Webentwicklung

In der Praxis

Inhalt dieser Einheit

1. Entwicklung: CSS
   - CSS-Präprozessoren
2. Entwicklung: JavaScript
   - Sprach-Erweiterungen
   - Wiederverwendbare Komponenten
   - Frontend-Frameworks
3. Qualitätssicherung
   - Unit-Tests
   - Funktionstests
4. Gedanken zur Technologieauswahl

Moderne Webentwicklung in der Berufspraxis

- **Spannungsfeld** in der Praxis:
  - Einerseits **Spieltrieb**:
    - tolle technische Möglichkeiten
    - die teilweise echt relevante Probleme lösen
    - die *in jedem Fall* und *unbedingt* ausprobiert werden müssen
  - Andererseits **Konsequenzen**:
    - jede weitere Technologie erhört die Gesamtkomplexität
    - Auswirkung auf Nachhaltigkeit, Wartung, Personal, …
- Letzte Woche: “Optimierung”
  - nicht alles optimieren was geht, sondern dosiert & Ziel-geleitet
- Diese Einheit:
  - Erklärung & Einordnung weiterer bekannten Technologien
  - Ziel: Zweck kennen und Relevanz pro Fall erkennen können

Offene Punkte (Ausschnitt)

- **Anwender** (A)
  2. *Schnell, zufrieden, angenehm*
     1. Lade-/Wartezeiten ✓
     2. Datumvolumen ✓
- **Betreiber** (B)
  2. ***Entwicklungs- und Wartungskosten***
     1. **Entwicklungsaufwand (CSS & JavaScript)**
     2. **Qualitätssicherung**
     3. **Technologieauswahl**
  3. *Betriebskosten*
     1. Netzbandbreite ✓
     2. CPU-Zeit ✓

Entwicklung: CSS

CSS-Präprozessoren

Entwicklungsaufwand reduzieren

- CSS-Präprozessoren: z.B. Less oder SASS
  - erweitern Sprachumfang von CSS
  - Browser kann nur CSS: SASS zu CSS “kompilieren”
    - simpel bei Verwendung von Symfonys *Webpack Encore*:

```
 /* webpack.config.js */
 Encore
   .addEntry('app', './assets/app.js')
+  .enableSassLoader()
 ;
```

      - dann einfach `*.scss` statt `*.css`-Dateien erstellen
    - CSS-Präprozessoren gibt es auch *stand-alone*
- Einige Erweiterungen:
  - Variablen (z.B. für Farben, Schriftarten)
  - Geschachtelte Selektoren (nah am DOM)
  - Mixins (zur Regel-Wiederverwendung)
  - und noch viel mehr

SASS: Variablen & Operatoren

```scss
/* SASS-Quelle */
$color: #ff4400;
$size: 14pt;

h1 {
  color: $color;
  border-bottom: 2px solid $color;
  font-size: $size * 2 + 2pt;
}
h2 {
  color: $color + #222222;
  font-size: $size * 1.5;
}
```

```css
/* CSS-Ausgabe */
h1 {
  color: #ff4400;
  border-bottom: 2px solid #ff4400;
  font-size: 30pt;
}

h2 {
  color: #ff6622;
  font-size: 21pt;
}
```

- Schriftgrößen berechnet
- Farbe von `h2` aufgehellt

**) Variablen im CSS-Standard sind ↗seit 2 Jahren "fast fertig", werden aber von großen Browsern bereits unterstützt (Syntax: `--color`).*

SASS: Schachtelung

```scss
/* SASS-Quelle */
#main {
  width: 97%;

  p, div {
    font-size: 2em;
    a {
      font-weight: bold;
    }
  }

  pre {
    font-size: 3em;
  }
}
```

```css
/* CSS-Ausgabe */
#main {
  width: 97%;
}

#main p, #main div {
  font-size: 2em;
}

#main p a, #main div a {
  font-weight: bold;
}

#main pre {
  font-size: 3em;
}
```

- orientiert sich am DOM

Quelle: http://sass-lang.com/documentation/file.SASS_REFERENCE.html#nested_rules

SASS: Schachtelung (2)

```scss
/* SASS-Quelle */
p.primary a {
  font-weight: bold;
  text-decoration: none;

  &:hover {
    text-decoration: underline;
  }

  footer & {
    font-weight: normal;
  }
}
```

```scss
/* CSS-Ausgabe */
p.primary a {
  font-weight: bold;
  text-decoration: none;
}

p.primary a:hover {
  text-decoration: underline;
}

footer p.primary a {
  font-weight: normal;
}
```

- `&` fügt *parent* ein

Quelle: http://sass-lang.com/documentation/file.SASS_REFERENCE.html#nested_rules

SASS: Mixins

```sass
/* SASS-Quelle */
@mixin large-text {
  font: {
    family: Arial;
    size: 20px;
    weight: bold;
  }
  color: #ff0000;
}

h1 {
  @include large-text;
}
.page-title {
  @include large-text;
  padding: 4px;
  margin-top: 10px;
}
```

```sass
/* CSS-Ausgabe */
h1 {
  font-family: Arial;
  font-size: 20px;
  font-weight: bold;
  color: #ff0000;
}

.page-title {
  font-family: Arial;
  font-size: 20px;
  font-weight: bold;
  color: #ff0000;
  padding: 4px;
  margin-top: 10px;
}
```

- Auch: Vererbung & Funktionen

Quelle: http://sass-lang.com/documentation/file.SASS_REFERENCE.html#mixins

Entwicklung: JavaScript (1)

Sprach-Varianten

Entwicklungsaufwand reduzieren

- Spracherweiterungen für JavaScript
  - ähnliche Idee wie CSS-Präprozessoren
  - Quellcode in “besserer” Sprache
  - Browser kann nur JS: also Kompilieren zu JS
- CoffeeScript:
  - kompaktere Syntax
  - keine `;`, keine `{ }` (stattdessen: Einrückung)
- TypeScript:
  - Ergänzung um strengeres Typsystem

CoffeeScript: Beispiel

```coffeescript
square = (x) -> x * x
cube   = (x) -> square(x) * x

kids =
  brother:
    name: "Max"
    age:  11
  sister:
    name: "Ida"
    age:  9
```

```javascript
var cube, square, kids;

square = function(x) {
  return x * x;
};

cube = function(x) {
  return square(x) * x;
};

kids = {
  brother: {
    name: "Max",
    age: 11
  },
  sister: {
    name: "Ida",
    age: 9
  }
};
```

Quelle: http://coffeescript.org/#language

TypeScript: Beispiel

```typescript
interface Person {
  firstName: string;
  lastName: string;
}

function greeter(person: Person) {
  return "Hello, " +
    person.firstName + " " +
    person.lastName;
}

let user = {
  firstName: "Jane",
  lastName: "Doe"
};

console.log(greeter(user));
```

```javascript
function greeter(person) {
  return "Hello, " +
    person.firstName + " " +
    person.lastName;
}
var user = {
  firstName: "Jane",
  lastName: "Doe"
};
console.log(greeter(user));
```

- Häh? 🤨

Quelle: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html

TypeScript: Typsicherheit

- Fehlerhafter Code:

```
interface Person { firstName: string; lastName: string; }
function greeter(person: Person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}
let user = { first: "Jane", last: "Doe" };
console.log(greeter(user));
```

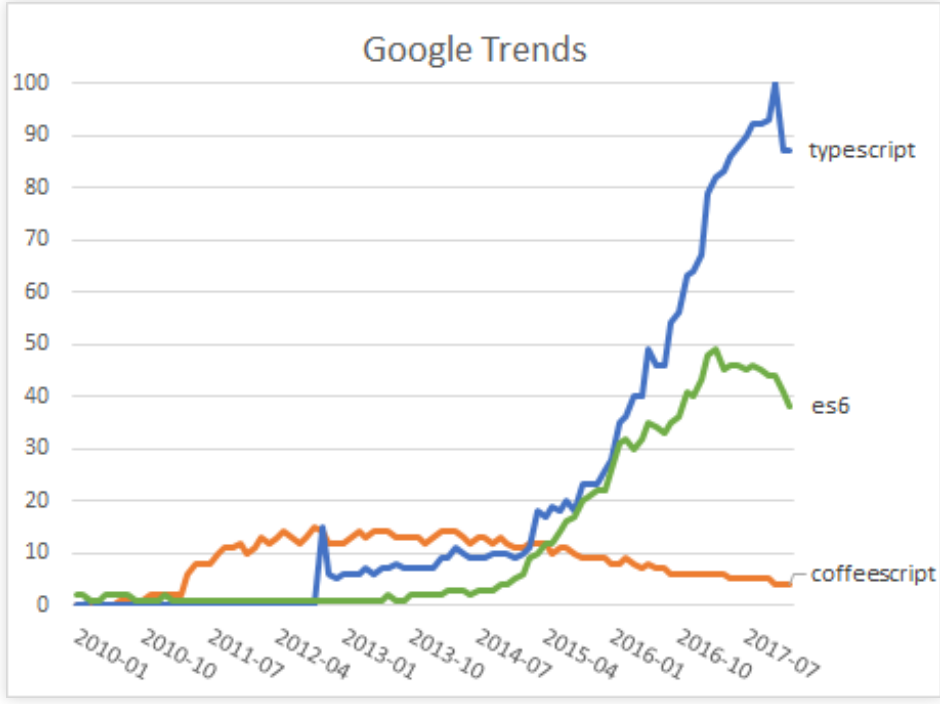
- Fehlermeldung beim Kompilieren:

```
$> tsc greeter.ts
greeter.ts(15,21): error TS2345: Argument of type
'{ first: string; last: string; }' is not assignable to parameter
of type 'Person'.
  Property 'firstName' is missing in type
  '{ first: string; last: string; }'.
```

Quelle: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html

Zusammenfassung: JS-Varianten

- *CoffeeScript* war eine zeitlang recht populär
  - einige Ideen in JavaScript Version *ES6* übernommen
  - damit viele Projekte wieder zu JavaScript zurück
- *TypeScript* wird z.B. für Angular benutzt
  - (siehe später in diesem Foliensatz)



Quelle: https://trends.google.com/trends/explore?date=2010-01-01%202018-01-17&q=coffeescript,typescript,es6

Entwicklung: JavaScript (2)

Wiederverwendung von Komponenten

Paket-Verwaltung in JavaScript

- bekannt aus PHP:
  - Verwaltung von Abhängigkeiten mit *composer*
    - `composer.json`: Verweise auf PHP-Bibliotheken
      - (Quelle: https://packagist.org/)
    - `composer install`: lädt diese herunter
    - `vendor/`: Zielordner
- analog für JavaScript:
  - zwei große Tools: *npm* und *yarn*
    - `package.json`: Verweise auf JS-Bibliotheken
      - (Quelle: https://www.npmjs.com/)
    - `npm install` bzw. `yarn install`: lädt diese herunter
    - `node_modules/`: Zielordner

Pakete benutzen in JavaScript

- Pakete installieren, z.B. Mathe-Paket *mathjs*

```
$> npm install mathjs
```

- Paket verwenden in Datei `calc.js`

```
var { gcd } = require('mathjs');
console.log("Größter gemeinsamer Teiler: " + gcd(357, 574));
```

- Aufruf in Node.js (stand-alone JavaScript) funktioniert

```
$> node calc.js
Größter gemeinsamer Teiler: 7
```

- Problem: Web-Browser kennen kein `require`

```
<!-- Datei `calc.html`. Das klappt nicht: -->
<script src="calc.js"></script>
```

Modulares JavaScript für Browser

- Lösung: Bundler, die aus vielen Dateien *eine* machen
  - Webpack (kennen wir schon aus Einheit *Optimierung*),
  - Browserify, oder Parcel, oder …
- Beispiel mit Parcel (simpel, schnell):

```
$> npm install parcel-bundler
$> parcel build calc.html
```

  - Ausgabe: `dist/calc.html`
    - minified, mit angepasstem `<script src="...">`
  - Ausgabe: `dist/calc.js`
    - minified, Abhängigkeiten gebündelt, für Browser
- Öffnen der `dist/calc.html`: Konsolen-Ausgabe

```
Größter gemeinsamer Teiler: 7
```

Zusammenfassung: JS-Pakete

- NPM-Verzeichnis: 475k Pakete
  - riesiges Ökosystem, große Auswahl
- Komponenten-basiert entwickeltes JS:
  - muss für Browser gebündelt werden
  - (selbst wenn Browser selbst Dateien laden könnte:
    - es wären *sehr* viele
    - *mathjs*: >800 einzelne JavaScript-Dateien)
  - bekannte Bundler sind: *Webpack* und *Browserify*

Entwicklung: JavaScript (3)

Frameworks im Frontend

Vue.js: Idee

- Teil-Baum im DOM wird von `Vue`-Instanz kontrolliert
  - `Vue`-Instanz ist Objekt mit Zustand (= Attribute) und Methoden
  - Werte der Attribute zur Anzeige im DOM verwenden
  - *Data-Binding*: Änderung an `Vue`-Instanz → DOM-Anpassung

```
<div id="app">{{ message }}</div>
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
<script>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hallo, Welt!'
  }
});
</script>
```

- Komplett im Browser (siehe oben)
  - Alternativ: Einbindung via NPM

Quelle: https://vuejs.org/v2/guide/index.html#Declarative-Rendering

Vue.js: Data-Binding

HTML

```html
<div id="app">
  {{ message }}
</div>
```

JS

```js
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hallo, Welt!'
  }
});
app.message = 'Hallo, X?';
```

Hallo, X?

Quelle: https://vuejs.org/v2/guide/index.html#Declarative-Rendering

Vue.js: Darstellung in der View

HTML

```html
<div id="app">
  <ul>
    <li v-for="zutat in zutaten">
      {{ zutat }}
    </li>
  </ul>
</div>
```

JS

```js
var app = new Vue({
  el: '#app',
  data: {
    zutaten:
      ['Milch','Eier','Zucker']
  }
});
```

- Milch
- Eier
- Zucker

Quelle: https://vuejs.org/v2/guide/index.html#Conditionals-and-Loops

Vue.js: Two-Way Binding

HTML

```html
<div id="app">
  <input v-model="message">
  <p>Message: {{ message }}</p>
</div>
```

JS

```js
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hallo, Welt!'
  }
});
```

- Änderungen im Model ↔ Änderung im DOM

Quelle: https://vuejs.org/v2/guide/index.html#Handling-User-Input

Einfaches Todora mit Vue.js

```
var todora = new Vue({
  el: '#todora',
  data: { todos: [], newTodo: '' },
  methods: {
    remove: function(todo) {
      this.todos.splice(
        this.todos.indexOf(todo),
        1);
    },
    add: function() {
      if (this.newTodo !== "") {
        this.todos.push({
          text: this.newTodo,
          checked: false
        });
        this.newTodo = "";
      }
    }
  }
});
```

```
<ul>
  <li v-for="todo in todos">
    <input type="checkbox"
           v-model="todo.checked">
    {{ todo.text }}
    <a v-on:click="remove(todo)">
      Löschen</a>
  </li>
</ul>
<form v-on:submit.prevent="add">
  <input v-model="newTodo"
         type="text">
</form>
```

- Anzeige, Hinzufügen, Löschen, Abhaken: 👍
  - Demo (inkl. Persistenz)

Zusammenfassung: Vue.js

- Übernimmt:
  - Data Binding (*one-way* oder *two-way*)
  - Templating
  - (und viel mehr, siehe Vue-Website)
- Erlaubt:
  - Objekt-Orientierte Entwicklung
  - Wiederverwendbare GUI-Komponenten
- Vergleichbare Frameworks:
  - **Angular** (Google) und **React** (Facebook)
  - Ausführlicher Vergleich dieser 3

Qualitätssicherung (1)

Backend: Unit-Tests

Unit-Tests in PHP

- *xUnit*: **jUnit** für Java, **PHPUnit** für PHP

```
$> composer require --dev phpunit   # Symfony-Integration
```

- Beispiel-Testklasse (Klasse `Calculator` ist zu testen):

```php
class CalculatorTest extends TestCase {
    public function testAdd() {
        $calc = new Calculator();
        $result = $calc->add(30, 12);
        // Ergebnis prüfen
        $this->assertEquals(42, $result);
    }
}
```

- Ausführung (in Symfony-Projekt, sonst *stand-alone*):

```
php bin/phpunit  # analog zu `php bin/console`
```

Quelle: https://symfony.com/doc/current/testing.html

Qualitätsicherung (2)

Backend: Funktionstests

Funktionstests

- betrachten Web-Anwendung aus Nutzerperspektive
  - in Symfony: technisch auch PHPUnit-Testklassen
  - aber: testen Zusammenspiel von Routing, Controller, View, …
- *Smoke Test*: Minimaler Funktionstest
  - *“Alle definierten URLs liefern einen Status 200.”*

```php
class SmokeTest extends WebTestCase {
    /** @dataProvider urlProvider */
    public function testPageIsSuccessful($url) {
        $client = self::createClient();
        $client->request('GET', $url);
        $this->assertTrue($client->getResponse()->isSuccessful());
    }
    public function urlProvider() {
        yield ['/'];
        yield ['/posts'];
        yield ['/post/fixture-post-1'];  // weitere URLs ...
    }
}
```

Quelle: https://symfony.com/doc/current/best_practices/tests.html#functional-tests

Funktionstests: Arten

- Unterschiedlich “teure” Testaufbauten
  - unabhängig von dem *was* geprüft wird
- Varianten:
  - **intern**: kein Webserver, HTTP-Anfragen simuliert
  - **extern**: Anwendung auf Webserver, Kommunikation über HTTP
    - **Browser-Dummy**: standard-konformer Client, keine GUI
    - **Browser-Tests**: echte Browser fernsteuern, Tests von Interaktion

Funktionstest: Intern, kein HTTP

- Symfony-Komponente `BrowserKit`

```
composer require --dev browser-kit css-selector
```

- In der Test-Methode (*ohne* laufenden Webserver):

```php
// HTTP-Anfrage an Symfony-Kernel simulieren
$crawler = $client->request('GET', '/post/hello-world');

// In der Antwort Link mit Text "Homepage" suchen
$link = $crawler->selectLink('Homepage')->link();
// Link anklicken
$crawler = $client->click($link);

// Testergebnis: Linkziel mit HTTP-Status 200
$this->assertTrue($client->getResponse()->isSuccessful());
```

- Crawler kann: lesen, Links und Formulare benutzen, HTTP-Header auswerten, …

Funktionstests: Extern, Dummy

- Goutte (vom Macher von Symfony)

```
composer require fabpot/goutte
```

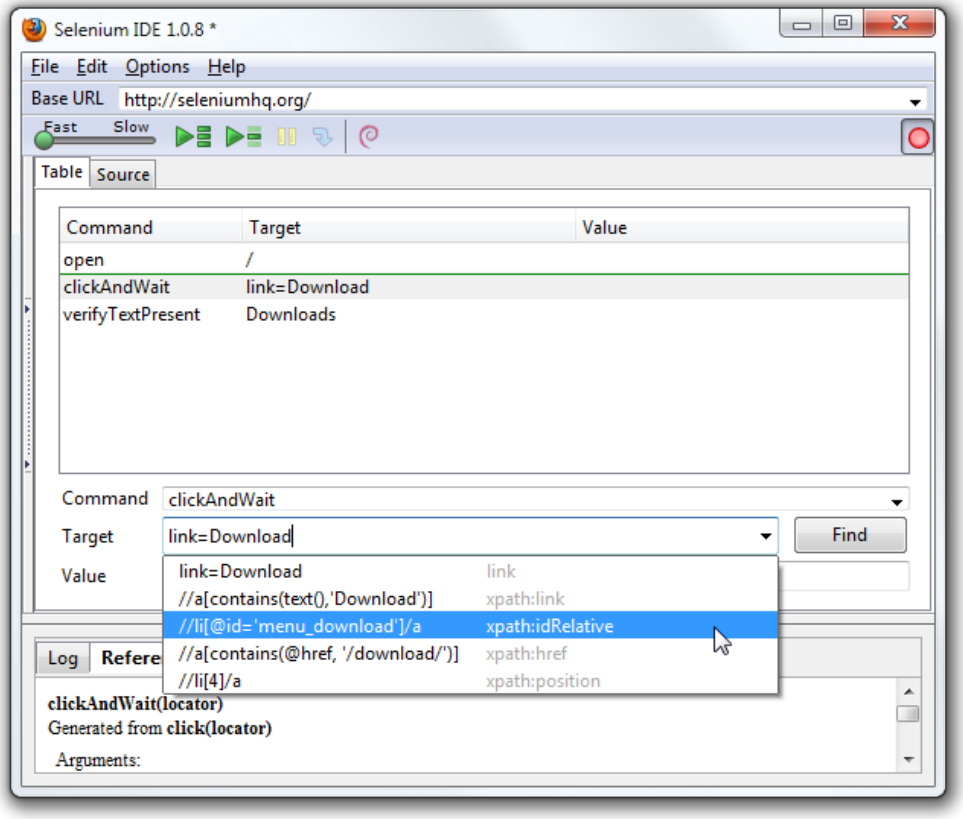
- *Gleiches* Interface wie `BrowserKit`:

```
// echte HTTP-Anfrage an (beliebigen) Webserver
$crawler = $client->request('GET', $host . '/post/hello-world');

// In der Antwort Link mit Text "Homepage" suchen
$link = $crawler->selectLink('Homepage')->link();
// Link anklicken
$crawler = $client->click($link);

// Testergebnis: Linkziel mit HTTP-Status 200
$this->assertTrue($client->getResponse()->isSuccessful());
```

- Aber: keine Interaktion auf geladener Seite
  - arbeitet syntaktisch auf DOM, nicht grafisch

Funktionstests: Browser-Treiber

- Benutzung echter Webbrowser, z.B. mit Tools wie Selenium
  - Nutzen Treiber um Browser fernzusteuern
    - Chrome, Firefox, Internet Explorer, Safari, …
- Grober Ablauf (*Record and Replay*):
  - Benutzung der Seite “aufzeichnen” → initiale Testfälle
  - Testfälle anpassen, mit Assertions anreichern
  - Testfälle automatisiert auf mehreren Browsern laufen lassen



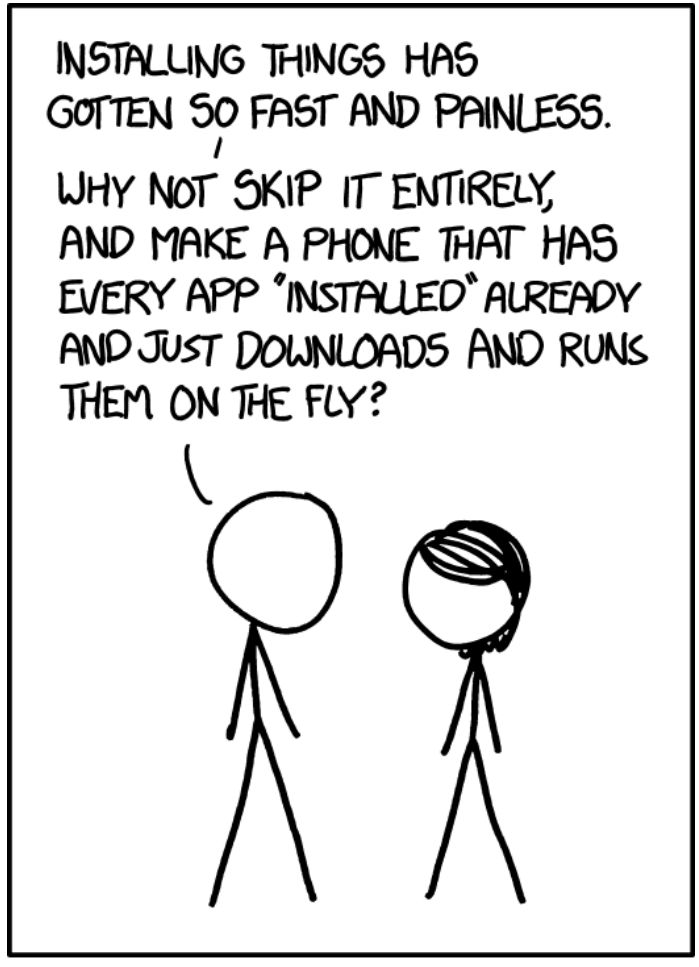
Quelle: http://www.seleniumhq.org/docs/02_selenium_ide.jsp#locator-assistance

Technologieauswahl

Ein paar Gedanken

Frontend-Backend-Grenze

- viele Möglichkeiten für Frontends:
  - npm: 475k Bibliotheken
  - (packagist für PHP: 170k)
- Wie viel ins Frontend?
  - Kein JavaScript im Browser: Request/Response, oder
  - Gelegentlich mal AJAX-Requests, oder
  - Alles im Browser, außer “Installation” und Persistenz/Wirkung



Quelle: Source: https://xkcd.com/1367/

Frontend-Backend-Grenze

- **Entwicklung**:
  - Komplexer mit Frontend-Build-Schritt
    - Gewicht Stack-abhängig:
      - Node.js: serverseitiges JavaScript im Backend
      - GWT: Frontend in Java schreiben, kompiliert zu JS
  - Ökosystem unübersichtlich
- **Funktionale Anf.**:
  - Browser-APIs (Geolokation, Batterie, LocalStorage)
  - Wirkung (Datenbank, Rechenleistung)
- **Sicherheit**:
  - Vertraulichkeit:
    - Daten & Logik an Client?
  - Soziotechnik: “Kultur”
    - Verfügbarkeit (Fall “left-pad”)
    - Vertraulichkeit (XSS via npm)
- **Betrieb**:
  - Volumen vs. Requests:
    - MBytes JavaScript an Client?
  - Rechenleistung:
    - Schwergewichtige Ausführung im Client?

Frontend: (noch?) relativ instabil

- Ping-Pong zwischen zwei Kultur-Extremen:
  1. Zentralisiert: **“batteries included”**
     - direkt loslegen mit Bibliothek/Framework
     - wenig Konfiguration/-möglichkeiten
  2. Dezentralisiert: **“configure everything”**
     - flexibel, viele Möglichkeiten
     - Konfiguration (und evtl. weitere Teile) nötig
- Beispiele (*configure everything* ↔ *batteries included*):
  - Framework: React ↔ Angular
  - CSS-Verarbeitung: PostCSS ↔ SASS
  - Bundler: Webpack ↔ Parcel

Quelle: https://blog.logrocket.com/frontend-in-2017-the-important-parts-4548d085977f

Aktuelle JS-Entwicklungs-Kultur

- typisch: extrem kleine Module & viele Abhängigkeiten
- **Frontend**-Bibliotheken der Veranstaltungswebsite:
  - `package.json`: 3 Einträge (SASS, Webpack für Bundle & Minify)
  - `node_modules/`:
    - 735 Ordner auf 1. Ebene
    - 14.119 Dateien (69,6 MByte)
- **Backend**-Bibliotheken:
  - `composer.json`: 17 Einträge (Symfony komplett und viele Extras)
  - `vendor/`:
    - 55 Ordner auf 2. Ebene
    - 6.986 Dateien (25,4 MByte)
- Siehe auch: *NPM: Have We Forgotten How To Program?*

Vergleich: PHP-Kultur

- wenige, größere, gereifte Komponenten
- Beispiel: Größter Symfony-“Konkurrent” Laravel
  - 13% eigener Code, 87% wiederverwendete Komponenten
  - **30%: Symfony-Komponenten**
- Symfony-Komponenten in anderen PHP-Projekten:
  - Drupal, Joomla, Typo3, und weitere CMSs
  - Lavaral, Yii, und weitere Frameworks
  - Google APIs Client Library und weitere SDKs
  - phpMyAdmin, Piwik, phpBB und selbst Composer

Quelle: https://medium.com/@javiereguiluz/30-of-laravel-code-is-symfony-a49dcf30e809

Problem: JS-Abhängigkeiten

- Bibliothek *Ember.js* (Stand: August 2016):
  - Gesamtgröße: 112 kBytes (gzip)
  - davon 95 kByte (gzip) für “Glimmer”
  - davon 93 kByte (gzip) für die *Encyclopedia Brittanica* (!)
  - *nur* für die Begriffsdefinition des Wortes *“glimmer”*
  - für eine Konsolen-Ausgabe von `glimmer.help()`
- **Meine Bitten an Sie**:
  - Seien Sie für einen solchen Scherz nicht verantwortlich
  - Seien Sie bei einem solchen Ökosystem zumindest *skeptisch*
- Es folgen: Zwei Beispiele der Früchte, die das trägt

Quelle: https://medium.com/friendship-dot-js/i-peeked-into-my-node-modules-directory-and-you-wont-believe-what-happened-next-b89f63d21558

Der Fall `left-pad`

- Entwickler 273 kleiner JS-Module unzufrieden mit NPMs Firmenpolitik
  - entfernt alle von NPM
  - darunter `left-pad` (⟶)
    - 11 Stars auf GitHub
    - aber: >2 Mio Downloads/Monat

```
module.exports = leftpad;
function leftpad (str, len, ch) {
  str = String(str);
  var i = -1;
  if (!ch && ch !== 0) ch = ' ';
  len = len - str.length;
  while (++i < len) {
    str = ch + str;
  }
  return str;
}
```

- *tausende* Projekte konnten nicht mehr gebaut werden
  - meistens indirekte Abhängigkeiten; *React*, *Babel*, …
- Ersatz zwar innerhalb von 2½ Stunden
  - aber immense weltweite Verwirrung, unbezifferter Schaden

Quelle: https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/

XSS via npm

- Idee zur Verbreitung auf *tausenden* Websites:
  - kleines attraktives Modul als Trojanisches Pferd
    - z.B. bunte `console.log()` Ausgaben
  - sammeln: Inputs (`password`, `cardnumber`, …) & `document.cookie`
  - AJAX-Request mit Daten an eigenen Server
- Entdecken?
  - Heisenberg-Manöver: inaktiv wenn DevTools aktiv sind
  - inaktiv auf `localhost`, u.ä.; inaktiv zwischen 7 und 19 Uhr
  - nur bei ca. ⅐ der Aufrufe aktiv; pro Client nur einmalig
  - sauberer Code auf GitHub; minified, uglified auf NPM
- Abwehr:
  - für kritische Seiten (wie Login): *keine* JS-Abhängigkeiten

Quelle: https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5

Zusammenfassung

- Browser sind durch CSS und JavaScript eingeschränkt
  - Idee: Neue Sprachen/-features erfinden und CSS/JS erzeugen
  - CSS: Less/SASS
  - JavaScript: CoffeeScript/TypeScript, Modularisierung (und Bündelung), Frameworks für strukturierte Entwicklung
- Qualitätssicherung (des Backends):
  - Unit-Tests
  - Funktionstests (intern, extern mit Dummy- und echtem Browser)
- Soziotechnik (relevant für Technologieauswahl):
  - zu einer Technologie gehört die Community untrennbar dazu

Danke!