Cowboy Lab

Leo Rutten

18/11/2016

Table of Contents

1. Inleiding

Dit is de begeleidende tekst voor het Erlang Cowboy labo. Cowboy is de in Erlang ingebouwde webserver. Deze webserver laat toe om een webinterface in een Erlang applicatie in te bouwen.

In dit labo worden een aantal voorbeelden bekeken en uitgebreid. Al deze voorbeelden hebben als gemeenschappelijk kenmerk dat ze uit meerdere modules bestaan. Daarom worden alle voorbeelden met behulp van buildtools gecompileerd. Hiervoor heb je een commandolijn nodig.

Het doel van dit labo is niet enkel de kennismaking met Cowboy. Er wordt ook uitgelegd hoe je *applications* en *releases* kan maken. Tot nu toe hebben we Erlang modules getest met de `erl` commandolijn. Deze methode is handig om snel één of enkele modules te testen. Om grotere gehelen te bouwen die uit meerdere modules bestaan, is deze werkwijze omslachtig.

Voor we beginnen worden enkele concepten verduidelijkt.

2. Erlang/OTP concepten

Kort samengevat is Erlang/OTP de combinatie van Erlang als taal met OTP als omgeving met een aantal faciliteiten die de uitbating van Erlang applicaties vergemakkelijken. Door meerdere modules

te groeperen tot een applicatie is het gemakkelijker om deze modules als een geheel te behandelen. Je kan in een node meerdere applicaties starten en beheren. Ook de afhankelijkheid tussen de applicaties kan Erlang/OTP voor zijn rekening nemen.

Om applicaties te kunnen bouwen moet je een aantal ontwerppatronen volgen. Om deze patronen gemakkelijker in te voeren heeft men in Erlang behaviours ingevoerd. Een behaviour is te vergelijken met een Java `interface`. Wanneer je een module voorziet van een behaviour ben je verplicht om een aantal functies te implementeren. Een behaviour werkt zoals een contract: een module die een contract aangaat, belooft een bepaalde functionaliteit te voorzien. Deze behaviours zijn er gekomen omdat de Erlang ontwerpers een aantal veel voorkomende ontwerppatronen op een formele manier wilden ondersteunen. Het bestaan van behaviours is een voordeel voor Erlang/OTP, vooral voor het OTP deel. OTP kan er dan vanuit gaan dat bepaalde modules een bepaald contract volgen.

Dit is een overzicht van de meest voorkomende behaviours:

- `gen_server`

  Dit is een contract voor een algemene server. Deze server kan allerlei soorten berichten verwerken en is in staat om door middel van een proces zijn eigen toestand bij te houden. Als je dit contract volgt, hoef je zelf geen loop meer te schrijven.

- `supervisor`

  Dit is een opzichterproces dat als enige taak de bewaking van een ander proces uitvoert. Je hoeft voor dit type contract geen Erlang te schrijven. Het volstaat om de `supervisor` te configuren. Zo moet je onder andere vastleggen wat er moet gebeuren als het te bewaken proces crasht: herstarten of niet. Een supervisor is nuttig in complexe systemen waarin mogelijk nog processen door fouten kunnen crashen.

- `application`

  Dit is een bundeling van een aantal modules tot een applicatie. Erlang/OTP is in staat om applicaties te beheren. Om te weten hoe een applicatie moet gestart en gestopt kunnen worden, moet je dit contract volgen.

Het gebruik van behaviours is niet verplicht. In dit deel van de tekst wordt alleen maar de `appliction` behaviour uitgelegd en toegepast in een voorbeeld. De andere behaviours komen niet aan bod. Applicaties kunnen gebouwd worden zonder de `gen_server` en `supervisor` behaviours maar de `application` behaviour heb je wel altijd nodig.

Om een beter zicht te krijgen hoe software in Erlang/OTP in verschillende delen (modularisering) kan opgedeeld worden, is hier een overzicht van de faciliteiten die Erlang/OTP hiervoor aanbiedt.

- functie

  Een functie is een klein deel van de functionaliteit met een eigen naam.

- `module` (Erlang):

  Een module is een Erlang broncodebestand met `-module()`. Dit kan apart gecompileerd en geladen worden.

- `application` (Erlang/OTP)

  Dit is een bundeling van meerdere modules, met tenminste de `application` behaviour. Overige behaviours zijn niet verplicht. De `application` behaviour is dat wel. Een application kan apart gestart en gestopt worden. Alle modules die deel uit maken van een application worden dan samen gestart en gestopt.

- `release` (Erlang/OTP)

> Dit is een bundeling van een aantal applications samen met de nodige Erlang/OTP runtime tot een release. Een release kan als een aparte node gestart worden. Alle applications erin worden automatisch gestart. En er kan altijd een nieuwe versie van een release gestart worden. Een release kan ook gestart worden op een platform waar geen Erlang/OTP geïnstalleerd is.

Het is duidelijk dat een release het verst gaat. Het is zelfs mogelijk in Erlang/OTP om een nieuwe release te starten die de werking van een oude release overneemt. Dit betekent dat je een nieuwe versie van een softwaresysteem in gebruik kan nemen zonder dat het systeem eerst buiten dienst genomen wordt. Uiteraard is dit maar mogelijk indien de betrokken processen in staat zijn om deze overgang te maken. Door de structuur van Erlang/OTP systemen, processen met een lokale toestand, is dit niet zo moeilijk.

Het bouwen van een release werd oorspronkelijk gedaan met tools die als functies binnen Erlang/OTP gestart moeten worden. Deze werkwijze is omslachtig en wordt door bijna niemand meer toepast. Er zijn ondertussen nieuwe tools zoals `rebar`, `mad` en `erlang.mk`/`relx` ontstaan die het bouwen van releases sterk vereenvoudigd hebben. Wij maken gebruik van de tool `rebar3`; dit is een fork van `rebar`.

Het gebruik van `rebar3` is nieuw voor het academiejaar 2016-2017. Vorig academiejaar werd de combinatie `erlang.mk`/`relx` gebruikt. Deze combinatie werd voornamelijk gebruikt om Cowboy applicaties te bouwen. Nu is gebleken dat `rebar` populairder was. Ondertussen is `rebar` (de ontwikkeling ervan ligt stil) vervangen door `rebar3` en zo is het `rebar3` geworden.

3. Kennismaking met `rebar3`

Om een zicht te krijgen op de structuur van applicaties en releases maken we gebruik van de mogelijkheid die `rebar3` biedt om skeletprojecten te genereren. We beginnen met een `application` project.

3.1. Een `application` project genereren

Zoals eerder uitgelegd is een `application` een bundeling van modules die binnen een release gestart en gestopt kan worden.

Maak nu een map waar je de projecten kan genereren.

```
mkdir -p erlang/rebar3
cd erlang/rebar3
```

Genereer een `application` project.

```
rebar3 new app vb1
```

Ga naar de map en bekijk de bestanden.

```
cd vb1
tree
.
|-- LICENSE
|-- README.md
|-- rebar.config
`-- src
    |-- vb1.app.src
    |-- vb1_app.erl
```

```
    `-- vb1_sup.erl
```

Dit project bevat de volgende bestanden:

3.1.1. `rebar.config`

Dit bestand beschrijft het project voor `rebar3`.

```
{erl_opts, [debug_info]}.
{deps, []}.
```

In deze versie is dit een zeer eenvoudig bestand. Wanneer er afhankelijkheden bijkomen, worden die is de `{deps, []}` bijgevoegd.

3.1.2. `vb1.app.src`

De OTP richtlijnen eisen dat er een projectbeschrijving bestaat die er minstens zo uitziet:

```
{application, vb1,
 [{description, "An OTP application"},
  {vsn, "0.1.0"},
  {registered, []},
  {mod, { vb1_app, []}},
  {applications,
   [kernel,
    stdlib
   ]},
  {env,[]},
  {modules, []},

  {maintainers, []},
  {licenses, []},
  {links, []}
 ]}.
```

De tuple `{application, ...}` beschrijft de applicatie. Na het atoom `application`, dat er altijd zo moet staan, volgt de naam van de applicatie. Hier is dat `vb1`. Je kiest hier best een naam zonder spaties. Er volgt dan een lijst met extra opties. Met `description` wordt een beschrijving vast gelegd. En `vsn` legt een versienummer vast. Dit kan nuttig zijn om later afhankelijkheden tussen applicaties vast te leggen. `modules` bepaalt welke modules deel uitmaken van deze applicatie. Deze lijst mag je nu leeg laten omdat die later automatisch wordt ingevuld.

Bij `registered` kan je de atomen vermelden waarmee bepaalde Pid's geregistreerd zijn.

Met `applications` wordt een lijst van applicaties vastgelegd die zeker moeten gestart worden voordat deze applicatie mag starten. In deze lijst kan je extra applicaties vermelden zoals de `observer`. Denk eraan dat je dan ook alle afhankelijkheden van `observer` moet vermelden. De enige overige applicaties zijn hier `kernel` en `stdlib`. Tot slot wordt er met `mod` vastgelegd welke de module is die het `application` behaviour heeft. Hier is dat de module `vb1_app`. Het is via deze module dat de applicatie gestart wordt.

3.1.3. `vb1_app.erl`

Deze module volgt het `application` behaviour. Dit betekent dat hierin het start- en stoppunt van de applicatie worden vastgelegd.

```
```

```
%%%-------------------------------------------------------------------
%% @doc vb1 public API
%% @end
%%%-------------------------------------------------------------------

-module(vb1_app).

-behaviour(application).

%% Application callbacks
-export([start/2, stop/1]).

%%====================================================================
%% API
%%====================================================================

start(_StartType, _StartArgs) ->
    vb1_sup:start_link().

%%--------------------------------------------------------------------
stop(_State) ->
    ok.

%%====================================================================
%% Internal functions
%%====================================================================
```

We zien in de bovenstaande code de twee functies `start/2` en `stop/1`. Deze functies moet je verplicht implementeren als je de `application` behaviour volgt. Bij het starten van de applicatie wordt `start/2` gestart en bij het stoppen is dat `stop/1`.

Het enige wat er bij start gebeurt, is het starten van de supervisor. In de latere voorbeelden wordt geen supervisor gebruikt en wordt hier meteen de eigen module gestart.

3.1.4. `vb1_sup.erl`

Dit is de supervisor. het is zijn taak om andere processen te starten en te bewaken. Meestal is er binnen de supervisor een strategie om de processen die stoppen te herstarten. De onderstaande code is er een van een supervisor die niet veel doet; er worden geen processen in bewaking gestart.

```
%%%-------------------------------------------------------------------
%% @doc vb1 top level supervisor.
%% @end
%%%-------------------------------------------------------------------

-module(vb1_sup).

-behaviour(supervisor).

%% API
-export([start_link/0]).

%% Supervisor callbacks
-export([init/1]).

-define(SERVER, ?MODULE).

%%====================================================================
```

```
%% API functions
%%====================================================================

start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

%%====================================================================
%% Supervisor callbacks
%%====================================================================

%% Child :: {Id,StartFunc,Restart,Shutdown,Type,Modules}
init([]) ->
    {ok, { {one_for_all, 0, 1}, []} }.

%%====================================================================
%% Internal functions
%%====================================================================
```

3.1.5. Een applicatie compileren en starten

Je kan compileren met `rebar3`. Eerst worden de afhankelijkheden (als die er zijn) opgelost.

```
rebar3 deps
```

In dit voorbeeld zal deze stap niet veel doen omdat er geen afhankelijkheden zijn. Dan volgt de eigenlijke compilatie.

```
rebar3 compile
```

Nu zijn er een aantal bestanden bijgekomen.

```
.
|-- LICENSE
|-- README.md
|-- _build
|   `-- default
|       `-- lib
|           `-- vb1
|               |-- ebin
|               |   |-- vb1.app
|               |   |-- vb1_app.beam
|               |   `-- vb1_sup.beam
|               |-- include -> ../../../../include
|               |-- priv -> ../../../../priv
|               `-- src -> ../../../../src
|-- rebar.config
|-- rebar.lock
`-- src
    |-- vb1.app.src
    |-- vb1_app.erl
    `-- vb1_sup.erl
```

`rebar3` plaatst de `.beam` bestanden in de map `_build/default/lib/vb1/ebin`. Als je dit pad meegeeft met `erl`, kan je de applicatie starten.

```
erl -pa _build/default/lib/vb1/ebin
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]

Eshell V7.1  (abort with ^G)
1> application:start(vb1).
ok
2> application:which_applications().
[{vb1,"An OTP application","0.1.0"},
 {stdlib,"ERTS  CXC 138 10","2.6"},
 {kernel,"ERTS  CXC 138 10","4.1"}]
3>
```

De applicatie is nu wel geladen maar doet niet veel. Het is handiger om een `application` binnen een `release` te plaatsen. `rebar3` kan een `release` skeletproject genereren.

3.2. Een `release` project genereren

Ga eerst terug naar de hogere map.

```
cd ..
```

```
rebar3 new release vb2
```

Ga naar de gegenereerde map en bekijk de bestanden.

```
cd vb2
tree
.
|-- LICENSE
|-- README.md
|-- apps
|   `-- vb2
|       `-- src
|           |-- vb2.app.src
|           |-- vb2_app.erl
|           `-- vb2_sup.erl
|-- config
|   |-- sys.config
|   `-- vm.args
`-- rebar.config
```

Ditmaal staat de broncode voor vb2 `application` in de map `apps/vb2`. Deze bestanden verschillen niet van die van het `vb1` voorbeeld. Met de structuur die hier nu staat, is het mogelijk om een `release` te bouwen. Deze `release` kan dan gestart worden. Dit betekent dat alle applicaties die erin voorkomen of vermeld worden als afhankelijkheid, gestart worden.

De configuratie van de release staat in de map `config`.

3.2.1. `sys.config`

Dit is de configuratie van de release

```
[
  { vb2, []}
].
```

Momenteel staat die enkel een lege configuratie van de `vb2` applicatie.

3.2.2. `vm.args`

De `erl` virtuele machine wordt automatisch gestart bij het het starten van de release. In `vm.args` kunnen de commandolijn argumenten voor `erl` vermeld worden.

```
-sname vb2

-setcookie vb2_cookie

+K true
+A30
```

Er wordt hierboven een short name en een cookie ingesteld. Dit betekent dat de release als gedistribueerde node zal draaien.

3.2.3. De release bouwen en starten

Het bouwen gebeurt zo:

```
rebar3 deps
rebar3 compile
rebar3 release
```

Het starten gebeurt zo:

```
_build/default/rel/vb2/bin/vb2 console
```

Je krijgt een console waar je de release kan verifiëren.

```
(vb2@oker)2> application:which_applications().
[{sasl,"SASL  CXC 138 11","2.6"},
 {vb2,"An OTP application","0.1.0"},
 {stdlib,"ERTS  CXC 138 10","2.6"},
 {kernel,"ERTS  CXC 138 10","4.1"}]
(vb2@oker)3>
```

Je ziet nu dat de applicatie draait.

4. Broncode afhalen

Alle voorbeelden staan in een enkele `git` repository. Je kan de broncode van de Cowboy voorbeelden zo met `git` afhalen:

```
git clone http://eaict.technologiecampusdiepenbeek.be/gitblit/r/erlang/erlangotp
```

Hierdoor krijg je een lokale kopie van de repository. Lokaal wijzigen is mogelijk maar de lokale wijzigingen naar `http://eaict.technologiecampusdiepenbeek.be/gitblit` doorsturen is niet mogelijk.

Ga nu naar de map `erlangotp-voorbeelden` en maak een overzicht van alle bestanden.

```
```

```
cd erlangotp-voorbeelden
tree
.
|-- README.md
`-- otp
    |-- teller
    |   |-- Makefile
    |   |-- apps
    |   |   `-- teller
    |   |       `-- src
    |   |           |-- teller.app.src
    |   |           |-- teller.erl
    |   |           `-- teller_app.erl
    |   |-- config
    |   |   |-- sys.config
    |   |   `-- vm.args
    |   `-- rebar.config
    |-- teller-cowboy
    |   |-- Makefile
    |   |-- apps
    |   |   `-- teller
    |   |       `-- src
    |   |           |-- teller.app.src
    |   |           |-- teller.erl
    |   |           |-- teller_app.erl
    |   |           `-- teller_handler.erl
    |   |-- config
    |   |   |-- sys.config
    |   |   `-- vm.args
    |   `-- rebar.config
    `-- teller-cowboy-template
        |-- Makefile
        |-- apps
        |   `-- teller
        |       |-- priv
        |       |   `-- templates
        |       |       `-- index.dtl
        |       `-- src
        |           |-- teller.app.src
        |           |-- teller.erl
        |           |-- teller_app.erl
        |           `-- teller_handler.erl
        |-- config
        |   |-- sys.config
        |   `-- vm.args
        `-- rebar.config
```

De `README.md` geeft een overzicht van de 3 voorbeelden. Dit zijn de Erlang/OTP voorbeelden uit de introductiecursus. Elk voorbeeld staat in zijn eigen map en wordt met `rebar3` gebouwd.

- `otp/teller/`

  Dit is een tellerapplicatie.

- `otp/teller-cowboy/`

  Dit is de tellerapplicatie uitgebreid met de geïntegreerde Cowboy webserver.

- `otp/teller-cowboy-template/`

Dit is de voorgaande applicatie waarin het genereren van de HTML met een ErlyDTL template verloopt. Dit voorbeeld is een geschikt sjabloon voor alle applicaties waar een ingebouwde webserver noodzakelijk is.

5. De `teller` applicatie

Dit is een eenvoudige applicatie met een teller. Ga naar de map en maak een overzicht van de bestanden.

```
cd teller
tree
.
|-- Makefile
|-- apps
|   `-- teller
|       `-- src
|           |-- teller.app.src
|           |-- teller.erl
|           `-- teller_app.erl
|-- config
|   |-- sys.config
|   `-- vm.args
`-- rebar.config
```

Er is een `Makefile` maar die is niet echt nodig omdat het bouwen met `rebar3` gebeurt. De `Makefile` dient enkel als geheugensteuntje om te weten welke `rebar3` subcommando's er zijn.

Bouw de release.

```
rebar3 deps
rebar3 compile
rebar3 release
```

Nu is de release gebouwd en de `teller` applicatie maakt daar deel van uit. Start nu de applicatie met de gegenereerde shellscript.

```
_build/default/rel/teller/bin/teller console
```

Er verschijnt een `erl` console en hier kan je de applicatie verifiëren.

```
application:which_applications().
whereis(teller).
teller:get().
teller:get().
teller:get().
observer:start().
```

Het voordeel van het bouwen van een release is dat het starten van de applicatie veel vlotter gaat. Een ander voordeel van de release is dat alle applicaties die deel uitmaken van de release ook automatisch gestart worden.

Verlaat de Erlang vm met een dubbele ^C.

```
```

```
rebar3 clean
```

6. De `teller` applicatie uitbreiden met een afhankelijkheid

We gaan een applicatie als extra afhankelijkheid bijvoegen. Pas daarom `rebar.config` aan. De regel met `{lijst, ...}` is nieuw. Hiermee weet `rebar3` dat er een extra afhankelijkheid is die met `git` moet afgehaald worden.

```
{erl_opts, [debug_info]}.

{deps,
   [
      {lijst, {git, "http://eaict.technologiecampusdiepenbeek.be/gitblit/r/erla
   ]
}.

{relx, [{release, { teller, "0.1.0" },
         [teller,
          sasl]},

        {sys_config, "./config/sys.config"},
        {vm_args, "./config/vm.args"},

        {dev_mode, true},
        {include_erts, false},

        {extended_start_script, true}]
}.

{profiles, [{prod, [{relx, [{dev_mode, false},
                            {include_erts, true}]}]
            }]
}.
```

Nog een noodzakelijke aanpassing is de descriptor van de `teller` applicatie. Dit het `teller.app.src` bestand. Hier moet jet het woord `lijst` bijvoegen. Dat is nodig om bij de start van de release ook de `lijst` applicatie te laten starten.

```
{application, teller, [
    {description, "Teller applicatie met lijst!"},
    {vsn, "0.1.0"},
    {modules, []},
    {registered, [teller]},
    {applications, [
        kernel,
        stdlib,
        lijst,
        runtime_tools,
        wx,
        observer
    ]},
    {mod, {teller_app, []}},
    {env, []}
]}.
```

Doe de nodige aanpassingen, bouw en start de release. Ga ook na waar de broncode van de `lijst` applicatie terug te vinden is.

7. Cowboy bijvoegen

Dit is de applicatie waaraan Cowboy is toegevoegd. De mappenstructuur ziet er zo uit:

Ga naar de map en bouw de applicatie.

```
cd teller-cowboy
rebar3 deps
rebar3 compile
rebar3 release
```

En start:

```
_build/default/rel/teller/bin/teller console
```

Bekijk nu de URL `localhost:8080` in een browser. Ga na of de teller verhoogd wordt bij een reload in de browser. Waarom wordt de teller niet verhoogd als je te snel en te vaak op reload klikt?

Leg de applicatie stil met een dubbele ^C, verwijder de release en ga een map hoger.

```
rebar3 clean
```

8. HTML genereren met een template

In dit voorbeeld wordt de HTML met een template gegenereerd. De mappenstructuur ziet er zo uit:

```
.
|-- Makefile
|-- apps
|   `-- teller
|       |-- priv
|       |   `-- templates
|       |       `-- index.dtl
|       `-- src
|           |-- teller.app.src
|           |-- teller.erl
|           |-- teller_app.erl
|           `-- teller_handler.erl
|-- config
|   |-- sys.config
|   `-- vm.args
`-- rebar.config
```

Het templatebestand `index.dtl` staat in de map `apps/teller/priv/templates/`

Ga naar het derde voorbeeld en bouw.

```
cd teller-cowboy-template
make
```

En start:

```
_build/default/rel/teller/bin/teller console
```

Ook deze applicatie kan je in de browser bekijken via de URL `localhost:8080`. En stop dan de applicatie en ga een map hoger.

```
cd ..
```

9. Extra uitbreidingen

In een aantal stappen wordt het laatste voorbeeld uitgebreid met een aantal extra webtechnieken. Het doel is te tonen hoe Cowboy deze technieken ondersteunt en hoe je de Erlang code moet aanpassen om deze technieken mogelijk te maken.

Al deze aanpassingen gebeuren stap voor stap in hetzelfde project. Maak eerst een nieuwe map en kopieer het laatste Cowboy voorbeeld (dat met de template). Dit voorbeeld dient als basis voor de extra uitbreidingen.

```
mkdir teller-oefening
cd teller-oefening
rsync -rav ../teller-cowboy-template/* .
```

9.1. Bootstrap layout bijvoegen

Met Bootstrap http://getbootstrap.com/ kan je de layout van een website verbeteren zonder alle details van CSS te moeten leren. Een bijkomend voordeel van Bootstrap is dat de layout zich automatisch aanpast aan het platform (GSM, laptop of PC).

Neem deze inhoud over voor `apps/teller/priv/templates/index.dtl`:

```
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Teller!</title>

    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.

    <!-- Optional theme -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.
  </head>
  <body>

    <nav class="navbar navbar-inverse">
      <div class="container-fluid">
        <div class="navbar-header">
          <a class="navbar-brand" href="#">Teller</a>
        </div>
        <div>
          <ul class="nav navbar-nav">
            <li class="active"><a href="#">Home</a></li>
            <li><a href="#">Over</a></li>
```

```
            <li><a href="#">Statistiek</a></li>
          </ul>
        </div>
      </div>
    </nav>

    <div class="container">
       <p>De teller is nu {{ waarde }}.</p>
    </div>

    <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min
    <!-- Latest compiled and minified JavaScript -->
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.m
  </body>
</html>
```

Dit is link naar deze template: index.dtl [https://gist.github.com/lrutten/8c41c69446bc2d28df736d6430275bd1].

Deze template verschilt op een aantal punten van de vorige versie.

- In de `<head>` staan een twee links naar de Bootstrap CSS stylesheets.
- Op het einde van de `body` staan twee links naar Javascript bestanden: één voor JQuery en één voor Bootstrap zelf.
- Heel de navigatiebalk is bijgevoegd, dat zijn de tags tussen `<nav>` en `</nav>`. Hoe de navigatiebalk opgebouwd wordt, kan je in de Bootstrap documentatie terugvinden. De balk bestaat uit de naam van de applicatie `Teller` en de menupunten `Home`,`Over` en `Statistiek`. De URL's voor deze menupunten zijn voorlopig nog niet ingevuld.
- Er worden een aantal CSS klassen gebruikt. Zo bepaalt de klasse `navbar-inverse` in de `<nav>` tag het uitzicht van de navigatiebalk: witte letters op een donkere achtergrond. Als je dat wilt omkeren, gebruik dan `navbar-default` in plaats van `navbar-inverse`.

Uitleg over het maken van een navigatiebalk vind je hier:

- http://www.w3schools.com/bootstrap/bootstrap_case_navigation.asp

Nu kan je deze aanpassing testen.

```
rebar3 deps
rebar3 compile
rebar3 release
_build/default/rel/teller/bin/teller console
```

Door de parameter `console` te vervangen door `foreground` blijft de console weg; je kan de applicatie stoppen met een enkele ^C.

9.2. Statische bestanden afhandelen

De enige pagina die tot nu toe door de webapp wordt gegenereerd is een dynamische pagina, dit wil zeggen dat de inhoud dynamisch door Erlang en de template worden bepaald. Het is ook mogelijk om een handler te voorzien die statische pagina's en vaste bronnen zoals figuren kan afhandelen. Deze handler is de module `cowboy_static` die in Cowboy reeds bestaat. Deze modulenaam moet je vermelden in een nieuw URL-patroon.

Zorg ervoor dat de toekenning aan de `Dispatch` variabele in `teller_app.erl` er zo uitziet:

```
   Dispatch = cowboy_router:compile(
   [
      %% {URIHost, list({URIPath, Handler, Opts})}
      {'_', % host
         [
            % static information
            {
               "/res/[...]",  % path
               cowboy_static, % handler
               {              % options
                  priv_dir, teller, "",
                  [{mimetypes, cow_mimetypes, all}]
               }
            },
            % main page
            {'_', teller_handler, []}
         ]
      }
   ]),
```

Het patroon dat met de commentaar `% static information` start, is nieuw. De tuple die dit patroon beschrijft, bestaat uit 3 elementen:

- Het pad `"/res/[...]"` geeft aan dat alle URL's die met `res/` starten door deze handler worden verwerkt.
- `cowboy_static` is de naam van de handler.
- De opties bestaan uit:
  - `priv_dir` geeft aan dat de inhoud van een directory afgehandeld wordt.
  - `teller` is de naam van de applicatie.
  - `[{mimetypes, cow_mimetypes, all}]` zorgt ervoor dat elk bestand automatisch van een correct mimetype voorzien wordt. Hierdoor kan de browser bepalen op welke wijze het bestand moet weergegeven worden.

Maak een nieuwe directory waarin je een figuur plaatst.

```
mkdir -p priv/img
```

Neem de figuur flag_be.svg en plaats die in `apps/teller/priv/img`.

```
<div class="container">
   <p>De teller is nu {{ waarde }}.</p>
   <img src="res/img/flag_be.svg"/>
</div>
```

Bij het testen zal je de figuur zien.

9.3. Afhandeling van knoppen

In deze stap is het bedoeling om een knop bij te voegen waarmee je de teller kan resetten. Deze knop moet je in een formulier plaatsen en dit formulier zal met de POST methode de `/reset` URL doorsturen.

Voeg dit fragment bij in `apps/teller/priv/template/index.dtl`.

```
```

```
<form role="form" method="POST" action="reset">
   <button type="submit" class="btn btn-default">reset</button>
</form>
```

Je ziet dat bij een submit de `POST` methode gebruikt wordt en dat de URL met het `action` attribuut wordt vastgelegd als `reset`. De knop wordt met de `<button>` tag gemaakt en deze krijgt `reset` als tekst.

Omdat voor de afhandeling een aparte URL `/reset` wordt gebruikt, is er een extra patroon met een corresponderende handler nodig. De patronen in `teller_app.erl` zien er nu zo uit:

```
   Dispatch = cowboy_router:compile(
   [
      %% {URIHost, list({URIPath, Handler, Opts})}
      {'_', % host
         [
            % static information
            {
               "/res/[...]",  % path
               cowboy_static, % handler
               {              % options
                  priv_dir, teller, "",
                  [{mimetypes, cow_mimetypes, all}]
               }
            },
            % reset POST action
            {"/reset", reset_handler, []},
            % main page
            {'_', teller_handler, []}
         ]
      }
   ]),
```

Je ziet dat de URL `/reset` door de `reset_handler` module wordt verwerkt. Maak in de `src/` map een nieuw bestand `reset_handler.erl` en geef het deze inhoud:

```
-module(reset_handler).
-behavior(cowboy_http_handler).

-export([init/3]).
-export([handle/2]).
-export([terminate/3]).

init(_Type, Req, _Opts) ->
   {ok, Req, undefined_state}.

handle(Req, State) ->
   io:format("reset requestobject ~p~n", [Req]),
   io:format("      state ~p~n", [State]),

   teller:reset(),

   % Antwoord met een redirect
   Req2 = cowboy_req:reply(302, [{<<"Location">>, <<"/">>}],
       <<"Redirecting with Header">>, Req),

   io:format("      requestobject2 ~p~n", [Req2]),
```

```
   {ok, Req2, State}.

terminate(_Reason, _Req, _State) ->
    ok.
```

De `handle/2` functie zorgt ervoor dat de teller nul gemaakt wordt met `teller:reset/0`. Het antwoord dat teruggegeven wordt, is een redirect. Het wordt met `cowboy_req:reply/4` opgebouwd. De belangrijkste parameters zijn de returncode 302 en de URL `/` waarheen de browser moet springen.

De notatie [{<<"Location">>, <<"/">>}] noemt men een property list. Het is een lijst van key/value tuples. Deze constructie komt regelmatig voor om parameterlijsten van variabele lengte kunnen maken.

De `io:format/2` oproepen zijn niet echt nodig maar je kan ermee te weten komen wat de waarde van de `Req` parameter is.

De `teller.erl` module moet ook aangepast worden; de resetfunctionaliteit moet bijgevoegd worden. De `reset/0` functie moet je exporteren.

```
reset() ->
   teller ! {reset},
   ok.
```

En in de loop moet je de verwerking van het `{reset}` bericht bijvoegen:

```
   {reset} ->
      loop(0);
```

Na al deze wijzigingen kan je testen.

9.4. Extra pagina's

Momenteel heeft de navigatiebalk de tabs *Home*, *Over* en *Statistiek*. Alleen bij de eerste *Home* hoort een pagina. Het doel van deze stap is het bijvoegen van 2 extra pagina's bij de tabs *Over* en *Statistiek*.

Elke pagina wordt gegenereerd met een `.dtl` bestand. Voor de eerste pagina is dat `index.dtl`. Voor de twee overige pagina's is er nog geen template. Kopieer daarom `index.dtl` naar `over.dtl` en `statistiek.dtl`.

Pas in elk van de nieuwe bestanden de titel aan. Je moet ook de actieve tab wijzigen. Voor de `over.dtl` wordt de navigatiebalk:

```
<nav class="navbar navbar-inverse">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" href="#">Teller</a>
    </div>
    <div>
      <ul class="nav navbar-nav">
        <li><a href="/">Home</a></li>
        <li class="active"><a href="#">Over</a></li>
        <li><a href="/statistiek">Statistiek</a></li>
      </ul>
    </div>
  </div>
</nav>
```

Je ziet dat in het bovenstaande fragment de klasse `class="active"` verplaatst is. Maak ook de container zo goed als leeg:

```
<div class="container">
   <h1>Over</h1>
</div>
```

Doe dezelfde aanpassingen bij `statistiek.dtl`. En uiteraard moet je in de drie templates de URL's in de `href` attributen aanpassen: `/`, `/over` en `/statistiek`.

Door de nieuwe templates is het nu mogelijk om drie verschillende pagina's weer te geven. Elke pagina krijgt zijn eigen URL. Nu moet de routing aangepast worden zodat elke URL leidt tot de weergave van de corresponderende template. De routing staat in `teller_app.erl`. Dit zijn de twee routes die je moet bijvoegen.

```
   % extra pagina's
   {"/over",       over_handler, []},
   {"/statistiek", statistiek_handler, []},
```

Elke pagina wordt afgehandeld door een eigen handler. Kopieer daarom `teller_handler.erl` naar `over_handler.erl` en `statistiek_handler.erl`. De laatste doet niet meer dan de corresponderende template terug te geven.

```
-module(statistiek_handler).
-behavior(cowboy_http_handler).

-export([init/3]).
-export([handle/2]).
-export([terminate/3]).

init(_Type, Req, _Opts) ->
   {ok, Req, undefined_state}.

handle(Req, State) ->
   {ok, Body} = statistiek_dtl:render([]),
   Headers = [{<<"content-type">>, <<"text/html">>}],
   {ok, Req2} = cowboy_req:reply(200, Headers, Body, Req),
   {ok, Req2, State}.

terminate(_Reason, _Req, _State) ->
    ok.
```

De bovenstaande handler geeft geen bindings mee aan de template, daarom zie je een lege lijst bij de oproep van `render/1`:

```
   {ok, Body} = statistiek_dtl:render([]),
```

In `over_handler.erl` stoppen we extra functionaliteit zodat we gegevens uit het request object kunnen tonen. Zorg ervoor dat de `handle/2` functie er zo uitziet:

```
handle(Req, State) ->
   {Url, _}    = cowboy_req:url(Req),
   {Method, _} = cowboy_req:method(Req),
   {Host, _}   = cowboy_req:host(Req),
   {Port, _}   = cowboy_req:port(Req),
```

```
{Path, _}   = cowboy_req:path(Req),
{Peer, _}   = cowboy_req:peer(Req),

io:format("Method ~p~n", [Method]),

ReqProps =
[
   {<<"url">>,    Url},
   {<<"method">>, Method},
   {<<"host">>,   Host},
   {<<"port">>,   Port},
   {<<"path">>,   Path},
   {<<"peer">>,   Peer}
],
{ok, Body} = over_dtl:render([{<<"req">>, ReqProps}]),
Headers = [{<<"content-type">>, <<"text/html">>}],
{ok, Req2} = cowboy_req:reply(200, Headers, Body, Req),
{ok, Req2, State}.
```

De eerste parameter `Req` van de `handle/2` functie is request object. Hierin vind je allerlei gegevens die met de HTTP aanvraag te maken hebben. Met een aantal functies uit de `cowboy_req` module kan je deze gegevens opvragen.

Als deze functies blijken een tuple terug te geven waarin telkens het eerste element de gevraagde waarde is, vandaar de patroonherkenning bij het resultaat. De `io:format/2` staat er alleen maar voor de test, achteraf mag je die schrappen.

Met alle gegevens die opgevraagd zijn, wordt de property list `ReqProps` samengesteld. Voor elk gegeven wordt een key/value tuple aangemaakt. De key is telkens een binaire string. Deze property list wordt meegegeven aan `render/1` als waarde voor de ErlyDTL variabele `req`.

In de template kan je op een elegante manier de gegevens ophalen. Voeg dit fragment bij in `<div class="container">` in `over.dtl`:

```
<h2>Request</h2>
<ul>
  <li>Method: {{ req.method }}</li>
  <li>Url: {{ req.url }}</li>
  <li>Host: {{ req.host }}</li>
  <li>Port: {{ req.port }}</li>
  <li>Path: {{ req.path }}</li>
  <li>Peer: {{ req.peer }}</li>
</ul>
```

Het ophalen van de gegevens in ErlyDTL gebeurt met uitdrukkingen die tussen `{{` en `}}` geschreven worden. Je ziet dat hier de puntnotatie wordt gebruikt alsof `req` een object is en `method`, `url`, `host`, `port`, `path` en `peer`. De voorwaarde om dit te doen werken is wel dat je vanuit Erlang een property list moet doorgeven.

Dit zijn alle aanpassingen, nu kan je testen.

9.5. Tabellen weergeven

In deze stap willen we een tabel met een variabele lengte tonen. Om dit doel te bereiken wordt eerst de module `teller.erl`, die het model van deze applicatie voorstelt, aangepast. De eenvoudigste manier om een tabel te genereren, is gewoon in een lijst bijhouden wanneer de `get` aanvraag ontvangen wordt. Bij elke `get` aanvraag onthouden we het tijdstip en de stand van de teller. We houden een Erlang list bij als toestand in het `teller` proces. Bij elke aanvraag voegen we één element

vooraan in de lijst bij. De lijst die zo ontstaat is misschien wel niet in de juiste volgorde: de jongste aanvragen staan eerst maar in Erlang is vooraan bijvoegen in lijsten veel gemakkelijker. We kunnen achteraf nog altijd de lijst omkeren met `lists:reverse` als dat nodig zou blijken.

Dit is de nieuwe `teller.erl` module.

```
-module(teller).

-export([start/0, stop/0]).

-export([init/0, loop/2, get/0, reset/0, getlist/0, format_utc_timestamp/0]).

start() ->
   Pid = spawn(?MODULE, init, []),
   register(teller, Pid),
   {ok, Pid}.

stop() ->
   teller ! {stop}.

init() ->
   loop(0, []),
   {ok, []}.

get() ->
   teller ! {get, self()},
   receive
      Result -> Result
   end.

reset() ->
   teller ! {reset},
   ok.

getlist() ->
   teller ! {getlist, self()},
   receive
      Result -> Result
   end.

loop(Teller, List) ->
   receive
      {get, From} ->
         From ! Teller,
         List2 = [{format_utc_timestamp(), Teller} | List],
         loop(Teller, List2);
      {reset} ->
         loop(0, List);
      {getlist, From} ->
         From ! List,
         loop(Teller, List);
      {stop} ->
         ok;
      _ ->
         loop(Teller, List)
   after
      1000 ->
```

```
        loop(Teller + 1, List)
   end.

format_utc_timestamp() ->
   TS = {_,_,Micro} = os:timestamp(),
   {{Year,Month,Day},{Hour,Minute,Second}} =
    calendar:now_to_universal_time(TS),
   Mstr = element(Month,{"Jan","Feb","Mar","Apr","May","Jun","Jul",
             "Aug","Sep","Oct","Nov","Dec"}),
   list_to_binary(io_lib:format("~2w ~s ~4w ~2w:~2..0w:~2..0w.~6..0w",
          [Day,Mstr,Year,Hour,Minute,Second,Micro])).
```

Je ziet meteen dat de `loop/1` functie nu `loop/2` geworden, de extra parameter is nu de lijst die wordt bijgehouden. In `init/0` wordt een lege lijst als startwaarde meegegeven. In alle oproepen van `loop/2` wordt een ongewijzigde of een uitgebreide lijst meegegeven. Bij de verwerking van het `{get, From}` bericht wordt de lijst uitgebreid met een `{Timestamp, Teller}` tuple.

De functie om een timestamp om te zetten naar een string komt van deze documentatie.

- http://www.erlang.org/doc/man/os.html#timestamp-0

In deze functie wordt niet `now/0` maar wel `os:timestamp/0` gebruikt. Beide functies werken vergelijkbaar en zijn eigenlijk uitwisselbaar. Ze geven elk de tuple `{Megaseconds, Seconds, Microseconds}` terug. De enige aanpassing van de overgenomen functie is de omzetting van string als list naar string als binair. Dit was nodig omdat anders de timestamp in de browser niet leesbaar was.

Met de `getlist/0` functie kan de lijst opgevraagd worden. Deze functie wordt nu gebruikt in `statistiek_handler.erl`.

```
handle(Req, State) ->
   List = teller:getlist(),
   io:format("List ~p~n", [List]),

   {ok, Body} = statistiek_dtl:render([{"list", List}]),
   Headers = [{<<"content-type">>, <<"text/html">>}],
   {ok, Req2} = cowboy_req:reply(200, Headers, Body, Req),
   {ok, Req2, State}.
```

De verkregen lijst staat nu in de variabele `List` en wordt ongewijzigd doorgegeven aan de `statistiek.dtl` template met de naam `list`. De tabel wordt als volgt gegenereerd:

```
 <div class="container">
   <h1>Statistiek</h1>

   <h2>Lijst</h2>

   <table class="table table-striped table-bordered">
     <thead>
       <tr>
         <td>Tijdstip</td>
         <td>Teller</td>
       </tr>
     </thead>
     <tbody>
   {% for stamp in list %}
      <tr><td>{{ stamp.1 }}</td><td> {{ stamp.2 }}</td></tr>
   {% endfor %}
     </tbody>
```

```
   </table>
 </div>
```

Vermits de tabel een variabele lengte heeft, is er een herhaling nodig om doorheen alle elementen van de lijst te lopen. ErlyDTL voorziet een for-herhaling die hier gebruikt wordt. De schrijfwijze lijkt op die van Java en C++ voor het doorlopen van lijsten. De variabele `stamp` is de lusteller en zal telkens een tuple als waarde hebben. Je kan elk element van de tuple ophalen met de puntnotatie.

De `table` tag heeft de nodige klassen zodat Bootstrap voor een aangepaste layout kan zorgen. Een tabelvoorbeeld met Bootstrap is hier terug te vinden:

- http://www.w3schools.com/bootstrap/default.asp

Het voorbeeld is nu klaar om getest te worden.

De ErlyDTL Wiki staat hier:

- https://github.com/erlydtl/erlydtl/wiki

Zotonic is een CMS in Erlang is geschreven en die ook gebruik maakt van ErlyDTL. Zij hebben wel een aantal uitbreidingen gemaakt die niet bestaan in de ErlyDTL die wij gebruiken. Toch is deze documentatie nuttig.

- http://zotonic.com/docs/latest/ref/tags/index.html

9.6. Parameters via de URL

In deze stap maken we een uitbreiding die elke timestamp in de lijst een eigen webpagina geeft. Hiervoor maken we gebruik van de mogelijkheid om variabele delen in een URL op te vangen met een routingvariabele. Deze techniek komt ook voor bij REST en leidt tot duidelijke URL's; je hoeft immers geen extra parameters met de URL mee te geven. Alle noodzakelijke gegevens zijn in de URL verwerkt.

```
/universiteit/uhasselt/student/567
/universiteit/kuleuven/student/4512
/universiteit/ucc/student/1814
```

Deze drie URL's geven telkens de pagina van de betrokken student. In de routing moet je het patroon dan zo schrijven:

```
/universiteit/:univname/student/:studentname
```

In de handler kan je dan de waarde van de variabelen `univname` en `studentname` opvragen.

De eerste aanpassing is de `statistiek.dtl` template. Hier geven we de tabel een extra kolom. In deze kolom plaatsen we het volgnummer van de timestamp.

```
<table class="table table-striped table-bordered">
  <thead>
    <tr>
      <td>Nummer</td>
      <td>Tijdstip</td>
      <td>Teller</td>
    </tr>
  </thead>
  <tbody>
{% for stamp in list %}
   <tr>
```

```
    <td><a href="/timestamp/{{ forloop.counter }}" class="btn btn-default">{{
    <td>{{ stamp.1 }}</td>
    <td>{{ stamp.2 }}</td>
  </tr>
{% endfor %}
 </tbody>
</table>
```

Het volgnummer kan je verkrijgen met `{{ forloop.counter }}`. Dit is een in ErlyDTL ingebouwde variabele die de lusteller van de for-herhaling bijhoudt. Dit volgnummer gebruiken we tweemaal: éénmaal als waarde in de URL en éénmaal als tekst in de knop. De `<a>` tag krijgt het uitzicht van een knop; Bootstrap zorgt hiervoor. Elke timestamp krijgt nu zijn eigen URL.

```
/timestamp/1
/timestamp/2
/timestamp/3
```

Als je op de knop klikt, ga je naar de pagina die met de betrokken timestamp overeen komt. Hiervoor is extra routing noodzakelijk. Voeg dit routingpatroon bij in `teller_app.erl`:

```
  % timestamp
  {"/timestamp/:nr", timestamp_handler, []},
```

Het nummer van de timestamp wordt opgevangen in de variabele `nr` en die heb je nodig om de juiste timestamp in zijn eigen pagina te kunnen tonen. De `/timestamp/:nr` URL heeft zijn eigen handler: `timestamp_handler.erl`. Maak een kopie van één van de bestaande handlers naar deze nieuwe handler en zorg ervoor dat de afhandeling er zo uitziet:

```
handle(Req, State) ->
  % geef alle variabelen uit de URL weer
  {Bindings, _} = cowboy_req:bindings(Req),
  io:format("Bindings ~p~n", [Bindings]),

  % haal de variabele nr op
  % dit is een binaire string
  {NrStr, _} = cowboy_req:binding(nr, Req),
  io:format("NrStr ~p~n", [NrStr]),

  % zet de binaire string om naar list string
  % en dan naar integer
  {Nr, _} = string:to_integer(binary_to_list(NrStr)),
  io:format("Nr ~p~n", [Nr]),

  List = teller:getlist(),
  io:format("List ~p~n", [List]),

  El = lists:nth(Nr , List),
  io:format("El ~p~n", [El]),

  {ok, Body} = timestamp_dtl:render([{"timestamp", El}]),
  Headers = [{<<"content-type">>, <<"text/html">>}],
  {ok, Req2} = cowboy_req:reply(200, Headers, Body, Req),
  {ok, Req2, State}.
```

De eerste oproep van `cowboy_req:bindings/1` is niet nodig. Dit is alleen maar om te testen of de variabele `nr` voorkomt in de bindings.

De tweede oproep, die met `cowboy_req:binding/2` is wel nodig. Hiermee haal je de waarde op van de variabele `nr`. Uit testen bleek dat dit een binaire string oplevert. Deze wordt eerst omgezet naar een string als list en daarna wordt dit omgezet naar een integer. Zoals je ziet, wordt er niets ondernomen om na te gaan of de string wel effectief naar integer kan omgezet worden. Deze controle ontbreekt.

Daarna wordt de lijst van timestamps opgevraagd en wordt het n-de element eruit gehaald. Ook hier is geen controle op het nummer. Het element dat zo verkregen wordt, wordt aan de template doorgegeven.

In de afhandeling van `getlist/0` in `teller.erl` moet je nog `lists:reverse/1` bijvoegen om de terug te geven lijst om te keren.

De timestamp kan je zo weergeven:

```
<div class="container">
  <h1>Timestamp</h1>

  <table class="table table-striped table-bordered">
    <thead>
      <tr>
        <td>Tijdstip</td>
        <td>Teller</td>
      </tr>
    </thead>
    <tbody>
     <tr>
       <td>{{ timestamp.1 }}</td>
       <td>{{ timestamp.2 }}</td>
     </tr>
    </tbody>
  </table>
</div>
```

Opnieuw kan er nu getest worden.

10. Besluit

Na al deze kleine voorbeelden kan je inschatten wat de mogelijkheden zijn die Cowboy biedt. Een eerste vaststelling is dat Cowboy gebruiken zonder ErlyDTL te omslachtig is. ErlyDTL is dan wel een losstaand project, toch is de combinatie Cowboy/ErlyDTL handig omdat ze complementair zijn en omdat de compilatie met `rebar3` ook de template compilatie voor zijn rekening neemt.

Een tweede vaststelling is dat Cowboy uitstekend werk verricht.

Betekent dit dan dat Cowboy de beste oplossing is waarmee je alle webapps in de toekomst zal maken? Helemaal niet. Andere frameworks in andere programmeertalen zullen ook valabele oplossingen zijn. Trouwens de gelijkenissen tussen Cowboy en de rest is groter dan je zou denken. Cowboy is tamelijk recent en heeft heel wat kenmerken die van de andere frameworks zijn overgenomen. Het was dus nog niet zo slecht om Cowboy te leren; zo ga je veel gemakkelijker de andere webframeworks begrijpen.

Als je liever geen gebruik maakt van Cowboy, zijn er enkele redenen om het toch te doen:

- Erlang is de gekozen taal.

  Het kan zijn dat door de aard van een op te lossen probleem Erlang de gekozen taal is. Als je deze applicatie een webinterface wil geven, is Cowboy de beste oplossing.

- Hoog interactieve webapplicaties met een eenvoudig model

Indien er veel parallelle aanvragen verwerkt moeten kunnen worden, die ook nog eens onderling een grote interactie hebben, dan is Erlang de aangewezen taal en is Cowboy bijgevolg de oplossing voor de webinterface.

- Gedistribueerde oplossingen of schaalbare oplossingen

  Dat Erlang hier goed scoort, mag geen verrassing zijn.

Als laatste bedenking moet nog vermeld worden dat deze labotekst ook uitlegt hoe je *applicaties* en *releases* maakt. Met deze kennis kan je beter inschatten hoe Erlang kan ingezet worden voor (server)applicaties.