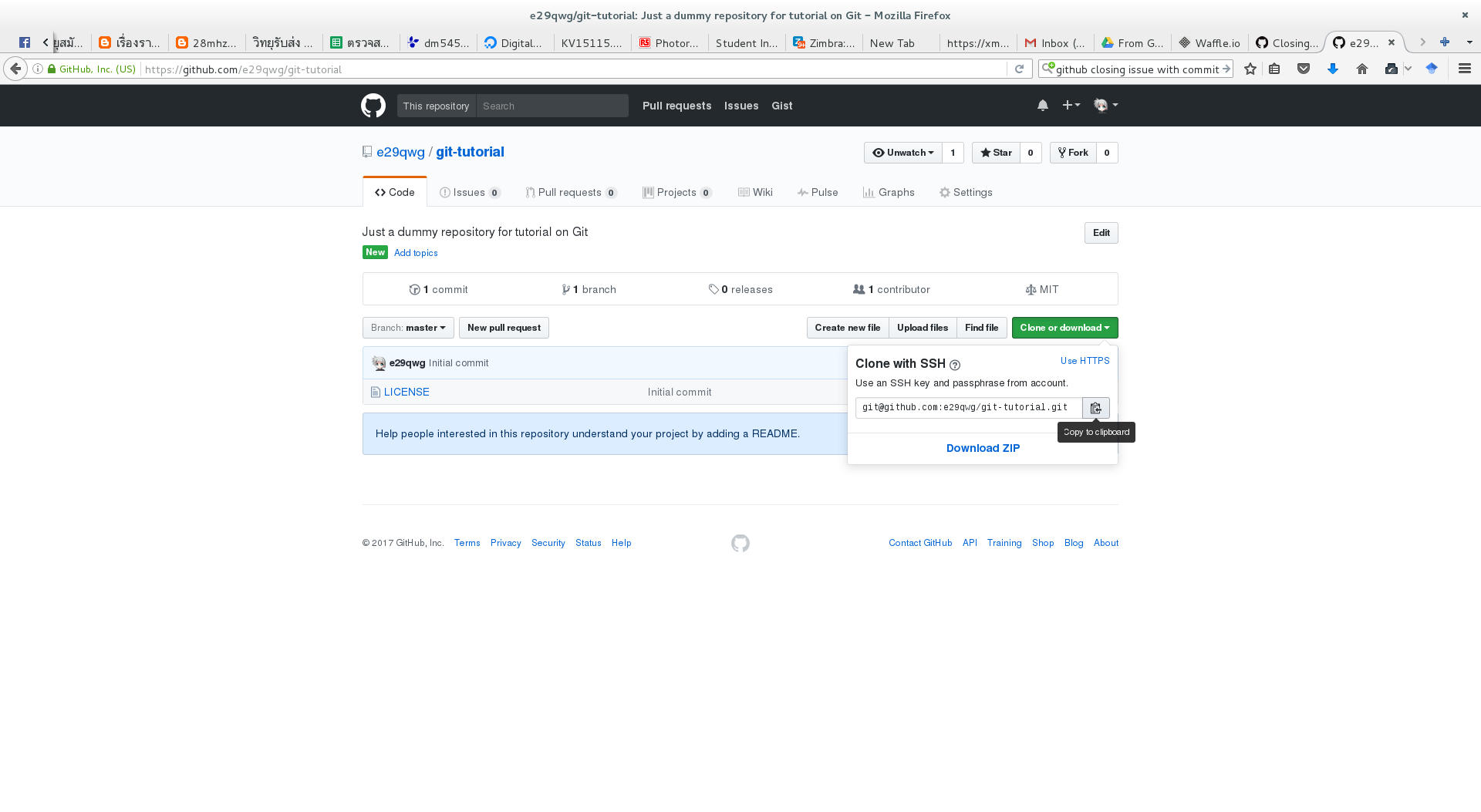


จากภาพตัวอย่างข้างต้น repository ที่จะสร้าง จะถูกตั้งชื่อเป็น git-tutorial เป็น repository แบบ สาธารณะ และอยู่ภายใต้ใบอนุญาต MIT นอกจากนี้ยังสามารถระบุ .gitignore, README และคำอธิบาย repository ได้หากต้องการ

1. เมื่อกรอกรายละเอียดเสร็จแล้ว กดปุ่ม Create repository ก็เป็นอันเรียบร้อย



1. การสร้าง repository ตามขั้นตอนข้างต้นจะได้ remote repository แต่ในขณะที่กำลังพัฒนา จำเป็นจะต้องทำงานใน local repository ดังนั้นจึงจะต้องใช้ git เพื่อ clone repository ในครั้งแรก โดยกดปุ่ม Clone เลือกแบบ SSH หรือ HTTPS (แนะนำใหใช้ SSH) และกด Copy to clipboard



1. จากนั้นใช้คำสั่ง git ดังตัวอย่างด้านล่าง เพื่อ clone remote repository มายัง local

$ git clone git@github.com:e29qwg/git-tutorial.git

### 

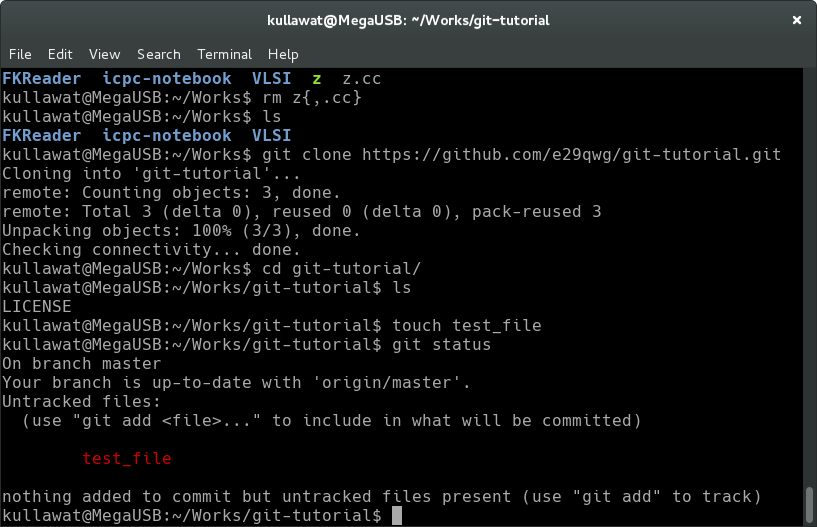
### 

### Staging, Committing, and Pushing

1. ทดลองสร้างไฟล์ใหม่ใน working directory ด้วย editor ใดก็ได้ หรือแก้ไขไฟล์เก่า
2. ตรวจดูสถานะด้วยคำสั่ง

$ git status

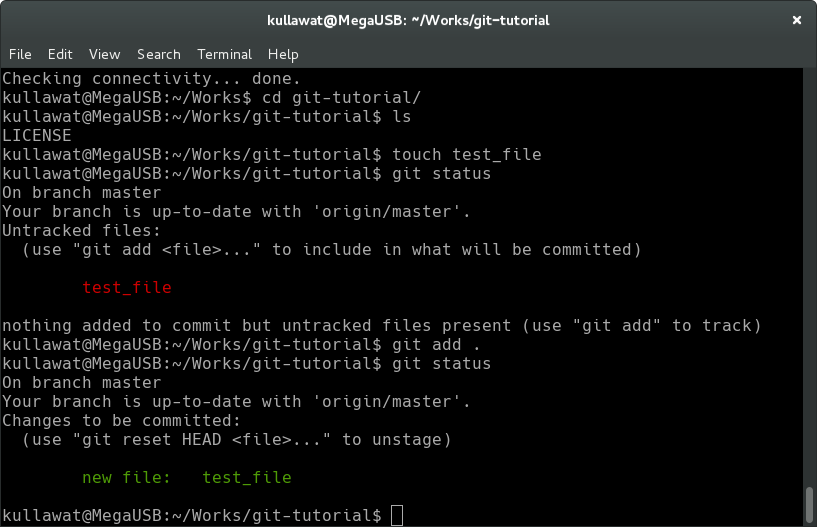
จะพบว่ามี untracked file ในกรณีที่สร้างไฟล์ใหม่ หรือมี modified file ในกรณีที่แก้ไขไฟล์เก่า อย่างไรก็ตาม file ดังกล่าวยังไม่อยู่ใน staging area



1. เคลื่อนไฟล์ไปยัง staging area โดยใช้คำสั่ง

$ git add <filename> หรือ

$ git add . \*\* ต้องสั่งที่ root ของ project directory จึงจะสามารถ add file ทั้งหมด



1. เมื่อตรวจสอบสถานะอีกครั้ง จะพบไฟล์ถูก staged แล้ว แต่ยังไม่อยู่ใน local repository เราสามารถนำไฟล์เข้าสู่ local repository ได้โดยใช้คำสั่ง

$ git commit -m "message"

โดยที่จะต้องใส่ commit message เสมอ เพื่อบอกว่า commit นี้เราได้ทำอะไรลงไปบ้าง

1. จากนั้นนำไฟล์จาก local repository ขึ้นสู่ remote repository โดยการใช้คำสั่ง

$ git push

### Fetching, Pulling, and Traversal

ในกรณีที่ทำงานร่วมกับผู้อื่นในระบบ git เราจำเป็นต้องติดตามอย่างสม่ำเสมอว่า remote repository มีการเปลี่ยนแปลงเกิดขึ้นอย่างไรบ้าง เราสามารถใช้คำสั่งต่อไปนี้เพื่อติดตามการเปลี่ยนแปลงได้

$ git fetch --all

เมื่อ fetch แล้ว เราสามารถตรวจสถานะได้ว่า working directory ที่เราทำงานอยู่นั้นนำหน้าหรือล้าหลัง repository อยู่เท่าไร

หากเราต้องการปรับปรุง working directory ให้ตรงกันกับ repository ที่เพิ่ง fetch มา เราสามารถใช้คำสั่ง

$ git pull

การ pull อาจทำให้ตำแหน่งของ HEAD (และ tag) เปลี่ยนไป (เบื้องหลังของการ pull คือ fetch + merge) ดังนั้นจึงต้องมั่นใจว่าทดสอบเรียบร้อยแล้ว ก่อนจะใช้คำสั่ง pull ใน production

เราสามารถท่องไปยัง commit ต่าง ๆ ที่ผ่านมาแล้วได้ โดยการหาเลข commit (ปัจจุบันใช้ SHA-1 hash) ผ่านทาง log หรือ network graph อย่างไรก็ตาม ไม่จำเป็นต้องระบุเลข commit เต็มทั้ง 40 hexadecimal digits แต่ใช้เพียงแค่ 7 หลักแรกก็เพียงพอแล้วสำหรับ repository ที่มีจำนวน commit ไม่เกิน 268,435,456 commits

ตัวอย่างเช่น ต้องการไปยัง commit หมายเลข 6fa60bda2012c58960a82d4a0da2f3d8a3ee64ea ก็ใช้คำสั่งต่อไปนี้

$ git checkout 6fa60bd

เมื่อทำเช่นนี้ HEAD จะย้ายไปยังตำแหน่งที่ไม่มี reference เมื่อตรวจสถานะ จะพบว่า git ระบุว่าเป็น detached HEAD เพื่อให้เราทราบ

### Branching and Merging

เราสามารถแบ่งสายการทำงานได้ด้วยระบบ branch ซึ่งในความเป็นจริงแล้ว branch เป็นเพียง reference pointer ที่ชี้ไปยัง commit ทำให้เราสามารถตั้งชื่อแต่ละสายในการพัฒนาได้ ตามความนิยมแล้ว เรามักจะแบ่ง branch ตามหน้าที่ เช่น

* develop สำหรับการพัฒนาทั่วไป
* feature สำหรับการสร้างคุณลักษณะหรือความสามารถใหม่ของระบบ
* hotfix สำหรับการแก้ไขข้อผิดพลาด / patch
* master สำหรับการออก stable release

อย่างไรก็ตาม ที่นำเสนอข้างต้นเป็นเพียงความเห็นส่วนตัว ในองค์กรหรือทีมพัฒนาอาจเลือกใช้ชื่ออื่น หรือใช้เพื่อจุดประสงค์อื่นไม่ตายตัว แล้วแต่จะตกลงกัน

เราสามารถสร้าง branch ใหม่จากตำแหน่งที่ HEAD ของ local repository อยู่ปัจจุบัน ได้ด้วยคำสั่ง

$ git branch <branch\_name>  
 $ git checkout <branch\_name>

หรืออาจรวบเป็นคำสั่งเดียวได้ว่า

$ git checkout -b <branch\_name>

และสามารถลบ branch ที่ไม่ใช้งานแล้วทิ้งได้โดยใช้คำสั่ง

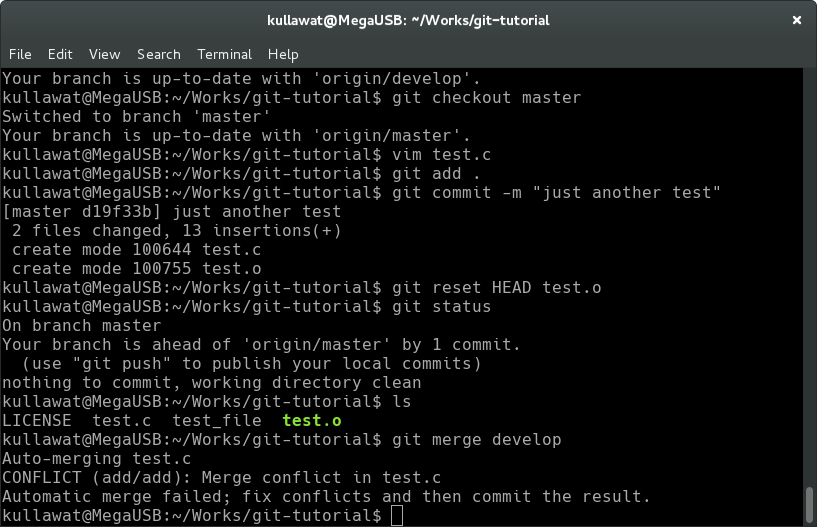
$ git branch -d <branch\_name> ลบ local branch  
 $ git push origin --delete <branch\_name> ลบ remote branch

และเมื่อต้องการนำงานจาก branch หนึ่ง รวมเข้ากับอีก branch หนึ่ง ก็สามารถใช้คำสั่ง merge เช่นหากต้องการรวมการพัฒนาใน branch develop เข้าสู่ master สามารถใช้คำสั่งต่อไปนี้

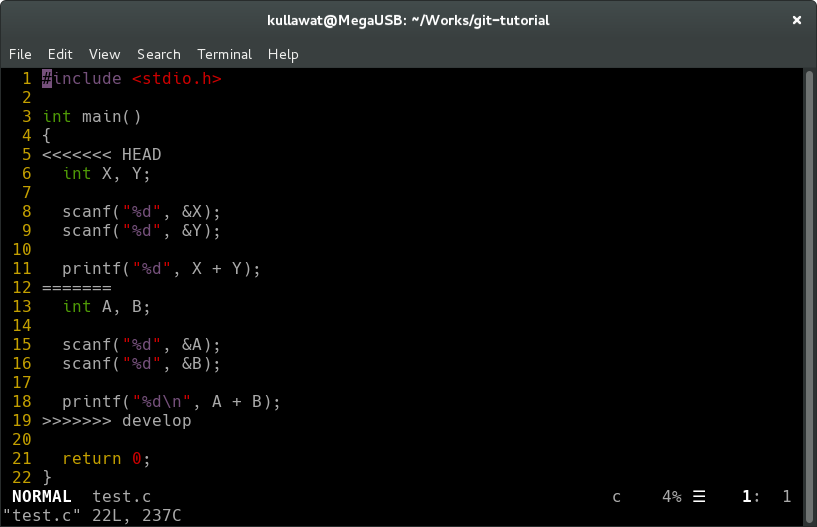
$ git checkout master  
 $ git merge develop อาจใส่ flag --no-ff เพื่อให้เกิด commit ใหม่

### Merge Conflict Resolution

ในการทำงานด้วยระบบ branch อาจเกิดเหตุการณ์ที่มีการแก้ไข file เดียวกัน บรรทัดเดียวกัน ด้วยเหตุนี้ จึงมีข้อขัดแย้ง (conflict) เกิดขึ้น ทำให้การ merge ไม่สำเร็จ เราจำเป็นต้องเข้าไปยัง file ที่เกิด conflict และแก้ไข จากนั้นจึงค่อย commit ด้วยตนเอง



ดังภาพข้างตน จะพบว่าไฟล์ test.c เกิด merge conflict เมื่อเข้าไปตรวจเนื้อหาไฟล์แล้วเป็นดังนี้



จากภาพข้างต้น จะพบว่า git ได้แยกส่วนของเนื้อหาที่เกิด conflict ไว้ในรูปแบบ

<<<<<<<<<< HEAD  
 ...  
 ==========  
 ...  
 >>>>>>>>>> develop

ส่วนที่อยู่เหนือเครื่องหมาย ==== คือเนื้อหาของ HEAD (ปัจจุบันอยู่ที่ master branch) ส่วนด้านใต้เครื่องหมาย ===== คือเนื้อกหาจาก develop branch ที่ถูก merege เข้ามา

วิธีการแก้ไขนั้นขึ้นอยู่กับดุลยพินิจของผู้พัฒนาว่าต้องการให้การทำงานของ code หรือเนื้อหาส่วนที่ conflict เป็นไปอย่างไร เมื่อแก้ไขแล้ว ให้ stage file อีกครั้งด้วยคำสั่ง git add จากนั้นจึงสั่ง commit

### Tagging

นอกจากระบบ branch แล้ว ยังมีระบบ tag ที่ช่วยให้เราสามารถติด reference ไว้ตาม commit ต่าง ๆ ได้ อย่างไรก็ตาม เรามักจะติด tag เป็นหมายเลข version เช่น

$ git tag v1.0

สำหรับการ push tag ไปยัง remote repository สามารถทำได้โดย

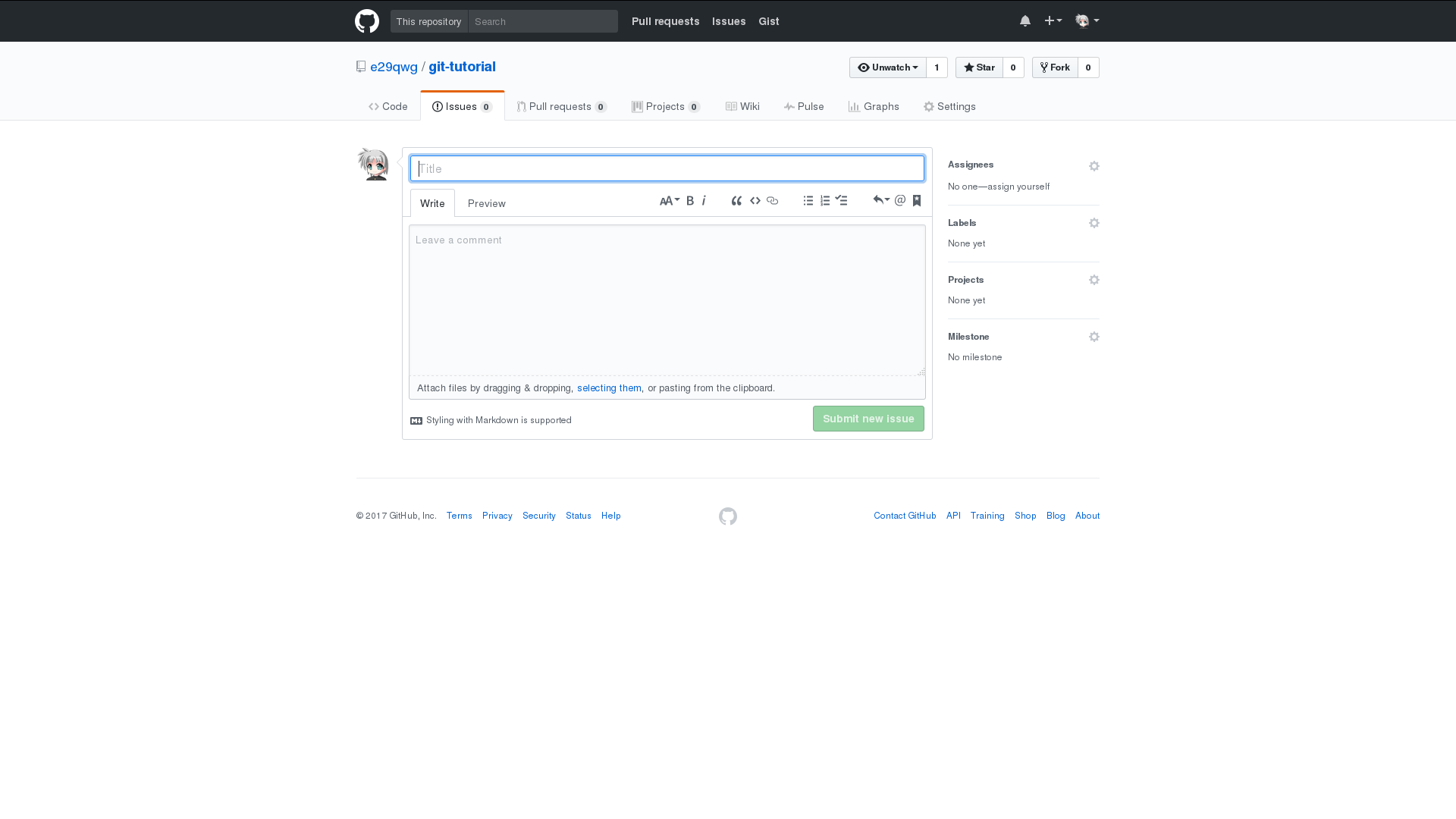
$ git push --tags

และเราสามารถกลับมายัง tag ได้โดยการ checkout (แต่จะเป็นลักษณะ detached HEAD) เช่น

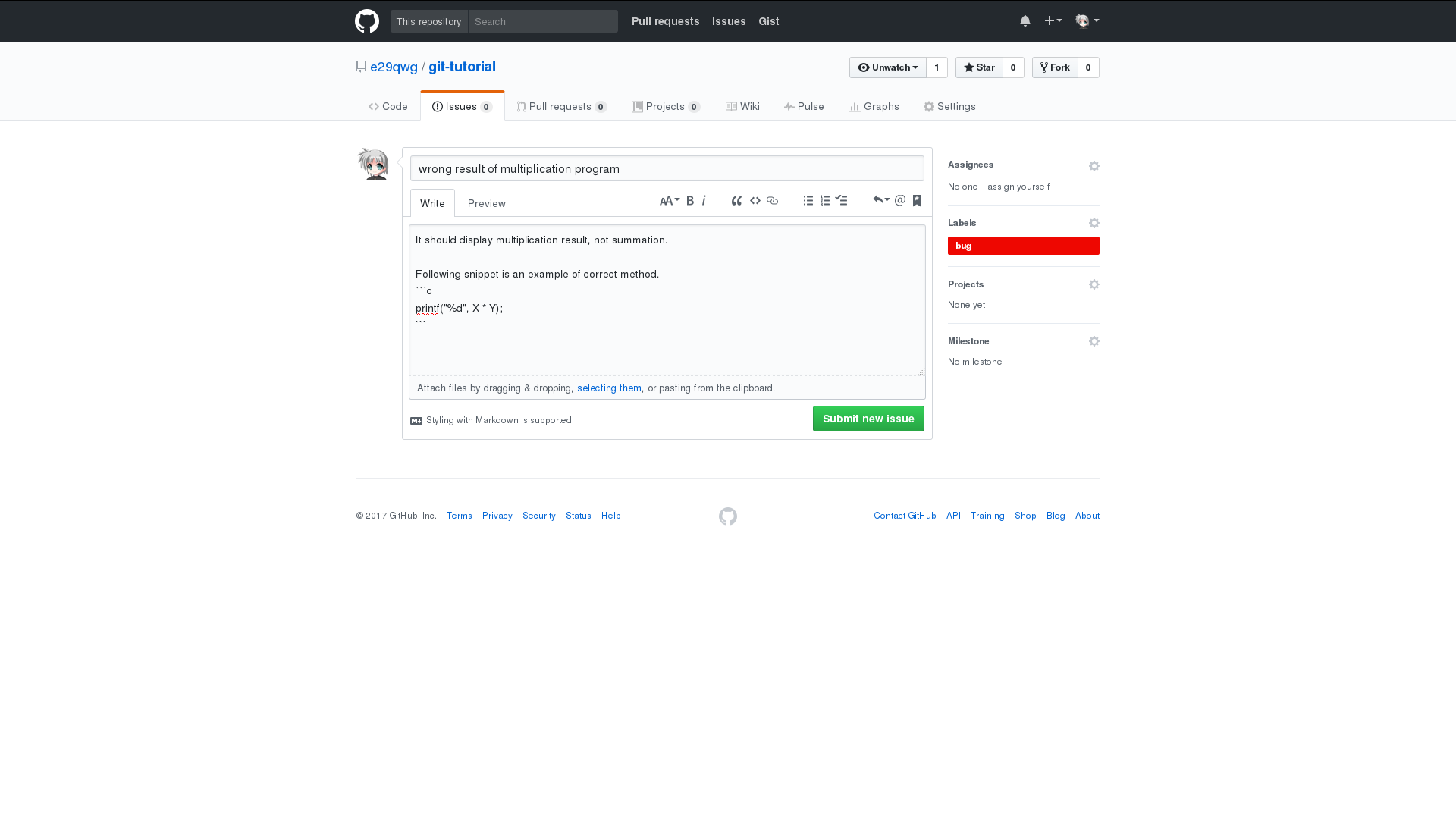
$ git checkout v1.0

### Issues and Milestones

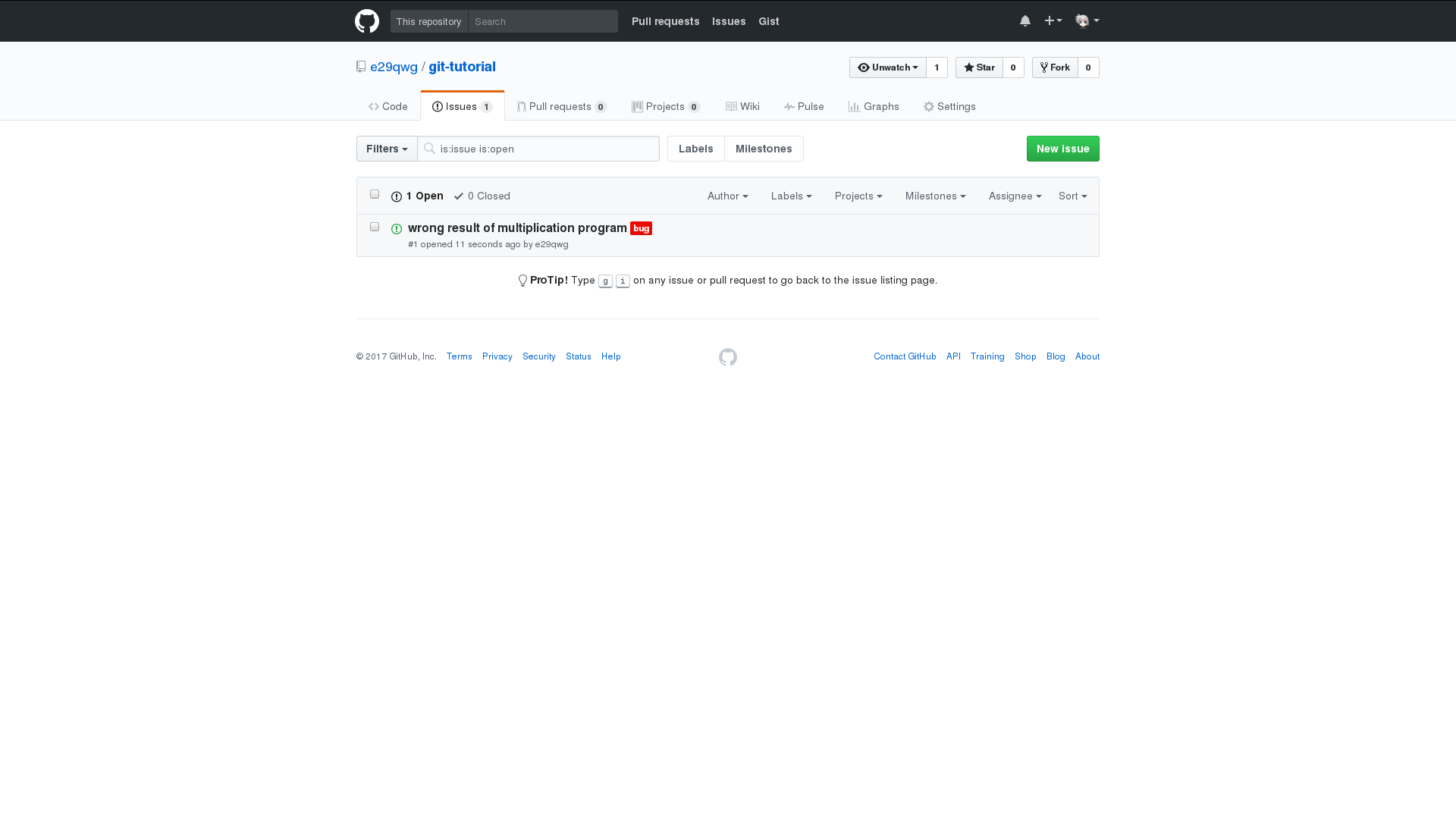
GitHub รวมถึงผู้ให้บริการ git repository อื่น ๆ มักมีระบบ issue ให้บริการ ก่อนอื่นต้องทำความรู้จักกับ issue กันก่อน issue คือประเด็นต่าง ๆ ที่เกิดขึ้นระหว่างการพัฒนา อาจเป็นได้ตั้งแต่ bug, malfunction, feature request, enhancement, question, หรืออื่น ๆ ตามแต่จะตกลงกันในทีมพัฒนา สำหรับ GitHub มีตัวอย่างระบบ issue ดังภาพด้านล่าง



การส่ง issue นั้นจำเป็นจะต้องระบุเหตุการณ์ที่เกิดขึ้น / ที่ควรจะเป็นให้ชัดเจน หากสามารถแสดงต้นตอ หรือตัวอย่าง code ที่ถูกต้องได้ ก็ควรจะใส่ข้อมูลเหล่านี้ไว้ด้วยเพื่อเป็นประโยชน์ต่อทีมผู้พัฒนา ดังจะเห็น ตัวอย่างการส่ง issue ได้ในภาพด้านล่าง



แต่ละ issue จะมีหมายเลขกำกับ และจะมีสองสถานะหลักคือ open และ close สถานะทั้งสองนี้ใช้ระบุว่า ประเด็นดังกล่าวได้รับการพิจารณาหรือแก้ไขแล้วหรือไม่

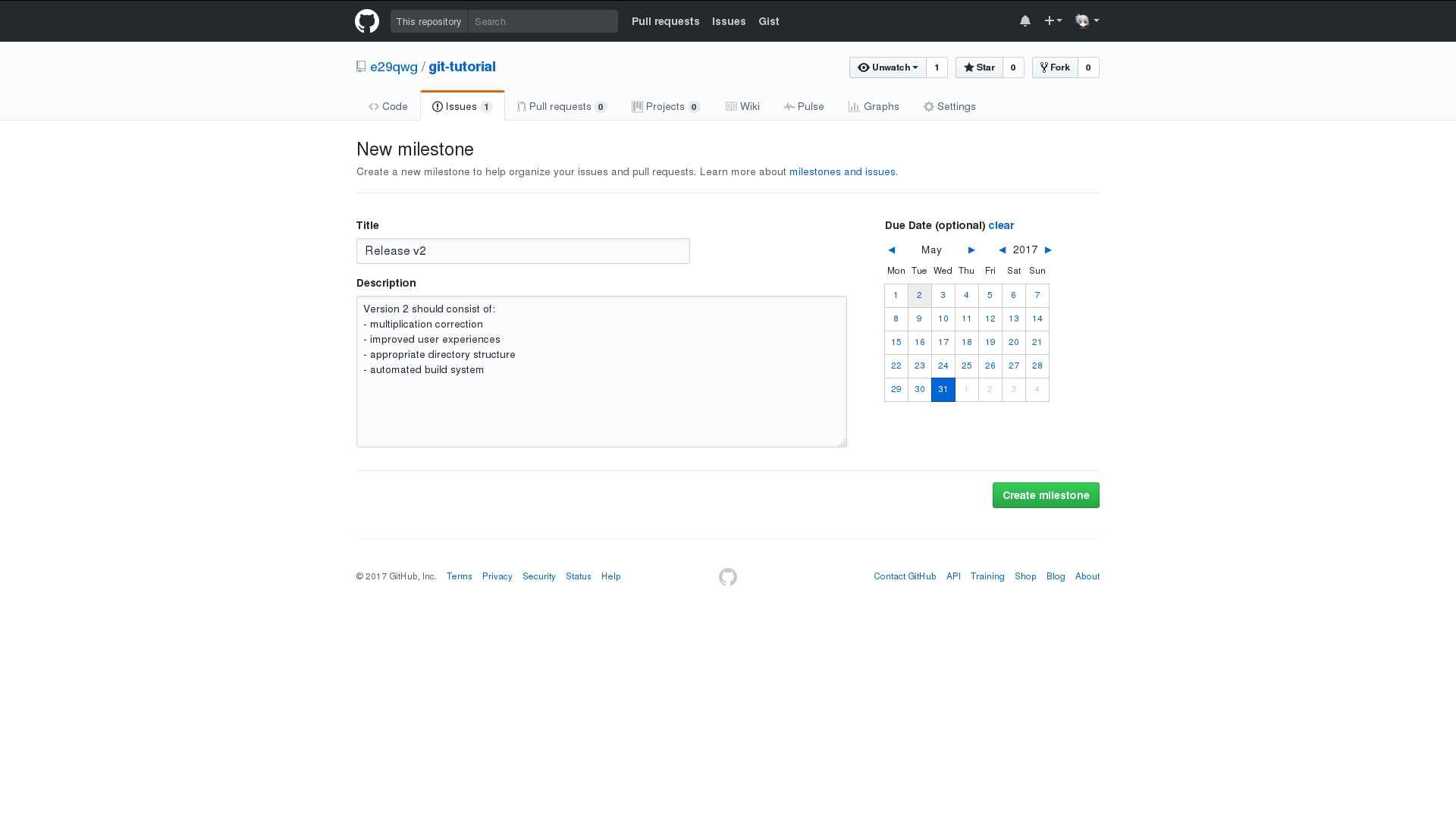


ด้วยระบบหมายเลขกำกับ issue ทำให้อ้างอิง และสามารถปิด issue จาก commit message ได้ด้วย ดังตัวอย่างมี issue หมายเลข 1 สามารถปิด issue จาก commit message หลังจากแก้ไขได้โดย

$ git commit -m "fix #1 resolve multiplication code"

ไม่ว่า commit นี้จะอยู่ใน branch ใดก็แล้วแต่ เมื่อ merge เข้าสู่ master branch ระบบจะ close issue ให้โดยอัตโนมัติ

นอกจากระบบ issue แล้ว ยังมีระบบ milestone ทำให้เราสามารถระบุเป้าหมายของงานว่า release ถัดไปจะต้องประกอบด้วยอะไรบ้างวันที่สิ้นสุดวันไหน และสามารถผู้ issue กับ milestone ได้อีกด้วย (กล่าวคือต้องปิด issue ให้ทัน milestone)



## 

## 

## Advanced Topics

### Stashing

ในกรณีที่ทำงานใน working repository ค้างอยู่ แต่กลับเกิดสถานการณ์ต่อไปนี้

* พบว่ากำลังทำงานอยู่ผิด branch
* มีความจำเป็นต้อง merge กระทันหัน (รวมถึงการ pull ด้วย)
* ลืมสร้าง branch ใหม่ก่อนเริ่มพัฒนา (การ commit บน master branch เป็นบาปในบาง model)
* ลืมว่าเดิม code นี้ต้องทำงานอย่างไร แต่ลบหรือ refactor จนมั่วไปแล้ว ต้องกลับไปดูของเก่า

ปัญหาเหล่านี้สามารถแก้ไขได้ด้วย stashing ซึ่งเป็นระบบย่อยสำหรับพักงานชั่วคราว เมื่อต้องการพักงาน โดยยังคงเก็บการพัฒนาที่เคยทำไว้ เราสามารถใช้คำสั่ง

$ git stash

และเมื่อเราแก้สถานการณ์ดังกล่าวข้างต้น (เช่น ถ้าทำงานผิด branch ก็ checkout ไปยัง branch ที่ควร หรือ pull ในกรณีที่จำเป็น) เราสามารถเรียกงานที่ค้างอยู่คืนมาได้ด้วยคำสั่ง

$ git stash apply หรือ  
 $ git stash pop \*\* ระวัง หลัง stash apply / pop อาจมี conflict

เราสามารถดูรายการ stash ทั้งหมดที่เก็บไว้ได้ด้วยคำสั่ง

$ git stash list

และสามารถลบ stash ได้ด้วยคำสั่ง

$ git stash drop [<stash>] เพื่อลบ stash ที่เลือก หรือ  
 $ git stash clear เพื่อล้าง stash ทั้งหมด

ในกรณีที่งานที่ทำค้างอยู่ควรจะขึ้นเป็น branch ใหม่ เราไม่จำเป็นต้องสร้าง branch ใหม่ด้วยตัวเอง แล้วสั่ง apply แต่อย่างใด เราสามารถใช้คำสั่งย่อยของ stash เพื่อสร้าง branch ใหม่ได้ทันที โดยมีรูปแบบคำสั่งคือ

$ git stash branch <branchname>

การประยุกต์ใช้ stash นั้นจำเป็นต้องเรียนรู้เกี่ยวกับพื้นฐานของ git ให้เข้าใจ สามารถแยกแยะระหว่าง working directory และ repository ได้และเข้าใจถึงสถานะของไฟล์ ผู้ใช้จำเป็นต้องฝึกฝนทักษะการใช้งาน stash และเรียนรู้ถึงวิธีการประยุกต์ใช้ในเหตุการณ์ต่าง ๆ จึงจะเกิดประสิทธิผล

แม้ stashing จะไม่ใช่เรื่องพื้นฐาน แต่อย่างไรก็ตาม ประโยชน์ของ stashing เป็นสิ่งที่ประเมินค่าไม่ได้ และควรค่าแก่การเรียนรู้

### Bugs Searching

1. Clone project ทดสอบจาก <https://github.com/e29qwg/git-tutorial>

$ git clone https://github.com/e29qwg/git-tutorial.git

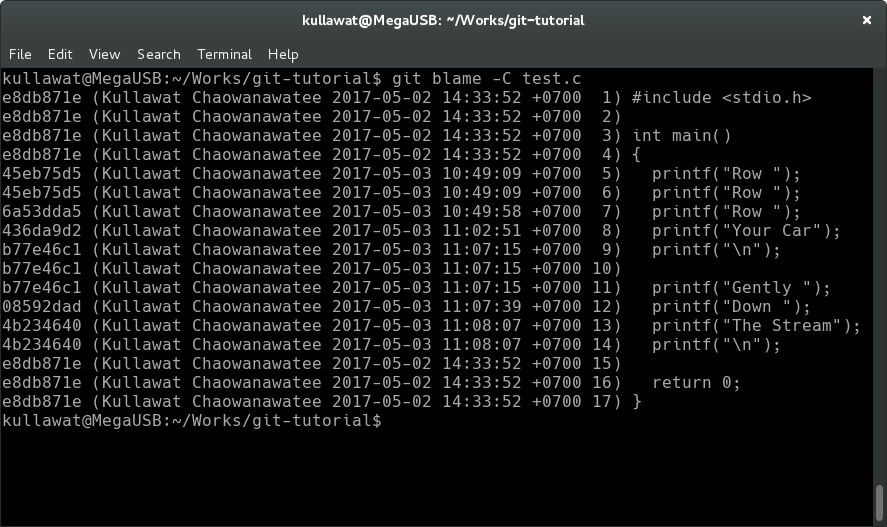
1. ทดลอง compile และ run โปรแกรม test.c

$ gcc -Wall -O2 -o test.o test.c

ข้อความที่ควรจะเป็นคือ Row Row Row Your Boat, Gently Down The Stream แต่โปแกรมพิมพ์ข้อความนี้ออกมาผิด เป็น Row Row Your Car, Gently Down The Stream

1. **วิธีที่ 1** - ใช้ blame เพื่อตรวจดูความเคลื่อนไหวล่าสุดของไฟล์ test.c

$ git blame -C test.c



จะพบว่าบรรทัดที่ทำให้เกิดความผิดพลาดคือบรรทัดที่ 8 ซึ่งเกิดจาก commit 436da9d2

นอกจากนี้ยังสามารถระบุบรรทัดได้จาก option -L หรืออาจใช้ blame ผ่านทางหน้าเว็บ GitHub ก็ได้เช่นกัน อย่างไรก็ตาม ในบางกรณี blame อาจไม่ได้ระบุถึงต้นตอของปัญหาอย่างแท้จริง

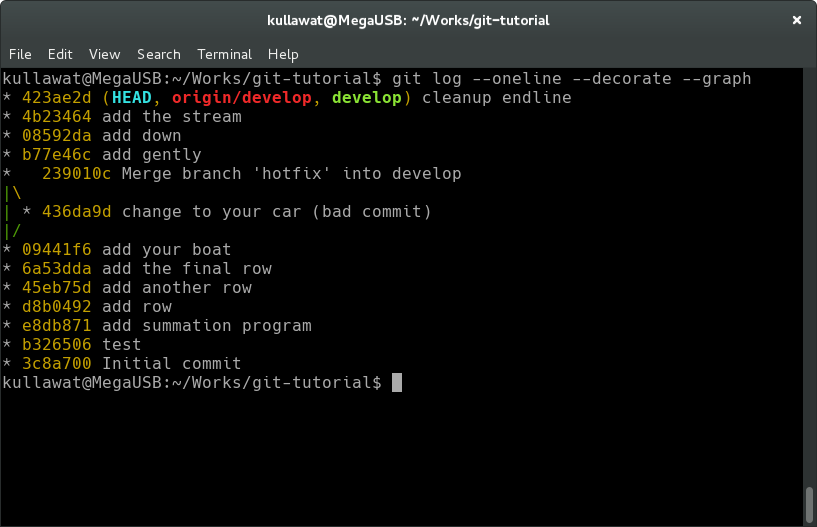
1. ให้ทดลอง checkout ไปยัง develop branch แล้วตรวจด้วย blame อีกครั้ง

$ git checkout develop  
 $ git blame -C test.c

จากนั้นสังเกตผลจาก blame จะเห็นว่ามันไม่อาจระบุถึงต้นตอได้อย่างแท้จริง ยิ่งไปกว่านั้นคือ หากมี commit ตามหลังจำนวนมาก จะยิ่งทำให้ยากต่อการย้อนรอย

1. **วิธีที่ 2** - ตรวจกราฟ network ผ่านทางเว็บไซต์ GitHub หรือผ่านทางคำสั่ง log ของ git

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



เพื่อหา commit ที่ทดสอบแล้วว่าทุกอย่างยังโอเคดีอยู่ ทั้งนี้ ขอเลือก commit 6a53dda หรือ 09441f6 เนื่องจากเป็น commit ที่ยังไม่มีคำสั่งสำหรับพิมพ์คำว่า Car

1. เริ่มขั้นตอนการหา bug ด้วยวิธี binary search โดยการใช้คำสั่ง

$ git bisect start  
 $ git bisect good 6a53dda  
 $ git bisect bad

1. จากนั้น git จะประเมินจำนวน step ที่ใช้ในการหา bug และค่อย ๆ ย้อนรอยแบบ binary search เราจะต้องทดสอบ code ด้วยตนเองว่ายังเกิด bug อยู่หรือไม่ ในที่นี้ให้ใช้คำสั่ง

$ gcc -Wall -o test.o test.c && ./test.o

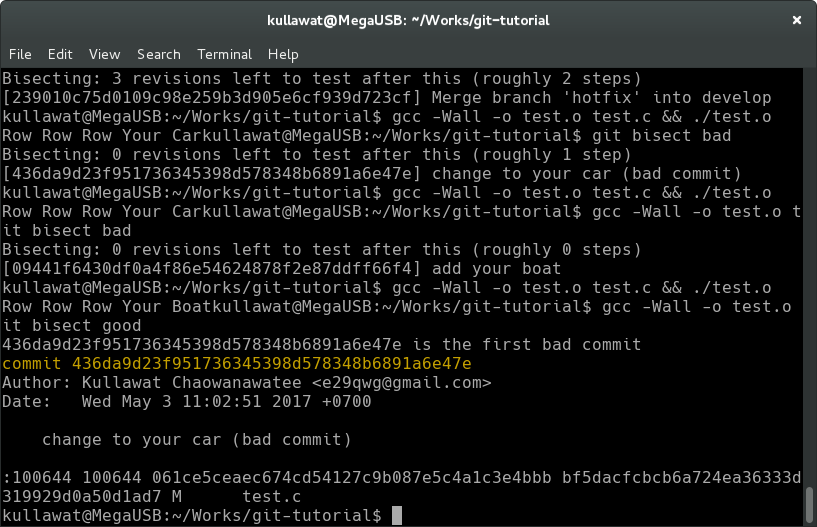
หากพบว่ายังมีผลลัพธ์ผิดพลาด (คำว่า Car) อยู่ ให้ mark bad ด้วยคำสั่ง

$ git bisect bad

แต่หากไม่มีผลลัพธ์ผิดพลาดปรากฏ ให้ mark good ด้วยคำสั่ง

$ git bisect good

ทำเช่นนี้ไปเรื่อย ๆ จนกระทั่งครบ step แล้ว git จะระบุ commit แรกสุดที่ผิดพลาดให้



1. ผลจากการ bisect จะได้เป็น detached HEAD และมี reference พิเศษสำหรับ bisect จากจุดนี้สามารถส่งรายงานความผิดพลาดให้ผู้พัฒนาได้ และเมื่อต้องการกลับไปยัง original branch ให้ใช้คำสั่ง

$ git bisect reset

### Rebasing and Cherry Picking

จากการค้นหา bug ดังที่ได้กล่าวไปแล้วนั้น เราสามารถแก้ไขได้หลายวิธี เช่น อาจะสร้าง hotfix ขึ้นมาเป็น branch ใหม่ แล้ว merge เข้าสู่ branch ต่าง ๆ โดยเฉพาะอย่างยิ่ง release เก่าได้รับผลกระทบ

อย่างไรก็ตาม ในกรณีที่ไม่มี release เก่า หรือจำนวน commit นับจาก first bad commit จนถึง commit ล่าสุดยังมีจำนวนไม่มาก (1 - 16 commits / 1 - 2 branches) เราอาจเลือกใช้ทางออกอื่น เช่น Cherry Picking หรือ Rebasing ซึ่งต้องใช้เวลาในการทดสอบและ resolve conflict

**วิธีที่ 1 - Cherry Picking**

1. จากการทำ bisect เราพบว่า commit แรกสุดที่ผิดพลาดคือ 436da9d แต่เราจะไม่ detach HEAD ไปที่จุดนี้ แต่จะย้อนไป 1 commit ก่อนจะผิด (last good)

$ git checkout 436da9d^

เครื่องหมาย ^ ที่ใส่ตามหลัง หมายถึง ก่อนหน้า 1 commit หรืออาจใช้ ~1 แทนก็ได้

1. จากนั้นตรวจ log เพื่อใช้ดูหมายเลข commit

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

1. เลือกเก็บแต่ commit ดี (เหมือนเราไปเก็บเชอร์รี่ในสวน เราก็เลือกแต่ผลดี ผลไหนเน่าเสียก็เด็ดทิ้ง)

$ git cherry-pick b77e46c 08592da 4b23464 423ae2d

ระหว่างนี้อาจเจอ conflict เมื่อ resolve แล้ว ให้ staging, commit และ cherry pick ต่อ โดยสั่ง

$ git add -A && git commit  
 $ git cherry-pick --continue ทำเช่นนี้ไปเรื่อย ๆ จนกระทั่งครบทุก commit

1. หากตรวจ log จะพบว่าจุดที่ทำงานอยู่ปัจจุบันเป็น detached HEAD ให้เราย้าย branch develop ที่เป็นปัญหามายัง HEAD (ซึ่งเราเพิ่งแก้ปัญหาด้วย cherry picking)

$ git branch -f develop HEAD

หลังจากนั้นตรวจสอบความถูกต้อง จึง stage, commit และ push ตามลำดับ

**วิธีที่ 2 - Rebasing \*\***

1. จากการทำ bisect เราพบว่า commit แรกสุดที่ผิดพลาดคือ 436da9d ให้ตั้งจุด new base ที่ commit ก่อนหน้า และเริ่ม rebase โดยการใช้คำสั่ง

$ git rebase -i 436da9d^

สังเกตว่าเราไม่จำเป็นต้อง detach HEAD

1. เมื่อสั่ง rebase แล้วจะมี commit ระหว่างทางจาก new base ไปยัง HEAD ให้ pick เฉพาะ commit ที่ดี ส่วน commit 436da9d ที่เราหมายไว้ว่าผิดพลาด ให้ลบทิ้งไป จากนั้น save และปิด editor
2. ระหว่างการ rebase อาจเจอ conflict ให้ resolve จากนั้น staging, commit และ rebase ต่อ โดยสั่ง

$ git add -A && git commit  
 $ git rebase --continue

1. จากนั้นตรวจสอบความถูกต้อง จึง stage, commit และ push ตามลำดับ

\*\* หมายเหตุ ในกรณีที่มี branch จำนวนมาก การ rebase อาจไม่ใช่ทางออกที่ดีนัก

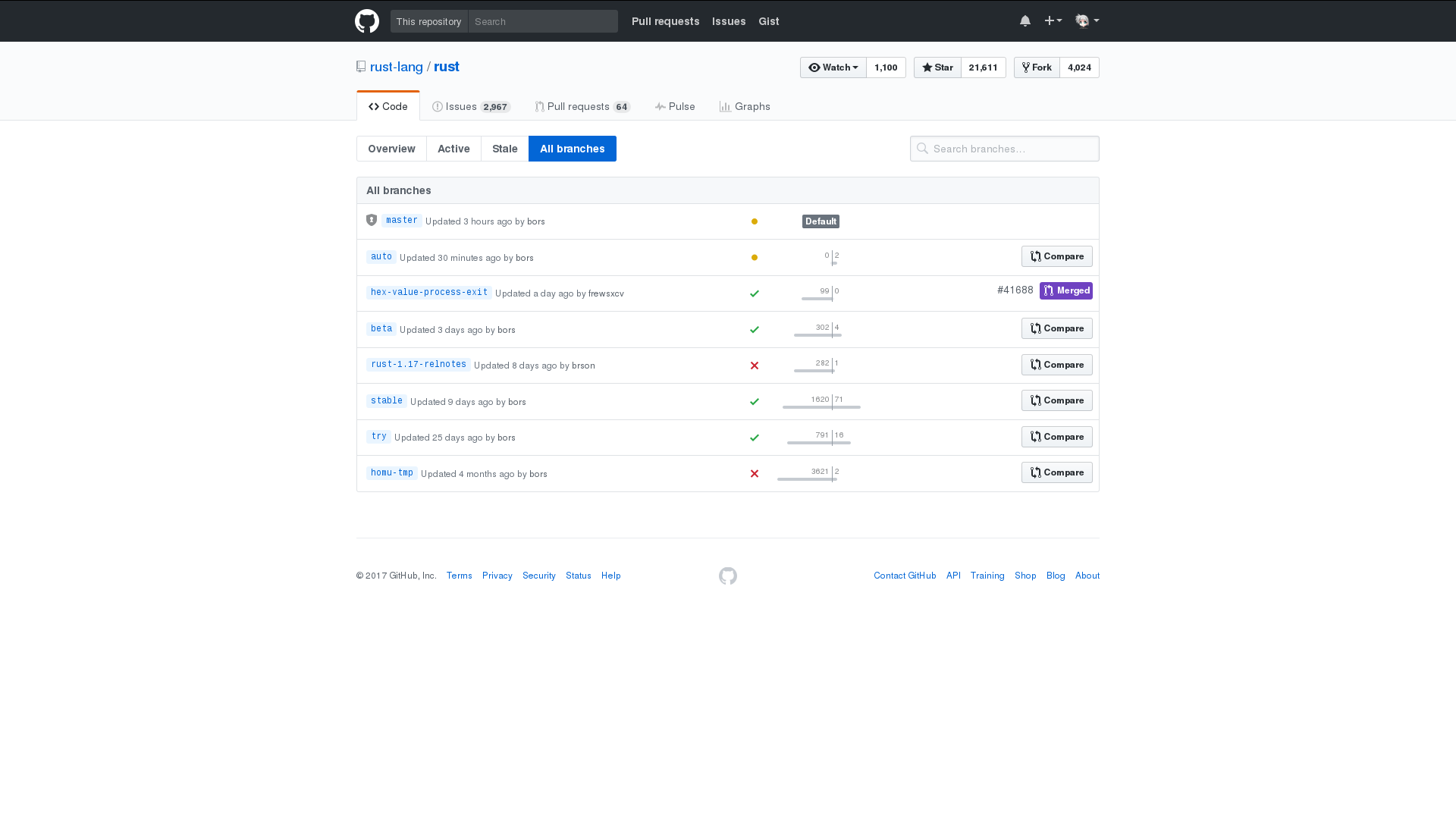
นอกจากการใช้ cherry picking และ rebasing ในการทิ้ง commit เสียแล้วนั้น ยังสามารถประยุกต์ได้ เช่น การรวม branch เข้าด้วยกันโดยไม่ใช้การ merge โดยการนำ commit เฉพาะบางอันจาก branch หนึ่งไปรวมกับอีก branch หนึ่ง (การ merge จะรวม**ทุก** commit ของ branch แยก)

## 

## 

## Git Flow Model

แต่ละองค์การอาจมี model ในการทำงานร่วมกัน หรือแนวทางในการใช้งาน branch แตกต่างกัน ตัวอย่างเช่น project Rust ของ Mozilla มี 8 branches ทำงานกันใน master branch เป็นหลัก และแตก branch เมื่อต้องทำ hotfix หรือมี feature ใหม่ และมี branch รองคือ beta และ stable



การตกลงกันในทีมเกี่ยวกับการใช้งาน branch ร่วมกันเป็นเรื่องที่สำคัญมาก ดังที่เปาบุ้นจิ้นได้กล่าวไว้ว่า “บ้านมีกฏบ้าน เมืองมีกฏหมาย” หรือเลอเชโร่ในเรื่อง Prison Break ก็กล่าวไว้ว่า “กฏต้องเป็นกฏ ถ้าไม่มีกฏ เราก็เป็นคนเถื่อน”

อย่างไรก็ตาม Vincent Driessen (nvie) ได้แนะนำ model ไว้แบบหนึ่ง (ซึ่งตามความเห็นส่วนตัว เป็น model ที่เหมาะกับทีมพัฒนาขนาดเล็ก - กลาง และเหมาะกับงานที่มีกลยุทธ์ในการออก release ชัดเจน) ซึ่ง nvie เรียก model นี้ว่า Git Flow นอกจากจะเป็นระบบแล้วยังสามารถทำความเข้าใจได้ง่าย สามารถศึกษาได้จาก <http://nvie.com/posts/a-successful-git-branching-model/>

## Scrum Board using Waffle.io (for GitHub)

จากที่ได้เรียนรู้เกี่ยวกับระบบ issue และ milestone ไปแล้วนั้น เราสามารถนำ issue มาดำเนินงาน ในลักษณะ scrum board / todo list ได้อีกด้วย ซึ่งสามารถใช้งานเบื้องต้นได้ฟรีผ่านทาง waffle.io อย่างไรก็ตามหากต้องการจะใช้กับ private repository ควรศึกษา terms of service, disclaimer และ privacy policy ให้กระจ่างก่อนตัดสินใจใช้งาน

(สาธิต Waffle.io)

## Git Cheatsheet

ขอแนะนำ Cheatsheet จาก GitHub เอง ซึ่งได้รวบรวมคำสั่งไว้เพียงพอต่อการใช้งานทั่วไปแล้ว

<https://education.github.com/git-cheat-sheet-education.pdf>

## Suggested Playground and References

Sandbox สำหรับศึกษาและทดลอง git พื้นฐาน - ขั้นสูง <http://learngitbranching.js.org/>

หนังสือ Git แจกฟรีจาก Git เอง (ProGit 2) <https://git-scm.com/book>

รวม Reference การใช้งานคำสั่งต่าง ๆ <https://git-scm.com/docs>

เรื่องอื่น ๆ เกี่ยวกับ git ที่ไม่ได้สอน แต่ควรค่าต่อการเรียนรู้

* Submodules -- สำหรับ project ที่แบ่งออกเป็น project ย่อย ๆ
* Subtree Merging -- ใช้แทน Submodules
* Client-side Hooks --
* Setting up Git server
* Rerere
* Bundling
* Large File Storage (LFS)
* Git Annex (Archive and Synchronize)
* GPG Tagging
* Advanced Rebasing, Replacing and Revert
* Working on multiple remote branches
* Repository Maintenance and Data Recovery

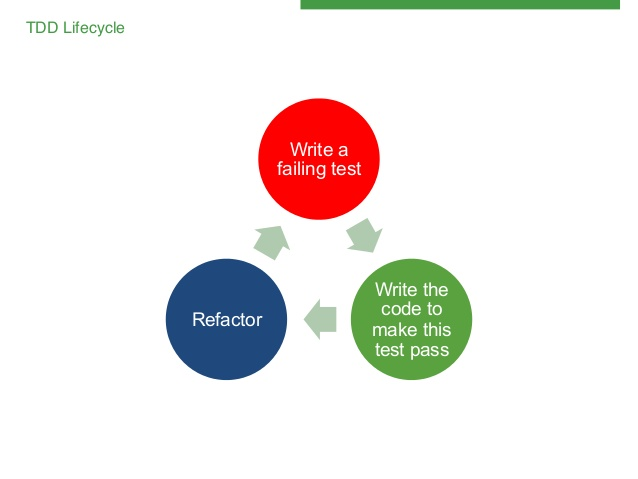
# Test Driven Development

## Why TDD?

(ยาวไปไม่อ่าน) การใช้ TDD เป็นเขียนการทดสอบ เพื่อนำไปสู่การพัฒนาระบบ ทำให้มั่นใจว่าโปรแกรมจะทำงานได้อย่างสมบูรณ์ (ตามกรณีที่เขียนการทดสอบไว้)

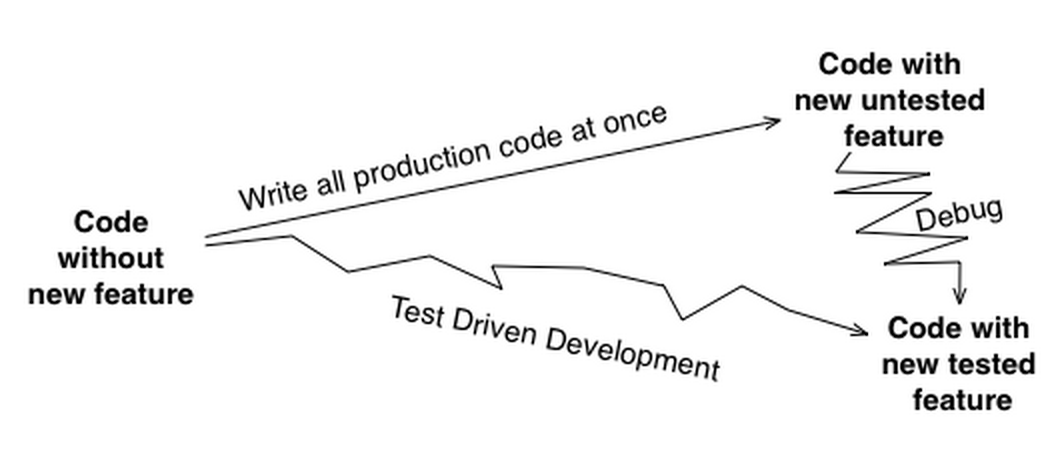
---------------

### TDD Lifecycle



(from: https://image.slidesharecdn.com/tdd-140120052130-phpapp02/95/test-driven-development-workshop-17-638.jpg)

### TDD Small Steps



(from: http://www.somkiat.cc/wp-content/uploads/2014/11/Screen-Shot-2557-11-17-at-8.54.20-PM.png)

การทำ Test จะส่วนใหญ่แบ่งเป็น 2 ระดับ

* API Test - ใช้การเขียน Unit test เพื่อตรวจสอบว่า function ที่เขียนนั้น ทำงานได้ถูกต้องหรือไม่
* Integration Test - ใช้การจำลองการกรอกข้อมูลลงใน form ของ web browser, submit และตรวจสอบว่าข้อมูลที่ได้ใช้ในการทดสอบทำงานได้ถูกต้องหรือไม่

มาเริ่มต้นกับ API Test โดยใช้ phpunit กัน

## Unit testing in PHP

ให้ติดตั้ง Laravel 5.3 เพื่อจะทดสอบการเขียน Test แบบง่าย ๆ โดยใช้ phpunit การเขียน test ใน Laravel สามารถทำได้ทั้งในส่วนที่เป็น API test และ Web Test (ในส่วนของ web test ตั้งแต่ Laravel 5.4 ขึ้นไป จะใช้ ChromeDriver ใน Laravel Dusk เป็น Integrating test แทน)

composer create-project --prefer-dist laravel/laravel blog "5.3.\*"

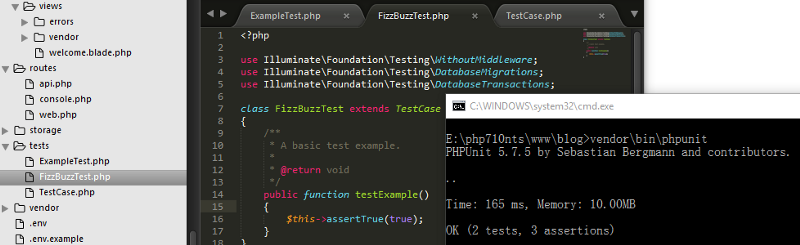
### เริ่มต้นที่การทำ API test

โดยการสร้าง Test Case มาดังนี้

php artisan make:test FizzBuzzTest

ทดลอง run Test

.\vendor\bin\phpunit



หากสั่ง phpunit โดยไม่มีการระบุ class phpunit จะทดสอบ Test ทั้งหมดใน folder tests ซึ่งตอนนี้ ระบุ assertTrue(true); อยู่ จึง test ผ่านทั้งหมด ดังรูปที่แสดงข้างบน

คราวนี้ลองมาเขียน test fizz เอา case ง่ายที่สุด (test method ใน phpunit จำเป็นต้องขึ้นด้วย testXxxxMethod() )

use App\Http\Controllers\FizzBuzzController;

class FizzBuzzTest extends TestCase

{

public function testInput3shouldReturnFizz()

{

$fizzBuzz = new FizzBuzzController();

$result = $fizzBuzz->process("3");

$this->assertEquals($result, "Fizz");

}

}

### สร้าง controller FizzBuzz

php artisan make:controller FizzBuzzController

จากนั้นสร้างเขียน controller เพื่อทำให้ Test ผ่านดังนี้

class FizzBuzzController extends Controller

{

public function process($number) {

return "Fizz";

}

}

### ทดลอง run test

.\vendor\bin\phpunit tests\FizzBuzzTest.php

จะเห็นว่าเราทำการ hardcode เพื่อ return ค่า Fizz ไปโดยตรง เนื่องจากว่า Test ที่ใช้ทดสอบ มีกรณีเดียว เราก็เขียนเพื่อให้ตอบสนองเฉพาะกรณี คราวนี้ พอมีกรณี 5 เพิ่มขึ้นมา เราก็เขียนใหม่เพิ่มไป …

public function testInput5shouldReturnBuzz()

{

$fizzBuzz = new FizzBuzzController();

$result = $fizzBuzz->process("5");

$this->assertEquals($result, "Buzz");

}

และ แก้ code เป็นดังนี้

class FizzBuzzController extends Controller

{

public function process($number) {

if ( $number % 3 == 0)

return "Fizz";

return "Buzz";

}

}

… เราก็เพิ่ม test และ แก้ code ไปเรื่อย ๆ … จนสุดท้าย สิ่งสำคัญคือ เราค่อย ๆ เติม test เข้ามา และ ก็ใช้ test นำการเขียนโปรแกรมไปเรื่อย ๆ เขียนเสร็จแล้วก็ทดสอบระบบ

## 

## 

## Web-UI Testing

### เริ่มเขียน GUI

Update controller โดยเพิ่ม 2 ฟังก์ชันนี้เข้าไปใน FizzBuzzController.php

public function index() {

return view('index');

}

public function fb(Request $request) {

$result = $this->process($request->number);

return view('index')->with('result',$result)

->with('number',$request->number);

}

แก้ไข route.php

Route::get('/fb','FizzBuzzController@index');  
 Route::post('/fb','FizzBuzzController@fb');

สร้าง resources\views\index.blade.php

<div class="container">

<h1 class="text-xs-center">โปรแกรมฟิซบัส</h1>

<form action="/fb" method="POST" role="form">

<div class="form-group">

<label for="">Number:</label>

<input type="number" class="form-control" id="number" name="number" required value="{{ $number or '' }}">

{{ csrf\_field() }}

</div>

<button type="submit" name="submit" class="btn btn-primary">Submit</button>

</form>

<br/><br/>

<h2>Result: {{ $result or '' }} </h2>

</div>

* เขียน HTML ให้ถูกครบ และ อ้างอิง CSS ให้ครบ

ทดลองเรียกหน้า web site อีกครั้ง ก็จะมี Fizz Buzz Web Page เป็นอันเสร็จพิธี

### 

### เขียน Web Test

เริ่มจากสร้าง FizzBuzzWebTest ขึ้นมาใหม่

php artisan make:test FizzBuzzWebTest

จากนั้นเขียน test ที่ทดสอบดังนี้

public function testIndexPage()

{

$this->visit('/fb')

->see('Result')

->dontSee('FizzBuzz');

}

เป็นการเรียกหน้า /fb โดย Method GET แล้วตรวจสอบว่ามีข้อความ “Result” อยู่ในหน้าแรกหรือไม่ ถ้าการทดสอบผ่าน ในหน้า /fb จะต้องมีข้อความ Result และ จะต้องไม่มีข้อความ FizzBuzz ในหน้าแรก

การทดสอบข้อความเป็น case-insensitive (FizzBuzz มีค่าเท่ากับ FIZZBUZZ)

ลองทดสอบเพิ่มเติม โดยการจำลองการป้อน 3 ใน input text และหลังจากกด submit แล้ว จะต้องเจอข้อความ “Buzz” จะเห็นว่าข้อความที่เราใช้ตรวจสอบจะต้องไม่ซ้ำกับข้อความที่มีอยู่แล้วในหน้า Index (ในตัวนี้เลยตั้งชื่อว่า โปรแกรมฟิซบัส)

public function testInput3shouldReturnFizz()

{

$this->visit('/fb')

->type('3','number')

->press('submit')

->see('Buzz');

}

### ทดสอบกรณีอื่น ๆ เพิ่มเติม

public function testInput5shouldReturnBuzz()

{

$this->visit('/fb')

->type('5','number')

->press('submit')

->see('Buzz');

}

public function testInput7shouldReturnOK()

{

$this->visit('/fb')

->type('7','number')

->press('submit')

->see('OK');

}

public function testInput15shouldReturnFizzBuzz()

{

$this->visit('/fb')

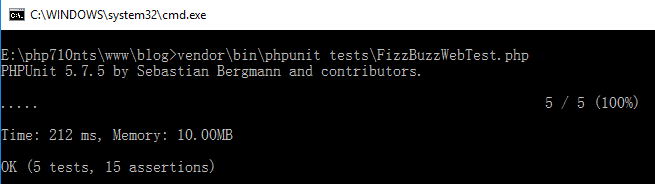
->type('15','number')

->press('submit')

->see('FizzBuzz');

}

ทดสอบการทำงาน



หลังจากนี้ หากมีการ debug program หรือเพิ่ม คุณสมบัติใหม่ อื่น ๆ เพิ่มเติม เราก็สามารถเรียกใช้ชุดทดสอบ เพื่อยืนยันว่าโปรแกรมเรายังสามารถทำงานได้ถูกต้อง ตามหลักการของ Test Driven Development (TDD) !!

### 

### 

### Laravel Dusk (For Laravel 5.4)

composer require laravel/dusk  
register “provider” in config/app.php (Laravel\Dusk\DuskServiceProvider)  
php artisan dusk:install  
php artisan dusk:make FizzBuzzTest  
Check: APP\_URL (in .env)  
php artisan dusk (run test)

## 

## PHP Code Coverage

เป็นเครื่องมือที่เอาไว้ตรวจสอบว่า Test ที่เราเขียนไว้ครอบคลุม Code ของเราเท่าไร ซึ่ง Laravel ที่ติดตั้งไว้ได้ รวม /vendor/phpunit/php-code-coverage/ ไว้แล้ว

### Configuration

วิธีการใช้งาน ให้เพิ่ม configuration ในไฟล์ phpunit.xml ดังนี้

<phpunit>

…

<logging>

<log type="coverage-html" target="./public/report" charset="UTF-8"

yui="true" highlight="true"

lowUpperBound="50" highLowerBound="80" />

</logging>

</phpunit>

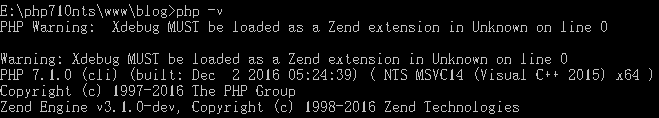
จาก configure นี้ เป็นการกำหนดให้หา coverage test โดยเก็บ output ที่ /report

เมื่อสั่ง vendor/bin/phpunit tests/FizzBuzzTest.php จะเกิด Error: no code driver avaliable

### Require Xdebug library

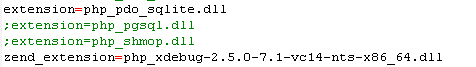
เพราะว่า php7.1 ที่ load มา ไม่ได้ enable Xdebug มาด้วย

ตรวจสอบโดยสั่ง php -v

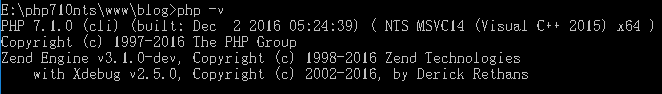


ให้ enable xDebug ดังนี้

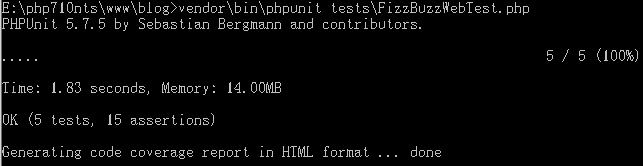
* download: <https://xdebug.org/files/php_xdebug-2.5.0-7.1-vc14-nts-x86_64.dll> (Non ThreadSafe/ 64 bits)
* Copy ใส่ใน php extension folder: (E:\php710nts\ext)
* แก้ไข php.ini โดยเพิ่มในส่วนของ extension (ต้องใช้คำว่า zend\_extension)



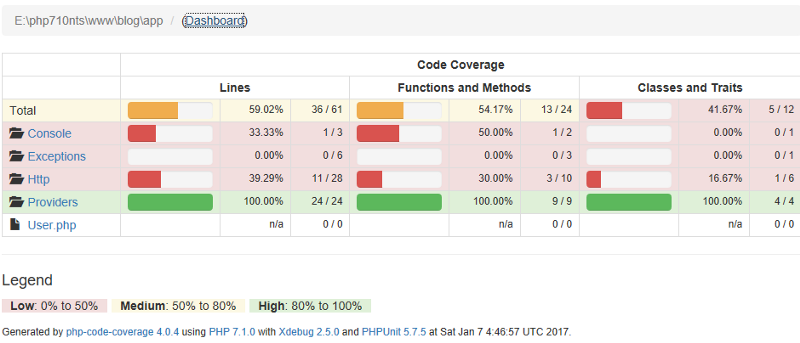
ตรวจสอบ โดยสั่ง php -v อีกที Xdebug enable เรียบร้อยแล้ว



### ทดลอง run test

จากนั้นให้ run test อีกรอบ

จะได้ HTML output ใน folder /report



# Continuous Integration / Continuous Delivery (CI/CD)

## Jenkins

### Installation: Jenkins 2.39 on Debian 8.6

Jenkins เป็น Continuous Integration (CI) ตัวหนึ่งที่เป็นที่นิยม ใช้ในการทำ automation ไม่ว่าจะเป็นการ pull code มาจาก GitHub, Compile, Test และ Deploy ขึ้น production server

Jenkins ไม่ได้เป็น standard package ที่ติดมากับ debian เราต้องแก้ไข sourcelist ดังนี้

1. load key ที่ใช้มาก่อน

wget -q -O -<https://pkg.jenkins.io/debian/jenkins.io.key> | sudo apt-key add -

2. แก้ไข /etc/apt/sources.list โดยการเพิ่มบรรทัดนี้เข้าไป

deb<https://pkg.jenkins.io/debian> binary/

3. สั่ง update source list และติดตั้ง jenkins

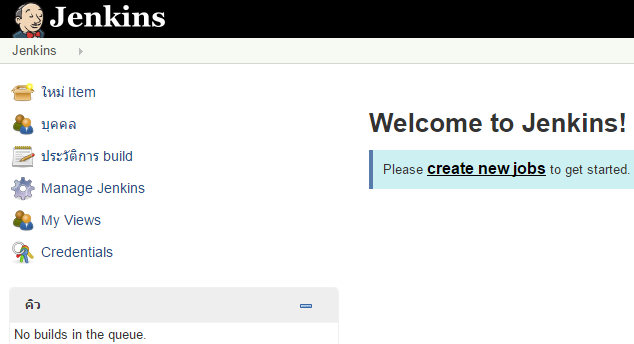
sudo apt-get update

sudo apt-get install jenkins

เมื่อติดตั้งเสร็จเรียบร้อยแล้ว jenkins จะถูก start อัตโนมัติที่ port 8080 เข้าไปที่ <http://server_ip:8080/> การเข้า jenkins ในครั้งแรก jenkins จะให้ใส่ adminPassword ซึ่ง อยู่ที่ /var/lib/jenkins/secrets/initialAdminPassword

ติดตั้ง plug-in ที่จำเป็นต้องใช้งาน สร้าง user/password สำหรับ admin เป็นอันเสร็จพิธี

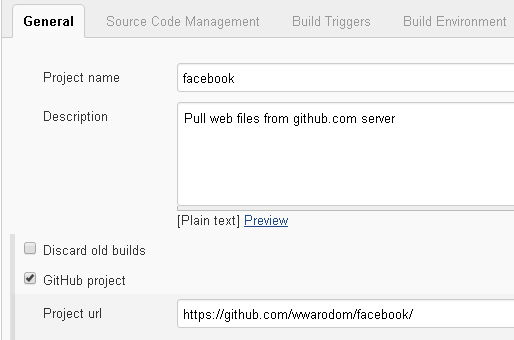
เริ่มใช้งาน jenkins หน้าแรก จะมีหน้าตาแบบนี้



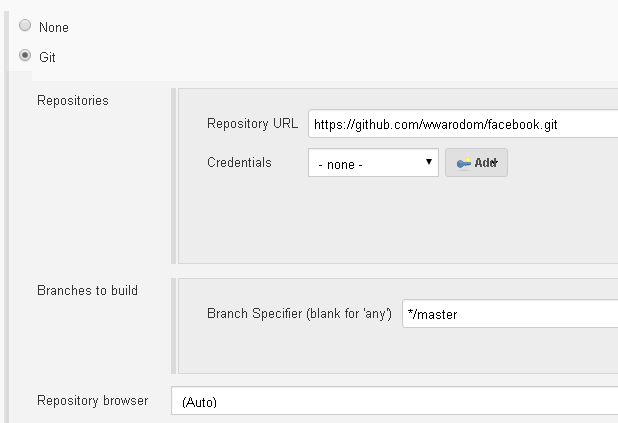
สามารถติดตั้ง plugin อื่น ๆ เพิ่มเติมได้ที่ Manage Jenkins -> Manage plugins แล้วเลือกลงได้ตามอัธยาศัย ที่แนะนำสำหรับ Laravel คือ Checkstyle, phploc, Clover PHP, HTML Publisher, Warnings เป็นต้น (เพิ่มเติม: <http://jenkins-php.org/installation.html>) ทั้งนี้ server จะต้องมี Tool ติดเหล่านี้ ติดตั้งอยู่ใน PATH ที่ jenkins สามารถอ้างอิงถึงได้

### Jenkins GitHub Service Integration (CI)

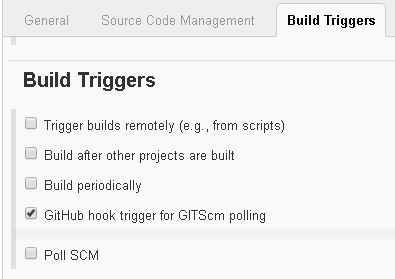
* ที่หน้า Jenkins ให้เลือก New Item (Jenkins job) -> ป้อนชื่อ Project ที่ต้องการ แล้วเลือก Freestyle project กดปุ่ม OK
* Configure jenkins job



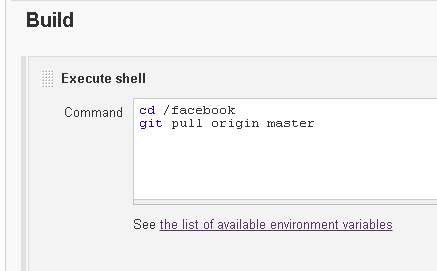
กรอกข้อมูลอธิบายโครงงาน และ ใส่ Link Github



.ใส่ GitHub Repository ถ้าเป็น public repository ไม่จำเป็นต้องใส่ Credentials



เลือกให้ Jenkins Build เมื่อมีการ commit github hook (กรณีนี้คือ commit master branch)



เมื่อมีการ notify มาจาก github ก็สั่งให้ build code มา

* ตรงนี้ อาจจะเพิ่ม test, packaging, ก่อนจะสั่ง deploy อีกทีก็ได้ แล้วแต่ความต้องการของ project นั้น ๆ

### 

#### Jenkins permission และ Local user

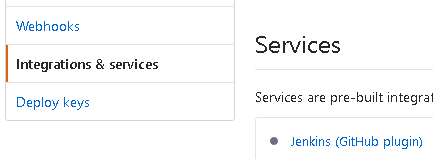
services ของ Jenkins จะใช้ user ที่ชื่อว่า jenkins และ group jenkins ในการทำงาน ดังนั้น หากมีการ เขียนไฟล์ใด ๆ ที่ไม่ใช่ user jenkins เป็นเจ้าของ ก็จะมีปัญหาเรื่อง permission ในกรณีนี้ สมมติว่า local folder มีเจ้าของชื่อว่า wwarodom.wwarodom ผมเปลี่ยน group ของ folder ที่จะทำงาน ให้อยู่ในกลุ่ม jenkins และ กำหนดสิทธิ์ให้ group นั้นสามารถเขียนไฟล์ได้ ดังนี้

chown wwarodom:jenkins -R /folder

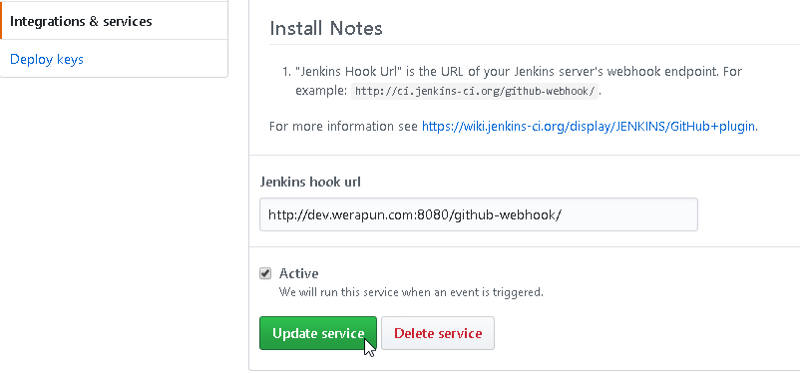
chmod g+rw -R /folder

#### Github Setting for Jenkins Hook URL

ทีแรกผมพยายาม configure Github ในส่วนของ web hook (เพราะมันบอกว่า external services will be notified) ที่อยู่เหนือ Integrations & Services



แต่จริง ๆ แล้ว การ Github ไป Trigger services อื่น ๆ (เช่น Jenkins) เราต้อง ไป configure ที่ Integrations & Services ดังนี้



ป้อน URL เป็น Jenkins Server และ ตามด้วย /github-webhook/

#### 

#### ทดสอบการทำงาน

โดยการ commit งานใน folder ปัจจุบัน ขึ้น branch master

git remote add origin<https://github.com/wwarodom/facebook.git>

git add .

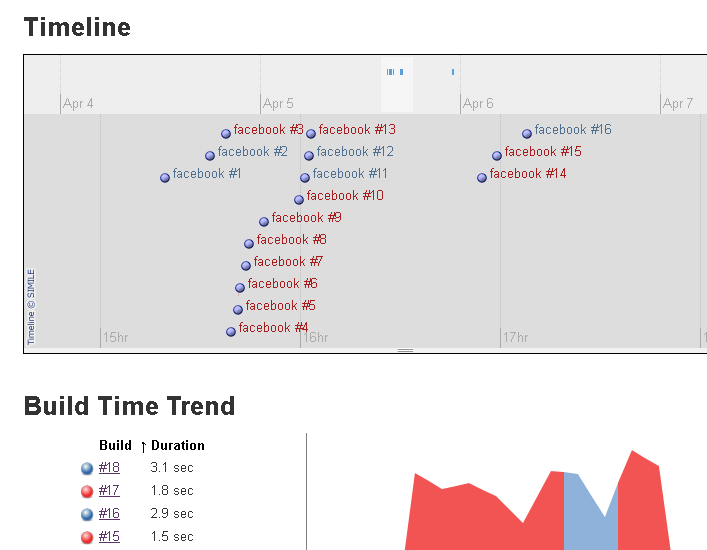
git commit -m "test "

git push -u origin master

จากนั้น Jenkins server ก็จะสั่ง Build โดยอัตโนมัติ



Build ผ่านแล้ว



build ไปหลาย step กว่าจะผ่าน

### Jenkins User/Port Configuration

หากต้องการเปลี่ยน port: 8080 หรือ user: jenkins ที่ใช้ในการ start jenkins สามารถมาปรับแต่งได้ที่ /etc/default/jenkins และ สั่ง service jenkins restart

### Jenkins ไม่ทำงาน เพราะ RAM หมด

Jenkins นี้ run บน Digital Ocean VPS server ที่ RAM 512MB โดยที่มี Docker service ที่ run mysql, nginx, php พอ run jenkins อีกตัว สถานะเลยขึ้นเป็น Active (exit) เลย (ทำให้ jenkins down และไม่มี Error log อื่น ๆ) เนื่องจาก RAM หมด ถ้าจะ start Droplet อีกตัว ก็ต้องจ่ายเงินเพิ่มอีก ก็เลยยอมทำงานช้าหน่อย แต่ไม่ต้องสร้าง Droplet เพิ่ม โดยการสร้าง SWAP ไฟล์ ตามวิธีการของ Linux สมัยที่ RAM ยังมีน้อยดังนี้ => <https://www.digitalocean.com/community/tutorials/how-to-add-swap-space-on-ubuntu-16-04>

# 

# 

# 

# Moby Containerize

## Moby

### Why Moby?

(ยาวไปไม่อ่าน) การใช้ Moby ดีกว่าการไม่ใช้ Moby

### Installation: Moby on Debian

1. ติดตั้งตั้ง curl

apt-get update && apt-get install -y curl

1. ติดตั้ง Moby โดยใช้ curl

curl -fsSL https://test.docker.com/ | sh

1. ทดสอบว่าสามารถใช้งานได้ด้วย Hello world

docker run --rm test hello-world

1. ติดตั้ง Docker Compose

curl -L https://github.com/docker/compose/releases/download \  
 /1.13.0/docker-compose-`uname -s`-`uname -m` > \  
 /usr/local/bin/docker-compose  
chmod +x /usr/local/bin/docker-compose

### Jenkins & Docker Configuration

#### ย้าย jenkins มารันบน docker

1. สร้าง compose file

version: ‘3’  
services:  
 jenkins:  
 image: jenkins  
 user: root  
 ports:  
 - 8080:8080  
 volumes:   
 - <PATH TO known\_hosts file>:/root/.ssh:ro  
 - <JENKINS\_HOME path>:/var/jenkins\_home  
 - /var/run/docker.sock:/var/run/docker.sock  
 container\_name: jenkins

อธิบาย

* version: เป็นการระบุว่า compose file ใช้เวอร์ชัน 3
* services: เป็นการระบุว่าส่วนของ services
* jenkins: เป็นการกำหนดรายละเอียดของ service ชื่อ jenkins
* image: สร้าง container จาก image ไหน
* user: กำหนดให้ใช้ user ใดในการรัน service
* ports: ระบุว่าจะ publish port ใด <Host>:<Container>
* volumes: Mount file จาก host ให้กับ container <Host>:<Container>
* container\_name: ตั้งชื่อ container

ข้อควรระวังในการสร้าง compose file ห้ามใช้ Tab ให้ใช้ spacebar เท่านั้น

1. รัน jenkins ขึ้นมาทำงาน

docker-compose up -d

1. เข้าไปที่ http://server\_ip:8080/ เพื่อตรวจสอบความเรียบร้อย

#### ติดตั้ง plugins ที่ต้องการใช้เพิ่ม

ติดตั้ง plugins ที่ต้องการใช้เพิ่มเติมโดยในที่นี้ จะติดตั้ง plugins ตามรายการด้านล่าง

1. CloudBees Docker Traceability
2. CloudBees Docker Build and Publish plugin
3. CloudBees Docker Custom Build Environment Plugins
4. CloudBees Docker Hub/Registry Notification
5. GitLab Authentication plugin
6. GitLab Plugin

#### เพิ่ม Credentials ที่จำเป็นต้องใช้

เพื่อให้ Jenkins สามารถทำงานได้ จำเป็นต้องมีการเพิ่ม Credentails เพื่อใช้ในการยืนยันตัวตนกับ service ต่าง ๆ ดังนี้

1. GitLab/GitHub Credentail สำหรับเชื่อมต่อ git
2. Git ssh\_key สำหรับใช้ เชื่อมต่อ git
3. Docker Hub สำหรับใช้ในการ Publish image

#### เตรียม Repository บน Docker Registry

1. เข้าไปที่ https://hub.docker.com พร้อมลงทะเบียนให้เรียบร้อย
2. Click Create Repository แล้วตั้งค่าตามความต้องการ

#### สร้าง Jenkins job

* ที่หน้า Jenkins ให้เลือก New Item (Jenkins job) -> ป้อนชื่อ Project ที่ต้องการ แล้วเลือก pipeline กดปุ่ม OK
* Configure jenkins job

#### ทดสอบการทำงาน

โดยการ commit งานใน folder ปัจจุบัน ขึ้น branch master

git checkout master

git add .

git commit -m "test"

git push

จากนั้น Jenkins server ก็จะสั่ง Build สร้าง Image และ upload ขึ้น Registry ให้โดยอัตโนมัติ

https://goo.gl/Jqcjms