Herzlich Willkomen zum Workshop: Micro-Frontends mit Webpack - Module Federation

Von

Maik Roth

Agenda

• Micro-Frontends: Quiz, Vergleich, Komposition, H. vs. V., Ansätze, Beispiele	~ 10 min
• Webpack – Module Federation: Vergleich, Vorteile, Konzepte, Funktion, Erstellung	~ 10 min
• Aufgabe 1: Anwendungen Erstellen	~ 20 min
• Konfiguration von Webpack und Module Federation	~ 10 min
• Pause	10 min
• Aufgabe 2: Anwendungen Exposen und laden	~ 10 min
• Kommunikation und Best Practices	~ 20 min
• Aufgabe 3: Kommunikation und Lazy Loading implementieren	~ 10 min
• Takeaways	~ 5 min
• Fragen	~ 5 min
Gesamt	~ 120 min

Frage 1: Was sind Micro-Frontends?

a. Ein Ansatz zur Modularisierung von Backend-Diensten.
b. Ein Designmuster für Microservices.
c. Eine Architektur, die es ermöglicht, Frontend-Anwendungen in kleinere, unabhängige Teile zu zerlegen.
d. Ein Framework zur Erstellung von monolithischen Anwendungen.

Antwort: c

Frage 2: Welche Vorteile bieten Micro-Frontends?

a. Erhöhte Skalierbarkeit und Unabhängigkeit der Entwicklungsteams
b. Reduzierte Ladezeiten durch kleinere Bundles
c. Erhöhte Abhängigkeiten von monolithischen Architekturen
d. Vereinfachte Datenverwaltung

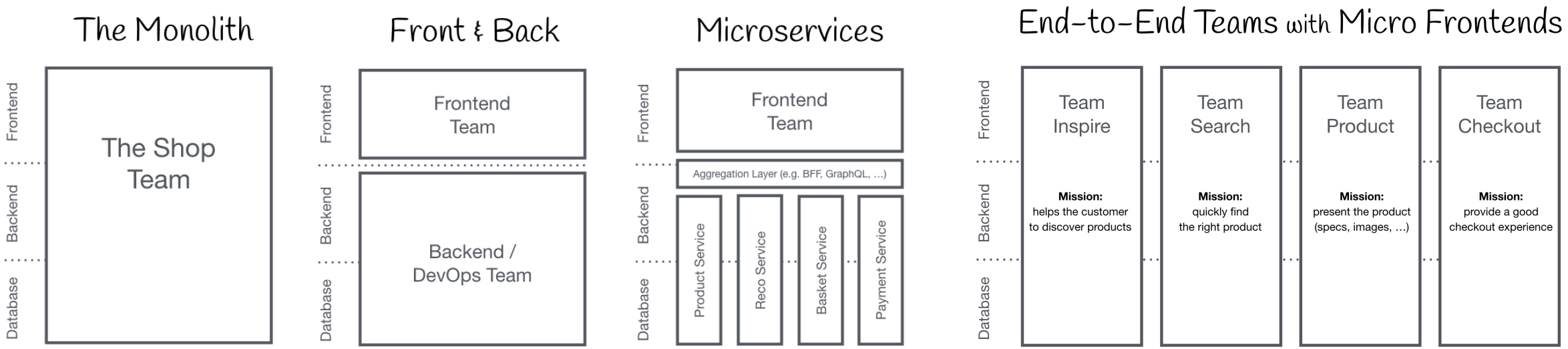
Antwort: a und b

Frage 3: Was ist Webpack – Module Federation?

a. Ein Plugin zum Verwalten von CSS-Dateien
b. Ein Webpack-Plugin zum Laden von Modulen zur Laufzeit aus verschiedenen Build-Kontexten
c. Ein Tool zur Verwaltung von Webpack-Konfigurationen
d. Ein Plugin zum Komprimieren von JavaScript-Dateien

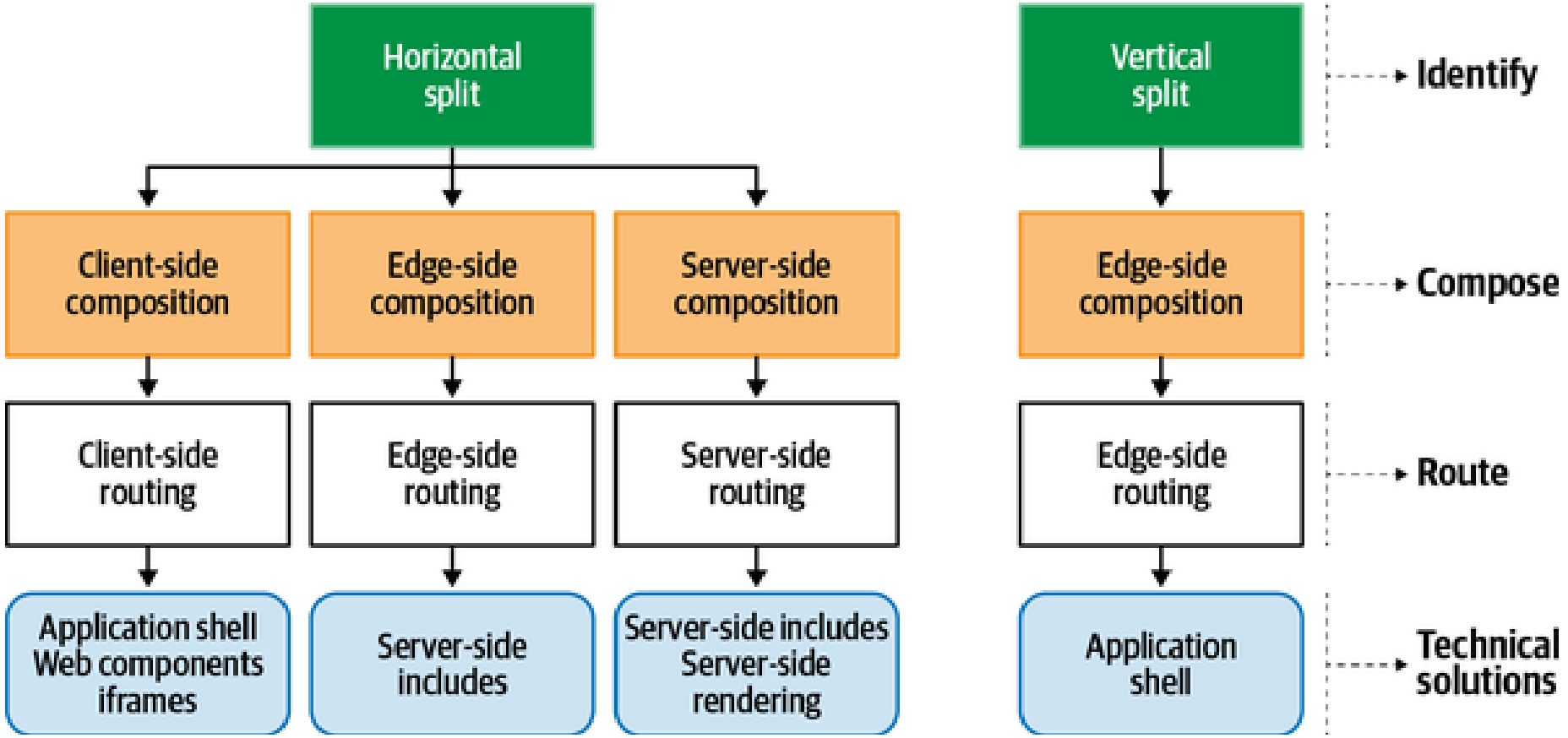
Antwort: b

Vergleich mit traditionellen Frontend-Architekturen

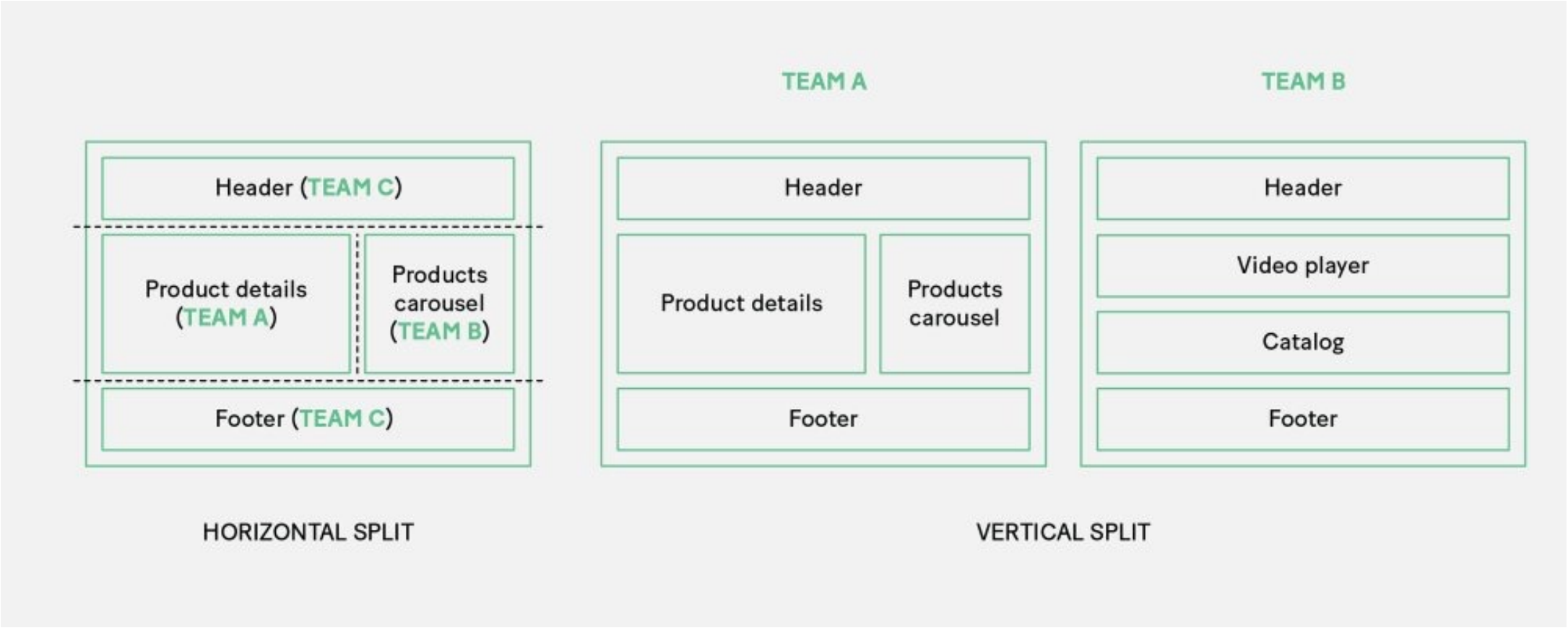


Source: https://micro-frontends.org/

Micro-Frontends Komposition



Micro-Frontends: Vertical vs. Horizontal Split



Ansätze zur Implementierung von Micro-Frontends

Ansatz	Vorteile	Nachteile	Resourcen
Iframes	- Vollständige Isolation - Einfache Integration unterschiedlicher Technologien - Unabhängige Deployments	- Eingeschränkte Kommunikation - SEO und Performance-Probleme - Inkonsistente Benutzererfahrung	MDS - Iframes
Single-Spa	- Flexibles Laden basierend auf Routen - Unterstützung mehrerer Frameworks - Zentrale Verwaltung von Routing und Lifecycle	- Komplexe Konfiguration - Setup-Aufwand für gemeinsame State-Management	Single-Spa GitHub Single-Spa Documentation

Ansätze zur Implementierung von Micro-Frontends

Ansatz	Vorteile	Nachteile	Resourcen
Web Components	- Standardisiert - Wiederverwendbar - Kann in verschiedenen Projekten verwendet werden	- Begrenzte Browser-Unterstützung - Komplexität bei State-Management-Integration	MDN – Web Components
Webpack - Module Federation	- Dynamisches Laden zur Laufzeit - Unterstützung von Code-Splitting und Sharing - Schnelles Setup für Anwendungen	- Abhängigkeit von Webpack - Erfordert ein tiefes Verständnis der Webpack-Konfiguration	Webpack-Module Federation Module-Federation Beispiele

Real-World Beispiele

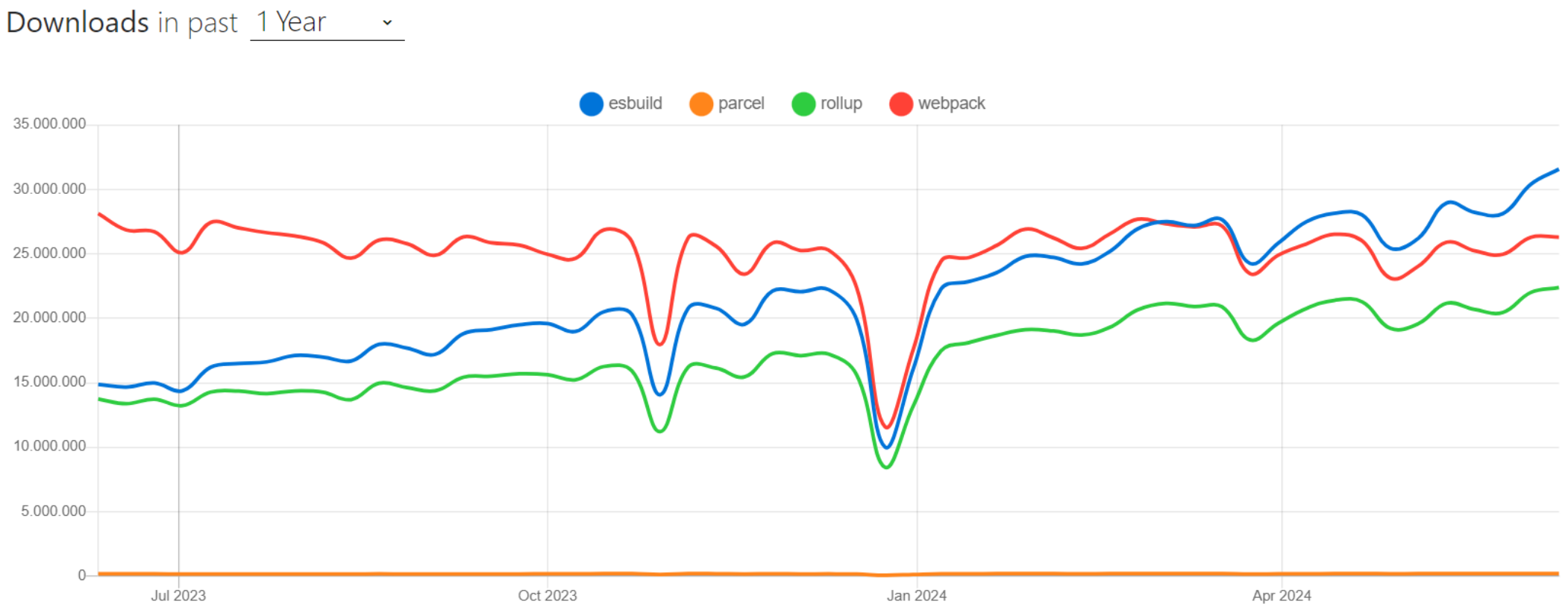
Lattice, verwendet
Webpack – Modul Federation
Source: TechBlog

Web Components,
Webpack - Module Federation
Source: TechBlog

Bootstrap,
Webpack - Modul Federation
Source: Dazn-Tech

SAP Luigi,
Module Federation

Webpack im Vergleich zu anderen JS-Bundlern



Warum Webpack?

- Flexibilität und Modularität
- Features wie
  - Code Splitting
  - Tree Shaking
  - Caching
- Aktive und große Community
- Ermöglicht Micro-Frontends durch das **Module Federation Plugin**

Was ist das Module Federation Plugin?

Eingeführt Webpack 5

- Ermöglicht das dynamische Teilen und Laden von Code zur Laufzeit zwischen unterschiedlichen Projekten.

Warum ist das wichtig?

- Ideal für die modulare Struktur von Micro-Frontends.
- Fördert Wiederverwendbarkeit und Modularität

Hauptkonzepte von Module Federation

Host – Application

- Container, der andere Micro-Frontends integriert
  - Z.B. Ein Dashboard, das verschiedene Widgets (Micro-Frontends) integriert.

Remote – Application

- Ein Modul, das in der Host-Application verwendet wird
  - Z.B. Ein Kalender-Widget, das im Dashboard angezeigt wird.

Hauptkonzepte von Module Federation

Exposed Modules

- Module, die eine Remote-Application bereitstellt.
  - Z.B. Ein "DatePicker"-Modul, das die Kalender-Logik exportiert.

Shared Modules

- Abhängigkeiten, die zwischen Host und Remote geteilt werden.
  - Z.B. Gemeinsame Nutzung von React und React-DOM.

Wie funktioniert Module Federation?

Der Host lädt die Module dynamisch zur Laufzeit

Remote stellt diese Module bereit

Shared Modules optimieren die Ladezeit durch gemeinsame Nutzung

Wie Micro-Frontends mit Webpack Module Federation erstellen?

Manueller Ansatz:

- Projektstruktur festlegen
- Dependencies installieren
- Komponenten erstellen
- Webpack Konfiguration schreiben

Verwendung von Hilfs-Tools

- npx webpack init -> erstellt grundlegende Webpack Konfiguration

Komplettlösungen

- npx create-mf-app -> Generiert eine vollständige, lauffähige Anwendung

Beispiel, mit create-mf-app

1. Erstelle ein neues Projekt
2. Im Terminal „npx create-mf-app“ ausführen
3. Gib der App einen
    1. Namen (host, header, content),
    2. Type (Application),
    3. Port (8080, 8081, 8082),
    4. Framework (react),
    5. Language (javascript),
    6. CSS (CSS),
    7. Bundler (Webpack).
4. Nächste Schritte:
    1. cd <NameDerApp>
    2. npm install
    3. npm start

Für jedes weitere Micro-Frontends muss ein weiteres Terminal geöffnet werden.

Aufgabe 1

1. Erstelle eine Host-Anwendung („host“) und zwei Remote Anwendungen („header“ und „content“).
2. Füge die Komponenten aus dem Workshop-Repo den Anwendungen hinzu (jeweils ins /src directory).
3. Starte die Anwendungen. (npm start)

Die Anwendungen sollten ohne Fehlermeldungen laufen.

Zeitaufwand: ~ 20 min

Webpack Config

```
module.exports = (_, argv) => ({
  output: {
    publicPath: "http://localhost:8081/",
  },

  resolve: {
    extensions: [".tsx", ".ts", ".jsx", ".js", ".json"],
  },

  devServer: {
    port: 8081,
    historyApiFallback: true,
  },
```

Webpack Config

```
module: {
  rules: [
    {
      test: /\.m?js/,
      type: "javascript/auto",
      resolve: {
        fullySpecified: false,
      },
    },
    {
      test: /\.(css|s[ac]ss)$/i,
      use: ["style-loader", "css-loader", "postcss-loader"],
    },
    {
      test: /\.(ts|tsx|js|jsx)$/,
      exclude: /node_modules/,
      use: {
        loader: "babel-loader",
      },
    },
  ],
},
```

Webpack Config

```
const deps = require("./package.json").dependencies;
```

```
plugins: [
  new ModuleFederationPlugin({
    name: "host",
    filename: "remoteEntry.js",
    remotes: {
      remote1: "remote1@http://localhost:8081/remoteEntry.js",
      remote2: "remote2@http://localhost:8082/remoteEntry.js",
    },
    exposes: {
      "./Button": "./src/Button",
      "./TextInput": "./src/TextInput",
    },
    shared: {
      ...deps,
      react: {
        singleton: true,
        requiredVersion: deps.react,
      },
      "react-dom": {
        singleton: true,
        requiredVersion: deps["react-dom"]
      },
    },
  }),
  new HtmlWebPackPlugin({
    template: "./src/index.html",
  }),
  new Dotenv()
],
```

Module Federation Konfigurieren

Exposen von Modulen

```
new ModuleFederationPlugin({
  name: "app1",
  filename: "remoteEntry.js",
  remotes: {},
  exposes: {
    "./Button": "./src/Button",
  },
```

Laden von Remotes (Modulen)

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    app1: "app1@http://localhost:8081/remoteEntry.js",
  },
```

Verwenden der Komponenten in der Host-Anwendung

```
import React from "react";
import ReactDOM from "react-dom/client";

import Button from "app1/Button";
const App = () => (
  <div>
      <Button />
  </div>
);
ReactDOM.createRoot(document.getElementById("app")).render(<App />);
```

Pause (10 min)

Erinnerung

Exposen von Modulen

```
new ModuleFederationPlugin({
  name: "app1",
  filename: "remoteEntry.js",
  remotes: {},
  exposes: {
    "./Button": "./src/Button",
  },
```

Laden von Remotes (Modulen)

```
new ModuleFederationPlugin({
  name: "host",
  filename: "remoteEntry.js",
  remotes: {
    app1: "app1@http://localhost:8081/remoteEntry.js",
  },
```

Verwenden der Komponenten in der Host-Anwendung

```
import React from "react";
import ReactDOM from "react-dom/client";

import Button from "app1/Button";
const App = () => (
  <div>
      <Button />
  </div>
);
ReactDOM.createRoot(document.getElementById("app")).render(<App />);
```

Aufgabe 2

1. Nutze das ModuleFederation-Plugin, um die Komponenten zu Exposen und um Remotes zu referenzieren und zu laden.
2. Verwende die Komponenten dann in der App.jsx der Host-Anwendung.
3. Starte die Anwendungen neu.

Generell:

Änderungen an der webpack.config.js-Datei werden erst nach einem Neustart der Anwendung wirksam!

Zeitaufwand: ~ 10 min

Kommunikation zwischen Micro-Frontends

- Globales Zustandsmanagement: Redux, Context API
- Eventbasierte Kommunikation: „Custom Events“
- URL-basierte Kommunikation: URL-Parameter und Hashes
- Shared Services: z.B. EventBus

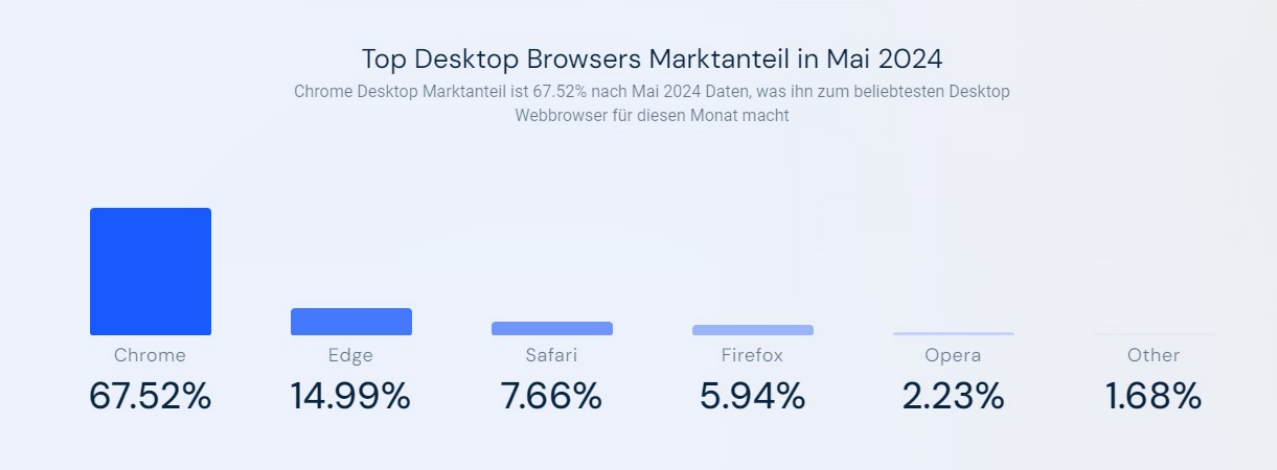
In diesem Workshop werden wir **Custom Events** benutzen.

Warum Custom Events?

- Custom Events fördern eine lose Kopplung zwischen Komponenten
- Sie sind Flexibel und leicht zu implementieren, ohne zusätzliche Bibliotheken
- Nutzen das Native Event-System des Browsers, was zu einer besseren Performance führen kann

CustomEvents - Browser Kompatibilität

- Verwendbar auf den beliebtesten Browsern



https://www.similarweb.com/de/browsers/worldwide/desktop/
Stand: Mai 2024

Browser	Unterstützt seit
Chrome	Version 15
Firefox	Version 11
Opera	Version 6
Safari	Version 6.1
Edge	Alle Versionen

https://caniuse.com/?search=custom%20event
Stand: Juni 2024

Wie nutzt man Custom Events (in JavaScript)?

Erstellen:

```
const event = new CustomEvent('meinCustomEvent', {
  detail: "Hallo, ich bin ein Custom Event!"
});
```

Dispatchen:

```
window.dispatchEvent(event);
```

Empfangen:

```
window.addEventListener('meinCustomEvent', handleCustomEvent);
```

EventListener Entfernen:

```
window.removeEventListener('meinCustomEvent', handleCustomEvent);
```

Beispiel Verwendung:

JS-A.-Funktion die das CustomEvent logt:

```
const handleCustomEvent = (event) => {
  console.log('Received event:', event.detail);
};
```

Best Practices

- **Code-Splitting**: Teilt das Bundle in kleinere Stücke, die bei Bedarf geladen werden können.
- **Tree Shaking**: Entfernt unbenutzte Module.
- **Lazy Loading**: Lädt bestimmte Teile deines Codes nur, wenn sie benötigt werden.
- **Shared Dependencies**: Reduziere doppelte Abhängigkeiten, die in verschiedenen Teilen deiner Anwendung verwendet werden.

Beispiel Code Splitting

```
module.exports = {
  mode: 'development',
  entry: './src/index.js',
  output: {
    filename: '[name].bundle.js',
  },
  optimization: {
    splitChunks: {
      chunks: 'all',
    },
  },
};
```

```
document.getElementById('loadButton').addEventListener('click', () => {
    import('./module')
      .then((module) => {
        module.doSomething();
      })
      .catch((err) => console.error('Error loading module:', err));
  });
```

```
export function doSomething() {
  console.log('Module loaded and function executed');
}
```

Beispiel Tree Shaking

```
module.exports = {
  mode: 'production', // Tree Shaking ist nur im Productions-Modus aktiv
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
  },
};
```

```
// src/utils.js
export function usedFunction() {
    console.log('This function is used');
}

export function unusedFunction() {
    console.log('This function is not used');
}
```

```
// src/index.js
import { usedFunction } from './utils';

usedFunction();
```

Beispiel Lazy Loading (React)

```jsx
import React, { Suspense, lazy } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <h1>Main Application</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

Beispiel Shared Dependencies

```
// webpack.config.js (Remote)
const { ModuleFederationPlugin } = require('webpack').container;

module.exports = {
  mode: 'development',
  entry: './src/index.js',
  output: {
    publicPath: 'http://localhost:3001/',
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'remote',
      filename: 'remoteEntry.js',
      exposes: {
        './Button': './src/Button',
      },
      shared: {
        react: { singleton: true, requiredVersion: '^17.0.2' },
        'react-dom': { singleton: true, requiredVersion: '^17.0.2' },
      },
    }),
  ],
```

```
// webpack.config.js (Host)
const { ModuleFederationPlugin } = require('webpack').container;

module.exports = {
  mode: 'development',
  entry: './src/index.js',
  output: {
    publicPath: 'http://localhost:3000/',
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'host',
      remotes: {
        remote: 'remote@http://localhost:3001/remoteEntry.js',
      },
      shared: {
        react: { singleton: true, requiredVersion: '^17.0.2' },
        'react-dom': { singleton: true, requiredVersion: '^17.0.2' },
      },
    }),
  ],
};
```

Aufgabe 3

- Nutze die Komponenten aus dem GitHub-Repo und vervollständige sie so, dass sie Custom Events nutzen.
- Nutze Lazy Loading im Host um die Komponenten zu laden.

Zeitaufwand: ~ 10 min

Das wichtigste auf einen Blick

- Micro-Frontends ermöglichen Modularität, Flexibilität, Skalierbarkeit
- Webpack Module Federation bietet Modularität und Flexibilität
  - Konzepte:
    - Host-Application
    - Remote-Application
    - Exposed-Modules
    - Shared Modules
  - Techniken:
    - CustomEvents zur einfachen und schnellen Kommunikation zwischen Micro-Frontends
    - Lazy Loading um Ladenzeiten zu verringern

Fragen?