




Sommige inhoud in deze presentatie is gegenereerd of ondersteund met behulp van ChatGPT en/of Copilot.



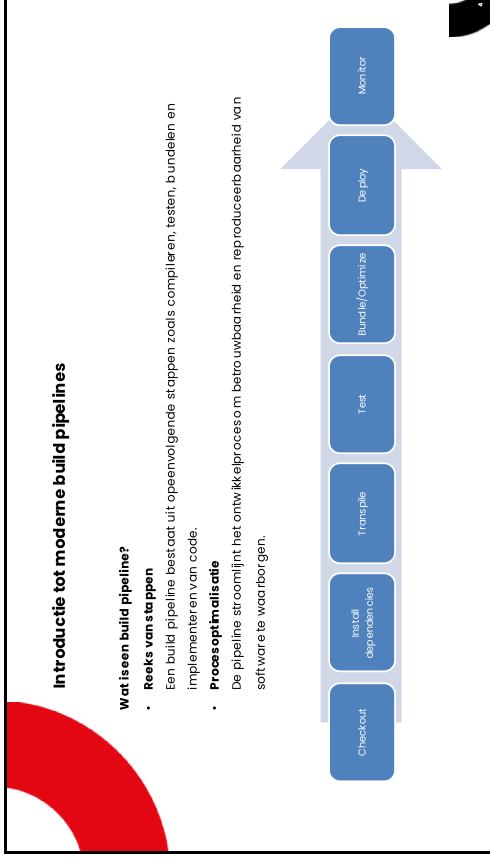
Agenda van de Presentatie

- Introductie tot moderne build pipelines
 - De rol van bundlers in het buildproces
 - Transpilors: broncode moderniseren en compatibel maken
 - Dev server voor snelle iteratie en testen
 - Optimalisatie: Workbox en Progressive Web Apps
 - Integratie en automatisering met CI/CD
 - Conclusie
- 



INTRODUCTIE TOT MODERNE BUILD PIPELINES

Een build pipeline is een geautomatiseerd proces dat broncode omzet in deploybare software. We bekijken waarom deze pipelines onmisbaar zijn in moderne softwareontwikkeling om consistentie, kwaliteit en snelheid te garanderen.



Stappen in een Build Pipeline voor Webapps

1. Code ophalen (Source stage)

Checkout van de repository (GitHub, Azure DevOps, ...).

Dependencies installeren (npm install / yarn install).

2. Compileren / Transpileren

TypeScript → JavaScript (via tsc of esbuild/Babel).

JSX → JavaScript (via Babel).

Sass/Less → CSS (indien gebruikt).

3. Testen

Unit tests (Jest, Vitest, Jasmine ...).

Integratie-/end-to-end tests (Cypress, Playwright ...).

Linting en static analysis (ESLint, Prettier).

4, Bundelen / Optimaliseren (Build stage)

Bundler maakt productiebuild:

- Tree-shaking (ongebruikte code weg).
- Code splitting (lazy loading).

- Minificatie / compressie.

- Tools: Vite (Rollup), Webpack, of Angular CLI.

5. Extra optimalisatie

- Genereren van service worker / caching strategieën (Workbox).

- Image/CSS optimalisatie (minify, purgeCSS, ...).

6. Implementeren / Deploy (Release stage)

- Artefacten (bv. /dist map) naar hosting sturen.

- Static site hosting (Netlify, Vercel, GitHub Pages).


- Cloud (Azure App Service, AWS S3+CloudFront, Firebase Hosting).

- Eventueel **staging** → **production** met approvals.

7. Monitoring & Feedback (Ops stage, DevOps)

- Applicatie monitoren (logs, performance, uptime).

- Feedback terug naar dev-team (issues, metrics).



Introductie tot moderne build pipelines

Waarom zijn build pipelines essentieel in softwareontwikkeling?

- **Verhoogde productiviteit**
Build pipelines automatiseren taken en verhogen de efficiëntie in softwareontwikkeling.
- **Vermindering van fouten**
Automatische processen minimaliseren menselijke fouten tijdens het bouwen en testen.
- **Versnelde releases**
Pipelines zorgen voor snellere en betrouwbaardere softwareleveringen.
- **Ondersteuning voor Agile en DevOps**
Build pipelines faciliteren continue integratie en continue levering in moderne ontwikkelingsmethoden.

🔥 Agile

Wat is het?

Een methode om software te ontwikkelen en plannen.

Focus: samenwerken, korte iteraties (sprints), snel feedback krijgen van de klant.

Bekend door: Scrum, Kanban, user stories, retrospectives.

Doel: sneller waarde leveren en flexibel kunnen inspelen op verandering.

👉 Agile = “Hoe we software *maken* en *organiseren*.”

🔧 DevOps

Wat is het?

Een cultuur en set praktijken die ontwikkeling (Dev) en operations (Ops) dichter bij elkaar brengt.

Focus: automatiseren van build, test, en deployment → **CI/CD**, monitoring, schaalbaarheid.


Bekend door: GitHub Actions, Azure Pipelines, Docker, Kubernetes, Infrastructure as Code.

Doel: software sneller en betrouwbaarder in productie krijgen.
👉 DevOps = “Hoe we software *opleveren* en in de praktijk draaiend houden.”



DE ROL VAN BUNDLERS IN HET BUILDPROCES

Bundlers combineren verschillende bronbestanden tot één of meerdere geoptimaliseerde bestanden. Dit vereenvoudigt het laden van applicaties en verbetert de performance.




De rol van bundlers in het buildproces

Het samenvoegen van bronbestanden:

- **Verwerking van bestanden**
Bundlers verwerken verschillende bronbestanden zoals JavaScript en CSS om ze te optimaliseren.
- **Afhankelijkheden resolutie**
Ze lossen afhankelijkheden op om correcte volgorde en functionaliteit te garanderen.
- **Efficiënte bundels creëren**
Bundels verbeteren laadtijd en caching voor snellere websiteprestaties.

Bundlers verwerken JavaScript, CSS en andere assets, resolutie afhankelijkheden en creëren efficiënte bundles die de laadtijd verminderen en caching optimaliseren.




De rol van bundlers in het buildproces

Populaire bundlers: Webpack, Rollup, Parcel, Esbuild

- **Webpack veelzijdigheid**
Webpack is een krachtige bundler die veelzijdigheid biedt voor complexe webprojecten en uitgebreide configuraties mogelijk maakt.
- **Rollup optimalisatie**
Rollup richt zich op ES modules en creëert efficiënte, kleine bundles voor optimale laadtijden en prestaties.
- **Parcel gebruiksgemak**
Parcel biedt een zero-config ervaring en is ideaal voor snelle ontwikkeling zonder ingewikkelde instellingen.
- **Esbuild snelheid**
Esbuild is extreem snel dankzij native implementatie in Go en richt zich op snelle bundling en transpiling, met minimale configuratie.

Webpack is veelzijdig en krachtig, Rollup focust op ES modules en kleine bundles, terwijl Parcel gebruiksgemak en zero-config biedt. Elk heeft unieke voordelen afhankelijk van het project.



De rol van bundlers in het buildproces

Hands-on met esbuild

- Voltooi de volgende tutorial: [Getting Started with esbuild](#) op Better Stack Community.
- Voeg een afbeelding toe aan je project.
- Maak gebruik van het esbuild content type [Data URI](#) voor de afbeelding en onderzoek hoe deze wordt gebundeld.

Oplossing: [hans-haert/Modern-Frontend-Build-Pipeline-at-Esbuild](#)

1) Hands-on esbuild-pipeline

Step 1 – Installeer esbuild

npm init -y

npm install --save-dev esbuild

Step 2 – Basis build-script toevoegen

Voeg in package.json aan "scripts" bijvoorbeeld:

```
"scripts": {
  "build": "esbuild src/index.ts --bundle --outfile=dist/bundle.js --minify --sourcemap"
}
```

Wat gebeurt er?

src/index.ts wordt **getranspiled** en gebundeld.

Bundling en minificatie vindt plaats.

Een **source map** wordt gegenereerd voor debugging.

Step 3 – uitbreiden met meerdere entry points / assets

Je kan esbuild ook als API in een JavaScript- of TypeScript-bestand gebruiken:

```
// build.js
require('esbuild').build({
  entryPoints: ['src/index.ts', 'src/another.ts'],
  bundle: true,
  outdir: 'dist',
  minify: true,
  sourcemap: true,
  loader: { '.png': 'file', '.css': 'css' },
}).catch(() => process.exit(1));
```

Daarna past je "build" script aan:

```
"build": "node build.js"
```

Stap 4 – integratie in CI-pipeline

Transpiler & bundler: door npm run build of node build.js.

Testing: je kan vóór of na de build een test-run invoegen (npm test), linten etc.

Output-artifact: de dist/-map kan je dan doorgeven naar het deploy-proces.

Voorbeeld met dataurl loader

Projectstructuur

/src

└─ index.js

└─ logo.png

index.js

```
import logo from './logo.png';
```

```
const img = document.createElement("img");
img.src = logo; // dit wordt een Data URL dankzij esbuild
document.body.appendChild(img);
```

build.js

```
import { build } from "esbuild";

build({
  entryPoints: ["src/index.js"],
  bundle: true,
  outdir: "dist",
  loader: {
    ".png": "dataurl" // <--- belangrijk: zet PNG om naar data URL
  },
}).catch(() => process.exit(1));

Run build
node build.js
```

🔥 Wat er gebeurt

esbuild leest logo.png.

In de output (dist/bundle.js) wordt logo geen pad naar een bestand, maar een **inline Data URL**, bv.:

```
var logo = "data:image/png;base64,iVBORw0KGgoAAAANSUheUgAA...";
```

Hierdoor heb je **geen apart logo.png** meer nodig bij deployment.

🚀 Wanneer handig?

Kleine afbeeldingen (icons, logos, svg's).


Vermijden van extra HTTP requests.

Snellere delivery voor simpele static sites.



TRANSPILERS: BRONCODE MODERNISEREN EN COMPATIBEL MAKEN

Transpilars zetten moderne broncode om naar oudere versies die in meer browsers en omgevingen werken. Dit zorgt voor bredere compatibiliteit zonder concessies aan moderne features.




Transpilars: broncode moderniseren en compatibel maken

Wat doet een transpiler?

- **Codevertaling tussen versies**
Een transpiler zet code van de nieuwste taalversies om naar oudere, behoudt functionaliteit en compatibiliteit.
- **Compatibiliteit waarborgen**
Transpilars zorgen dat nieuwe programmeerfuncties werken op oudere systemen zonder fouten.

Een transpiler vertaalt code van een nieuwere taalversie of syntaxis naar een oudere, bijvoorbeeld ES6+ naar ES5. Dit maakt gebruik van moderne syntax en functies mogelijk zonder compatibiliteitsproblemen.



Transpilers: broncode moderniseren en compatibel maken

Veelgebruikte transpilars: Babel, TypeScript

- **Babeltranspiler**
Babel zet moderne JavaScript om naar versies die compatibel zijn met oudere browsers en omgevingen.
- **TypeScriptfunctionalies**
TypeScript voegt type-checking en extra functionalies toe bovenop JavaScript voor betere codekwaliteit.

Babel transpileert moderne JavaScript naar compatibele versies, terwijl TypeScript ook type-checking en extra functionaliteiten biedt door bovenop JavaScript te bouwen.

Transpiliers: broncode moderniseren en compatibel maken

Hands-on TypeScript/Babel

- Bouw verder op je vorige oefening (Hands-on met esbuild).
- Breek je project uit zodat je TypeScript gebruikt in plaats van gewone JavaScript.
- Integreer daarbij Babel om extra transpilatiestappen mogelijk te maken (bv. ondersteuning voor oudere browsers).
- Zorg ervoor dat je bestaande functionaliteit behouden blijft, inclusief de afbeelding die via het esbuild content type datauri is toegevoegd.
- Onderzoek hoe de combinatie van TypeScript, Babel en esbuild samenwerkt in de build pipeline en bespreek dit kort.

Oplossing: <https://noet/Modern-Frontend-Build-Pipeline-ot-typescript-babel>

Projectstructuur

/esbuild-demo

```
├── src/
│   ├── index.ts
│   └── logo.png
├── build.js
├── index.html
└── package.json
```

1. package.json

Na initialisatie (npm init -y) en installatie:

npm install --save-dev esbuild typescript @babel/core @babel/preset-env esbuild-plugin-babel package.json (scripts):

```
{
  "name": "esbuild-demo",
  "version": "1.0.0",
  "scripts": {
    "build": "node build.js"
  },
  "devDependencies": {
    "esbuild": "^0.24.0",
    "typescript": "^5.0.0",
    "@babel/core": "^7.23.0",
    "@babel/preset-env": "^7.23.0",
    "esbuild-plugin-babel": "^0.2.3"
  }
}
```

2. TypeScript code (src/index.ts)

```
import logo from "/logo.png";

function greet(name: string): string {
  return `Hello, ${name}!`;
}

document.body.innerHTML = `<h1>${greet("TypeScript + Babel + esbuild")}</h1>`;

// Voeg logo toe (inline via dataurl loader)
const img = document.createElement("img");
img.src = logo;
document.body.appendChild(img);
```

3. esbuild config met Babel (build.js)

```
import { build } from "esbuild";
import babel from "esbuild-plugin-babel";

build({
  entryPoints: ["src/index.ts"],
  bundle: true,
  outdir: "dist",
  sourcemap: true,
  loader: {
    ".png": "dataurl" // afbeelding inline als Data URL
  },
  plugins: [
    babel({
      config: {
        presets: ["@babel/preset-env"], // transpile naar oudere browsers
      }
    })
  ],
  platform: "browser",
  target: ["es2017"], // esbuild transpile-target
}).catch(() => process.exit(1));
```

4. HTML-bestand (index.html)

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
<meta charset="UTF-8">
<title>esbuild + TS + Babel demo</title>
</head>
<body>
<script src="dist/index.js"></script>
</body>
</html>
```

5. Runnen

npm run build

Open index.html in de browser → je ziet de begroeting + logo (inline via Data URL).

Analyse (wat de student moet onderzoeken)

TypeScript → omgezet naar JavaScript door esbuild.

Babel plugin → extra transpilatiestap voor compatibiliteit (bv. oudere browsers).


DataURL loader → logo.png wordt inline opgenomen als een lange data:image/png;base64,... string in dist/index.js.

Pipeline → TypeScript → esbuild → Babel → bundel (inclusief inline afbeelding).



DEV SERVER VOOR SNELLE ITERATIE EN TESTEN


Dev servers versnellen het ontwikkelproces door directe feedback te geven bij codewijzigingen. Ze ondersteunen live reload en hot module replacement voor een soepele workflow.



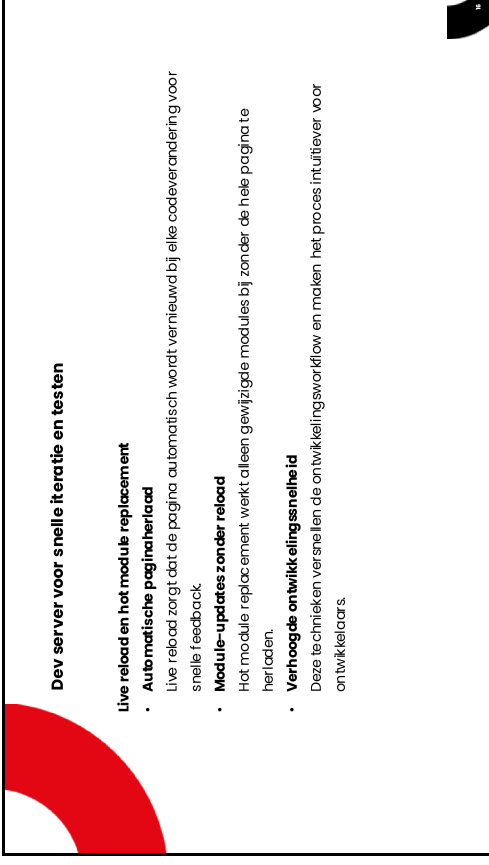
Dev server voor snelle iteratie en testen

Waarom een dev server gebruiken?

- **Verminderde wachttijd**
Een dev server vermindert de tijd tussen code wijzigingen en resultaatweergave drastisch voor snellere ontwikkeling.
- **Hogere productiviteit**
Snelle feedback van een dev server verhoogt de efficiëntie en productiviteit van ontwikkelaars.
- **Snelle foutopsporing**
Ontwikkelaars kunnen bugs snel opsporen en oplossen dankzij directe resultaten via de dev server.



Een dev server verlaagt de wachttijd tussen het aanbrengen van wijzigingen en het zien van het resultaat, wat de productiviteit verhoogt en snelle foutopsporing mogelijk maakt.




Dev server voor snelle iteratie en testen

Live reload en hot module replacement

- **Automatische paginaherlaad**
Live reload zorgt dat de pagina automatisch wordt vernieuwd bij elke codeverandering voor snelle feedback.
- **Module-updates zonder reload**
Hot module replacement werkt alleen gewijzigde modules bij zonder de hele pagina te herladen.
- **Verhoogde ontwikkelingssnelheid**
Deze technieken versnellen de ontwikkelingsworkflow en maken het proces intuïtiever voor ontwikkelaars.

Live reload herlaadt de pagina automatisch bij veranderingen, terwijl hot module replacement alleen gewijzigde modules bijwerkt zonder volledige reload. Dit maakt ontwikkeling sneller en intuïtiever.

Dev server voor snelle iteratie en testen		
Vergelijking tussen Dev server met esbuild en Vite		
Feature	esbuild serve	Vite
Simpele static server	✔	✔
Bundling	✔	✔ (met Rollup voor productie)
HMR (auto refresh zonder hele reload)	✗	✔
Ecosysteem & plugins	beperkt	groot
Aanbevelen voor echte projecten	• kleine demo's/prototypes	✔ ja



Dev server voor snelle iteratie en testen

Hands-on Vite

- Maak een Vite project aan met als template "Vanilla-ts"
- Start de dev server
- Maak een typescript functie "greet" aan en voeg de response toe aan het HTML document.
- Voeg een afbeelding toe.
- Stel Vite zo in dat de afbeelding wordt gebundeld met een DATAURL-loader.

Oplossing: basemart/Modern-Frontend-Build-Pipeline-at-Vite

Het mooie is dat **Vite** intern esbuild gebruikt voor transpilen, dus je verliest geen snelheid.
We houden **TypeScript** + **Babel** + **DataURL loader** erin.

Projectstructuur

```
/vite-demo
├-- src/
|   ├── main.ts
|   └-- logo.png
├-- index.html
├-- vite.config.ts
└-- package.json
```

1. Init project + dependencies

```
npm create vite@latest vite-demo -- --template vanilla-ts
cd vite-demo
npm install
# Babel erbij indien nodig:
npm install --save-dev @babel/core @babel/preset-env
```

2. TypeScript code (src/main.ts)

```
import logo from "/logo.png";

function greet(name: string): string {
  return `Hello, ${name}!`;
}

document.body.innerHTML = `<h1>${greet("Vite + TS + Babel")}</h1>`;

// Voeg logo toe (inline via dataurl loader)
const img = document.createElement("img");
img.src = logo;
document.body.appendChild(img);
```

3. Vite config (vite.config.ts)

👉 Hier zetten we de **DataURL loader** (voor kleine afbeeldingen).

```
import { defineConfig } from "vite";
```

```
export default defineConfig({
  build: {
```

```

target: "es2017",
},
assetsInclude: ["**/*.png"], // laat Vite weten dat .png bestanden assets zijn
esbuild: {
  loader: "ts", // transpile TS met esbuild
  target: "es2017",
}
});

```

i Vite behandelt kleine afbeeldingen (<4kb) automatisch al als **DataURL**.
 Wil je altijd inline (ook grote)? Dan kan je `assetsInlineLimit` aanpassen:

```

build: {
  assetsInlineLimit: 1000000 // alles inline als DataURL (in bytes)
}

```

4. HTML (index.html)

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Vite + TS + Babel demo</title>
</head>
<body>
  <script type="module" src="/src/main.ts"></script>
</body>
</html>

```

5. Run dev server

npm run dev

Je krijgt een **snelle dev server met HMR** (geen browser reload meer nodig).

Hoe zit Babel hierin?

Vite gebruikt **esbuild** voor dev en transpilen.

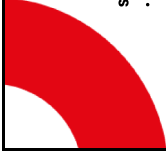
Voor productie (Rollup) kan je Babel inzetten als extra stap, bv. voor polyfills of oudere browsers.

In vite.config.ts kan je Babel via een plugin toevoegen, maar vaak is dat **niet nodig** tenzij je **IE11** of heel oude browsers wilt ondersteunen.



OPTIMALISATIE: WORKBOX EN PROGRESSIVE WEB APPS


Workbox helpt bij het implementeren van service workers voor caching en offline gebruik, essentieel voor Progressive Web Apps (PWA's). Dit verbetert performance en gebruikerservaring.




Optimalisatie: Workbox en Progressive Web Apps

Service workers en caching

- **Achtergrondwerkersfunctie**
Service workers opereren op de achtergrond en beheren netwerkverzoeken onafhankelijk van de hoofdthread.
- **Contentcaching voor snelheid**
Cached content zorgt voor snelle toegang tot data en verbetert de laadtijden van de applicatie.
- **Offline beschikbaarheid**
Caching maakt offline gebruik van de app mogelijk, wat de betrouwbaarheid versterkt.




Service workers werken op de achtergrond en beheren netwerkverzoeken, waardoor content kan worden gecached voor snelle toegang en offline beschikbaarheid, wat de betrouwbaarheid van de app verhoogt.




Optimalisatie: Workbox en Progressive Web Apps

Introductie tot Workbox

- **Eenvoudige service worker configuratie**
Workbox maakt het eenvoudig om service workers te configureren zonder complexe code te schrijven.
- **Geautomatiseerde cachingstrategieën**
Workbox automatiseert cachingstrategieën voor efficiënte opslag en snelle toegang tot webinhoud.
- **Offline functionaliteiten**
Workbox ondersteunt offline functionaliteiten zodat applicaties blijven werken zonder internetverbinding.



Workbox is een verzameling libraries die het eenvoudig maken service workers te configureren en te beheren. Het automatiseert cachingstrategieën en offline functionaliteiten zonder complexe code.



Optimalisatie: Workbox en Progressive Web Apps


Generatie van service worker met Workbox

Optie 1: generateSW


- Workbox maakt volledig automatisch een service worker.
- Alle logica voor preaching, updates en runtime caching kan via configuratie bepaald worden.
- Geschikt als je geen eigen service worker-code nodig hebt.

Optie 2: injectManifest

- Je schrijft zelf een custom service worker. Workbox injecteert enkel de preaching manifest (—WB_MANIFEST).
- Meer flexibiliteit; je kan eigen cachingstrategieën, event handlers en push-notificaties toevoegen.
- Handig bij complexere projecten of wanneer je volledige controle nodig hebt.



[What is Workbox? | Chrome for Developers](#)



Optimalisatie: Workbox en Progressive Web Apps

Hands-on Workbox & esbuild

- Maak met behulp van esbuild en Workbox een serviceworker aan via injectManifest. Voeg een lijst met afbeeldingen toe aan je app en controleer of deze correct worden geprecached. Voeg vervolgens een application manifest toe en test of de app installeerbaar is en offline functioneert.

Oplossing: <https://modern-frontend-build-pipeline.at.esbuild-workbox>

Hands-on Workbox & VITE

- Gebruik Vite met de PWA-plugin om een installeerbare app te creëren die ook offline gebruikt kan worden.

Oplossing: <https://modern-frontend-build-pipeline.at/vite-workbox>

✓ Voorbeeldoplossing 1 — Esbuild + Workbox (injectManifest)

1. Install dependencies

npm init -y


npm install esbuild workbox-build --save-dev

2. Appstructuur

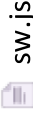
```
/project
/src
  index.html
  index.ts
  images/
    img1.png
    img2.png
```

```
sw.js  
build.js  
manifest.json
```

3. Esbuild config

```
 build.js  
import * as esbuild from "esbuild";  
import { injectManifest } from "workbox-build";  
  
// Step 1: Bundel de app  
await esbuild.build({  
  entryPoints: ["src/index.ts"],  
  bundle: true,  
  outfile: "dist/bundle.js",  
  loader: {  
    ".png": "dataurl" // afbeelden embedden  
  }  
});  
  
// Step 2: Workbox injectManifest  
await injectManifest({  
  swSrc: "sw.js",  
  swDest: "dist/sw.js",  
  globDirectory: "dist",  
  globPatterns: ["**/*.{html,js,png,json}"]  
});  
  
console.log("✅ Build + service worker klaar!");
```


4. Service Worker

```
 sw.js
import { precacheAndRoute } from "workbox-precaching";
import { clientsClaim } from "workbox-core";

self.skipWaiting();
clientsClaim();

// Workbox injecteert __WB_MANIFEST tijdens build
precacheAndRoute(self.__WB_MANIFEST);
```

5. Manifest.json (Application Manifest)

```
 manifest.json
{
  "name": "Esbuild Workbox App",
  "short_name": "EsbuildPWA",
  "start_url": ".",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#317EFB",
  "icons": [
    {
      "src": "img1.png",
      "sizes": "192x192",
      "type": "image/png"
    },
  ],
```

```
{
  "src": "img2.png",
  "sizes": "512x512",
  "type": "image/png"
}
]
```

6. Resultaat

Afbeeldingen (img1.png, img2.png) worden **geprecached**.

App is **offline bruikbaar**.


Dankzij manifest.json is de app **installeerbaar**.

✓ Voorbeeldoplossing 2 — Vite + PWA Plugin

1. Install dependencies

```
npm create vite@latest vite-pwa-app
cd vite-pwa-app
npm install
npm install vite-plugin-pwa workbox-window --save-dev
```

2. Vite config

```
 vite.config.ts
import { defineConfig } from "vite";
import { VitePWA } from "vite-plugin-pwa";

export default defineConfig({
```


```

plugins: [
  VitePWA({
    strategies: "injectManifest",
    srcDir: "src",
    filename: "sw.ts",
    manifest: {
      name: "Vite PWA App",
      short_name: "VitePWA",
      start_url: ".",
      display: "standalone",
      background_color: "#ffffff",
      theme_color: "#317EFB",
      icons: [
        {
          src: "pwa-192x192.png",
          sizes: "192x192",
          type: "image/png"
        },
        {
          src: "pwa-512x512.png",
          sizes: "512x512",
          type: "image/png"
        }
      ]
    },
    injectManifest: {
      globPatterns: ["**/*.js,css,html,png,svg,json"]
    }
  })
]

```

```
]
});
```


3. Service Worker

```
 src/sw.ts
import { precacheAndRoute } from "workbox-precaching";
import { clientsClaim } from "workbox-core";

self.skipWaiting();
clientsClaim();

// precache lijst wordt hier door Vite + Workbox geïnjecteerd
precacheAndRoute(self.__WB_MANIFEST);
```

4. Client integratie

```
 src/main.ts
import { Workbox } from "workbox-window";

if ("serviceWorker" in navigator) {
  const wb = new Workbox("/sw.js");

  wb.addEventListener("waiting", () => {
    console.log("Nieuwe versie klaar, herladen...");
    wb.messageSkipWaiting();
    window.location.reload();
  });
}
```

```
wb.register();  
}  
  
document.querySelector<HTMLDivElement>('#app')!.innerHTML = `  
<h1>Vite + PWA Demo</h1>  
  
  
`;  
`;
```

5. Resultaat

De afbeeldingen (pwa-192x192.png, pwa-512x512.png) worden **geprecached**.


App werkt **offline**.

App is **installeerbaar** dankzij het manifest.



INTEGRATIE EN AUTOMATISERING MET CI/CD


CI/CD automatiseert het bouwen, testen en implementeren van software. Dit zorgt voor een betrouwbare, snelle en herhaalbare releasecyclus die fouten vermindert en kwaliteit garandeert.




Integratie en automatisering met CI/CD

Wat is CI/CD en waarom is het belangrijk?

- **Continuous Integration (CI)**
CI automatiseert het samenvoegen en testen van code om integratiefouten snel te detecteren.
- **Continuous Deployment/Delivery (CD)**
CD automatiseert het uitrollen van geteste code naar productie voor snelle releases.
- **Belang van CI/CD**
CI/CD maakt snelle feedback en frequente, betrouwbare software-updates mogelijk.




Continuous Integration (CI) en Continuous Deployment/Delivery (CD) zijn processen die codewijzigingen automatisch integreren, testen en uitrollen. Dit maakt snelle feedback en frequente releases mogelijk.




Integratie en automatisering met CI/CD

Typische CI/CD workflow in een build pipeline:

- **Code Commit**
Ontwikkelaars voegen code toe aan een gedeelde repository als eerste stap in de CI/CD workflow.
- **Automatische Build en Test**
Na code commit start een geautomatiseerd proces van build en testen om fouten vroegtijdig te detecteren.
- **Statische Analyse en Beveiligingscans**
Code ondergaat statische analyse en beveiligingscans om kwetsbaarheden en kwaliteitsproblemen te identificeren.
- **Deployment naar Omgevingen**
De laatste stap is geautomatiseerde deployment naar test- of productieomgevingen voor snelle release.



Een typische workflow omvat code commit, automatische build en test, statische analyse, beveiligingscans en uiteindelijk deployment naar productie- of testomgevingen, alles geautomatiseerd via pipelines.




Integratie en automatisering met CI/CD

Populaire CI/CD tools: Jenkins, GitHub Actions, GitLab CI

- **Jenkins's Open Source**
Jenkins is een krachtige open-source CI/CD tool met een breed scala aan plugins voor flexibiliteit en aanpassing.
- **GitHub Actions integratie**
GitHub Actions integreert naadloos met GitHub repositories voor gestroomlijnde workflows en automatisering.
- **GitLab CI Pipelines**
GitLab CI biedt geïntegreerde pipelines binnen GitLab voor efficiënte continue integratie en levering.
- **Azure Pipelines**
Azure Pipelines biedt CI/CD functionaliteit voor meerdere talen en platformen en integreert met zowel Microsoft- als externe tools.

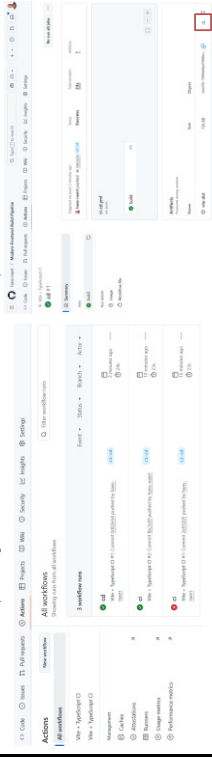
Jenkins is een krachtige open-source tool met veel plugins, GitHub Actions integreert naadloos met GitHub repositories, GitLab CI biedt geïntegreerde pipelines binnen GitLab, en Azure Pipelines biedt CI/CD-functionaliteit voor meerdere talen en platformen met integratie van zowel Microsoft- als externe tools. De keuze hangt af van de projectbehoeften.



Integratie en automatisering met CI/CD

Hands-on GitHub Actions met Vite + Typescript

- Basisopdracht CI:**
Maak een eenvoudige GitHub Actions workflow die automatisch de vite-projectbuild uitvoert en de Typescript-code compileert bij elke push naar de repository.
- Uitbreiding opdracht CD:**
Breid de pipeline uit met **testen** en **artifact-opslag**.
Oplossing: [hans-naert/Modern-Frontend-Build-Pipeline-at-ci-cd](#)



◆ Hands-on Opdracht: GitHub Actions met Vite + TypeScript

Deel 1: Basis CI

Doel:

Maak een workflow die de **TypeScript-code controleert** en een **Vite-build** uitvoert.

Stappen

Maak een nieuwe repository aan en initialiseer een **Vite + TypeScript project**.

npm create vite@latest my-app -- --template react-ts

cd my-app

npm install

Voeg in package.json de nodige scripts toe:

```
{
  "scripts": {
    "dev": "vite",
    "build": "vite build",
```

```
"preview": "vite preview",  
"type-check": "tsc --noEmit",  
"test": "vitest run"  
}
```

```
}
```

Maak een map **.github/workflows/** en voeg **ci.yml** toe.

Voorbeeldoplossing (ci.yml):

name: Vite + TypeScript CI

on:

 push:

 branches: ["main"]

 pull_request:

 branches: ["main"]

jobs:

 build:

 runs-on: ubuntu-latest

 steps:

 - name: Checkout repository

 uses: actions/checkout@v4

 - name: Set up Node.js

 uses: actions/setup-node@v4

 with:

 node-version: "18"

 - name: Install dependencies

run: npm install

- name: TypeScript check
run: npm run type-check

- name: Build project
run: npm run build

Deel 2: Uitbreidingsopdracht

Breid de pipeline uit met **testen** en **artifact-opslag**.

Vereisten

Voeg een **test stap** toe die npm test draait (bijvoorbeeld met **Vitest**).

Laat de build output (dist/) bewaren als **artifact**, zodat je deze achteraf kan downloaden via de Actions-tab.

Uitgebreide oplossing (ci.yml):

name: Vite + TypeScript CI/CD

on:

push:

branches: ["main"]

pull_request:

branches: ["main"]

jobs:

build-and-test:

runs-on: ubuntu-latest

steps:

- name: Checkout repository

```
uses: actions/checkout@v4

- name: Set up Node.js
  uses: actions/setup-node@v4
  with:
    node-version: "18"

- name: Install dependencies
  run: npm install

- name: TypeScript check
  run: npm run type-check

- name: Run tests
  run: npm test

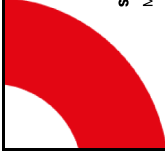
- name: Build project
  run: npm run build

- name: Upload build artifacts
  uses: actions/upload-artifact@v4
  with:
    name: vite-dist
    path: dist/
```

👉 Na deze uitbreiding kunnen studenten zien:
Of hun **TypeScript correct compileert**.
Of hun **testen slagen**.

Dat hun **build artifacts (dist/)** beschikbaar zijn om te downloaden.

CONCLUSIE



Conclusie

Samenstelling van build pipelines

Moderne build pipelines combineren bundlers, transpilars, dev servers, optimalisatietools en CI/CD voor optimale softwareontwikkeling.

Voordelen van integratie

Goede integratie van deze tools maakt snellere, efficiëntere en betrouwbaare softwarelevering door teams mogelijk.

