QtechNG: Ontwikkelen voor Brocade

Mijn blog behandelt de design van QtechNG en de rol die dit instrument speelt bij het vervaardigen van software voor Brocade.

Deze blog kan je ook raadplegen als PDF

Blog

Deze *blog* gaat over *Brocade* en *qtech(ng)*.

Er zijn verschillende redenen waarom ik deze blog opstart:

- In de eerste plaats wil ik discussieteksten aanreiken. In ben volop bezig met de constructie van de nieuwe `qtech`. Dit is een belangrijk, veelzijdig instrument dat voor een groot gedeelte het comfort van de ontwikkelaars van Brocade bepaalt. Deze discussies kunnen de uitbouw van `qtechng` sterk beïnvloeden. Het stopt echter daar niet: zoals elk goed instrument, reikt `qtechng` verschillende manieren van ontplooiing aan. Dit leidt tot keuzes en deze keuzes leiden soms tot consequenties die verder gaan dan persoonlijke inzichten.
- Deze blog wil ook openstaan tot inzichten die komen uit het Anet-team: elke tekst, behoorlijk geformuleerd en relevant tot de discussies, is welkom!

Er zijn ook redenen die zich eerder aan de rand bevinden maar daarom niet mnder belangrijk zijn:

- Het gebruik van Hugo als motor van een statische website.
- Het gebruik van responsive technologieën.
- Het gebruik van GitHub en GitHub Pages

Deze blog wordt geschreven in Markdown en meer bepaald in het dialect CommonMark dat streeft naar een rigoureuze specificatie van markdown.

De afbeeldingen zijn in SVG. Ook daar wil ik kijken in hoeverre ik *SVG* met tools als Inkscape naar mijn hand kan zetten.

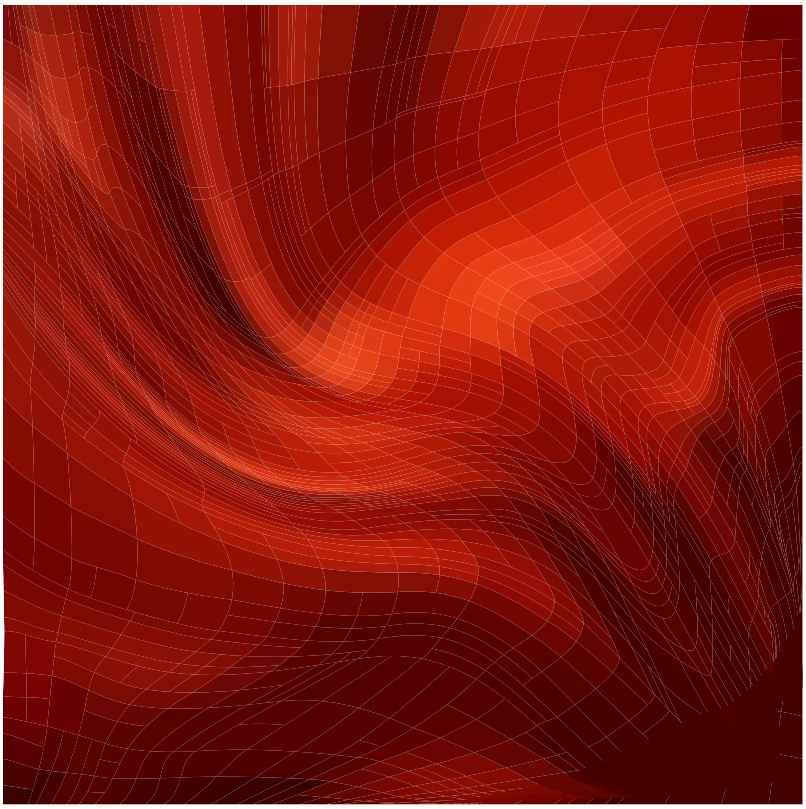
Hugo boeit me enorm. Niet alleen is het een software die zich schitterend kwijt van de hoofdtaak: produceren van statische websites, het is ook een uitmuntende software geschreven in Go.

Ik koos ervoor om te gaan voor een sobere blog. Het Ananke them zorgt daarvoor.

Normaal gezien zou deze pagina moeten verschijnen als eerste in de reeks. De waarheid gebied me te vertellen dat dit nummer 12 is.

Dit heeft alvast als voordeel dat ik kan terugkijken op het succes van de gekozen technologieën: ze voldoen allemaal. Toch imponeert `Hugo` het meest: in het bijzonder de preview mode en de snelheid waarmee aanpassingen ogenblikkelijk zichtbaar zijn, zijn zonder meer indrukwekkend. Ook de combinatie `Git`/`GitHub`/`GitHub Pages` is een plezier om mee te werken.

Brocade



Deze blogbijdrage gaat over **Brocade**.

Ik ga niet uitleggen wat de functionaliteit is die de *Brocade software* aanbiedt: daarvoor kan je veel beter terecht op de Wikipedia pagina.

Deze tekst gaat over de software zelf: wat designprincipe zijn, hoeveel software er is en hoe deze wordt beheerd.

Het belangrijkste principe is wel dat *Brocade* een werk van lange adem is. De wereld van (wetenschappelijke) bibliotheken en archieven verandert razendsnel. Dit gaat samen met de aard van informatieverschaffing maar evengoed met het veranderend statuut en aanzien van wetenschapsbeoefenaars.

Dit betekent dat het software park voortdurend moet worden aangevuld met instrumenten om die nieuwe uitdagingen passend weerwerk te kunnen bieden. De eerste release van Brocade kwam er op 1 januari 2000 (om meteen ook het jaar 2000 probleem te tackelen) en in de 28 major releases die daarop volgden is er nog geen software weggevallen (soms wel vernieuwd).

Het is echter ook nog op een andere manier een werk van lange adem: onze

bibliothecarissen en archivarissen krijgen er nieuwe taken bij maar de *oude* blijven evengoed bestaan: de klassieke bezigheden zoals beheer van de aanwinsten, meta datering en het ter beschikking stellen van de materialen, blijven belangrijk en diversificeren zelfs. Dit betekent nog dat weinig bibliotheken hun software pakket veranderen. De meeste klanten van Anet zijn dat al vele, vele jaren.

Het Anet team, verantwoordelijk voor de uitbouw van Brocade, kan je opsplitsen in 2 groepen: software ontwikkelaars en mensen die functionele ondersteuning geven. Die grens is niet scherp te trekken: in de praktijk moet iedereen toegang hebben tot de source code en ook de mogelijkheid hebben om aanpassingen aan te brengen.

De Brocade software is opgesplitst in **projecten**. Hoewel iedereen aan dergelijk project kan werken, is er toch altijd een ontwikkelaar die het voortouw neemt en het project onderhoudt.

Release 5.00 van Brocade telt **1208** projecten. Deze projecten bevatten samen **14036** bestanden. De source code - zonder afgeleide producten - bedraagt **462M**.

Een overzicht van de voornaamste soorten bestanden: (enkel de bestandstypes waarvan er meer dan 100 bestaan, worden opgenomen)

#	Extensie
116	.svg
168	.js
173	.xml
181	.css
207	.txt
224	.html
254	.yml
267	.phtml
404	.l
445	.b
466	.cfg
497	.gif
532	.png
585	.d
765	.x
1040	.rst
2714	.py
3116	.m

In een latere blog bijdrage ga ik zeker dieper ingaan op deze bestandstypes en hoe projecten worden beheerd.

[mlq]tech(ng)?

mtech/ltech/qtech/qtechng is het **beheersinstrument** van de Brocade software.

Al heel snel bleek dat we een adequaat instrument nodig hadden om de bibliotheeksoftware te beheren. Dit was al zo in het begin van de jaren '90 toen we nog automatiseerden met Vubis.

Ook in die dagen hadden we al heel wat software die eigen was aan de Universiteit Antwerpen en de bibliotheken die toen deel uitmaakten van Vubis Antwerpen: er was natuurlijk het Impala project met VirLib, Agrippa (onze archiefsoftware). Ook de deelname aan Europese projecten legden eisen op aan de software die nog moeilijk handmatig konden worden ingevuld.

Taken opgenomen door qtech

Beheersen van software betekent in onze context verschillende dingen:

- Hulp tijdens het ontwikkelproces

- Integriteit van het code repository bewaken
- Samenwerking mogelijk maken
- Eenvoudige ontplooiing op diverse servers
- Administratieve en beheersmatige informatie verschaffen

Hulp tijdens het ontwikkelproces

Het staat ontwikkelaars vrij om hun ontwikkeltools te kiezen. Er zijn ondertussen verschillende programmer's editors de revue gepasseerd: Crisp, Brief, Emacs, XEmacs, Textadept, Sublime Text, Visual Studio Code. Maar ook meer geavanceerde instrumenten zoals Eclipse en Netbeans werden (kort) gebruikt in het ontwikkelproces.

De verscheidenheid aan technologieën, de diverse OS waarop ontwikkelaars werken samen met hun persoonlijkheid zorgt ervoor dat er geopteerd wordt voor eerder eenvoudige tools waarvan hun functionaliteit dan wordt aangevuld met behulp van extensies. Visual Studio Code is daar een goed voorbeeld van. 10 speciaal ontworpen extensies zorgen ervoor dat de Brocade specifieke bestanden snel kunnen worden opgespoord en bewerkt.

Integriteit van het code repository

Qtech moet ervoor zorgen dat de integriteit en de eenheid van de software duidelijk is: zowel voor ontwikkelaars als voor het management. De software is *niet* OpenSource en zorgt voor een belangrijke bron van inkomsten voor de Universiteit Antwerpen en haar bibliotheek.

Samenwerking mogelijk maken

Er werken diverse mensen samen aan het software repository. Het zijn niet allemaal ontwikkelaar en ze spelen allemaal een belangrijke rol bij de uitbouw van Brocade: aanpassen van verwoordingen, opbouw van de menustructuur, vertalingen, design van webpagina's …

Deze diverse groep moet vlot kunnen werken aan de source en niet gehinderd worden door afspraken en handelingen die ver buiten hun expertise terrein liggen.

Eenvoudige ontplooiing op diverse servers

Installatie van major releases en bugfixes moeten gemakkelijk kunnen worden doorgetrokken naar de diverse productieservers, demo- en opleidingsservers.

Deze acties moeten zowel interactief als 'scheduled' kunnen plaatsvinden.

Installaties en updates komen soms ook met contractuele verplichtingen en moeten dus ook kunnen worden hard gemaakt in de vorm van 1 bestand (bijvoorbeeld gedeponeerd bij een notaris)

Administratieve en beheersmatige informatie verschaffen

Er is geregeld noodzaak om te weten hoeveel aanpassingen er zijn aangebracht en door wie.

Dit gaat soms verder dan wat een versie controle systeem standaard kan aanleveren.

Een stukje geschiedenis

De eerste versie van de software heette *mtech*.

Deze software werd geschreven in *C* en construeerde een Makefile. De functionaliteit was beperkt en richtte zich tot ontwikkeling op SCO UNIX. De enige bestandstypes die werden ondersteund waren Mumps files.

Er kwamen voortdurend uitbreidingen en tenslotte verscheen een tweede versie die ook gebruik maakte van AWK. Met deze versie kwam er ook interesse van buitenaf en *mtech* werd ook gebruikt bij tutorials en congressen rond de Mumps programmeertaal. In deze versie werd ook voor de eerste keer gebruik gemaakt van macro's door middel van de m4 preprcoessor

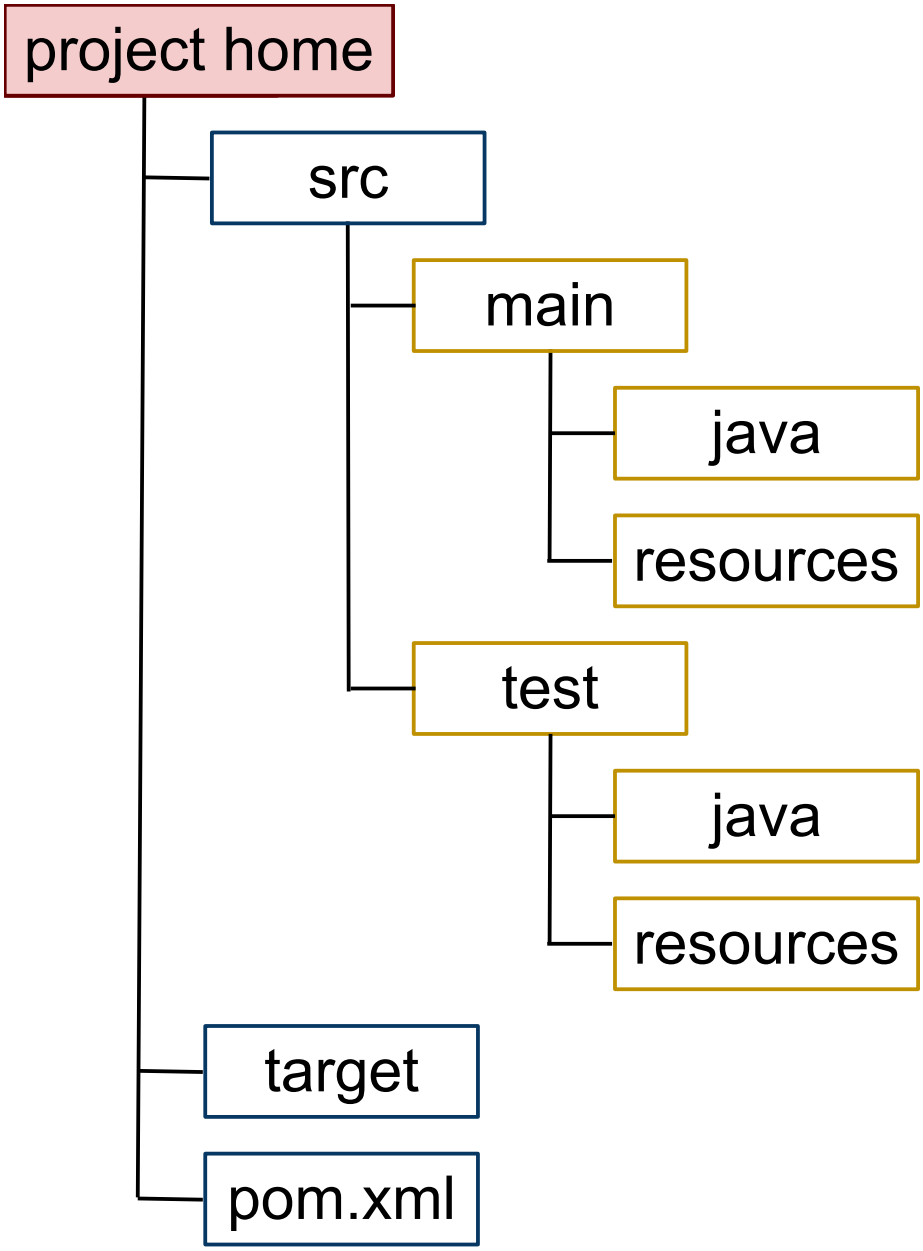
Toen Vubis Antwerpen deelnam aan het HyperLib project was het onmogelijk om nog langer te blijven werken met een instrument die zich richtte op 1 soort technologie. De software werd nogmaals herwerkt maar nu in Perl. Tevens werd ook de naam veranderd naar *ltech*. De oude naam botste immers met een populaire OpenSource tool die beschikbaar was onder Linux.

Toen Brocade eraan kwam was het weer tijd om het instrument aan te passen. Alle voorgaande versies waren gericht op file sharing: de ontwikkelaars werken op Windows workstations en deelden bestanden op de ontwikkelserver. Dit was niet langer houdbaar: filesharing veroorzaakt diverse problemen en was ook moeilijk om vanop afstand in te zetten. De software kreeg als naam *qtech* en werd ontwikkeld in Python. De software werd gesofistikeerd: transport gebeurde op basis van SSH, er waren diverse soorten servers, projecten, registry.

In 2003 werd deze software aangevuld met *wxqtech* een GUI op basis van wxpython. Tot op de dag van vandaag gebruikt iedereen binnen Anet deze software.

Qtech werd ook voortdurend geintegreerd in geavanceerde programmer's editors.

Projecten



Projecten: een stand van zaken

Brocade 5.00 telt meer dan 14000 bestanden.

Om beheer mogelijk te maken, moet hier structuur worden aangebracht. Het voornaamste structuurelement is het *project*. Een project staat voor een deelverzameling van deze bestanden, fysiek ondergebracht in een directory.

Van oudsher kent Brocade een thematische opsplitsing: zo werden bestanden rondom het bestellen van dingen gegroepeerd in projecten met namen als: `/requests/impala`, `/requests/repository`, `/requests/virlib`. Deze werden dan aangevuld met minder voor de hand liggende projecten zoals `/requests/admin`, `/requests/gateway`, …

Deze projecten hebben gemeen dat ze gaan over het bestellen van artefacten uit de bibliotheekwereled, met alles wat daarbij komt kijken. Niet zelden is er een hoofdontwikkelaar die gaat over deze projecten. Kennis en creativiteit vergaren is immers dikwijls een werk van jaren.

De meeste projecten in Brocade 5.00 hebben 2 componenten (vb. "requests" en "virlib"). Dat komt gewoon zo goed uit en is zeker geen verplichting. In `qtech` kan een project nooit een directory bevatten: enkel bestanden zijn welkom.

Er zijn ook andere organisatievormen mogelijk dan de thematische. Er zijn reeds voorstellen gedaan om de projecten op te splitsen naar de gebruikte technologie: sommige projecten zijn in `python2`, andere in `python3` of `mumps`.

Indeling naar technologie heeft grote voordelen voor het ontwikkelproces. Anderzijds zijn de meeste projecten heterogeen: `Mumps`, `Python`, `reStructuredText` … ze komen dikwijls samen voor.

`qtech` had hiervoor een oplossing met `properties`: files en projecten konden worden gelabeld en, op basis van deze labels, worden uitgecheckt/ingecheckt/verwerkt. Er was zelf een bestandstype (`*.q` files) waarmee eigenschappen *automatisch* konden worden toegekend.

In die kleine 20 jaar dat `qtech` in gebruik was, ben ik ongeveer de enige die deze feature heeft gebruikt. Overigens, met weinig succes!

In `qtechng` wordt de definitie van een project aangescherpt.

QtechNG: projecten kunnen directories bevatten

Moderne software projecten komen in een directory tree. Node, Go, Hugo, reStructuredText … noem maar op: ze kennen allemaal het concept van project en ze zijn in boom-vorm. Dikwijls gaat de ondersteunende software hier ook handig op inspelen.

Als dergelijke projecten een plaats moeten krijgen binnen `qtechng`, dan moet deze boomstructuur ook kunnen.

QtechNG: projecten kunnen kind-projecten bevatten

`qtechng` gaat nog verder: projecten kunnen ook kind-projecten bevatten.

Dit heeft grote implicaties (en ook grote mogelijkheden) over de installatie van een project op een systeem.

In een latere blog post gaan we dit uitvoerig bespreken.

QtechNG: brocade.json

Elk project moet in zijn root-directory het bestand `brocade.json` bevatten. Dit bestand vervangt het vroegere `install.cfg`. Er werd een andere naam gekozen omdat het dan voor bijvoorbeeld Visual Studio Code onmiddellijk duidelijk was hoe dergelijke bestanden moesten worden bewerkt.

In tegenstelling tot `install.cfg`, is `brocade.json` verplicht! Het aantal mogelijkheden is ook drastisch uitgebreid.

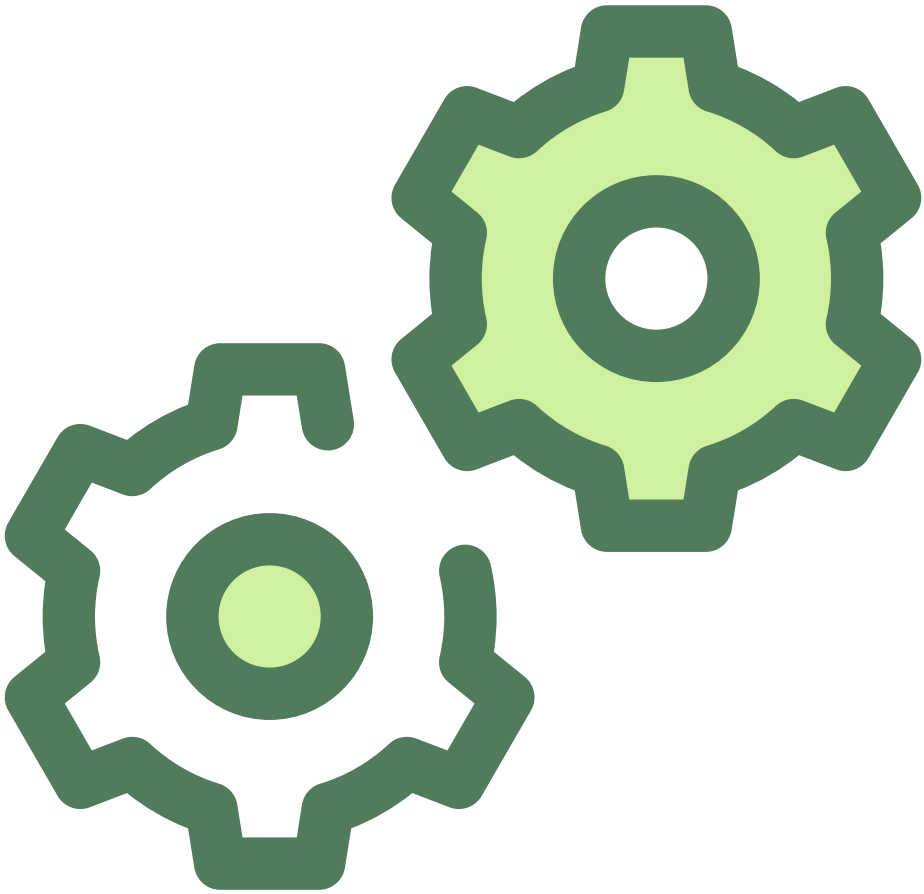
Andere bestanden

Projecten komen nog met andere *standaard* bestanden:

- install.py: help bij de installatie van het project op een server
- release.py: help bij het opzetten van allerlei registry waarden
- check.py: voert diverse controles uit die passen bij de installatie van het project
- local.py: wordt enkel uitgevoerd op werkstations

Later zullen we deze bestanden in meer detail bespreken.

Configuratie bestand



Elk `qtechng` project komt met een uniek configuratiebestand. Dit bestand bevindt zich steeds in de root-directory van het project en heeft een vaste naam `brocade.json`. (m.a.w. de `qtechng` path-naam van de configuratie file is steeds de `qtechng` path-naam van het project, aangevuld met `brocade.json`)

Dit bestand is een *JSON* bestand dat gestructureerd is volgens het schema qtechng.schema.json.

Dit heeft zo zijn voordelen: Visual Studio Code kan gebruik maken van deze specificatie om te helpen bij het editeren.

Er zijn nog andere voordelen maar daar vertellen we over als we het hebben over het gebruik van *Go* als implementatie taal.

Hou er mee rekening dat `brocade.json` het enige bestand is dat **NIET** wordt ingecheckt in het software repository als `qtechng` een fout merkt in het bestand.

`brocade.json` kan je opsplitsen in 2 delen:

- Richtlijnen voor installatie
- Specificatie en afhandeling van de bestanden in het project.

Voorbeeld

Het volgende voorbeeld toont een eenvoudig `brocade.json` bestand.

Let op het `$schema` attribuut: het vertelt de editor welk schema er van toepassing is. In Visual Studio Code is er meteen uitgebreide documentatie en ondersteuning van het editeer-proces.

```
{
    "$id": "https://dev.anet.be/brocade/schema/qtechng.schema.json",
    "$schema": "http://json-schema.org/draft-07/schema#",
    "core": true,
    "mumps": [
        "gtm"
    ],
    "py3": true,
    "binary": [
        "*.zip"
    ]
}
```

Richtlijnen voor installatie

De attributen “passive”, “mumps”, “groups”, “names”, “roles”, “versionlower”, “versionupper” helpen uit te maken of het project moet worden geïnstalleerd. Deze waarden worden vergeleken met registry waarden op de actuele server.

“py3”, “core”, “priority” hebben geen impact of het project moet worden geïnstalleerd, maar beïnvloeden wel het installatieproces.

Meteen een gouden regel voor de installatie van kind-projecten: een kind-project kan enkel worden geïnstalleerd als *ALL* ouder-projecten ook zijn geïnstalleerd.

Dit betekent nog dat “core” weinig zin heeft voor een kind-project. “priority” is relatief onder de kind-projecten van dezelfde ouder.

Specificatie en afhandeling van de bestanden

In `QtechNG` ken het *speciaal* statuut van bestanden worden weggenomen. Bij voorbeeld kan worden vastgelegd dat `helloworld.d` geen bestand is met macro’s (maar vb. een programma in `dlang`).

Zeer handig is ook de specificatie van wat *binary* is.

Hou er mee rekening dat deze specificaties steeds komen als *glob patterns* (met forward slashes) tegenover het relatieve path binnen het project.

Overzicht van de attributen

Let er op: alle attributen zijn in *lowercase*

binary

- `type`: *array*
- `description`: De bestanden uit het project waarbij de relatieve `qtechpath` overeenkomt met 1 van de elementen van de array, worden als `binary` bestand beschouwd. Er gebeurt geen r4/i4/m4/l4 substitutie. Deze waarde wordt *NIET* overgenomen in kind-projecten.
- `default`: []
- `uniqueItems`: True
- `items`:
  - type: string
  - description: Een wildcard voorstelling op de relatieve `qtechpath` naam van bestanden (dit is relatief ten opzichte van het project).

core

- `type`: *boolean*
- `description`: Indien `true`, dan wordt dit project geïnstalleerd samen met andere core projecten. Deze waarde specificeren binnen kind-projecten heeft geen effect. Indien `true`, dan zijn alle kind-projecten, core projecten.
- `default`: false

groups

- `type`: *array*
- `description`: Dit project wordt enkel geïnstalleerd als 1 van de array elementen overeenkomt met de registry waarde `system-group`. Bij niet-installatie, worden ook de kind-projecten *niet* geïnstalleerd.
- `default`: ["*"]
- `uniqueItems`: True
- `items`:
  - type: string
  - description: Een wildcard voorstelling voor de registry waarde `system-group`.

mumps

- `type`: *array*

- `description`: Dit project wordt enkel geïnstalleerd als de registry waarde `m-os-type` in `mumps` voorkomt. Bij niet-installatie, worden ook de kind-projecten *niet* geïnstalleerd.
- `default`: [“gtm”, “cache”, ””]
- `uniqueItems`: True
- `items`:
  - type: string
  - description: Identifier voor de M versie op het target systeem.
  - enum: [‘gtm’, ‘cache’, ”]

names

- `type`: *array*
- `description`: Dit project wordt enkel geïnstalleerd als 1 van de array elementen overeenkomt met de registry waarde `system-name`. Bij niet-installatie, worden ook de kind-projecten *niet* geïnstalleerd.
- `default`: [”*”]
- `uniqueItems`: True
- `items`:
  - type: string
  - description: Een wildcard voorstelling voor de registry waarde `system-name`.

notbrocade

- `type`: *array*
- `description`: De bestanden uit het project waarbij de relatieve `qtechpath` overeenkomt met 1 van de elementen van de array, worden niet als een `*.[bdilmx]` file beschouwd. Deze waarde wordt *NIET* overgenomen in de kind-projecten.
- `default`: []
- `uniqueItems`: True
- `items`:
  - type: string
  - description: Een wildcard voorstelling op de relatieve `qtechpath` naam van bestanden (dit is relatief ten opzichte van het project).

notconfig

- `type`: *array*
- `description`: De bestanden uit het project waarbij de relatieve `qtechpath` overeenkomt met 1 van de elementen van de array, worden niet als een Brocade configuratie file (brocade.json) beschouwd. Deze waarde wordt *NIET* overgenomen in de kind-projecten.
- `default`: []
- `uniqueItems`: True
- `items`:

  - type: string
  - description: Een wildcard voorstelling op de relatieve `qtechpath` naam van bestanden (dit is relatief ten opzichte van het project).

objectsnotreplaced

- `type`: *object*
- `description`: De objecten, gedefinieerd als sleutel (met prefixen i4, l4, m4, r4), worden *NIET* vervangen in de bestanden die. Deze waarde wordt *NIET* overgenomen in kind-projecten.
- `default`: {}

passive

- `type`: *boolean*
- `description`: De waarde `true` zorgt ervoor dat het project *niet* wordt geïnstalleerd. Bij niet-installatie, worden ook de kind-projecten *niet* geïnstalleerd.
- `default`: false

priority

- `type`: *integer*
- `description`: Geeft binnen de 2 groepen, core en niet-core, wat de volgorde is van installatie: hoge prioriteit wordt eerder geïnstalleerd. Kind-projecten worden altijd geïnstalleerd binnen hun ouder (in volgorde van prioriteit)
- `default`: 10000

py3

- `type`: *boolean*
- `description`: Indien `true`, dan worden install.py/check.py/local.py/release.py met `python3` uitgevoerd, zoniet met python2. Deze waarde wordt *NIET* overgenomen in kind-projecten.
- `default`: false

roles

- `type`: *array*
- `description`: Dit project wordt enkel geïnstalleerd als *alle* rollen uit de registry waarde `system-roles` overeenkomen met minstens 1 element van de array. Bij niet-installatie, worden ook de kind-projecten *niet* geïnstalleerd.
- `default`: ["*"]
- `uniqueItems`: True
- `items`:

  - type: string
  - description: Een wildcard voorstelling voor de registry waarde `system-roles`.

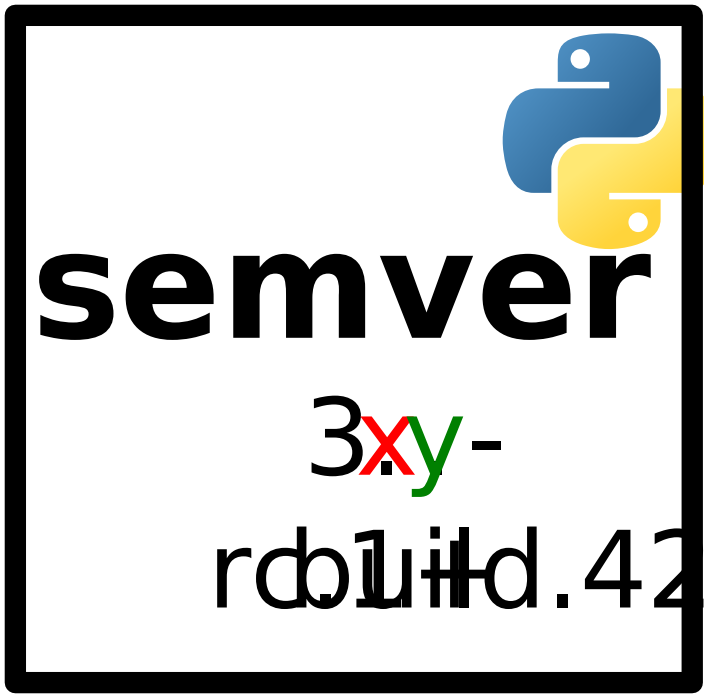
versionlower

- `type`: *string*
- `description`: Dit project wordt enkel geïnstalleerd als de registry waarde `brocade-release` lexicografisch kleiner is dan deze waarde. Bij niet-installatie, worden ook de kind-projecten *niet* geïnstalleerd.
- `default`: "~"

versionupper

- `type`: *string*
- `description`: Dit project wordt enkel geïnstalleerd als de registry waarde `brocade-release` lexicografisch groter is dan deze waarde. Bij niet-installatie, worden ook de kind-projecten *niet* geïnstalleerd.
- `default`: ""

Version



Voor software projecten die lang bestaan in de versie aanduiding belangrijk.

De communicatie rond de mogelijkheden van een software, worden gevoerd rond deze versie aanduiding. Release notes zijn bijvoorbeeld gecentreerd rondom een versie.

Niet zelden vinden versie aanduidingen hun weg naar contacten. Dat kan verschillende vormen aannemen: verplichtingen die gelden tot een versie (dit impliceert meteen dat er een zekere ordening bestaat in de versie aanduidingen), of het aantal releases die worden opgeleverd in een gegeven tijdspanne.

In Brocade hebben we sinds 2000 de gewoonte aangenomen om het versienummer te schrijven als `[0-9]+\.[0-9][0-9]`. De betekenis van deze nummer is vrij eenvoudig: de tweede numerieke component wordt bij elke release (ongeveer jaarlijks) met `10` verhoogd. Dit verhogen met `10` biedt de mogelijkheid om - voor administratieve reden - tussen releases te kunnen definieren.

De eerste component verhogen is eerder een kwestie van opportunisme: bijvoorbeeld we zijn overgeschakeld van `4` naar `5` om de breuk in een contractuele verplichting duidelijk in de verf te zetten.

Een tijd - ondertussen een paar jaar - geleden hadden Luc en ik een discussie over het gebruik van *Semantic Versioning".

Semantic versioning

De discussie die Luc en ik voerden, startte over de betekenis van een projectnummer van de gedaan `0.98.12`. Het gebruik van een software waarvan het versie nummer begint met `0.` is niet erg geruststellend: meestal duidt dit op een `beta` versie, nauwelijks beproefd in `the wild`. Maar eerlijk gezegd, dat slaat nergens op: de betekenis van die nummertjes ligt immers niet vast.

Is deze software een eindpunt op zich - bijvoorbeeld een tekstverwerker - dan is dat geen probleem: je gebruikt immers de software of je gebruikt hem niet en daar stopt het.

Wordt de software echter een bouwsteen van je eigen software (bijvoorbeeld: Lucene), dan wordt dit een ander paar mouwen. Er zijn immers dan 2 software projecten - het jouwe en Lucene - die onafhankelijk van elkaar evolueren. Als je Lucene wil gebruiken en er komt een nieuwe versie aan, dan wil je echt wel goed weten welke inpact dat deze software heeft op Brocade.

Maar, eerlijk gezegd, ook dat is best goed beheersbaar: Luc en Marc, kennen de twee software door en door, installeren *eventueel* en doen de nodige incompatibiliteitsaanpassingen.

Er is echter nog een ander probleem: ik noem dat graag het probleem van de kleine componenten.

Moderne software maakt graag gebruik van de inspanningen elders. Ik geef daar meteen een voorbeeld van in `qtechng`.

`qtechng` is een software project geschreven in Go

Op dit ogenblik zijn er reeds tal van *vreemde* projecten - buiten het package systeem van Golang zelf - die worden gebruikt:

- github.com/natefinch/atomis: Op atomische wijze wgeschrijven van data
- github.com/xanzy/ssh-agent: Het gebruik van ssh-agent in go
- github.com/atotto/clipboard: OS onafhankelijk omgaan van clipboard data in go
- github.com/spf13/afero: Virtuele filesystemen in go
- github.com/spf13/cobra: Toolcat-achtig framewerk in go

Deze *vreemde* projecten worden zelf ook gebouwd uit andere componenten, en dan weer …

Elk van deze deelcomponenten worden aangepast en hebben rechtstreeks invloed op de goede werking van je software.

Hier komt nog bij dat je software de deelcomponenten ook *automatisch* kan updaten (zie je het verschil met lucene ?)

Welkom in *dependency hell* !

Semantic Versioning is een initiatief uit de software industrie die tracht om precies dit probleem aan te pakken.

In dit nummerschema wordt betekenis gegeven aan de diverse componenten: `Major.Minor.Patch`

- Major: verandering van deze component houdt een hoog risico in. Functionaliteit verandert.
- Minor: verandering van deze component houdt een miniem risico in. Bijvoorbeeld de signatuur van de API krijgt een argument bij.
- Patch: verandering van deze component houdt een klein risico in. Meestal een eenvoudige bugfix.

Sommige ontwikkelsystemen (zoals vb. Go) gaan hier op uiterste gesofistikeerde manier mee om: het module systeem gebruikt die 3 componenten en gaat bijvoorbeeld wel automatische updates toelaten van `Patch`. De rest moet expliciet worden geconfigureerd.

Niet alle software projecten gaan hier even gezwind mee om of gaan hier licht van afwijken (vb. Python)

Ik denk dat Semantic Versioning zijn plaats heeft in Brocade: niet zozeer om de bestaande versie aanduiding te vervangen maar wel om deelcomponenten zoals `toolcat`, `clib` e.d. te specificeren.

Versie aanduidingen hebben zeker een rol te spelen in het verwante *versie controle*.

Maar dat is een onderwerp voor een ander blog post.

Registry

Brocade is verregaand geparametriseerd: meta-informatie bepaalt de specifieke functionaliteit en het uitzicht. Het minitieus parametriseren van een complex systeem kan enkel in een getrapt systeem en met verschillende technologieeen.

- Sommige meta-informatie wordt handmatig verzameld en zijn intrinsiek verbonden met de doelstellingen van een bibliotheeksysteem: catalografische beschrijvingen, gebruikersbeschrijvingen
- Om deze meta-informatie adequaat te beschrijven heb je instrumenten nodig. Er mee rekening houdend dat niet alle Brocade installaties volgens dezelfde procedures en richtlijnen werken, is er weer meta-informatie nodig om deze instrumenten scherp te stellen naar een specifieke situatie toe.
- Software en data moeten tenslotte worden geexploiteerd in services: software, data en services moeten concreet worden georganiseerd op servers. Om deze passend - denk: performantie, afscherming, systeemonafhankelijkheid - in te bedden op een actieve machine, moeten er keuzes worden gemaakt die te maken hebben met filesystemen, hardware, netwerking. Deze keuzes moeten op hun beurt worden vastgelegd in meta-informatie. Deze keuzes noemen we in Brocade de *registry*. De *registry* is een gestructureerd bestand ergens op de server.
- Diverse systeem componenten moeten de *registry* kunnen terugvinden. Daarom wordt de plaats waar de *registry* zich bevindt vastgelegd in een `environment variabele` die ter beschikking is van elk proces op de server (`BROCADE_REGISTRY`).

Bij het opzetten van deze environment variabele moet je ermee rekening houden

dat een Operating System op verschillende manieren deze variabele kan zichtbaar maken:

- in *M* sessies
- in *web* sessies
- in *SSH* sessies
- in *SHELL* sessies

Deze doorgedreven visie op meta-informatie - *een bibliotheeksysteem is het beschrijven van meta-informatie met een paar aanvullende services* - biedt zo zijn voordelen:

- een eenvoudige richtlijn om bibliotheek software te construeren: laat de software passen in dit getrapt model
- terugvindbaarheid van de diverse componenten door het getrapte systeem af te dalen of op te klimmen
- afstelbaarheid van het bibliotheeksysteem naar lokale noden
- onafhankelijkheid van de hardware
- onafhankelijkheid van het operating system

Structuur van de registry

De registry is opgeslagen in een file op het filesysteem van de server. De structuur is een eenvoudige key/value store, verwoord in een JSON object. De `key` is wat in de Brocade software wordt gebruikt en welke in runtime of bij installatie door de onderliggende software wordt omgezet in de `value`.

De plaats waar de registry wordt bewaard komt uit een environment variabele `BROCADE_REGISTRY`. De systemadministrator stelt deze environment variabele (en de bijhorende locatie) ter beschikking van elk Brocade process (interactieve shells, Web interface, SSH).

Beste praktijk (op een `UN*X` systeem):

```
export BASE=/library/database

export BROCADE_REGISTRY=$BASE/registry/registry.json
touch $BROCADE_REGISTRY

chmod --reference=$BASE $BASE/registry
chown --reference=$BASE $BASE/registry/registry.json
chmod u=rw,g=r,o= $BASE/registry/registry.json

touch $BASE/registry/localregistry.json
chown --reference=$BASE/registry/registry.json $BASE/registry/localregistry.json
chmod --reference=$BASE/registry/registry.json $BASE/registry/localregistry.json
```

```
touch $BASE/registry/saltregistry.json
chown --reference=$BASE/registry/registry.json $BASE/registry/saltregistry.json
chmod --reference=$BASE/registry/registry.json $BASE/registry/saltregistry.json
```

Er zijn verschillende manieren hoe en sleutel/waarde paar kan worden toegevoegd aan de `registry`:

- Door middel van de *ansible installatie*: de systemadministrator weet immers het beste waar bepaalde bestanden thuis horen.
- Manueel via de `delphi` applicatie
- Als onderdeel van `qtechng` via de `release.py` scripts in de projecten

Voor redenen van performantie en gebruikersgemak wordt deze `JSON` structuur ook omgezet naar een `M` structuur. Deze omzetting gaat automatisch.

De sleutels zijn strings en worden samengesteld volgens de reguliere uitdrukking :regexp:`[a-z][a-z0-9-]*[a-z0-9]`. In de sleutel wordt het koppelteken |MINUS|

Er zijn een aantal (niet-bindende) afspraken:

- de eerste component duidt het grote gebied aan waarvoor de sleutel wordt gebruikt: vb. `process-base-dir`
- een directory eindigt op `-dir`
- een executable eindigt op `-exe`
- een URL eindigt op `-url`
- bestandsnaam eindigt op `-file`
- registry sleutels die een *wachtwoord* bevatten, *moeten* eindigen op `-password`!
- de Brocade software moet zowel met `http` als met `https` kunnen worden gebruikt, afhankelijk van de keuze van de gebruiker. Gebruik daarom voor URLs steeds een absolute-path reference
- sommige waarden zijn commando's uit te voeren in de shell:
  - beperk dit zoveel mogelijk
  - laat `PATH` zijn werk doen
  - gebruik *nooit* wildcards
  - zorg ervoor dat deze commando's geconstrueerd zijn volgens de richtlijnen van de *Bourne shell* (POSIX)

Schema

zijn `registry.json` bestanden die voldoen aan het registry schema

Samenstellen van de registry

Sleutels in de registry zijn *niet* evenwaardig:

- sommige sleutels zijn *berekenbaar* en worden enkel opgenomen omdat het berekenen ervan te omslachtig is of te veel tijd zou nemen. Een voorbeeld hiervan is `os-sep` (separator van directories en bestandsnamen)
- sommigen zijn afhankelijk van andere sleutels: vb. `xml-catalog-dir` is afhankelijk van `web-base-dir`.

Een ontwikkelaar die een nieuwe registry waarde nodig heeft, volgt de volgende stappen:

- Is de nieuwe registry waarde afhankelijk van een reeds bestaande registry waarde, definieer dan in de projectfile `release.py` de nieuwe registry waarde
- Is de nieuwe registry waarde in te stellen door de system administrator, definieer dan de registry waarde in `/core/brocade/release.py`. De ontwikkelaar kan 2 strategieën volgen:
  - is er een goede default waarde, neem dan deze
  - kies anders een waarde die de installatie van het project doet falen

  De ontwikkelaar documenteert deze registry waarde in het project `/doc/registry`.

Gebruik van de registry

De registry wordt het best interactief bevraagd via het `delphi` commando.

Er zijn een paar details die belangrijk zijn om weten:

- bij de bevraging van de registry mag `-` in de sleutel ook worden vervangen door een `_`
- eindigt de sleutel op een `-` en begint de waarde met een `/`, dan wordt de inhoud van `registry(web-base-url)` geplaatst vóór de waarde in de sleutel (zonder eindigende `-`)
- het opvragen van een sleutel die eindigt op een `:` geeft de documentatie van deze sleutel *zonder* deze |:|
- in `M` gebruik je het beste de `m4_getDelphiValue` macro.
- in `python2` is er de constructie `from anet.core.registry import registry; value = registry(key, 'default')`
- in `python3` is er de constructie `from anet.core.base; value = base.registry(key, 'default')`

In tekstbestanden kan steeds de constructie `r4_key` worden gebruikt. *key* is een registry sleutel met `_` notatie. Het installatieproces gaat dan deze constructie vervangen door de waarde in de registry.

Deze techniek is soms de enige om aan parametrisering te doen. Toch is deze niet aan te raden: in situaties (|M|, |Py2|, |Py3|) waar zonder performantie verlies andere oplossingen bestaan, zijn deze te verkiezen.

Documentatie van de registry

De registry waarden worden gedocumenteerd in project `/doc/registry`.

Is `xyz` een registry sleutel, maak dan een bestand `xyz.rst` aan en noteer in dit bestand wat de registry waarde betekent.

Hanteer daarbij de volgende afspraken:

- gebruik `/doc/registry/os-sep.rst` als voorbeeld
- documentatie in `reST`
- in de documentatie heeft de effectieve waarde geen zin (het systeem gaat dat zelf wel tonen)
- vermeld zeker de intentie van de registry sleutel en bijzonderheden (restricties) op de waarde.

De *documentatie* van de registry waarden die niet langer worden gebruikt in de software, moeten handmatig worden geschrapt in het project `/doc/registry`.

Maak het onderscheid tussen registry waarden die niet gedefinieerd zijn maar die *toch* worden gebruikt in de software. De documentatie van deze registry waarden moet vanzelfsprekend blijven bestaan.

QtechNG

Er worden diverse registry waarden aangemaakt die bedoelt zijn om `QtechNg` goed te laten lopen. Deze sleutels hebben allen de prefix `qtechng-`.

Golang en QtechNG

Elke nieuwe versie van `qtechng` werd gemaakt met een nieuwe technologie: Awk, Makefiles, C, Perl en Python passeerden de revue.

De Python versie - zowel de commandline interface (CLI) als de GUI - deden het heel erg goed.

Er moet dan wel een heel goede reden zijn om dit succesverhaal te stoppen en te opteren voor een andere oplossing.

Mijns inziens zijn er diverse redenenen.

Opstarttijd

Een eerste reden ligt in de opstarttijd van de huidige `CLI`. Nu zal je zeggen, veel last heb ik daar nog niet van ondervonden!

Dat klopt, maar dat komt omdat de huidige Python toepassingen niet zo dikwijls worden opgestart: de GUI wordt meestal slechts 1x opgestart en maakt dan intern gebruikt van de beschikbare API's. De nieuwe - op Visual Studio Code gebaseerde - oplossing start wel degelijk de API zelf op en er is een merkbare vertraging. Daar komt bij dat de huidige `CLI` in `Python2` werd vervaardigd. `Python3` heeft een merkbare tragere opstart. Brocade's nieuwe toolcat framewerk gaat daar ook niet veel aan helpen!

Go code komt in één enkel executable en heeft een veel beter opstarttijd: er komt geen interpreter aan te pas en er hoeven ook niet meerder source bestanden worden opgestart (naargelang de setup van de Python installatie kunnen dat

10-tallen scripts zijn). Deze vraag naar betere opstart dan Python3 kan leveren, komt niet alleen uit `qtechng`: ook projecten als Mercurial en Firefox noteren het probleem.

Je zou kunnen argumenteren dat we de techniek van `wxqtech` kunnen imiteren en maar moeten gebruik maken van de `API`. Dat is toch heel wat moeilijker: bijvoorbeeld Visual Studio Code is zelf niet geprogrammeerd in `Python`. Bijgevolg moeten er tussenoplossing gezocht worden in een client/server model. Dit gaat de bedrijfszekerheid, laat staan de installeerbaarheid, niet ten goede komen.

Daarmee komen we aan een de volgende reden.

Installeerbaarheid

`qtechng` is een instrument dat niet alleen voor ontwikkelaars moet werken.

Om de een check-out met de huidige `qtech` te kunnen doen met eerste de juiste versie van `Python` worden geïnstalleerd. Vervolgens moet een SSH cliënt worden opgezet. Met `scp` (of een gelijkaardige software) moeten dan handmatig een 5-tal Brocade projecten worden getransfereerd vanop de goede plaats. Deze moeten dan zorgvuldig worden geïnstalleerd en, als de passende environment variabelen zijn gezet, kunnen we aan de slag.

Zelfs ontwikkelaars verliezen hierbij gemakkelijk hun weg.

Met `go` is het mogelijk om een executable in het `PATH` te plaatsen en men kan aan de slag.

De mogelijkheid om fout te gaan is gewoon vele malen kleiner: gedaan met reeds geïnstalleerde *systeem pythons*, oude versie van python, niet goed afgestelde systeem registry, dubbel python versies, afstellen van `PYTHONPATH` zodat zowel `python2` als `python3` kunnen werken.

Ingebouwde SSH

Er bestaan effectief SSH libraries in Python, deze zijn echter weinig efficiënt en werken moeilijk samen met een bestaande SSH infrastructuur (zoals aanwezige private/public keys en `ssh-agents`). In `qtech` zag ik me genoodzaakt om beroep te doen op de `ssh` en `scp` clients van het platform wat opnieuw vertragingen meebracht.

`Go` komt met een state-of-the-art implementatie van het `SSH` protocol: de communicatie gaat merkbaar sneller en bovendien kan deze perfect samenwerken met de gegeven SSH infrastructuur.

De implementatie speelt perfect in op het standaard reader/writer mechanisme van de programmeertaal.

CLI framewerk

Met Cobra beschikt `Go` over het perfect framewerk om POSIX-compliant toepassingen te maken met alles er op en er aan:

- Commando's en sub-commando's
- Interactieve design met Viper
- Automatische help en documentatie (o.a in `reStucturedText`)
- `Bash autocomplete` ondersteuning

Eens de `CLI` aangemaakt, is de executable onafhankelijk van Cobra of Viper

Snelheid van uitvoering

Een executable, gebaseerd op Go, werkt snel: daar is de compilatie verantwoordelijk voor.

Echter er zijn ook andere redenen: de ingebouwde Go libraries zijn exceptioneel. Google heeft kosten nog moeite gespaard om kwaliteit te leveren.

Toch ligt de voornaamste reden in de snelheidswinst in de mogelijkheden om, op gemakkelijke wijze, softwarecomponenten *concurrent* uit te voeren. Zeker in `I/O` gevoelige toepassing als `qtechng` kan hier grote winsten worden geboekt: eerste testen tonen 20-voudige snelheidswinst aan ten opzichte van de `Python` versie.

Deze snelheidswinst is niet alleen van belang bij interactieve operaties maar is ook van groot belang bij operaties zoals Continuous Integration, zeg maar de `sweep`.

Ook het installatie proces van een nieuwe Brocade release zou op deze wijze drastisch kunnen worden herleid. Dit is zeker van belang als we van plan zijn meerdere, tussentijdse, releases te gaan voeren.

Descriptieve software componenten

Deze snelheidswinst hoeft echter niet helemaal te gaan naar snelheid van uitvoering: een gedeelte kan ook besteed worden om bepaalde componenten van `qtechng` eerder descriptief aan te pakken dan manueel - en dus (meestal) sneller - te implementeren.

Ik geef twee voorbeelden om dit in het licht te zetten.

De configuratie bestanden `brocade.json` is duidelijk veel complexer (en ook veel krachtiger) dan in vorige versie. Door de structuur rigoureus te beschrijven via een `JSON schema` kan validatie en implementatie software ook automatisch worden gegenereerd. Go leent zich heel goed tot dergelijke operaties.

Een ander voorbeeld is bijvoorbeeld de macro structuur. Macro's zijn heel erg belangrijk voor `qtechng`: het zijn de API's waarrond alles draait. In `qtechng` winnen macro's in belangrijke mate aan kracht. In plaats van een handmatige

parser te schrijven kan nu de kracht van een PEG-parser worden aangewend. Ja, deze parsers geven minder efficiënte code, maar ze geven meteen ook een rigoureuze definitie van het macro systeem. Iets waar Brocade al langer nood aan heeft.

Cobra

Als je een goede CLI wil bouwen, moet je op zoek naar een goed framewerk. Voor onze in-house `Python` code hebben we `Toolcat` maar voor een software

gebaseerd op `Go` hadden we nog niet iets dergelijks.

Een paar jaar geleden volgen Luc en ik een tutorial over Go: het was een duizelingwekkende ervaring aan sneltrein-tempo gegeven door Steve Francia. Deze man werkt voor Google en was rechtstreeks betrokken bij het Go-project. Zijn taak was daarbij niet zozeer het implementeren van Go. Neen, hij moest zorgen voor een aantal instrumenten zodanig dat ontwikkelaars binnen Google snel aan de slag konden met Go.

Hij vervaardigde verschillende outstanding libraries en softwares, allen gebaseerd op Go.

Zo maakte hij Hugo, een generator voor statische websites. Deze blog is bijvoorbeeld gemaakt met Hugo. Hugo ontpopte zich ondertussen als één van de meest populaire web builders.

Zijn meest populaire product is echter het duo Cobra/Viper. Cobra is een framewerk om CLI's te maken. Viper is dan weer een interactieve tool die hierbij kan helpen.

Cobra is simpelweg het meest gebruikte CLI-framewerk in de Go-wereld: Hugo en Kubernetes zijn maar een paar producten die hiermee gebouwd zijn.

De redenen van het succes van Cobra zijn velerlei:

- in de eerste plaats is er de kwaliteit van de code: Cobra is simpelweg een tutorial voor Go
- Cobra is platform onafhankelijk
- De software is zeer flexibel en opgewassen tegen alle taken die je er kunt tegen aan gooien
- De software gaat verder dan mogelijk maken van commando's: help en documentatie worden automatische gegeneerd en in allerlei formaten
- Aangemaakte software is POSIX-compliant

Cobra is zeker een goede kandidaat om er `qtechng` mee te vervaardigen.

Ik heb ondertussen verschillende CLI's gemaakt met `Cobra`. Een goed voorbeeld is `Qq`. `Qq` is een voorloper van `qtechng`.

```
/home/rphilips/Desktop$ Qq
Qq maintains the Brocade software:

    - Development
    - Installation
    - Deployment
    - Version Control
        - Management

Usage:
  Qq [command]
```

```
Available Commands:
  about       Useful information about `Qq`
  help        Help about any command
  project     Project functions
  system      System information
  version     Version functions

Flags:
      --cwd string         working directory
      --directory string   qpath of a directory under a project
      --editor string      editor name
      --force              with force
  -h, --help               help for Qq
      --info               lists arguments, flags and global values
      --jq string          jq command
      --lower              transforms to lowercase
      --pattern string     Posix glob pattern
      --project string     project to work with
      --recurse            recursively walks through directory and subdirectories
      --regexp             searches as a regular expression
      --remote             execute on the remote server
      --tree               handles filenames as rpaths in the project
      --uid string         user id
      --unhex .            unhexify the arguments starting with .
      --verbose            verbose output (default true)
      --version string     version to work with

Use "Qq [command] --help" for more information about a command.
```

Een paar voorbeelden om te werken met de CLI:

```
Qq project create /take/application --version=5.10
Qq project list /take/*
```

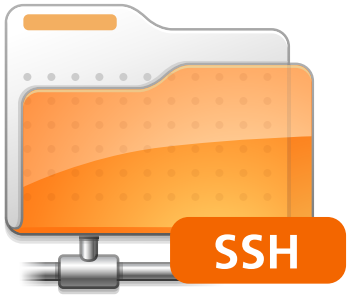
Het programmeren zelf is descriptief. Om bijvoorbeeld `Qq project create` te maken is het voldoende om in de code de volgende beschrijving te plaatsen:

```
var projectListCmd = &cobra.Command{
    Use:     "list",
    Short:   "List projects",
    Long:    `Command lists all project matching a given pattern`,
    Args:    cobra.MaximumNArgs(1),
    Example: "Qq project list /stdlib/t*",
    RunE:    projectList,
    PreRun:  func(cmd *cobra.Command, args []string) { preSSH(cmd) },
    Annotations: map[string]string{
        "remote-allowed":    "yes",
```

```
        "always-remote-onW": "yes",
        "fill-version":      "yes",
        "with-QtechType":    "BW",
    },
}
```

De `PreRun` en de `Annotations` zijn nog wat verwarrend maar daarover later meer!

SSH



Waarom gebruik je SSH in `qtechng`?

Wel, toen we in 1997 begonnen aan de bouw van Brocade, was dit niet zo!

We hadden daarvoor 2 redenen: een goede en een slechte.

Laten we beginnen met de slechte reden. Zoals steeds, als ik gegrepen ben door een idee, dan heb ik de neiging om dat idee tot in het extreme te volgen. In 1997, 1998 was ik er van overtuigd dat web technologie de wereld van informatiebeheerders ging bepalen. Ik wou daarom alle onze instrumenten web gebaseerd maken. Ik zag duidelijk de voordelen: geen gesukkel meer met firewalls, het vergaren van diepe kennis omtrent web technologie en ja, ook het marketing aspect daarvan (Brocade: het Web en niet sdan het Web).

Ik experimenteerde met shells in de webbrowser (en geprogrammeerd in Java!). Voor beveiligde communicatie was er dan TLS en zijn voorganger SSL. En dat allemaal in Netscape Communicator 4.71.

Het is echter een veeg teken als je meer met de instrumenten bezig bent dan met datgene wat die instrumenten moeten bouwen. Gelukkig was er Wim Holemans die me wees op `SSH`. Dat viel samen met het moment waarop een kwalitatief hoogstaande cliënt voor Windows beschikbaar kwam: PuTTY.

In de jaren daarna heb ik steeds gezocht naar een SSH library voor Python. Deze bestaat wel degelijk. Conch is een onderdeel van Twisted en dat is precies één van de problemen. Twisted is een omvangrijke platform dat bovendien niet echt aansloot bij de standaard manier van ontwikkelen voor Python. Conch was té moeilijk te integreren.

Met `Go` is de situatie helemaal anders: de SSH library wordt onderhouden door het Go-team (hoewel het nog geen echt onderdeel is van de standaard packages: er zijn nog geregeld kleine aanpassingen aan de functionaliteit). Dat is begrijpelijk: SSH is cruciaal van belang voor Docker en Kubernetes.

In `qtech` wordt de taak van SSH uitbesteed aan CLI's zoals `pscp.exe`. `plink.exe`, `ssh`, `scp`. Per `qtech` invocatie worden er telkens 2 externe

executables opgestart wat de performantie zeker niet ten goede komt. De verschillen tussen OpenSSH en `PuTTY` veroorzaken heel wat configuratieproblemen en communicatie tussen de verschillende onderdelen gebeurt dan over het filesysteem met als gevolg bestanden die moeten worden opgeruimd en meer van dat fraais.

In `qtech` (en ook in `qtechng` zijn er essentieel 3 communicatievormen die plaatsvinden over `SSH`:

- Een opdracht, geformuleerd op de cliënt (werkstation of productiemachine), wordt ingepakt in een Python pickle, getransfereerd naar de ontwikkelmachine. Daar wordt het antwoord ook ingepakt en teruggestuurd naar de cliënt. Een typische voorbeeld is een zoekactie naar files: `qtech -list edit '*.m' find='m4_getCatIsbdTitles'`. Deze kan typisch worden uitgevoerd door een `SSH` instructie.
- Deze opdracht heeft betrekking op bestanden op de lokale computer. Deze moeten samen met de opdracht verstuurd worden naar de ontwikkelmachine en daar worden verwerkt. Dit kan heel wat moeilijker worden ingebed in een `SSH` instructie. In `qtech` worden de bestanden ingepakt in een ZIP-bestand en getransfereerd. Daarna wordt de opdracht verstuurd, het ZIP-bestand wordt uitgepakt en de opdracht wordt uitgevoerd. Een typsich voorbeeld is de checkin van bestanden: `qtech -ci *.*`
- Een opdracht heeft betrekking op bestanden reeds aanwezig op de ontwikkelmachine en vraagt om deze bestanden naar de lokale machine over te hevelen. Een goed voorbeeld is: `qtech -sync`.

In `qtech` en `qtechng` is er sprake van 2 machines: 1 daarvan is steeds de ontwikkelmachine `dev.anet.be`, de andere is ofwel een workstation, ofwel een productiemachine.

Het is belangrijk om zich te realiseren dat de ontwikkelmachine, `dev.anet.be`, ook moet fungeren als werkstation én productiemachine. Als we de bovenstaande `qtech` operaties uitvoeren, dan mag het communicatie gedeelte geen overhead genereren. Ideaal voor het ontwikkelen van `qtechng` zou zijn dat gans dit proces wordt weggemoffeld.

De `Go` library `golang.org/x/crypto/ssh`, samen met Cobra, maakt dit mogelijk.

Een ander aspect aan het communicatiegebeuren is natuurlijk de serializering van de diverse concpeten: bestanden, opdrachten.

Er zijn 2 wegen die we kunnen opgaan:

- Gestandaardiseerde serializering: de data wordt ingepakt in bijvoorbeeld JSON of Protocol Buffers.
- Een specifieke oplossing wordt uitgewerkt.

Ik heb eerst de eerste benadering uitgewerkt. De overhead voor binaire bestanden maakte `JSON` geen goede kandidaat. `JSON` wordt in `qtechng` wel degelijk veelvuldig gebruikt maar dan als datastructuur voor opslag op schijf. Hoewel `Protocol Buffers` een excellente oplossing is, heb ik het toch niet weerhouden: immers in het communicatieproces zijn de beiden einden in dezelfde taal (`Go`). Hadden we 2 verschillende technologien gehad dan waren `Protocol Buffers` de juiste weg.

Uiteindelijk heb ik geopteerd voor `encoding/gob`, het elegante en performante serializeringsprotocol ingebouwd in `Go`.

Ik geef nu een voorbeeld van hoe gebruik te maken van `Cobra` en `SSH in Go`:

```
package cmd

import (
    "encoding/hex"
    "os"

    qresult "brocade.be/qtechng/result"
    "github.com/spf13/cobra"
)

var aboutCmd = &cobra.Command{
    Use: "about",
    Short: "Useful information about `Qq`",
    Long: `Version and builttime information about Qq`,
    Args: cobra.ArbitraryArgs,
    Example: "Qq about",
    RunE: about,
    PreRun: func(cmd *cobra.Command, args []string) { preSSH(cmd) },
    Annotations: map[string]string{
       "remote-allowed": "yes",
    },
}

func init() {
    rootCmd.AddCommand(aboutCmd)
}

func about(cmd *cobra.Command, args []string) error {
    msg := map[string]string{"!BuildTime": BuildTime, "!BuildHost": BuildHost, "!BuildWith"
    host, e := os.Hostname()
    if e == nil {
        msg["!!uname"] = host
    }
    if len(args) != 0 {
```

```
    for _, arg := range args {
        msg["hexified "+arg] = hex.EncodeToString([]byte(arg))
    }
    }
    Fmsg = qresult.ShowResult(qresult.RMAnswer(msg))
    return nil
}
```

Je kan de code op 2 manieren activeren: (`Qq` is de voorlopige naam voor `qtechng`)

```
Qq about
```

en:

```
Qq about --remote
```

In het eerste geval vertelt de software je iets over de versie van `Qq` op je werkstation:

```
(base) ~/tmp$ Qq about
{
    "host": "rphilips-home",
    "time": "2020-05-26T10:05:48+02:00",
    "RESULT": {
        "!!uname": "rphilips-home",
        "!BuildHost": "rphilips-home",
        "!BuildTime": "2020.05.24-11:30:43",
        "!BuildWith": "go1.14.3"
    }
}
(base) ~/tmp$
```

In het tweede geval krijg je informatie over de ontwikkelmachine:

```
(base) ~/tmp$ Qq about --remote
{
    "host": "rphilips-VirtualBox",
    "time": "2020-05-26T10:07:13+02:00",
    "RESULT": {
        "!!uname": "rphilips-VirtualBox",
        "!BuildHost": "rphilips-home",
        "!BuildTime": "2020.05.15-15:36:53",
        "!BuildWith": "go1.14.2"
    }
}
(base) ~/tmp$
```

Bij de ontwikkeling van `func about` is het communicatiegedeelte (over SSH) onzichtbaar: je ontwikkelt voor het gemakkelijkste geval waarbij er gewoonweg

geen communicatie is!

Vbox

Een virtuele machine of Docker ?

`qtechng` werkt essentieel steeds met 2 servers waarbij de ene machine de ontwikkelmachine (dev.anet.be: qtechtype == 'B') is en de andere ofwel je werkstation (qtechtype == 'W') ofwel een productiemachine (qtechtype == 'P').

Dat betekent dat je voor de ontwikkeling van `qtechng` ook moet kunnen beschikken over 2 machines. De aangewezen oplossing hiervoor is om op je werkstation ofwel een *virtual machine* ofwel een *docker container* op te zetten.

Ik koos ervoor om te gaan voor de `virtual machine`. Deze keuze is niet zo vanzelfsprekend: Docker is immers *gemaakt* om dit soort setups te tackelen. Ik ben echter meer vertrouwd met *virtual machines*. Bovendien wil ik zo dicht mogelijk aanleunen bij de situaties waarin `qtechng` worden ontplooid.

Er zijn echter ook nog wat andere redenen: ik wil graag eens een aantal Linux versies uitproberen waarover ik veel goeds heb gehoord maar er zelf nooit heb mee gewerkt: Manjaro, Ubuntu Budgie en UbuntuDDE.

Één van de mooie dingen van een VM is dat je ook kunt experimenteren met het aantal CPU's (cores) die je toewijst aan een VM.

Het opzetten van een VM aan de hand van Virtual Box is niet bepaald uitdagend. Toch een wijze raad: moderne Linux distributies komen met een software store. Het voorkomt heel wat problemen als je *Virtual Box* installeert vanuit deze software store: nieuwere versies vertonen immers vaak compatibiliteitsproblemen.

Vanzelfsprekend mag je niet nalaten de `Guest Additions` te installeren: deze maken het werken met muis, clipboard en het filesysteem een stuk gemakkelijker.

Voor wat netwerking betreft, kies ik voor de eenvoudigste en tegelijkertijd ook veiligste oplossing: `NAT` met Port-forwarding voor `SSH`.

- Mijn host (dagdagelijkse machine) is Linux Mint
- De `guest` die ik installeer is `Ubuntu Budgie` (20.04)
- Met Linux `Guest Additions`
- Networking: NAT met Port Forwarding: SSH: IP 127.0.0.1 op poort 2222 naar poort 22 op de guest
- In /etc/hosts op de host plaats ik een entry: `127.0.0.1 localhost budgie`. Op deze wijze kan ik mijn `guest` steeds adresseren als `budgie`.

Op `budgie` installeer ik de environment variabele `BROCADE_REGISTRY` in `etc/environment`:

```
BROCADE_REGISTRY=/home/rphilips/brocade/registry.json
```

De relevante inhoud van de registry:

```
{
    "qtechng-test": "test-entry",
    "qtechng-version": "9.92",
    "qtechng-hg-backup": "ssh://root@backup.anet.be//qtech/hg/{version}",
    "qtechng-type": "B",
    "qtechng-repository-dir": "/home/rphilips/qtechng",
    "qtechng-hg-enable": "1",
    "qtechng-max-openfiles": "64"
}
```

Aangezien `budgie` de rol gaat spelen van de ontwikkelserver moet hier een `SSH` service worden op gedefinieerd.

`Ubuntu` is een SystemD distributie en bijgevolg is het opzetten en starten van de `SSH` service vrij standaard:

```
sudo apt-get install openssh-server  # installeren van de software uit de softare store
sudo systemctl enable ssh # enable de service
sudo systemctl start ssh # opstarten van de start
sudo ufw allow ssh # firewall configuratie
sudo ufw enable
sudo systemctl status ssh
```

Vanzelfsprekend moet je nu een account aanmaken op `budgie`.

Vervolgens moet je je SSH gegevens kopieren naar `budie`. Is je host een UNIX-achtige machine, dan kan je dit meer dan waarschijnlijk eenvoudig doen door middel van de instructie:

```
ssh-copy-id -i ~/.ssh/id_rsa.pub rphilips@budgie -p 2222 # Voer uit van op je eigen machine
```

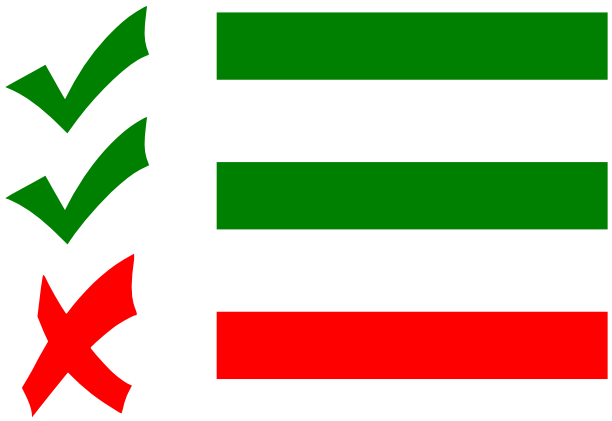
Op de host (je eigen werkstation) zien de relevante registry waarden er als volgt uit:

```
{
    "qtechng-releases": "4.20; 4.30; 4.40; 5.00; 5.10",
    "qtechng-test": "test-entry",
    "qtechng-diff": "[\"meld\", \"$target\", \"$source\"]",
    "qtechng-version": "9.92",
    "qtechng-uid": "rphilips",
    "qtechng-editor-exe": "code",
    "qtechng-server": "budgie:2222",
    "qtechng-type": "W",
    "qtechng-workstation-basedir": "/home/rphilips/projects/brocade/edit",
    "qtechng-max-openfiles": "64"
}
```

Let op de adressering van de server!

Zo, we hebben nu twee systemen klaar voor actie!

Test



Dit is een bijdrage van Bart: interpreteer deze tekst als een overzicht van de problematiek. `testen` is een munt met verschillende kanten. Later worden deze van nabij besproken.

Inleiding

In deze blog gaan we het hebben over het belang van softwaretesten op verschillende niveaus. Ik denk dat niemand overtuigd moeten worden van het belang maar het is wel belangrijk om de niveaus van softwaretesten te onderscheiden en zo ook communicatie er over te verbeteren.

Deze blog wil dan ook de discussie starten om in de toekomst verder te gaan met software testen binnen Anet met een duidelijk jargon.

Belang van software testing

Goede softwaretesten kunnen zorgen voor

- klantentevredenheid
- documentatie
- performantie
- clean code

En dat is natuurlijk belangrijk. Laat me deze punten verder uitlichten.

Klantentevredenheid

Softwaretesten kunnen fouten/defecten van een softwaresysteem in een vroeg stadium identificeren, wat op zijn beurt de kwaliteit/stabiliteit van een product verbetert en vertrouwen in het product opbouwt. Met "vroege fase" bedoel ik, waar het nog steeds haalbaar en effectief is om de bestaande gebreken makkelijk te verwijderen en dus zeker niet na een release.

1 negatieve ervaring met software kan 100 goede ervaringen teniet doen.

Documentatie

Iets dat vaak over het hoofd gezien is dat softwaretesten ook een goede (aanvullende) documentatie van de code zijn. Een goede softwaretest geeft meteen ook duidelijk aan wat verwacht wordt van een stuk software om te doen. Zowel in verwachte als onverwachte situaties.

In ideale omstandigheden zou een bug moeten leiden tot een extra software test of gewoon een extra case voor een bestaande software test.

De meest extreme variant om softwaretesten te gebruiken als documentatie is *Test Driven Development (TDD)*. Hierbij is het idee dat voordat je software schrijft je eerst alle mogelijke test-scenarios schrijft die initieel allemaal moeten falen aangezien code ontbreekt en gaande weg de ontwikkeling wel allemaal zullen slagen.

Performantie

Softwaretesten kan je keer op keer automatisch laten runnen. En zo kan je ook makkelijk monitoren of het herschrijven van een stuk code de performantie ten goede komt. Of zien wel stuk code wel heel veel geraakt wordt en een goede kandidaat is om performanter te maken en/of nog beter te testen om het zo kritisch is.

Clean code

Software testen kunnen op meerdere manieren leiden tot *clean code*. 1 aspect hierbij is het gebruik van *code coverage*. Met code coverage kan je zien welke lijnen code geraakt worden door testen. En zo kan je opmerken dat het onmogelijk is bepaalde lijn code te raken. Neem bijvoorbeeld volgende snippet

```
function not_smart():
    if a:
        ...
        a = None
    ...
    if a:
        print("...")
```

Als tussen beide if-testen niks gebeurd met `a` zal je de 2de if-test nooit waar zijn. Nu dit voorbeeld is heel duidelijk maar wat als er 100 regels code staan tussen deze 2 if-testen en je net een software aanpassing wil doen/bug oplossen in de buurt van deze 2de if-test. Je kan je te pleuris zoeken achter nut van 2de if-test. Maar als je goede softwaretesten hebt met coverage had je deze 2de if-test al lang verwijderd.

Soorten testen

Je kan testen op meerdere niveaus indelen en in literatuur vind je hier ook heel wat tegenspraken/spraakverwarring. Hier een aanzet om de testen te omschrijven binnen Brocade om zo tot uniform taalgebruik te komen

Ik zie binnen brocade volgende testen

- operational acceptances testing (OAT)
- unit testen
- systeem testen
- integratie testen
- code testen

En het is niet altijd eenvoudig om een test in één van deze hokjes te delen. Maar je hebt wel testen nodig op verschillende niveaus. Het internet staat vol met memes en grappige filmpjes waarom er nood is aan verschillende soorten testen. Zoals Unit versus integration tests door lock of Unit versus integration tests umbrella

Laat in volgende secties dieper ingaan op (praktische) verschillen. En het onderstaande is geen wet, maar voer tot discussie!

OAT

Operational acceptances testing (OAT) is de verzameling van testen om te kijken of alles aanwezig en werkend is om de software te laten draaien. Ik ben zelf geen specialist in Ansible, maar mijn inziens moet Ansible hier in een groot deel voorzien. Natuurlijk werkt Brocade niet als bijvoorbeeld `m4_strUpper` niet werkt. Het feit dat deze niet werkt kan veel oorzaken hebben, Mumps, C-library.

Zoals ik het begrijp, is `Check.py` oorspronkelijk bedoeld om OAT-testen te doen. Maar nu bevat deze zowel OAT als unit testen.

Mijn voorstel is `Check.py` geleidelijk aan te verwijderen en te vervangen door `CheckOAT.py` en `CheckUnit.py` met dus respectievelijk OAT en unit testen.

Nu voor we dat kunnen moeten we wel overeenkomen om duidelijk de twee werelden te scheiden.

Mijn inziens is het alvast niet aan OAT-testen om resultaten te controleren. Dus een OAT-test kan zijn `m4_strUpper(RDstr, "ibm")` maar of `RDstr="IBM` valt

buiten de scope van OAT-testen. Pas als OAT-testen heeft het zin om verdere software testen te doen.

Een ander voorbeeld ik weet dat de slack-tool een specifieke versie van van slackclient nodig heeft, dus mijn inziens is een goede OAT-test

```
from packaging import version
from slack import WebClient

assert version.parse("2.5.0") <= version.parse(slack.version.__version__)  <= version.parse(
```

Deze gaat crashen als slack ontbreekt als package en fout geven als versie niet in orde is.

Om terug te komen op macro's, we zouden bij bepaalde essentiële macro's een speciaal `example` kunnen zetten die OAT-testen kunnen oppikken met een example dat moet worden. Nogmaals, niet om juistheid macro te testen, maar wel dat macro oproepbaar is. Denk hierbij aan essentiële macro's zoals die waar C-code achterzit om onderscheid met GT.M en Cache.

Unit testen

Unit testen, zoals de duidelijk in de naam, is het testen van 1 unit, functie, … Naar mijn ervaring helpt het schrijven van unit testen ook in het opsplitsen van functies. Het is vaak eenvoudiger en leesbaarder om een hulpfunctie apart te testen dan als deze hulpcode in de hoofdfunctie zit.

Zoals eerder vermeld zou ik deze unit testen onderbrengen in `CheckUnit.py`. Voor Python is een handige tool voor unit-testen PyTest, als kan PyTest ook dienen voor OAT, Systeem, en Integratie-testen. Meer info over PyTest vind je in een rst over PyTest

Belangrijk bij een unit testen is zoveel mogelijk de functionaliteit te isoleren. Stel dat jouw functie een andere functie oproept die heel veel logica doet, dan mock je idealiter deze functie uit. En injecteer dus de mogelijk resultaten van deze functie. Anders ben je eigenlijk met een systeem test bezig. Nu, dit is de theorie en bijvoorbeeld binnen Python is dit haalbaar, maar niet binnen Mumps.

Systeem/ integratie testen

Van het moment dat je meer dan 1 unit tegelijk test spreekt men strikt genomen over systeem-test. Ik zelf merk de meeste verwarring over de naam "Systeem"-test, want soms wordt deze ook gebruikt voor OAT.

De meerwaarde van een systeem test zit hem vooral als je een groot systeem test. Dus de volledige flow van een call. Bijvoorbeeld het resultaat testen van `https://dev.anet.be/oai/abua/server.phtml?`.

Het grote verschil met integratie-testen is dat je binnen je systeem blijft. Bijvoorbeeld als je een centaur-import wil testen is dat een systeem test als je het resultaat van de externe service mocked. Als je effectief data binnenhaalt van een externe server of data naar een externe server stuurt, spreken we van integratietesten.

Integratietesten hebben vooral meerwaarde t.o.v. systeemtesten om te kijken of de afgesproken API nog werkt en dat er niet opeens meer data komt dan verwacht of onder ander formaat.

Wat men in praktijk vaak doet, is een vlag zetten of een test tegen live-server spreekt of gebruik zal maken van mock-servers.

De meerwaarde van systeem/integratie testen t.o.v. unit testen komt aan bod onder "Coverage".

Nu binnen brocade denk ik dat we op meerdere levels systeem testen kunnen maken.

- louter mumps
- louter python en mumps data mocken

En als integratie-testen beschouwen de hele flows. Je zou binnen Brocade Mumps als externe service kunnen zien. Het is wel belangrijk dat we effectief import testen, maar bijvoorbeeld het offline zijn van een externe server moet in de "sweep", of binnen welk proces dit ook terecht komt, een andere prioriteit krijgen dan het mislukken van oai-export of falen van leen.

Dit is zeker nog stof tot discussie hoe we deze theorie rond best-practises in praktijk brengen.

Praktisch kunnen voor deze testen zowel klassieke testen (???t*.m, PyTest, …) gebruikt worden en kan ook Selenium een goede kandidaat zijn.

Code testen

Naast functionaliteit testen kun je ook testen op kwaliteit van broncode.

Ten eerste kun je een Linter gebruiken. We zijn allemaal vertrouwd met de Mumps parser en bewust van de meerwaarde om fouten te voorkomen. Flake8 is een tegenhanger voor Python.

Wat betreft Flake8 moet zeker nog eens de discussie gevoerd worden wat we als standaard nemen. Momenteel worden nog bepaalde errors genegeerd zoals "E722 do not use bare 'except' " en kunnen we strikter gaan door te testen op relatieve imports, *-import, gebruik van enkele versus dubbele quote.

Ik zou daar ver in durven gaan maar belangrijk is vooral dat we hiervoor tot gezamenlijk akkoord komen.

Naast linting zijn er nog mogelijkheden om (test) code kwaliteit te verbeteren die ik hieronder beschrijf.

Coverage

Nog een handige tool om code te verbeteren is test-coverage. Test coverage bepaalt in hoeverre het runnen van de testen, code raakt. Typisch wordt dit uitgedrukt in een percentage. Dus een test-coverage van 80% wil zeggen dat als je alle testen runt, je 80% van de code hebt aangeraakt en dus gebruikt.

Daarnaast kun je ook zien welke code wel en niet wordt aangeraakt. In extremis kun je zelfs zien welke code het meest wordt aangeraakt, maar aangezien de coverage bepaald wordt door het aantal testen moet je dit dan met korrel zout nemen. Het interessantste is om te gaan kijken welke code niet wordt geraakt want dit kan twee dingen betekenen:

- deze code is niet getest
- je kunt deze code niet testen want onbereikbare code zie `function not_smart()` in het begin van de blog.

Coverage berekenen voor Python kun je makkelijk bekomen m.b.v. PyTest en werkt zelfs al, bijvoorbeeld `scrutiny pytest coverage=yes`. Voor Mumps heeft Bashkar een voorzet gegeven om dit te implementeren en dat maakt onderdeel uit van BR510-128

Nu hoe ver gaan in test coverage? 100% test-coverage kan zeker geen kwaad, maar vergt wel een hele inspanning. Als je op het net zoekt naar blogs enzovoort over coverage is dit vaak punt van discussie.

Ik denk dat wat betreft core-software, 100% test-coverage een must is. En voor de rest moeten we streven naar 100%. Ik denk alvast dat als we eenmaal de tools hebben voor test-coverage we moeten streven elke release in test-coverage te stijgen.

Waarbij we onderscheid moeten maken tussen coverage van unit en systeem/integratie testen. Je kunt met unit-testen normaal makkelijker 100% coverage bekomen. Maar de kans is dat je in een unit-test code kan testen die in de praktijk niet kan aangeraakt worden. Dit komt dan naar boven met systeem testen. Maar dat wil wel zeggen dat je je testen slim moet opdelen tussen systeem en unit-testen.

Een systeem-test zou geen code mogen aanraken die in de praktijk nooit aangeraakt wordt.

Wat dan met bijvoorbeeld macro's die future proof gemaakt worden en momenteel nog niet ten volle functionaliteit gebruikt worden? Ten eerste, maak macro's niet te complex dus test coverage kan je daar bij helpen. En coverage wijst je dan op feit dat code misschien best herzien wordt.

Nu, binnen brocade zitten we ook met routines die in meta zitten en die geven de grootste blind-spot. Specifiek binnen Mumps als je een routine die niet met `%` begint, niet kan coveren met een systeem test is dit een goede kandidaat om van dichterbij te bekijken. Maar met `%` is het complexer. Dus ik denk dat we ook

hier nog werk is om een overzicht te bekomen van routines in meta. Misschien dat we i.p.v. UTF8-scanner voor globals een routine scanner moeten maken.

Mutatietesten

Een iets meer omstreden techniek die toch zijn meerwaarde bewezen heeft is *mutatietesten*. Hierbij stel je eigenlijk je softwaretesten op de proef. Het idee is dat een software tool op doordachte wijze kopieën maakt van de originele code en daar wijzigingen aan doet om vervolgens de software testen opnieuw te runnen. Als software testen dan slagen, is er een probleem …

Voorbeeld:

```
fn %dubbel(PDa):
. n a,b
. s a=2
. s a=PDa*2
. q a

sub %TstDubbel:
. w:0'=$$%vb(0) !,"FAIL!"
. w:4'=$$%vb(2) !,"FAIL!"
. q
```

`%TstDubbel` zorgt ervoor dat `%dubbel` 100% test coverage heeft. Wat dus wil zeggen elke lijn code van `%dubbel` wordt aangeraakt als de test loopt. Het is op het zicht duidelijk (omdat het zo een klein voorbeeld is) dat de lijn `s a=2` onnuttig is. Iets wat een mutatietest bijvoorbeeld gaat doen is `s a=2` vervangen door `s a=3`. Als dan testen nog slagen is er iets mis. In dit geval onnodige code.

Maar mutatietesten gaan ook typisch operaties wijzigen en dus `s a=PDa*2` vervangen door `s a=PDa**2` en in dit geval gaan testen nog altijd slagen. Dus in dit geval mist er een test-case zoals `w:10'=$$%dubbel(5) !,"FAIL!"`

Ansible Automation Platform en Qtech

Deze blog-post van Luc gaat over het beheer van Ansible met *Qtech(ng)*.

In 2016 hebben wij de afweging gemaakt tussen *Salt* en *Ansible* voor IT-automatisering. Salt leek op dat moment de beste keuze. De software had een lage leercurve en enorm veel integraties. Ansible daarentegen was toen de “new kid on the block” maar wel met veel potentieel !

Intussen heeft Ansible onder de vleugels van Red Hat een inhaalbeweging gemaakt en is vooral dankzij support van Red Hat **het** automatiseringsplatform geworden.

Wat heeft dan voor ons de doorslag gegeven om over te stappen ?

Ansible heeft een aantal eigenschappen die het mogelijk maken om het beheer van *Ansible draaiboeken* en *Ansible roles* onder Qtech te plaatsen. Dit was met Salt niet of nauwelijks realiseerbaar of toch niet zonder in te grijpen in de manier waarop Qtech werkt.

Nu, vooraleer we dieper ingaan in de ontplooing van een Ansible folderstructuur op onze servers met Qtech, toch eerst even een woordje uitleg over het gebruik

van Ansible zelf en wat onze doelstellingen zijn.

- De Ansible Software wordt op al onze servers geinstalleerd.
- De installatie van de Ansible folderstructuur gebeurt door Qtech.
- Op elk van onze servers wordt een Ansible folderstructuur geplaatst. Dit zijn de *Ansible playbooks* en de *Ansible roles* afgestemd voor die specifieke server.
- De uiteindelijke installatie van software en services op onze servers start met het Ansible commando:

```
ansible-playbook playbook.yml
```

- De opdrachten in een Ansible draaiboek zijn idempotent en kunnen meermaals uitgevoerd worden zonder neveneffecten.

De Ansible folderstructuur op een server ziet er als volgt uit:

```
* playbook.yml
* playbook-gtm.yml
* playbook-apache.yml
* ...
* files/
  * gtm.tar.gz
  * apache.tar.gz
  * ...
* group_vars
  * all/
    * role-common.yml
    * role-make.yml
    * role-gtm.yml
    * role-apache.yml
    * ...
* roles/
  * gtm/
    * handlers/
      * main.yml
      * ...
    * meta/
      * main.yml
      * ...
    * tasks/
      * main.yml
      * ...
    * templates/
      * execcache.j2
      * ...
  * apache/
```

```
  * htmldoc/
  * ...
```

Deze Ansible folderstructuur is vrij eenvoudig aan te maken met Qtech indien een aantal afspraken rond *Qtech bestandsnamen* en *Qtech folders* worden gevolgd.

De Qtech repository voor Ansible ziet er als volgt uit:

```
* /ansible/
  * roles/
    * gtm/
      * install.py
      * task-main.yml
      * meta.yml
      * handler-main.yml
      * plugins.xc.j2
      * ...
    * apache/
    * ...
  * servers/
    * presto/
      * install.py
      * playbook.yml
      * vars-common.yml
      * vars-apache.yml
      * vars-gtm.yml
      * ...
    * moto/
    * dolce/
    * ...
```

- *Roles* worden beheerd in Qtech projecten `'/ansible/roles/[role name]'`
- Voor elke server bestaat een Qtech project `'/ansible/servers/[server name]'`
- Het Qtech bestand `'/ansible/servers/[server name]/playbook.yml'` wordt vertaald naar een server *Ansible draaiboek*
- Qtech bestanden `'/ansible/servers/[server name]/vars-*.yml'` worden vertaald naar *Ansible variabelen* in `'group_vars/all/'`
- Qtech bestanden `'/ansible/roles/[role name]/task-*.yml'` worden vertaald naar *Ansible taken*
- Het Qtech bestand `'/ansible/roles/[role name]/meta.yml'` wordt vertaald naar *Ansible meta data*
- Qtech bestanden `'/ansible/roles/[role name]/handler-*.yml'` worden vertaald naar *Ansible handlers*
- Qtech bestanden `'/ansible/roles/[role name]/*.j2'` worden vertaald naar *Ansible templates*

- ...

Uiteindelijk is het `'/core/python3/ansible.py'` die de vertaalslag doet van Qtech bestanden naar een Ansible folderstructuur.

bv. `'/ansible/servers/presto/install.py'`

```
# -*- coding: utf-8 -*-
# /ansible/servers/presto/install.py

from anet.core import ansible
ansible.install_playground(host='presto')
```

bv. `'/ansible/roles/gtm/install.py'`

```
# -*- coding: utf-8 -*-
# /ansible/roles/gtm/install.py

from anet.core import ansible
ansible.install_role('gtm')
```

Bij het inchecken van een Ansible Qtech project worden ook de bestanden uit de Ansible repository (een remote site bepaald door een registry-waarde) gekopieerd naar de Ansible folderstructuur. Dit gaat voornamelijk over bestanden die niet thuishoren in Qtech zoals tarbals, zip, e.a. Op die manier is alles aan boord en is in principe geen netwerktoegang vereist tijdens de installatie en configuratie van de server.

Je voelt het "kip-en-het-ei-verhaal" al aankomen ? Hoe kan je nu een Ansible folderstructuur opbouwen op een server waarop nog geen Qtech aanwezig is ? bv. een installatie op een nieuwe server.

Wel, op de ontwikkelserver (de server met system-role **'dev'**) wordt bij het inchecken van een Ansible Qtech project, voor **elke** server een Ansible folderstructuur opgebouwd. De nieuwe server kan dan geinstalleerd en geconfigureerd worden ofwel vanaf de ontwikkelserver ofwel na kopiëren van de Ansible folderstructuur naar de nieuwe server. De keuze is aan ons.

Repository

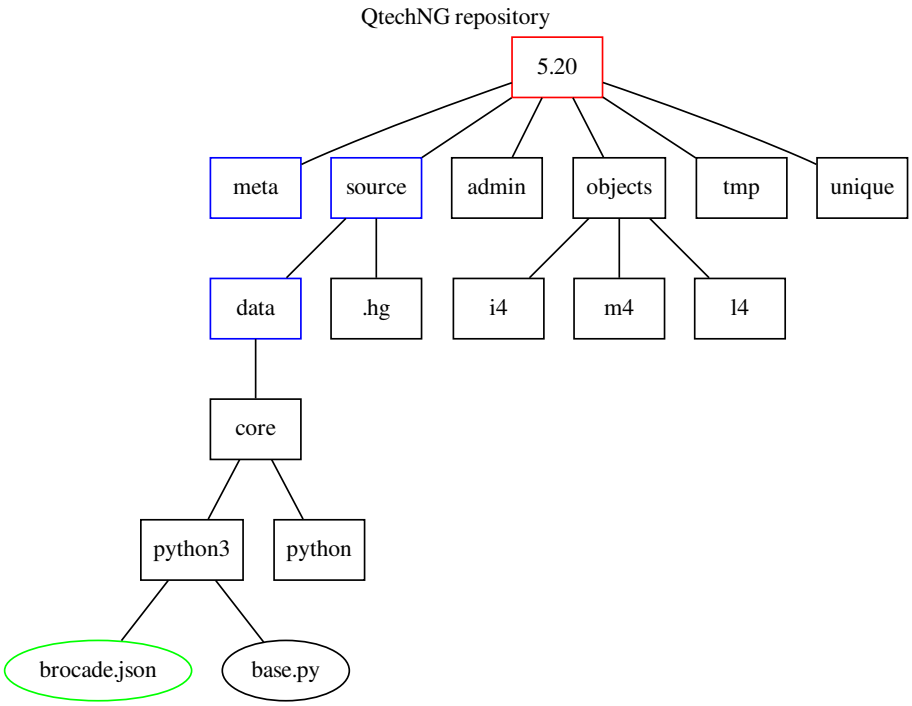


Figure 1: 5.20 repository

`QtechNG` werkt met een software repository. De basis directory waarop dit repository rust, komt - u raadt het al - uit de registry: `qtechng-repository-dir`: vb. `/library/repository`

Deze directory - beschikbaar op alle productieservers en ontwikkelservers - wordt door `root:db` aangemaakt door Ansible met het commando:

```
export REPO=`delphi key qtechng-repository-dir`
sudo mkdir -p $REPO
sudo chown root $REPO
sudo chgrp db $REPO
sudo chmod 2770 $REPO
```

Dit maakt dat het `setgid` bit van de directory wordt gezet: alle bestanden en directories gaan to groep `db` behoren.

In deze directory worden dan - per release - subfolders aangemaakt die dan de naam van de release dragen.

De release directory heeft een aantal subfolders. In latere posts in deze blog gaan we deze wel nader bekijken.

Nu wil ik vooral beklemtonen dat deze structuur werkt als een eenvoudige databank. Ik wou geen gebruik maken van een bestaand databank product: ofwel zijn deze van de gedaante one-writer/multiple readers (zoals SQLite) ofwel maken ze gebruik van een client-server opstelling.

Ik wou de opzet primitief houden: geen extra onderhoudswerken, geen extra installatieproblemen, geen locks die op de meest vervelende momenten gelockt blijven, maar wel een robuuste opstelling die in alle omstandigheden kan werken. Dit systeem, gebaseerd op eigenschappen van bestanden en directories, heeft ook het voordeel dat het zonder licentie of prijsoverwegingen op andere servers kan worden geplaatst. Hoewel Brocade zelf niet opensource is (in de betekenis dat de software gratis is), wordt de source code op elk productiesysteem in alle openheid geïnstalleerd en bijgewerkt.

Het is niet alleen `QtechNG` die deze benadering volgt: nagenoeg alle versie controle systemen (Git, Mercurial) ondersteunen deze aanpak. Het `QtechNG` repository leent overigens nogal wat uit de trukendoos van dergelijke systemen.

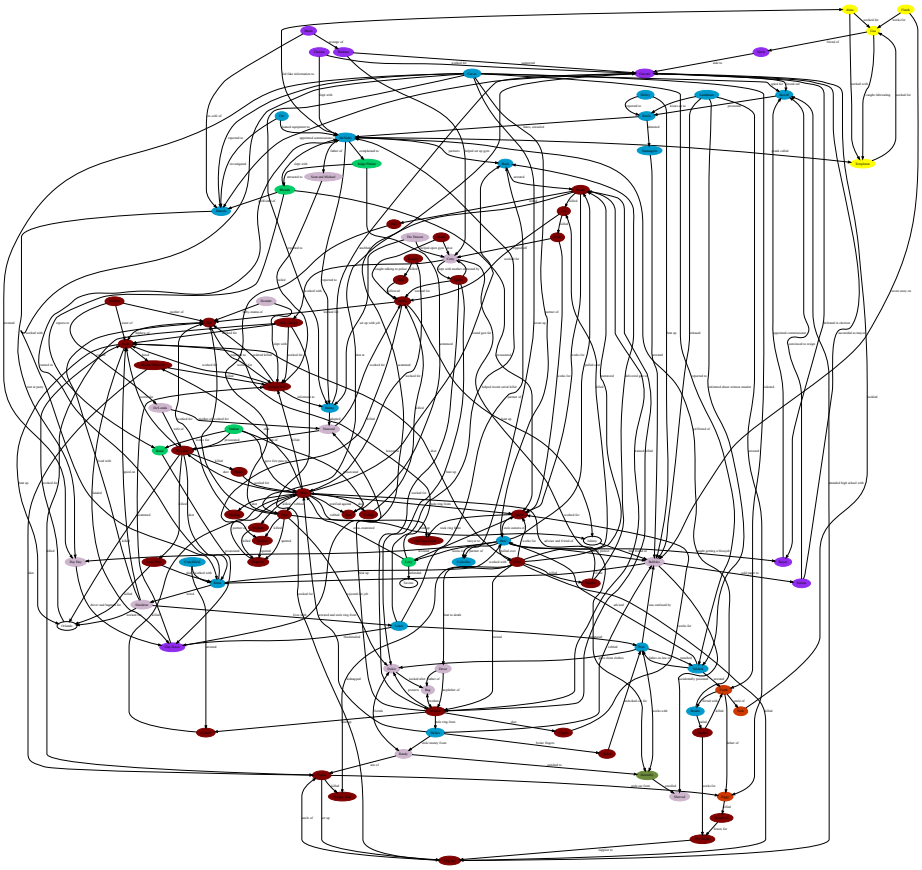
De directories met de belangrijke data zijn:

- meta: elementaire meta data omtrent de ontwikkelbestanden
- source: ontwikkelbestanden gegroepeerd in projecten

Uit deze 2, kan de rest van de data worden hersteld.

In volgende blogposts gaan we het belang en de structuur van de verschillende directories uit de doeken doen.

Interludium: Graphiz & Dot



In de vorige blog post toonde ik een SVG.

Ik was eerst van plan deze met Inkscape aan te maken maar de goesting ging snel over. `SVG` is een `XML` toepassing en dat met de hand schrijven leek me nog minder appetijtelijk.

Ik herinnerde me dat het in Sphinx, ons documentatie platform, mogelijk is om grafen in te bedden en besloot om de technologie daarachter, eens van nabij te bekijken.

`Sphinx` maakt gebruik van Graphviz. In `Graphviz` wordt een graf rigoureus beschreven en dan met behulp van gespecialiseerde software, omgevormd naar diverse formaten: PDF, PNG, … en SVG.

De specificatie van een `graf` gebeurt met DOT. Dit is een gespecialiseerde taal waarmee men de knopen en takken en hun samengang, zorgvuldig kan beschrijven.

Ik geef meteen het voorbeeld wat ik in de vorige blog post heb gemaakt.

```
cat repo.dot # Toon de inhoud van de graf specificatie in dot

digraph "QtechNG repository" {
      /* paper size in inches */
      size="11.0,8.5";
      /* locate label at top of drawing */
      labelloc=t;
      label="QtechNG repository";
      /* no directional arrow on connectors */
      edge [dir=none];
      /* nodes below are boxes (folders) */
      node [shape=box;color=red];
      version [label="5.20"];
      node [shape=box;color=blue];
      meta [label="meta"];
      source [label="source"];
      data [label="data"];
      node [shape=box;color=black];
      admin [label="admin"];
      objects [label="objects"];
      i4 [label="i4"];
      m4 [label="m4"];
      l4 [label="l4"];
      hg [label=".hg"];
      python3 [label="python3"];
      python [label="python"];
      core [label="core"];
      tmp [label="tmp"];
      unique [label="unique"];
      /* nodes below are ellipses (files) */
      node [shape=ellipse;color=green];
      bjson [label="brocade.json"];
      node [shape=ellipse;color=black];
      basepy [label="base.py"];
      /* parent -> child, to draw the tree */
      version -> admin;
      version -> meta;
      version -> objects;
      version -> source;
      version -> tmp;
      version -> unique;
      objects -> i4;
      objects -> l4;
```

```
        objects -> m4;
        source -> data;
        source -> hg;
        data -> core;
        core -> python3;
        core -> python;
        python3 -> bjson;
        python3 -> basepy;
}

dot -Tsvg repo.dot -o repo.svg # Maak een SVG aan
```

`repo.svg` ziet er uit als volgt:

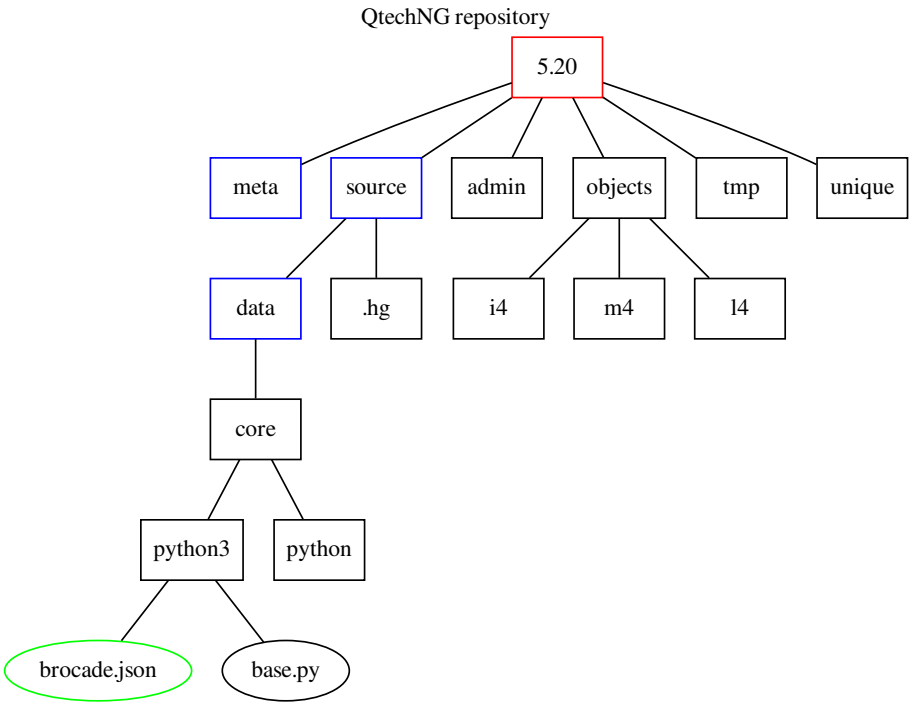


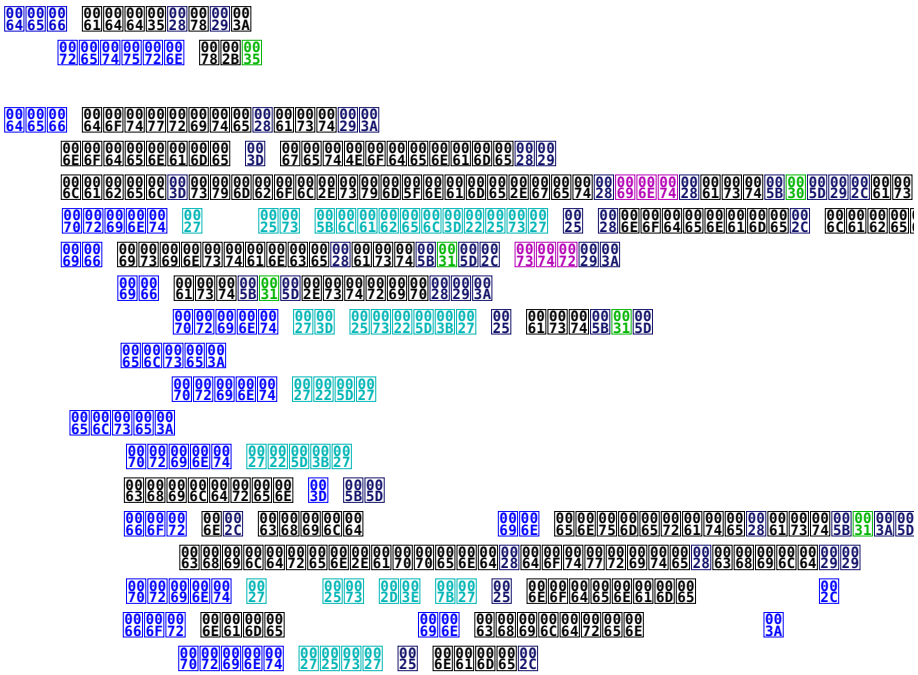
Figure 2: 5.20 repository

`SVG` kan een belangrijke plaats innemen in Brocade. Vooral Alain is de kampioen van deze technologie: het gebruik van `SVG` in bijvoorbeeld statistieken, opent immers heel wat mogelijkheden.

Het is te omslachtig om `SVG` met behulp van `Python` te genereren. Er bestaat echter een library voor Graphviz in Python.

Aanmaken van SVG's - via DOT - wordt hiermee een koud kunstje.

Source



Een bestand wegschrijven in het `QtechNG` repository ?

Dat kan toch niet zo moeilijk zijn!

Neen, het is inderdaad niet zo moeilijk maar je moet wel heel erg minutieus te werk gaan!

Ik wil beklemtonen dat deze bijdrage enkel gaat over gewone bestanden: met andere woorden het gaat niet over *object definiërende* bestanden (dat zijn de bestanden van de gedaante `*.[ild]`): we gaan die afzonderlijk bespreken.

Ook het installeren in de productieomgeving wordt niet behandeld: dus we gaan niet uitleggen hoe een bestand van de gedaante `*.m` in `M` terechtkomt.

Er zijn diverse aspecten die we willen bespreken.

Waar precies wordt een bestand weggeschreven ?

Dit hangt af van verschillende parameters:

- in de eerste plaats de registry waarde in de sleutel `qtechng-repository-dir`
- vervolgens de release (deze moet steeds - impliciet of expliciet - worden opgegeven)
- het `qtechng-path` van de source file: dit is een systeem onafhankelijke voorstelling

Om het met `python` termen te zeggen:

```
release = "5.20"
repository = registry("qtechng-repository-dir")
qtechngpath = "/take/application/takwone.m"

filename = os.path.join(repository, release, "source", "data", *qtechngpath.split("/")[1:])
```

Wat die "data" component daar komt bij doen, houden we voor later!

Bepalen van het project waartoe een bestand behoort

Eens dat de filenaam is bepaald, kunnen we gaan zoeken naar het project waartoe de file behoord.

Te beginnen vanaf `os.path.join(repository, release, "source", "data")`, schuimen we alle directories af, op zoek naar naar de `brocade.json` files.

De laatste `brocade.json` file die we vinden, zegt ons meteen wat het `project` is waartoe ons bestand behoord.

Je vraagt je wellicht af waarom het niet beter is van *omgekeerd* te werk te gaan: eerst in dezelfde directory kijken waar onze bestand staat en dan afdalen, richting `os.path.join(repository, release, "source", "data")`. Dat lijkt zeker efficiënter.

Het is echter zo dat brocade configuratie files (`brocade.json` bestanden) met behulp van de sleutel `notconfig`, sommige `brocade.json` bestanden kunnen merken als *niet-configuratie bestand*!

Het kunnen bepalen tot welk project een bestand behoort is echt wel belangrijk: een bestand zonder project wordt simpelweg niet opgeslagen in het repository!

Wat met `brocade.json` ?

Bestanden met een basename gelijk aan `brocade.json`, moeten speciaal worden behandeld. Als `QtechNG` uitvist dat het om een waarachtige configuratiefile gaat, dan wordt zorgvuldig gecontroleerd of die voldoet aan het JSON schema en of er geen overtollige - en mogelijke schadelijke - data in staat. Pas als aan alle testen voldaan zijn, kan het bestand worden geplaatst.

De reden voor deze voorzichtigheid is eenvoudig: eens een foutieve configuratiefile in het systeem terecht komt, kan die niet meer met de `QtechNG` software worden verwijderd en moet deze manueel worden aangepast.

Unieke bestanden

Sommige bestanden moeten een unieke basename hebben!

Denk maar aan de M routines (en hun schermfiles `*.x`). De precieze structuur van die unieke bestanden wordt gespecificeerd door `registry("qtechng-unique-ext")`.

Om de uniciteit van deze bestanden te bewaken, maakt `QtechNG` in `os.path.join(repository, release, "unique")` een structuur aan voor *alle* bestanden. (De registry waarde kan immers veranderen en dan moeten we *klaar* staan voor die verandering).

In `unique` wordt er voor ons bestand een path gedefinieerd als volgt:

```
basename = os.path.basename(filename)
bdigest = SHA512(basename)
fdigest = SHA512(filename)

uniquename = os.path.join(repository, release, "unique", bdigest[:2], bdigest[2:], fdigest)

# in uniquename wordt `{"path": qtechng-path van bestand} weggeschreven
```

Op deze wijze kan snel worden uitgevist, of er al een bestand bestaat met dezelfde basename.

Nog even meegeven dat de constructie van uniqename schaamteloos geleend is uit andere technologieën: zowel `git` als `hg` gebruiken gelijkaardige constructies. Zo is de splitsing van `bdigest` ingegeven om de directories niet over te bevolken!

Bestanden die zondigen tegen de uniciteits-regel worden niet weggeschreven in het repository: ook deze fouten kunnen immers niet altijd door `QtechNG` worden rechtgezet.

Het uniciteitsverhaal is hiermee nog niet ten einde: in de configuratie kan immers met de sleutel `notunique` uitzonderingen op de regel worden gemaakt. Dit is belangrijk om bijvoorbeeld M routines toe te laten die verschillen op `GT.M` en `Cache`. Een uitzondering kan gewettigd zijn als we er maar voor zorgen dat ze niet beiden worden geinstalleerd.

Meta informatie

Per source file wordt een beperkte set aan meta data bijgehouden. Deze komt terecht in `os.path.join(repository, release, "unique")`. Deze meta informatie is een JSON structuur die tijdstippen van aanmaak en laatste update bijhouden, samen met de userid van de verantwoordelijken.

De plaats waar deze meta informatie terecht komt, is in grote mate gelijk aan de constructie van de unieke namen:

```
fdigest = SHA512(filename)

metaname = os.path.join(repository, release, "meta", fdigest[:2], digest[2:]+".json")
```

Test op token

Als het bestand wordt aangeboden om te worden weggeschreven, moet dit vergezeld zijn van een `token`. Dit token moet bewijzen dat je begonnen bent met dit bestand te downloaden uit het repository en dat je daar dan veranderingen hebt aan aangebracht.

Dit `token` reist steeds mee met dit bestand. Dit `token` (een SHA512 digest) wordt gecontroleerd tegenover het reeds aanwezige bestand. Klopt het `token` niet, dan wordt de opslag geweigerd en moet de gebruiker het bestand opnieuw downloaden (samen met het juiste token), zijn aanpassingen aanbrengen en terug aanbieden.

Overzicht van de situaties waarbij een bestand wordt geweigerd

Even een opsomming van alle situaties waarbij een bestand wordt geweigerd:

- `source.new.noproject`: geen project gevonden dat past bij het bestand
- `source.store.config.invalid`: de file (`brocade.json`) is geen geldig configuratiebestand
- `source.store.notunique`: het bestand beantwoordt niet aan de uniciteitsregels
- `source.store.forbidden`: kan het bestand niet wegschrijven (vermoedelijk door token probleem)

Dit *weigeren* resulteert in een aangepaste foutboodschap. Er kunnen nog heel wat andere foutboodschappen komen - zoals het bestand bevat een M parse fout - maar dan wordt het wel opgenomen in het repository!

Atomair wegschrijven van de bestanden

Het definitief wegschrijven van de bestanden is een belangrijk zaak: Alain zou zeggen: “Dit moet in steen worden gebeiteld”.

Alle voorzorgen moeten worden genomen opdat het wegschrijven compleet gebeurd: het niet wegschrijven van een bestand is erg, maar het corrupt of partieel wegschrijven is nog veel erger.

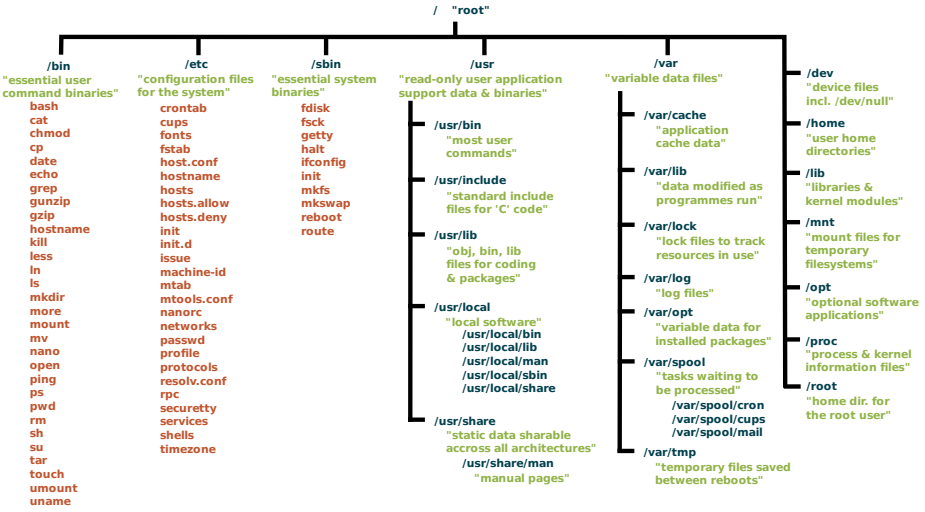
Gelukkig bestaan er systeem onafhankelijke, atomaire operaties die dit mogelijk maken: eerst wordt het bestand opgeslagen onder een andere naam in dezelfde directory en tenslotte wordt er een `rename` uitgevoerd (onder windows moet er eerst wel een delete van het originele bestand gebeuren).

Eind goed, al goed ...

Dat dacht je maar.

In volgende blog posts gaan we het nog hebben over *virtuele filesystemen* en *parallelle* acties.

Virtuele filesystemen



Met deze blogbijdrage wil ik een lans breken voor een techniek die niet alleen een groot gebruikersgemak oplevert voor de ontwikkelaar maar ook een paar (dramatische) fouten kan voorkomen!

We gaan het hebben over Virtuele Filesystemen.

Een `VFS` vertrekt steeds van een reeds bestaand filesysteem: dat kan zijn het goed gekende filesysteem zoals het op je notebook staat, maar dat kan ook een Dropbox folder zijn, een simpele ZIP-file, of iets dat op AWS staat of op Google Drive.

Wij, Brocade mensen, kennen maar al te goed WebDAV: door middel van een instrument zoals Webdrive wordt een webserver aangeboden net alsof het een lokale schijf is.

Kenmerkend is dat er een soort bemiddelbaar bestaat die het `VFS` op een meer vertrouwde manier aanbiedt.

Deze bemiddelaar kan echter ook een bibliotheek zijn die door de ontwikkelaar kan worden aangewend om bestanden te lezen, te schrijven, te wissen, te doorzoeken, …

Een goed voorbeeld van een `VFS` uit de `Python` wereld, is PyFilesystem.

Ook in `Go` zijn er talrijke `VFS` te vinden en dat hoeft je niet te verwonderen: de designers van deze programmeertaal hebben het I/O systeem - cruciaal voor filesystemen - op een waarlijk voortreffelijke manier uitgebouwd. `VFS`-en in `Go` kunnen op een zeer natuurlijke manier worden uitgebouwd.

Zo'n `VFS` heeft diverse voordelen: met een minimum aan inspanningen kan de code operationeel worden gemaakt op Amazon Web Services of simpelweg in

RAM. Dit laatste is bijvoorbeeld zeer handig als je de `QtechNG` software wil testen: voor je repository switch je van harde schijf naar RAM en, na afloop van je testen, is alles mooi opgeruimd!

In `QtechNG` gebruik ik diverse `VFS`.

Laat ik eerst de problemen (of: gevaren) schetsen die ik tegenkom met de *gewone* filesystemen.

Het is duidelijk dat het `QtechNG` repository tot diepe folderstructuren leidt. Dit zal trouwens nog erger worden als we ook de opslag van objecten (samen met hun gebruik in bestanden) gaan behandelen.

Dit heeft tot gevolg dat de aanmaak en afbouw van folders en subfolders heel zorgvuldig moet gebeuren. Ik wou daarom dat de klassieke `mkdir` in `QtechNG` dezelfde functionaliteit kreeg als `mkdir -p`: alle, nog niet gedefinieerde, tussenliggende folders worden automatisch aangemaakt.

Echter, vergeet niet dat het repository als een databank moest fungeren: ik wou dus ook dat, als folders leeg zijn, deze ook automatisch worden opgeruimd!

Kijk, met klassieke file operaties is dat om moeilijkheden vragen:

- je vergeet al eens een folder te schrappen (als hij verder geen bestanden bevat)
- maar nog veel erger, het is echt niet uit de lucht gegrepen dat het schrappen te ver gaat en folders worden opgeruimd waar dat beter niet zou bij gebeuren.

De oplossing ligt in het organiseren van een `VFS`. Zoals dat in Brocade context meestal het geval is, stopt het niet bij het in gebruik nemen van een software. Neen, er wordt een heuse fabriek opgezet om dergelijke `VFS` te genereren.

De basis ligt in Afero, een software van de makers van *Cobra*.

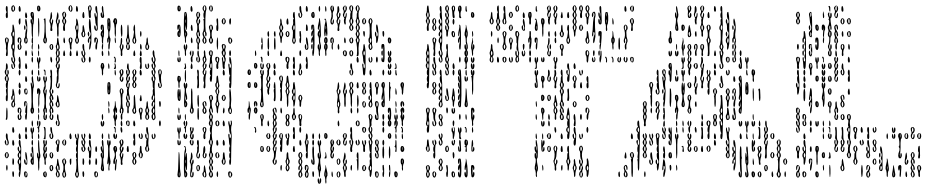
Hier heb ik een aantal methodes aangepast:

- Bij het schrappen van een bestand wordt er gekeken of er nog bestanden in de parentfolder staan en is dat niet zo, *en het gaat hier niet om de rootfolder*, dan wordt de folder zelf meedogenloos geschrapt
- Alle schrijfacties zijn atomair
- Bestanden behoren steeds tot de groep `db` en ze kunnen enkel gelezen en beschreven worden door leden van die groep
- Aanmaak van een directory maakt ook de tussenliggende directories aan
- Filesystemen kunnen `readonly` zijn (zeer nuttig bij zoekacties)
- De separator is steeds `/`

Er worden (op dynamische wijze) filesystemen aangemaakt voor de verschillende Brocade versies, voor elk project, voor de objectbomen, enz. De `root` van elk dergelijk filesysteem is zodanig dat je daar niet kunt buiten opereren.

Het mag duidelijk zijn dat ik echt opgetogen ben over deze werkwijze, een werkwijze die we trouwens nog op heel wat andere plaatsen kunnen aanwenden in Brocade.

Tekst versus binair



Wat is een tekstbestand ?

Een eenvoudige definitie is: een tekstbestand bestaat uitsluitend uit (Unicode) karakters. Het vervelende hierbij is dat je, behalve het tekstbestand en zijn inhoud, ook de encoding moet opgeven: als je `CP-1252` gebruikt, is elk bestand tekst.

Om het onderscheid te maken in `qtech`, volgde ik een pragmatische benadering.

Ik werkte in 3 stappen:

- Had het bestand een filenaam een extensie en behoorde die extensie tot een bepaalde verzameling (vb. `{".js", ".m", ".d", ".html", ".zwr"})` dan concludeerde ik: dit is een `tekstbestand`.
- Had het bestand een filenaam een extensie en behoorde die extensie tot een andere verzameling (vb. `{".pdf", ".zip", ".png", ".xls", ".odt"})` dan verklaarde ik het bestand `binair`
- Had het bestand geen extensie, of lag de extensie buiten deze beide verzamelingen, of … werd de data aangeleverd zonder filenaam, dan volgde ik een algoritme dat ik in de code van de Perl programmeertaal vond. (`Perl` had een operator - `-T` als ik het goed heb - waarmee je kan testen of een bepaald bestand binair was). Dit algoritme nam de eerste 4K bytes en deed daar allerlei analyses op en besloot daaruit met `tekst` of `binair`.

Waarom is dat belangrijk ?

Dit is belangrijk omdat toepassingen nu eenmaal anders omgaan met tekstbestanden en binaire bestanden: open je een binair bestand met je - van Emacs verschillende - editor, dan is de kans groot dat je editor crasht.

Gebruik je grep om bestanden te doorzoeken naar tekst, dan gaat `grep` enkel de tekstbestanden beschouwen. Daar kan je je aardig aan verbranden zoals ik onlangs tot mijn scha en schande heb ervaren: ik doorzocht een `*.zwr` file (`tekst`, weet je wel) naar afbeeldingen die in Docman stonden. Als ik de afbeeldingen *niet* vond in het bestand, dan mocht ik ze schrappen uit Docman. Wel, `grep` concludeerde dat de betreffende `*.zwr` file, `binair` was! Er stond immers bitmapindexen in, die het algoritme deden misleiden. Gevolg: `grep` vond niets. Gevolg: ik schrapte teveel afbeeldingen… (gelukkig met backup :-)

In `qtech` werd het onderscheid tussen tekstbestanden en binaire bestanden voor 2 redenen gemaakt:

- Bij zoekopdrachten: enkel tekst werd doorzocht
- Bij substitutie van `m4/i4/r4/l4` constructies: ook deze vonden enkel bij tekstbestanden plaats.

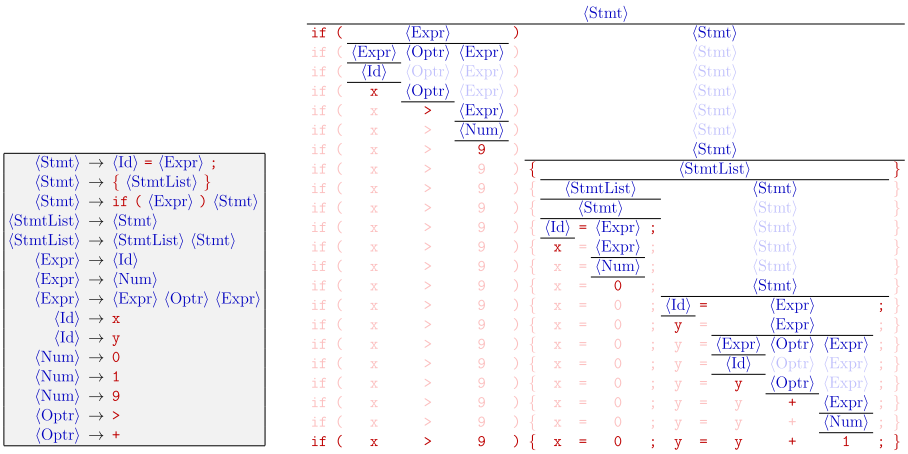
Hoe wordt het onderscheid in `QtechNG` gemaakt ?

`QtechNG` stapt af van deze willekeurige behandeling en verklaart: **alle bestanden zijn tekst totdat het tegendeel wordt geconfigureerd**

In de configuratie file `brocade.json` kan je, per project, perfect opsommen - eventueel met wildcards - wat de binaire bestanden zijn: daartoe bestaat de sleutel `binary`.

Aanverwant aan deze sleutel is er nog de sleutel `objectsnotreplaces`. Hiermee kan, per project en per object, worden aangegeven in welke *tekstbestanden* er geen substitutie mag plaatsvinden: handig bij documentatie bijvoorbeeld.

Interludium: PEG parsers



PEG (Parser Expression Grammar) parsers gaan in `QtechNG` een belangrijke rol spelen. Als parser systeem is dit een vrij recente ontwikkeling: het basis artikel van Bryan Ford verscheen in 2004: toen waren de hoogdagen van parsers al lang voorbij. Ik herinner me nog de jaren '80 toen het fameuze boek Compilers: Principles, Techniques, and Tools van Aho en Ullman (en anderen) de de facto bijbel was.

PEG parsers kwamen onder mijn aandacht toen ik veel werkte met Lua: PEG technologie zat immers standaard ingebakken in de programmeertaal en bewezen daar hoe krachtig dit wel was.

Onlangs bracht Guido Van Rossem deze parsers in de belangstelling met zijn blogreeks op Medium over zijn experimenten met Python. In PEP 617 geeft hij zelfs een aanzet om Python zelf te formuleren aan de hand van een PEG.

De reden waarom hij dit doet is precies dezelfde reden als waarom ik PEG wil gebruiken in `QtechNG`: je kan op rigoureuze wijze complexe dingen definiëren. Wist je dat er tot nu toe nog altijd geen ondubbelzinnige specificatie bestaat voor Python ? Het antwoord van het ontwikkelteam was steeds hetzelfde: *De implementatie is de specificatie.*

Onmogelijke vol te houden natuurlijk: er bestaan op dit ogenblik minstens 4 implementaties van Python, allemaal lichtjes verschillend. Hierbij zijn nog niet eens de verschillende versies van Python gerekend!

Programmeertalen maar ook andere, complexe, structuren hebben nood aan een *grammatica* die de structuur op ondubbelzinnige wijze beschrijven. Denk maar aan onze macro's en lgcodes!

In de informaticawereld heeft men daartoe een structuur van productieregels ontwikkeld. Meestal worden deze productieregels zelf geformuleerd in Backus-

Naur notatie.

Dit is een systeem waarmee men *geldige zinnen* kan maken (produceren) in de grammatica. Het *parsen* van een tekst is dan het onderzoeken of deze tekst een geldige producties is in deze grammatica. Als het even kan worden dan diverse onderdelen uit de zin herkend en daar worden dan weer *acties* mee verbonden.

Dit is precies hetzelfde als met omgangstaal: tekst is een geldige zin, we kunnen dan onderwerp, werkwoord, meewerkend voorwerp e.d. herkennen en zo begrijpen wat de zin doet.

Je kan wel denken dat het voor een parser wel heel erg complex kan zijn om een zin als *geldig* te kunnen herkennen: soms zet een gedeelte van de zin de parser op het verkeerde been en moet de parser op den duur 'backtracken' om toch de ware structuur te herkennen. Soms is dat zelfs niet mogelijk. In mijn opleiding Informatica, zovele jaren geleden, studeerde ik Algol 68, een uiterst krachtige en complexe programmeertaal. Het gerucht deed de ronde dat er geen enkele *niet-ambigue* parser voor bestond. Dat gerucht werd in ieder geval versterkt doordat de Siemens mainframe van UGent, geen enkel compilatie tot een goed einde bracht. Studenten hielden wedstrijd om de mainframe zo snel mogelijk te doen crashen met geldige Algol 68.

PEG parsers pretenderen niet dat ze ALLE grammatica's aankunnen. Ze kunnen er echter heel wat aan. Bryan Ford creëerde trouwens een nieuwe techniek - de zogenaamd Packrat parsers - die PEG kunnen laten parsen in een tijd evenredig met de lengte van de aangeboden tekst! Dit ging dan wel ten koste van het gebruikte geheugen. Ach, hoe dikwijls heb ik dit nu al gezien in informatica: de balans tussen tijd, cpu en geheugen!

PEG parsers zijn niet ambigue: met andere woorden elke geldige tekst kan maar op 1 manier geldig zijn. Anders gezegd: de specificatie is ondubbelzinnig. Dat heeft ook nadelen: PEG parsers kunnen dan ook geen grammatica's aan die van nature uit ambigue zijn. Zoals het Nederlands bijvoorbeeld. Hoe zou je een PEG parser kunnen vervaardigen die in alle omstandigheden om kan met een zin zoals *"Het baasje zocht de hond met de verrekijker"*? Je kan stellen dat PEG parsers heel erg geschikt zijn om om te gaan met *computertext* en heel wat minder met *natuurlijke taal*.

PEG parsers zijn ook context-vrij: in de parser wereld is dit *a big deal*. Het betekent dat de resultaten van 1 productieregel mogen worden gebruikt op *elke* plaats waar die productieregel staat en het resultaat zal nog steeds een geldige zin zijn.

Een PEG parser bereikt dit context-vrij zijn en het niet-ambigue karakter door op een specifieke wijze om te gaan met de keuzes in een productieregel. Een productieregel is samengesteld uit andere, eenvoudiger productieregels: net zoals teksten samengesteld zijn uit paragrafen die samengesteld zijn uit zinnen die samengesteld zijn uit woorden. De PEG parser gaat die keuzes 1 voor 1 af en zodra hij een keuze vindt die overeenkomt, zegt hij: "dat is hem" en keert niet

meer op zijn stappen terug.

Dit betekent nog dat als je een specificatie voor een bepaalde structuur uitwerkt, je wat moet opletten op de volgorde waarop je dingen test.

Ik geef een voorbeeld. In XML heb je markers voor het begin en einde van een element: vb, `<HTML>` en `</HTML>`.

Als nu je productieregel van de gedaante is:

```
marker ← ( "<" | "</") name ">"

name ← [a-zA-Z]+
```

Dan zal de PEG parser wel de zin `<HTML>` vinden maar nooit de zin `</HTML>`: hij ziet immers de "<" , die matcht en kijkt niet meer verder!

Een goede productieregel is:

```
marker ← ( "</" | "<") name ">"

name ← [a-zA-Z]+
```

Met deze specificatie zijn zowel `<HTML>` als `</HTML>` geldig!

Elke zichzelf respecterende programmeertaal heeft wel een library om PEG-parsers te vervaardigen voor tal van problemen.

Zo ook `Go` (de taal waarin `QtechNG` wordt geschreven). Er zijn er zelfs meerdere.

Mijn voorkeur gaat uit naar Pigeon, PEG … heb je hem …

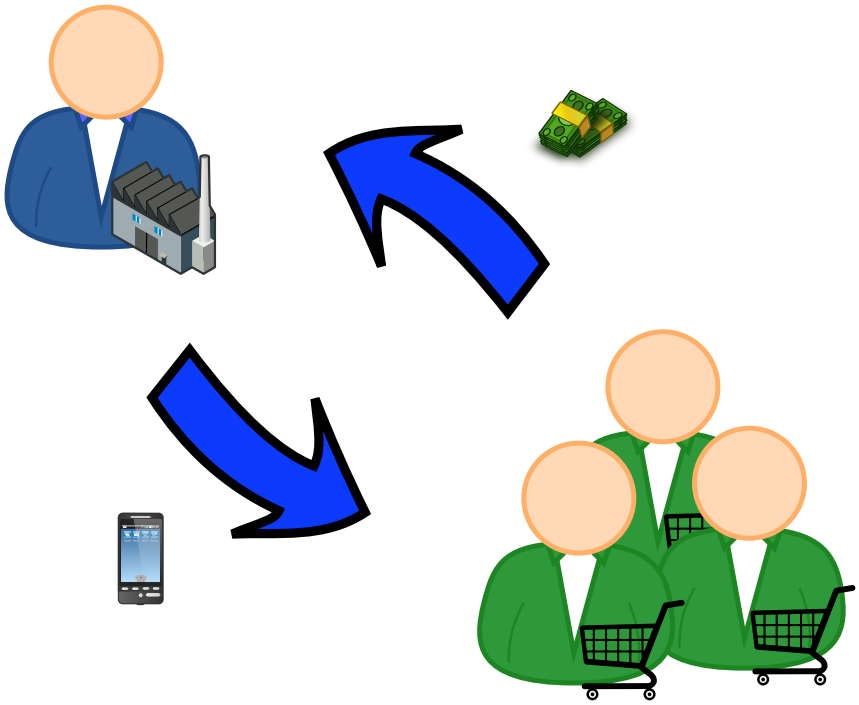
Niet alleen is Pigeon een kwalitatief hoogstaande parser generator, maar je hebt de library zelfs niet nodig in `QtechNG` !

Hoe zit dat nu?, zal je zeggen. Wel, stel dat je een PEG grammatica maakt voor pakweg een `l-file`, dan kan je Pigeon een go-software laten genereren die compleet op zichzelf staat. Je mag nadien Pigeon gewoon weggooien: je kan rustig verder met de gegenereerde code. Straf!

In latere posts gaan we dieper in op het gebruik van Pigeon bij specifieke grammatica's.

Alain en ik zijn ook aan het uitzoeken of we een generator kunnen bouwen in M. Ik zie dit als de perfecte tegenhanger van het Brocade template systeem: in het template systeem stel je een zin samen op basis van diverse onderdelen terwijl je met een PEG parser een zin opsplitst in zijn structuren en daarmee dingen doet.

Parallel



Producer - Consumer

Een doelstelling van `QtechNG` is dat overkoepelende acties zoals de `bootstrap` en de `sweep` sneller verlopen.

Met *sneller* bedoel ik niet een 10% winst maar heb ik het eerder over 2 tot 3x zo snel.

Cruciale momenten in het leven met Brocade zijn de oplevering van nieuwe releases en het installeren van een nieuwe ontwikkelversies.

Beiden nemen nu uren in beslag. Niet alleen zou het een stuk comfortabeler zijn indien dit *sneller* zou kunnen, maar het zou bovendien de mogelijkheid scheppen om het aantal uitgebrachte releases drastisch op te voeren. Voeg je daarbij ook de systeem configuratie, via Ansible, ook opgenomen wordt in `QtechNG`, dan is het duidelijk dat elk beetje extra helpt bij het bereiken van de doelstelling.

De voornaamste troef om deze snelheid te bereiken is parallellisme bij het uitvoeren van de diverse taken.

… en laat nu dat zijn waarin `Go` excelleert!

Ik wil echter van meet af aan stellen dat *het onderste uit de kan halen* niet erg verstandig zou zijn!

Het werken in parallel, diverse taken die op hetzelfde moment worden uitgevoerd, kunnen subtiele problemen opleveren die zelden naar boven komen in testscenario's maar die op het moment dat je het net niet kunt hebben, opduiken.

Trouwens, het heeft geen zin een paar seconden af te schaven als de totale tijd van de operatie toch een uur of langer neemt!

Bovendien zijn er heel wat situaties waarin je parallellisme best mijdt: lees er eens "Async Python is not faster" op na!

Daarom hanteerde ik als aanpak voor parallellisme, de volgende vuistregels:

- Ik gebruik enkel parallellisme bij intensieve I/O operaties
- Ik gebruik enkel het 1 producer / multiple consumer patroon via een software die ikzelf heb geschreven en door en door begrijp
- Ik structureer mijn software als pipelines

In een vorige blog had ik het over de verschillende stappen die nodig zijn om een bestand op te slaan in `QtechNG`. Om bijvoorbeeld 1 Mumps file weg te schrijven zijn er gemakkelijk acties nodig op 20 verschillende bestanden! (Hierin is dan nog niet de installatie van de M file in de Mumps omgeving zelf opgenomen). Je zou nu kunnen denken: Ok! Een goed moment om deze 20 acties *gelijktijdig* te laten gebeuren. Wel, dat gebeurt dus niet! Het is gewoon veel te moeilijk voor mij om te doorgronden wat ik moet doen als er 1 van deze 20 stappen misloopt. Voeg daar nog bij dat diverse van die stappen afhankelijk van elkaar zijn: m.a.w. het succesvol aflopen van de ene actie is afhankelijk van het succesvol aflopen van een andere actie. En dan zijn er nog tal van andere problemen die zich aandienen. Maar daarover later meer …

Pipelines

Met *pipelines* wordt software opgedeeld in een aantal delen die één voor één (na elkaar) worden uitgevoerd: m.a.w. het tegenovergestelde van parallellisme! Binnen elk deel kan je echter voluit gaan. Het grote voordeel is dat je na elk deel *weet* wat de status is van je toepassing: het is een moment waarop de chaos voorbij is (en hoogst waarschijnlijk een nieuwe chaos begint).

Ik geef een voorbeeld van een pipeline in `QtechNG`.

Stel, we gaan een nieuw project opladen in het repository. Dit project bestaat uit 20 M-files. Een hoogst efficiënte manier van werken zou kunnen zijn: het project aanmaken in het repository en 20 simultane taken starten: 1 per M-file. Gaan we ervan uit dat elke taak bestaat uit het bewerken van 10 andere bestanden en het installeren van de M-file in de Mumps omgeving, nog eens 2 files raakt. In `Go` is het een koud kunstje om die `20 * (10 + 2) = 240` *goroutines* op te starten die dit voor hun rekening moeten nemen. Dit is werken *ZONDER* pipelining: uiterst snel en … onbeheersbaar! Denk maar eens na wat er zou gebeuren indien er, behalve die 20 M-files, ook nog een `d-file` met macro's in het project zou bestaan. (Tussen (), hoeveel projecten ken je zonder d-files ?) Ik geef je op een briefje, dat bij diverse M-files, de macro's niet zouden zijn omgevormd!

Met *pipelining* pakken we het wat bescheidener aan: we splitsen de werkzaamheden in 2 delen.

In het eerste deel stoppen we de 10 M-files en de eventuele d-files in het repository. Deze werkzaamheden zijn onafhankelijk van elkaar: we besteden 1 `goroutine` per file. Is dit deel afgelopen, dan weten we dat de bestanden perfect in het `QtechNG` repository staan. Het tweede deel kan beginnen.

In het tweede deel installeren we de M-routines en of we dit in parallel afwerken moet ik nog uitvissen!

Producer / Consumer

De valkuilen van het *producer/consumer* design patroon zijn heel goed gekend en ik voel me bekwaam genoeg om hier een go functie voor te maken die aan de eisen van `QtechNG` tegemoet komt: in de package `brocade.be/base/parallel` staat de functie `NMap`

```
func NMap(n int, parmax int, fn func(m int) (r interface{}, err error)) (resultlist []interf
...
}
```

Deze functie werkt uitstekend in de context van een `closure`: voor elke getal beginnend van 0 tot en met `n`-1 moet een functie `fn` worden uitgevoerd. `NMap` verzamelt de resultaten en de fouten van `fn`.

De bedoeling is om deze opdrachten in parallel uit te voeren. We houden het aantal taken die tegelijkertijd lopen, strikt onder controle:

- `parmax`: is dit getal positief, dan mogen er nooit meer *goroutines* worden gelanceerd
- de registry waarde `qtechng-max-parallel`: is deze waarde niet leeg en `parmax` negatief, dan worden er nooit meer *goroutines* opgestart
- is deze waarde toch leeg en `parmax` is negatief, dan worden er nooit meer `goroutines` opgestart dan het aantal cores op de machine (min 1).

Waarom de wens om het aantal goroutines te beperken? Dit is iets dat je met scha en schande leert: ik kwam vlug tot de ontdekking dat er plots problemen opduiken die je niet zo snel ziet aankomen. Bijvoorbeeld dat het aantal files die tegelijkertijd open zijn, beperkt is. Deze beperkingen kunnen per proces en overkoepelend voor het ganse werkend Operating System zijn!

`NMap` werkt volgens het *producer/consumer* patroon. Er worden een beperkt aantal *consumers* (werkers) opgestart, die elk een aantal getallen afwerken. De *producer* zorgt ervoro dat er voldoende werk uitstaat.

Toch nog opletten!

Zelfs met deze voorzichtige aanpak kunnen er nog problemen rijzen: in package `brocade.be/qtechng/source` definieer ik 2 functies die cruciaal zijn voor `QtechNG`:

- `StoreList`: deze gaat een lijst met source bestanden in parallel *onderbrengen* in het repository
- `UnlinkList`: deze gaat een lijst met source bestanden in parallel *schrappen* uit het repository

Ondanks mijn voorzichtige aanpak maakte ik een kapitale blunder: ik hield geen rekening met de speciale aard van de `brocade.json` bestanden. Bij de opslag moeten deze *eerst* worden gedaan: ze kunnen immers verhinderen dat andere bestanden mogen worden opgeslagen (denk maar aan de uniciteitsregel). Bij het schrappen moeten ze helemaal op het einde worden verwijderd. Gebeurt dit allemaal in parallel, dan weet je ooit welke file er eerst wordt behandeld.

Zelfs het splitsen van deze taak is dat nog niet voldoende: je moet er rekening mee houden dat projecten sub-projecten kunnen hebben. Dit betekent nog dat de `brocade.json`'s bij de opslag moeten worden gesorteerd en bij het schappen zelfs in omgekeerde volgorde.

(Zucht)

Eerste resultaten

Is het dit allemaal wel waard ?

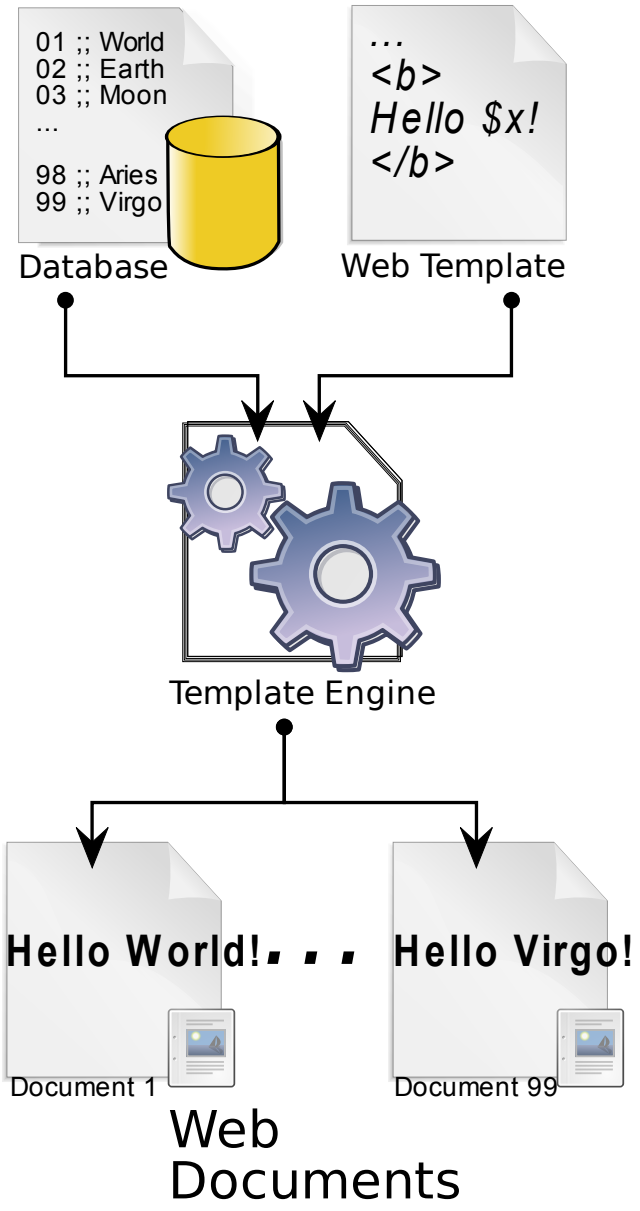
Wel, de volgende test overtuigde me: De volgende cijfers geven geen sluitend beeld en ze zijn bekomen op mijn notebook (XPS 13 van 2017, 8 GB RAM, 4 cores, Linux Mint 19.3).

- 1000 projecten
- elk 100 bestanden
- elk bestand 32K
- elk bestand telde een 10-tal objecten
- maximaal 16 parallelle acties (qtechng-max-parallel)

Het opbouwen (`StoreList`) en afbreken (`UnlinkList`) van het repository nam minder dan 15 minuten in beslag.

Wel bemoedigend!

Interludium: Het Brocade template systeem



Het *Brocade Template System* (BTS) is werkelijk uit zijn voegen gebarsten. Niet alleen worden de mogelijkheden nog geregeld uitgebreid maar ook de toepassingsgebieden nemen steeds toe.

BTS is het ideale niveau tussen programmeren enerzijds en metadatering an-

derzijds. Gebruikers hebben vrij snel door wat de grote contouren zijn van het template systeem. Het zit hem - net zoals bij programmeren - echter in de details, en dat vergt grondige kennis van de diverse mogelijkheden en, ja, ook enige creativiteit.

Daarom organiseren we een tutorial over BTS en deze tekst dient om deze tutorial wat te begeleiden.

Basisinstrumenten

Eerst en vooral is er de documentatie: dit is een wat droge maar uitgebreide omschrijving van de mogelijkheden.

Er werd ook een speeltuin voorzien waarmee templates kunnen worden getest en ideeën uitgeprobeerd. Zeker als je complexe templates wil maken, is deze faciliteit aan te raden. Je experimenten worden er ook bewaard! (Ben je er wat beschaamd over, geen nood, je kan ze ook wissen!)

Geschiedenis

Voordat *BTS* werd ontwikkeld, werkten we in Brocade met een ander template systeem. Dat systeem heeft veel gemeenschappelijks met BTS maar is complexer, minder rigoureus gedefinieerd en is ook heel wat minder krachtig. Spijtig genoeg is dit systeem nog op diverse plaatsen in gebruik. We belichten later welke technieken er kunnen worden ontplooid om dit probleem aan te pakken.

De onderdelen van een template

Een template bestaat uit 2 delen:

- een *voorschrift* (ook wel de template genaamd)
- een verzameling van placeholder/waarde paren: de sleutels

Deze onderdelen komen precies overeen met de 2 *partijen* die betrokken zijn bij het werken met templates:

- de persoon die de template samenstelt
- de persoon die de nodige placeholders ter beschikking stelt

De persoon die de templates samenstelt, moet weten welke *placeholders* (de eerste component van een sleutel) ter beschikking staan.

Bedoeling van een template

Teksten, webpagina's, data … ze bestaan allemaal uit karakters. Sommige onderdelen van deze teksten zijn vaste gegevens, andere zijn dan weer afhankelijk van de context waarin ze worden gebruikt.

Het *voorschrift* van de template gaat zo goed mogelijk de bewuste teksten voorstellen. De dingen in de tekst die veranderen, gaan dan worden voorgesteld door een *placeholder*.

Dit *vervangen* van de placeholder door zijn waarde, lijkt eenvoudig maar dit gaat meestal gepaard met een logica en deze logica vatten, is het moeilijke in het omgaan met templates.

Dit vervangen van de placeholders samen met de bijhorende logica noemen we *het evalueren van de template tegenover de sleutels*

BTS is tekstueel: dit betekent dat zowel *template*, *placeholders* en *resultaat* steeds karakterrijen (strings) gaan zijn.

Het is een goede raad om *templates* te formuleren in ASCII, toch mogen ze - net zoals waarden en het resultaat - perfect in UTF-8 zijn.

Nog even benadrukken dat BTS gericht is om te worden gebruikt in een M omgeving en vanzelfsprekend komt het systeem vooral tot zijn recht in Brocade.

Voorbeelden van templates

In de volgende voorbeelden gaan we spaties en dergelijke wat 'speciaal' voorstellen. De student weet dan precies wat er met de `whitespace` gebeurt!

Een template zonder sleutels (en zonder context: er is slechts 1 mogelijk resultaat)

```
De hoofdstad van België is Brussel
```

evalueert tot:

```
De␣hoofdstad␣van␣België␣is␣Brussel
```

Als we een overzicht van alle landen met hun hoofdstad willen maken, dan moeten we veranderlijkheid inbouwen in onze template.

Merk nog op dat het `$` teken een placeholder inleidt. Er bestaan verschillende soorten placeholders, de meest voorkomende is diegene die met een `$` begint en verder uit kleine letters en cijfers bestaat.

Een neveneffect van het werken met speciale karakters is dat niet alle letters letterlijk mogen worden genomen: sommigen - zoals de `$` hebben een speciaal gebruik. BTS draagt er zorg voor dat deze speciale karakters *zeldzaam* zijn in het gewone gebruik.

```
Land: $country
Hoofdstad: $capital
```

Met sleutels:

- country: België
- capital: Brussel

wordt het resultaat:

```
Land:␣België Hoofdstad:␣Brussel
```

Ok, maar wat als je nu een dollar symbool in je tekst wil zetten? Dan moet je `escapen`: dit is laten voorafgaan door een `\`. Meteen ken je een tweede speciaal karakter.

```
Bij McDonald's kost een kop koffie 1\$
```

Na evaluatie:

```
Bij␣McDonald's␣kost␣een␣kop␣koffie␣1$
```

Placeholders

Gewone sleutels

We hebben reeds een eerste type van templates gezien: deze die beginnen met `$` en verder uit letters en cijfers bestaan.

Maar hoe weet je nu waar de placeholder *eindigt* in de template ? (Beginnen is gemakkelijk: dat is een `$`)

Stel dat je "Kareltje" wil zeggen, en je placeholder is `$kind` met de waarde `Karel`.

Wordt de template dan `$kindtje` ?

Neen, natuurlijk niet: de template kan onmogelijk vermoeden dat de placeholder `$kind` is!

Daarom moet je de *reikwijdte* van de placeholder afschermen met `{` en `}`. De template wordt:

```
{$kind}tje
```

of zelfs:

```
${kind}tje
```

`{` en `}` zijn veel gebruikte speciale karakters in `BTS` maar hun doel is steeds hetzelfde: een gebied markeren en afschermen.

Tekstfragmenten

Een andere vorm van placeholder is die met een `.` en voor de rest cijfers en letters.

De placeholder verwijst dat naar een tekstfragment.

Met:

```
lgcode alphatoomega:
N: «A tot Ω»
```

```
E: «A to Ω»
F: «A to Ω»

lgcode menu.templateexec:
N: «Gecontroleerde code voor templates»
E: «Checked code for templates»
F: «Code contrôlé pour templates»
```

De template

```
Van $.alphatoomega: $menu.templateexec
```

evalueert in het Nederlands tot:

```
Van␣A␣tot␣Ω:␣Gecontroleerde␣code␣voor␣templates
```

Meteen leren we dat het evalueren van een template steeds in de context van een taal gebeurt.

Andere templates

Een placeholder kan ook beginnen met een `@`. De placeholder verwijst dan naar een andere template.

Omgevingsplaceholders

Als een placeholder begint met een `$` gevolgd door een *HOOFDLETTER* en het is geen van de vorige placeholders, dan wordt deze gezocht in de werkende M omgeving

Vaste placeholders

Als er gebruik gemaakt wordt van de een placeholder die niet tot de vorige groepen behoort, dan wordt hij gezocht in de collectie van de vaste placeholders.

- technische waarden: `maxindex`
- context: `lg`, `workstation`, `staff`, `job`, `date`, `time`, `session`, `page`, `desktop`, `service`, `ip`
- alle Griekse letters: `alpha`, `beta`, `gamma`, …
- buitenbeentjes: `e`, `smile`, `unique`
- nuttig: `true`, `false`, `grave`, `question`, `empty`, `quotation`, `minus`, `apostrophe`, `percent`, `space`, `crlf`, `wspace`

Meervoudige placeholders

Sommige placeholders kunnen meer dan 1 waarde hebben. Standaard is er de placeholder `$planet`, deze heeft als (opeenvolgende waarden): Mercurius, Venus, Aarde, Mars, Jupiter, Saturnus, Uranus, Neptunus, Pluto.

We tonen verder hoe deze kunnen worden gebruikt.

De default waarde

Er kan een default waarde zijn voor placeholders. Indien de placeholder nergens wordt gevonden en er *is* een default waarde, dan wordt dit de waarde.

Een default waarde kan worden gespecificeerd in de sleutelverzameling, in de template of in de oproepende code.

Is er geen default waarde, dan crasht de code!

De pijplijn

Templates vormen een echte pijplijn: op elk moment is er een resultaat en dat resultaat kan worden vervormd to een ander resultaat en dan weer …

Het speciale karakter dat de volgende fase van de transformatie inluidt, is `|`. Het wordt gevolgd door een *werkwoord* en argumenten. Werkwoord en argumenten zijn gescheiden door een spatie.

De argumenten kunnen placeholders zijn, getallen of strings tussen dubbele aanhalingstekens.

Eens je in de pijplijn zit, weet BTS steeds of je met een placeholder te maken hebt. Bijgevolg hoeven de `$` niet meer. Ook het `escapen` valt weg.

De werkwoorden heten `modifiers`. Welke modifiers er zijn en hoe ze worden gebruikt moet je uit de documentatie halen. Er zijn er op dit ogenblik reeds 136 gedefinieerd en deze behandelen diverse situaties. Elk op zich zijn het eenvoudige constructies maar samen vormen ze een uiterst krachtig systeem.

Met behulp van `{` en `}` moet je steeds aangeven welk gebied er onderhevig is aan de pijplijn.

`{` en `}` mogen genest zijn. Dit betekent dat je grote flexibiliteit hebt in het manipuleren van je tekst of delen van je tekst.

String manipulatie

De template

```
{Dag | after " allemaal"}
```

evalueert tot:

```
Dag␣allemaal
```

De template

```
{Dag | after staff | upper}
```

evalueert tot:

```
DAGRPHILIPS
```

De template

```
{dag | after wspace | after staff | ucfirst}
```

evalueert tot:

```
Dag␣rphilips
```

Brocade concepten

Hier schittert BTS: de voornaamste Brocade concepten worden op een simpele manier ter beschikken gesteld van de template bouwer.

De template

```
{dag {$staff|staff} | ucfirst}
```

evalueert tot:

```
Dag␣<a␣href="mailto:richard.philips@uantwerpen.be">Richard␣Philips</a>
```

De template

```
{Titelbeschrijven | menu "bibrec"}
```

evalueert tot:

```
<a␣href="/menu/bibrec">Titelbeschrijven</a>
```

De template

```
{ | menu "bibrec"}
```

evalueert tot:

```
<a href="/menu/bibrec">Titelbeschrijvingen invoeren/wijzigen</a>
```

De template

```
{Titelbeschrijven | menu "bibrec" "" "c:lvd:3"}
```

evalueert tot:

```
<a␣href="/menu/bibrec/c:lvd:3">Titelbeschrijven</a>
```

BTS concepten

```
{menu.templateexec | text}
```

evalueert tot:

```
Gecontroleerde␣code␣voor␣templates
```

Stel dat we de sleutel *menuitem: templateexec* hebben,

dan:

```
{menu.$menuitem | text}
```

evalueert tot:

```
Gecontroleerde␣code␣voor␣templates
```

Booleaans waarden

Elke waarde heeft ook een `waarheidswaarde` en kan worden gebruikt in een Booleaanse uitdrukking:

- een lege string is `vals`
- `0` is `vals`
- de rest is `waar`

Dan zijn er de Booleaanse uitdrukkingen:

- `uitspraak` is enkel waar indien `!uitspraak` vals is
- `uitspraak1&&uitspraak2` is enkel waar indien beiden waar zijn
- `uitspraak1||uitspraak2` is enkel vals indien beiden vals zijn

Stel dat we 2 sleutels hebben:

- `$title`: `Leven en dood in den ast`
- `$author`: `Stijn Streuvels`

De template:

```
{$title {/ $author | stay $author}}
```

evalueert tot:

```
Leven␣en␣dood␣in␣den␣ast␣/␣Stijn␣Streuvels
```

Zijn de sleutels echter:

- `$title`: `Recipes for Disaster: An Anarchist Cookbook`
- `$author`: “

dan evalueert de template tot:

```
Recipes␣for␣Disaster:␣An␣Anarchist␣Cookbook␣
```

Wiskundige constructies

Met `<` en `>` kunnen uit wiskundige constructies Booleaanse waarden worden gemaakt.

De `?` modifier

Deze modifier moet steeds vooraan staan in de pijplijn en er kunnen 1 of 3 argumenten zijn.

Is het eerste argument `false`, dan verdwijnt het ganse gebied! Het gebied zelf wordt niet eens geëvalueerd!

```
{Karel en Herman{ en Vincent | ? 1>2}}
```

evalueert tot:

```
Karel␣en␣Herman
```

Met 3 argumenten kan de modifier gebruikt worden in een loop:

```
{{$planet, | after " "} | ? true 1 ##planet}
```

evalueert tot:

```
Mercurius,␣Venus,␣Aarde,␣Mars,␣Jupiter,␣Saturnus,␣Uranus,␣Neptunus,␣Pluto,␣
```

Nog beter:

```
{{{$planet, | after " "} | ? true 1 ##planet-1} | rstrip ", "} en {$planet | ? true ##planet
```

evalueert tot:

```
Mercurius,␣Venus,␣Aarde,␣Mars,␣Jupiter,␣Saturnus,␣Uranus,␣Neptunus␣en␣Pluto
```

Eigenlijk kan het eenvoudiger :-)

```
{planet | join ", " 1 -1} en {planet | subscript -1}
```

evalueert tot:

```
Mercurius,␣Venus,␣Aarde,␣Mars,␣Jupiter,␣Saturnus,␣Uranus,␣Neptunus␣en␣Pluto
```

Merken we op: een `##` voor een placeholder in de argumenten van een modifier betekent: het `aantal elementen` bij de placeholder.

Bij een meervoudige placeholder (zoals `$planet`) is dit eenvoudig. Voor een enkelvoudige placeholder is het resultaat verrassend:

```
{| after ##title}
```

evalueert tot:

```
9999999
```

Tips & tricks

Oud versus nieuw template systeem

Gebruik steeds BTS!

Als je een template laat beginnen met de constructie:

```
{|ng}
```

dan weet het 'oude' template systeem van Brocade dat je de template in BTS wil laten evalueren. Dus als je twijfelt, of je template met het oude of het nieuwe template systeem gaat worden geëvalueerd, dan voeg je best `{|ng}` in.

Default waarden

De template bouwer kan zelf een `default` waarde zetten:

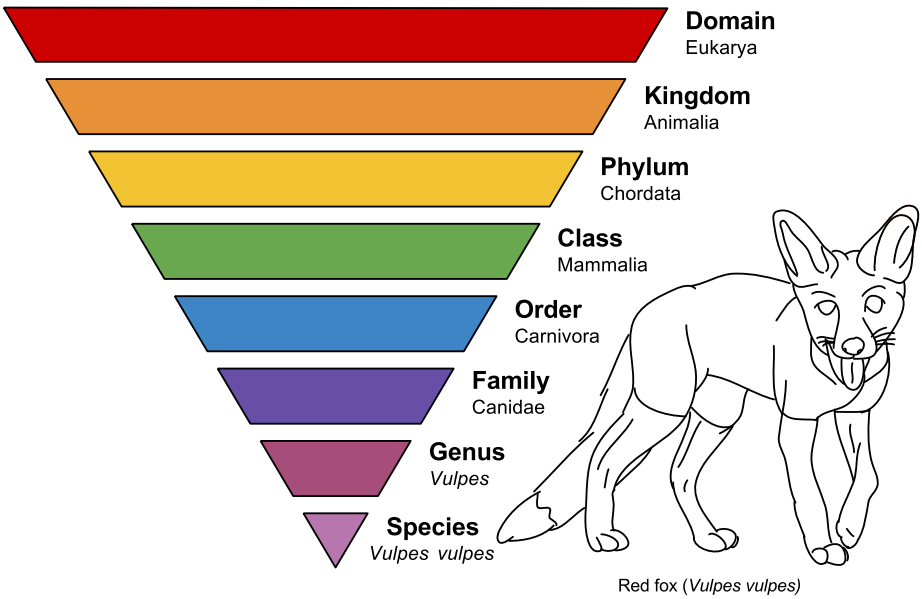
```
{| setdefault "World"}Hello $onbekend
```

geeft:

```
Hello␣World
```

Wees echter voorzichtig hiermee: deze praktijk kan fouten in je template maskeren.

Naturen



Met QtechNG wordt er ook een classificatie van de bestanden geïntroduceerd.

Dit is in feite een formalisering van wat reeds jaar en dag gebruikt wordt door het ontwikkelteam.

Een bestand kan meerdere naturen hebben.

Deze naturen hebben verschillende functies:

- ze sturen de installatie
- ze zijn belangrijk voor m4/r4/i4 substitutie
- ze kunnen worden gebruikt bij `queries`

`binary` en `text`

Deze naturen sluiten elkaar uit. Standaard wordt elk bestand als `text` beschouwd. Enkel het corresponderend configuratiebestand (`brocade.json`) kan door middel van de parameter `binary` sommige bestanden als binair markeren.

`config`, `install`, `release`, `check`

Elk project heeft juist 1 configuratiebestand (`brocade.json`). Dit bestand heeft de natuur `config` (en `text`).

In dezelfde directory als het configuratiebestand kunnen zich ook de `Python` scripts

- `install.py` (natuur: `install` + `text`)
- `release.py` (natuur: `release` + `text`)
- `check.py` (natuur: `check` + `text`)

bevinden. De werking en de betekenis van deze bestanden worden later uitvoerig behandeld.

`bfile`, `dfile`, `ifile`, `lfile`, `mfile`, `xfile`

Bestanden die de natuur `text` hebben en die niet voorkomen in het lijstje bestanden bepaald door de parameter `notbrocade` uit de configuratiefile, kunnen gekarakteriseerd worden door hun extensie: de bestanden met extensie `*.b`, `*.d`, `*.i`, `*.l`, `*.m`, `*.x` krijgen een corresponderende natuur.

`auto`, `objectfile`

Bestanden met natuur `dfile`, `lfile`, `ifile` krijgen ook nog eens de natuur `objectfile` mee.

Bestanden met natuur `bfile`, `dfile`, `ifile`, `lfile`, `mfile` of `xfile` zijn ook van natuur `auto`.

Search

De meest gebruikte faciliteit van `qtech` is ongetwijfeld het zoeken in het software repository.

Dit moet nog beter (lees: sneller) worden in `QtechNG`.

De basis van de zoek-software is de volgende golang structuur:

```
type Query struct {
    Release string
    CmpRelease string
    Patterns []string
    Natures []string
    Cu []string
    Mu []string
    CtBefore string
    CtAfter string
    MtBefore string
    MtAfter string
    ToLower bool
    Regexp bool
    PerLine bool
    Contains []string
    Any []func(qpath string, blob []byte) bool
    All []func(qpath string, blob []byte) bool
}
```

Attributen

Elk van deze attributen stelt een specifiek type van zoeken voor. Zijn er meerdere attributen gedefinieerd, dan zijn dit **AND** searches: aan elk attribuut moet worden voldaan.

Dit impliceert ook dat - indien er meerdere attributen ingevuld zijn - de volgorde erg belangrijk wordt: sommige zoekacties vergen nu éénmaal meer resources dan andere.

Laten we even de verschillende attributen overlopen. Nadien zullen we uitleggen hoe één en ander in elkaar zit.

Release

Indien verschillend van leeg, dan moeten zoekresultaten steeds binnen de gegeven `Release` worden gezocht.

straight forward, dit attribuut moet steeds gespecificeerd zijn (impliciet of expliciet).

CmpRelease

Ook dit attribuut stelt een release voor (m.a.w. een string van de gedaante `5.10`). Een bestand komt in het zoekresultaat indien er aan één van de volgende voorwaarden wordt voldoen:

- het bestand bestaat **niet** in release CmpRelease
- het bestand verschilt van inhoud ten opzichte van release CmpRelease

Patterns

Wat de snelheid van zoeken betreft, is dit een cruciaal attribuut: het is een lijst met wildcards. Enkel bestanden waarvan het `QtechNG` path voldoet aan één of meer van deze patronen, worden weerhouden.

Belangrijk is dat de bestanden zelf niet moeten worden ingelezen.

Natures

In een vorige blog hebben we het al gehad over de naturen van source code. Enkel de bestanden met een natuur in deze lijst worden weerhouden.

De bestanden zelf hoeven niet te worden ingelezen. Wel de corresponderende configuratiefiles `brocade.json`. Hiervan zijn er natuurlijk heel wat minder.

CtBefore

Hiervoor wordt de meta informatie van de betreffende source code opgehaald. Enkel indien de *creation time* kleiner of gelijk is aan CtBefore, wordt het bestand weerhouden.

De tijden worden steeds weergegeven in de vorm `YYYY-MM-DDThh:mm:ss` (ongeldige tijdstippen worden steeds genormaliseerd door overflow: vb. als hh = 25, dan schuift DD op met 1 en hh wordt 01)

CtAfter

Hiervoor wordt de meta informatie van de betreffende source code opgehaald. Enkel indien de *creation time* groter of gelijk is aan CtBefore, wordt het bestand weerhouden.

De tijden worden steeds weergegeven in de vorm `YYYY-MM-DDThh:mm:ss` (ongeldige tijdstippen worden steeds genormaliseerd door overflow: vb. als hh = 25, dan schuift DD op met 1 en hh wordt 01)

Cu

Is een lijstje met de userid's van de mogelijke creators. Enkel indien de daadwerkelijke creator in het lijstje zit, wordt het bestand weerhouden.

MtBefore

Hiervoor wordt de meta informatie van de betreffende source code opgehaald. Enkel indien de *modification time* kleiner of gelijk is aan MtBefore, wordt het bestand weerhouden.

De tijden worden steeds weergegeven in de vorm `YYYY-MM-DDThh:mm:ss` (ongeldige tijdstippen worden steeds genormaliseerd door overflow: vb. als hh = 25, dan schuift DD op met 1 en hh wordt 01)

MtAfter

Hiervoor wordt de meta informatie van de betreffende source code opgehaald. Enkel indien de *modification time* groter of gelijk is aan CtAfter, wordt het bestand weerhouden.

De tijden worden steeds weergegeven in de vorm `YYYY-MM-DDThh:mm:ss` (ongeldige tijdstippen worden steeds genormaliseerd door overflow: vb. als hh = 25, dan schuift DD op met 1 en hh wordt 01)

Mu

Is een lijstje met de userid's van de mogelijke last modifiers. Enkel indien de daadwerkelijke modifier in het lijstje zit, wordt het bestand weerhouden.

Contains

Dit is een lijstje met strings (`needles`). De precieze werking is afhankelijk van de waarden van `PerLine`, `ToLower` en `Regexp`.

In essentie wordt de inhoud van de bestanden gecontroleerd tegen het lijstje met needles. **ALLE** needles moeten tot het bestand behoren.

Er zijn nu diverse situaties:

- is `PerLine` waar, dan moeten alle needles in dezelfde lijn worden gevonden. Anders wordt gewoon gekeken in het bestand.
- is `Regexp` waar, dan worden de needles omgevormd tot reguliere uitdrukkingen.
- is `ToLower` waar, dan wordt de inhoud van het bestand omgezet naar kleine letters. Is `Regexp` onwaar, dan worden ook de needles omgezet naar kleine letters (met `Regexp` waar, werkt dit niet: de betekenis van de reguliere uitdrukking zou immers kunnen veranderen. vb. `\D` omzetten naar `\d` verandert de zoekactie helemaal.)
- is `SmartCase` waar en `Regexp` onwaar, dan wordt `ToLower` op waar gezet indien ALLE needles in lowercase staan. Is er echter een needle die grote letters bevat, dan wordt `ToLower` op onwaar gezet.

Any

Dit is een gespecialiseerde zoekattribuut: het attribuut is een lijst van functies. Elke functie werkt met 2 argumenten: een `QtechNG` path naam en een inhoud in bytes. Onderhuids wordt de path naam van elk bestand dat we tegenkomen, samen met zijn inhoud gegeven aan deze functies. Van zodra er 1 functie waar is, wordt het bestand geselecteerd.

All

Dit is een gespecialiseerde zoekattribuut: het attribuut is een lijst van functies. Elke functie werkt met 2 argumenten: een `QtechNG` path naam en een inhoud in bytes. Onderhuids wordt de path naam van elk bestand dat we tegenkomen, samen met zijn inhoud gegeven aan deze functies. Enkel indien *ALLE* functies waar zijn, wordt het bestand geselecteerd.

Uitwerking

Uit de beschikbare attributen is het duidelijk dat `Query` niet enkel is bedoeld om door een UI te worden aangestuurd: **Any** en **All** maken de software ook heel erg geschikt voor interne doeleinden.

De effectieve implementatie gebeurt geheel in termen van concurrency en pipelining.

De pipeline bevat 3 fases:

- Optimalisering van de `Query`
- in de tweede fase worden de kandidaat bestanden opgelijst

- in de derde fase wordt elke kandidaten getest tegen `Query`.

In de eerste fase wordt de `Query` zelf geanalyseerd en geoptimaliseerd: reguliere uitdrukkingen worden gecompileerd, overbodige patronen worden geschrapt en passende Boyer-Moore skiptabellen worden aangemaakt. Voor alle duidelijkheid: de searches - als die al bestaan - zijn niet altijd volgens Boyer-Moore. Er zijn immers nogal wat situaties waar dit te weinig zou opbrengen. De lengte van de te doorzoeken bestanden speelt hierbij de hoofdrol: bevatten deze bestanden minder dan 64 karakters dan wordt er gewoon *brute force* gezocht. Die '64' valt niet zomaar uit de lucht (en heeft ook niets te maken met het schaakspel). Diverse zoekstrategieën in de golang code zelf maken gebruik van dit getal.

Het Boyer-Moore algoritme werd nog verbeterd met de (Galil)[https://en.wikipedia.org/wiki/Zvi_Galil] aanpassingen.

Indien het bestand de neiging heeft om binair te zijn (een NULL-byte in de eerste 1024 bytes), schakelt het zoekalgoritme over naar (Rabin-Karp)[https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm].

In de tweede fase wordt er maximaal gebruik gemaakt van de mogelijkheden geboden door `Patterns` (en `Release`). De kandidaten worden parallel opgespoord door analyse van de QtechNG paths. Om sneller te werken worden in de tweede fase van de pipeline een aantal *fouten* toegelaten.

In de derde fase wordt echter elk bestand uit de eerste fase rigoureus (en in parallel) getest tegen de *ganse* Query. Indien deze werkwijze je doet denken aan de filosofie achter het gebruik van Bloom filters, dan zit je goed!.

Benchmarks

Tijd voor benchmarks!

We gaan ons niet bezig houden met micro benchmarks: laten we meteen een realistisch scenario opzetten.

Ik converteerde qtech's 5.00 release naar QtechNG's 9.98. Deze conversie was compleet: sources, objecten en meta informatie.

Ik wou 3 searches uitvoeren: deze waren op zich niet zo heel belastend maar wel representatief voor het concrete gebruik.

Volg nog even een overzicht van de hardware van mijn XPS-13 (2016):

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) per core:              2
Core(s) per socket:              2
```

```
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           61
Model name:                      Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
Stepping:                        4
CPU MHz:                         936.564
CPU max MHz:                     3000,0000
CPU min MHz:                     500,0000
BogoMIPS:                        4789.15
Virtualization:                  VT-x
L1d cache:                       64 KiB
L1i cache:                       64 KiB
L2 cache:                        512 KiB
L3 cache:                        4 MiB
NUMA node0 CPU(s):               0-3
```

Search 1

```
Query{
    Release: "9.98",
    Patterns: []string{"/*"},
    Contains: []string{"m4_getCatIsbdTitles"},
}
```

Deze zoekactie doorzocht alle bestanden naar de string “m4_getCatIsbdTitles”.

De sublieme benchmarking van Go gaf het volgende resultaat:

```
Running tool: /usr/local/go/bin/go test -benchmem -run=^$ brocade.be/qtechng/source -bench 1

goos: linux
goarch: amd64
pkg: brocade.be/qtechng/source
BenchmarkQuery77-4      1000000000           0.0259 ns/op          0 B/op          0 allocs/
PASS
ok      brocade.be/qtechng/source   0.311s
```

Met andere woorden, een zoekresultaat binnen (ongeveer) een 0.3 seconden.

Search 2

```
Query{
    Release: "9.98",
    Patterns: []string{"/*.m"},
    Contains: []string{"m4_getCatIsbdTitles"},
}
```

Deze zoekactie doorzocht alle bestanden naar de string “m4_getCatIsbdTitles” maar beperkt zich enkel tot de m-files.

```
Running tool: /usr/local/go/bin/go test -benchmem -run=^$ brocade.be/qtechng/source -bench 1

goos: linux
goarch: amd64
pkg: brocade.be/qtechng/source
BenchmarkQuery78-4      1000000000           0.0234 ns/op          0 B/op          0 allocs/
PASS
ok      brocade.be/qtechng/source   0.196s
```

Met wat goede wil, een halvering van het vorige zoekresultaat.

Search 3

In de derde zoekactie willen we de bestanden opzoeken die - net zoals in de vorige zoekacties, de string “m4_getCatIsbdTitles” bevatten, maar waarvan de bestandsnaam niet voldoet aan de style guide.

Ik geef de volledige benchmarkfunctie:

```
func BenchmarkQuery79(t *testing.B) {
    r := "9.98"

    query := &Query{
        Release:  r,
        Patterns: []string{"/*"},
        Contains: []string{"m4_getCatIsbdTitles"},
        All: [](func(qpath string, blob []byte) bool){
            func(qpath string, blob []byte) bool {
                k := strings.LastIndex(qpath, "/")
                if k == -1 {
                    return false
                }
                base := qpath[k+1:]
                if len(base) < 4 {
                    return false
                }
                fourth := rune(base[3])
                ok := strings.HasPrefix(base, "g") || strings.ContainsRune("wuts", fourth)
                return !ok
            },
        },
    }

    result := query.Run()
```

```
    if len(result) != 13 {
        t.Errorf("Should have found 72: %d", len(result))
        fmt.Println(query)
        for _, x := range result {
            fmt.Println(x.String())
        }
        return
    }
}
```

Deze test gaf het volgende resultaat:

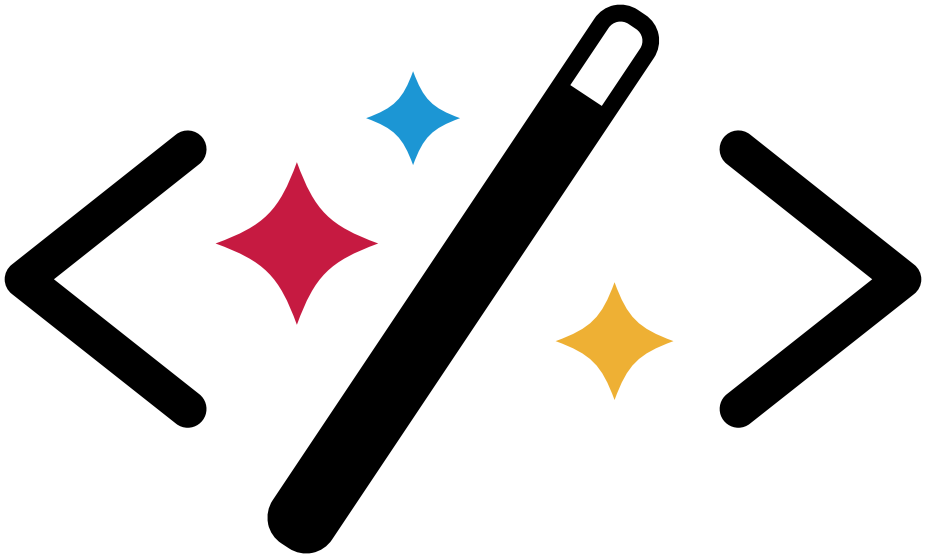
```
Running tool: /usr/local/go/bin/go test -benchmem -run=^$ brocade.be/qtechng/source -bench 1

goos: linux
goarch: amd64
pkg: brocade.be/qtechng/source
BenchmarkQuery79-4      1000000000           0.0247 ns/op          0 B/op          0 allocs/
PASS
ok      brocade.be/qtechng/source   0.289s
```

Besluit

Deze benchmarks werden verschillende keren uitgevoerd (ook met een reboot tussen de verschillende benchmarks). De resultaten waren in grote lijnen consistent.

Format



Formatter en *linters* worden nogal eens verward. Nochtans zijn ze essentieel verschillend: een linter verandert zelf de source code niet maar meldt waar er ergens iets schort. Een formatter gaat de code verfraaien zonder de betekenis te veranderen.

In middens van software ontwikkelaars zijn er 2 dingen waarmee je snel een heilige oorlog kunt ontketenen: editors en formatters. Met de komst van `Go` lijkt hier een abrupt einde aan gekomen: de designers - Turing Award winners - gebruiken een gezagsargument en zeggen: Go code moet er zo uitzien. Meer nog, het instrumentarium bekrachtigt deze visie en formatteert zonder toestemming te vragen.

`QtechNG` volgt deze aanpak, ten minste wat de gespecialiseerde formaten, X-files, D-files, L-files, B-files, I-files en M-files, betreft.

Formatters moeten worden geïntegreerd met editors (zoals `vscode`) en spelen zich dus af op file niveau, op het werkstation van de ontwikkelaars.

De aanpak die ~QtechNG‘ volgt, is steeds dezelfde:

- Eerst wordt het bestand onderworpen aan een linter
- Zijn er problemen, dan stopt het proces!
- Vervolgens worden - indien opportuun - de verschillende onderdelen van de bestanden gesorteerd:
  - In `L-files` worden tekstfragmenten en scopes bij elkaar gebracht. verder wordt de volgorde van het originele bestand gerespecteerd.

  - In `D-files` worden `m4_get`, `m4_set`, `m4_del` en `m4_upd` bij elkaar gebracht en verder wordt de volgorde van het originele bestand gerespecteerd.

- Na het sorteren worden de diverse lijnen en onderdelen van het bestand passend geformatteerd

Linter

Linters zijn een belangrijke onderdeel van *QtechNG*.

Een linter onderzoekt source code en meldt indien er iets schort aan de software. Er zijn zijn essentieel twee soorten berichten:

- De code is onwerkbaar: hij beantwoordt niet aan de vereisten van de specificatie
- De code is werkbaar maar is samengesteld tegen de afspraken in: de stijl van de software past niet.

Brocade werkt met een grote verscheidenheid aan specificaties: behalve aan goed gekende structuren (HTML, JSON, Python, reST, …) zijn er ook minder gekende specificaties zoals M en interne formaten zoals X-files, D-files, L-files, B-files, I-files en M-files.

QtechNG houdt zich met deze laatsten bezig.

In de eerste versie van *QtechNG* worden linters vervaardigd voor D-files, L-files en I-files. Dit is niet toevallig: deze bestanden definiëren *objecten* (`macro`, `inlude` en `lgcode`). De controle gebeurt op drie niveaus.

Op het **eerste niveau** worden de bestanden zelf onderzocht:

- zijn ze voorzien van een overkoepelend commentaarveld ?
- zijn ze in *UTF-8* ?
- Beantwoorden ze aan de passende `PEG`-specificatie ?
- zijn de gegenereerde objecten uniek binnen de file ?

Cruciaal is het afchecken tegen de `PEG`-specificatie. In deze blog heb ik het al gehad over `PEG` in Go en de gegeneerde software met `Pigeon`. Pas indien de bestanden aan de specificatie voldoen, kunnen macros, includes en lgcodes worden geconstrueerd.

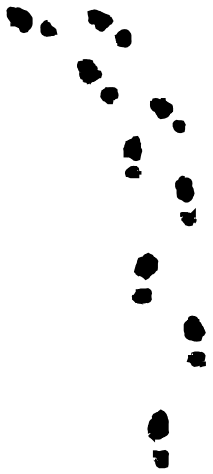
Dan kan het **tweede niveau** van linten beginnen: linten op het niveau van het object. Hier worden allerlei elementen van de diverse objecten onderzocht. Dit zijn dingen die zich niet lenen tot het linten of file niveau (vb.: het onderzoeken van documentatie rekening houdend met standaarddocumentatie)

Het **derde niveau** verlaat het pure file niveau en gaat controles uitvoeren tegenover het *repository*:

- Zijn de objecten ook uniek in het repository ?
- Worden nog in gebruik zijnde objecten geschrapt ?

Het was belangrijk dat deze linters eerst klaar waren: vooraleer er naar *QtechNG* kan worden gemigreerd, moeten de bestaande objectfiles worden gecontroleerd op hun geldigheid. Fouten moeten manueel worden gecorrigeerd. Werk aan de winkel!

Interludium: JSONPath



Het zoeken in tabulaire data hebben we aardig onder de knie: einde jaren '70, begin jaren '80, zagen we de ene query taal na de andere verschijnen. Op het einde van de rit bleef er SQL over. Niet dat er geen valabele alternatieven bestonden! Ik studeerde toen informatica en herinner me maar al te goed dat er nog heel wat andere kandidaten bestonden. Maar in het evenwicht tussen kracht en eenvoud, wist dit IBM geesteskind de gulden middenweg te vinden.

Tabulaire data is zeker belangrijk. Maar het is mijns inziens niet de natuurlijke manier om data te ordenen: hiërarchische structuren lijken veel beter aan te sluiten bij hoe mensen kijken naar data. Een bibliotheek bevat boeken, boeken bevat hoofdstukken met paragrafen en deze bestaan dan weer uit zinnen.

Talen zoals XML gaan voluit voor deze hiërarchieën. De laatste jaren is er een nieuwkomer zich ook gaat toeleggen op dit soort van data:JSON werd immens populair. Dit heeft alles te maken met zijn (ogenschijnlijke) eenvoud. Je kan er niet omheen: je bent veel sneller op weg met JSON dan met XML. De specificatie vult amper het scherm van een monitor.

Echter - het lijkt wel onontkoombaar - JSON verliest stilaan zijn status van 'eenvoud': er bestaan ondertussen 3 standaardisatie initiatieven en er bestaan niet minder dan 25 RFCs met 'JSON' in de titel.

Je merkt het ook aan de tooling rondom JSON en ik heb het dan niet over de faciliteiten ingebouwd in moderne programmeertalen. Zo hebben we JSON

schema, de tegenhanger van XML Schema.

Wat veel moeilijker is bij hiërarchische data dan bij tabulaire data, is het zoeken naar gekwalificeerde informatie. Zo werd voor XML, XPath ontwikkeld en ja, nu is er ook een dergelijk instrument gemaakt voor JSON.

De oorspronkelijk tekst die `JSONPath` introduceerde, kan je vinden bij Stefan Goessner. Dit was zeker niet de enige poging om een XPath-alike te construeren maar Goessner deed het voortreffelijk: een toegankelijke, doch rigoureuze specificatie gekoppeld aan implementaties in de twee belangrijkste programmeertalen op het Web: Javascript en PHP. Beiden trouwens ook in gebruik in Brocade.

Het volgende voorbeeld heb ik geleend bij SMARTBEAR. Samen met `JSONPath` online evaluator, overigens een uitstekende site om je te bekwamen in `JSONPath`.

Stel, je hebt de volgende data:

```
{
  "store": {
    "book": [
      {
        "category": "history",
        "author": "Arnold Joseph Toynbee",
        "title": "A Study of History",
        "price": 5.50
      },
      {
        "category": "poem",
        "author": "Aneirin",
        "title": "Y Gododdin",
        "price": 17.00
      },
      {
        "category": "fiction",
        "author": "James Joyce",
        "title": "Finnegans Wake",
        "isbn": "9788804677628",
        "price": 15.99
      },
      {
        "category": "fiction",
        "author": "Emily Bronte",
        "title": "Wuthering Heights",
        "isbn": "0-486-29256-8",
        "price": 3.30
      }
    ],
    "bicycle": {
```

```
      "color": "purple",
      "price": 25.45
    }
  },
  "expensive": 10
}
```

Deze data vertelt je welke boeken er in de winkel liggen.

Stel je wil enkel de titels hebben van alle boeken.

De selectie met behulp van de volgende JSONpath doet precies dat: (Oefen op `JSONPath` online evaluator)

```
$.store.book[:].title
```

```
[
    "A Study of History",
    "Y Gododdin",
    "Finnegans Wake",
    "Wuthering Heights"
]
```

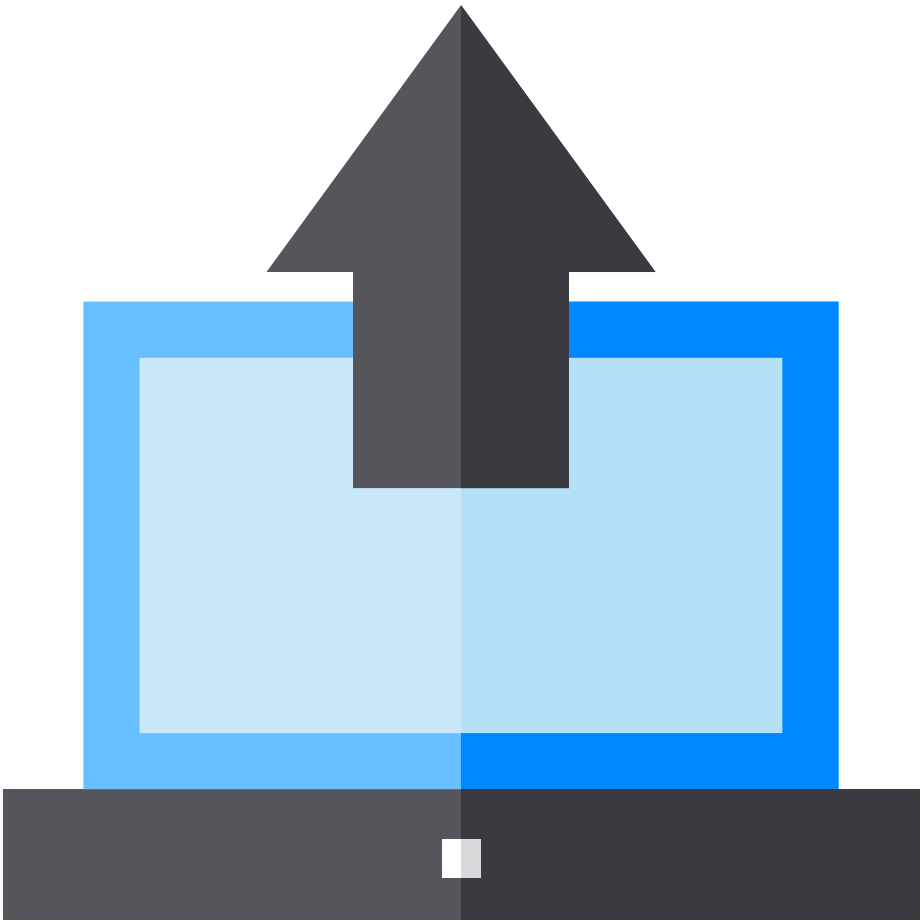
Als je enkel fictie wil, gebruik je:

```
$.store.book[?(@.category == 'fiction')].title
```

```
[
    "Finnegans Wake",
    "Wuthering Heights"
]
```

Ik denk dat je nu wel een goed idee hebt over wat de doelstellingen en de mogelijkheden zijn van `JSONPath`.

Output



Najaar 2006.

Microsoft bracht een nieuwe shell uit, de PowerShell. Nogal wat Unix/Linux afficionados konden enkel een smalend schouderophalen opbrengen. Ik dacht daar toch anders over: Unix/Linux had een krachtig pipe-systeem waarbij toepassingen informatie konden uitwisselen door de output en input streams te ketenen.

Essentieel was deze informatie overdracht echter beperkt tot tekst.

Niet zo in PowerShell: daar konden heuse objecten worden getransfereerd. Het klinkt allemaal wat abstract, maar ik kan je het verschil het beste uitleggen met `docman`. Docman is een Brocade toepassing die blobs (documenten) beheert.

Deze documenten worden gekarakteriseerd door een identificatie die lijkt op een file path, bijvoorbeeld: `/exchange/66fe04/lc.bin`.

Als je de eigenschappen wil weten van dit document gebruik je de `docman`

toepassing (geschreven in *Python*):

```
moto /library/tmp> docman -properties /exchange/66fe04/lc.bin
path=/exchange/66fe04/lc.bin
file=/library/database/docman/exchange/x6/6f/e04lc.bin.rm20200421
size=4053
md5=fcf4aff144fd688333a889951f6cc72b
shorturl=/docman/exchange/66fe04/lc.bin
longurl=https://anet.be/docman/exchange/66fe04/lc.bin
```

Het komt geregeld voor dat je deze informatie wil delen met een andere (niet-python) toepassing. Typisch gebeurt dit via piping of via een tussenliggend bestand.

Het hoeft geen betoog dat dit heel foutgevoelig, weinig flexibel en inefficiënt is.

In PowerShell kan je deze informatie vatten als een object en als dusdanig uitwisselen met andere toepassingen.

In `qtech` ben ik verschillende malen op dit probleem gestoten. Ik maakte de toepassing met het oog op menselijke consumptie (zoals trouwens de meeste Unix toepassingen). Al snel liep ik op tegen allerlei hindernissen. De resultaten van de toepassingen moesten kunnen worden getoond in een `wxPython` toepassing. Aha, dergelijke toepassingen kunnen ook HTML aan en een extra parameter was geboren: `HTML=yes`. Vlijtig herwerkte ik de output van (10-tallen) commando's en sub-commando's zodat ook HTML output mogelijk was. Toen kwamen Emacs, Textadept, Sublime Text en Visual Studio Code. Telkens kon ik aan de slag: zowel in de `qtech` toepassing zelf als in de bewuste applicaties.

Met `qtechng` wil ik het beter doen. Ik kon wel geen gebruik maken van *echte* objecten: we willen immers `qtechng` gebruiken op diverse platformen en samenwerkend met diverse toepassingen.

The Next Best Thing is `JSON`.

`JSON` - goed geformatteerd - is best leesbaar en uitstekend geschikt om met allerlei technologieën te worden verwerkt. Alle commando's en sub-commando's van `qtechng` gaan dus `JSON` produceren (tenminste indien dit opportuun is voor dit commando). Deze `JSON` heeft ook een heel herkenbare structuur.

Een paar voorbeelden die de kracht van deze aanpak illustreren.

Op mijn werkstation:

```
cd /home/rphilips/qtechng/brocade/edit/lab/application
qtechng file add file1.txt file2.txt # maak 2 nieuwe bestanden
{
"host": "rphilips-home",
"time": "2020-08-08T11:01:58+02:00",
"RESULT": [
{
```

```
"arg": "file1.txt",
"version": "9.92",
"qpath": "/lab/application/file1.txt"
},
{
"arg": "file2.txt",
"version": "9.92",
"qpath": "/lab/application/file2.txt"
}
]
}
```

Het JSON object heeft steeds dezelfde structuur: *host*, *time* en *RESULT* komen altijd terug.

Hm, ga je zeggen, dit is toch wel heel erg *verbose*. Dat is zo. Het is immers de bedoeling dat als deze gegevens dienen te worden verwerkt, er *NIET* meer aan `qtechng` dient te worden gesleuteld: de gemakkelijk te verwerken JSON output moet voldoende zijn.

Trouwens …

In een vorige blogpost heb je kunnen kennismaken met `JSONPath`. Deze technologie kan worden gebruikt om de output drastisch te vereenvoudigen. Het goede nieuws is dat er in `qtechng` een `JSONPath` processor is ingebouwd! Deze kan eenvoudig worden geactiveerd door een `jsonpath` flag mee te geven aan het commando.

Voorbeelden:

```
cd /home/rphilips/qtechng/brocade/edit/lab/application
qtechng file add file3.txt file4.txt --jsonpath="$.RESULT"
[
{
"arg": "file3.txt",
"version": "9.92",
"qpath": "/lab/application/file3.txt"
},
{
"arg": "file4.txt",
"version": "9.92",
"qpath": "/lab/application/file4.txt"
}
]
```

of nog korter:

```
cd /home/rphilips/qtechng/brocade/edit/lab/application
qtechng file add file5.txt file6.txt file7.txt --jsonpath="$.RESULT[:].qpath"
[
```

```
"/lab/application/file5.txt",
"/lab/application/file6.txt",
"/lab/application/file7.txt2"
]
```

Een volgende blog post zal het verwerken van fouten onder de loep nemen.