howest
hogeschool

Git

Continuous Integration Basics

Graduaat in het Programmeren

INHOUD

1 BRANCHES

1.1 WAT ZIJN BRANCHES?

Kort door de bocht zouden we kunnen stellen dat een nieuwe branch in een repository leidt tot een nieuwe 'kopie' van deze repository. Vervolgens kan je zonder probleem op deze 'kopie' aan het experimenteren zonder hier de originele repository te bevuilen. Met andere woorden, als er iets foutloopt tijdens het experimenteren, kan je simpelweg altijd nog terug naar de originele situatie. Wat in deze beknopte uitleg bijvoorbeeld onze master branch kan zijn.

Maar wat als je dan toch een aantal updates in je nieuwe branch staan hebt die je wel graag wil toevoegen aan de algemene code in de repository? Het is net daar waar Git het ons makkelijk maakt om deze te laten samensmelten. Mocht je later alsnog van mening veranderen kan je ook nog steeds terug naar het punt net voor de samensmelting.

Een branch in Git is dus een **onafhankelijk ontwikkelingspad**. Je kan nieuwe commits aanmaken in een branch zonder dat deze invloed hebben op andere branches. Meer nog, zonder dat deze invloed hebben op het werk van je collega developers!

In de vorige hoofdstukken werkte je reeds op een branch. De **master** branch was de enige branch waar we gebruik van maakten, maar is op zich dus ook reeds een branch! In Git bevind je je standaard in de master branch bij aanvang van je repository. De naam "master" impliceert niet dat deze branch superieur is maar deze is bij conventie het startpunt van een Git repository.

Branch conventies

Hoewel je vrij bent om eender welke branch te gebruiken als je basisbranch in Git, verwachten mensen meestal de laatste, up-to-date code van een bepaald project in de master/main branch terug te vinden.

1.2 WAAROM BRANCHEN WE?

We werken in Git met commits, deze commits stellen ons in staat om steeds terug te keren naar een bepaald punt in de code. Je kan je dus de vraag stellen, als we steeds op elk moment naar elk punt terug kunnen waarom hebben we dan in hemelsnaam branches nodig?

Stel je echter een situatie voor waar je als developer je project aan de klant moet gaan tonen en een aantal van je collegas is druk bezig om bijkomende functionaliteit te programmeren die op dat moment nog niet volledig werkt. Omdat branching wordt gebruikt om verschillende ideeën / taken te scheiden, is de code die wel werkt en de functionaliteit die wel reeds klaar is afgescheiden van de lopende zaken. Je kan dus prima aan de slag met je demo zonder dat je rekening moet houden dat je op een deeltje botst wat nog niet klaar is voor gebruik.

Werk je binnen een project (individueel) aan een deeluitwerking (bv. een extra klasse) dan zal je gebruik maken van Branching om verder te werken aan de code zonder dat deze overgenomen wordt vooraleer je klaar bent en met zekerheid je uitwerking met je collega developers kan delen.

1.3 WERKEN MET EEN BRANCH

Om de basis van het werken met branches uit de doeken te doen gaan we even van start met een nieuwe folder die we als lokale git repository aanmaken.

1.3.1 STARTSITUATIE

Console

```
ConsoleFriend@Desktop ~/Desktop
$ git init BranchDemo
Initialized empty Git repository

ConsoleFriend@Desktop ~/Desktop
$ cd branchdemo

ConsoleFriend@Desktop ~/Desktop/branchdemo (master)
```

Folder

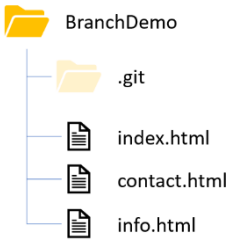
Branching

We voegen drie documenten toe in onze BranchDemo folder waar we zo meteen even mee aan de slag kunnen gaan. We opteren even om hier basis html documenten te voorzien. Na het aanmaken voorzien we een commit per html document en geven hier uiteraard een passende commit message mee.

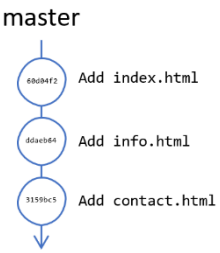
Console

```
ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git add index.html

ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git commit -m "Add index.html"
[master (root-commit) 60d04f2] Add index.html
 1 file changed, 11 insertions(+)
 create mode 100644 index.html

ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git add info.html

ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git commit -m "Add info.html"
[master ddaeb64] Add info.html
 1 file changed, 11 insertions(+)
 create mode 100644 info.html

ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git add contact.html

ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git commit -m "Add contact.html"
[master 3159bc5] Add contact.html
 1 file changed, 11 insertions(+)
 create mode 100644 contact.html

ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git log --oneline
3159bc5 (HEAD -> master) Add contact.html
ddaeb64 Add info.html
60d04f2 Add index.html
```

Folder



Branching



1.3.2 EEN EERSTE BRANCH

We starten eerst even met het uitvoeren van het `git branch` commando. Dit commando toont ons meteen welke branches er aanwezig zijn in onze huidige repository.

```
ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git branch
* master
```

We zien meteen dat de branch met naam master reeds bestaat en dat we ons ook op deze branch bevinden. We kunnen nu gebruik maken van hetzelfde commando gevolgd door de naam van onze branch die we wensen te maken om een nieuwe branch aan te maken.
`git branch contact-chat-feature` levert ons een nieuwe branch op met de naam **contact-chat-feature**. We zien door het uitvoeren van `git branch` vervolgens dat we nu **2 branches** in onze repository staan hebben. Onze **master** branch en de **contact-chat-feature** branch. Git geeft ons tevens terug mooi aan dat we ons in de **master branch** bevinden *(dit kunnen we visueel ook afleiden uit de branchnaam die tussen haakjes naast onze locatie staat)*

```
ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git branch contact-chat-feature

ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git branch
  contact-chat-feature
* master
```

Om nu effectief naar onze nieuwe branch te gaan, kunnen we gebruik maken van het git commando `git checkout` gevolgd door de naam van de branch waar we heen willen. In dit geval maken we dus gebruik van onderstaand commando om naar de contact-chat-feature branch te gaan: `git checkout contact-chat-feature`

```
ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git checkout contact-chat-feature
Switched to branch 'contact-chat-feature'

ConsoleFriend@Desktop ~/Desktop/BranchDemo (contact-chat-feature)
$ git branch
* contact-chat-feature
  master
```

We zien meteen dat de branch tussen haakjes bij onze locatie gewijzigd werd naar **contact-chat-feature** en als we dit nog eens extra gaan controleren met het `git branch` commando zien we ook dat deze momenteel aangeduid staat als onze actieve branch. Als we dit vergelijken met onze startsituatie kunnen we vaststellen dat de wijzigingen die we reeds op onze master branch gedaan hadden nu ook op onze **contact-chat-feature** branch aanwezig zijn.

Console

```
ConsoleFriend@Desktop ~/Desktop/BranchDemo (contact-chat-feature)
$ git log --oneline
3159bc5 (HEAD -> contact-chat-feature, master) Add contact.html
ddaeb64 Add info.html
60d04f2 Add index.html
```

Folder

Branching

We zien dus dat alles wat we reeds hadden op onze **master** branch ook aanwezig is op onze **contact-chat-feature** branch. We openen even onze contact.html pagina en voegen daar een stukje html aan toe. Dit mag/kan je op eender welke manier doen. Wil je dit via de command line doen? Dan kan dit makkelijk via het `code contact.html` commando.

Voeg eender welke regel html toe aan de reeds bestaande pagina. De inhoud op zich heeft voor dit hoofdstuk en deze module even geen nut.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Contact</title>
</head>
<body>
    <h1>Contact page</h1>
    <h2>Fancy chat feature</h2>
    <p>right here</p>
</body>
</html>
```

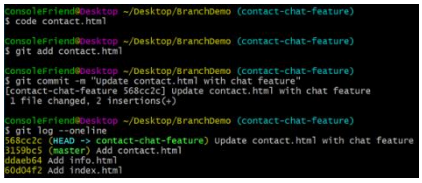
Voorbeeld waar we de regels in het geel toevoegden

We hebben nu een wijziging gemaakt aan onze **contact.html** pagina terwijl we in de **contact-chat-feature** branch aan het werk zijn!

We hanteren even de reeds vertrouwde commando's om deze wijzigingen aan te vullen in onze lokale git repository.

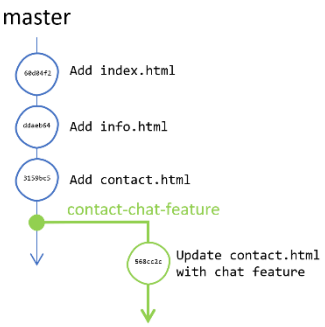
```
git add contact.html
git commit -m "Update contact.html with chat feature"
```

Console



Folder

Branching

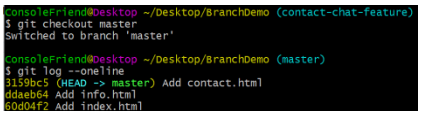


Onze git log geeft ons hier meteen een mooi overzicht dat ons duidelijk maakt dat de eerste drie commits in onze huidige branch hun oorsprong hebben vanuit de **master** branch *(op het punt/moment waar we onze branch aangemaakt hebben)*. Onze laatste commit is mooi toegevoegd op onze **contact-chat-feature** branch.

We gaan nu even terug naar onze master branch. Wisselen van branch kunnen we doen door het `git checkout` commando te hanteren.

```
git checkout master
```

Console



Folder

Branching

We zien na het wisselen van branch dat we ons terug bevinden op de **master** branch en dat er in de logs geen spoor te bekennen is van de wijzigingen die we maakten in onze **contact-chat-feature** branch. De **contact-chat-feature branch** is er wel degelijk nog steeds, maar de wijzigingen die we daar aan het contact.html document maakten zijn niet kenbaar in onze **master branch**.

We controleren dit even, nu we van branch wisselden, door de inhoud van onze contact.html pagina te bekijken.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Contact</title>
</head>
<body>
    <h1>Contact page</h1>
</body>
</html>
```

Voorbeeld contact.html pagina

We zien dus dat de regels die we toevoegden niet langer aanwezig zijn!

Omdat we wel een zekere *(gesimuleerde)* functionaliteit toevoegden aan onze repository in **de contact-chat-feature branch** willen we deze dus wel naar onze *(algemene)* **master branch** brengen. Dit doen we in het volgende deel van deze cursus.

1.4 SAMENVOEGEN VAN BRANCHES

Om inhoud, wijzigingen, functionaliteit samen te voegen in de branch die symbool staat voor de volledige uitwerking *(veelal de master branch)* kunnen we met het `git merge` commando aan de slag.

Let op!

Verder in dit hoofdstuk bespreken we de basis van de ideale workflow waar je effectief mee aan de slag zal gaan in modules vanaf volgend semester.

Daar we onze **master branch** aanzien als de leidraad in onze repository verwachten we dus alle 'finale' code in deze branch. De wijzigingen in onze **contact.html** staan momenteel gecommit op onze **contact-chat-feature branch** en willen we dus graag opnemen in onze **master branch**.

Om te mergen maak je gebruik van volgend commando

```
git merge contact-chat-feature
```

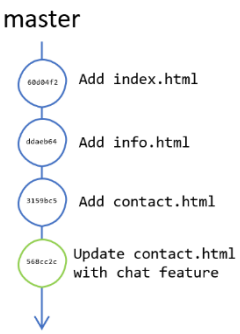
We ontleden even kort dit commando **git merge naam-van-branch**
Voeg samen wat je in deze branch vindt met de huidige branch waar je je bevindt.

Wees dus aandachtig bij het uitvoeren van dit commando dat je je wel degelijk bevindt in de branch waar je **naartoe** wil mergen.

Console

Folder

Branching



We zien nu ook in onze console dat de inhoud van commit 568cc2c op zowel onze master als onze **contact-chat-feature branch** aanwezig is.

```
ConsoleFriend@Desktop ~/Desktop/BranchDemo (master)
$ git log --oneline
568cc2c (HEAD -> master, contact-chat-feature) Update contact.html with chat feature
3159bc5 Add contact.html
ddaeb64 Add info.html
60d04f2 Add index.html
```

1.5 BASIS BRANCHES EN NAAMGEVINGSCONVENTIES

1.5.1 MASTER

Om alles vlot te laten verlopen, spreken we af dat we niet langer rechtstreeks op de master branch zullen werken. De master branch behouden we voor de finale versie van ons werk. Dit is de stabiele productiecode die altijd (altijd!) operationeel en lang binnen en buiten getest moet zijn.

1.5.2 DEV

Al onze 'ontwikkeling' zal plaatsvinden op onze dev branch (soms wordt hiervoor ook de voluit geschreven naam "develop"gekozen). Echter zullen we ook niet rechtstreeks op deze branch gaan comitten. De dev branch is wel steeds het 'aftak' punt voor nieuwe branches waar je mee aan de slag zal gaan, bv. om een specifieke nieuwe feature of bugfix te implementeren. Er zullen typisch meerdere zulke wijzigingen tegelijk in de pipeline zitten, elk op hun eigen branch, zodat hier ook verschillende personen/teams aan kunnen werken.

Eenmaal je klaar bent met een toevoeging van een stuk functionaliteit, zal je deze dan ook met de **dev branch** en **niet met de master branch** gaan mergen. Door eerst alle parallele ontwikkelingen naar dev te mergen, kunnen we hun integratie daar grondig testen, vooraleer het geheel naar de productiecode (master) doorstroomt.

Pas **na het voltooien** van verschillende features/fixes, en het grondig testen van hun samenwerking, zal je de finale uitwerking dan van dev naar master gaan mergen. In de praktijk zit er vaak nog een stap tussen— de zogenaamde "release" branches. Hier gaan we verder in dit hoofdstuk nog dieper op in.

1.5.3 CONVENTIE ROND DE NAAMGEVING

Voor de naamgeving van je branches maken we de afspraak dat je de kebab-case zal hanteren. Deze conventie vereist dat je alle woorden in kleine letters plaatst en een koppelteken (-) gebruikt om deze te linken aan elkaar.

We trachten uiteraard ook steeds zo duidelijk (en passend) mogelijk te zijn in de naamgeving van de branch.

✔	✘
login-screen	MyNewFeature
attack-animations	attackAnimations
refactor-email-service	refactorEmailService

1.6 EEN AANTAL COMMANDO'S

Commando	Beschrijving
`git branch` *my-branch*	Aanmaken van een branch (tak) met de naam *my-branch.* Deze nieuwe branch takt af van de huidige branch waarop je je bevindt wanneer je het commando uitvoert.
`git checkout` *branchname*	Wisselen van de huidige branch naar de branch met opgegeven naam.
`git checkout -b` *my-branch*	Aanmaken van een branch met de naam *my-branch* en onmiddellijk een checkout uitvoeren naar deze branch. Ook hier takt de nieuwe branch af van de huidige branch waarop je je bevindt.
`git merge` *some-branch*	Zal de branch *some-branch* mergen in de branch waar je je op dat ogenblik op bevindt.

1.7 EXPERIMENTEREN (MET BRANCHES)

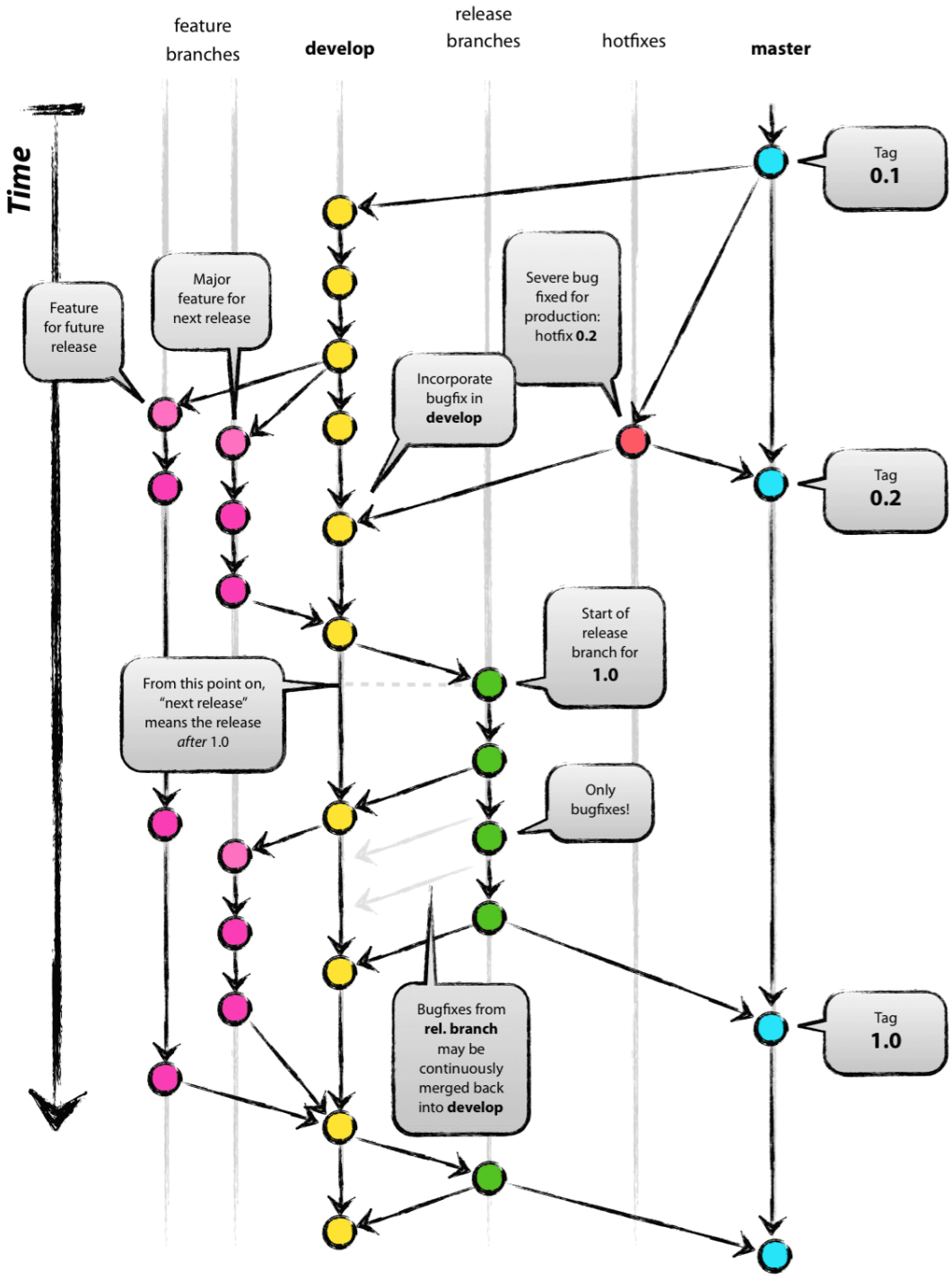
Branching kan best wel even verwarrend zijn op het moment dat je er kennis mee maakt. Wil je zelf even aan de slag om een aantal zaken uit te testen dan kan je steeds terecht op de visualising git website. https://git-school.github.io/visualizing-git/

Hier kan je naar hartenlust aan de slag met git en krijg je ook meteen een visuele voorstelling van de commando's die je uitgevoerd hebt.

2 GIT BRANCHING WORKFLOW

Elk bedrijf, organisatie, of development team dat met versiebeheer werkt, kan een eigen implementatie hebben omtrent het gebruik van branches. Afhankelijk van het aantal developers, grootte van het project en andere parameters zullen deze bedrijven een strategie bepalen voor het omgaan met versiebeheer.

In deze cursus zullen wij een workflow bespreken die als een **best-practice** omschreven wordt, omdat ze eenvoudig te begrijpen en makkelijk toe te passen is. Ze wordt ook in veel bedrijven (al dan niet gedeeltelijk) toegepast in de praktijk.

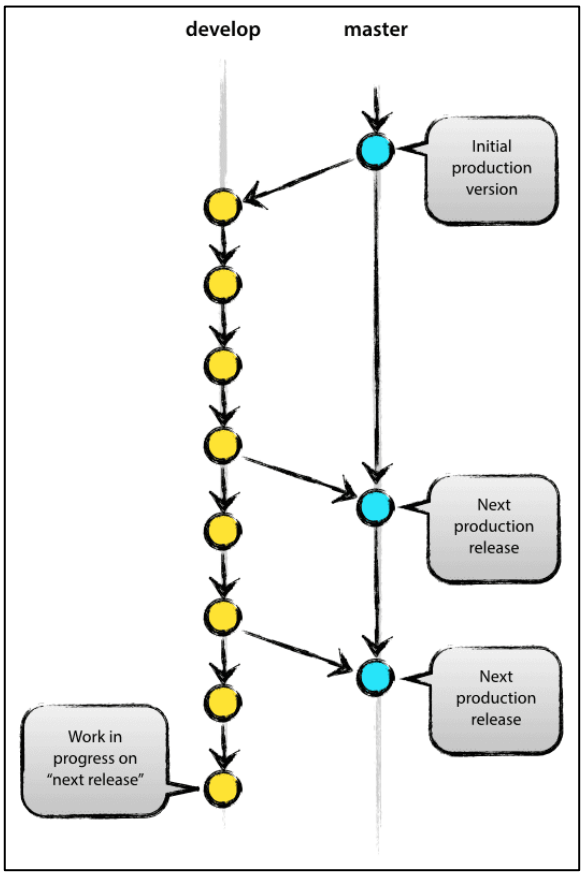


Bron: https://nvie.com/posts/a-successful-git-branching-model/

2.1 DE HOOFDBRANCHES

In de branching workflow die wij gaan bespreken, maken we gebruik van een repository die twee branches heeft die elk een oneindige levensduur hebben. Deze twee branches zullen dus nooit verwijderd worden.

- master (of main)
- dev (of develop)



Elke git repository die we maken heeft een master-branch. Maar parallel aan deze master-branch hebben we ook nog een dev branch.

We beschouwen de **master**-branch als de hoofdbranch. Alle broncode in deze branch wordt gezien als production-ready. Het bevat de laatste, stabiele versie van het project.

De **dev**-branch is de branch waarop dagelijks gewerkt wordt. Alle developers integreren hun nieuwste toevoegingen en wijzigingen aan de broncode met deze branch. Daarom wordt deze branch soms ook wel de **integration** branch genoemd.

Wanneer de broncode in de **dev**-branch klaar is om gereleased te worden zullen we de **dev**-branch mergen in de **master**-branch. Daarna wordt die commit getagged met met een release nummer. Hoe we dit doen, bespreken we verder in deze cursus.

Door deze workflow te volgen zal telkens een nieuwe productie release gecreëerd worden wanneer we veranderingen mergen naar de **master**-branch.

2.2 ONDERSTEUNENDE BRANCHES

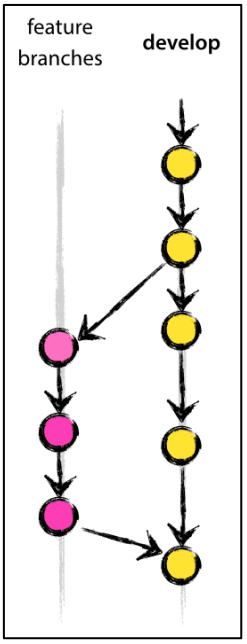
Naast de hoofdbranches **master** en **dev**, gebruikt ons branching model nog een aantal ondersteunende branches voor het faciliteren van development werk tussen teamleden, opvolgen van features, klaarmaken voor productie releases en om vlug bugs en problemen op te kunnen lossen. In tegenstelling tot de hoofdbranches hebben deze ondersteunende branches altijd een beperkte levensduur, aangezien deze ondersteunende branches ooit terug verwijderd zullen worden.

De verschillende types van ondersteunende branches kunnen zijn:

- Feature branches
- Release branches
- Hotfix branches

Elk van deze branches hebben hun eigen specifiek doel en zijn gebonden aan regels en conventies die bepaald worden door de organisatie of het development team.

3 FEATURE BRANCHES



Deze branches worden gebruikt om nieuwe features te programmeren voor één van de volgende releases. Wanneer we beginnen te programmeren aan een feature weten we meestal nog niet wanneer deze feature zal opgenomen worden in de release.

De levensduur van een feature branch is net zolang men nog aan het programmeren is aan de feature. Daarna wordt een feature-branch geïntegreerd in de **dev**-branch door middel van een **merge**.

Een feature-branch hoeft niet altijd gemerged te worden. Men kan zo'n branch gebruiken om bijvoorbeeld een experiment uit te proberen. Als dat experiment faalt, dan kan de bijbehorende **feature**-branch (en code) gewoon verwijderd worden—zonder dat hier ooit iets van op de **dev** of **master** branch zichtbaar werd!

Feature branches zijn steeds afkomstig van de volgende branch:

- **dev**

Mogen enkel gemerged worden in de volgende branch:

- **dev**

Naamgevingsconventies:

- alles kan, behalve master, dev, release- of hotfix-
- kebab-case
- begint met de prefix `feature/`

✔	✕
`feature/login-screen`	`features/login-screen`
`feature/attack-animations`	`attack-animations`
`feature/refactor-email-service`	`Feature/refactorEmailService`

3.1 AANMAKEN FEATURE BRANCH

Volgend git commando zorgt voor een nieuwe **feature**-branch genaamd "my-feature" vanuit de **dev**-branch die we ook in één adem "uitchecken":

```
$ git checkout -b my-feature dev
Switched to a new branch "my-feature"
```

Merk op dat we hier een tweede argument "dev" meegeven aan het commando, zodat we de nieuwe branch "my-feature" zeker en vast aftakken vanaf dev, ook al zitten we op dit moment eventueel op een andere branch. Ter herinnering: default wordt de nieuw aangemaakte branch afgetakt van de branch waarop je je op dit moment bevindt.

Als we echter regelmatig feature branches maken kan het handig zijn om deze te bundelen. Dit kunnen we doen door **<foldernaam>/** toe te voegen aan de naam van de nieuwe **feature**-branch. Onderstaand voorbeeld toont hoe we dit kunnen doen:

```
$ git branch feature/my-feature-branch1 dev
$ git branch feature/my-feature-branch2 dev
```

3.2 MERGEN VAN EEN FEATURE BRANCH

Wanneer we klaar zijn met een feature dan kunnen we deze mergen in de **dev**-branch om onze wijzigingen op te nemen voor de (latere) release:

```
$ git checkout dev
Switched to branch 'dev'
$ git merge feature/my-feature
Updating ea1b82a..05e9557
(Summary of changes)
```

Aangezien de ”feature/my-feature” branch gemerged werd in de **dev**-branch mag die feature branch nu verwijderd worden. Dit doen we met volgend commando:

```
$ git branch -d feature/my-feature
Deleted branch feature/my-feature (was 05e9557).
```

Ook al is de ”feature/my-feature”-branch nu verwijderd, toch zullen we zien in de git log dat de betrokken commits nu ook opgenomen zijn in de “dev”-branch. Dit omdat we eerst deze commits in dev hebben gemerged. Het is dus uiteraard belangrijk om de feature branch pas te verwijderen NA de merge naar dev.

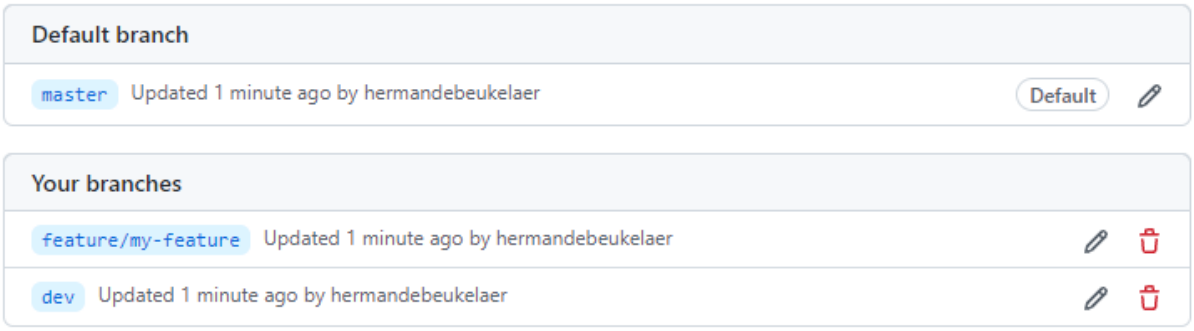
Om alle lokale wijzigen (op de dev branch) door te sturen naar de remote repository gebruiken we het volgend commando:

```
$ git push
```

Je zal zien dat een lokaal verwijderde branch niet automatisch ook op de remote (GitHub) verwijderd wordt. Je kan dit doen via volgend commando:

```
$ git push origin --delete feature/my-feature
```

Als alternatief kan je ook branches verwijderen via de interface op de site van GitHub. In een later hoofdstuk komen we nog uitgebreid terug op de functionaliteiten die GitHub aanbiedt.



4 RELEASE BRANCHES

Release branches dienen om de voorbereidingen te treffen van een nieuwe productie release. Deze branches dienen om de laatste puntjes op de i te zetten. Bijvoorbeeld:

➜ Voorbereiden van alle meta-data voor de release
- Versie nummer
- Build datums
- …

➜ Grondig testen van de release. Hier wordt vaak over *user acceptance testing* (UAT) gesproken.

➜ Eventuele kleine bugfixes kunnen hierin gebeuren.
- Uiteraard dienen al deze bugfixes ook in de dev-branch terecht te komen. Er moet dus onmiddellijk een merge met de develop-branch gebeuren na een bugfix.

Er wordt een nieuwe **release**-branch gecreëerd zodra alle gewenste features voor de eerstvolgende release aanwezig zijn in de **dev**-branch. Met andere woorden: op een aantal kleinigheden na is de broncode in de dev-branch klaar om in productie te gaan.

Door die kleinigheden uit te voeren op een aparte release-branch, blijft de dev-branch vrij om daarop features te integreren voor releases verderop in de toekomst. Zolang er nog geen **release**-branch gemaakt is voor de eerstvolgende release, dan mogen features bedoeld voor latere releases **nog niet** geïntegreerd worden in de dev-branch.

Pas bij het aanmaken van een **release**-branch zullen we een versienummer toekennen aan deze release. Vanaf dit punt kunnen we nieuwe features (die voorzien zijn voor de volgende release) gaan toevoegen aan de **develop**-branch.

Release branches zijn steeds afkomstig van de volgende branch:

➜ **dev**

Moeten gemerged worden in de volgende branches:

➜ **dev** en **master**

Naamgevingsconventies:

➜ begint met de prefix `release/`

➜ volgens teamafspraak: vaak “release-” gevolgd door een versienummer, of enkel een versienummer

✔	✕
`release/release-1.3`	`releases/release-1.3`
`release/rel-1.3.1`	`first-release`
`release/v-1.3.1`	`Release/v-next`

4.1 AANMAKEN RELEASE BRANCH

Laat ons er even van uitgaan dat de huidige productieversie 1.2 is en dat de komende grote release versie 1.3 is. We hebben dus al een **release**-branch met de naam “release/release-1.2“.

Alle features voor 1.3 zijn geïntegreerd in de **dev**-branch, maar er moeten nog een aantal bugs weggewerkt worden. Intussen staan er al een paar teamleden te popelen om enkele features voor 1.4 op de dev-branch te plaatsen.

We kunnen nu de **dev-branch vertakken** naar de nieuwe **release**-branch voor versie 1.3. Dit geeft ons ruimte om de bugs voor versie 1.3 te fixen. De dev-branch is nu beschikbaar voor de andere teamleden, die vanaf nu hun features voor versie 1.4 kunnen integeren.

```
$ git checkout -b release/release-1.3 dev
Switched to a new branch "release/release-1.3"
```

In de nieuwe release branch maken we wat wijzigingen in de meta-data, zoals versienummer aanpassen naar 1.3. Daarna maken we een commit met deze wijzigingen.

```
$ git add .
$ git commit -m "Bump version number to 1.3"
[release-1.3 74d9424] Bump version number to 1.3
1 files changed, 1 insertions(+), 1 deletions(-)
```

De verdere commits op deze release branch dienen nu enkel om de bugs te fixen en het product helemaal klaar te stomen voor oplevering. Deze wijzigingen en fixes kunnen nog continu geïntegreerd worden **naar de dev-branch**, maar **nooit** meer vanaf dev-branch naar de release-branch (omdat er op dev intussen misschien al zaken staan die pas voor de volgende release bedoeld zijn).

4.2 AFWERKEN VAN EEN RELEASE BRANCH

Eens de **release**-branch volledig klaar is om gereleased te worden, moeten we nog enkele acties ondernemen.

Omdat elke aanpassing van de **master**-branch per definitie een nieuwe release is, gaan we de afgewerkte **release**-branch mergen in de **master**-branch. Deze commit (als gevolg van het mergen) zullen we dan ook taggen om later te kunnen refereren naar deze versie.

```
$ git checkout master
Switched to branch 'master'
$ git merge release/release-1.3
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.3
```

Als laatste stap moeten we nu de **release**-branch “release/release-1.3” gaan mergen in de **dev**-branch zodat alle bugfixes die gemaakt werden in de release-branch nu ook gefixed worden in de **dev**-branch.

```
$ git checkout dev
Switched to branch 'dev'
$ git merge release/release-1.3
Merge made by recursive.
(Summary of changes)
```

In deze stap kunnen we te maken krijgen met een “merge conflict” omdat het versienummer ergens aangepast is in onze code/bestanden. Wanneer dit het geval is zal dit merge conflict opgelost moeten worden en dient er hierdoor nog een commit te gebeuren. Hoe we merge-conflicten oplossen komt later nog aan bod in deze cursus.

Nu zijn we volledig klaar en kunnen we de **release**-branch verwijderen:

```
$ git branch -d release/release-1.3
Deleted branch release/release-1.3 (was ff452fe).
```

Wil je deze ook op de remote verwijderen, dan eindig je met dit commando:

```
$ git push origin --delete release/release-1.3
```

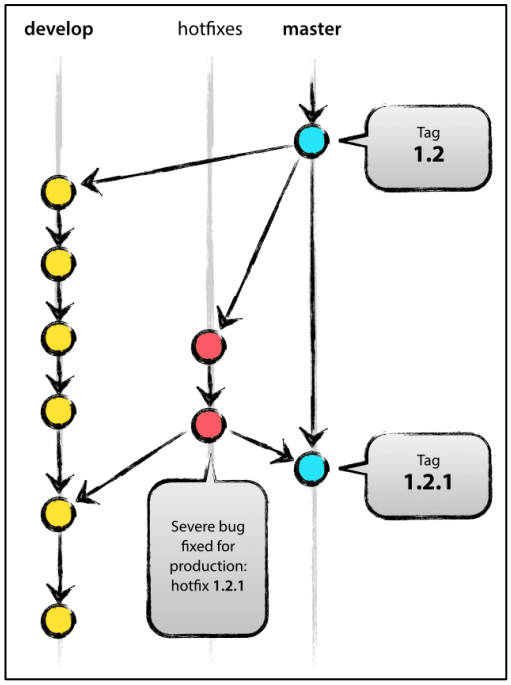
Je zou misschien denken dat we de release branch beter bewaren om later nog terug te kunnen keren naar een specifieke oudere release versie. Dit is echter niet nodig, net omwille van het feit dat we op de master branch een tag hebben voorzien na het mergen van release 1.3 in master. Een tag kan je net zoals een branch gewoon uitchecken, bv:

```
$ git checkout 1.3
```

Zo kan je dus altijd achteraf nog eens gaan rondneuzen in een specifieke oudere versie. Dit kan bv. handig zijn om uit te zoeken sinds welke versie een bepaalde bug werd geïntroduceerd, om een beter zicht te kunnen krijgen op de mogelijke oorzaak.

5 HOTFIX BRANCHES

Hotfix branches zijn erg gelijkaardig aan **feature**-branches. Deze hotfix branches zijn bevatten noodzakelijke, **niet geplande wijzigingen** voor de huidige productie release.



Het gaat hier meestal om kritieke bugfixes die zo snel mogelijk moeten worden opgelost voor de eindgebruiker. Hierdoor kunnen we niet wachten op de planning om dit pas in een volgende grote release (bijvoorbeeld 1.3) op te nemen. We gaan daarom rechtstreeks van master aftakken en de fix ook rechtstreeks naar master mergen. Dit is een shortcut die zoveel mogelijk dient vermeden te worden, maar soms noodzakelijk is.

Eens de wijzigingen in zo'n **hotfix**-branch voltooid zijn, worden ze rechtstreeks geïntegreerd op de **master**-branch. Deze kleine release krijgt een bijbehorend (minor) versienummer, in dit voorbeeld is dat 1.2.1. Een voltooide hotfix-branch wordt ook geïntegreerd met de **dev**-branch, zodat de wijzigingen meekomen met alle volgende releases.

Door deze manier van werken kan het development team verder werken in de **dev**-branch terwijl een ander lid van het team zich kan bezighouden met het oplossen van kritieke bugs in een **hotfix**-branch.

Het aantal hotfixes die nodig waren in een bepaalde release, is een belangrijke maatstaf voor de codekwaliteit en het development proces. Liefst hebben we geen hotfixes nodig. Als er veel hotfixes moeten gebeuren, wordt het proces best even onder de loep genomen. Dit duidt er bijvoorbeeld doorgaans op dat er te weinig of niet grondig genoeg getest wordt, waardoor er fouten door de mazen van het net heen glippen. Het is belangrijk om het proces dan trachten te verbeteren om het aantal hotfixes naar beneden te krijgen in nieuwe releases.

Hotfix branches zijn steeds afkomstig van de volgende branch:

- **master**

Moeten gemerged worden in de volgende branches:

- **dev** en **master**

Naamgevingsconventies:

- begint met de prefix `hotfix/`
- volgens teamafspraak: vaak het bedoelde versienummer en een summiere beschrijving

✔	✕
`hotfix/hotfix-1.2.1`	`hotfixes/hotfix-1.2.1`
`hotfix/v-1.2.1-caching-problem`	`gerard-messed-things-up-again`
`hotfix/1.2.1-data-leak`	`Hotfix/v-1.2.1`

5.1 AANMAKEN HOTFIX BRANCH

Hotfix branches worden aangemaakt vanaf de **master**-branch. Laat ons er even van uit gaan dat de versie 1.3 nu in productie is. We merken op dat er in deze versie een bug aanwezig is die voor verschillende problemen zorgt. In de **dev**-branch wordt nog volop gewerkt waardoor deze niet stabiel genoeg is om nu al te releasen.

We zijn dus genoodzaakt om vanaf de **master**-branch een **hotfix**-branch te maken om het probleem op te lossen. We doen dit als volgt:

```
$ git checkout -b hotfix/hotfix-1.3.1 master
Switched to a new branch "hotfix/hotfix-1.3.1"
```

In deze nieuwe **hotfix**-branch zullen we eerst onze meta-data aanpassen zoals de versie veranderen van 1.3 naar 1.3.1. We doen hierna een commit:

```
$ git add .
$ git commit -m "Bump version number to 1.3.1"
[hotfix-1.3.1 41e61bb] Bump version number to 1.3.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

Daarna pakken we de effectieve bug aan. We zorgen ervoor dat het probleem opgelost is en dat alle testen (automatisch en/of manueel) slagen. Tijdens het oplossen van het probleem doen we één of meerdere commits:

```
$ git add .
$ git commit -m "Fix severe security issue"
[hotfix-1.3.1 abbe5d6] Fixed severe security issue
5 files changed, 32 insertions(+), 17 deletions(-)
```

5.2 AFWERKEN VAN EEN HOTFIX BRANCH

Eens we klaar zijn met het oplossen van de problemen moeten we deze **hotfix**-branch nu gaan mergen in de **master**-branch. De procedure is analoog aan het afwerken van een **release**-branch.

```
$ git checkout master
Switched to branch 'master'
$ git merge hotfix/hotfix-1.3.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.3.1
```

Uiteraard mogen we niet vergeten om ook deze **hotfix**-branch te mergen in onze **dev**-branch aangezien ook de opgeloste problemen nog aanwezig zijn is in de **dev**-branch.

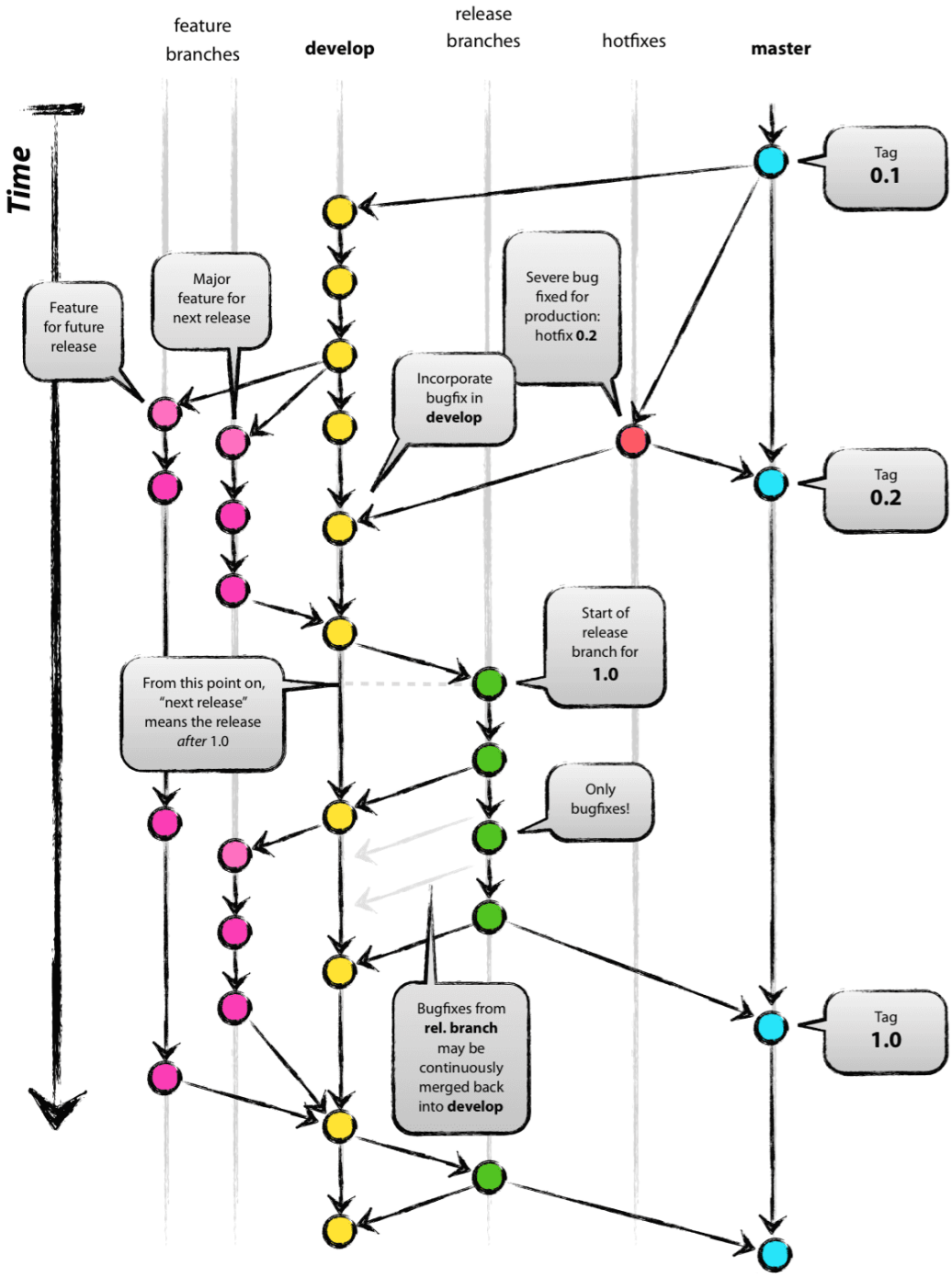
```
$ git checkout dev
Switched to branch dev
$ git merge hotfix/hotfix-1.3.1
Merge made by recursive.
(Summary of changes)
```

Er is echter één uitzondering! Indien er al een **release**-branch bestaat voor versie 1.4 dan wordt er niet langer naar deze release geïntegreerd vanaf de dev-branch. De hotfix-branch moet dan in deze release branch gemerged worden, zodat de fix meekomt met versie 1.4.

Als laatste stap gaan we nu deze **hotfix**-branch verwijderen (lokaal en remote):

```
$ git branch -d hotfix/hotfix-1.3.1
Deleted branch hotfix/hotfix-1.3.1 (was abbe5d6).
$ git push origin --delete hotfix/ hotfix-1.3
```

Bekijk nog eventjes onderstaande figuur, hopelijk is deze nu al heel wat duidelijker:



6 EXTRA EIGEN BRANCHES MAKEN

Uiteraard staat het je vrij om nog extra branches aan te maken. Je wenst bijvoorbeeld een nieuwe nuget package uit te proberen in je project. Dan kan je vanuit een **feature**-branch nog een extra branch maken waarin je experimenteert met deze nieuwe NuGet package. Wanneer zou blijken dat dit experiment niet voldoet aan de eisen, dan kan je deze branch terug verwijderen zonder dat de code in je **feature**-branch aangetast werd.

Mocht blijken dat de nieuwe NuGet package bruikbaar en handig is in het volledige project kan deze eigen gemaakte branch gewoon gemerged worden in je **feature**-branch die later dan kan gemerged worden in de **dev**-branch.

7 TERUGBLIK OP DE BRANCHING WORKFLOW

Zoals aangehaald in het begin van deze cursus is de Branching Workflow een best-practice voor het continu integreren van wijzigingen aan de broncode. De besproken workflow schetst ook de voordelen van het werken met branches.

De workflow vormt een goede basis, maar kan afhankelijk van de noden en voorkeuren van een organisatie of team worden uitgebreid of net vereenvoudigd.

Dergelijke variaties komen tot stand via duidelijke afspraken. Zo kan het voorvallen dat de klassieke benaming van de “master” branch niet gehanteerd wordt, maar in plaats daarvan wordt een “production” branch gebruikt met dezelfde principes.

howest.be