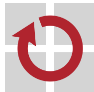
Web-basierte Systeme – Übung

05: TypeScript und WebAssembly

Wintersemester 2023

Arne Vogel, Maxim Ritter von Onciul



Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Übersicht

TypeScript

WebAssembly

Aufgabe 4

TypeScript

Dynamische vs. Statische Sprachen

Dynamische Sprachen

- Kein Buildsystem nötig
  - \+ Geringerer Entwicklungsaufwand
  - \+ Compilezeit entfällt
- Typinformationen zur Laufzeit
  - \+ Debug-Informationen
  - \+ Objekte erweiterbar
- Code dynamisch
  - \+ Kann zur Laufzeit nachgeladen werden
- ...

Statische Sprachen

Dynamische vs. Statische Sprachen

Dynamische Sprachen

- Kein Buildsystem nötig
  - \+ Geringerer Entwicklungsaufwand
  - \+ Compilezeit entfällt
- Typinformationen zur Laufzeit
  - \+ Debug-Informationen
  - \+ Objekte erweiterbar
- Code dynamisch
  - \+ Kann zur Laufzeit nachgeladen werden
- ...

Statische Sprachen

- Codeanalyse
  - \+ (Offensichtliche) Fehler vor Ausführung bemerkt
  - \+ Findet Fehler in nicht ausgeführtem Code
- Optimierungen
  - \+ Bessere Performance zur Laufzeit
- ...

Fehlerfindung in JavaScript

- Fehler in JavaScript erst zur Laufzeit bemerkbar
- In vielen Fällen lösen „unsinnige“ Operationen keinen Fehler aus!
- Beispiel: Undefinierte Variablen:

```
let myName = 'David';
// ...
name = 'Manuel';
```

- name nicht mit `let` deklariert
  - ⇒ Globale Variable name wird erzeugt

Fehlerfindung in JavaScript

- Beispiel: Fehlende/Überflüssige Funktionsparameter

```
1  function concat(first, second) {
2          console.log(first + ' ' + second);
3  }
4  // ...
5  concat('Hallo'); // Ausgabe: 'Hallo undefined'
```

- Jeder Parameter ist optional
  - Fehlende Parameter sind `undefined`
  - Überflüssige Parameter werden ignoriert

Fehlerfindung in JavaScript

- Beispiel: Rechenoperationen

```
1/0;       // Ergebnis: NaN
'test'/5; // Ergebnis: NaN
[] + {};  // Ergebnis: '[object Object]'
{} + [];  // Ergebnis: 0
```

- JavaScript hat unterschiedliche Regeln für Rechenoperationen

- Viele nicht intuitiv

Fehlerfindung in JavaScript

- Fehlerüberprüfung kann selbst implementiert werden
  - Check ob `undefined`, `NaN`, `0`, ...
  - Anzahl von Parametern und deren Typen

$\Rightarrow$ Extrem aufgeblähter Code

Strict Mode

- Besserer Ansatz: Strict Mode
  - Seit ECMAScript 5
  - Aktiviert mit `"use strict";` als erster Anweisung
  - Abwärtskompatibel
- Weist JS-Engine an bestimmte Operationen zu verbieten
  - Entweder Syntaxfehler beim Parsen oder Laufzeitfehler
- Beispiele:
  - Zuweisung nicht deklarierter Variablen
  - Nutzung des `with` Statements
    - Fehleranfällig und schlecht für Performance
  - Löschen von Variablen und Funktionen
  - Keywords als Variablennamen

TypeScript

- **TypeScript**[1] ist eine Programmiersprache auf Basis von JavaScript
  - Open Source: `https://github.com/Microsoft/TypeScript`
  - Entwickelt von Microsoft
  - Jeder JS Code ist auch TypeScript Code
- TypeScript kann nicht direkt vom Browser ausgeführt werden
  - TypeScript Code wird zu JavaScript **kompiliert**
- Größter Unterschied: Statische Typisierung

  ⇒ Typfehler werden zur Compile-Zeit gefunden

[1] `https://www.typescriptlang.org/docs/home.html`

TypeScript - Typen

- Typen werden über Annotationen festgelet

```
1  let myName: string = "Manuel";
2
3  function add(first: number, second: number): number {
4    return first + second;
5  }
```

- Nicht jeder Typ muss angegeben werden

```
1  function add(first: number, second: number) {
2    // Compiler erkennt, dass return-Wert Typ number hat
3    return first + second;
4  }
```

TypeScript - Typen

- Basis-Typen:
  - `boolean`, `number`, `string`, `array`, `enum`
    - Array entweder als `number[]` oder `Array<number>`
  - `tuple`
    - Array fester Größe
  - `any`
    - Beliebiger Typ
    - $\Rightarrow$ Keine Typechecks durchgeführt
  - `void` und `never`
    - Für Funktionen ohne Rückgabewert
  - `undefined` und `null`
    - Genutzt um anzuzeigen, welche Variablen den Wert undefined/null annehmen dürfen
- Außerdem Interessant: `HTMLElement`
  - Interface für alle HTML Objekte

TypeScript - Unions

- Mit **Unions** ist es möglich mehr als einen Typen zu erlauben

```
let id : number | string;
id = 12345;   // OK
id = "12345"; // OK
id = true;    // Fehler
```

- Besonders mit `undefined` und `null` sinnvoll

```
let id : number | string | undefined; // undefined erlaubt
let name : string;                     // undefined nicht erlaubt
```

TypeScript - Weitere Features

- Definition von Klassen
  - Ab ECMAScript 6 auch in JS
  - Einige zusätzliche Features
    - Vererbung
    - Public, protected und private Variablen
    - Kurzformen für Konstruktor, getter/setter
    - ...
- Interfaces
  - Kann (optionale) Felder und Funktionen enthalten

TypeScript - Build-Vorgang

- TypeScript ist kompilierte Sprache
  - TypeScript → Compiler → JavaScript
- Compiler entfernt allen TS-spezifischen Code
  - Ergebnis is äquivalenter JS Code
  - Kompilieren dient nur Überprüfung auf Fehler

```
1 interface Person {
2   name : string;
3 }
4 function printName(p: Person) {
5   console.log(p.name);
6 }
7 let me: Person = {name: "Manuel"};
8 printName(me);
```

→

```
1 function printName(p) {
2   console.log(p.name);
3 }
4 let me = { name: "Manuel" };
5 printName(me);
```

TypeScript - Build-Vorgang

Compiler kann konfiguriert werden

- ECMAScript-Version des ausgegebenen Codes
  - Standard is ECMAScript 3
  - Erscheinungsjahr 1999
- Strenge der Fehlerchecks[2]
  - `noImplicitAny`
    - Fehler, wenn fehlender Typ nicht erkannt werden kann
  - `strictNullChecks`
    - Überprüft, ob Werte null/undefined sein dürfen
  - `noImplicitReturns`
    - Fehler, wenn nicht jeder Code-Pfad einen Rückgabewert hat

---

[2] https://www.typescriptlang.org/tsconfig#Strict_Type_Checking_Options_6173

WebAssembly

Übersicht

TypeScript

WebAssembly

Aufgabe 4

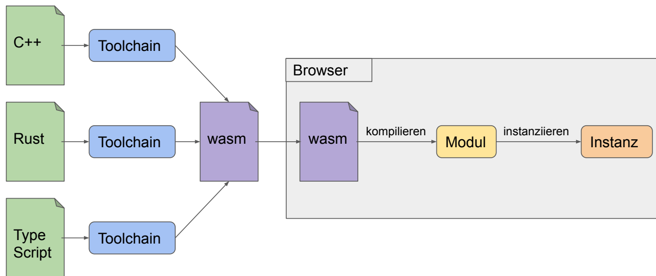
WebAssembly - Überblick

- **WebAssembly (WASM)** ist ein Bytecode-Format für die Ausführung im Web
- Ziele:
  - Gute Performanz
    - Ahead-of-Time Optimierungen
    - Kompaktes Format
  - Sicherheit
    - Eingeschränkter Speicherzugriff
    - Kein Zugriff auf DOM, Cookies, ...

Außerhalb des Webs

- Mittlerweile wird WASM auch außerhalb des Browsers eingesetzt
- WebAssemly Funktionen in der Cloud
- Eigenständige WASM-Ausführungsumgebungen
  - WebAssembly als Docker-Alternative
- Teile von Firefox werden aus WebAssembly kompiliert

Toolchains



- WASM wird nicht von Entwicklern direkt geschrieben, sondern aus anderen Sprachen generiert
- Nach kompilieren ist bekannt, was gebraucht wird
  - Funktionen, Speicher, ...
  - Wenn alles erfüllt, kann instanziiert werden

Toolchains

- Mittlerweile viele Ausganssprachen unterstützt
  - Unterschiedlich ausgereift
- Überblick hier: `https://github.com/appcypher/awesome-wasm-langs`

.Net
AssemblyScript
~~Astro~~ `Unmaintained`
Ballerina
Brainfuck
C
C#
C++
C4wa
Clean
Co
COBOL
Crystal
D
Eclair
Eel
Elixir
F#
Faust
Forest
Forth
Go
Grain
Haskell
Java
JavaScript
Julia
~~Idris~~ `Unmaintained`
KCL
Kotlin/Native
Kou
Lisp
Lobster
Lua
Lys
Never
Nim
Ocaml
Pascal
Perl
PHP
Plorth
Poetry
Python
Prolog
Ruby
Rust
Scheme
Scopes
~~Speedy.js~~ `Unmaintained`
Swift
~~Turboscript~~ `Unmaintained`
TypeScript
Wa
~~Wah~~ `Unmaintained`
~~Walt~~ `Unmaintained`
~~Wam~~ `Unmaintained`
Wase
WebAssembly
~~Wracket~~ `Unmaintained`
xcc
Zig

Toolchains - wasm-pack

- **wasm-pack**[3] ist eine Toolchain zum Generieren von WASM aus Rust
- Intern wird **wasm-bindgen** genutzt
- wasm-pack erzeugt zusätzlichen Code zur besseren Integration

[3] `https://rustwasm.github.io/wasm-pack/`

Toolchains - wasm-bindgen

- wasm-bindgen verwendet Annotationen um zu signalisieren, dass Funktion von außen aufgerunfen werden kann

```rust
// Export a `greet` function from Rust to JavaScript, that alerts
// hello message.
#[wasm_bindgen] // <---- Annotation
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

- Aus Code + Annotationen generiert wasm-bindgen WASM und JavaScript

Aufgabe 4

Übersicht

TypeScript

WebAssembly

Aufgabe 4

Aufgabe 4

- Der Chat-Client aus Aufgabe 3 soll um Funktionalität erweitert werden
  - Rechtschreibprüfung
  - Wortvorhersage

- Statt JavaScript sollen diesmal TypeScript und WebAssembly eingesetzt werden

Aufgabe 4.1 - Vorbereitung

- Installation der Toolchains für TypeScript und WebAssembly

- Anpassen von Webpack für neuen Build-Vorgang
  - Bestehender Code aus Aufgabe 3 soll weiterverwendet werden

Aufgabe 4.2 - Rechtschreibprüfung

- Falsch geschriebene Worte werden markiert und Verbesserungsvorschläge gegeben
  - Code für Fehlerfindung und Vorschläge wird gestellt
  - Wörter nicht im Wörterbuch sollen hinzugefügt werden können
- Selbst erstellter Code soll in TypeScript geschrieben werden
  - Soll mit `noImplicitAny` kompilieren
  - Gegebene Hilfsfunktionen um Typen erweitern

Aufgabe 4.3 - Wortvorhersage

fertig wahrgenommen

Das Layout ist

- Aus letztem geschriebenen Wort soll nächstes vorgeschlagen werden
- Dazu Rust-Library zu WASM kompiliert und einbinden
  - Library muss um Annotationen erweitert werden
  - Vortrainiertes Modell wird gestellt
- Generierter JavaScript Code soll nachvollzogen werden