# Organizing Larger Programs

# Bij grote Pyhton projecten moeter rekening gehouden worden bij de opbouw van structuur. Dit wordt de modulaireo organisatie genoemd. We introduceren of bespreken packages, namespace packages, executable packages, recommended file en directory layouts, en het distribueren van een nieuwe package.

## Modules en packages

# Main block concept: de functie main() functie wordt enkel uitgevoerd indien het geïmporteerde bestand wordt uitgevoerd. We kunnen dus via het main block concept code importeren zonder dat dit wordt uitgevoerd.

# Speciale methoden: \_\_<method-name>\_\_ : dunder ipv dubbel underscore

# Modules: basic tool van python om code te organiseren. Het is meestal één enkel bestand dat we importeren via een import. Na het importeren is het zoals een object van type module.

# Packages: Dit is een type module, dat andere modules en packages kan bevatten. Packages bezitten een dunder path, in deze list van paden wordt er gezocht naar geneste modules, terwijl modules deze dunder niet hebben.

# Localiseren van modules : Als we aan python vragen om een module te importeren, kijkt hij naar het path attribuut van de sys module (sys.path). Dit is een lijst van directories. We kunnen ook aan de hand van PYTHONPATH dingen toevoegen aan de lijst van directories. Hierbij moeten we dus niet sys.path expliciet aanpassen.

## Implementing packages

Creating packages: voor een module is het gewoon het aanmaken van een source bestand in een directory in sys.path. Voor een package maken we eerst een map aan in een directory in sys.path. In deze map werd er een bestand genaamd \_\_init\_\_.py aangemaakt dat vaak leeg is. Vanaf Python 3.4 is dit niet meer noodzakelijk. Dit geeft aan dat het een package is en wordt dus meestal toch nog aangemaakt. Als we het package importeren zal het dunder init bestand worden uitgevoerd. In deze map kunnen we dan .py bestanden schrijven en subpackages definiëren.

Creating a subpackage: Map in package aanmaken en ook hierin een dunder init file aanmaken. In deze subpackage kunnen we dan andere .py bestanden schrijven.

Relative imports: Hiermee kunnen we teruggaan in de directory aan de hand van ‘..’ en dan een bepaald bestand importeren. Per punt gaan we één stap terug, dus naar de parent. Via ‘..’ gaan we dus naar de grandparent. *Bvb: from ..module\_name import name*

Using dunder all: dit attribuut van modules laat toe te controleren welke attributen er worden geïmporteerd bij een from x import \*

## Namespace and Executable Packages

Namespace packages: Dit wordt gebruikt om een Python package over verschillende directories op te delen. Deze mogen geen dunder init files bevatten. Het scant elke directory in sys.path, kijkt of het een package met een bepaalde naam vindt een init bestand. Indien wel importeert het deze, indien niet zal het andere directories met deze naam importeren als deel van een namespace package.

Executable Directories: we willen een executable hebben voor een directory. Hiervoor maken we gebruik van een dunder main python bestand. (\_\_main\_\_.py). Dit kunnen we dan aanroepen door python directory in de terminal te schrijven. Deze directory wordt eerst aan het sys.path bestand toegevoegd zodat het packages of modules kan importeren dat zich in dezelfde map bevinden.

Executable Zip Files: We kunnen zip files op een gelijkaardige manier gebruiken als executable drectories want python kan deze lezen. We zippen niet de directory maar de volledige inhoud hiervan!

Executable Packages: dunder main python bestand in package zal worden gerunt indien we het oproepen alsvolgt: python -m package

## Recommended Project Layout

Een **algemene opbouw** van een project dat altijd goed is:

* Afbeelding met tekst

  Automatisch gegenereerde beschrijvingProject name (niet het package)
* README is uitleg over package in reStructuredText formaat:

het is simpel en verwijst nr de docs

* docs: documentatie over package
* src: Bevat de package name met daarin:

1. dunder init.py
2. modules
3. subpackages

* test: bevat al de tests voor het project
* setup.py : setup file adhv setuptools package

via **plugins** kunnen we nieuwe features toevoegen aan een package zonder de src code zelf aan te passen. In de package worden extension points gedefinieerd. Het package zal dan discovery technieken gebruiken om deze extensies te vinden en in te laden bij runtime. Er zijn verschillende manieren om dit te doen. Hier worden er twee besproken.

Plugins with namespace packages: dit maakt gebruik van pkgutil module

Plugins with setuptools: maakt gebruik van de entry points functionaliteit van het setuptools package. 🡺 PLUGINS MOEILIJK

## Package Distribution

Om het te delen met anderen. De standaard methode om dit te doen is door een distribution package aan te maken. Dit is een archief van de inhoud van het package en is gemakkelijk te installeren. Ze komen in verschillende formaten zoals ZIP, Tarballs en Wheels. Er zijn twee typen van distribution packages. Built en Source Distribution packages. Built: kan in installation directory geplaatst worden en achteraf meteen gebruikt worden. Dit is vaak specifiek voor een platform. Source: Dit bevat alles om het package te bouwen. Het moet dus worden gebouwd aan de hand van sdist en dist.

Om het te delen uploaden we ze op de python package index op pypi.org. Om iets te uploaden wordt er gebruik gemaakt van Twine.

# Getting started with GIT 2

# Het is een version control system (VCS). Het wordt gebruikt om samen aan een project te werken. In deze les begrijpen we de basics van GIT. Het houdt de bewerkingen bij aan files zodat we een vorige versie kunnen terugzetten. We kunnen zien wie op welk moment wijzigingen heeft aangebracht.

## Get up and running

GIT en code hosting providers zijn twee verschillende dingen maar ze zijn complementair. We zullen gebruik maken van een coffee shop analogie. Het is belangrijk om in te zien hoe GIT met de data omgaat. De eigenaar past recepten aan maar wil steeds kunnen terugkeren naar een vorige versie indien dit lekkerder was.

Er zijn 3 stages van files:

1. committed: data in bestand is opgeslagen in de eigenaar zijn locale database
2. modified: Terwijl hij aanpassingen maakt is de file modified. Work in progress
3. staged: Wanneer hij klaar is met de aanpassingen. Dan kan deze commit worden en bevinden we ons in een nieuwe commit snapshot. Nieuw toegevoegde bestanden in GIT komen in de staged files, zodat deze achteraf mee committed worden.

Een GIT project is opgedeeld in 3 delen:

1. .git directory (= repository): Dit is de oorsprong van zijn de data van het project. Het wordt gepulled van de remote server. (kan ook een lokale computer zijn). Hier wordt dus de metadata alsook de object database van het project opgeslagen.
2. Working directory: een kopie van één versie van het project. Met andere woorden, een kopie van de repository. Dit kan dan aangepast worden.
3. Staging area: dit zit tussen de repository en de working directory en wordt ookwel de index genoemd. Er wordt een snapshot genomen van de delen in de working directory waar aanpassingen werden op aangebracht. Zo kan de gebruiker kiezen wat hij al dan niet commit. (de delen die hij in de staging area heeft geplaatst)

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Opdrachtprompt commando’s

Mkdir: new empty folder

Dir: alle paden weergeven

Cd: navigeren

Cls: clear

Nieuwe txtfile: echo "new txtfile" > newFile.txt

Edit file met notepad: notepad somefile.txt met VSC= code somefile.txt

Dir /b /s

GIT basic commands

git help geeft een kort overzicht van meest gebruikte functies

git config –list geeft alle ingestelde configs (naam,email)

Nieuwe repo: naar map waar we een repo willen maken en git init typen. Dan wordt er een .git bestand gemaakt dat standaard hidden is. We kunnen ernaar navigeren met cd .git en dit lezen door ls te typen. Het bevat de metadata.

Om vanop verschillende apparaten een git repository te kunnen binnenhalen, moet deze git repository zich op een cloud achtige structuur bevinden. Een bekende service hiervoor is **GitHub**. Het is een web based git repository hosting service. Zulke services bieden ook andere functies aan, zoals samenwerken te integreren in een GIT repository.

**GITHUB** account:

Cred: [milan.velle.ext@rbfa.be](mailto:milan.velle.ext@rbfa.be)

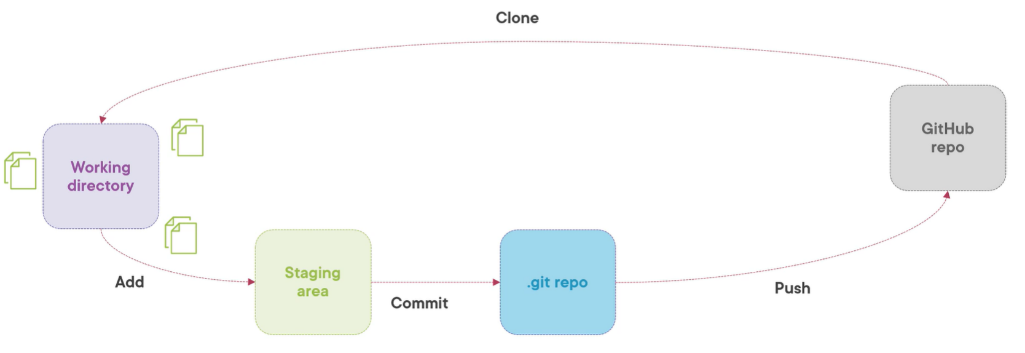
“Git add .” zet alles in de staging area (lokaal).

“git commit -m “first commit”“ **commit** alles een eerste keer naar lokale repo.

**Linken** van lokale repo met online github repo:

git remote add orgin <https://github.com/mivelRBFA/Example_GIT_repo.git>

Push locale repo: git push -u origin master



## Basic & extended commands of everyday GIT

Wanneer we in GIT iets aanpassen doen we dit op een specifieke branch. Met GIT status kunnen we zien op welke branch we ons bevinden en of deze al dan niet up to date is met de origin/master branch.

We kunnen de commit message uitbreieden met een body.

Indien we een bestand willen verwijderen, moeten we git eerst zeggen dat ij dit niet meer moet volgen en verwijderen.

Branches

*Afbeelding met tekst, elektronica

Automatisch gegenereerde beschrijving*

# GitHub: Getting started

## An overview of Git and GitHub

Git is open source and free. GitHub is very fast. Every developer is working locally. It is powerful and easy to use. Branching is one of the most used features of Git. We keep the master branch as quality code. Another powerful tool is the pull request. It is mostly used on the command line, but there are also some GUI’s available like GitHub desktop, SourceTree and Tower. GitHub is a Git hosting site. It is much more than only source control for your code. We can make public and private repo’s with the free version of GitHub.

## Getting started with GitHub

GitHub is owned by Microsoft. Het pull request zorgt ervoor dat veranderingen kunnen worden binnengehaald in een andere branch.

In een repo op GitHub zijn er meerdere **tabs**. Ten eerste is er de code zelf. Dan volgt er een tab genaamd issues om bijvoorbeeld een bug aan te geven. Dan zijn er ook pull requests, dat worden gebruikt om aanpassingen samn te voegen in een branch. Actions worden gebruikt om dingen te automatiseren. Projects worden gebruikt om het werk aan een repo op te delen. De wiki wordt gebruikt om documentatie over een repo op te slaan. Dan volgen er security settings en insights. Insights geeft een overzicht van hoeveel commits er worden gedaan, wie eraan werkt, … Als laatste zijn er ook enorm veel settings.

**SSH vs HTTPS**: om te connecteren met GitHub kunnen we deze twee opties gebruiken. Soms is SSH een betere optie, aangezien het bij HTTPS het eval kan zijn dat we meerdere keren onze gebruikersnaam en ww moeten indienen bij bijvoorbeeld het pushen van commits naar GitHub. SSH is gemakelijker om mee te werken. Er word een SSH key aangemaakt dat wordt gelinkt aan de Gitub acc. We connecteren dan met onze locale machine met GitHub en we hoeven niet steeds onze credentials ingeven. Het is beide even veilig.

Eerst moet er een **SSH key** worden aangemaakt, dan moet de key worden toegevoegdaan de SSH agent. Dit is lokaal. Achteraf oeten we de SSH key ook toevoegen aan GitHub. Dit doen we door de inhoud van het .pub bestand in de .ssh map te kopiëren, naar GitHub te gaan, te klikken op new SSH key, en de inhoud te kopiëren na het apparaat een naam te geven.

**Searching in GitHub:** We kunnen een global search (alle repos), een scope definieëren (één repo, de issues, …) of een filter toepassen.

## Working with repositories

Een repo is de folder voor het project. GitHub bevat een remote repo. Gitignore file is voor bestanden die niet moeten worden geupload inde remote git repo. Hiervoor kiezen we een bestandstype voor deze file. We kunnen ook een license kiezen want dit bepaalt wat mensen met onze repo kunnen doen. Linksboven de code beqtanden vd repo zien we hoeveel branches er zijn en in welke we ons bevinden.

Werken met een repo is mestal met meerdere mensen tegelijk. Dit betekent dat we de aanpassingen moeten binnenhalen. Hiervoor gebruiken we de fetch en pull commandos. In het **README** bestand zetten we informatie over de repo. Informatie zoals wat het doet, we het bezit, wie er aan bijdraagt, waarvoor het dient,… Dit moet geplaatst worden in de root folder, de .github of docs folder. Het is automatisch gerendered op de landing page van de repo en is meestal opgemaakt in een markdown formaat (.md). De **LICENSE.MD** bevat de open-source licentie. De **contributing** file bevat een lijst aan personen of organisaties die hebben geholpen aan het project. Het **Changelog** bestand bevat een lijst met de aanpassingen tussen verschillende versies. De **support** file informeert mensen van mogelijke support voor het project. De **code of conduct** file bevat regels waar contibuters zich aan moeten houden. **CODEOWNERS** bestand geeft aan wie verantwoordelijk is voor de code in de repo.

De **markdown format** is heel simpel om te zetten in HTML. H1 is aan de hand van een #, H2 aan de hand van ##, enzovoort.. wanneer we een lijst willen opsommen gebruiken we een \* of -. De README markdown file wordt omgezet nr een HTML pagina.

Door op de settings wheel te duwen, kunnen we **topics** toevoegen aan de repo zodat het makelijker vindbaar is en zodat gelijkaardige repos worden aangeraden. **Issues** worden gebruikt om vanalles te rapporteren, zoals bugs of problemen. **Projects** worden gebruikt om een project board aan te maken. Hiervoor moeten we een template kiezen.

Collaborators: core team dat werkt aan het project. Deze kunnen committen naar de main of master branch. Contributers zijn alledaagse mensen die aanpassingen aanraden, maar deze kunnen niet committen nr de master branch. We kunnen meerdere **mensen toevoegen** door naar settings te gaan en dan manage acces. Van daaruit kunnen we mensen uitnodigen. **Insights** geven een overzicht van de belangrijkstecontributers. Een tweede inisight is een overzicht van de commits. Traffics geven weer hoe vaak een repo werd gekopieerd of bezocht. De code frequency geeft een idee van de activiteit van het project.

## Collaborating using Github Flow

Deze module gaat over pull request en branches. Achteraf wordt de GitHub Flow en forking aangehaald.

**Branches**

= divergeren van de master of main development line. Het is in feite een pointer naar een commit.

The HEAD pointer is een pointer dat wijst naar de branch waar er op gewerkt wordt. Aan de hand van een checkout veranderen we van branch. We kunnen dan de aanpassingen van de nieuwe branch eenvoudig mergen in de master branch indien er geen aanpassingen werden gedaan aan de master branch. We kunnen vanuit GitHub ook een nieuwe branch aanmaken.

**Branching strategie** is belangrijk! Het is goed om een paar regels op te stellen. We definiëren een workflow. Enkele voorbeelden hiervan zijn:

* **Centralized** workflow: één branch waar alle commits op worden gemerged.
* **GitFlow** workflow: Het is gebaseerd op twee verschillende branches, namelijk de master of main branch en de development branch.
* **Forking** workflow: elke developper werkt op een copy van de repo.
* **Github workflow**: branches worden steeds aangemaakt voor het oplossen van een bug, de ontwikkeling van een nieuwe feature, … Het werkt volledig rond pull requests. We maken aanpassingen op een branch en willen deze dan toevoegen aan een andere branch zoals bijvoorbeeld de main branch. We zullen dan een pull request aanmaken. Het wordt dan opengesteld voor discussie, andere collaboraters kunnen de aanpassingen dan reviewen en eventueel zelf nog commits toevoegen. Als de aanpassingen worden goedgekeurd kunnen ze gemerged worden met bijvoorbeeld de main branch.

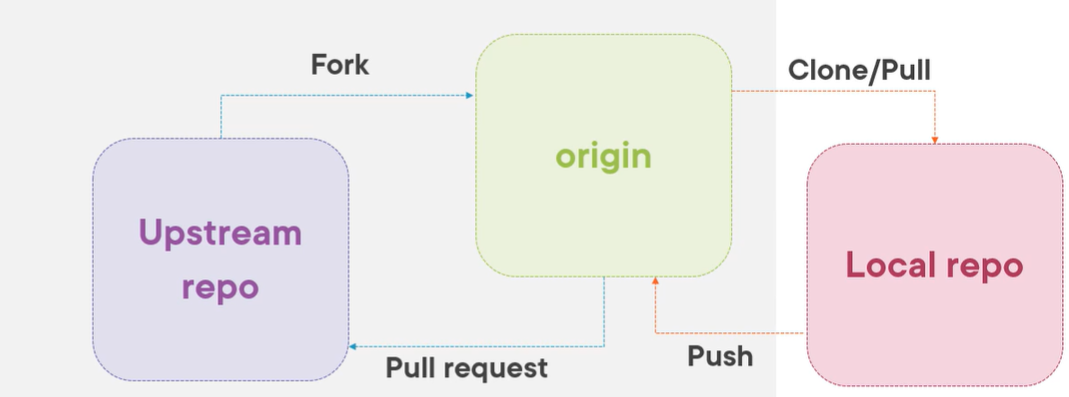
Bij het mergen kunnen er zich **problemen** voordoen indien de twee branches aanpassingen hebben aangebracht aan dezelfde lijn code of er reeds een aangepast bestand werd verwijdert in één van de branches. We moeten dan de problemen **resolven** alvorens we deze kunnen mergen. We kunnen dit oplossen via de GitHub interface of via de command line.

Er kan een **Pull Request (PR) template** worden aangemaakt zodat contributers steeds eenzelfde template invullen voor een PR. Hiervoor moeten we een bestand aanmaken genaamd pull\_request\_template.md. Hierbij wordt vaak gebruik gemaakt van “- []” dat een checkbox aangeeft.

Er kunnen Pull request ge **revert** worden door een pull request te openen dat reeds werd gemerged en deze te reverten. Hiervoor wordt er opnieuw een pull request aangemaakt.

We kunnen branches instellen als een **default branch** en ook reeds eerder verwijderde branches terugzetten. Dit is een default branch waar PR worden voor aangemaakt. We kunnen deze instellen onder settings en dan branches.

**Forks:** het is een kopie van de repo. Het past het orgineel niet aan. Het bijftwel gelinkt zodat we aanpassingen in de orginele repo kunnen binnenhalen. We kunnen ook een PR doen zodat de aanpassingen in de fork terug in het orgineel worden opgehaald. Iedereen kan dit doen en kan achteraf zelf contributers enzo toevoegen aan deze fork. Het is eigen aan GitHub en niet aan GIT.



## Tracking issues and Creating Releases

**Issues** is een naam om vanalle soorten dingen aan te geven dat niet perse negatief zijn. Zo vallen bijvoorbeeld bugs, verbeteringen, taken en ideeen onder issues. Ze kunnen gelinkt worden aan een PR zodat wanneer deze geïntegreerd wordt de issue verwijdert kan worden. We kunnen ook notificaties krijgen indien er iemand een comment plaatst op een issue. Issues kunnen ook gebruikt worden met milestones, dat eigelijk een soort van punt is dat we willen bereiken. Labels kunnen worden toegpast op issues en pull requests. We kunnen op die manier bvb prioriteiten toekennen. Er zijn labels voorzien maar er kunnen er ook extra worden aangemaakt. We kunnen issues assignen aan een bepaalde persoon.

Issues en PR krijgen een orgineel **ID** toegekend. Op deze manier kunnen we een opmerking toevoegen en hierin met #ID een PR aan een issue koppelen of omgekeerd. Wanneer we een PR aanmaken kunnen we aan de hand van “Closes #8” aangeven dat dit issue 8 oplost. Deze issue zal dan automatisc worden gesloten na het mergen van deze pull request.

We kunnen ook een **template aanmaken voor issues**. Dit doen we bij settings en dan naar beneden te scrollen naar issues-> setup templates.

**Milestones** worden gedefinieerd door een groep issues. We zien dan bijvoorbeeld de completion percentage en kunnen prioriteiten toekennen aan issues. Onder issues druk op milestones, dan geven we het een titel en deadline datum. Wanneer we dan naar een issue gaan kunnen we rechts de issue toekennen aan een milestone.

Tags worden gebruik voor het aanduiden van een belangrijke versie van de repo in het verleden. Zo kunnen we bijvoorbeeld aangeven dat een bepaalde versie V1.0 was. Er zijn tee soorten tags, een lightweigt tag, dat gwn een pointer is. Het tweede type is een annotated tag. Dit bevat bijvoorbeeld een datum, message, etc. Dit doen we door git tag te gebruiken. Via deze tag kan op een later moment een bepaalde versie gedownload worden als ZIP via GitHub.

**Releases:** zijn gebaseerd op tags maar hierbij kunnen we release notes toevoegen. We kunnen ook binaire documenten toevoegen dat gerelateerd zijn aan deze release.

## Automating tasks with GitHub Actions

We kunnen taken automatiseren adhv Actions. GitHub wordt gebruikt voor de SDLC (Software Development Life Cycle). Het bevat de planning, de vereiste, .. Een event kan een workflw triggeren. Actions zijn code bestanden dat tot een repo behoren. Ze worden gemaakt in YAML syntax. Er zijn reeds een groot aantal templates beschikaar voor actions. We maken dus een Workflow aan. Het is opgebouwd van één of meerdere jobs. De WorkFlow wordt getriggered door een event. Een event kan bijvoorbeeld een Push, een issue, een PR zijn maar ook een externe trigger van bvb een webpagina. Een job is opgebouwd uit één of meerdere stappen. De uitvoering van deze stappen gebeurd op een Runner. Deze runner is de server dat de automatisatie code zal uitvoeren. Het kan de server van GitHub zijn, maar ook een eigen server.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijvingIn dit geval zal telkens de workflow worden getriggerd indien er iets gepushed wordt in deze repo. De check-bats-version is de naam van de job in de action.Daarachter volgt dat de action wordt gerunt op een ubuntu server. De uses/checkout@v2 zal ervoor zorgen dat een action wordt aangeroepen waarvoor we v2 gebruiken. Dit wilt zeggen dat v2 zal gedownload worden naar de runner. De command uses: actions/setup-node@v1 zal de package manager installeren op de runner. Op die manier krijgen we toegang tot de npm command waarmee we de bats package installeren dat meestal gebruikt wordt om te testen. Dan roepen we het bats commando zelf op en gebruiken -v om de versie van het package te verkrijgen. We moeten deze YAML file in de .github/workflows folder plaatsen.

Afbeelding met tekst

Automatisch gegenereerde beschrijvingGebruiken van npm voor het installeren van netlify en het verkrijgen van de versie van netlfy:

## Types van events:

* GitHub events (push, pull, issue bvb)
* Combination (push or pull bvb)
* Scheduled events (kleinste interval = 5m)
* Webhook events

In de WorkFlow vinden we actions:

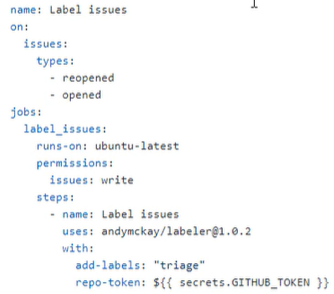
* We kunnen eigen actions maken
* De community actions gebruiken

Types of WorkFlows:

* Automation (bvb automatisch label toekennen aan issue)
* Alles omtrent CI/CD:

1. Continuous integration: continu code samenvoegen om errors snel op te merken
2. Continuous deployment: merged code deployen op een app service

Automatisch labelen issues:



## Creating a github wiki

Wiki: mensen kunnen samenwerken aan kennis over een repo. Op GitHub kunnen we een wiki opzetten voor de repo. Hoe de code te gebruiken enzovoort staat in de wiki. We kunnen de wiki ook offline aanpassen en dan pushen naar GitHub. Er kunnen meerdere pagina’s op de wiki worden aangemaakt (Home, Getting started, …). In de SideBar kunnen we een structuur aanbrengen.

Afbeelding met tekst

Automatisch gegenereerde beschrijvingAfbeelding met tekst

Automatisch gegenereerde beschrijving

=

## Working with social features

Aanmaken van **Gists** en het publiceren van informatie op GitHub pages. Gist is een site van GitHub om snippets (delen van code) en notes te delen. Er zijn public en private gists. Hiervoor gaan we naa gist.github.com

**Projects Board**: een project board geeft een overview van het status vh project. We kunnen een board aanmaken voor een specifieke feature, checklist of een roadmap. Ook hiervoor zijn reeds templates beschikbaar. Via Cards kunnen er issues worden geslepen naar de verschillende geselecteerde kolommen.

**GitHub Pages:** het is een site hosting service. Het is voor static pages, dus er is geen server side code. Ze zijn dus ook deel van een repo. Hiervoor gaan we nr setting 🡪 pages. We kiezen welke branch we willen hosten. Wanneer we dus aanpassingen maken in deze branch, zal dit autmatisch worden aangepast op de site.

|  |  |
| --- | --- |
| **COMMANDO** | **UITLEG** |
| git help | kort overzicht van meest gebruikte functies |
| git config –list | geeft alle ingestelde configs (naam,email) |
| git init | In folder van nieuwe repo. Het maakt een master branch |
| git add . | zet alles in de staging area |
| git commit -m “first commit” | commit alles een eerste keer naar lokale repo |
| git remote add orgin <link.git> | Linken van lokale repo met online github repo |
| git push -u origin master | Push locale repo van branch master |
| git status | zien op welke branch we ons bevinden en of deze al dan niet up to date is met de origin/master branch. We kunnen ook zien welke files in welke stage zijn (committed, staged of modified) |
| git status -s | korte versie van status (M: modified, A: new file added to staging area, ??:  new file untracked by GIT) |
| git status --short | korte versie van status |
| git diff --staged | Welke aanpassingen heb ik gedaan die staged zijn en dus klaar om te worden committed. De aanpassingen staan op de onderste lijn, in de chunk header en chunk changes. Chunk header geeft aan welke lijnen werden aangepast: - is oude content, + is nieuwe content |
| git diff | Welke aanpassingen heb ik gedaan maar niet gestaged |
| git commit -a | shortcut staged area via git add (dus meteen van modified naar committed) |
| git commit -a -m "add new txtfile" | Met een message (geeft aan wat commit betekent) |
| git log | Geeft een overzicht vd geschiednis van de commits |
| git log -5 | Enkel de laatste 5 commits |
| git log --oneline | lijst van al de commits dat elk slechts 1 lijn innemen (beter overzicht) |
| git log --stat | uitebreidere lijst (geeft aan welke files werden aangepast in de commits) |
| git log --patch | Geeft al de aanpassingen in de files in elke commit (de diff) => geruik q om te quitten |
| git rm newFile.txt | Zorgen dat git dit bestand niet meer volgt en dat het verwijdert wordt |
| git rm --cached newFile.txt | Zorgen dat git dit bestand niet meer volgt maar dat het NIET verwijdert wordt |
| git mv README.md README | bestand hernamen |
| git branch [naam] | nieuwe branch aanmaken |
| git checkout -b [naam] | checkout:uitchecken naar een andere branch -b: dit wilt zeggen dat we willen uitchecken nr een  nieuwe branch |
| git checkout master | HEAD verwijst terug nr master branch |
| git stash | Als we de rm --cashed gebruiken, en er dus een untracked bestand is, en we willen veranderen van branch, zou dit bestand overschreven worden, we kunnen verwijderen of terug laten tracken met git add. Indien we dit niet willen, kunnen we deze aanpassing stashen. Dit doen we door eerst git add . te gebruiken en achteraf git stash, Dit wordt dan opgeslagen als een WIP (work in progress) |
| git stash list | lijst van WIP's en hun branch |
| git stash show | Beter gedetailleerd overzicht van WIPs met de aangepaste bestanden |
| git stash pop | De aanpassing wordt teruggeplaatst in de working directory |
| git merge new\_branch | Merge new\_brach into master branch |
| git reset --soft <commit-id> | moves changes back into staging area |
| git reset --mixed <commit-id> | gelijk aan git reset: moves changes back to the workin directory |
| git reset --hard <commit-id> | het verplaatst veranderingen naar de trash ==> opassen!! Veranderingen in working directory of staging area gaan  ook verloren |
| git push origin master | lokale repo nr github |
| git clone <link-repo> | Repo clonen naar ander apparaat van GITHUB |
| git config --global core.editor "code  --wait --new-window" | VS code als gloable editor |
| git fetch | brengt veranderingen in de remote repo dat nieuw zijn sinds clone naar  lokale repo |
| git pull | Een som van een git fetch en git merge. |
| git push -u origin [branch name] | pushes t branch and the -u creates link between local and remote branch |
| git branch -r | overzicht van al de lokale branches |
| git tag | Geeft een overzicht vd tags |
| git tag [naam] [branch] | creeert pointer (lightweight tag) met naam [naam] naar branch [branch] |
| git tag -a v0.1 -m "1.0 Release"  3a06a16 | annotated tag met naam v0.1 en message "1.0 release" naar commit 3aO6a16 |
| git push --tags | om tags naar github te pushen |

# Google Cloud Platform Fundamentals – core infrastructure (GCP)

## Introduction

**GCP voorziet 4 functionaliteiten**.

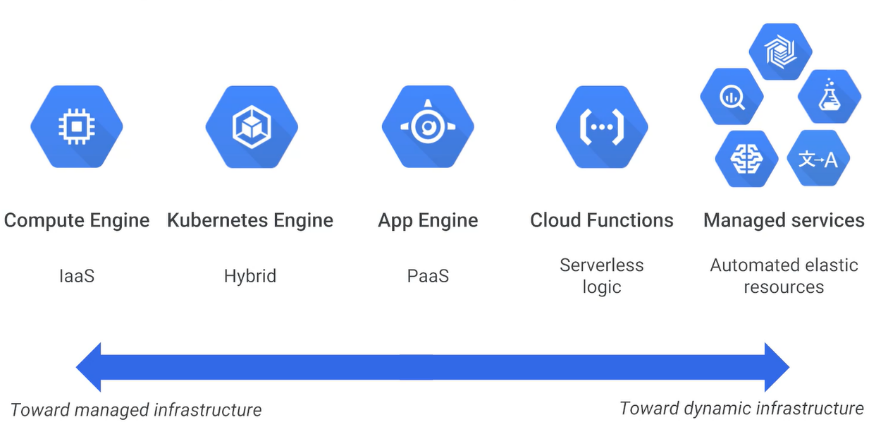
* Compute
* storage
* Big data
* Machine learning

Deze les gaat voornamelijk over de eerste twee hiervan. Eerst wordt er een **definitie van een Cloud** gegeven: Cloud computing is een manier waarbij IT gebruikt wordt en het bevat volgende 5 eigenschappen:

1. We kunnen data gebruiken on-demand en ook self-service. We kunnen een eenvoudige interface gebruiken en we verkrijgende opslag, computing power en netwerk dat we nodig hebben.
2. We kunnen deze steeds bereiken over het net.
3. De provider heeft een overload aan deze resources en verdeelt deze onder de klanten of gebruikers.
4. De gebruiker kan eenvoudig meer of minder resources in gebruik nemen
5. De gebruiker betaalt enkel voor wat hij in gebruik neemt.

**Architectuur van GCP**

Vroeger moest er op voorhand ingeschat worden welke resources men nodig had voor hun toepassing. Dit is vaak gokken. In IaaS, Infrastructure as a service, verkrijgen we computing storage en network, er wordt betaald voor wat we toewijzen. PaaS, Platform as a service, laat toe om de gebruiker code te schrijven en geeft dan toegang tot de benodigde infrastructuur. Hierbij betalen we voor wat we gebruiken. Beide zijn beter dan het oude inschatten van de infrastructuur. IN GCP moeten we hier geen rekening meer mee houden.



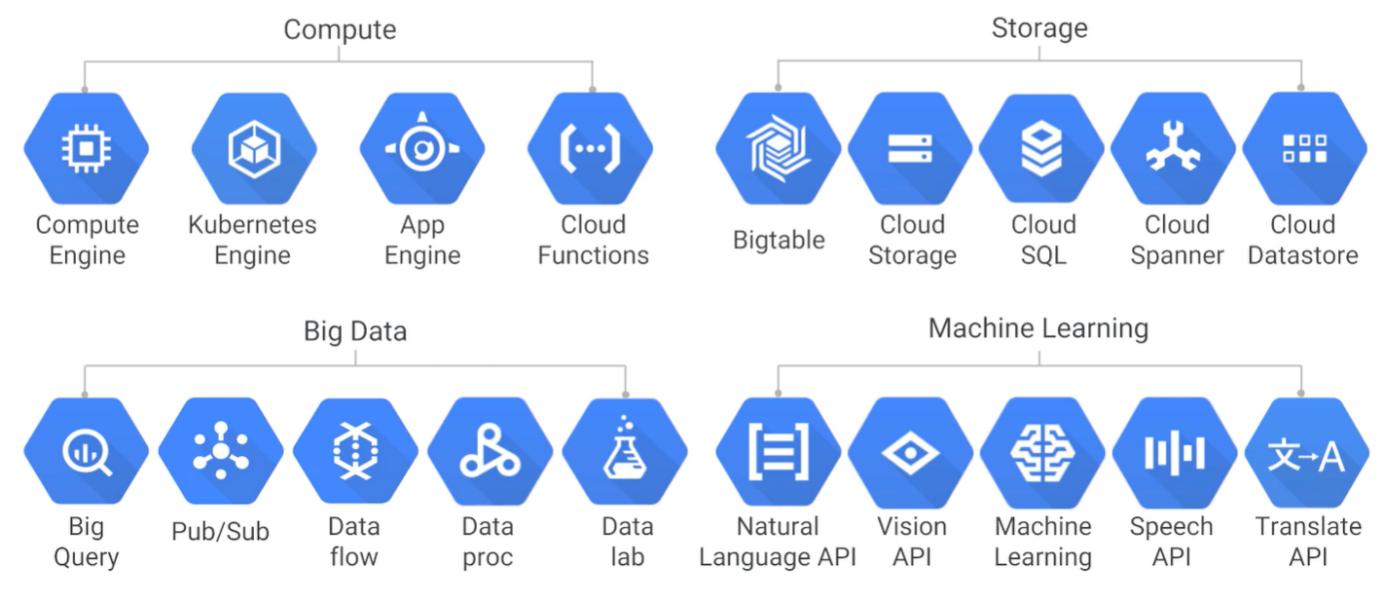
Google heeft een enorm groot netwerk, at ervoor zorgt dat ze vaak een edge point hebben dat dichtbij ligt bij de gebruiker. Hierdoor kan Google enorm snel data bezorgen na een request van de gebruiker. GCP is als volgt opgebouwd: er zijn zones, bvb europe-west2-a, dat behoort tot een regio europe-west, wat op zijn beurt behoort tot de multi-regio europe behoort. Een zone is niet perse gelinkt aan één datacenter, het kunnen er meerdere zijn. We kunnen kiezen in welke regio we onze GCP resources kunnen plaatsen. Sommige bedrijven of instanties kiezen ervoor om hun resources op meerdere zones of regio’s te plaatsen, dat zorgt dat er een soort beveiliging is (stel er valt een datacenter uit) en ook dat de resources zich steeds zo dicht mogelijk bij een gebruiker bevinden (minder latency). Google zal zijn eigen data redundant opslaan in een multi region. Dit wil bvb zeggen dat het minstens 2 keer wordt opgeslagen in verschillende datacenters die zich op een afstand van minimum 160km van elkaar bevinden.

Al de datacenters in de wereld consumeren ongeveer 2% van de elektriciteit. Google doet er dus alles aan om zo efficiënt mogelijk te zijn. In Finland gebruiken ze het zeewater om te koelen bvb.

Google biedt een per seconde facturering aan voor hun Cloud. De meest bekende services van GCP, zoals Kubernetes Engine en Compute Engine, worden heel snel geboud. Voor Compute engine zijn er kortingen indien we deze over een grote tijdspanne gebruiken in de facturatiemaand.

**SAMENGEVAT: GOOGLE CLOUD PLATFORM maakt het mogelijk voor developers om applicaties te ontwikkelen, testen en te implementeren door gebruik te maken van Google zijn veilig, betrouwbaar en schaalbare infrastructuur**

Zoals reeds werd aangehaald, zijn GCP services categoriseerbaar onder

1. Compute
2. Storage
3. Big data
4. Machine learning
5. Networking
6. Operations and tools

Het is heel belangrijk dat Google de data van zijn klanten veilig houdt. Het neemt daarvoor enorm veel maatregelen zoals cryptografie etc.

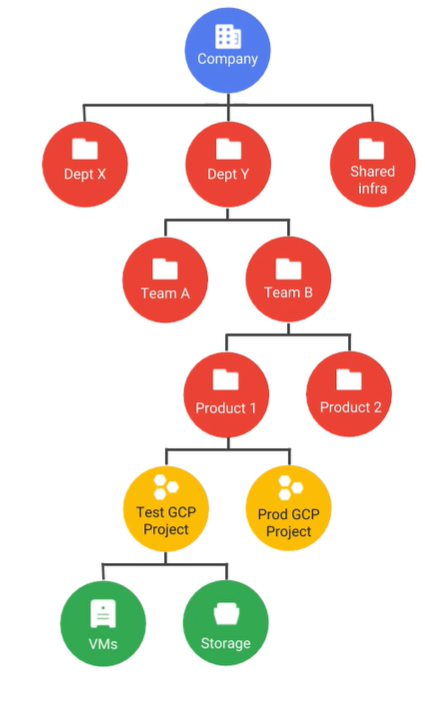
GCP voorziet 4 tools voor het budgeteren en factureren.

1. Budgets & alerts: We kunnen een limiet instellen
2. Billing export zorgt ervoor dat we facturen kunnen exporteren met veel info zodat we ze achteraf gemakelijk kunnen analyseren,..
3. Reports is een visualisatietool voor het verloop van de uitgaven.
4. Quota beschermt gebruikers alsook de hele GCP community. Op deze manier wordt er gegarandeerd dat er geen overgebruik is van resources. Rate quota: 1.000 requests per 100 sec. Allocation quota: max 5 networks per project.

## Getting started

Aan de hand van IAM controleren we wie wat kan doen in ons project. Aan de hand van projects bepalen we welke resources we gebruiken in GCP. We maken gebruik van een least-privilege principe. Dit wilt zeggen dat elke gebruiker enkele de privileges kan verwerven die hij nodig heeft om zijn job uit te voeren. We noemen dit gedeelte de management layer. We kunnen dit configureren aan de hand van de web based console, via de STK en zijn command line, via een API en via een mobile app. We gaan voorrnamelijk ingaan op de webbased console en de command line tools.

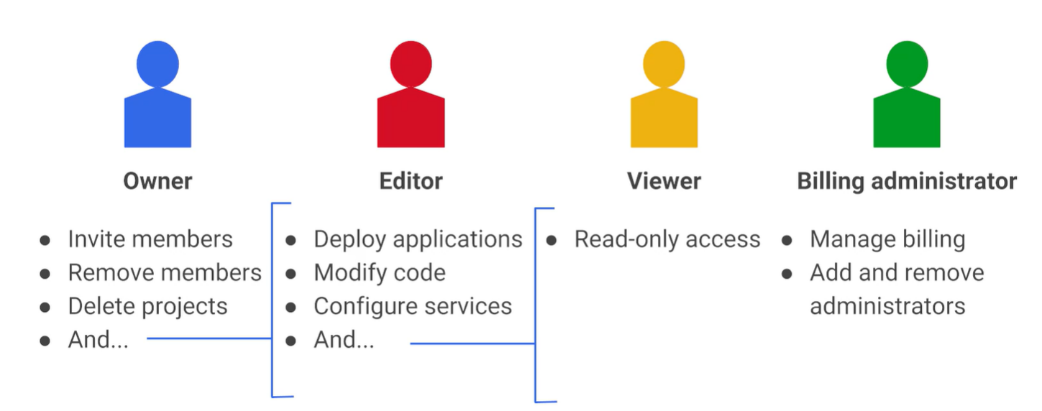
We kunne projecten in folders plaatsen, en folders kunnen andere folders bevatten.



Elke resource behoort tot een project. Elk project heeft een aparte ID, naam en nummer. IAM policies worden meestal giëmplementeerd per folder. We plaatsen dus projects met dezelfde policies in deze folder. Om folders te gebruiken hebben we een organisation node nodig bovenaan de hiërarchie.

Een IAM (identity and access management) bevat een wie gedeelte, een kan wat doen gedeelte en een op welke resource gedeelte. Een kan doen wat geddlte wordt toegekend aan de hand van een IAM role, wat de privileges vastlegt. Er zijn **3 soorten roles**:

1. Primitive: gaat over al de resources in een project: viewer, editer roles en owner roles, owners kunnen ook de billing doen.



1. Predefined: deze laten toe om meer specifieke permissions toe te kennen per service.
2. Custom: deze kunnen makelijk minimum permissions policy implementeren. Deze kunnen enkel gebruikt worden in project of organisation level, niet op filder niceau.

Service accounts laten toe om rollen toe te kennen aan een bepaalde VM ipv personen. We kunnen dan een service account aanmaken zodat authenticatie van deze VM mogelijk is. Zo een service account is ook een resource, waarvan dus ook IAM de rollen bepaalt.

Er kan op 4 manieren interactie zijn tussen de gebruiker en GCP.

1. **Cloud Platform console through the web user interface**

Als we een applicatie maken gebruiken we dit. Het laat ons de projecten en resources beheren. We kunnen ook API’s van GCP bekijken, enablen of disablen. Ten derde geeft het ons ook toegang tot Coud Shell. We kunnen hierbij de tools van SDK (software development kit) gebruiken zonder dat we dit moeten installeren. Er is reeds een tijdelijke VM voorzien.

1. **Cloud Shell and Cloud SDK through the command line**

SDK bevat command-line tools voor GCP producten en services. Bvb gcloud, gsutil (cloud storage) en bq (BigQuery). We kunnen SDK zelf ook installeren ipv de reeds vooziene VM te gebruiken in GCP console. Het is beschikbaar als een docker image.

1. **Cloud Console Mobile APP for iOS or Android devices**

Deze app laat toe om de resources van GCP te managen. We kunnen dashboards maken zodat we snel informatie kunnen verwerken over onze resources.

1. **REST-based API for custom applications**

Programmeerbare access tot producten en services, het gebruikt typisch JSON en OAuth2.0 voor authenticatie en autorisatie. Met de GCP console kunnen we API uit- of aanzetten. Er is een API Explorer in de GCP console om Google API’s te gebruiken.

Stel we hebben een API gevonden, en we willen een APP bouwen dat hier gebruik van maakt. Dan maken we gebruik van de reeds voorziene libraries van GCP. Er zijn twee soorten **libraries**.

1. Cloud client libraries: beheert door de community
2. Google API client Libraries: open source, support voor heel veel programmertalen.

**Google Cloud Launcher:** een tool om snel een functioneel software package te implementeren op GCP. Er zijn free launchers maar ook betalende third party software.

**QWIKLABS**: hiermee kunnen we praktische ervaring opdoen met GCP. Het voorziet ons met GCP credentials zodat we kunnen oefenen zonder kosten. We hebben een timer voor hoelang we GCP kunnen gebruiken.

## Virtual machines in the cloud

Compute engine maakt het mogelijk om een VM te draaien op Google’s infrastructure. Hierbij kunnen we de computing power, het Operating system en memory enzovoort configureren. Een VPC (Virtual Private Cloud) kan men zelf definiëren in een GCP project. Een tweede mogelijkheid is de default VPC te gebruiken. Deze VPC verbindt de gekozen resources met elkaar alsook met het internet. We kunnen het netwerk dan opdelen, firewalls gebruiken om de acces aan te passen en statische routes definiëren naar specifieke destinations. De VPC heeft een global scope. Ze kunnen subnets hebben in elke GCP regio.

Compute Engine laat ons VM’s aanmaken en draaien op Google Infrastructure. Dit kunnen we doen met de GCP console of de gcloud command line tool. Bij het aanmaken van een VM kiezen we hoeveel virtuele CPU’s het heeft alsook het geheugen. GPU’s kunnen we ook kiezen indien nodig. We kunnen ofwel standard disks kiezen of SSD als persistent disks. Een andere mogelijkheid is een local SSD maar deze data gaat verloren indien de VM terminates. We kiezen ook een boot image, er zij veel soorten linux en windows servers beschikbaar. We kunnen een startup scrips schrijven zodat de VM start met een bepaalde configuratie. Zoals het installeren van libraries. We kunnen de data eenvoudig exporteren van deze disks.

Preemptible VM’s laten toe om uitgeschakeld te worden indien er resources ergens anders nodig zijn. Dit zorgt ervoor dat we geld kunnen besparen.

VPC heeft routing tables om workload van de ene naar de andere instantie te versturen binnen het netwerk.

We kunnen VPC’s van twee verschillende projects met elkaar laten communiceren via VPC peering.

Bij VPC kunnen we gebruik maken van Autoscaling om te reageren op een veranderende load. Dit wil zeggen dat er bvb 5 VM’s zijn op één moment en later 40 VM’s. Gebruikers kunnen dan via Cloud Load Balancing toch connecteren.

Cloud CDN kan gebruikt worden om latency te verlagen in het netwerk. Dit doet het door Google’s wereldwijde edge cashes te gebruiken om data op te slaan.

## Storage in the cloud

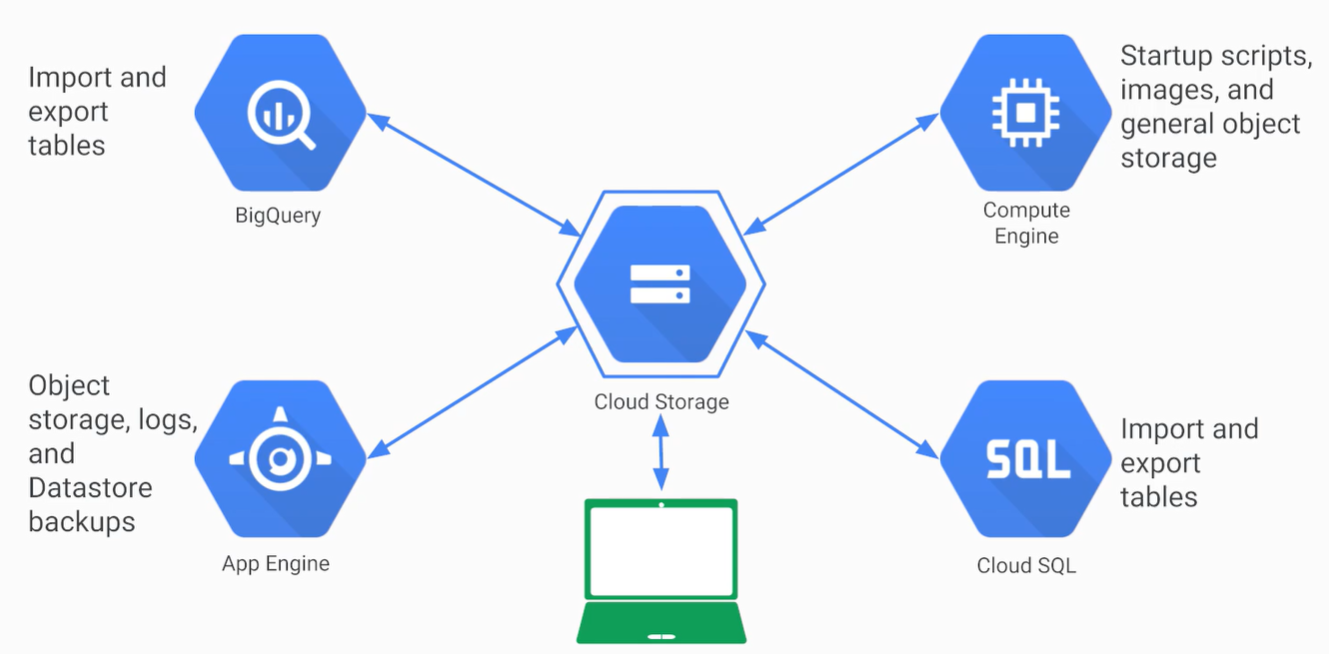
We kunnen gebruik maken van de peristent disks, maar ook van enkele andere manieren om data op te slaan:

* Cloud storage
* Cloud SQL
* Cloud Datastore
* Cloud Bigtable

Afhankelijk van de toepassingen zullen we één of meerdere van deze moeten gebruiken.

**Cloud storage**

Dit werkt zoals object storage. Dit wil zeggen dat we tegen het geheugen zeggen ‘hou deze bytes bij’. We kunnen dit dan opvragen adhv een unieke key. Deze keys zijn vaak url’s. We moeten niet specifiëren hoeveel data we willen opslaan. Een object is gelijkaardig aan een file. CloudStorage zal steeds de data encrypteren alvorens het opslaat. Cloud storage files zijn in buckets geplaatst. Deze verwerven dus de IAM policies. We kunnen dit aanpassen door een ACL te maken (acces control list). We kunnen object versioning aanzetten, waardoor verschillende versies van een object worden bijgehouden en het niet steeds overschrevene wordt.



**Cloud Bigtable**

Dit is Google’s NoSQL Big Data database service. Bij NoSQL moeten niet al de rijen dezelfde kolommen hebben. De tabellen in Bigtable kunnen gigantisch zijn, als in miljarden rijen en duizenden kolommen. Het is ideaal voor data met een unieke key, en zorgt ervoor dat we heel veel data kunnen opslaan maar deze toch kunnen opvragen met een zeer lage latency. Het werkt met dezelfde API als HBase. Bigtable support ook bijvoorbeeld google search en gmail. Data kan via een application API of via batch processing worden toegevoegd of uitgelezen. Het kan ook via streaming worden ingevoegd.

**Cloud SQL**

Dit maakt gebruik van een database schema. Met andere woorden, rijen hebben steeds dezelfde kolommen. Hiervoor kunnen we gebruik maken van mySQL of PostgreSQL. Automatic replication tussen verschillende zones.We kunnen gebruik maken van on-demand of scheduled back-ups. We kunnen de CloudSQL instanties toekennen aan een App Engine application.

**Cloud Datastore**

Dit is een andere schaalbare NoSQL database service. Het wordt vaak gebruikt om gestructureerde data op te slaan van app engine apps. Dit schaalt automatisch door middel van de load te analyseren.

Afbeelding met tafel

Automatisch gegenereerde beschrijvingAfbeelding met tafel

Automatisch gegenereerde beschrijving

## Containers in the cloud

Deze module gaat over het draaien van containers in Kubernetes engine. Het lijkt op IaaS aangezien het ons de infrastructuur opzetten bespaart. Anderzijds is het ook gelijkaardig aan PaaS omdat het is gemaakt met de benodigdheden van developers in het achterhoofd.

**Introductie in containers**

Het idee is om de gebruiker een onafhankelijke schaalbaarheid te geven van workload zoals in en PaaS omgeving en een abstractie van het OS en hardware zoals in een IaaS omgeving. We verkrijgen eigelijk een doorzichte doos rondom onze code en zijn dependencies met een gilimiteerde toegang tot zijn file system en hardware. Zo een container start even snel op als een process in een OS, en is dus veel sneller dan een volledige VM dat er lang over duurt om op te starten. We hebben op de host wel steeds een operating system nodig dat containers ondersteunt. Het komt er dus op neer dat we eigelijk de operating system virtualiseren in plaats van de hardware. Het is schaalbaar zoals PaaS en geeft ons bijna dezelfde flexibiliteit als IaaS. Het maakt de code zeer portable of overzetbaar, aangezien we de hardware en de operating system kunnen zien als een black box.

Stel we willen bijvoorbeeld een server schalen, dan kan dit binnen seconden en bekomen we honderden tot duizenden servers. Normaal willen we veel verschillende containers bouwen om een applicatie te ontwikkelen, waarbij elke container zijn eigen functie uitoefent. De afzonderlijke containers kunnen met elkaar communiceren. Stel dat een host uitvalt, dan zal een andere host meer containers starten en vormt dit geen probleem.

Een tool dat dit helpt te doen is **Kubernetes**, het maakt het gemakkelijk om veel containers op veel hosts te beheren, te schalen, nieuwe versies uit te brengen of teruggaan naar vorige versies.

De meest voorkomende formaat van containers is die dat gebruik maken van de open-source tool Docker. Hiermee gaan we dus een applicatie en zijn dependencies bundelen in een container. (andere tools dan Docker zijn Google Built)

Stel dat we een simpele API in Python, dat gebruik maakt van de FLASK bibliotheek, in een container willen steken. We weten dat er dan een specifieke versie van Python nodig is, alsook een specifieke versie van de FLASK bibliotheek. Dit controleren we door het python requirements.txt bestand.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

We maken dan gebruik van een Dockerfile om te specifiëren hoe de code in een container wordt geplaatst.

**Afbeelding met tekst

Automatisch gegenereerde beschrijving**

We gebruiken ubuntu en installeren Python op dezelfde manier als we zouden doen in onze development omgeving. We kopiëren dan de requirements.txt file en gebruiken het om onze dependencies te installeren. We kopiëren ook de bestanden dan onze applicatie opbouwen. Als laatste zeggen we ook aan de omgeving dat de container lanceert hoe het te draaien.

**Afbeelding met tekst

Automatisch gegenereerde beschrijving**Dan gebruiken we de docker build command om het effectief te bouwen. Dit zal dan een runable image opslaan op de local machine. Achteraf gebruiken we dan de docker run command om het te runnen.

In het echt zal de runable image meestal geupload worden naar een container registry service, zoals de Google container registry, en downloaden we het van daar.

Dus nu is een applicatie in een dockercontainer geplaatst. Indien we echter een betrouwbare, schaalbare en gedistribueerd systeem willen, komt er veel meer aan te pas. Hier komt Kubernetes erbij. Het zorgt onder meer voor configuratie, service discovery, management van updates en monitoring.

Kubernetes is een open-source orchestrator (beheerder, orgnisator) voor containers. Het voorziet een API zodat geauthorizeerde personen de werking kunnen controleren door middel van enkele tools. Een hiervan is de kubectl command. Kubernetes laat toe de container te implementeren op een set van nodes, genaamd een cluster. Zo een cluster bestaat uit een set van master components, dat het systeem controleren,en meerdere nodes waar de containers op draaien. Een node representeert dus een computing instance. In Google cloud zijn nodes VM’s dat draaien in cumpute engine. Om Kubernetes te gebruiken, kunnen we een set van applicaties omschrijven en hoe deze met elkaar moeten interageren. Het is dan Kubernetes die uitzoekt hoe het dit moet realiseren. Het maakt het dus gemakkelijk om containers te gebruiken.

Google Cloud voorziet Google Kubernetes Engine, dit is dus Kubernetes als een beheerde service in de cloud. We kunnen dus een cluster maken met de GCP console of met de gcloud command dat wordt voorzien door de cloud SDK. GKE clusters kunnen worden aangepast en ondersteunen verschillende soorten machines, aantal nodes en netwerk instellingen. Aan de hand van gcloud kunnen we eenvoudig een cluster maken met volgend commando.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Er is dan reeds een volledige cluster dat werd geconfigureerd en klaar is voor gebruik. Telkens wanneer Kubernetes een container opstart of een set van gerelateerde containers, doet het dit in een abstractie genaamd een pod. Het is een soort van draaiend proces op je cluster. Het kan een component zijn van je applicatie of een gehele applicatie. Meestal plaatsen we maar één container in een pod. Elke pod krijgt een uniek IP adress en een aantal poorten in Kubernetes. Éen manier voor het runnen van een container in een pod in Kubernetes is het kubectl run commando. Een deployment is dan een groep van replicas van dezelfde pod. Het zorgt ervoor dat deze blijven draaien. Om te draaiende pods te zien gebruiken we het kubectl get pods commando. Deze pods zijn default enkel toegankelijk van binnen de cluster. Indien we deze openbaar willen maken, zodat bvb mensen vanop het internet de inhoud kunnen gebruiker, moeten we gebruik maken van de kubectl expose command. Kubernetes maakt dan een service aan met een vast IP-adress voor de pods. Deze service zorgt voor de load balancing van Kubernetes. Elke klant zal dus gerouteerd worden naar een pod in de service. De service groepeerd dus een set van pods en voorziet een stabiel eindpunt voor ze.

## 

Deployments zullen dus pods aanmaken en vernietigen. Pods krijgen hun eigen Ip-adressen maar deze blijven niet stabiel in de tijd. Services voorzien dat stabiele eindpunt dat we nodig hebben. Kubectl get services geeft ons hetstabiele externe IP-adress. Om een deployment te scalen maken we gebruik van het kubectl scale commando. We kunnen autoschalen aan de hand van het kubectl autoscale commando.

In de realiteit zullen we echter een configuratiebestand voorzien aan Kubernetes om te specifiëren hoe we willen dat de staat van de cluster eruit ziet.



Na het aanpassen van een config file moeten w de kubectl apply command gebruiken. De kubectl get replicas command kan gebruikt worden om de replicaten te zien. Aan de hand van kubectl get deployments zien we welke draaien.

Indien we de code willen updaten, willen we ervoor zorgen dat de users geen downtime ondervinden. Daarom kunnen we in de Kubernetes config file een update strategy definiëren. Zo is er bijvoorbeeld een rollingupdate methode. Dit wil zeggen dat enkel de nieuwe pods de update bevatten.

Containers in Hybrid and multicloud architectures

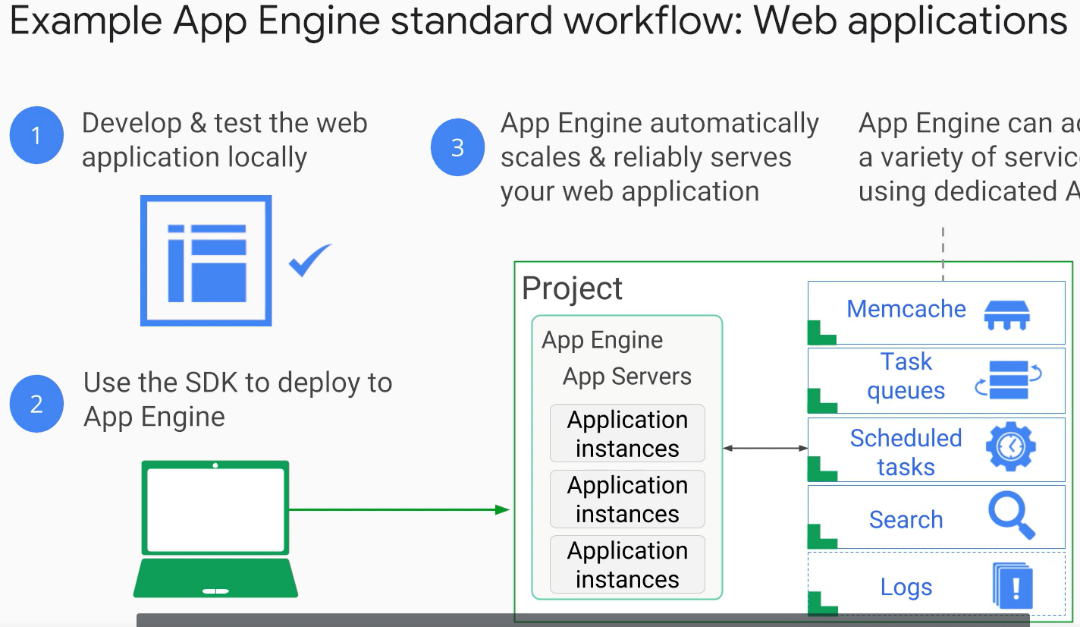
Het zorgt ervoor dat we delen van de reeds huidige architectuur kunnen blijven gebruiken terwijl we andere delen naar de Cloud kunnen verhuizen. Anthos is de moderne oplossing van Google voor hydbrid and multi-cloud systemen en services management. Het maakt gebruik van Kubernetes en Kubernetes engine.

## Applications in the cloud

We hebben twee GCP producten besproke dat de compute infrastructuur voor applications: Compute engine en Kubernetes engine. Wij kiezen de infrastructuur waarop onze applicatie runt. Het zijn VM’s op compute engine en containers op Kubernetes Engine. Stel dat we niet op de infrastructuur willen focuseen maar enkel op de code. Daarvoor is **App Engine**. Heriner dat PaaS staat voor Platform as a service. Bij App engine leveren we onze code aan en zal app engine de rest doen. App engine schaalt automatisch naar mate het verkeer dat hij binnen krijgt. E betalen dus enkel voor de resources dat we gebruiken. Het is geschikt voor fluctueel verkeer zoals web applications. App engine biedt twee werkomgevingen aan, namelijk standaard en flexibel.

**1 Standaard**

Dit is eenvoudig. Het is eenvoudiger te deployen, zal de workload autoscalen, heeft een gratis daily quota, wat wil zeggen dat applications met weinig verkeer gratis kunnen zijn. We gebruiken een runtime dat voorzien is door Google. Het voorziet specifieke runtimes voor Java, Python,PHP en GO. Willen we coderen in een andere taal, dan moeten we gebruik maken van de flexibele werkomgeving. Deze standaard werkomgeving zet limieten op je code aangezien het werkt in een sandbox. Dit is software dat onafhankelijk is van de hardware, operating system en fysieke locatie van de server. Dit is de reden dat de autoscaling zeer goed werkt. De applicatie kan echter niet schrijven naar het locale file system, maar moeten naar een database service schrijven. Al de requests dat onze applicaties ontvangen hebben een 60s timeout. Er kan ook geen third party software geïnstalleerd worden.



**2 Flexibel**

Geen sandbox 🡪 we kunnen de container zelf definiëren.

**Afbeelding met tafel

Automatisch gegenereerde beschrijving**

Er zijn twee API management tools in GCP. We kunnen bijvoorbeeld kiezen welke versie van de API gebruikt wordt in een call**. Cloud Endpoints** zijn er voor het bepalen welke end user de call maakt. Een tweede tool is **Apigee**. Dit is eerder gemaakt voor een business doeleinde.

## Developing, Deploying and Monitoring in the cloud

Veel gebruikers van GCP gebruiken GIT voor het opslaan en managen van hun source code. Dit is geweldig maar stel dat er een manier is om code privé tehouden en IAM permissions te gebruiken om het te beveiligen. Hiervoor wordt **Cloud Source repositories** gebruikt. Het geeft GIT service control. Je krijgt een oneindig aantal privé git repos. Het bevat ook een file explorer in GCP. Er bestaat ook zoiets als cloud functions om functies te integreren. We kunnen bijvoorbeeld een geuploade foto processen zonder dat we moeten denken aan servers of runtimes. We schrijven code in JavaScript in een Node.js omgeving dat GCP voorziet. We beslissen dus op welk event we willen reageren, ook wel triggers genoemd. Achteraf koppelen we dan een JavaScript function aan deze triggers.

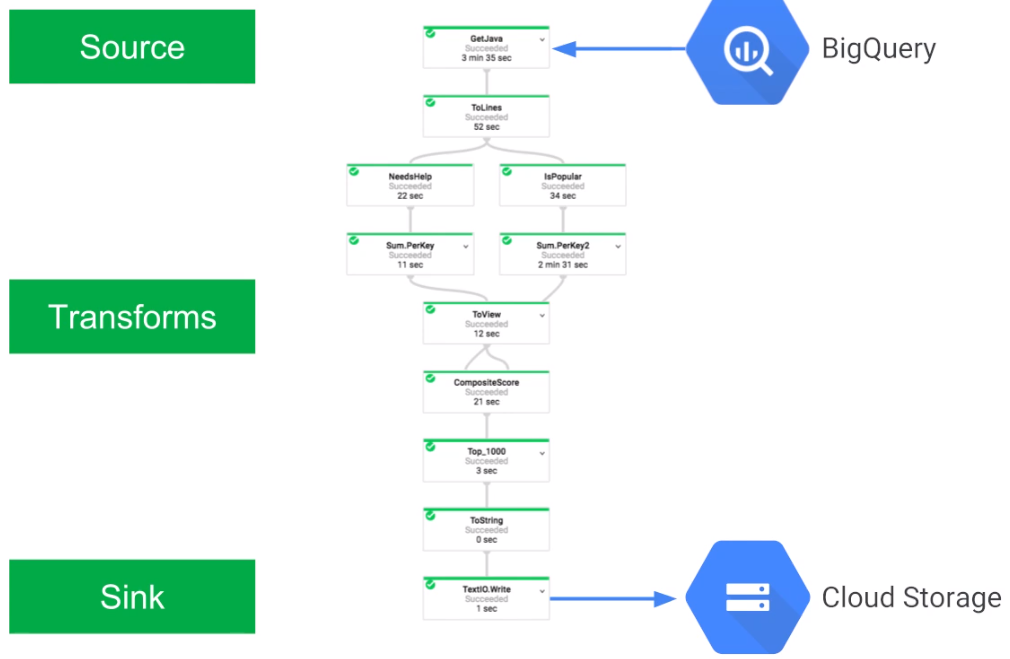
**Deployment manager** is een descriptieve manier voor het bepalen van de resources. Hiervoor maken we een template file in YAML of in Python waarin we beschrijven hoe we willen dat de componenten van onze omgeving eruit zien. We geven deze template dan aan de Deployment manager.

Monitoring van een application is heel belangrijk. **Stackdriver** is de tool van GCP voor monitoring , logging en diagnosticeren. Monitoring, loggging, trace, error reporting, debugger, profiler zijn de onderdelen.

## Big Data and Machine learning in the cloud

Dit gaat over de technologiën van Google om zo snel mogelijk het meest uit data te verkrijgen. Het kan zowel real-time analyses zijn als machine learning. Het is gemaakt zodat het eenvoudigkan worden toegepast op projecten van gebruikers. De gebruiker moet geen instanties voorzien voor computing, maar betaalt voor de resources dat hij gebruikt. Apache Hadoop is een open-source framework voor big data.

**Cloud Dataflow** is goed voor het verwerken van real-time data of data van ongekende grote. Hiervoor bouwen we dan pipelines.



Indien we van een dyname pipeline willen we bijvoorbeeld SQL queries uitvoeren op een immense dataset. Daarvoor gebruiker we **BigQuery**. Er wordt bepaald per Query.

**Cloud pub/sub** is voor het asynchrone versturen en verzenden van berichten tussen applicaties.

**Cloud Datalab** is Jupyter maar dan zonder de benodigdheid van computational power. Opnieuw is het enkel betalen van de gebruikte resources.

**Cloud Machine learning Platform** bevat reeds veel getrainde neural networks etc.

**Cloud Vision API** gebruikt een API om objecten in afbeeldingen te herkennen.

**Cloud natrual language API** herkent talen en kan het uitschrijven.

**Cloud Translation API** vertaalt een arbitraire string en herkent de taal.

**Cloud Video intellegence API** is om inhoud van videos te bepalen

# FastAPI

## Intro

* It is one of the fastest (high-performance) web frameworks available.
* It takes advantage of type annotation support of Python 3.6+ for better data validation and editor support.
* It utilizes Python's Async power, which is useful for building asynchronous APIs. 🡺 meerdere requests op zelfde moment afhandelen!!
* It generates standards based on open standards such as [OpenAPI](https://www.openapis.org/) and [JSON Schema](https://json-schema.org/).
* It is robust and easy to use and learn.
* Sneller dan FLASK
* Is gebouwd op Starlette

## Installation

* Pip install fastapi
* Nood aan een ASGI (Asynchronous Server Gateway Interface) zoals uvicorn 🡪 pip install uvicorn

## Simpele API

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read\_root():

    return {"hello": "world"}

in terminal naar map gaan dat dit main.pu bestand bevat en volgend commando uitvoeren:

uvicorn main:app --reload

Dan start de server en zien we het webadress waarop we dit kunnen zoeken.

## Path Parameters

We kunnen waarden in het webadress (de API call) gebruiken als parameter. In dit voorbeeld is course\_id een path parameter.

@app.get("/course/{course\_id}")

2def my\_course(course\_id: int):

3 return {"course\_id": course\_id}

Als we dan bijvoorbeeld [127.0.0.1:8000/course/2](http://127.0.0.1:8000/course/2) schrijven in de browser, krijgen we:

{"course\_id":2}

Merk op dat course\_id een integer moet zijn en we anders een error zullen krijgen.

## Query parameters

We kunnen ook parameters meegeven aan de API als query parameters zoals hieronder.

/my/page/items/?page=1&limit=10&order=0

We kunnen dan de parameters page, limit en order gaan gebruiken zoals hieronder

dummy\_data = [i for i in range(100)]

2

3@app.get("/my/page/items/")

4async def read\_item(page: int = 0, limit: int = 0, skip: int = 1):

5 return dummy\_data[page\*10: page\*10 + limit: skip]

## Request body

Als we data van de client nr de server willen sturen kunnen we de gebruik maken van de resuest body aan de hand van de post methode. Een veel gebruikte bibliotheek hiervoor is Pydantyc.

from pydantic import BaseModel

2

3class MyItem(BaseModel):

4 name: str

5 info: str = None

6 price: float

7 qty: int

8

9@app.post("/purchase/item/")

10async def create\_item(item: MyItem):

11 return {"amount": item.qty\*100, "success": True}

Als er dan een POST request is op endpoint /purchase/item/ dan zal het de attribuutwaarden aannemen en antwoorden met succes.

Dit kan worden getest aan de hand van de curl functie.

curl -X POST "http://127.0.0.1:8000/purchase/item/" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"name\":\"sample item\",\"info\":\"This is info for the item\",\"price\":40,\"qty\":2}"

dan krijgen we:

{"amount":200,"success":true}

## Form data

We kunnen in veel situaties form fields verkrijgen ipv JSON berichten. Dan zullen we gebruik maken van de FORM bibliotheek. Waarvoor eerst python-multipart moet worden geïnstalleerd.

We kunnen bijvoorbeeld username en password als form field nemen:

from fastapi import Form

2

3@app.post("/accounts/login/")

4async def login\_view(username: str = Form(...), password: str = Form(...)):

5 return {"success": True}

## Automatic docs

Zowel SwaggerUi als ReDoc maken het moeglijk om de API calls weer te geven in de browser.

# Pycharm

Deze IDE heeft veel voordelen ten opzichte van anderen. Het kanprojecten openen van elke map, of aparte files openen. Een leuke feature is het hernoemen van een variabele over heel de code 🡪 hiervoor gaan we de variabele highlighten en dan rechtermuisklik 🡪 refactor 🡪 rename. Dit werkt steeds in een scope. In een functie zal het dus enkel in de functie aanpassen.

CTRL+F : zoeken in code

ALT + SHIFT + C: veranderingen in code weergeven

CTRL + D: dupliceert lijn waar we op zijn

Vanonder is er een TO DO list zodat we het niet veregeten. We kunnen in de code dan schrijven ‘# todo afmaken functie’ dan zal de ‘afmaken functie’ in de todo lijst komen en waar deze zich bevind in de code

# Pre Commit

# When I commit staged Python files, before committing pre-commit hooks are executed - isort sorts imports then black formats the code and at last flake8 checks code compliance with PEP8. If every check passes, the commit is made else, code is automatically formatted and sent back for review. I then just review the changes made (and perform any other changes if required) and commit again. So, less time is spent on code formatting and more on code logic & implementation.

# To use pre-commit:

* Add pre-commit to requirements.txt or Pipfile.
* Create .pre-commit-config.yaml with hooks needed in the pipeline.
* At last, execute pre-commit install this will install git hooks in .git/ directory of the project.

# [Keeping python code clean with pre-commit hooks: black, flake8 and isort | by Rohit Gupta | Staqu Dev Logs | Medium](https://medium.com/staqu-dev-logs/keeping-python-code-clean-with-pre-commit-hooks-black-flake8-and-isort-cac8b01e0ea1)