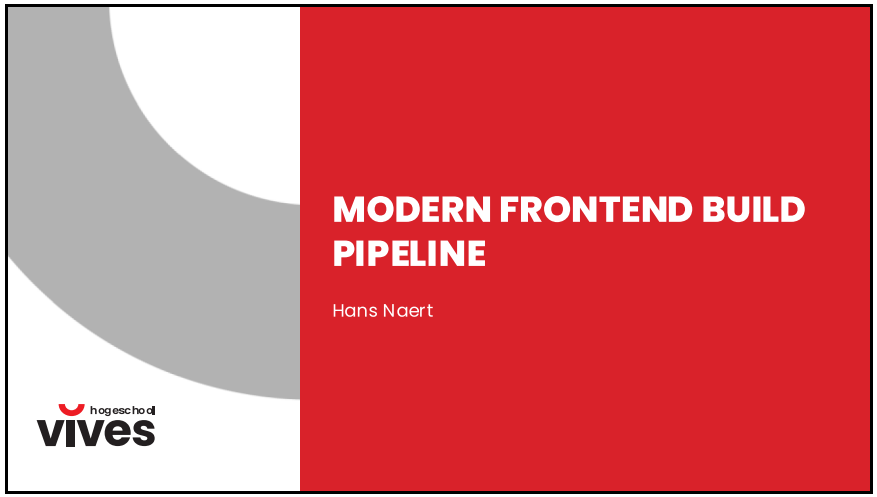
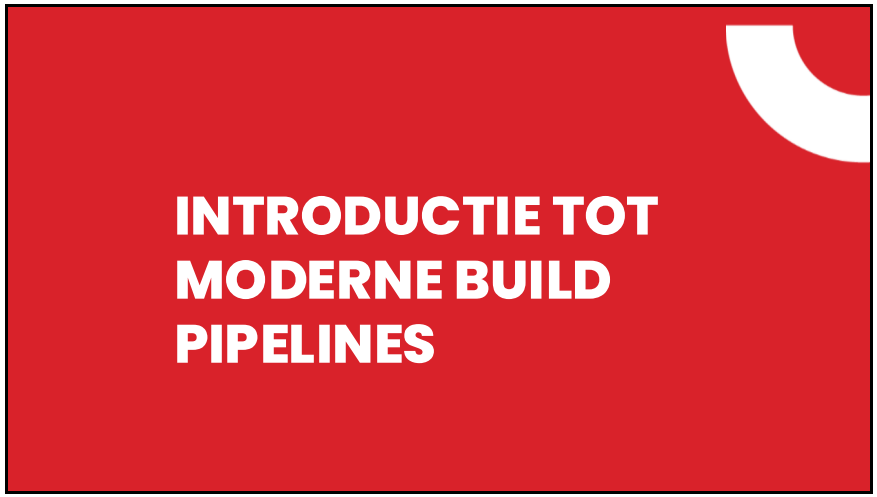


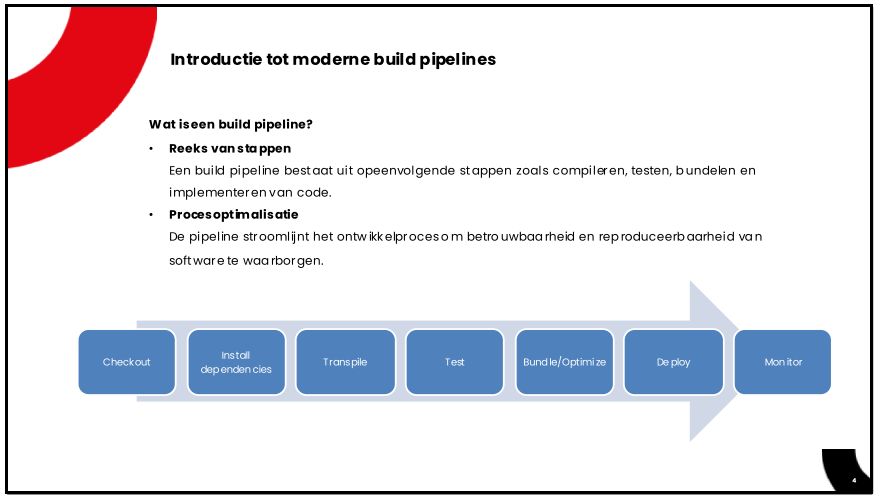
Sommige inhoud in deze presentatie is gegenereerd of ondersteund met behulp van ChatGPT en/of Copilot.

Agenda van de Presentatie

- Introductie tot moderne build pipelines
- De rol van bundlers in het buildproces
- Transpilers: 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



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)

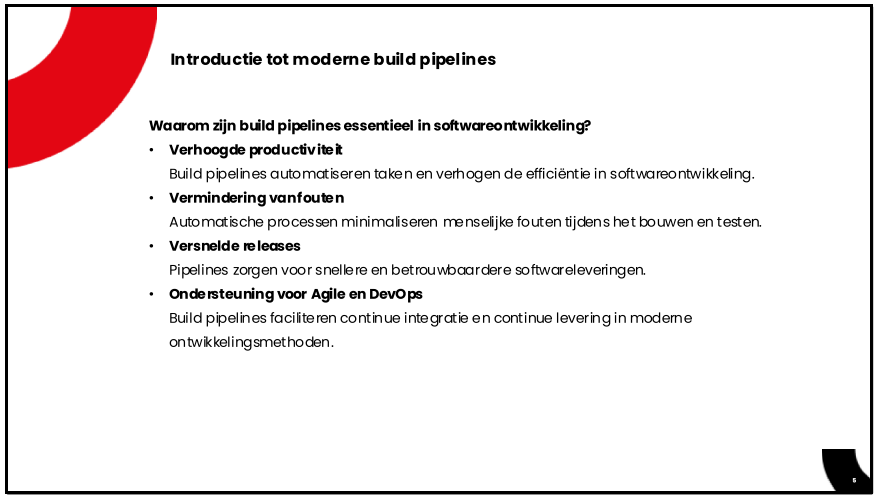
Artefacten (bv. /dist map) naar hosting sturen.

- 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).



⚡ **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?

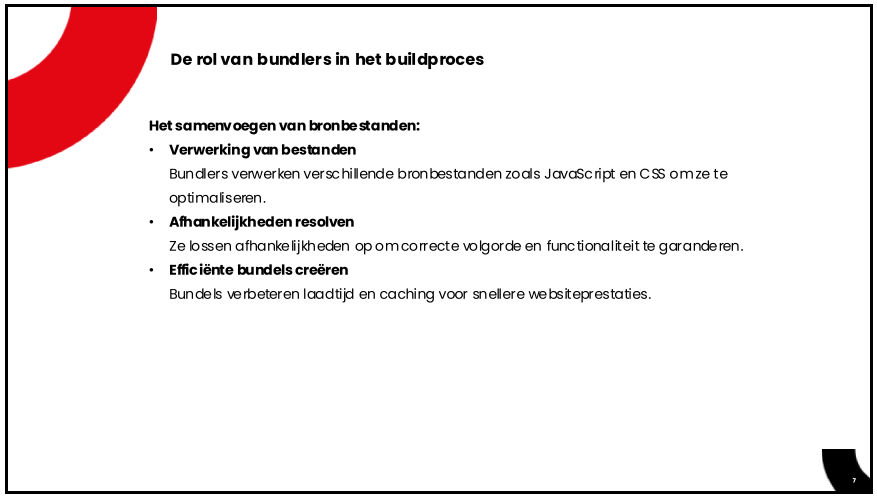
Een **cultuur en set praktijken** die ontwikkeling (**Dev**) en operations (**Ops**) dichter bij elkaar brengt.

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.”**

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



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

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.

1) Hands-on esbuild-pipeline

Stap 1 – Installeer esbuild

npm init -y

npm install --save-dev esbuild

Stap 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.

Stap 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.:

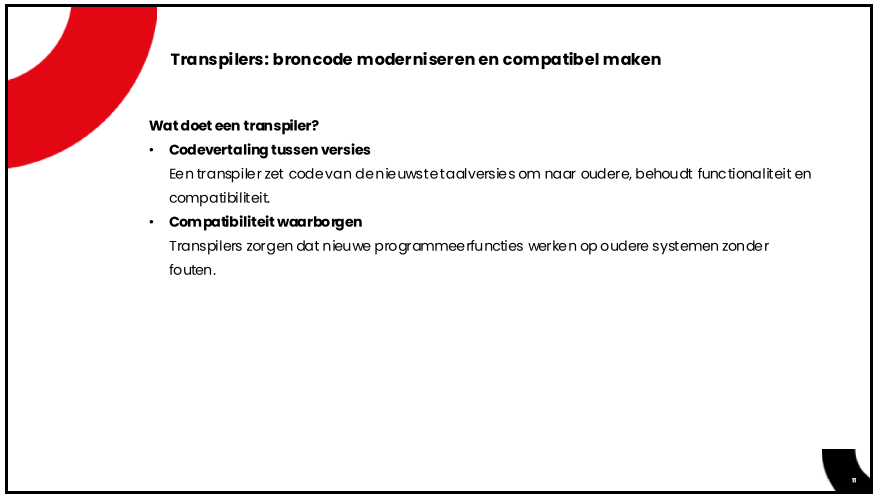
```
var logo = "...";
```

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

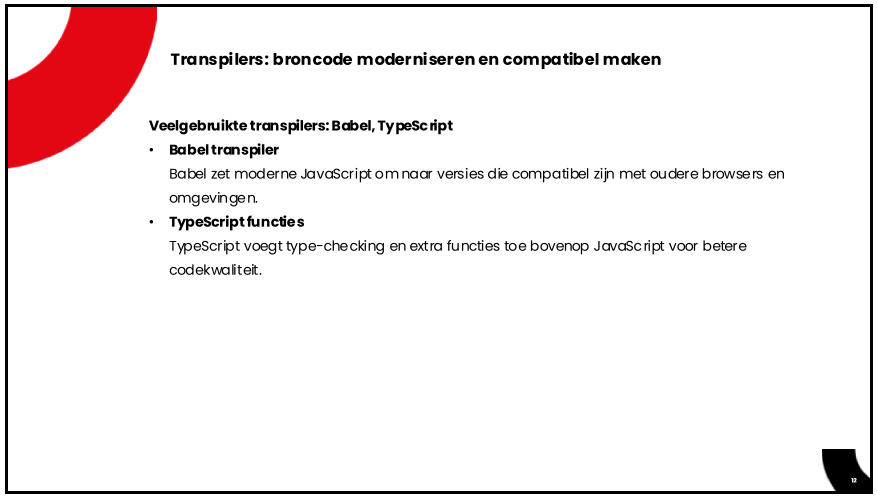
📌 **Wanneer handig?**

Kleine afbeeldingen (icons, logos, svg's).
Vermijden van extra HTTP requests.
Snellere delivery voor simpele static sites.

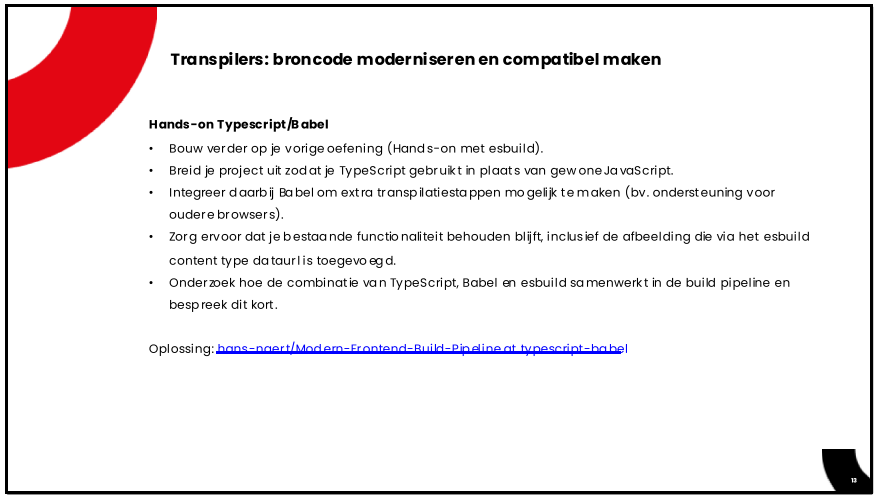
Transpilers zetten moderne broncode om naar oudere versies die in meer browsers en omgevingen werken. Dit zorgt voor bredere compatibiliteit zonder concessies aan moderne features.



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.



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



📁 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">
```

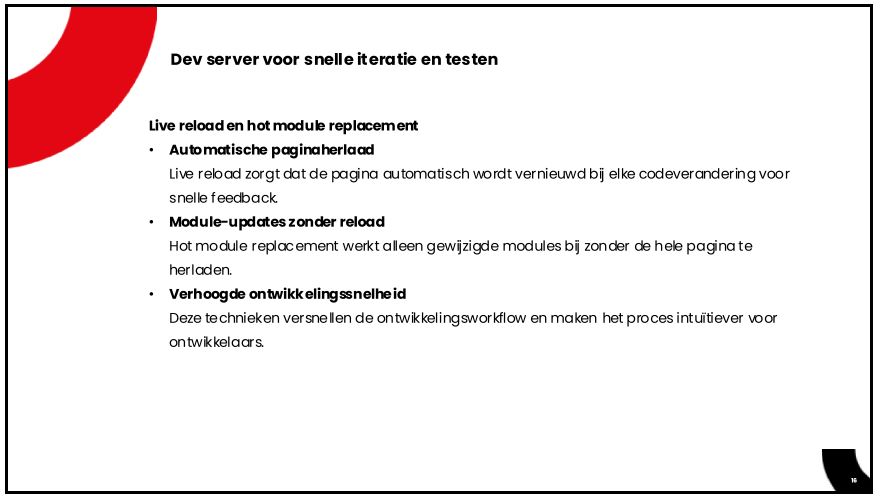
```
<head>
  <meta charset="UTF-8">
  <title>esbuild + TS + Babel demo</title>
</head>
<body>
  <script src="dist/index.js"></script>
</body>
</html>
```

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 ... string in dist/index.js.
Pipeline → TypeScript → esbuild → Babel → bundel (inclusief inline afbeelding).

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

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.



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
Aanbevolen voor echte projecten	🔸 kleine demo's/prototypes	✅ ja

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",
 }
});
```

ℹ️ 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)

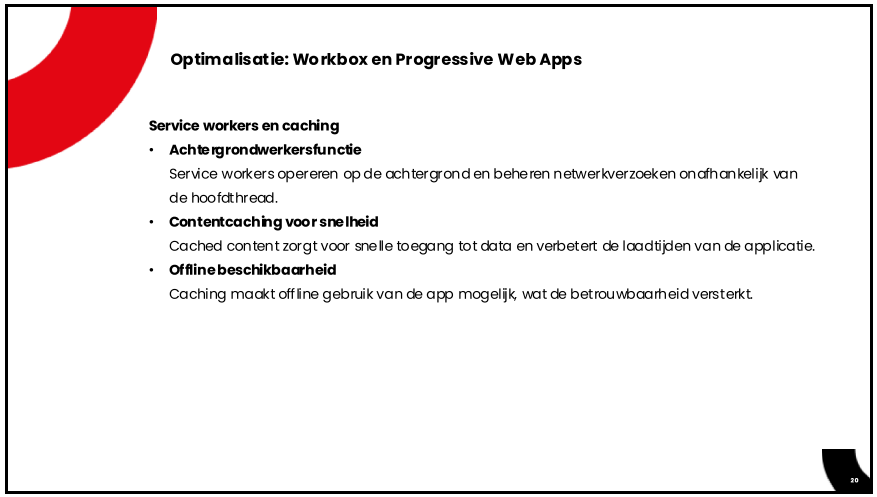
```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>Vite + TS + Babel demo</title>
</head>
<body>
 <script type="module" src="/src/main.ts"></script>
</body>
</html>
```

5. Run dev server

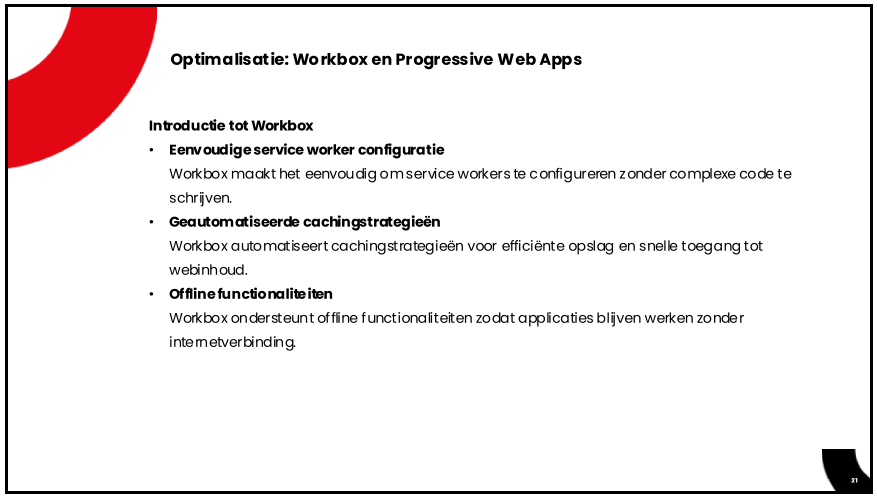
npm run dev
Je krijgt een **snelle dev server met HMR** (geen browser reload meer nodig).

🔎 **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.

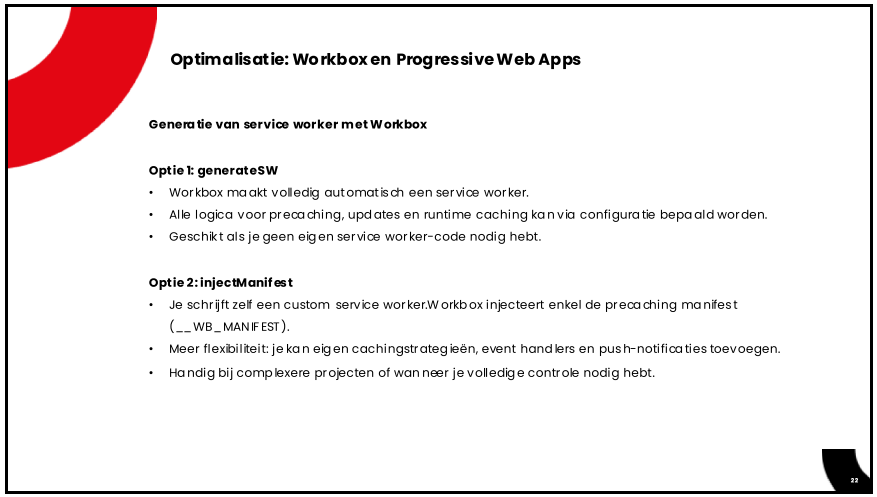
Workbox helpt bij het implementeren van service workers voor caching en offline gebruik, essentieel voor Progressive Web Apps (PWA's). Dit verbetert performance en gebruikerservaring.



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.



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.



[What is Workbox? | Chrome for Developers]

✅ **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";

// Stap 1: Bundel de app
await esbuild.build({
  entryPoints: ["src/index.ts"],
  bundle: true,
  outfile: "dist/bundle.js",
  loader: {
    ".png": "dataurl" // afbeeldingen embedden
  }
});

// Stap 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();
  });
```

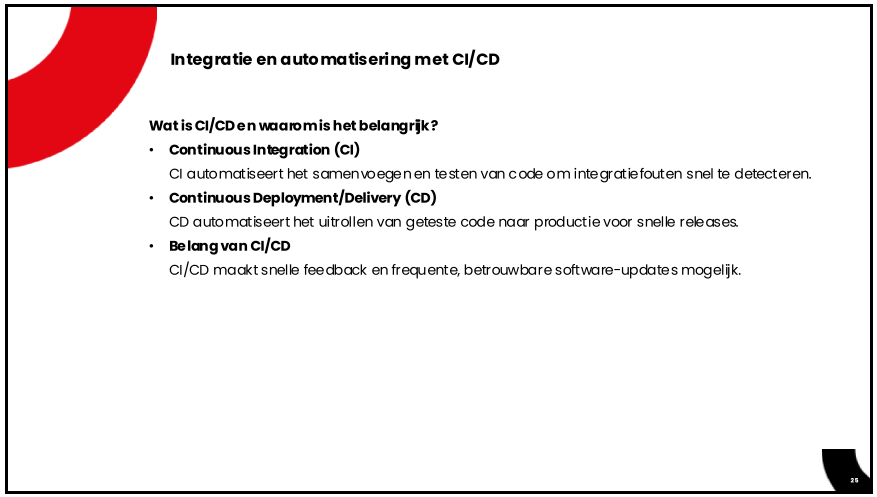
```
  wb.register();
}

document.querySelector<HTMLDivElement>('#app')!.innerHTML = `
  <h1>Vite + PWA Demo</h1>
  <img src="/pwa-192x192.png" />
  <img src="/pwa-512x512.png" />
`;
```

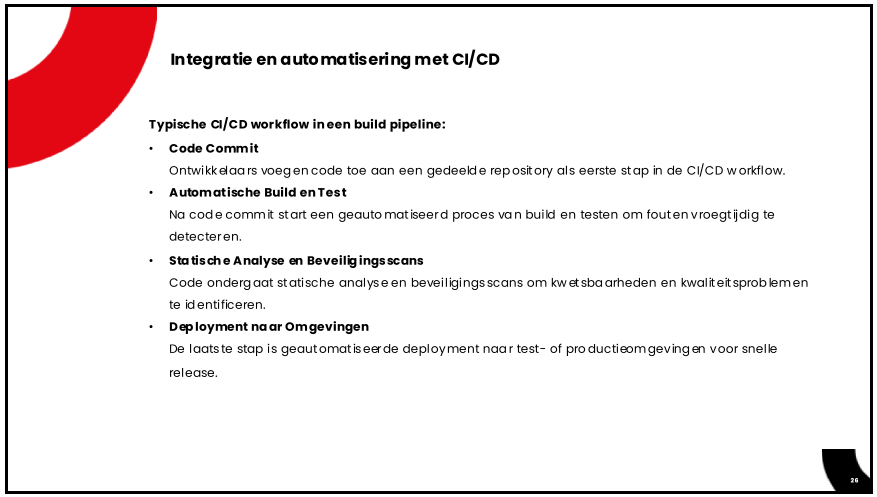
5. Resultaat

De afbeeldingen (pwa-192x192.png, pwa-512x512.png) worden **geprecached**.
App werkt **offline**.
App is **installeerbaar** dankzij het manifest.

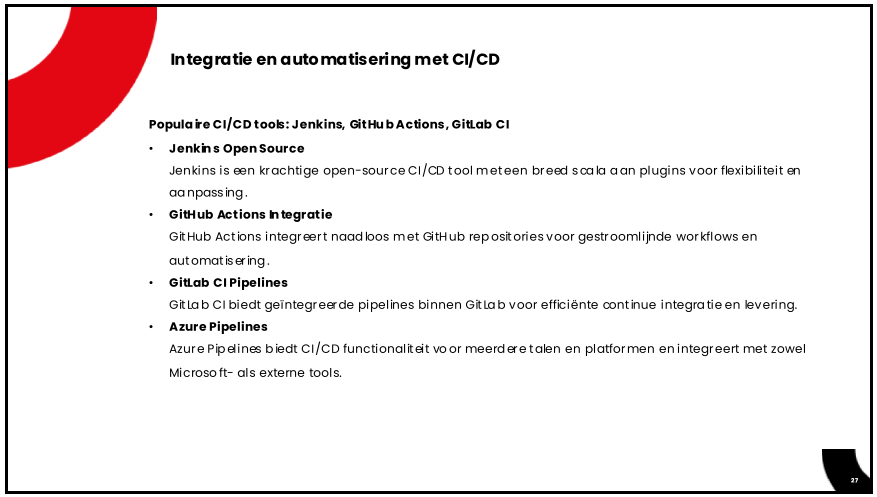
CI/CD automatiseert het bouwen, testen en implementeren van software. Dit zorgt voor een betrouwbare, snelle en herhaalbare releasecyclus die fouten vermindert en kwaliteit garandeert.



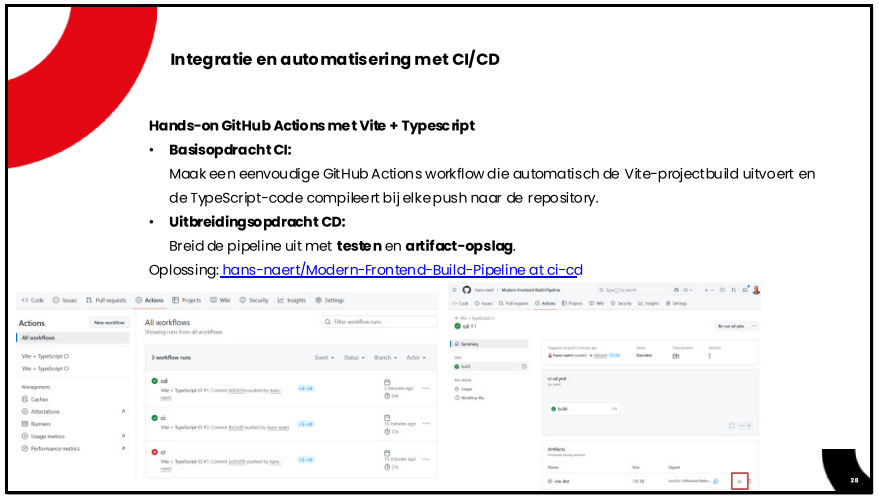
Continuous Integration (CI) en Continuous Deployment/Delivery (CD) zijn processen die codewijzigingen automatisch integreren, testen en uitrollen. Dit maakt snelle feedback en frequente releases mogelijk.



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



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.



- **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, transpilers, dev servers, optimalisatietools en CI/CD voor optimale softwareontwikkeling.

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