

Inleiding vue

Vue is een single-page application framework. Dit wil zeggen dat er slechts één enkele pagina in de browser ingeladen zal worden en de content van deze ene pagina gaat dynamisch wijzigen op basis van input van de bezoeker op de site.

Een groot voordeel van single-page applications is een betere user experience. De gebruiker zal geen refreshes van de pagina zien gebeuren, bijvoorbeeld als er naar een andere pagina gesurft wordt, hierdoor voelt het surfen door de pagina een pak vloeiender aan.

Vue biedt echter nog een pak meer voordelen buiten een betere user experience, zo maakt het developen van een site gemakkelijker door vele zaken die anders nogal ingewikkeld kunnen zijn, zoals bv routing of het managen van state, eenvoudiger te maken door een deel voorgeprogrammeerde tools te voorzien die die zaken voor jou gaan doen. Hierdoor hoeft je die zaken niet meer volledig from scratch zelf te schrijven.

Verder maakt vue ook gebruik van zijn eigen templating language waarmee het heel eenvoudig wordt om dynamisch data en content op bepaalde plaatsen in je pagina te injecteren.

Bovendien maakt vue het ook heel eenvoudig om op een simpele manier data tussen componenten aan elkaar door te geven. Het gebruik van componenten zorgt er op zijn beurt weer voor dat de code die je schrijft een pak herbruikbaar gaat zijn. Zo kun je een component op meerder plaatsen in je pagina hergebruiken en afhankelijk van de gegevens die je eraan meegeeft kan de content binnen het component dan wijzigen.

1. Vue toevoegen aan een project

Vue kan op verschillende manieren aan je project worden toegevoegd.

- Als CDN package op de pagina
- Door de javascript files te downloaden en ze zelf te hosten
- Door vue te installeren via een package manager zoals npm
- Door de officiële CLI (command-line interface) te gebruiken om een standaard project op te starten.

Het is deze laatste manier die we in deze cursus gaan gebruiken om een project op te starten. Dit is de gemakkelijkste manier om aan een vue project te beginnen aangezien een hele berg configuratiewerk al reeds voor ons gedaan gaat worden. Moest je meer info willen over de andere manieren om vue aan een reeds bestaand project toe te voegen kun je [hier](#) zeker eens een kijkje gaan nemen.

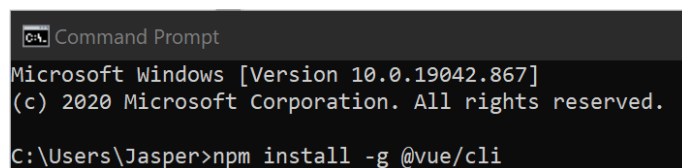
1.1 Installatie vue

Om vue te kunnen installeren via de CLI moeten we eerste de vue CLI downloaden. Om deze CLI te downloaden hebben we eerst npm (Node Package Manager) nodig. Deze staat normaal gezien al bij jullie op de laptop geïnstalleerd want die wordt mee geïnstalleerd als je Node installeert op je computer. Indien je geen Node hebt moet je die eerst gaan downloaden.

Eens Node geïnstalleerd is open je een nieuwe command prompt (opdrachtprompt) en daar typ je het volgende commando:

```
npm install -g @vue/cli
```

Like so:



```
Command Prompt
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Jasper>npm install -g @vue/cli
```

npm gaat nu de nodige packages downloaden om de vue CLI te installeren op je computer. De “-g” parameter die we meegeven wilt zeggen dat we de vue/cli package globaal gaan installeren op onze computer, zo kunnen we de vue CLI commando’s vanuit elke locatie (path) op de computer oproepen.

Lees de output in je command prompt nu zorgvuldig en kijk na of de installatie succesvol is gelukt. Als er iets is misgelopen zal dat beschreven staan in de output die je in je command prompt ziet. Een geslaagde installatie zal ongeveer de volgende output genereren, dit zal hoogstwaarschijnlijk lichtjes verschillen met wat jij ziet aangezien je het commando niet op hetzelfde moment uitvoerde als het

moment waarop de cursus geschreven werd.

```
C:\Users\Jasper>npm install -g @vue/cli
npm notice
npm notice New minor version of npm available! 7.7.6 -> 7.9.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v7.9.0
npm notice Run npm install -g npm@7.9.0 to update!
npm notice
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated @hapi/topo@3.1.6: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated @hapi/bourne@1.3.2: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated @hapi/hoek@8.5.1: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated @hapi/address@2.1.4: Moved to 'npm install @sideway/address'
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142
npm WARN deprecated @hapi/joi@15.1.1: Switch to 'npm install joi'

added 937 packages, and audited 938 packages in 27s

58 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
npm notice
npm notice New minor version of npm available! 7.7.6 -> 7.9.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v7.9.0
npm notice Run npm install -g npm@7.9.0 to update!
npm notice
C:\Users\Jasper>
```

Als extra controle of de installatie is geslaagd kunnen we kijken welke versie van de vue CLI er geïnstalleerd is. Als we 'vue is not a recognised command' als output krijgen wilt het zeggen dat de vue CLI niet correct is geïnstalleerd. Krijgen we wel een versie terug is de installatie gelukt.

```
C:\Users\Jasper>vue --version
@vue/cli 4.5.12
```

2. Een nieuw vue project creëren

Om een nieuw vue project aan te maken via de CLI navigeer je binnen een command prompt eerst naar de directory waar je het project wilt aanmaken met het cd-commando (change directory). Vervolgens gebruiken we het create commando van de vue CLI om een nieuw project aan te maken. De naam na het create commando wordt de naam van je project.

```
C:\Users\Jasper\WebstormProjects>vue create web_advanced_vue_
```

We gaan de laatste versie van vue gebruiken (v3) bij de installatie, kies bij de presets voor 'manually select features' door met de pijltjestoetsen te werken. Klik vervolgens op de enter-toets.

```
Vue CLI v4.5.12
? Please pick a preset:
  Default ([Vue 2] babel, eslint)
  Default (Vue 3 Preview) ([Vue 3] babel, eslint)
> Manually select features
```

Op het volgende scherm vinken we eslint (Linter / Formatter) uit en vink je Router en Vuex aan d.m.v. de spatiebalk.

```
Vue CLI v4.5.12
? Please pick a preset: Manually select features
? Check the features needed for your project:
> (*) Choose Vue version
  (*) Babel
  ( ) TypeScript
  ( ) Progressive Web App (PWA) Support
  (*) Router
  (*) Vuex
  ( ) CSS Pre-processors
  ( ) Linter / Formatter
  ( ) Unit Testing
  ( ) E2E Testing
```

Klik vervolgens weer op enter om naar het volgende scherm te gaan. Duid daar versie 3.x aan.

```
Vue CLI v4.5.12
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, Router, Vuex
? Choose a version of Vue.js that you want to start the project with
  2.x
> 3.x (Preview)
```

Vervolgens wordt gevraagd of we history mode willen gebruiken voor onze router, kies hier voor n (no) en klik op enter.

```
Vue CLI v4.5.12
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, Router, Vuex
? Choose a version of Vue.js that you want to start the project with 3.x (Preview)
? Use history mode for router? (Requires proper server setup for index fallback in production) (Y/n) n_
```

Vervolgens word ons gevraagd of we de configuratie van een aantal verschillende tools die we gaan gebruiken in onze package.json file willen steken of die in een aparte config file willen bijhouden voor elke tool. Als je hier al bekend mee bent mag je kiezen met welke manier je het liefste werkt. Binnen deze cursus gaan we werken met aparte config files en hier kies ik dus voor de eerste optie.

```
Vue CLI v4.5.12
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, Router, Vuex
? Choose a version of Vue.js that you want to start the project with 3.x (Preview)
? Use history mode for router? (Requires proper server setup for index fallback in production) No
? Where do you prefer placing config for Babel, ESLint, etc.? (Use arrow keys)
> In dedicated config files
  In package.json
```

Als laatste wordt ons nog gevraagd of we deze preset op willen slaan voor future use. Als je dit wilt kun je yes kiezen. Ik kies hier voor no.

```
Vue CLI v4.5.12
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, Router, Vuex
? Choose a version of Vue.js that you want to start the project with 3.x (Preview)
? Use history mode for router? (Requires proper server setup for index fallback in production) No
? Where do you prefer placing config for Babel, ESLint, etc.? In dedicated config files
? Save this as a preset for future projects? (y/N) n
```

Npm gaat nu beginnen met alle nodige packages te downloaden en zodra hij daarmee klaar is zou je het volgende als output moeten krijgen:

```
2) Successfully created project web_advanced_vue.  
2) Get started with the following commands:  
  
$ cd web_advanced_vue  
$ npm run serve  
  
C:\Users\Jasper\WebstormProjects>
```

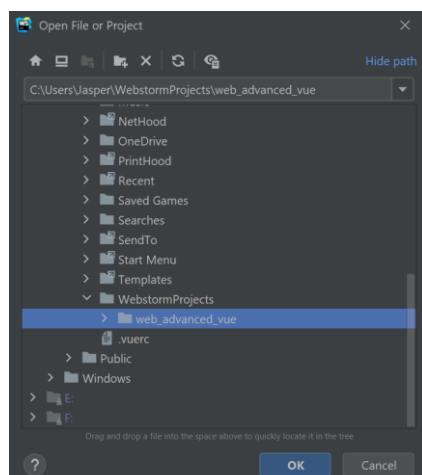
Hier zie je vervolgens ook hoe je je vue project kunt builden en runnen. Allereerst moeten we zorgen dat we in de directory van ons vue project zitten via het cd-commando. Vervolgens gaan we het serve-script runnen via het commando 'npm run'. Voer deze twee commando's uit.

Al je project-files worden nu gecompileerd en je vue spa (single-page application) draait op je lokale machine op poort 8080. Ga maar eens naar die url in de browser en dan zou je een vue welkomspagina moeten zien, controleer of dit werkt.

```
C:\Users\Jasper\WebstormProjects>cd web_advanced_vue  
C:\Users\Jasper\WebstormProjects\web_advanced_vue>npm run serve  
  
> web_advanced_vue@0.1.0 serve  
> vue-cli-service serve  
  
INFO Starting development server...  
98% after emitting CopyPlugin  
  
DONE Compiled successfully in 1331ms  
  
App running at:  
- Local: http://localhost:8080/  
- Network: http://192.168.0.229:8080/  
  
Note that the development build is not optimized.  
To create a production build, run npm run build.
```

De development server stoppen kan door ctrl + c in te drukken.

Om te beginnen gaan we ons vue project openen in webstorm. Klik op File > Open en selecteer de directory die je hebt aangemaakt met het create-commando van de vue CLI.



3. Een vue instantie aanmaken en mounten

Een vue website is wat men een single-page application noemt. Dit wil zeggen dat in plaats van telkens een nieuwe pagina te laden in de browser we slechts één pagina gaan laden en deze pagina zijn content dynamisch gaan veranderen.

De pagina waar onze vue instantie ingeladen wordt is index.html die je kan vinden in de public directory.

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

Binnen deze index.html file vinden we een div-element terug met het id `app`. Dit is het element waarbinnen we onze vue app gaan renderen.

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'

createApp(App).use(store).use(router).mount( rootContainer: '#app');
```

Het mounten van deze vue app binnen de div met id app gebeurt in main.js. Hier wordt aan de createApp methode die we importeren vanuit de vue package een nieuwe instantie van het App.vue component meegegeven.

createApp() gaat een vue applicatie voor ons creëren, daaraan kunnen we enkele extra tools die we binnen de app willen gebruiken meegeven zoals de router en de vuex store. Wat die router en store precies doen gaan we later in de cursus zien.

Nadat we de router en de store hebben toegevoegd aan onze applicatie gaan we de applicatie mounten binnen ons div-element d.m.v. de mount methode. Daarbinnen geven we het id van het

element mee waarbinnen we onze App willen renderen. In ons geval dus het element met id 'app' waarnaar we verwijzen via de #app.

4. De vue component

Binnen vue gaan we werken met herbruikbare componenten. Het herhalen van grote brokken HTML die steeds terugkomen op al je verschillende pagina's is hiermee dus van verleden tijd.

Vue componenten bestaan uit drie delen:

- Template

De template bevat alle HTML en vue templating language van het component.

- Script

Het script bevat alle properties (eigenschappen) en methods (methodes) van het component.

- Style

De style bevat alle styling van het component, bv CSS of SASS

4.1 App.vue

In hoofdstuk 3 zagen we hoe we binnen main.js een nieuwe vue app creëren via de createApp methode en hierbinnen het App.vue component gaan renderen. Laten we nu wat dieper ingaan op die App.vue component.

```
<template>
  <div id="nav">
    <router-link to="/">Home</router-link> |
    <router-link to="/about">About</router-link>
  </div>
  <router-view/>
</template>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
}

#nav {
  padding: 30px;
}

#nav a {
  font-weight: bold;
  color: #2c3e50;
}

#nav a.router-link-exact-active {
  color: #42b983;
}
</style>
```

Binnen de template tags staat de HTML die getoond gaat worden zodra een component gerenderd wordt.

In het geval van de App.vue component gaat er een div-element met twee router-link componenten en een router-view component gerenderd worden.

De router-link component

De **router-link** component is een component om navigatie toe te staan binnen een vue applicatie die de vue router gebruikt. 'to' is een prop van de router-link component. Een prop is een waarde die we aan een component meegeven, een beetje zoals attributen in HTML.

Het pad dat we dus meegeven als prop/attribuut aan de router-link moet verwijzen naar een route die we gedefinieerd hebben in onze router. Laten we dus eens gaan kijken hoe zo een route gedefinieerd wordt.

Onder de src map (directory) staat de map router, hierbinnen kun je de file index.js terugvinden. Hier worden de verschillende routes van de vue-router gedefinieerd.

```
1  import { createRouter, createWebHashHistory } from 'vue-router'
2  import Home from '../views/Home.vue'
3
4  const routes = [
5    {
6      path: '/',
7      name: 'Home',
8      component: Home
9    },
10   {
11     path: '/about',
12     name: 'About',
13     // route level code-splitting
14     // this generates a separate chunk (about.[hash].js) for this route
15     // which is lazy-loaded when the route is visited.
16     component: () => import(/* webpackChunkName: "about" */ '../views/About.vue')
17   }
18 ]
19
20 const router = createRouter( options: {
21   history: createWebHashHistory(),
22   routes
23 })
24
25 export default router
```

Op regel 20 wordt een instantie van de vue-router gecreëerd via de createRouter methode die we importeren uit de vue-router package op regel 1. Aan deze methode geven we een object met de opties die we in willen stellen in onze router mee. Op dit moment bevat dit object de history-mode die we willen gebruiken binnen de router en een array met de verschillende routes die we willen voorzien in de app. Dit is de history-mode die we bij het creëren van de app via de CLI hebben gekozen. Moest je je toen dus vergist hebben kan je hier het history type veranderen.

Zoals je kan zien is elke nieuwe route een afzonderlijk object binnen de routes array. Elke route object moeten we voorzien van een path, een name en vervolgens geven we mee welk component er gerenderd moet worden in het **router-view** component (App.vue regel 6) indien er naar die route genavigeerd wordt.

In het eerste route object zien we dat het **Home** component gerenderd moet worden als naar de `'/'`-route genavigeerd wordt. Welk component dat **Home** component is kun je zien op regel 2. Daar importeren we hetgeen we exporteren uit `'../views/Home.vue'` (Home.vue regel 12) en dat geven we de verwijzing `'Home'`. Hoe die export default precies werkt komt nog aan bod, wat nu belangrijk is is dat je begrijpt dat we aan de component property van ons route object het component dat we willen renderen als naar die route genavigeerd wordt moet worden meegegeven. Zo dus:

```
component: Home
```

De router-view component

De **router-view** component gaat de component die bij een bepaalde route hoort renderen.

Als iemand dus op de `<router-link to="/">Home</router-link>` (App.vue regel 3) **router-link** component klikt in de browser gaat vue kijken welk component bij de `'/'`-route hoort die we als prop hebben meegegeven.

router/index.js

```
const routes = [
  {
    path: '/', // <-- Dit is het pad dat overeenkomt met de 'to' prop
    name: 'Home',
    component: Home // <-- Dit is het component dat gerenderd zal worden door router-view
  },
```

Indien de gebruiker op de tweede route zou klikken zal **router-view** de Home component uit het DOM verwijderen en zal het About component (About.vue) gerenderd worden in de plaats.

<style>

Elk vue component kan zijn eigen stijlopmaken beheren. Voor elk component willen we afzonderlijk de styles die nodig zijn om het component van opmaak te voorzien bijhouden. De styles van afzonderlijke componenten moeten we dus steeds scoped maken tot enkel dat component, met de uitzondering van top-level App components. In top-level App components mag je wel globale styles plaatsen waarvan je wilt dat die je in meerdere componenten van toepassing zijn.

Een voorbeeld van globale styles vinden we terug in App.vue op regel 9. Daar schrijven we `<style>`, aangezien we niet het `'scoped'` attribuut meegeven zijn alle CSS styles die hierbinnen staan globale styles en zullen zij dus van toepassing zijn op alle elementen binnen de app die matchen met de CSS selectors. Willen we de styles enkel van kracht laten zijn binnen de elementen die we renderen binnen dit component dat moeten we het `'scoped'` attribuut toevoegen. Onze style-tag zou er dan als volgt uitzien: `<style scoped>`

4.2 Ons eerste eigen component schrijven

Laten we nu eens een eerste simpel component schrijven dat enkel wat tekst gaat weergeven op de pagina.

Maak onder de map (directory) components een nieuwe vue file aan genaamd BuildingItem.vue. Als je hiervoor de voorgedefinieerde template van webstorm gebruikt zou je file er nu zo uit moeten zien:

```
<template>

</template>

<script>
export default {
  name: "BuildingItem"
}
</script>

<style scoped>

</style>
```

BuildingItem.vue

Wat je vast al is opgevallen is dat er nu ook een <script>-tag is verschenen binnen de file. De script tag bevat alle properties en methodes van ons component. Een eerste property die elk element moet hebben is een name. Deze laten we steeds overeenkomen met de naam van de file, of beter gezegd, laat de naam van de file overeenkomen met de naam die je aan het component wilt geven. Wat er nog allemaal binnen de <script>-tag komt te staan gaan we gaandeweg leren doorheen de cursus. Daar wijden we later verder over uit.

Plaats binnen de template enkele HTML-elementen met tekst en binnen de style tag wat opmaak.

```
<template>
  <div class="building-item">
    <h2>Imperial Barracks</h2>
    <label>Gold: 5000</label>
    <label>resources needed: 100 steel, 100 ammunition</label>
  </div>
</template>

<script>
export default {
  name: "BuildingItem"
}
</script>

<style scoped>
.building-item{
  display: flex;
  align-items: flex-start;
  flex-direction: column;
  justify-content: space-evenly;
}
</style>
```

We hebben nu een basic component, nu moeten we die weergeven op de pagina. Laten we hem toevoegen aan de homepage die op route '/' gerenderd wordt (Home.vue).

De Home.vue component renderd al een hoop HTML, deze boilerplate code mag je verwijderen en we gaan in de plaats ons nieuwe component renderen binnen Home.vue.

```

1 <template>
2 <div class="home">
3   
4   <HelloWorld msg="Welcome to Your Vue.js App"/>
5 </div>
6 </template>
7
8 <script>
9   // @ is an alias to /src
10  import HelloWorld from '@components/HelloWorld.vue'
11
12  export default {
13    name: 'Home',
14    components: {
15      HelloWorld
16    }
17  }
18 </script>

```

```

1 <template>
2
3 </template>
4
5 <script>
6   // @ is an alias to /src
7
8  export default {
9    name: 'Home',
10   components: {
11
12   }
13 }
14 </script>

```

Indien we een component willen renderen binnen één van onze andere componenten moeten we deze eerst importeren. Dit doen we via de import.

Nadat we het component hebben geïmporteerd moeten we deze nog aan de lijst (let op! Ik zeg lijst maar het is een object!) van components die ons component gebruikt toevoegen. De alias die we achter het import keyword schreven schrijven we nu binnen het object 'components'.

```

<script>
// @ is an alias to /src
import BuildingItem from '@components/BuildingItem.vue'

export default {
  name: 'Home',
  components: {
    BuildingItem
  }
}
</script>

```

Nu ons BuildingItem component in ons Home component is geïmporteerd kunnen we hem gebruiken binnen onze <template> om een BuildingItem component te renderen. We gebruiken de naam die we aan het components object hebben toegevoegd.

```

<template>
  <BuildingItem />
</template>

```

Er zal nu een BuildingItem component gerenderd worden zodra de Home.vue component gerenderd wordt. Om nu te testen of onze code werkt gaan we onze development server starten met het commando 'npm run serve', dit kun je doen vanuit een terminal in webstorm.

```

Terminal: Local x +
C:\Users\Jasper\WebstormProjects\web_advanced_vue>npm run serve

```

Als je alles goed hebt gedaan zou je nu de HTML die we binnen ons BuildingItem component hebben geschreven moeten kunnen zien.

Imperial Barracks

Gold: 5000

resources needed: 100 steel, 100 ammunition

4.3 Component data

Elk component kan zijn eigen data bijhouden, de data die het component nodig heeft om weer te geven op de pagina of bewerkingen mee wilt doen. Of data die hij op zijn beurt weer door gaat geven aan een ander component.

Een component kan data op twee verschillende manieren bijhouden. Enerzijds kan een component data bevatten die we pas creëren of ophalen binnen het component, of anderzijds data die het component binnenkreeg (overerfde) van zijn parent component (het component dat dit component renderd). In dat geval spreken we over props.

Belangrijk! Telkens wanneer de waarde van een prop of een data property wijzigt binnen je component gaat er een automatische re-render van het component plaatsvinden. De oude component wordt dus verwijderd uit het DOM en een nieuwe met de geüpdate data zal worden gerenderd in de plaats.

Laten we beginnen bij het gemakkelijkste, component data.

Data binnen een component moet steeds worden opgeslagen binnen een object dat we returnen via de data() functie. Als simpel voorbeeld gaan we de waardes van gold, steel en ammunition die we eerder als vaste waardes hadden gedeclareerd nu bijhouden als data van het component. Dat doen we op de volgende manier:

```
export default {  
  name: "BuildingItem",  
  data(){  
    return {  
      gold: 5000,  
      steel: 100,  
      ammunition: 100  
    }  
  }  
}
```

Het component bevat nu 3 waardes die we kunnen gebruiken om weer te geven op te pagina of om berekeningen mee te doen.

*Belangrijk om te onthouden hierbij is dat als je wilt dat je component re-renderd op het moment dat één van deze state properties veranderd van waarde, dan **MOET** je deze een default waarde geven. Anders kan vue geen veranderingen in de waarde van de state property detecteren.*

Als we deze waarden willen gebruiken om dynamisch de content op de pagina aan te passen kunnen we de handige templating syntax van vue gebruiken. Vanaf nu gaat hopelijk stilaan de toegevoegde waarde van je website met vue te maken hopelijk duidelijk worden.

Als we de waarde uit één van onze data properties willen weergeven op de pagina kunnen we dit eenvoudig doen door de keyname (of property) van ons object te schrijven binnen de template, omringd door twee curly brackets aan weerszijden te schrijven. Dit noemen we **mustaches**. Like so:

```
<template>
  <div class="building-item">
    <h2>Imperial Barracks</h2>
    <label>Gold: {{ gold }}</label>
    <label>resources needed: {{ steel }} steel, {{ ammunition }} ammunition</label>
  </div>
</template>
```

Dit noemen we one-way data-binding. Dit wil zeggen dat de property enkel weergegeven zal worden op de pagina, maar de waarde binnen data() kan niet gewijzigd worden vanuit de template/HTML.

4.4 Component props

Vaak gaan we data doorgeven van het ene component naar het andere. Om data van een parent component door te geven naar een child component gaan we props gebruiken.

Het definiëren van props gaat heel eenvoudig. We beginnen met het toevoegen van de props property aan ons component.

```
<script>
export default {
  name: "BuildingItem",
  props: {}, // <-- Voeg props toe
  data(){...},
  methods: {...}
}
</script>
```

BuildingItem.vue

Binnen dit props object gaan we elke prop die het component binnenkrijgt definiëren en we geven telkens mee wat voor datatype deze prop hoort te zijn. Dit is niet verplicht in vue maar zolang als jullie studeren aan de PXL spreken we af dat het wel verplicht is om dit te doen.

De reden dat we aan type-checking willen doen is omdat dit het opsporen van fouten aanzienlijk gemakkelijker maakt. Moest er een type mismatch zijn zouden we dat als een error in de console kunnen zien. Hierdoor weten we meteen dat onze code niet werkt omdat we iets verkeerd doorgeven vanuit ons parent component.

We gaan het component updaten om nu de titel die in het h2-element moet komen te staan binnen te krijgen van zijn parent component. Allereerst moeten we dus een nieuwe String prop toevoegen binnen het props-object. De key die we gebruiken moet overeenkomen met de naam van de prop die we zodadelijk gaan meegeven vanuit de parent.

Binnen onze template kunnen we nu op dezelfde manier als data properties deze prop value binden. Verwijder de statische tekst 'Imperial barracks' en bind in de plaats de title prop.

```
<h2>{{title}}</h2>
```

Aangezien title een waarde is die we van de parent component door moeten krijgen moet er nu wel op de plaats waar we dit component creëren (Home.vue) een waarde aan title gegeven worden. Dat doen we op de volgende manier:

```
<BuildingItem title="Imperial Barracks" />
```

Home.vue

Start opnieuw je development server en kijk of het passen van de prop gelukt is.

In bovenstaand voorbeeld is de waarde die we aan title meegeven een statische waarde. Deze BuildingItem component zal dus altijd de waarde 'Imperial Barracks' meekrijgen.

Indien we een dynamische waarde mee willen geven, bijvoorbeeld een property uit data of een prop van de Home.vue component moeten we gebruik maken van de **v-bind directive**.

De code zou er dan als volgt uitzien:

```
<BuildingItem v-bind:title="titleToPass" />
```

```
export default {
  name: 'Home',
  data() {
    return{
      titleToPass : 'Imperial Barracks' // <-- De data property die we willen doorgeven toevoegen
    }
  },
  components: {BuildingItem: BuildingItem...}
}
```

Via de v-bind directive kunnen we dus data of props binden aan onze template. Hierover meer in het volgende hoofdstuk.

5. Directives

Attributen binden aan data of props met v-bind

De v-bind directive wordt gebruikt om data of props te binden binnen onze template, bijvoorbeeld om een HTML-attribuut een waarde te geven.

Ter illustratie gaan we het h2-element binnen **BuildingItem.vue** een id geven gekoppeld is aan een waarde die uit onze data komt.

In dit geval zouden we geen mustaches (dubbele curly brackets) kunnen gebruiken maar gaan we gebruik moeten maken van een *directive*. In dit geval de **v-bind directive**. Hier volgt een voorbeeld:

```
export default {
  name: "BuildingItem",
  props: {title: String...},
  data(){
    return {
      titleId: 'building-item__title', // <-- Voeg de titleId property toe binnen data
      gold: 5000,
      steel: 100,
      ammunition: 100
    }
  },
  methods: {...}
}
```

Via v-bind: kunnen we nu aan het id-attribuut de waarde die in titleId steekt koppelen.

```
<h2 v-bind:id="titleId">{{title}}</h2>
```

Het toevoegen van de v-bind directive zorgt ervoor dat de waarde die bij de nameld property binnen onze data functie hoort zal ingevuld worden als waarde van het id-attribuut van onze h2. Controleer of dit werkt door in je dev tools te gaan kijken naar het DOM.

```
<h2 id="building-item__title" data-v-025fd097>Imperial Barracks</h2>
```

Luisteren naar DOM events met v-on

Vue voorziet ook template syntax om naar DOM events te luisteren, daarvoor gebruiken we de **v-on** directive.

We gaan een nieuwe knop toevoegen binnen de template waar we aan het click-event een event-listener gaan toevoegen.

```
<template>
  <div class="building-item">
    <h2 v-bind:id="nameId">Imperial Barracks</h2>
    <label>Gold: {{ gold }}</label>
    <label>resources needed: {{ steel }} steel, {{ ammunition }} ammunition</label>
    <input v-on:click="purchaseBuilding"
      class="building-item__button"
      type="button"
      value="buy this building"
    />
  </div>
</template>
```

Telkens als er op deze input geklikt wordt gaat de purchaseBuilding() methode uitgevoerd worden. Deze hebben we echter nog nergens gedefinieerd, dat gaan we dus als volgende doen.

De nieuwe property van vue components die we hiervoor nodig hebben is de methods-property. Dit is een object waarin we methodes die ons component nodig heeft om te kunnen functioneren gaan toevoegen.

De purchaseBuilding() methode gaan we hier dus moeten toevoegen om hem op het moment van de klik uit te kunnen voeren.

```
export default {
  name: "BuildingItem",
  props: {
    title: String
  },
  data() { ... },
  methods: {
    purchaseBuilding() {
      alert('Congratulations, ' + this.title + ' built!')
    }
  }
}
```

Via this.title kunnen we de waarde die in de title prop steekt gebruiken en weergeven in het alert-venster. Wil je een data property gebruiken binnen je methodes kan dat op dezelfde manier, via **this.propertyName**

Test nu opnieuw je code in de browser en kijk of je ook effectief de alert ziet verschijnen als je op de knop klikt.

Een lijst mappen aan elementen met de v-for

Met de **v-for directive** kunnen we een lijst van items renderen gebaseerd op een array. Hiervoor gaan we een speciale syntax gebruiken in de vorm van `v-for="item in items"` waar items een array is waarover we itereren en item het element binnen de array is waarover geïtereerd wordt.

Pas de **Home.vue** component aan zodanig dat we nu niet één titleToPass hebben maar nu een array met meerdere objecten. Elk van deze objecten zal een building voorstellen en voor elk van deze object gaan we dus een BuildingItem component renderen.

```
export default {
  name: 'Home',
  data() {
    return {
      buildings : [
        {titleToPass : 'Imperial Guardsmen Barracks'},
        {titleToPass : 'Astartes Barracks'},
        {titleToPass : 'Psyker Barracks'}
      ]
    }
  },
  components: {BuildingItem: BuildingItem...}
}
```

Nu het component een array met 'buildings' heeft gaan we hier over itereren via de v-for directive. In onderstaand voorbeeld zal er voor ieder item in de buildings array een li-element gecreëerd worden met een BuildingItem component erin.

```
<template>
  <div class="buildings">
    <ul>
      <li v-for="building in buildings" :key="building.titleToPass">
        <BuildingItem v-bind:title="building.titleToPass"/>
      </li>
    </ul>
  </div>
</template>
```

We geven ook een key attribuut mee aan elke li. Dit is niet verplicht maar is handig voor het algoritme dat vue gebruikt om de identiteit van nodes te tracken. Het is aangeraden om in een v-for steeds een key mee te geven.

De homepage zou er nu als volgt uit moeten zien:

• Imperial Guardsmen Barracks

Gold: 5000

resources needed: 100 steel, 100 ammunition

buy this building

• Astartes Barracks

Gold: 5000

resources needed: 100 steel, 100 ammunition

buy this building

• Psyker Barracks

Gold: 5000

resources needed: 100 steel, 100 ammunition

buy this building

Indien we ook de index van het huidige item nodig hebben kunnen we een tweede optionele parameter gebruiken binnen de v-for.

```
<template>
  <ul>
    <li v-for="(building, index) in buildings">
      <h2>{{index}}</h2>
      <BuildingItem v-bind:title="building.titleToPass" />
    </li>
  </ul>
</template>
```

Deze code hoeft je niet over te nemen in het project.

Conditioneel renderen via v-if en v-show

De **v-if directive** wordt gebruikt om een bepaalde blok code enkel te renderen als de value die we in de v-if controleren truthy is.

We kunnen ook een **v-else directive** gebruiken om een nadere blok code te renderen indien de waarde falsy was. Deze v-else moet meteen na de v-if blok volgen of het zal niet werken.

```
<template>
  <div class="buildings">
    <ul v-if="buildings.length"> <!-- Renderd de ul als buildings.length truthy is -->
      <li v-for="building in buildings" :key="building.titleToPass">
        <BuildingItem v-bind:title="building.titleToPass"/>
      </li>
    </ul>
    <p v-else>Er zijn geen buildings.</p> <!-- Renderd de p als buildings.length falsy is -->
  </div>
</template>
```

De **v-show directive** werkt grotendeels hetzelfde als de v-if. Het verschil ligt in dat een v-show altijd het element zal renderen in het DOM en de display property zal togglen.

Een v-if gaat de blok code gewoon volledig niet renderen.

Ook ondersteunt de v-show directive geen v-else directive.

Two-way data binding met v-model

Eerder zagen we hoe we one-way data binding kunnen gebruiken om data of props van ons component op de pagina te weergeven.

One-way wil zeggen dat de data slechts in één richting stroomt.

Two-way data binding gaat ons toestaan om de gebruiker de waarde van een data property aan te passen. Bijvoorbeeld door het gebruik van een input-element.

Pas de code in Home.vue aan zodat we indien er geen buildings zijn een input zien waar we zelf een error message mee kunnen geven. Laat deze error message dan zien binnen een paragraaf.

```
<div v-else>
  <input type="text" v-model="emptyBuildingsArrayMessage" />
  <p>{{emptyBuildingsArrayMessage}}</p>
</div>
```

```
data() {
  return {
    buildings: [
      /*{titleToPass: 'Imperial Guardsmen Barracks'},
      {titleToPass: 'Astartes Barracks'},
      {titleToPass: 'Psyker Barracks'}*/
    ],
    emptyBuildingsArrayMessage: 'Er zijn geen buildings.'
  }
},
```

Zodra de div gerenderd wordt zal de paragraaf de initiële waarde die we in onze data gedefinieerd hebben krijgen. Door de two-way data binding van v-model kunnen we nu de waarde van emptyBuildingsArrayMessage in data veranderen door een andere waarde in het inputveld te typen. Vue zal detecteren dat er iets in de data van het component veranderd is en zal automatisch het component gaan re-renderen. Daarom zien we meteen onze aanpassingen verschijnen op het scherm.

5.1 Computed properties

In-template expressions zijn heel handig maar deze zijn bedoeld voor simpele operaties. Te veel logica in templates plaatsen maakt ze bloated en moeilijk te onderhouden. Neem als voorbeeld het volgende stukje code:

```
<span>{{ buildings.length > 0 ? 'Buildings available' : 'No buildings available' }}</span>
```

Binnen deze span wordt afhankelijk van de conditie 'buildings.length > 0' een bepaalde tekst weergegeven. In-template expressions zoals deze willen we liefst uit onze template trekken en in de plaats gebruiken we one-way data binding om de tekst te weergeven binnen de span.

Om one-way data binding te kunnen gebruiken moeten we een variabele hebben die we willen koppelen (binden) in de template. Aangezien we op het moment van de render nog niet weten welke van de twee tekstwaarden we willen weergeven gaan we hier gebruiken maken van een computed (berekende) property.

Computed properties zijn gegevens die je component gaat berekenen ipv ze meteen een initiële waarde te geven binnen data.

In plaats van de in-template expressie gaan we nu een computed property binden.

```
<span>{{ buildingsAvailableMessage }}</span>
```

Dit is een property die nog niet bestaat, vandaar krijgen we nu nog een groene onderlijning te zien. Om een computed property toe te voegen aan een component voegen we een property genaamd 'computed' toe aan ons component en de waarde hiervan is een object die de getter-methodes bevat van al onze computed properties (berekende properties).

```
export default {
  name: 'Home',
  data() {...},
  computed: {}, // <-- Computed property toevoegen
  components: {
    BuildingItem
  }
}
```

De naam van de getter moet overeenkomen met hoe we willen dat de computed property heet.

```
export default {
  name: 'Home',
  data() { ... },
  computed: {
    // a computed getter
    buildingsAvailableMessage() {
      // `this` points to the vm instance
      return this.buildings.length > 0 ? 'Buildings available' : 'No buildings available';
    }
  },
  components: { BuildingItem: BuildingItem... }
}
```

Afhankelijk van het aantal items in de buildings array zal nu de buildingsAvailableMessage getter de ene of de andere String value returnen. Het resultaat wordt weergegeven in het span element.

Opdrachten

Onze eerste componenten beginnen nu mooi vorm te krijgen. Tijd om te oefenen.

- Rename de 'Home'-view naar 'Buildings' en maak een nieuwe lege Home view aan.
 - Voeg op in de nieuwe Home view een image toe met het logo van het spel dat je wilt maken.
 - Verander de about route zodat deze naar de Buildings view route en pas ook de tekst in de navigation aan.
 - Voorzie de pagina van wat opmaak, zorg ervoor dat je logo in het midden van de pagina komt te staan.
 - Creëer een Header component. Render dit component binnen App.vue en verplaats de navigation naar het Header component.
- Zorg ervoor dat de verschillende views niet in de Header component worden geladen maar eronder.
- Voeg in de Header links een ul toe met 3 resources. De li's binnen deze ul worden gerenderd op basis van een array resources die je in data bijhoudt. Elke resource heeft een name, een amount en een image.
 - Voeg in de Header in het midden een h1 toe met de titel van je spel.
 - Zorg dat in de Header de navigatie rechts komt te staan.
 - Voeg net zoals bij de resources een image toe aan het BuildingsItem component. De src komt binnen als prop van de parent component. Zorg ook dat de verschillende resource costs als prop binnenkomen.

Het eindresultaat hoort er ongeveer gelijkaardig uit te zien, als je een ander thema hebt gebruikt is dat ook prima:

1000 1000 1000

FOR THE EMPEROR!

HomeBuildings



1000 1000 1000

FOR THE EMPEROR!

HomeBuildings



Gold: 5000
resources needed: 100 steel, 100 ammunition
[buy this building](#)



Gold: 5000
resources needed: 100 steel, 100 ammunition
[buy this building](#)



Gold: 5000
resources needed: 100 steel, 100 ammunition
[buy this building](#)

Buildings available

5.2 Custom events schrijven

Eerder in de cursus leerden we al hoe we props konden gebruiken om data van een parent component door te geven naar een child component. Wat nu als we data van een child component naar een parent component willen doorgeven? Om dit mogelijk te maken gaan we gebruik maken van **custom events**.

Een component kan een event afvuren via de **\$emit** methode.

```
this.$emit('myEvent');
```

Merk op hoe we camel-casing gebruiken voor de eventnaam in het child component.

In de parent component gaan we nu luisteren naar het event en hier gaan we een event handler koppelen aan het event. Naar ons emitted event luisteren kan hier via de kebab-case schrijfwijze van onze functie. Deze conversie gebeurt achterliggend door vue.

```
<my-component @my-event="doSomething"></my-component>
```

In bovenstaand voorbeeld zal de doSomething-methode in de huidige component uitgevoerd worden zodra de my-comonent het myEvent afvuurt.

Custom events met parameters

Een component kan ook data doorgeven bij het emitten van een event. Dat kan door een tweede parameter aan de \$emit methode mee te geven.

```
this.$emit('myEvent', 100);
```

Binnen de parent component gaan we dan onze event-listener moeten aanpassen zodat die nu een parameter verwacht.

```
doSomething(number){ this.totalPrice += number }
```

De number parameter zal de waarde '100' bevatten indien we het bovenstaande event zouden emitten.

Binnen de parent component kunnen we ook aan deze waarde via **\$event**. Stel dat we deze code binnen onze template zouden willen schrijven ipv er een aparte event handler functie voor te schrijven kunnen we ook dit doen:

```
<my-component @my-event="totalPrice += $event"></my-component>
```


Implementatie van custom events in ons spel:

Voeg aan de buildings-objecten in Buildings.vue een nieuwe property toe: amountBuilt.

```
{
  titleToPass: 'Imperial Guardsmen Barracks',
  gold: 5000,
  steel: 100,
  ammunition: 100,
  src: 'Canoncamp.png',
  amountBuilt: 1
},
```

Vanuit de BuildingItem component gaan we de 'amountBuilt' waarde in de Buildings.vue file met 1 ophogen telkens als er op de 'buy building'-knop geklikt wordt.

Om dit klaar te spelen gaan we de purchaseBuilding methode in de BuildingItem component aanpassen zodat er bij het klikken op de knop een custom event afgevuurd wordt.

```
methods: {
  purchaseBuilding() {
    this.$emit('purchaseBuilding', this.titleToPass);
  },
}
```

Binnen de parent component gaan we nu de template moeten aanpassen om te gaan luisteren naar het purchaseBuilding custom event. We gaan een eventlistener en eventhandler toevoegen.

```
<BuildingItem v-bind="building" v-on:purchase-building="updateBuildings"/>
```

Op het moment dat het custom event afgevuurd wordt gaan we het event verder afhandelen in een updateBuildings methode die je onder methods toe mag voegen. Binnen deze methode gaan we het buildings object dat aangepast moet worden zoeken binnen data(), zodra we dat object gevonden hebben verhogen we de amountBuilt waarde van dat object met 1.

```
methods: {
  updateBuildings(buildingToUpdate) {
    let buildingObjectToUpdate = this.buildings.find(building => {
      return building.titleToPass === buildingToUpdate;
    });
    buildingObjectToUpdate.amountBuilt++;
  }
},
```

6. Vuex

Tot nu toe zagen we hoe elke component zijn eigen state bijhoudt (data) en hoe we die data door kunnen geven naar child components. Dit is heel handig voor data die enkel die component of zijn children nodig hebben, echter gaat elke applicatie ook data hebben die door meerdere componenten gebruikt wordt. Die data gaan we in een globale data store steken die zal dienen als single source of truth voor deze data.

Telkens wanneer binnen deze store een datagegeven veranderd gaat vue automatisch elk component die dat gegeven gebruikt re-renderen met de nieuwe waarde. Alle componenten worden dan dus meteen geüpdate op basis van de nieuwe waarde.

In main.js creëren we onze App, hier geven we mee dat we willen dat onze App gebruik maakt van een router (router/index.js) en ook dat we een store willen gebruiken (store/index.js). Laten we eens gaan kijken naar onze store file.

```
import { createStore } from 'vuex'

export default createStore( options: {
  state: {
  },
  mutations: {
  },
  actions: {
  },
  modules: {
  }
})
```

Via de createStore methode die we importeren uit de vuex package creëren we een nieuwe vuex store. Deze bevat 4 onderdelen:

1. State

Dit werkt op dezelfde manier als data binnen onze componenten. Hier geven we de initial state (default values) mee voor onze store.

2. Mutations

De enige manier om de data binnen een vuex store aan te passen is door een mutation te committen. Deze mutations werken ongeveer zoals events. Een component zal de nextTurn mutation aanroepen en die mutation gaat op zijn beurt de state updaten.

Een mutation krijgt altijd als eerste parameter de state.

```
export default createStore( options: {
  state: {
    turn: 1
  },
  mutations: {
    nextTurn(state) {
      state.turn += 1;
    }
  },
  actions: {},
  modules: {}
})
```

3. Actions

Waar mutations steeds synchrone operaties zijn kunnen actions gebruikt worden om asynchrone operaties te verrichten. In plaats van de state direct te veranderen zal een action mutations committen. Die mutations gaan dan de state updaten.

Op actions gaan we niet dieper in in deze cursus.

4. Modules

Binnen een groot project kan een vue store vrij snel heel groot worden. Om de store gemakkelijker onderhoudbaar te maken kan hij opgesplits worden in verschillende modules. Elk van deze modules heeft een state, getters, mutations, actions en modules net zoals een vuex store. Hieronder ter illustratie een voorbeeld. We gaan hier verder ook niet dieper op in in deze cursus.

```
const moduleA = {
  state: () => ({ ... }),
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: () => ({ ... }),
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> `moduleA`'s state
store.state.b // -> `moduleB`'s state
```

Mutations

We zagen net al dat een mutation de state van onze vuex store zal updaten maar nog niet HOE we zo een mutation uitvoeren. Laten we opnieuw ons voorbeeld van daarnet erbij nemen:

```
export default createStore( options: {
  state: {
    turn: 1
  },
  mutations: {
    nextTurn(state) {
      state.turn += 1;
    }
  },
  actions: {},
  modules: {}
})
```

Onze store heeft een nextTurn mutation die de turn-waarde in onze store met 1 zal verhogen. Deze mutation kunnen we aanroepen binnen één van onze componenten op de volgende manier:

```
this.$store.commit('nextTurn');
```

De commit methode van onze store verwacht als eerste parameter altijd de naam van de mutation die we willen committen. Bovenstaande regel code zou de turn waarde in onze vuex store state met 1 verhogen.

We kunnen ook data meegeven aan de mutation vanuit de component moest dat nodig zijn. Net zoals bij events geven we na de naam nog een tweede parameter mee die de waarde (payload) bevat die we mee willen geven. Dit mag van elk datatype zijn maar meestal willen we een object meegeven zodat we meerdere gegevens door kunnen sturen.

Onze code zou er dan als volgt uit gaan zien:

```
mutations: {  
  nextTurn(state, payload) {  
    state.turn += payload.amount;  
  }  
},
```

```
nextTurn() {  
  this.$store.commit('nextTurn',{  
    amount: 10  
  });  
}
```

De turn waarde binnen onze state zou nu met 10 verhoogd worden.