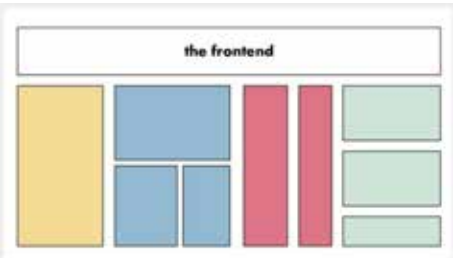
Micro Frontends

The what, the why and the how

In de afgelopen jaren zijn microservices enorm in populariteit gestegen. Veel organisaties gebruiken deze bouwstijl om de beperkingen van grote, monolithische backends te vermijden. Hoewel hier veel over geschreven is, blijven veel bedrijven worstelen met “monolithische frontends”.

In dit artikel zullen we een trend beschrijven die ‘frontend monolieten’ opsplitst in veel kleinere, beter beheersbare stukken. En hoe deze architectuur de effectiviteit en efficiëntie kan vergroten over teams heen. In figuur 1 wordt een applicatie getoond waarbij de frontend bestaat uit een monoliet en de backend bestaat uit meerdere microservices.



Figuur 1.

Wat zijn micro-frontends?

De definitie van micro-frontends is: *The idea of micro frontends is to extend the concepts of microservices to the frontend world.*
Het basisidee van micro-frontends is om je frontend op te splitsen in een reeks onafhankelijk inzetbare en los samenwerkende frontend-applicaties (micro-frontends genoemd). Deze micro-frontends worden vervolgens samengevoegd/gebundeld om één frontend-applicatie te creëren (zie figuur 2). Dit bundelen van micro-frontends wordt behandeld in paragraaf ‘Integration approaches micro frontends’.
De vraag is: Hoe splits je de frontend-applicaties?
Je kan 1 micro-frontend per pagina tonen en deze middels hyperlinks verbinden. Ook is het mogelijk om meerdere micro-frontends te tonen op een pagina (zie figuur 2).

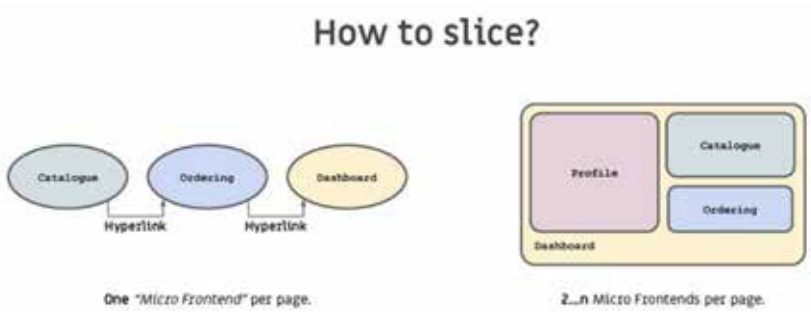
Om het ontwikkelen van een applicatie te versnellen, is het praktisch om de micro-frontends te beschouwen als afzonderlijke verticale segmenten (ook wel “verticals” genoemd) van een webapp. Elke “vertical” is verantwoordelijk voor een enkel bedrijfsdomein/*use case* zoals “Profile”, “Catalog”, “Ordering”. Het heeft zijn eigen presentatielaag, servicelaag (microservice), persistencylaag en een aparte database. Vanuit het ontwikkelingsperspectief wordt elke vertical door één team geïmplementeerd.

Waarom versnelt dit het ontwikkelproces? Ieder vertical team is gefocust op een bedrijfsdomein en hoeft daardoor minder af te stemmen met andere teams. Er wordt het liefst geen code gedeeld tussen de verschillende verticals. Dit is bevorderlijk voor de snelheid van het ontwikkelproces. Voor de eenvoud richten we ons in dit artikel uitsluitend op de presentatielaag van een vertical.

Voorbeeld applicatie

Hieronder beschrijf ik een voorbeeld applicatie, die gebruikmaakt van micro-frontends. Stel je een website voor waar klanten eten kunnen bestellen om te laten bezorgen.

Xxxxxx is een xxxxxxx xxxxx xxxxx xxxx xxx xxxx



Figuur 2.

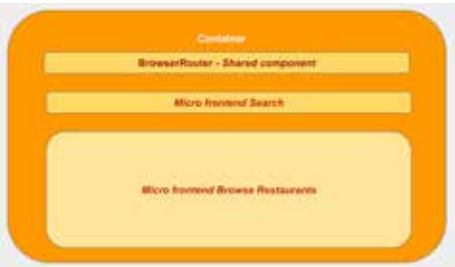
Als eerste heb je een landing page waar klanten kunnen zoeken en filteren naar restaurants. Hiervoor wordt deze micro frontend gebruikt (zie figuur 4): micro-frontend-browse-restaurants.

Elk restaurant heeft vervolgens een eigen pagina waarop de menu-items worden weergegeven en de klant kan kiezen wat hij of zij wil eten (zie figuur 3). Hiervoor wordt micro-frontend-order-food, gebruikt.
Als laatste hebben de klanten een profielpagina waarop ze hun bestelgeschiedenis kunnen zien, de bestelling kunnen volgen en hun betalingsopties kunnen aanpassen. Hiervoor wordt de micro-frontend-user-profile gebruikt.
Deze voorbeeld applicatie wordt gebruikt in de rest van het artikel.

Architectuur

De architectuur van deze applicatie is als volgt:

- Er worden meerdere micro-frontends getoond per pagina.
- Er wordt een container applicatie ingezet als 'main entry point' (zie figuur 4). Deze zorgt o.a. voor het volgende:
- Routering van requests en het aggregeren van responses afkomstig van de backend.
- Regelt cross-cutting concerns zoals authenticatie, autorisatie, logging, caching en navigatie.
- Brengt de verschillende (gedistribueerde) micro-frontends samen op de pagina en bepaalt welke micro-frontend waar en wanneer gerenderd moet worden.

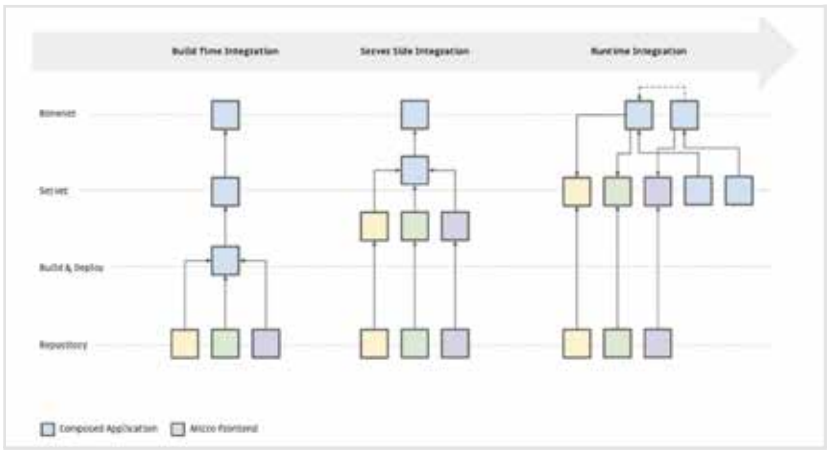


Figuur 4.

Integration approaches micro-frontends

Met 'integration approaches' wordt bedoeld: Hoe de micro-frontends te 'bundelen' in de frontend. In onderstaande afbeelding figuur 5 worden drie 'integration approaches' opgesomd.

Figuur 3.



Figuur 5.

Build time integration

Voor Build time integration publiceren we elke afzonderlijke micro-frontend als een package. At Build time worden deze aparte micro-frontends gebundeld middels een package.json van de container applicatie.

Voor Build time integration wordt gebruikgemaakt van een Monorepo. Met een Monorepo kun je binnen een repository al je code beheren in meerdere libraries. Libraries kunnen bestaan uit features, componenten, utils of een ui-kit. Het idee van deze Monorepo is dat je jouw gemaakte features geheel of gedeeltelijk kan hergebruiken.

Het grote nadeel van een Monorepo is dat we elke micro frontend in de Monorepo opnieuw moeten compileren en releasen om een wijziging in een afzonderlijke micro-frontend te releasen! Om een monorepo te kunnen onderhouden kan je bijvoorbeeld Lerna, Nrwl of Angular Workplace gebruiken.

Server Side integration

SSI is het renderen van HTML op de server uit meerdere sjablonen of fragmenten. Deze fragmenten stellen de micro-frontends voor. In onderstaand voorbeeld wordt een index.html weergegeven, die server-side includes gebruikt om een fragment HTML-bestanden te includen (zie figuur 6).

```
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Feed me</title>
  </head>
  <body>
    <h1>Feed me</h1>
    <!--# include file="$PAGE.html" -->
  </body>
</html>
```

Figuur 6.

index.html

We serveren dit bestand met Nginx (zie figuur 7), waarbij de $PAGE-variabele wordt geconfigureerd door te matchen met de url die wordt gevraagd. Dus als de gebruiker kiest voor url '/browse', dan wordt de $PAGE-variabele gevuld met HTML-browse fragment.

Nginx

Wat hier niet wordt weergegeven, is hoe die verschillende fragment HTML-bestanden op de webserver terechtkomen. De veronderstelling is dat ze elk hun eigen build pipeline hebben, waardoor we wijzigingen in één fragment kunnen doorvoeren zonder dat dit invloed heeft op een andere pagina.

Runtime integration

Onder Runtime Integration wordt verstaan het bundelen en configureren van de micro-frontends in de frontend at Runtime (zie figuur 5). In deze situatie wordt geen gebruikgemaakt van een package.json om de afzonderlijke micro-frontends te bundelen.

In onderstaand voorbeeld worden Web Components gebruikt als techniek voor het aanmaken van afzonderlijke micro-frontends. Deze Web Components kunnen ook ingezet worden voor de voorgaande 'integration approaches'.

Wat zijn Web Components?

In het kort gezegd zijn Web Components geïsoleerde componenten die je kan (her)

```
server {
    listen 8080;
    server_name localhost;

    root /usr/share/nginx/html;
    index index.html;
    ssi on;

    # Redirect / to /browse
    rewrite ^/$ http://localhost:8080/browse redirect;

    # Decide which HTML fragment to insert based on the URL
    location /browse {
      set $PAGE 'browse';
    }
    location /order {
      set $PAGE 'order';
    }
    location /profile {
      set $PAGE 'profile'
    }

    # All locations should render through index.html
    error_page 404 /index.html;
}
```

Figuur 7.

gebruiken in HTML-pagina's en webapplicaties. Web Components worden ook wel 'custom elements' genoemd. Als developer ben je in staat om je eigen Custom Element te schrijven, wat in principe dus je eigen HTML-element is met zijn eigen CSS, HTML en Javascript. Dit element is gebaseerd op web standaarden en kan in de meest gebruikte browsers toegepast worden. Web Components zijn toekomstbestendig, omdat ze niet afhankelijk zijn van een framework of library. En is daarom als techniek zeer geschikt om een micro-frontend te bouwen.

Hoe maak je een Web Component?

In dit voorbeeld gaan we zelf een Web Component maken van de 'order food' pagina (zie figuur 3) uit onze voorbeeld applicatie. De naam van dit Web Component is micro-frontend-order-food en heeft in dit voorbeeld (zie figuur 8) de volgende parameters: data-name, data-img en data-menu.

```
<micro-frontend-order-food
      data-name="Curry delights"
      data-img="http://delights.png"
      data-menu={menuItems}>
</micro-frontend-order-food>
```

Figuur 8.

De implementatie van dit Web Component ziet er als volgt uit (zie figuur 9). Om dit voorbeeld eenvoudig te houden, zijn de menuItems buiten beschouwing gelaten.
Voor dit Web Component definiëren we eerst een class die extends van HTMLElement. Met HTMLElement kan je een Custom HTML element maken. In de constructor wordt eerst `super()` aangeroepen, wat betekent dat er gebruikgemaakt kan worden van al de logica van HTMLElement om een Web Component te kunnen bouwen. Vervolgens koppelen we een shadow DOM aan de Web Component. Een shadow DOM is een geïsoleerde DOM (of 'View') om iets te kunnen tonen voor dit Web Component.
We instantiëren met `document.createElement('img')` de gewenste image en setten de 'alt' en 'src' attributes middels de doorgegeven parameters data-name en data-img. Vervolgens wordt de image toegevoegd aan onze shadow DOM met `shadow.appendChild(img)`.

En tenslotte wordt een nieuwe Custom Element / Web Component gedefinieerd met:

```
customElements.define('micro-frontend-order-food', MicroFrontendOrderFood)
```

Dit Web Component heeft de naam micro-frontend-order-food. Deze kunnen we gebruiken in onze HTML pagina's om een image met een tekst te tonen.

Voorbeeld Runtime integration met Web Components

In figuur 10 wordt een index.html getoond die onze voorbeeld applicatie (ordering food) simuleert. Deze index.html representeert hier de container applicatie die o.a. de routing en de rendering verzorgt van de micro-frontends. Bovenaan zijn onze micro-frontends opgenomen met een <script> -tag. De zojuist besproken micro-frontend-order-food is gedefinieerd in de JavaScript bundle: **https://order.example.com/bundle.js**
De `<div id="micro-frontend-root">` is de placeholder waar de geselecteerde micro-frontends gerendered worden. De constante webComponentsByRoute bevat een lookup table voor het bepalen van het web component / de micro-frontend die je wilt renderen, als je een route selecteert.
De constante webComponentType bevat de daadwerkelijk gekozen micro frontend op basis van de geselecteerde route via: `window.location.pathname`

```
class MicroFrontendOrderFood extends HTMLElement {

  constructor() {
    super();
    var shadow = this.attachShadow({mode: 'open'});

    var img = document.createElement('img');
    img.alt = this.getAttribute('data-name');
    img.src = this.getAttribute('data-img');
    shadow.appendChild(img);
  }
}
customElements.define('micro-frontend-order-food', MicroFrontendOrderFood);
```

Figuur 9.

```
<html>
  <head>
    <title>Feed me!</title>
  </head>
  <body>
    <h1>Welcome to Feed me!</h1>

    <!-- These scripts don't render anything immediately -->
    <!-- Instead they each define a custom element type -->
    <script src="https://browse.example.com/bundle.js"></script>
    <script src="https://order.example.com/bundle.js"></script>
    <script src="https://profile.example.com/bundle.js"></script>

    <div id="micro-frontend-root"></div>

    <script type="text/javascript">
      // These element types are defined by the above scripts
      const webComponentsByRoute = {
        '/': 'micro-frontend-browse-restaurants',
        '/order-food': 'micro-frontend-order-food',
        '/user-profile': 'micro-frontend-user-profile',
      };
      const webComponentType = webComponentsByRoute[window.location.pathname];

      // Having determined the right web component custom element type,
      // we now create an instance of it and attach it to the document
      const root = document.getElementById('micro-frontend-root');
      const webComponent = document.createElement(webComponentType);
      root.appendChild(webComponent);
    </script>
  </body>
</html>
```

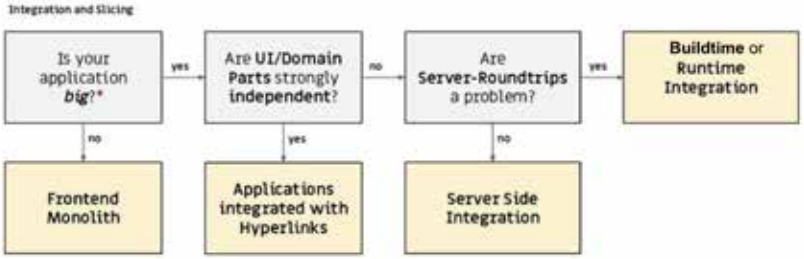
Figuur 10.

Middels `document.createElement (webComponentType)` instantiëren we de geselecteerde micro frontend. Tenslotte wordt deze gekoppeld aan de placeholder: `<div id="micro-frontend-root">`. Dit wordt gedaan met `root.appendChild(webComponent)`.

Het bovenstaande is duidelijk een primitief voorbeeld, maar het demonstreert de Runtime integration approach.

Welke integration Approach gebruiken?

In het schema in figuur 11 kan je afleiden welke integration approach je in welke situatie kan gebruiken. Voor kleine en/of niet complexe applicaties (waar 1 of 2 teams aan werken) kan je de integration approaches negeren en gewoon uitgaan van een frontend monoliet.



Figuur 11.

UI Component library

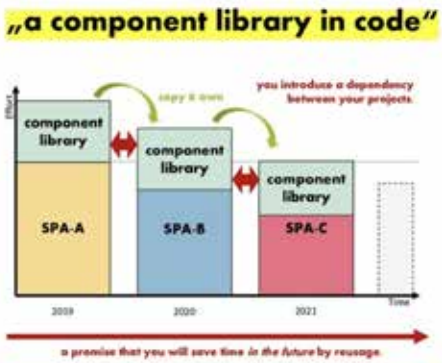
Een UI component library bestaat uit een reeks UI building blocks, zoals input elementen, lists, tab bars en grids etc.
Je zou ervoor kunnen kiezen om elke micro-frontend een eigen UI Component library te laten bezitten (zie figuur 12). Het nadeel hiervan is dubbele code en er is kans op minder consistentie in de styling en werking van de UI componenten. Voor meer consistency kunnen we ook een generieke UI Component library toepassen. Het nadeel hiervan is dat de micro-frontends dan gekoppeld zijn middels deze library. Als je kiest voor een generieke UI component, zorg er dan voor dat deze alleen UI-logica bevat en geen bedrijfs- of domeinlogica. Wanneer domeinlogica in een gedeelde bibliotheek wordt geplaatst, ontstaat er een hoge mate van afhankelijkheid tussen de micro-frontends.

Communicatie tussen micro frontends

Een van de meest gestelde vragen over micro-frontends is hoe je ze met elkaar kunt laten communiceren. Over het algemeen wordt aangeraden om de micro-frontends zo min mogelijk te laten communiceren, omdat dit een ongewenste koppeling introduceert dat we in de eerste plaats proberen te vermijden. Dat gezegd hebbende, is er vaak een zekere mate van communicatie tussen micro-frontends nodig. Door Custom events kunnen micro-frontends indirect communiceren, wat een goede manier is om directe koppeling te minimaliseren. Events kunnen als volgt worden gemaakt met de 'Event constructor': `New Event('build')` (zie figuur 13).
Het 'dispatch event' kan bijvoorbeeld geïnitieerd worden in micro frontend X voor het verzenden van een event met de naam 'build'. Micro-frontend Y luistert vervolgens naar dit event (met de addEventListener method) en handelt de verdere verwerking af.

Conclusie micro frontends

Micro-frontends hebben alles te maken met het opdelen van een grote webapp in Verticals. Onze technologie keuzes, onze codebases, onze teams en onze release processen (CI/CD) kunnen idealiter allemaal onafhankelijk van elkaar werken en evolueren, zonder overmatige coördinatie tussen andere teams. Deze architectuur heeft ook een keerzijde. Over de nadelen zou nog een artikel geschreven kunnen worden. We noemen hier enkele:
Als je een verandering wilt aanbrengen in de hele webapp, dan moet je wijzigingen aanbrengen in de afzonderlijke micro-frontends (en micro services) die door diverse andere teams geïmplementeerd worden.
Voor integratie testen van de gehele webapp moet je veel verschillende applicaties en servers opstarten. De moeilijkheid zit in het doortesten van de afhankelijkheden en de communicatie tussen de (gedistribueerde) micro-frontends.
Onafhankelijk gebouwde micro-frontends kunnen dubbele code bevatten. Hierdoor neemt het aantal bytes dat we via het netwerk naar onze eindgebruikers moeten verzenden, toe. Dubbele code betekent ook meer onderhoud, meer kans op fouten en minder consistentie in de styling en werking van de UI componenten.
Daarnaast zijn er veel praktijkgevallen waarin micro-frontends voordelen bieden. Grote organisaties als Spotify of IKEA zijn erin geslaagd micro-frontends geleidelijk in de tijd toe te passen op zowel oude als nieuwe codebases. Met micro-frontends kunnen deze bedrijven sneller inspelen op veranderingen in de markt en klantervaringen bieden die hun merken vooruithelpen. ■



Figuur 12.

```
const event = new Event('build');

// Listen for the event.
elem.addEventListener('build', function (e) { /* ... */ }, false);

// Dispatch the event.
elem.dispatchEvent(event);
```

Figuur 13.