Web-basierte Systeme – Übung

02: JavaScript Teil 2,Browser API, DevTools

Wintersemester 2024

Arne Vogel, Maxim Ritter von Onciul

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Übersicht

JavaScript, Teil 2

Übersicht

JavaScript: single-threaded

- JavaScript ist **single threaded**
- Ausnahme: mit einem **WebWorker** können Skripte im Hintergrund ausgeführt werden
- **Problem:** Wie Hardware auslasten?
- **Lösung:** Asynchronität bzw. non-blocking I/O
  - Callbacks
  - Promises
  - `async/await` (ab ES7)
- Beispiele für asynchrone/blockierende Funktionen
  - Datei lesen/schreiben
  - Netzwerk, z.B. DNS-Auflösung oder HTTP Requests
  - Datenbankabfrage

Exkurs: Callbacks

- Eine Funktion die als Argument an eine Funktion übergeben wird

```
1  const appendASmile = (text) => text + ' :)';
2
3  function log(text, callback) {
4    // Flashback zu truthy&falsy!
5    if(callback) text = callback(text);
6    console.log(text);
7  }
8  log("Hello World!");
9  log("Hello World!", appendASmile);
```

- Callbacks können auch anonym sein

```
1  // Machen genau das selbe:
2  log("Hello World!", (text) => text + ' :)');
3  log("Hello World!", function(text) { return text + ' :)'; });
```

Exkurs: Callbacks

- Eine Funktion die als Argument an eine Funktion übergeben wird

```
const appendASmile = (text) => text + ' :)';

fu
}
log
log
```

```
a(function (resultsFromA) {
  b(resultsFromA, function (resultsFromB) {
    c(resultsFromB, function (resultsFromC) {
      d(resultsFromC, function (resultsFromD) {
        e(resultsFromD, function (resultsFromE) {
          f(resultsFromE, function (resultsFromF) {
            console.log(resultsFromF);
          })
        })
      })
    })
  })
});
```

- Ca

```
// Machen genau das selbe:
log("Hello World!", (text) => text + ' :)');
log("Hello World!", function(text) { return text + ' :)'; });
```

Promises

- Objekt zum Arbeiten mit asynchronen Operationen
- "Verspricht " eine Operation auszuführen, jedoch kann diese auch fehlschlagen
- 3 Zustände:
  - `pending`: sprich die asynchrone Operation wird noch ausgeführt
  - `resolved`: sprich die Operation ist erfolgreich abgeschlossen
  - `rejected`: sprich die Operation ist gescheitert

then/catch

- **then** kann auf Promise Objekte aufgerufen werden
  - Übergabe eines Callbacks, um mit dem Promise weiterzuarbeiten
  - Innerhalb `then` ist das Promise `resolved` wodurch das Ergebniss der Operation verfügbar ist
- **catch** wird aufgerufen, falls ein Promise `rejected`

```
1  asyncOperation()
2    .then(result => console.log(result))
3    .catch(console.error)
```

Vorteil then/catch

- Wozu das Ganze?

```
asyncOperation()
  .then(result => anotherAsyncOperation(result)
                  .then(result => console.log(result))
                  .catch(error => /* Handle another error */))
  .catch(error => /* Handle error */)
```

Vorteil then/catch

- Wozu das Ganze?

```
asyncOperation()
  .then(result => anotherAsyncOperation(result)
                  .then(result => console.log(result))
                  .catch(error => /* Handle another error */))
  .catch(error => /* Handle error */)
```

- **Promise Chaining**: Promises können auch gereiht werden!

```
asyncOperation()
  .then(result => anotherAsyncOperation(result))
  .then(result => console.log(result))
  .catch(error => /* Handle any error */)
```

async/await

- Syntactic Sugar für Promises, ermöglicht klassischen Programmierstil
- Funktionen mit dem **async** Schlüsselwort können **await** benutzen
- **await** pausiert den Code bis das Promise abgeschlossen ist
- **async**-Funktionen geben immer ein Promise zurück!

```
1  async function runAnAsyncOperation(){
2      try {
3          const result = await asyncOperation();
4          return result;
5      }
6      catch(error) { /* Handle the error */ }
7  }
8
9  const result = runAnAsyncOperation();
```

- Der **try-catch** Block ist noch(!) optional, sollte aber immer benutzt werden.

Asynchronität im Code

- Während Promises ausgeführt werden, kann weiterer Code laufen
- Ausführungsreihenfolge kann etwas komplizierter sein: Reihenfolge ist nicht eindeutig

```
1  fetchDataFromGoogle().then(console.log)
2  fetchDateFromFacebook().then(console.log)
```

Reihenfolge: 1 -> 4 -> 3 -> 2

```
1  console.log('Synchronous 1');
2  // "warte" 0 Sekunden
3  setTimeout(_ => console.log('Timeout 2'), 0);
4  // löse das Promise direkt auf
5  Promise.resolve().then(_ => console.log('Promise 3');
6  console.log('Synchronous 4');
```

Promises & bad practices

- Callback Hell als Paradebeispiel wie man es nicht machen sollte!

```
connectToDatabase().then((db) => {
  return db.getUser('Tom').then((user) => {
    return getUserCredentials(user).then((credentials) => {
      return credentials.username === loginData.username &&
             credentials.password === loginData.password
    });
  });
});
```

- Mit then/catch bevorzugt Promise Chaining verwenden um Vernestungen zu vermeiden.

Promises & bad practices

- async/await kann unnötige Wartezeiten einfügen!

```
// both requests can be run concurrently so no
// need to wait for one before running the other
const result1 = await someHttpRequest();
const result2 = await httpRequestUnrelatedToResult1();
```

- Lösung: Warten, bis alle Promises erfüllt sind
- In welcher Reihenfolge diese erfüllt werden ist egal

```
const [result1, result2] = await Promise.all([
  someHttpRequest(),
  httpRequestUnrelatedToResult1()
]);
```

- Immer im Hinterkopf behalten, dass await euren Code **pausiert**!

Browser API

Übersicht

Window Objekt

- Beinhaltet alle Informationen und Schnittstellen zu einer Seite
- Ermöglicht Nutzerinteraktion

```
1  // opens an alert box with message
2  window.alert(message);
3  // opens confirmation window for user
4  result = window.confirm(message);
5  // opens prompt for user
6  result = window.prompt(message);
```

- Definierte Variablen/Funktionen sind dort auch hinterlegt

```
1  const test = "Hello World";
2  console.log(test === window.test);
```

https://developer.mozilla.org/docs/Web/API/Window

setTimeout

- Ruft einen Callback nach angegebener Zeit einmalig auf

```
1  const log = () => console.log("Hello World!");
2  // loggt nach 1s
3  setTimeout(log, 1000)
```

- Timeout kann mittels ID abgebrochen werden

```
1  const timeoutID = setTimeout(log, 1000);
2
3  // timeout gecleared also wird nichts geloggt
4  clearTimeout(timeoutID);
```

Quelle: https://www.w3schools.com/js/js_timing.asp

setInterval

- Wie setTimeout, nur in einem Intervall

```
1  const log = () => console.log("Hello World!");
2  // loggt jede Sekunde
3  setInterval(log, 1000)
```

- Intervall kann auch abgebrochen werden

```
1  const intervalId = setInterval(log, 1000);
2
3  // wartet 5 Sekunden, bis der Intervall gecleared wird
4  setTimeout(() => clearInterval(intervalId), 5000);
```

Quelle: https://www.w3schools.com/js/js_timing.asp

JSON

- JavaScript Object Notation (JSON) häufig genutztes Datenformat
- Häufig sprechen REST APIs **nur** JSON (und XML)

```
{
  "Nummer": "1234-5678-9012-3456",
  "Waehrung": "EURO",
  "BetragInCent": 100000,
  "Inhaber": {
    "Name": "Mustermann",
    "Vorname": "Max",
    "maennlich": true,
    "Kreditwuerdigkeit": 9.2,
    "Kinder": ["Emma", "Gustav"]
  }
}
```

- Newlines nur zur besseren Lesbarkeit, also optional!

JSON Parsing

- Browser API bietet JSON Objekt zum parsen von JSON Strings
- Entsprechend kann man auch JSON zu einem String umwandeln

```
1  const jsonString = '{"a": 5, "b": 7}';
2
3  // format to json
4  const items = JSON.parse(jsonString);
5
6  for(let [key, item] of Object.entries(items)) {
7    console.log(key, item);
8  }
9
10 // back to String
11 const jsonToString = JSON.stringify(items);
```

Ajax

- Ajax: **Asynchronous JavaScript and XML**
  - Begriff XML historisch bedingt
  - Heute wird meist JSON benutzt, da nativ in JavaScript unterstützt
- Sammlung von mehreren Technologien zur Entwicklung von **asynchronen Webanwendungen**
- Daten können im Hintergrund geladen und in die Seite eingefügt werden, ohne die Seite neuladen zu müssen.
- Begriff geprägt durch Aufsatz von Jesse James Garrett 2005, “Ajax: A New Approach to Web Applications”[1]

[1] http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

Ajax: Technologien

“Ajax ist eine Sammlung von mehreren Technologien”

- **HTML und CSS** zur Repräsentation
- **DOM** für dynamische Darstellung von / Interaktion mit Daten
- **JSON** (früher XML) für den Datenaustausch
- Das `XMLHttpRequest`**-Objekt** für asynchrone Kommunikation
- **JavaScript** um alles zu verbinden

Status Codes

- Zeigen den Status[2] von Antworten eines HTTP Request an
- Aufgeteilt in versch. Bereiche:
  - Informational responses (100–199)
  - Successful responses (200–299)
  - Redirects (300–399)
  - Client errors (400–499)
  - Server errors (500–599)

[2] https://developer.mozilla.org/de/docs/Web/HTTP/Status

Ajax: Requests

- JS bietet mehrere Möglichkeiten, um mit HTTP Requests zu arbeiten
- Vanilla JS stellt XHR (XMLHttpRequest) und fetch zur Verfügung
- Es gibt etliche weitere Module, die je nach Präferenz, das arbeiten mit Requests verschönern
- **Tipp**: spielt mit Requests herum und loggt die Responses um den Aufbau zu verstehen

Ajax: XHR

- XHR arbeitet mit Events (onreadystatechange)
- Für kleine Requests eher umständlich, kann aber Vorteile beim debuggen haben

```
1  const xhr = new XMLHttpRequest();
2  xhr.onreadystatechange = function() {
3    if (this.readyState == 4 && this.status == 200) {
4      document.getElementById("element").value = this.responseText;
5    }
6  };
7  xhr.open("GET", "/location");
8  xhr.send();
```

readyState:
UNSENT (0), OPENED (1), HEADERS_RECEIVED (2), LOADING (3), DONE (4)

Ajax: fetch

- Überschaulichere Version von XHR
- Arbeitet mit Promises!

```
1  // get Request
2  fetch(someUrl)
3    .then(response => response.json())
4    .then(console.log);
5
6  // post Request
7  fetch(someUrl, {
8    method: 'POST',
9    body: JSON.stringify({ name: "Tom" }),
10 }).then(/* continue as normal */);
11
```

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Ajax: fetch

- Status Codes 4XX & 5XX landen nicht automatisch im catch Block
- Error muss in dem Fall manuell geworfen werden

```
1  function checkError(response) {
2    if (response.status >= 200 && response.status <= 299) {
3      return response;
4    } else {
5      throw Error(response.statusText);
6    }
7  }
8
9  fetch(someUrl)
10   .then(checkError)
11   .then(/* continue as normal */)
12   .catch(console.error)
```

Browser Developer Tools

Übersicht

Browser Developer Tools

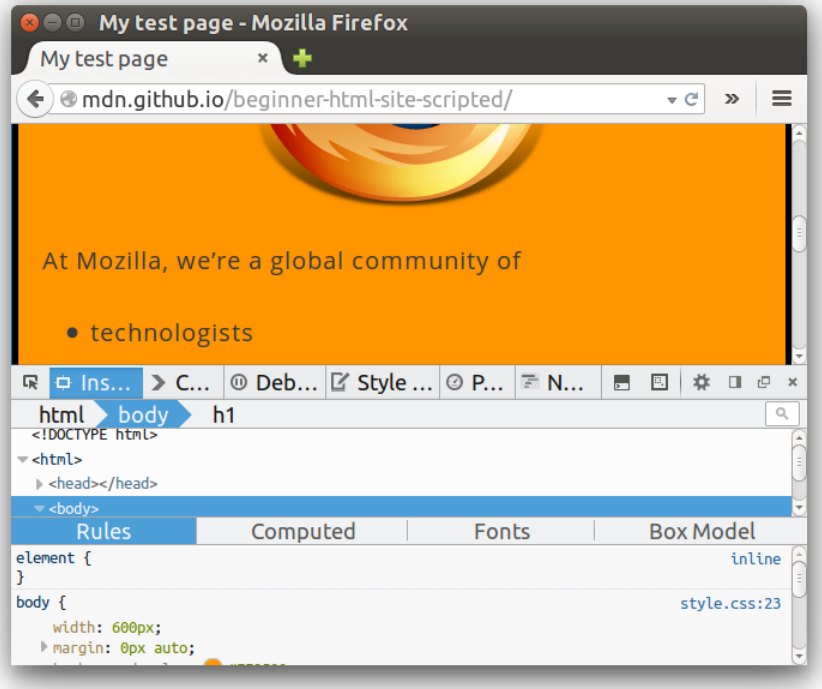
- Jeder moderne Webbrowser enthält leistungsstarke Entwicklertools
  - Firefox: “browser developer tools”
  - Chrome: “Chrome DevTools”
  - Safari: “developer tools”

  $\rightarrow$ Alle funktioneren ähnlich

  $\rightarrow$ Hier wird **Firefox** exemplarisch vorgestellt

- Die Werkzeuge haben mehrere Aufgaben:
  - Inspektion von geladenem HTML, CSS und JS
  - Bestandteile der Webseite und deren Ladezeiten
  - Debugging
  - Performance Analyse
  - ...

Browser Developer Tools

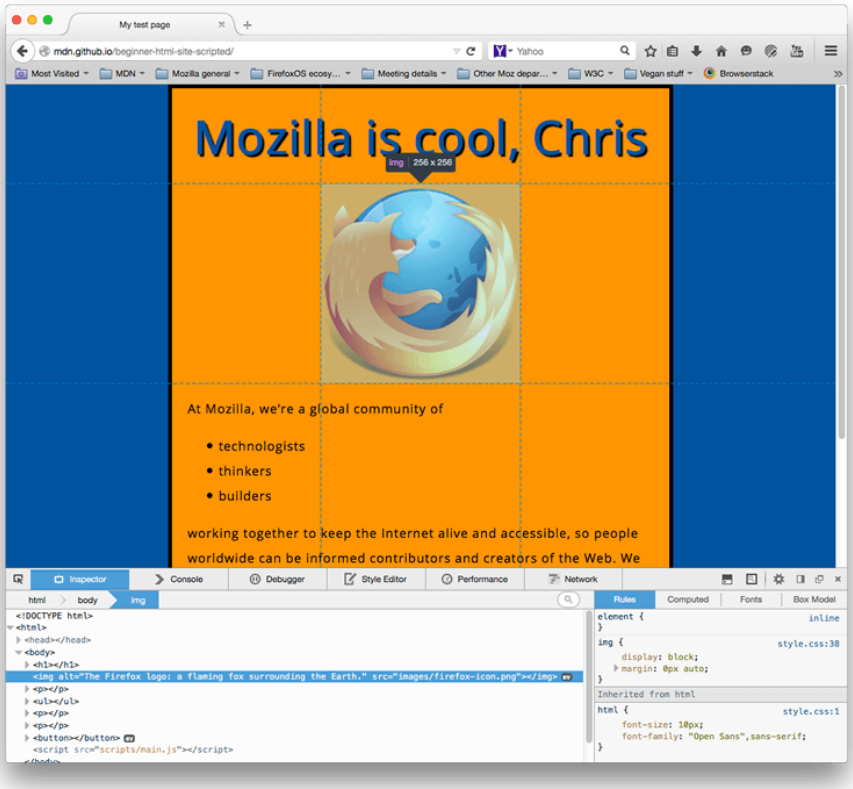


- Darstellung als Unterfenster
- Ctrl + Shift + I oder F12

Browser Developer Tools

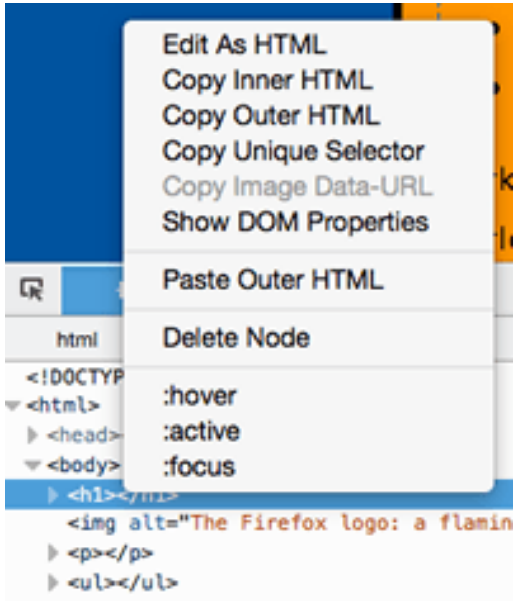
- Darstellung als Unterfenster
- Ctrl + Shift + I oder F12
- Kontext-Menü

Browser Developer Tools



- "Inspector" Tab:
  DOM Betrachter/Editor
- Änderungen im DOM
  (HTML/CSS/JS) sofort sichtbar

Browser Developer Tools

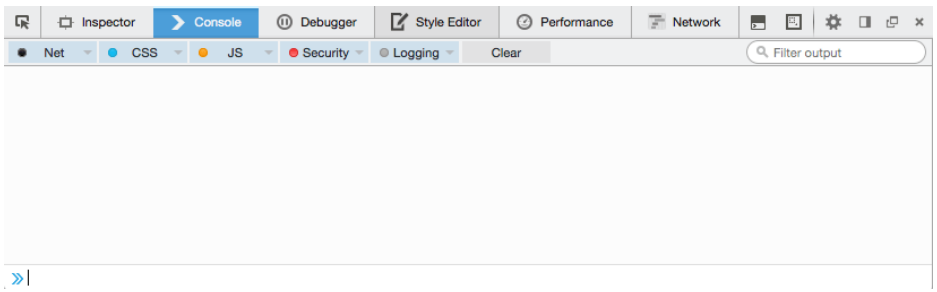


- “Inspector” Tab:
  DOM Betrachter/Editor
- Änderungen im DOM
  (HTML/CSS/JS) sofort sichtbar
- Kontextmenü von
  DOM-Elementen zeigt mögl.
  Operationen

Browser Developer Tools

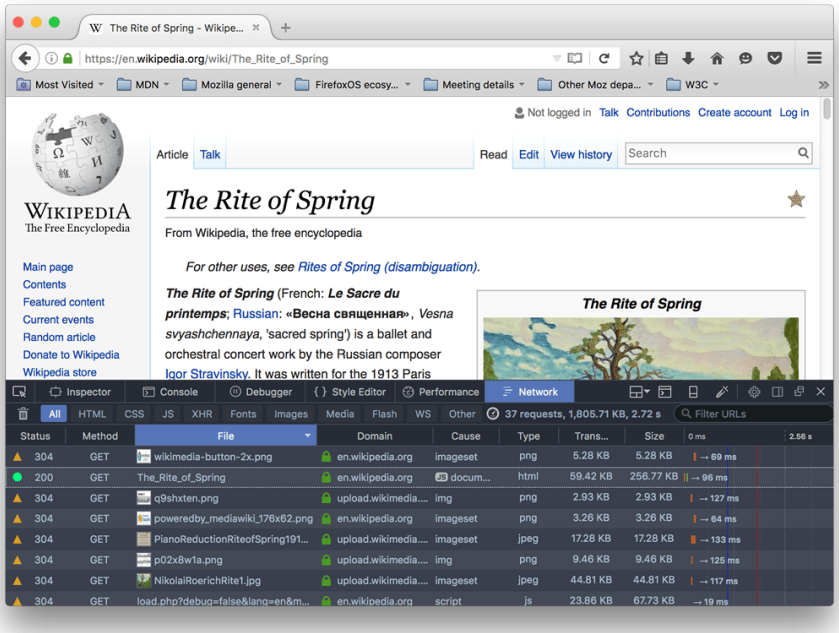
- “Inspector” Tab:
  DOM Betrachter/Editor
- Änderungen im DOM (HTML/CSS/JS) sofort sichtbar
- Kontextmenü von DOM-Elementen zeigt mögl. Operationen
- CSS Editor zeigt CSS Regeln des aktuellen Dokuments

Browser Developer Tools



- “Console” Tab: JS Konsole
- Sehr nützlich zum Debuggen
- Unterstützung von Filtern

Browser Developer Tools



- “Network” Tab: Zeigt alle für die Seite nötigen HTTP Requests und Metainformationen:
- HTTP Status Code und Methode
- Pfad/Datei
- Grund für den Request
- Dateityp
- Dateigröße
- Zeitverlauf

Node-RED

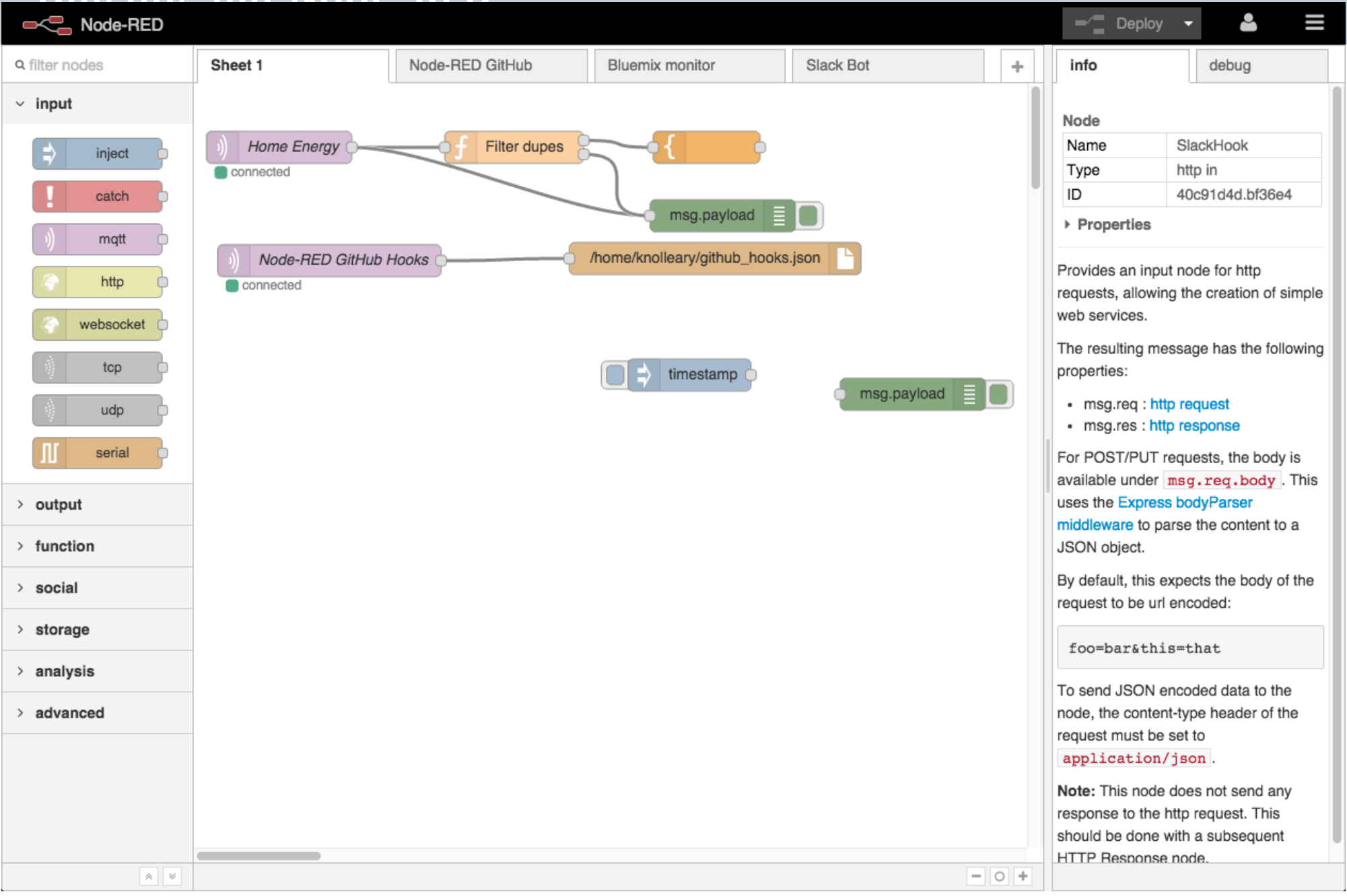
Node-RED Übersicht

- **Graphisches Entwicklungswerkzeug**, ursprünglich für IoT entwickelt
- Runtime basiert auf Node.js
- **Flow-based** („Fluss-basiert“)
- Für die erste Aufgabe wird der Server als „Node-RED Flow“ vorgegeben
- **Gefährlich!** Benutzt auf jeden Fall `web-sys-red.sh`

Umgang mit Node-RED

- Server starten: `./web-sys-red.sh server.json`
- Client im Browser öffnen `http://localhost:1880/chat`
- Node-RED Editor: `http://localhost:1880` (nicht wichtig für Aufgabe 1)

Umgang mit Node-RED

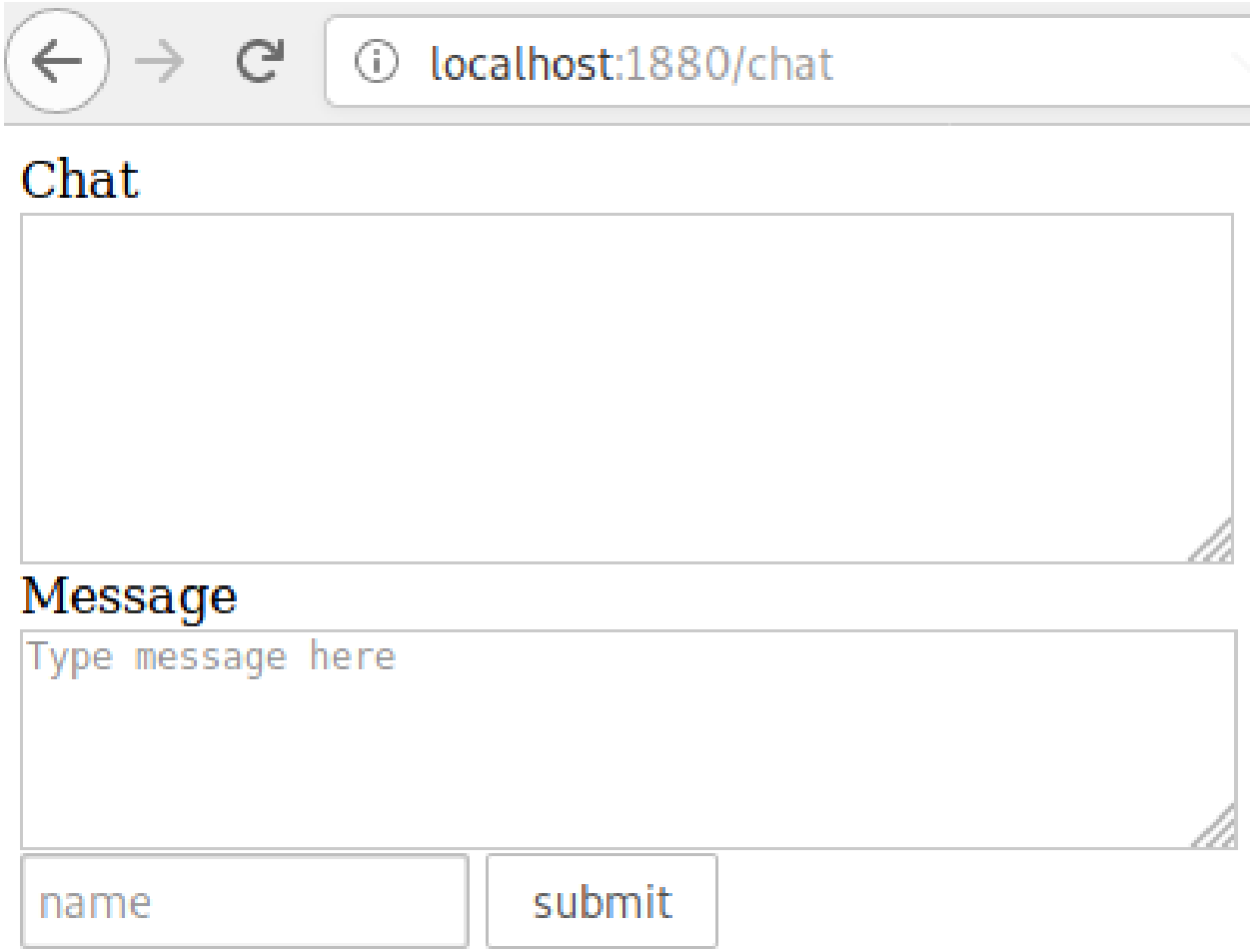


Aufgabe 1

Übersicht

Aufgabe 1.1: Schlanker Chat-Client (1/2)

- Entwicklung eines Chat-Clients auf Basis von HTML und JavaScript

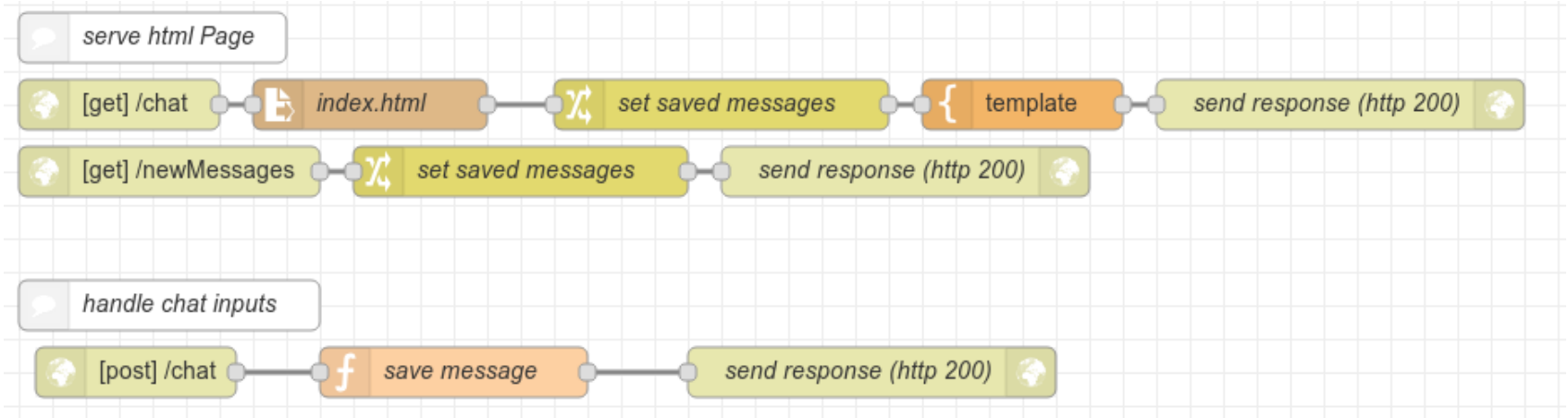


Aufgabe 1.1: Schlanker Chat-Client (2/2)

- Verwenden Sie Git! Wir empfehlen regelmäßige Commits und erwarten **mindestens** einen Commit pro Teilaufgabe
- Server wird als Node-RED[3] Flow vorgegeben
- Client wird als reine HTML-Datei wird vorgegeben, enthält kein JavaScript!
  - Funktionsfähig, aber Seite muss manuell neu geladen werden
- **Aufgabe:** Client soll um JavaScript erweitert werden
  - Regelmäßiges Abfragen von neuen Nachrichten mit Ajax im Hintergrund

[3] `https://nodered.org/`

Node-RED Flow für Aufgabe 1



- HTTP GET `/chat`: Response enthält Inhalt von `index.html`, inkl. gespeicherter Nachrichten
- HTTP POST `/chat`: Erwartet im Body die Variablen `messageInput` und `nameInput`
- HTTP GET `/savedMessages`: Response enthält gepeicherte Nachrichten
  - Format? Teil von Aufgabe 1 ist es, dies zu bestimmen!

Node-Red Sicherheit

- Password: *node-red admin init*