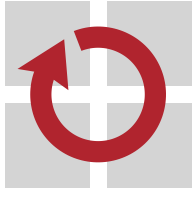
Web-basierte Systeme – Übung

04: Webpack, WebRTC, und Aufgabe 3

Wintersemester 2022

Arne Vogel



Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Übersicht

Webpack

WebRTC Grundlagen

PeerJS Library

Aufgabe 3

Webpack

Übersicht

Webpack

WebRTC Grundlagen

PeerJS Library

Aufgabe 3

Motivation für Module-Bundler wie Webpack

- Ursprünglich wurde JavaScript primär dazu benutzt HTML-Dokumente um animierte Elemente zu ergänzen
- JavaScript wurde/wird direkt in HTML-Dokumenten geschrieben oder verlinkt
- Für komplexe Webanwendungen ist dieses Vorgehen nicht zielführend
  - Kein Package-Manager
  - Kein Modulsystem
  - Kein Buildsystem (Linter, Compiler, usw.)

Klassisches Vorgehen

- JS-Datei wird in der HTML-Datei geladen

```
...
<script src="https://awesomeproject/awesome.min.js"><script>
...
```

- JS-Code wird meistens in separater Datei genutzt

```
const a = awesome_function()
```

- Dadurch werden alle Objekte in den **globalen Namespace** geladen
- Gefahr von **shadowing**
- Keine Modulverwaltung (automatische Updates, usw.)

Heutiges Vorgehen

- Module werden über ein Package-Manager verwaltet

```
npm install awesomeLib
```

- Einzelne Funktionen oder ganze Bibliotheken werden über **import statements** in einen **lokalen Namespace** geladen

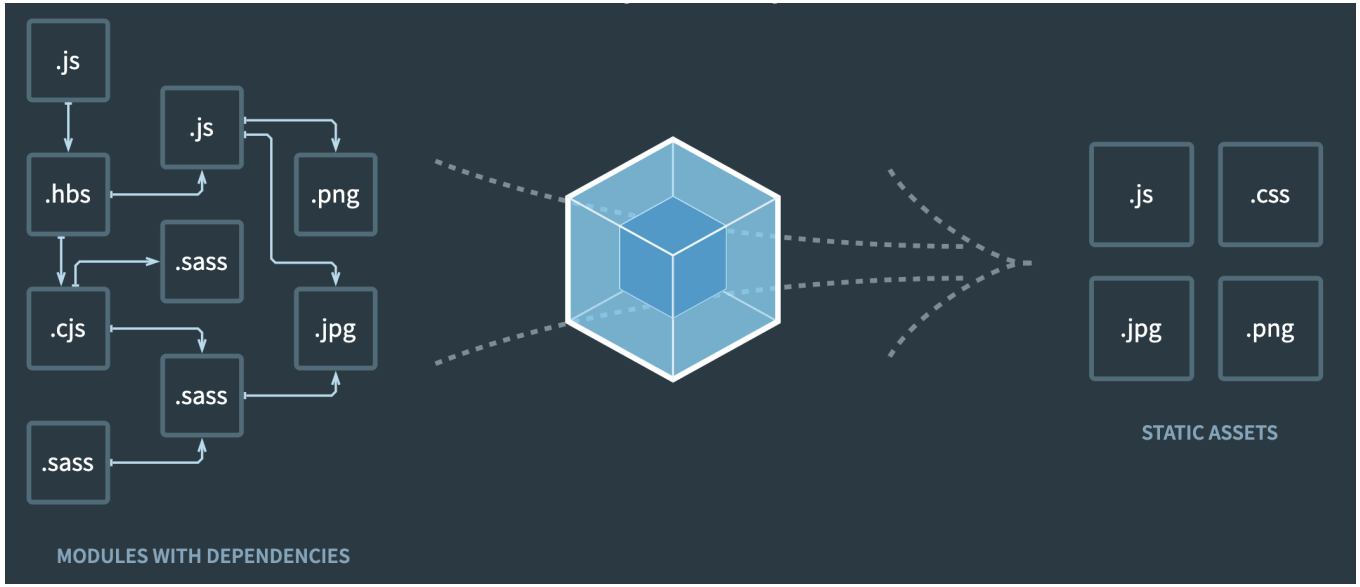
```
import awesome from 'awesomeLib'
const a = awesome.doAwesomeStuff()
```

npm (ehemals Node Package Manager)

- Paketmanager für die JavaScript-Laufzeitumgebung Node.js
- Über Bundler wie Webpack auch für clientseitiges JavaScript
- Alle build dependencies werden in der **package.json** verwaltet

```
1  {
2          "name": "test-project",
3          "version": "1.0.0",
4          "main": "src/main.js",
5          ...
6          "dependencies": {
7              "vue": "^2.5.2",
8              "awesomeLib": "0.5.3"
9            },
10 }
```

Webpack - Übersicht



- Statischer Module-Bundler
- Löst Abhängigkeitsbaum auf und geniert statische Assets (Bundles)
- Webpack wird von React, Angular, Vue verwendet

Webpack Projektstruktur

```
test-project
|- package.json
|- webpack.config.js
|- node_modules          # externe Libs - dynamisch von npm generiert
|- /src
  |- index.js
  |- myModule.ts
|- /styles
  |- chat.css
  |- main.css
  |- giphy.sass
|- /dist          # gebaute Webanwendung - dynamisch von Webpack generie
  |- main.js
  |- index.html
  |- style.css
```

Webpack import/export

Export:

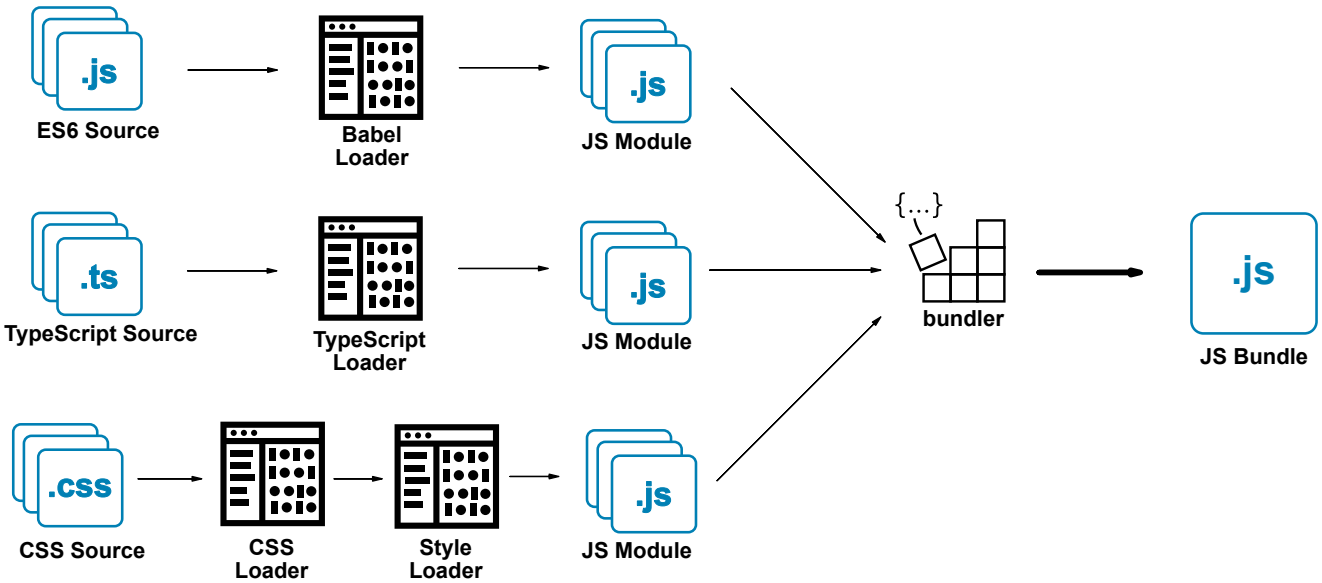
```
export const Count = 5;
export function Multiply(a, b) {
  return a * b;
}

// Default export
export default {
  // Some data...
};
```

Import:

```
import MyModule from './my-module.js';
import { NamedExport } from './other-module.js';
```

Webpack - Funktionsweise



- Webpack besteht aus Plugins und Loadern
- Webpack erlaubt das Importieren verschiedenster Dateitypen
- Abhängigkeiten werden vom **entry point** aus gesucht
- Typabhänge **Loader** formen Code in einheitliche JS-Module um

WebRTC Grundlagen

Übersicht

WebRTC Übersicht

- **WebRTC:** offener Standard, der
  - Web Real-Time Communication
  - “Web-Echtzeitkommunikation”
- Kommunikation heißt: Datenübertragung **direkt zwischen Browsern**
  - Video/Voice-Streams
  - Bidirektionale Daten: **DataChannels**
- Möglichst **ohne** System dazwischen
  - WebRTC implementiert **Fallbacks**, damit immer eine Verbindung zustande kommen kann
- Hauptprobleme: NAT, Firewalls

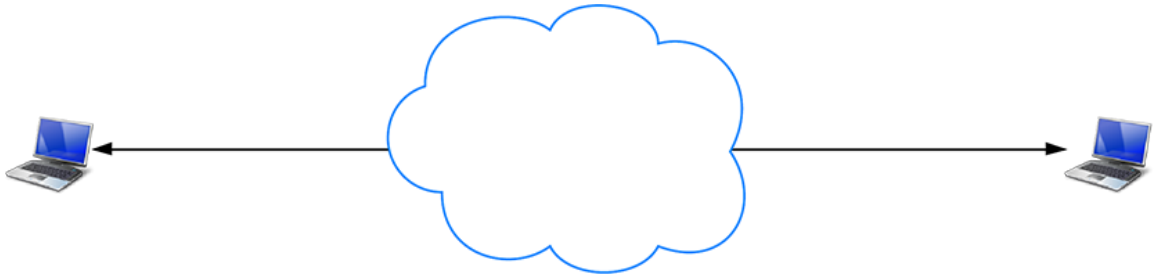
Signaling

- **Signaling** (Signalisierung) dient der Koordinierung der WebRTC-Kommunikation

- WebRTC-Clients tauschen dabei folgende Informationen aus:
  - Session control messages (Öffnen/Schließen der Verbindung)
  - Fehlernachrichten
  - Media Metadaten: Einigung auf Codecs/Bandbreite
  - Schlüssel für sichere Verbindungen
  - Netzwerk Metadaten (IP Adressen und Ports)

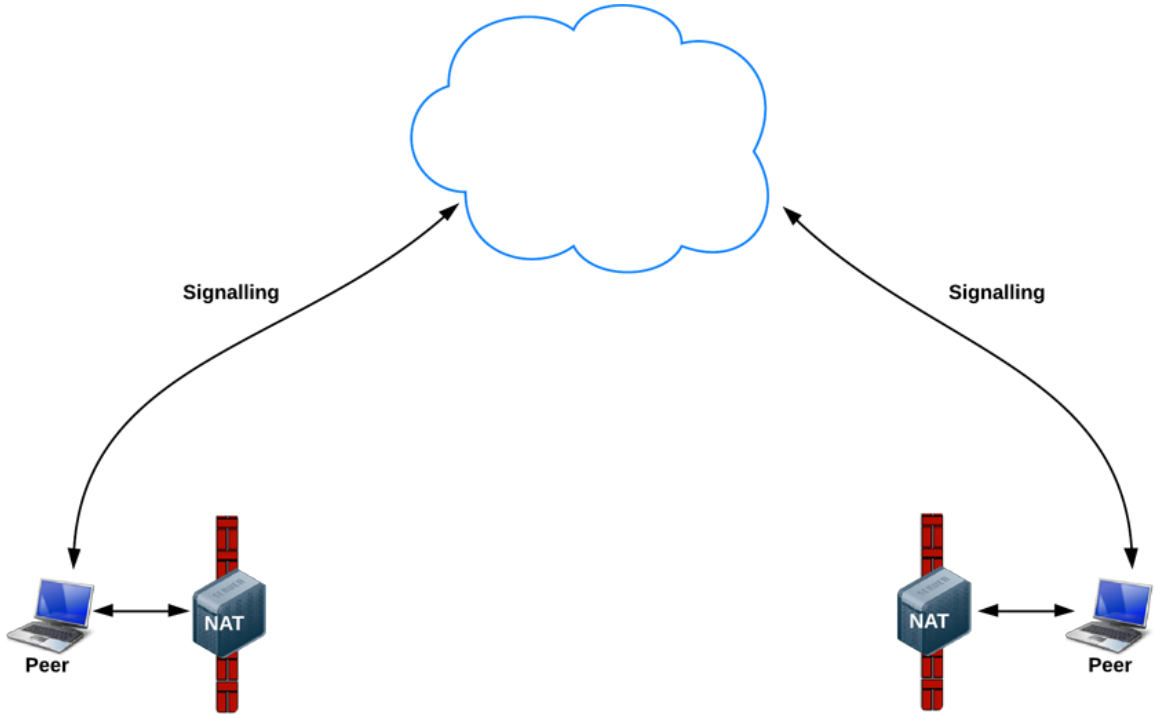
- Unterliegendes Protokoll ist im WebRTC-Standard **nicht definiert**
  - Maximale Kompatibilität mit bestehenden Technologien

Verbindungsaufbau in WebRTC



- Ideale Welt: Direkte Verbindung zwischen Endgeräten möglich
- Problem: NAT (Network Address Translation) und/oder Firewalls verhindern Verbindungsaufbau
- Lösung: ICE (Interactive Connectivity Establishment)

Verbindungsaufbau in WebRTC

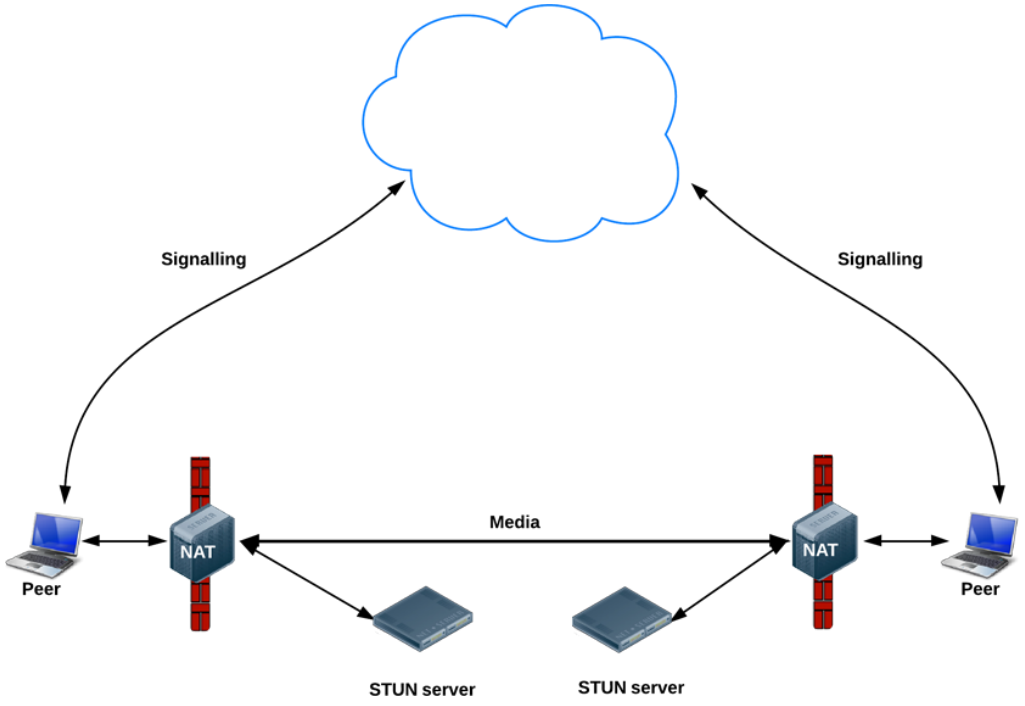


- Ideale Welt: Direkte Verbindung zwischen Endgeräten möglich
- Problem: NAT (Network Address Translation) und/oder Firewalls verhindern Verbindungsaufbau
- Lösung: ICE (Interactive Connectivity Establishment)

Verbindungsaufbau in WebRTC

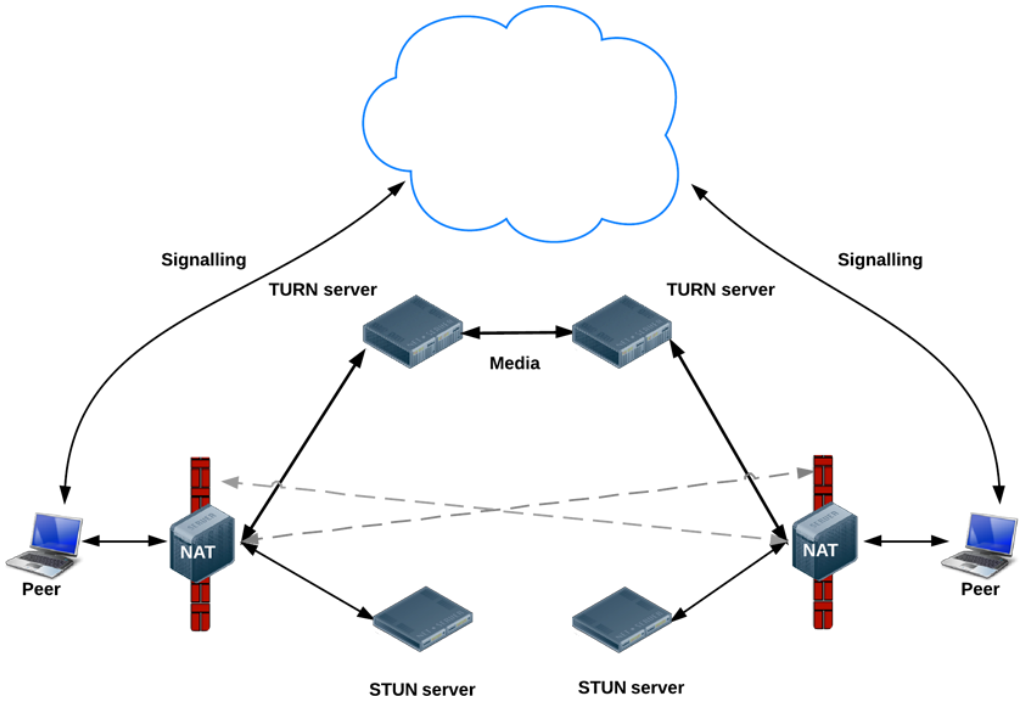
- Verbindungsaufbau mittels ICE
  (Interactive Connectivity Establishment)
  - 1. Versuch: direkte Verbindung
  - 2. Versuch: Session Traversal Utilities for NAT (STUN)
  - 3. Versuch: Traversal Using Relays around NAT (TURN)
- Weitere Protokolle werden unterstützt

Session Traversal Utilities for NAT (STUN)



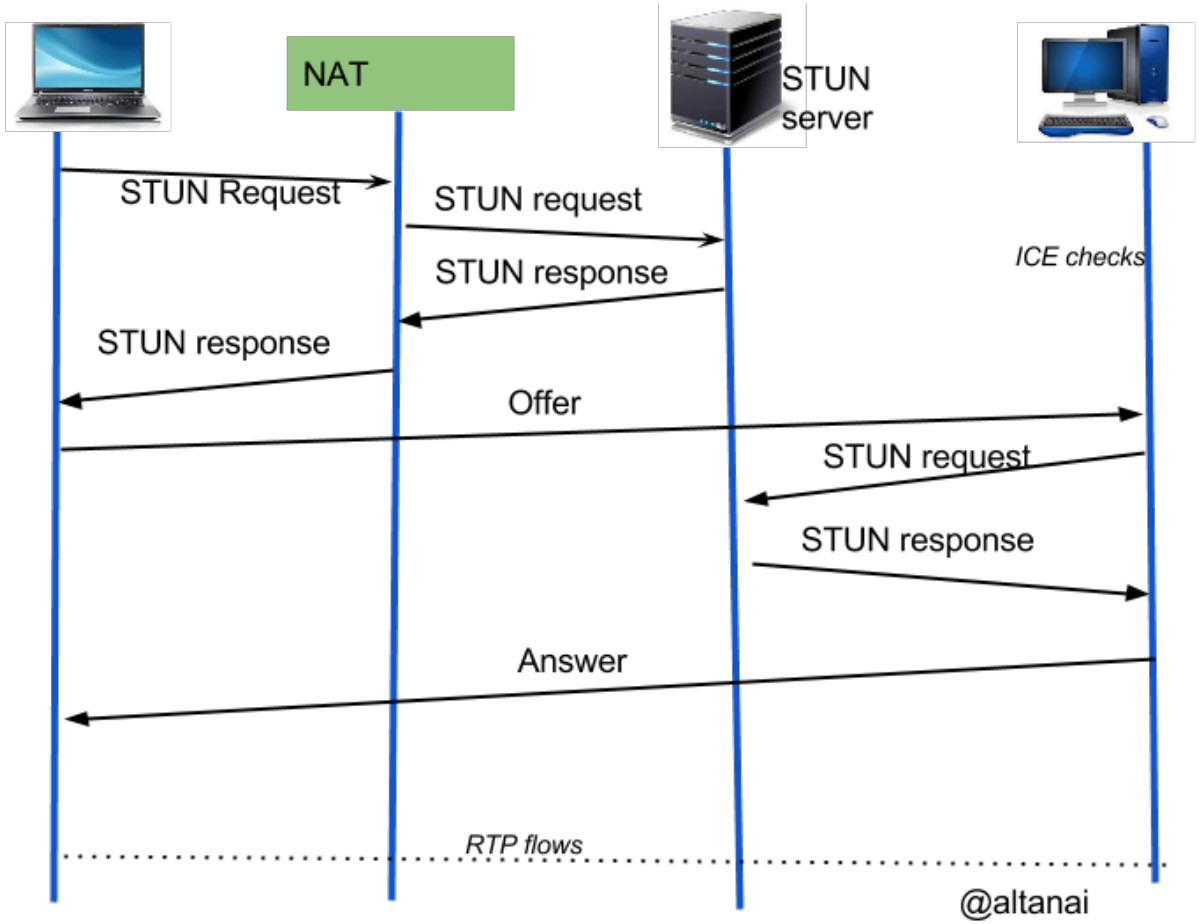
- Über STUN erfahren WebRTC-Clients externe IP+Port
- IP+Port werden dann an den anderen Client weitergeleitet

Traversal Using Relays around NAT (TURN)



- TURN-Server leiten Verkehr weiter
- Fallback, wenn über STUN keine Verbindung hergestellt werden kann

Vereinfachter STUN Nachrichtenaustausch



Quelle: `https://telecom.altanai.com/tag/sdp/`

WebRTC Browser API

- Hauptobjekt: `RTCPeerConnection`
- API umfangreich und zu Beginn unübersichtlich
- Es gibt mehrere Libraries, die die Komplexität abstrahieren
  - **PeerJS**
  - simpleWebRTC
  - holla
  - simple-peer
  - ...

PeerJS **Library**

Übersicht

Die `PeerJS` Library

- **Vereinfacht** die Nutzung von WebRTC (und ICE)
- Implementierung für Node.js und Browser
- Unterstützung für **Voice/Video Streams**
- Unterstützung für **DataChannels**
- Open Source: `https://github.com/peers`
- Unterliegendes Signaling-Protokoll **frei wählbar**
- **Programmierende sind für den Austausch der Peer-IDs zuständig**

PeerJS **Beispiel**

- Importieren der PeerJS-Bibliothek (über Webpack)
- Instanziierung des Peer-Objektes
- Peer-Objekt enthält die ID (`peer.id`)

```
1 import Peer from 'peerjs';
2 const peer = new Peer();
```

- Herstellen der Verbindung durch den zweiten Client

```
1 const conn = peer.connect('dest-peer-id');
```

- Erster Client erhält das Connection-Objekt (`conn`) über das Connection-Event

```
1 peer.on('connection', function(conn) { ... });
```

PeerJS **Events**

- `peer.on(event, callback)`
  Methode um Event-Listener einzurichten
  - `open`
    Wird ausgelöst wenn eine Verbindung aufgebaut wurde.
  - `connection`
    Wird ausgelöst wenn eine Datenverbindung zu einem anderen Peer aufgebaut wurde.
  - `close`
    Wird ausgelöst wenn der Peer beendet wurde.
  - `disconnected`
    Wird ausgelöst, wenn die Verbindung zwischen dem Peer und dem Signaling-Server unterbrochen wurde.
  - `error`
    Wird ausgelöst wenn ein Fehler aufgetreten ist. Fehler sind immer kritisch und beenden den Peer.

Aufgabe 3

Übersicht

Webpack

WebRTC Grundlagen

PeerJS Library

Aufgabe 3

Aufgabe 3

- Ausliefern des Char-Clients als Bundle

- Implementierung eines „Secret Messaging Modus“(SMM)
  - Nachrichten direkt zwischen Clients ausgetauscht

Aufgabe 3.1 + 3.2 - Webpack

- Anpassung bisheriger Lösung auf lokale Namespaces
  - Insbesondere Giphy-Integration

  $\Rightarrow$ Sinnvoller Ansatz:
    - `giphy.js`: Anfragen von Gifs an die Giphy-API
    - `client.js`: Einfügen von Gifs in den DOM

- Bundling von JS und CSS Code mit Webpack

Aufgabe 3.3 - WebRTC

- SMM Nachrichten direkt zwischen Clients aus
  - $\Rightarrow$ WebRTC
    - Keine Speicherung auf Server
- IDs für WebRTC Verbindung über Server ausgetauscht
- Behandlung von Verbindungsabbrüchen

Aufgabe 3.3 - WebRTC - UI

- Slider zum (De-)Aktivieren von SMM
- Geheime Nachrichten hervorgehoben
- CSS wird gestellt!

Node-Red Sicherheit

- Password: *node-red admin init*