# Korte uitleg: File descriptors en redirection in de shell (en pipes)

De zogenaamde Redirection faciliteit zoals gebruikt in de shell is soms lastig te overzien. Vooral `bash` heeft nogal wat uitbreidingen op de oorspronkelijke syntax, die niet allemaal even makkelijk te volgen zijn. Dit verhaal (want *Korte uitleg*) blijft beperkt tot de belangrijkste zaken, waarmee je echt goed vooruit kunt. Voor we verder gaan nu eerst enige uitleg over File descriptors.

Zoals je al hebt gelezen gaat alle gebruik van hardware via de Kernel, dus ook alle I/O (invoer/uitvoer). Daar zijn weer een heleboel library routines voor met namen als `fopen()`, `fread()`, `fwrite()`, `fclose()`. Omdat een proces natuurlijk tegelijkertijd op meer bestanden bezig kan zijn, moet aan ieder van die routines minstens worden verteld welk bestand bedoeld is. De Kernel heeft voor elk open bestand een tabel met gegevens over dat bestand en zijn toestand (waar staat het bestand, waar zijn we met lezen gebleven, enz., enz.). Deze tabel heet de File Descriptor. File descriptors hebben een unieke identificatie binnen de Kernel, ten eerste de `PID` en daarna een nummer uniek binnen de `PID`. Ieder proces dat wordt gestart krijgt al gelijk van de Kernel drie file descriptors met vastgestelde file descriptor nummers: standard input (`stdin` met nummer `0`), standard output (`stdout` met nummer `1`) en standard error (`stderr` met nummer `2`).

Interaktieve (in de CLI) programma's gebruiken `stdin` (`0`) om te lezen van de terminal, `stdout` (`1`) om te schrijven naar de terminal en `stderr` (`2`) om eventuele waarschuwingen en foutmeldingen op de terminal te zetten. Bij het opstarten van een proces dat met een terminal is verbonden zorgt de shell ervoor dat `stdin`, `stdout` en `stderr` ook inderdaad van en naar de terminal werken.

Maar daar kunnen we verandering in brengen. Als we aan de Kernel konden vertellen dat `stdout` niet naar de terminal moet maar naar een bestand, konden we de uitvoer van een programma zoals `ls` opvangen in een bestand en daar later iets leuks mee doen. En dat kan. We kunnen de Kernel vertellen om de bestemming in zijn file descriptor `1` te wijzigen in de naam van een bestand. En de shell geeft ons de mogelijkheid dat te gebruiken. Belangrijk voor het begrip is dat in de file descriptor dus een veldje is waarin staat waar het heen moet (of waar het vandaan komt). En de inhoud van dat veldje kunnen we veranderen.

De uitgangstoestand is dus zo:

| | |
|---|---|
| 0 | TTY |
| 1 | TTY |
| 2 | TTY |

Willen we nu de `stdout` van `ls` naar het bestand `/tmp/ls-uitvoer` sturen dan doen we:

```
ls -l >/tmp/ls-uitvoer
```

Het ls proces wordt nu opgestart met als tabel:

| | |
|---|---|
| 0 | TTY |
| 1 | `/tmp/ls-uitvoer` |
| 2 | TTY |

En na afloop kunnen we in `/tmp/ls-uitvoer` kijken. Dus bijv.:

```
henk@boven:~/test/bestanden> ls -l verf.jpeg lak.jpeg
ls: kan geen toegang krijgen tot lak.jpeg: Bestand of map bestaat niet
-rw-r--r-- 1 henk wij 157943 10 mei 20:38 verf.jpeg
henk@boven:~/test/bestanden> ls -l verf.jpeg lak.jpeg >/tmp/ls-uitvoer
ls: kan geen toegang krijgen tot lak.jpeg: Bestand of map bestaat niet
henk@boven:~/test/bestanden> cat /tmp/ls-uitvoer
-rw-r--r-- 1 henk wij 157943 10 mei 20:38 verf.jpeg
henk@boven:~/test/bestanden>
```

Inderdaad de uitvoer zit in `/tmp/ls-uitvoer` en we zien ook een foutmelding omdat `lak.jpeg` niet bestaat, die komt via `stderr` en dus via de TTY. Daar kunnen we wat aan doen:

```
henk@boven:~/test/bestanden> ls -l verf.jpeg lak.jpeg >/tmp/ls-uitvoer 2>/tmp/ls-fout
henk@boven:~/test/bestanden> cat /tmp/ls-uitvoer
-rw-r--r-- 1 henk wij 157943 10 mei 20:38 verf.jpeg
henk@boven:~/test/bestanden> cat /tmp/ls-fout
ls: kan geen toegang krijgen tot lak.jpeg: Bestand of map bestaat niet
henk@boven:~/test/bestanden>
```

En onze referentietabel is nu zo:

| | |
|---|---|
| 0 | TTY |
| 1 | `/tmp/ls-uitvoer` |
| 2 | `/tmp/ls-error` |

Nog even over de syntax. Het `>` wijst op redirection van uitvoer. Daarvoor staat het nummer van de file descriptor, dus `1` voor `stdout` en `2` voor `stderr`. Die `1` kun je weglaten.

Hetzelfde geldt voor redirection van invoer, alleen gebruiken we daar de `<`. En

weglaten van het nummer betekent `stdin`. stel je hebt de invoer van een programma voorbereid door met een editor een bestand met invoerregels te maken, dan kun je die aan het programma toevoeren alsof je ze had ingetikt na de programma aanroep:

```
programma <invoer
```

De bijbehorende tabel is:

| | |
|---|---|
| 0 | `invoer` |
| 1 | TTY |
| 2 | TTY |

En nu de bekende:

```
programma >/tmp/uitvoer 2>&1
```

Bijzonder is hier die `2>&1`. We redirecten `stderr` naar de plek waar `stdout` verwijst. Hiervoor is belangrijk te onthouden dat de redirections van links naar rechts worden verwerkt. We starten met:

| | |
|---|---|
| 0 | TTY |
| 1 | TTY |
| 2 | TTY |

dan komt `>/tmp/uitvoer`:

| | |
|---|---|
| 0 | TTY |
| 1 | `/tmp/uitvoer` |
| 2 | TTY |

en nu komt `2>&1`, dus in rij 2 komt hetzelfde als nu in rij 1 staat:

| | |
|---|---|
| 0 | TTY |
| 1 | `/tmp/uitvoer` |
| 2 | `/tmp/uitvoer` |

`stdout` zowel als `stderr` gaan nu naar `/tmp/uitvoer`.

Als je dit begrijpt, begrijp je ook waarom

```
programma 2>&1 1>/tmp/uitvoer
```

iets anders is. Je begrijpt dit als je het maar stap voor stap naspeelt. Start:

| | |
|---|---|
| 0 | TTY |
| 1 | TTY |
| 2 | TTY |

dan komt `2>&1`, dus in rij 2 komt hetzelfde als nu in rij 1 staat:

| | |
|---|---|
| 0 | TTY |
| 1 | TTY |
| 2 | TTY |

er is dus eigenlijk niets veranderd en nu komt `>/tmp/uitvoer`:

| | |
|---|---|
| 0 | TTY |
| 1 | `/tmp/uitvoer` |
| 2 | TTY |

en dus anders dan de vorige keer. Zo'n `n>&m` kopieëert dus de inhoud van het veld van `m` naar het veld van `n`. Verder niets.

Redirection wordt ook vaak gebruikt om uitvoer in "het zwarte gat" te laten verdwijnen. We willen wel een programma aanroepen, maar zijn niet geïntereseerd in zijn `stdout` en `stderr`:

```
programma >/dev/null 2>&1
```

Extra achtergrondinformatie: `/dev/null` is een character device file waar je naar toe kunt schrijven, maar er wordt niets mee gedaan. (Voor device special files zie: *Korte uitleg: Device files (/dev/sda en zo)*).

Behalve `>` kun je ook `>>` gebruiken. Wat is het verschil? Bij `>` wordt het bestand als het nog niet bestaat aangemaakt, als het wel bestaat wordt het

leeggemaakt. bij `>>` wordt het bestand als het nog niet bestaat aangemaakt, als het wel bestaat komt de nieuwe uitvoer er achteraan bij.

Dit wordt een lange *Korte uitleg*.

Je kunt ook de `stdin`, `stdout` en `stderr` van de shell waar je nu in zit redirecten. Dat gaat gek genoeg met het shell build in commando `exec` (dat eigenlijk heel ergens anders voor is).

```
exec >pagina.html
```

en alle uitvoer van alle commando's die daarna komen (ingetikt in een interaktieve shell, of vanuit een script) gaat naar `pagina.html`. Ik gebruik dit zelf in een script dat een HTML pagina opbouwt uit allerlei aanroepen. Daarbij stuitte ik op het volgende. Als de pagina klaar is, wil ik niet dat eventuele volgende uitvoer nog naar `pagina.html` gaat. Ik wil `stdout` dus weer "terugzetten" op wat hij was. Maar wat was hij? Het volgende lost dit op. Uitgangspunt:

| | |
|---|---|
| 0 | TTY |
| 1 | weetniet |
| 2 | TTY |

dan doen we:

```
exec 3>&1 >pagina.html
```

en de tabel wordt eerst:

| | |
|---|---|
| 0 | TTY |
| 1 | weetniet |
| 2 | TTY |
| 3 | weetniet |

en dan:

| | |
|---|---|
| 0 | TTY |
| 1 | `pagina.html` |
| 2 | TTY |

| | |
|---|---|
| 3 | weetniet |

Na afloop doen we:

```
exec >&3
```

dat levert:

| | |
|---|---|
| 0 | TTY |
| 1 | weetniet |
| 2 | TTY |
| 3 | weetniet |

en `stdout` is weer wat hij eerst was, zelfs al weten we niet wat dat was.

We hebben het al gehad over het redirecten van input van een bestand. Als dat bestand niet te groot is kun je besluiten het in het script zelf te zetten als een zogenaamd "here document". Dat ziet er zo uit:

```
programma <<EOF
invoer regel
nog een invoer regel
EOF
```

De shell neemt de regels na het commando en voor de regel met `EOF` apart en stuurt ze via de `stdin` van programma. De kreet `EOF` mag ook iets anders zijn. `EOF` (of wat je hebt gekozen) aan het eind moet geheel alleen op de regel staan. Er zijn wat varianten. Zoek dit op in de `man` pagina van `bash`.

Als laatste: "pipes".

Dit lijkt op redirection, maar het wordt door de shell anders gezien en staat ook heel ergens anders in de man pagina. Een pipe is een koppeling tussen twee programma's waarbij de `stdout` van het eerste programma wordt gekoppeld aan de stdin van het tweede programa. Dit is een zeer krachtig middel en met wat handig gebruik kun je hier al veel mee doen. Zowel gelijk ingetikt als in een script.

```
henk@boven:~/test/bestanden> ps -ef | grep '^henk' | tail
henk     19341 11627  0 17:03 pts/2    00:00:00 vi 5-Redirect
henk     20133  2860  0 17:21 ?        00:00:00 kio_http [kdeinit] http /tmp/ksocket-he
henk     20141  2860  0 17:24 ?        00:00:00 kio_http [kdeinit] http /tmp/ksocket-he
henk     20142  2860  0 17:24 ?        00:00:00 kio_http [kdeinit] http /tmp/ksocket-he
henk     20143  2860  0 17:24 ?        00:00:00 kio_http [kdeinit] http /tmp/ksocket-he
```

```
henk     20144  2860  0 17:24 ?        00:00:00 kio_http [kdeinit] http /tmp/ksocket-he
henk     20145  2860  0 17:24 ?        00:00:00 kio_http [kdeinit] http /tmp/ksocket-he
henk     20342 12879  0 17:25 pts/3    00:00:00 ps -ef
henk     20343 12879  0 17:25 pts/3    00:00:00 grep --color=auto ^henk
henk     20344 12879  0 17:25 pts/3    00:00:00 tail
henk@boven:~/test/bestanden>
```

Het programma `ps` levert een lijst van draaiende processen op `stdout`. Die lijst gaat naar de `stdin` van `grep` waar alleen de regels die beginnen met de tekst `henk` worden doorgelaten naar `stdout`. Die gaan naar `stdin` van `tail`, die alleen het staartje van de lijst laat zien.