Workshop Git

Overzicht van de gedachtengang van deze tutorial

1. Git is erg nuttig
2. Het is dus handig ergens een Git repository te kunnen aanmaken; normaal heb je zowel een Git repository lokaal als op een externe server, omdat je lokale computer kapot kan gaan of gestolen kan worden. En voor samenwerking met anderen is een remote Git repository gewoon noodzakelijk als je niet met z'n allen op 1 laptop wilt typen…
3. Één van die externe servers is GitHub (andere zijn bv BitBucket van Atlassian, en CodeCommit op AWS; CodeBerg als FOSS-alternatief, verder Gitea en GitLab - al zijn die twee meestal Self-Hosted (anders moet je betalen…))
4. Het doel van deze oefening is leren een Git repository aan te maken op GitHub, het te kunnen gebruiken voor updates, en te leren om te gaan met problemen die je tegenkomt als je in een groep werkt en/of per ongeluk bugs naar de hoofdbranch pusht.
5. Ook zal ik ingaan op hoe je een GitHub-repository voor een groep kunt managen.

Hoofdpunten om te leren/oefenen

1. Hoe Git in een groep te managen, dat wil zeggen:
    - met branch-beveiliging
    - met pull requests
2. Hoe om te gaan met conflicterende commits / merge conflicts
3. Hoe om te gaan met fouten in je eigen branch (git reset)
4. Hoe om te gaan met fouten die al naar main gepusht zijn (git revert)
5. Hoe branches te managen (dat je ze ook delete)
6. Een niet-geneste monorepo te maken

Noot: Deze tutorial is voor Windows geschreven: als je Linux gebruikt, is veel hetzelfde, maar moet je de equivalenten van de File Explorer en de Command Prompt en Notepad gebruiken.

Stap 1: een GitHub repository aanmaken

Noot: vaak maak je een nieuw project in IntelliJ, en van daaruit commit je naar GitHub. Dat werkt natuurlijk, maar nu doen we het op een andere manier, ook omdat dat handiger is als je een fullstack project maakt met aparte frontend en backend folders.

1. Ga naar je GitHub account
2. Ga naar Repositories
3. Druk op de `New`-knop
4. Maak een nieuwe *public* repository, "shoppinglist"
5. Als dat is aangemaakt kijk ik naar de optie "…or create a new repository on the command line"
6. …Maar het kan simpeler

Reflectie: is er een stap hier die onverwacht voor je was?

Stap 2: Het repository synchroniseren naar jouw computer

Noot: Je hebt natuurlijk altijd TWEE repositories nodig; één op de Git server (in dit geval GitHub) en één op je locale machine. En die twee moeten met elkaar gesynchroniseerd worden. Dat kàn met alle Git-commando's die GitHub geeft. Maar het kan ook eenvoudiger.

1. kopieer de link die je bovenin je repository ziet (iets als `https://github.com/${MIJN_GITHUB_HANDLE}/shoppinglist`. In mijn geval dus `https://github.com/EWLameijer/shoppinglist`)
2. Ga via de Explorer naar de directory waarin je het repository wilt zetten
3. In de adresbalk van de Explorer, druk rechts van het path (bijvoorbeeld `C:\Personal\JavaProjects`). Als het goed is zie je het filepath nu als tekst, als blauw geselecteerd.
4. type "cmd" in en druk op `Enter`; dat opent de command prompt.
5. Gebruik nu `git clone REPOSITORYNAAM`, in mijn geval dus `git clone https://github.com/ewlameijer/shoppinglist`
6. Git waarschuwt dat het repository leeg lijkt te zijn. Maar dat is geen probleem.
7. Ga nadat je gecloond hebt, ga naar je nieuwe lokale repository met `cd shoppinglist` (`cd` betekent 'change directory')

Reflectie: Check de stappen hierboven nog eens; begrijp je *waarom* je elke stap hebt gedaan en wat elke stap doet? Waren en stappen/'trucs' die je nog niet kende?

Stap 3: folders aanmaken en je eerste commit

1. Kijk voor de lol eens met de File Explorer naar je nieuwe directory. Zie je de .git directory (mogelijk moet je hidden files laten zien in de viewer)? Kijk eens 3 minuten in de .git directory en probeer te raden waar de verschillende directories en files voor zijn.
2. op de command prompt (of in de explorer) maak twee directories, `frontend` en `backend`. Maak ook een README.md met als tekst “Shopping list Git demo” (het is makkelijk die file te maken door `notepad README.md` in te typen)
3. Op de command prompt: voer uit `git status`. Je ziet nu vier dingen:
    - dat je op branch `main` bent
    - dat er nog geen commits zijn
    - dat er één ‘untracked file’ is, README.md
    - dat er niets is toegevoegd aan de commit maar er untracked files aanwezig zijn.
4. Zoals je mogelijk weet of kunt raden, betekent ‘untracked file’ zoiets als dat Git een file ziet, maar niet weet wat hij ermee moet; de file zit niet in het repository als onderdeel van een commit, maar de file wordt ook niet ignored.
5. We gaan de file adden aan de toekomstige commit, we volgen de instructie (use “git add” to track)
6. We doen dus "git add README.md" - dit addt, trackt of staget de file, afhankelijk van hoe je het wilt noemen. Hoe dan ook, het betekent dat README.md meegaat naar de volgende commit.
7. Merkte je overigens ook op dat Git niks zegt over de backend en frontend directories?
8. We doen git commit -m "Adding README.md" (-m betekent: geef als message het volgende - elke commit *moet* namelijk een commit message hebben)
9. We zien iets als

```
[main (root-commit) f46fa66] Adding README.md
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

Kun je raden wat al deze dingen betekenen? Vermoedelijk wel, f46fa66 is de hashcode van de commit.

Reflectie: Check de stappen hierboven nog eens; begrijp je *waarom* je elke stap hebt gedaan en wat elke stap doet? Waren en stappen/‘trucs’ die je nog niet kende?

Stap 4: we gaan pushen naar GitHub

Noot Je hebt weinig aan een versiebeheersysteem als je alleen lokaal versies kunt beheren. Allereerst ben je dan niet beveiligd tegen verlies of computerdefecten, maar ten tweede kan je dan niet samenwerken met anderen! Tijd dus om je lokale repository met je online repository te synchroniseren.

1. Check je online repository. Zie je daar je files al staan?
2. Inderdaad, ze staan er nog niet. Op de command prompt, doe `git push`
3. Je ziet allemaal tekst als

```
git: 'credential-manager-core' is not a git command. See 'git --help'.
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 252 bytes | 252.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/EWLameijer/shoppinglist
 * [new branch]      main -> main
```

Best interessant om eens door te lezen.

4. Check opnieuw! F5 in de browser laat nu je repository zien! Waar overigens nog steeds geen folders `frontend` of `backend` in staan. **Belangrijke les**: Git slaat alleen files met hun paden op, *geen* losse/lege directories.
5. Delete de backend folder, maak een standaard Java-project ShoppingList via de Spring initialzr (JPA en Postgres hoeven niet, dit gaat even voor de demo. Web is wel een handige dependency hier, misschien met Lombok), en zorg dat die in de hoofdfolder van het repository komt (waar de README staat)
6. Hernoem de hoofd-Java-folder (die bij mij shoppinglist heet) tot `backend`.
7. Maak een eenvoudige controller waarvan het GET-endpoint een lijst van “boter”, “kaas” en “eieren” teruggeeft, die, voor de oefening straks, op verschillende regels staan. Bij mij ziet de code er zo uit:

```
@RestController
@RequestMapping("api/v1/shopping-list-items")
public class ShoppingListController {
    @GetMapping
    public List<String> get() {
        return List.of(
                "boter",
                "kaas",
                "eieren"
        );
```

```
    }
}
```

8. Run de code en test het endpoint.
9. Run weer op de command prompt “git status”: wat zie je?
10. Je ziet in elk geval dat de backend folder zijn eigen .gitignore heeft. Dat is eigenlijk best netjes, dan zijn de ignores van Java en JavaScript gescheiden.
11. Je KUNT alle untracked files toevoegen met git add, maar het kan ook via IntelliJ (Alt+0). Als je dubbelklikt op files kan je zien hoe ze eruit zien of wat er veranderd is, wat vaak verstandig is- in elk geval bij codefiles.
12. Let wel: zelf vind ik Git Extensions (https://gitextensions.github.io/) handiger omdat ik dan niet hoef te dubbelklikken om de changes in een file te bekijken. Anderen gebruiken SmartGit, SourceTree of GitHub Desktop. Het is denk ik een goede investering om eens een uur of twee te besteden aan het downloaden en vergelijken van die vier alternatieven. Maar dat gaan we nu niet doen!
13. De Spring Initializr maakt overigens een heel goede .gitignore-file, als je *niet* de Spring initialzr gebruikt (bijvoorbeeld als je met een eenvoudig Java-project bezig bent), onthoud dan:

- WEL: source files (.java) en alle Maven en Gradle files (ook de Maven/Gradle .jar-files). Verder ongeveer alle tekstfiles die er zijn (zoals de application.properties, .md files, .xml files, etc.)
- NIET: de .class files (of out/results folder) of .jar files buiten die van Maven of Gradle. Niet de specifieke editor (.idea) files.

14. Ik check alle files (ik doe dat dus met Git Extensions, maar gebruik gerust IntelliJ daarvoor), commit met als message iets als ‘Backend works’, en push.
15. Check op GitHub: zie je nu wel de backend folder? Als het goed is wel.

Reflectie: Check de stappen hierboven nog eens; begrijp je *waarom* je elke stap hebt gedaan en wat elke stap doet? Waren en stappen/‘trucs’ die je nog niet kende?

Stap 5: Nu nog even een frontend

Noot: We maken een frontend. Zowel voor de lol, als om te oefenen met een 1-repo-2-folder-structuur, die best nuttig is. Als je verschillende teams hebt heb je vaak 2 of meer aparte repositories, maar de 1-repo, 2-folders (die NIET genest zijn) is vaak een best wel prettig werkend alternatief!

1. Je kent mogelijk de drill: je delete de frontend folder, je gaat naar de hoofdfolder van de app en maakt met `npm create vite@latest` een Vite React-JS+SWC (of TS+SWC)-project dat je frontend noemt, volgt alle stappen, en vervang dan de App.jsx door iets als:

```
import { useEffect, useState } from 'react'

const App = () => {
  const [items, setItems] = useState([])

  useEffect(() => {
    fetch("http://localhost:8080/api/v1/shopping-list-items").then(result => result.json().then(setItems))
  })

  return <ul>{items.map(item => <li key={item}>{item}</li>)}</ul>;
}

export default App;
```

(vergeet niet @CrossOrigin("http://localhost:5173") bij de @RestController te zetten en de backend opnieuw te runnen!)

2. Ik check de files (en maak van de gelegenheid gebruik om de .css files en .svg files en de imports daarvan te verwijderen), commit dan met een message als “Frontend works” en push.
3. Check dat er nu ook een frontend folder op GitHub staat!

Reflectie: Check de stappen hierboven nog eens; begrijp je *waarom* je elke stap hebt gedaan en wat elke stap doet? Waren en stappen/‘trucs’ die je nog niet kende?

Stap 6: Aan de slag met branches en branche-beveiliging!

Noot: Git branches zijn onnodig voor eenvoudige projecten en worden zelden gebruikt door beginners. Maar een iets gevorderder programmeur zal vaak branches aanmaken voor experimenten, en als je in een groep werkt zijn aparte branches meestal noodzakelijk (al gebruiken sommige groepen maar één branch, maar dan moet iedereen wel héél goed programmeren en moeten er veel beveiligingsmechanismen gemaakt zijn!) Nu heb ik tot nog toe alles gedaan in de main branch. Dat gaan we veranderen. En dat kan ook met GitHub!

1. Ik ga naar GitHub > Settings > Rules > Rulesets > New ruleset
2. Ik maak een nieuwe ruleset (ik noem hem `mainbranch`), zet die op active, en verander de settings: ik require een pull request before merging, met 1 approval, and conversation resolution before merging, en ik add als target branch de default branch

Noot: doe dit *alleen* voor groepsprojecten! Voor persoonlijke projecten zijn er geen baten en alleen maar nadelen aan zulke extra beveiliging! Maar we oefenen nu voor het komende groepsproject!

3. Ik druk Create (en moet dan authenticeren)
4. Ik maak een triviale verandering in mijn code (bijvoorbeeld een commentaarregel toevoegen), en probeer weer te pushen: ik krijg een foutmelding: `changes must be made through a pull request`
5. Ik ga weer naar de command prompt. In de hoofdfolder doe ik een `git checkout -b minor_changes` (al zou ik ook een nieuwe branch aan kunnen maken via IntelliJ of Git Extensions). Ik doe een `git push`
6. Git klaagt:

```
fatal: The current branch minor_changes has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin minor_changes

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

Ik doe dus wat Git vraagt: `git push --set-upstream origin minor_changes` (of liever`git push -u origin minor_changes`, dat doet hetzelfde maar is dus wat korter)

7. In GitHub zie ik nu op het hoofdscherm van mijn repo: “minor_changes had recent pushes 46 seconds ago”. Ik druk op de “Compare & pull request” knop ernaast. Ik voeg een description toe, en druk op “Create pull request”

8. Ik ga naar de “Files Changed”", check de boxen van elke gewijzigde file aan als “Viewed”, druk op “Review changes”, schrijf iets als review en druk dan op “Submit review”. Helaas kan ik het review niet approven, dus hoewel zulke branch protection handig is voor echte groepsprojecten, ga ik voor deze oefening terug naar de ruleset en zet `required approvals` op 0.

9. Nu kan ik het pull request mergen, en de merge confirmen.

10. En nu delete ik de branch, zowel op GitHub via de knop die nu verschijnt, als lokaal via mijn Git GUI of via de command line.

11. Maar als ik het in de command line doe met `git checkout main` krijg ik

```
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
```

En natuurlijk kan ik niet pushen, dat was juist verboden! Ik doe dus een `git pull`. Tenslotte de `git branch -d minor_changes`

Reflectie: Check de stappen hierboven nog eens; begrijp je *waarom* je elke stap hebt gedaan en wat elke stap doet? Waren en stappen/‘trucs’ die je nog niet kende?

Stap 7: Een merge-conflict veroorzaken: deel 1

Noot: Als je in je eentje programmeert werkt Git meestal vlekkeloos. Maar in een team kunnen merge-conflicten ontstaan, zeker als er òf weinig files zijn òf pull requests groot zijn en veel files omvatten. Meestal kan je met een goede taakverdeling, een goede architectuur en taken die klein genoeg zijn (een half uur tot 2 dagen) bijna alle mergeconflicten vermijden. Maar dus niet allemaal - dus moet je leren hoe je ermee om moet gaan!

1. De koster komt op bezoek – ik heb een andere lijst nodig. Ik maak dus een branch `de_koster_komt` (ehm… `the_sexton_is_coming`) en ga daarnaar toe.
2. Zoals je misschien weet, lust de koster geen eieren, maar wel spek (https://www.oudersenzo.nl/bim-bam-beieren/) vervang de
   “eieren” in je lokale ShoppingListController naar “spek”
3. Doe op de command prompt `git status`. Je ziet nu iets als

```
On branch the_sexton_is_coming
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   backend/src/main/java/org/ericwubbo/shoppinglist/ShoppingListController.java

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

4. Dit geeft een goede hint: ik zal nu git commit -am gebruiken om die wijziging toe te passen en gelijk te committen `git commit -am "The sexton does not like eggs"`
5. Dan `git push -u origin the_sexton_is_coming`
6. Kijkend op mijn Code pagina van GitHub zie ik inderdaad dat ik nu twee branches heb.
7. Ik ga naar de `the_sexton_is_coming` branch en check dat die inderdaad geen eieren meer bevat, maar spek!

Reflectie: Check de stappen hierboven nog eens; begrijp je *waarom* je elke stap hebt gedaan en wat elke stap doet? Waren en stappen/'trucs' die je nog niet kende?

Stap 8: Een merge-conflict veroorzaken: deel 2

1. We gaan nu een merge-conflict veroorzaken. Doe alsof je een huisgenoot bent en check de main-branch weer uit.
2. Maak een nieuwe branch aan (healthy_shopping), en schakel daarnaar over.
3. Kijkend in IntelliJ wil je de lijst gezonder hebben! Vervang eieren door "scharreleieren" en voeg biologisch sinaasappelsap toe.
4. Deze branch ga je in master mergen, weer via een pull request…
5. En dan delete je de branch op GitHub en lokaal.
6. Check op GitHub dat de healthy_shopping-branch echt niet meer bestaat, maar dat main wel de veranderingen heeft!

Stap 9: Een merge-conflict veroorzaken: deel 3

1. Als mezelf ga ik terug naar de sexton-branch. (ik was even vergeten hoe die precies heette, maar met `git branch` zie ik alle branches) Ik wil ook wat voor bij de koffie voor met de koster, en voeg kletskoppen toe aan de lijst. Ik maak een commit. Met `git status` zie ik dat ik 1 commit ahead ben van de `origin/the_sexton_is_coming`.
2. Dan check ik bij de vrouw van de koster – en hij blijkt geen kletskoppen te lusten! Ik ga in mijn GUI naar de laatste goede commit en doe een harde reset. Of, als ik het via de commandline wil: `git log` om alle commits te zien, ik krijg iets als

```
commit 748548de6905aadf3c32c18d3c5d0adecd633d0e (HEAD -> the_sexton_is_coming)
Author: Eric-Wubbo Lameijer <[email protected]>
Date:   Thu Dec 14 22:01:00 2023 +0100

    for the coffee!

commit faf5d1a9d2524a384613838452304d6fbabd5007 (origin/the_sexton_is_coming)
Author: Eric-Wubbo Lameijer <[email protected]>
Date:   Thu Dec 14 21:47:07 2023 +0100

    The sexton does not like eggs

commit 65daee05d96fa582603f0aecdfc966bf4af1daf8
Author: Eric-Wubbo Lameijer <[email protected]>
Date:   Thu Dec 14 21:38:15 2023 +0100

    Better page title

commit ef3a1b715cf62bed28fa4d2afcd23acccc7215e9
Merge: 7097f09 77fcda9
Author: Eric-Wubbo Lameijer <[email protected]>
Date:   Thu Dec 14 21:32:04 2023 +0100

    Merge pull request #1 from EWLameijer/minor_changes

    Small app changes

commit 77fcda9cd3e9da319781f73ef49113029d6f1993
Author: Eric-Wubbo Lameijer <[email protected]>
:
```

Ik druk op 'q' (dat is het Linux-tekstverwerkerssymbool voor quit)
En `git reset --hard faf5d1` gaat dan terug naar de juiste commit!

3. in de GUI en met `git status` lijkt alles weer okee! Ook in de file zijn de kletskoppen verdwenen.

4. Ik heb voldoende gedaan, vind ik, en ik besluit in main te mergen. Als ervaren programmeur pull ik eerst main (ik check main uit, dan pull ik hem).

5. Het officiële proces, zoals ik weet, is

    - check main uit
    - pull main
    - check de feature-branch uit

    - merge main in de feature branch
    - los eventuele merge-conflicten op
    - check dat de feature-branch nog werkt
    - push de nieuwe feature-branch
    - maak een pull request

6. Ik check nu dus de feature branch uit en merge de main erin met `git merge main`. Maar ik krijg nu

```
Auto-merging backend/src/main/java/org/ericwubbo/shoppinglist/ShoppingListController.java
CONFLICT (content): Merge conflict in backend/src/main/java/org/ericwubbo/shoppinglist/ShoppingListController.java
Automatic merge failed; fix conflicts and then commit the result.
```

7. Ik check IntelliJ: mijn file ziet er nu akelig uit, als

```
@RestController
@CrossOrigin("http://localhost:5173")
@RequestMapping("api/v1/shopping-list-items")
public class ShoppingListController {
    @GetMapping
    public List<String> get() {
        return List.of(
                "boter",
                "kaas",
<<<<<<< HEAD
                "spek"
=======
                "scharreleieren",
                "biologisch sinaasappelsap"
>>>>>>> main
        );
    }
}
```

En *ik* moet zien te onthouden wat hier de bedoeling is! Nu *kan* ik dit editen in IntelliJ, maar het is beter dan in IntelliJ of GitExtensions het op te lossen. In IntelliJ: Het Commit-window heeft nu `Merge Conflicts`, ik klik op `Resolve`, dan `Merge`, druk op de juiste pijltjes en voeg een komma toe zodat ik een geünificeerde lijst krijg met spek, scharreleieren èn biologisch sinaasappelsap. Ik druk op `Apply` en doe een `Commit and Push`. En via een pull request krijg ik het in mijn hoofdbranch.

8. Ik check de hoofdbranch en zie dat het goed is. Volgens mij dan :)

Reflectie: Check de stappen hierboven nog eens; begrijp je *waarom* je elke stap hebt gedaan en wat elke stap doet? Waren en stappen/‘trucs’ die je nog niet kende? Ook: als er een ‘harde’ reset is, wat voor andere resets zijn er dan? En wanneer zou je die gebruiken?

Stap 9: Waarin we zien dat een harde reset niet alles op kan lossen

Noot Ondanks je beste inspanningen komt er weleens een fout op de hoofdbranch terecht. Als ik me goed herinner was bij Microsoft de regel dat als een developer de eerste keer de hoofdbranch zo brak dat hij niet meer compileerde, hij of zij de hele afdeling op taart moest trakteren. Bij de tweede keer werd hij ontslagen. Bij ITvitae ontslaan we niemand, maar laten we de committer en de reviewer *samen* op taart trakteren. Maar dat terzijde: je kan je hoofdbranch verpesten. Wat dan?

1. Mijn huisgenoot ziet spek op de lijst staan en is woedend, wil die troep eraf. Ik besluit mijn main te resetten op de vroegere waarde (zonder de spek, maar met de scharreleieren) (via de command line dus weer met `git log`, en, in mijn geval `git reset --hard 77b4972f`)

2. Ik probeer weer te pushen! Maar ik krijg

```
git: 'credential-manager-core' is not a git command. See 'git --help'.
To https://github.com/EWLameijer/shoppinglist
 ! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/EWLameijer/shoppinglist'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. If you want to integrate the remote changes,
hint: use 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Dat is ergens ook wel logisch. Stel dat ik wèl zou kunnen pushen en de main zou kunnen overschrijven: als ik een slome updater was die vergeten was de main eerst te pullen voor ik mergde zou het werk van heel wat collega’s verloren gaan! (mocht je overigens denken: niemand is zo dom om zoiets te proberen: ik heb dit een keer of 2 a 3 geprobeerd bij mijn eerste baan met Git, bij IntelliMagic. Ik heb toen meerdere keren een Git-expert van een andere groep moeten raadplegen omdat ik niet wist hoe ik dat moest oplossen… Pas veel later leerde ik via de think-like-a-git-site de oplossing van de expert te begrijpen…)

3. Ik doe een git pull om mijn main weer ‘up to date’ te krijgen.

4. Omdat ik hier met pull requests moet werken, maak ik een fix_it branch en ga daarheen. Ik revert de huidige commit, push de branch en maak een pull request, voer die uit en delete de origin/fix_it en lokale fix_it.

5. Op GitHub check ik de code in main (nu weer zonder spek), en in IntelliJ zie ik bij Git log nu de grillige lijntjes lopen. Maar het reverten is gelukt!

Reflectie: Check de stappen hierboven nog eens; begrijp je *waarom* je elke stap hebt gedaan en wat elke stap doet? Waren en stappen/‘trucs’ die je nog niet kende?

Stap 10: Opruimen

1. Is alles gelukt? Verwijder dan het shoppinglist-project van GitHub (dat kan via Settings, onderaan, als je dat toevallig nog niet wist)

Stap 11: Eindreflectie

Kijk eens terug naar de “Hoofdpunten om te leren/oefenen (voor Stap 1)” In hoeverre heb je ze naar je mening gehaald? Kijk eventueel terug naar specifieke onderdelen als je merkt dat je dingen vergeten bent.

Sowieso kan het handig zijn dit de week voor het eindproject nog eens door te scannen…

Nawoord: algemene gedachten over Git

1. Bovenstaande is lang niet alles over Git. Op zijn minst zijn er nog rebase, squash en cherrypick. Beschouw die (en andere tools) als bonussen als je nieuwsgierig bent maar niet nodig voor het tussenproject of eindproject. Het kan wel handig zijn die dingen later eens uitgebreider te bekijken, er zijn teams die (om de commit-history schoner te maken) uitgebreid werken met squash en rebase, hoewel het de meeste teams volgens mij niet voldoende uitmaakt om de standaardmerge te verlaten…
2. Vaker nuttig zijn tags: meestal bookmark je daarmee de commits van belangrijke versies zodat je die makkelijk kunt terugvinden in de codebase. Als je geen tags hebt moet je ergens noteren dat versie 1.0.5 de hash 10284f1e had en 1.0.6 de tag 72feb5 en dat je dus die commits moet vergelijken om de oorzaak te vinden van de bug die in 1.0.6 bleek te bestaan maar niet in 1.0.5. Met tags is dat veel makkelijker!
3. Git stashes bestaan ook; voor als je snel naar een andere branch moet, maar je huidige wijzigingen nog niet wilt committen. Al moet ik zeggen dat ik persoonlijk zo weinig met stashes werk dat ik het makkelijker vind om desnoods een draft-branch te maken en uit te checken en later te mergen; zaken zijn zelden urgent genoeg om iets te moeten stashen. IntelliJ ‘shelft’ overigens in plaats van te stashen. Hoe dan ook, weet dat het bestaat, maar het is naar mijn mening zeker niet het belangrijkste om meester van te worden!
4. Meer informatie: https://think-like-a-git.net/ , https://www.youtube.com/watch?v=1ffBJ4sVUb4 , https://www.atlassian.com/git . Er is ook een groot Git-boek https://git-scm.com/book/en/v2 - waar ik zelf tot nog toe nauwelijks doorheen ben gekomen. Maar ik denk dat voor de gemiddelde developer Git tot en met revert essentieel is, de andere dingen kun je meestal vermijden of leer je van je team als die dat belangrijk vindt.
5. Bij een commit: het is handig te kijken wat je precies gewijzigd hebt door dubbel te klikken op de namen van de gewijzigde files.
   Alternatief kun je ook gemakkelijk(er) de wijzigingen zien via een git-tool als Git Extensions.
   Er zijn een paar vuistregels die nuttig zijn bij committen

- Commit regelmatig (ieder uur, iedere twee uur, of telkens als je naar je gevoel wat functionaliteit hebt toegevoegd). Vaak is 50
  gewijzigde regels code meer dan genoeg voor een commit. Als je de commits kort houdt, is punt 2 ook makkelijker.
- Check je commit op onduidelijke, slordig opgemaakte, foutief gespelde of mogelijk foute code, en corrigeer de fouten meteen,
  dus voordat je commit.
- Commit nooit als je haast hebt of moe bent – dan kan je namelijk niet goed je code controleren. Sommige bedrijven willen dat
  je aan het eind van de dag commit – maar meestal is het geen probleem als je in plaats daarvan elk uur/elke twee uur commit.