

Blockwoche: Web Programming Lab



Check-in



Check-in: Über Patrick

- Tech-Lead, Software-Architekt, Software-Entwickler bei [Fellow GmbH](#) 
- Co-Founder [Adhook GmbH](#) mit 
- Dozent Web Programming Lab, [CAS Software Architecture](#) und [CAS Digital Architect](#) an der HSLU

Schwerpunkte:

- Software Architektur & Engineering von Web- & Cloud-Anwendungen
- Web Technologien, Java, Docker, Cloud
- Tools für Software- & Requirements-Engineering

Kanäle:

- E-Mail: patrick.roos@fellow.gmbh
- [LinkedIn](#) | [Twitter](#) | [GitHub](#)
- Blog und Tech-Newsletter: [workingsoftware.dev](#)



Check-in: Über Andreas

- Co-Founder Adhook GmbH



Schwerpunkte:

- Produktentwicklung
- Begleitung Kundenprojekte
- Software Architektur & Engineering
- Digital Marketing Consulting

Kanäle:

- E-Mail: andreas@adhook.io
- [LinkedIn](#) | [Twitter](#) | [GitHub](#)



ALWAYS START with

WHY

Check-in: Always Start With Why (5')

- Warum habt Ihr Euch in die Web Programming Lab Blockwoche eingeschrieben?
- Habt ihr bereits **Fragen** zur Blockwoche?

[Link auf das Mural Board](#)

Check-In: Warum dieses Modul?



The image shows the landing page for the 2022 Developer Survey. It features a dark blue header with the Stack Overflow logo (orange and grey stylized bars) and the text "2022 Developer Survey". Below the header, there is a large white text block stating: "In May 2022 over 70,000 developers told us how they learn and level up, which tools they're using, and what they want." At the bottom, there are two buttons: "Read the overview →" and "Methodology →".



<https://survey.stackoverflow.co/2022/>

Programming, scripting, and markup languages



Rust is on its seventh year as the most loved language with 87% of developers saying they want to continue using it.

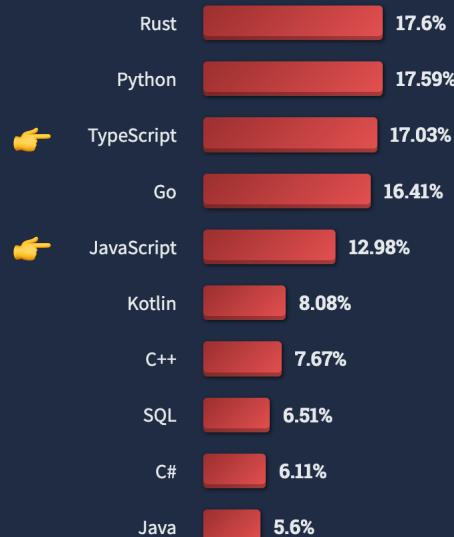
Rust also ties with Python as the most wanted technology with TypeScript running a close second.

Loved vs. Dreaded

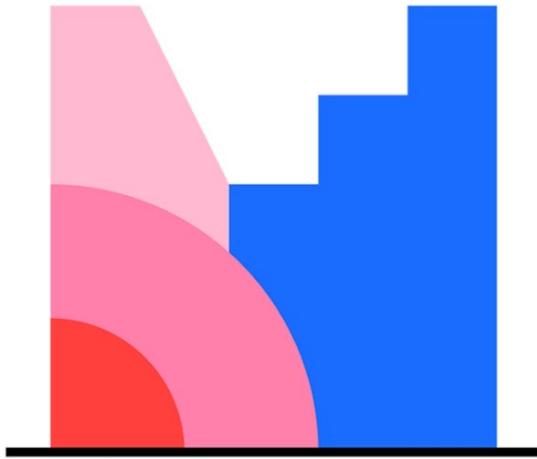
Want

71,467 responses

% of developers who are not developing with the language or technology but have expressed interest in developing with it



Check-in: Eure bisherigen Kenntnisse



Mentimeter

Check-in: Programm Blockwoche

Montag 	Dienstag  	Mittwoch  	Donnerstag   	Freitag   
Architekturansätze von Web Anwendungen JavaScript Sprachkonzepte	Client-Side-JavaScript I	Angular	Angular	Progressive Web Apps
JavaScript Sprachkonzepte	Client-Side-JavaScript II	Angular	Server-Side-JavaScript	Authentication @ Web Apps

Check-in: Modul Ziele

Die Studierenden können ...

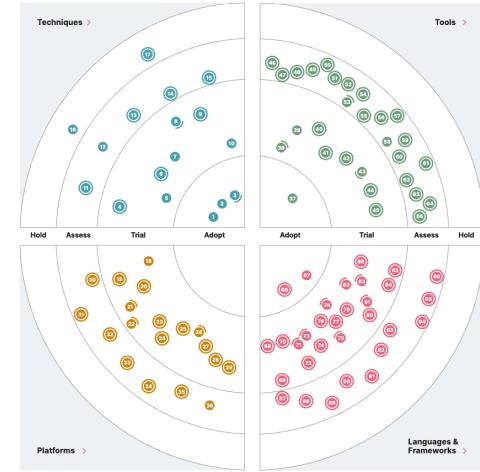
- ✓ eine Webapplikation mit modernen Architekturansätzen und Technologien umsetzen (F1).
- ✓ eine Webapplikation testen und debuggen (F2).
- ✓ kennen die üblichsten Bedrohungsszenarien für Webapplikationen. (F3)
- ✓ können ein Webapplication Deployment planen. (F4)
- ✓ eine grösseres Web-Projekt durchführen und die Resultate präsentieren. (M1)
- ✓ selbstständig Wissenslücken recherchieren und schliessen. (P1)
- ✓ Themen selbstständig erarbeiten. (P2)

Check-in: Zeiten

- Pro Tag acht Lektionen
- Vormittag: 08:30 – 12:00
- Nachmittag: 13:00 – 16:30
- Pausen: Pro Halbtag 20min + 2 x 5'

Check-in: Web Programming Lab Projekt

- Einzelarbeit
- Projekt “Technologie-Radar” oder eigene Projekt-Idee
- Alle Informationen zum Web Programming Lab Projekt
- Fokus Software Engineering mit Web Technologien
 - > Idealerweise mit Technologien ohne Praxiserfahrung
 - > inkl. (lokales) Deployment & Testing
- Bis **Mittwoch Abend via Ilias (8. Februar 2023)**
 - > Projektbeschreibung (für eigenen Projektvorschlag),
 - > Technologien
 - > Abgrenzungen



Check-in: MEP

- **Projekt-Ergebnisse (ohne Präsentation):**

Abgabe bis Mittwoch, 08. März 2023 18:00

- **Projekt-Präsentation:**

Samstag, 11.03.2022 08:30 – ca. 11:00

(5 Minuten Präsentation, 1 – 2 Minuten Fragen)

Check-in: Wie arbeiten wir im Kurs?

- Inputs
 - Übungsfragen- und Aufgaben (Break-out Sessions, Ask for help)
 - Falls Input schon bekannt und/oder Aufgaben gemacht → Zeit für Projektarbeit
- Feedback und Inputs zum Aufbau und den einzelnen Inhalten sehr erwünscht! ☺

Fragen zur Blockwoche?

Check-in: Programm Blockwoche

Montag 	Dienstag  	Mittwoch  	Donnerstag   	Freitag   
Architekturansätze von Web Anwendungen JavaScript Sprachkonzepte	Client-Side-JavaScript I	Angular	Angular	Progressive Web Apps
JavaScript Sprachkonzepte	Client-Side-JavaScript II Frameworks & Typescript	Angular	Server-Side-JavaScript	Authentication @ Web Apps

Ziele für heute

- ✓ Wir haben einen **Überblick** über Ansätze und **Architekturen von Web Anwendungen**.
- ✓ Wir kennen die gängigsten **JavaScript-Sprachkonzepte** und können diese anwenden.

Agenda für heute

- **Ansätze und Architekturen von Web Anwendungen (Vormittag)**
- **JavaScript Sprachkonzepte (Vormittag / Nachmittag)**
 - > Eigenschaften, History und Versionen, Setup Dev-Umgebung
 - > Variablen, primitive Datentypen, Operatoren
 - > Objects, Arrays, Functions, Lexical Scope
 - > Prototype Chains
 - > Classes
 - > Promises, Async & Await

Grundlegende Ansätze & Architekturen von Web Anwendungen

Qualitätsziele, Architekturansätze von Web Anwendungen



Looking for love in all the wrong frameworks



Hype Driven Development

Life on the Bandwagon

O RLY?

@ThePracticalDev

Erwartungen an heutige Web Applikationen

“Wir generieren dem User den **Mehrwert** xyz...”

“Die Webanwendung muss **24/7** verfügbar sein!”

“Wir müssen bei neuen Features ein kurzes **Time-To-Market** haben!”

“Wir müssen einfach **skalieren** können.”

“Unsere Webapp muss **performant** sein!”

“Die Webapp muss **responsive** sein und auf allen Geräten laufen!”

“Die Webanwendung muss **einfach zum bedienen** sein, sie muss eine **gute User Experience** haben!”

“Die Webapp muss einfach **wartbar** und **erweiterbar** sein!”

Mittels Qualitätsszenarien fassbar machen....

- Qualitätsszenarien beschreiben exemplarisch die Verwendung des Systems und zwar so, dass ein Qualitätsmerkmal die Hauptrolle spielt.
- Ein Text mit 1 – 3 Sätze



[Software Architecture in Practice, Third Edition, Kap. 7.1](#)

- Man muss sinnvoll darüber reden können.
- Man muss es (theoretisch) überprüfen können.

Mittels Qualitätsszenarien fassbar machen....

→ “Einfach erweiterbar”

«*Ein erfahrener Web-Entwickler will Jira um ein einfaches Plugin erweitern. Der Setup des Plugins dauert maximal einen halben Personentag, um anschliessend mit der Umsetzung der Fachlichkeit starten zu können.*»

Mögliche daraus resultierende Architekturansätze und Technologieentscheide:

- Komponentenbasierte Frontend-Architektur (Atlassian)
- Micro-Frontend Ansatz

Mittels Qualitätsszenarien fassbar machen....

→ Integrierbarkeit

Für dich empfohlen

[Mehr Empfehlungen](#)



149.–

Bose SoundSport (In-Ear, Schwarz)



79.90

RAVPower RP-PB043 (20100mAh, Quick



2599.–

Canon EOS 5D Mark IV Body 3 Jahre Premium-



37.–

Samsung EVO+ microSD UHS-I (128GB, Class 10,

Mittels Qualitätsszenarien fassbar machen....

→ Integrierbarkeit

Für dich empfohlen



279.-

Apple AirPods Pro (In-Ear, Weiss, ANC)



299.-

Sony WH-1000XM3 (Over-Ear, Black, ANC)



535.-

Dji Mavic Mini Fly More Combo (12Mpx, 2.7K)

Mehr Empfehlungen



365.-

Xiaomi M365 (25km/h, 250W)



Mittels Qualitätsszenarien fassbar machen....

→ Integrierbarkeit

«Wird ein neuer Baustein für den Online-Shop entwickelt, wie z.B. die ‘Produktempfehlung’, kann dieser unabhängig von den anderen fachlichen Bausteinen entwickelt und innerhalb weniger Personentagen integriert werden.»

Mögliche daraus resultierende Architekturansätze und Technologieentscheidungen:

- Microservices
- Self Contained System (SCS)

Mittels Qualitätsszenarien fassbar machen....

- ✓ Qualitätsziele resp. nichtfunktionale Anforderungen sind **Architekturtreiber** und die Basis für **Technologieentscheidungen**.
- ✓ Ohne entsprechendes Qualitätsziel (mit Qualitätsszenario) sollte man **keine Technologie- oder Architekturentscheidung** treffen!
- ✓ Architekturentscheidungen müssen entsprechend beschrieben und begründet werden → Kapitel 9 Entwurfsentscheidungen in arc42 / Architectural Decision Records.

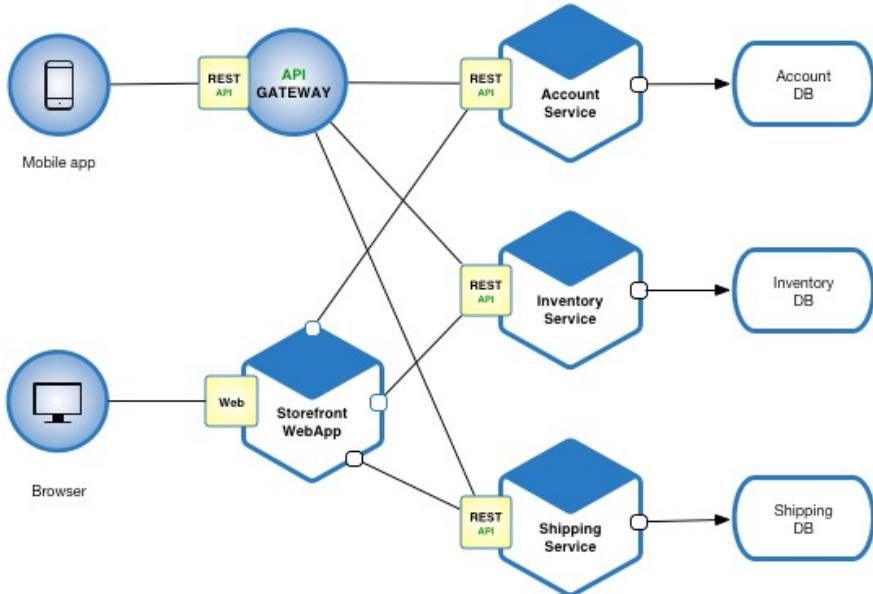
Backend- und Infrastruktur Architekturansätze

- Microservices
- Self Contained Systems (SCS)

Microservices Architecture

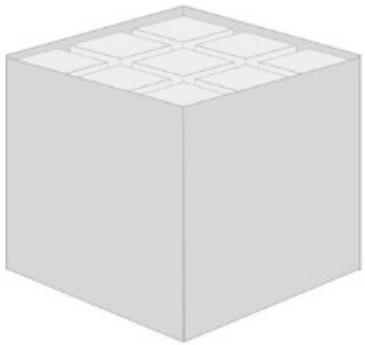
- microservices.io
- Buch: [Building Microservices von Sam Newman](#)

- Small, and focused on doing one thing well
- Autonomous
- **Key Benefits**



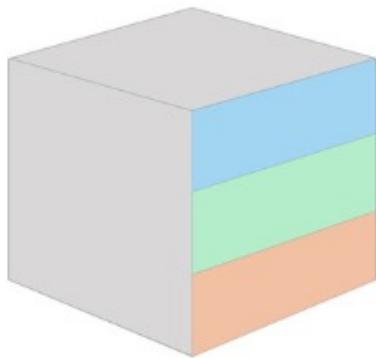
- + Technology Heterogeneity
- + Resilience
- + Scaling
- + Ease of Deployment
- + Organizational Alignment
- + Composability
- + Optimizing for Replacability

Self Contained System



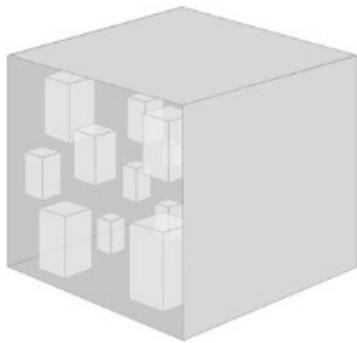
Various Domains

Self Contained System



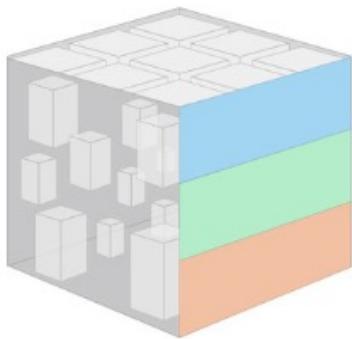
User interface
Business logic
Business logic
Persistence

Self Contained System



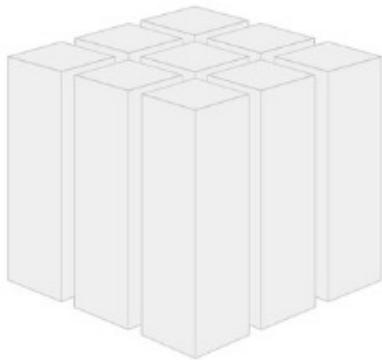
... as well as a lot of
modules, components,
frameworks and libraries.

Self Contained System



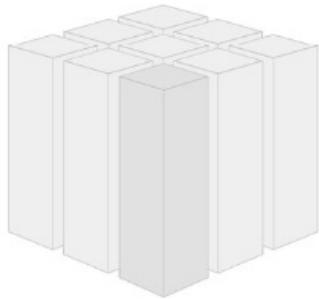
With all these layers
in one place, a
monolith tends to
grow.

Self Contained System



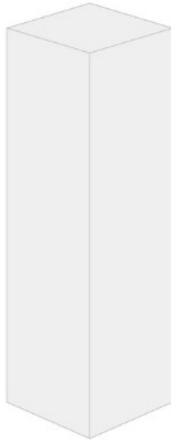
If you cut a monolithic system along its very domains ...

Self Contained System



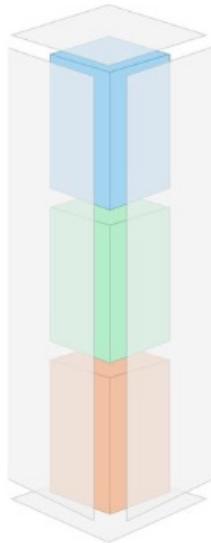
... and wrap every domain in
a **separate, replaceable**
web application ...

Self Contained System



... then that application can
be referred to as a **self-**
contained system (SCS).

Self Contained System



An SCS contains its own
user interface, specific
business logic and
separate **data storage**

Self Contained System



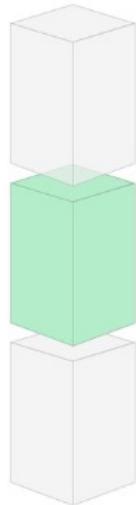
The user interface consists of web technologies that are composed according to ROCA principles.

Self Contained System



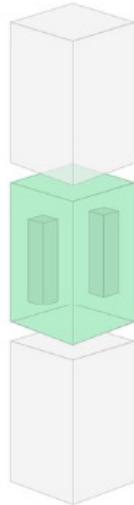
Besides a web interface a self-contained system can provide an optional API.

Self Contained System



The business logic part only solves problems that arise in its core domain. This logic is only shared with other systems over a well defined interface.

Self Contained System



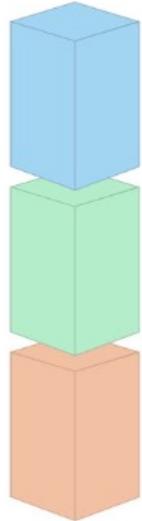
The business logic can consist
of **microservices** to solve
domain specific problems.

Self Contained System



Every SCS brings its **own data storage** and with it redundant data depending on the context and domain.

Self Contained System



Inside of a self-contained system a bunch of **technical decisions** can be made independently from other systems, such as programming language, frameworks, tooling or workflow.

SCS Eigenschaften

1. Each SCS is an autonomous web application.
2. Each SCS is owned by one team.
3. Communication with other SCSs or 3rd party systems is asynchronous wherever possible.
4. An SCS can have an optional service API.
5. Each SCS must include data and logic.
6. An SCS should make its features usable to end-users via its own UI.
7. To avoid tight coupling an SCS should share no business code with other SCSs.
8. To make SCSs more robust and improve decoupling shared infrastructure can be minimized.

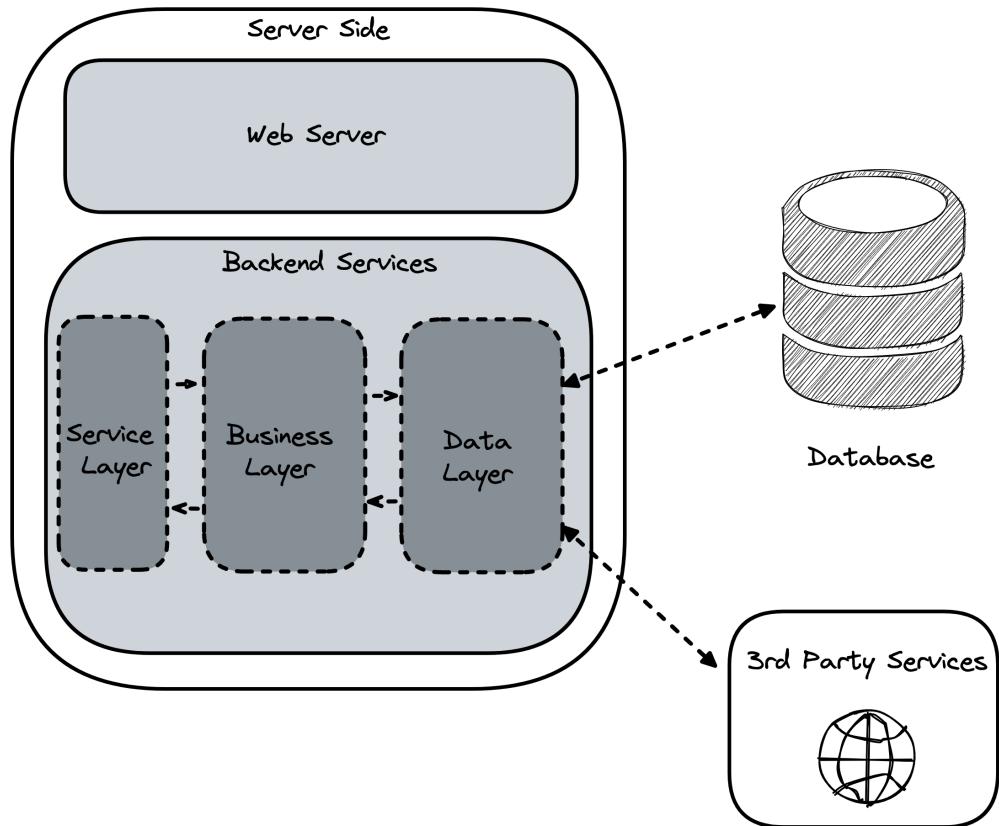
Client Architekturansätze

- Single Page Application (SPA)
- Resource-Oriented Client Architecture (ROCA)
- Javascript, APIs and prerendered Markup – JAMStack

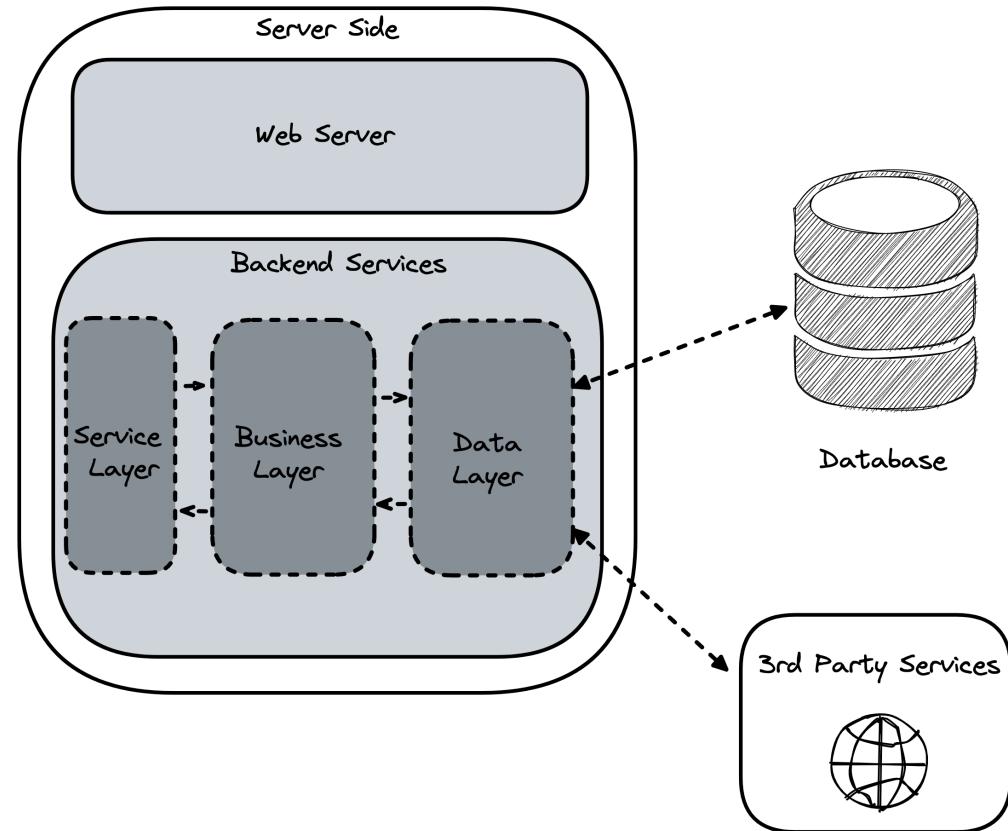
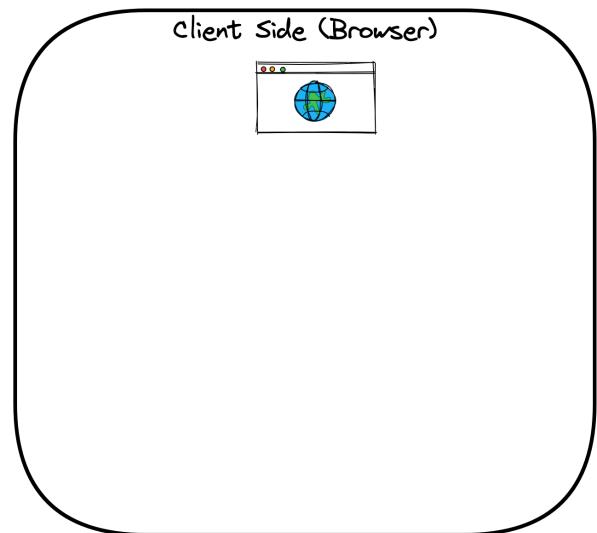
Komplementäre Architektur Ansätze

- Microfrontends
- Progressive Web Apps

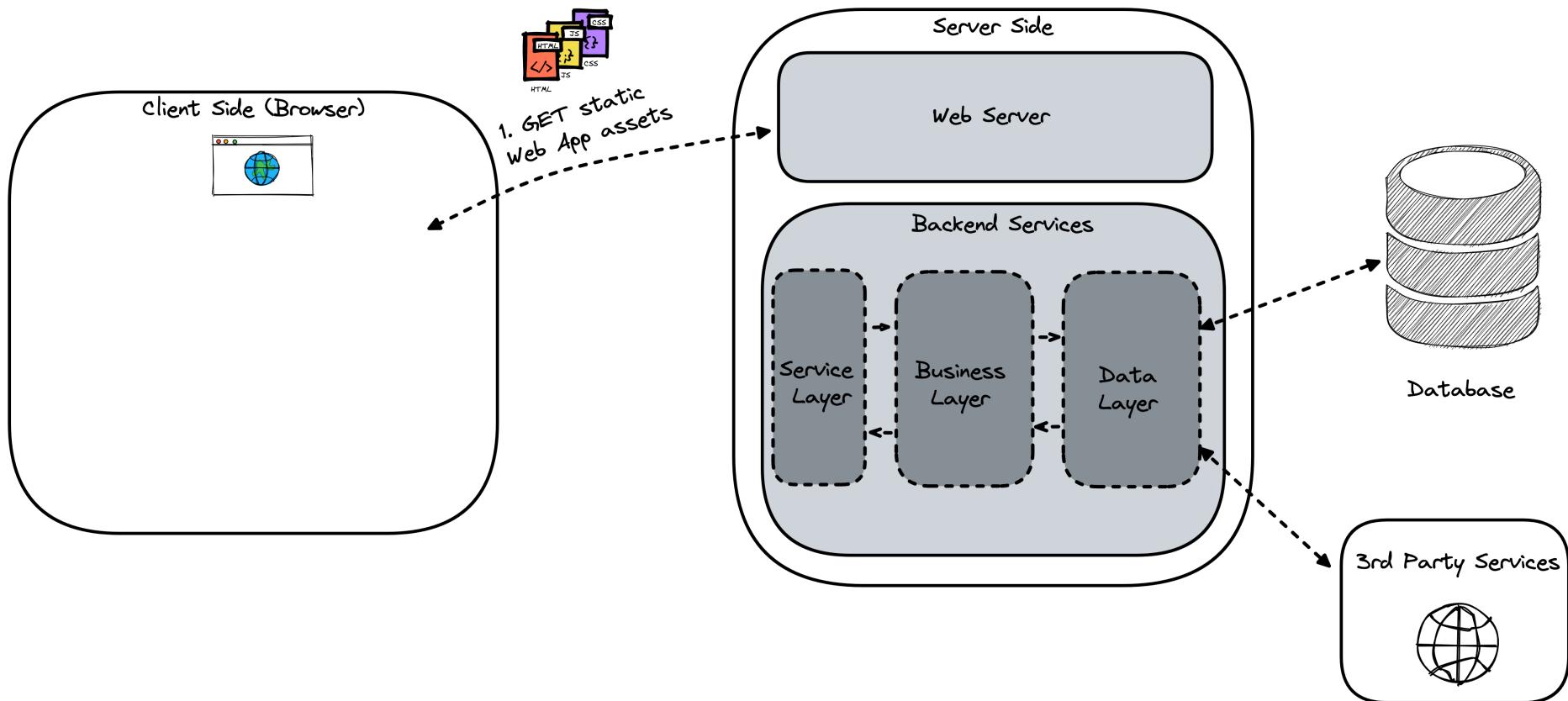
Single Page Application (SPA)



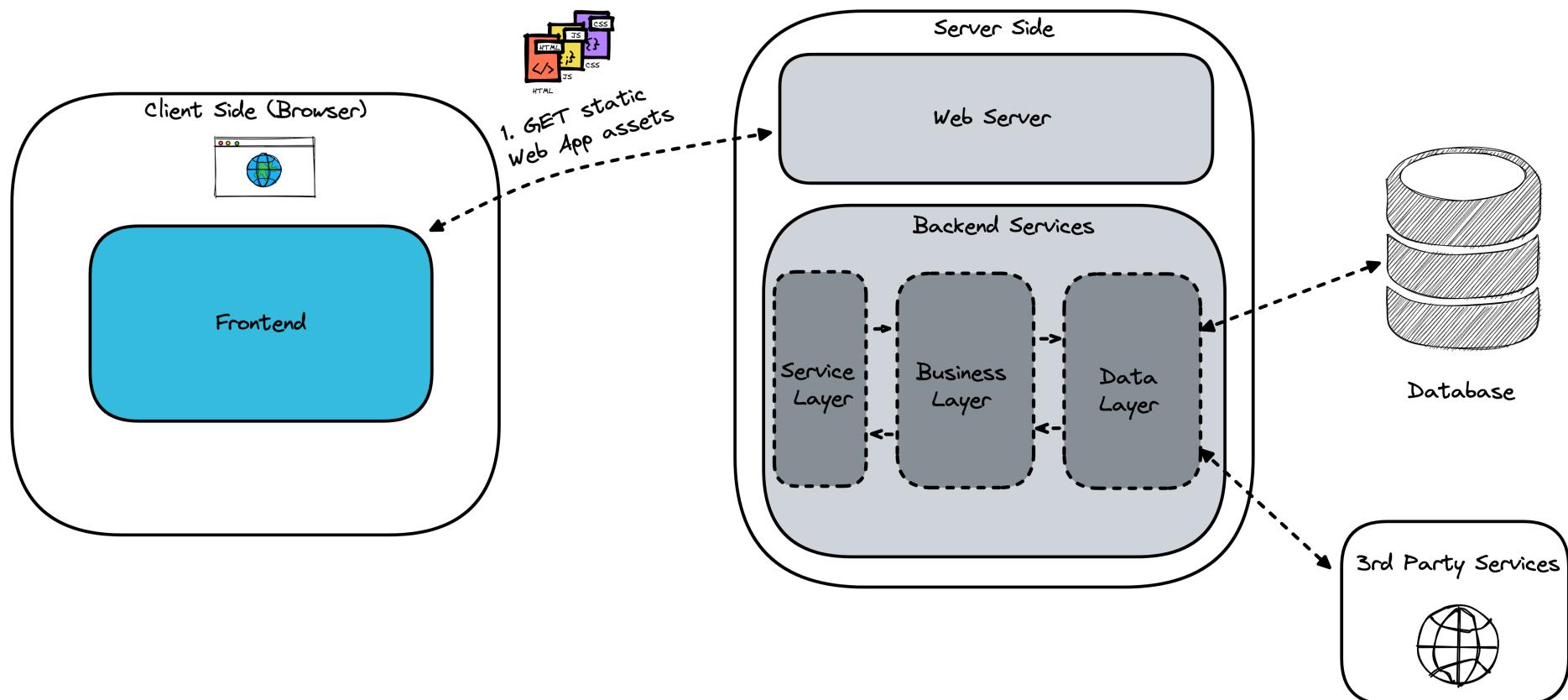
Single Page Application (SPA)



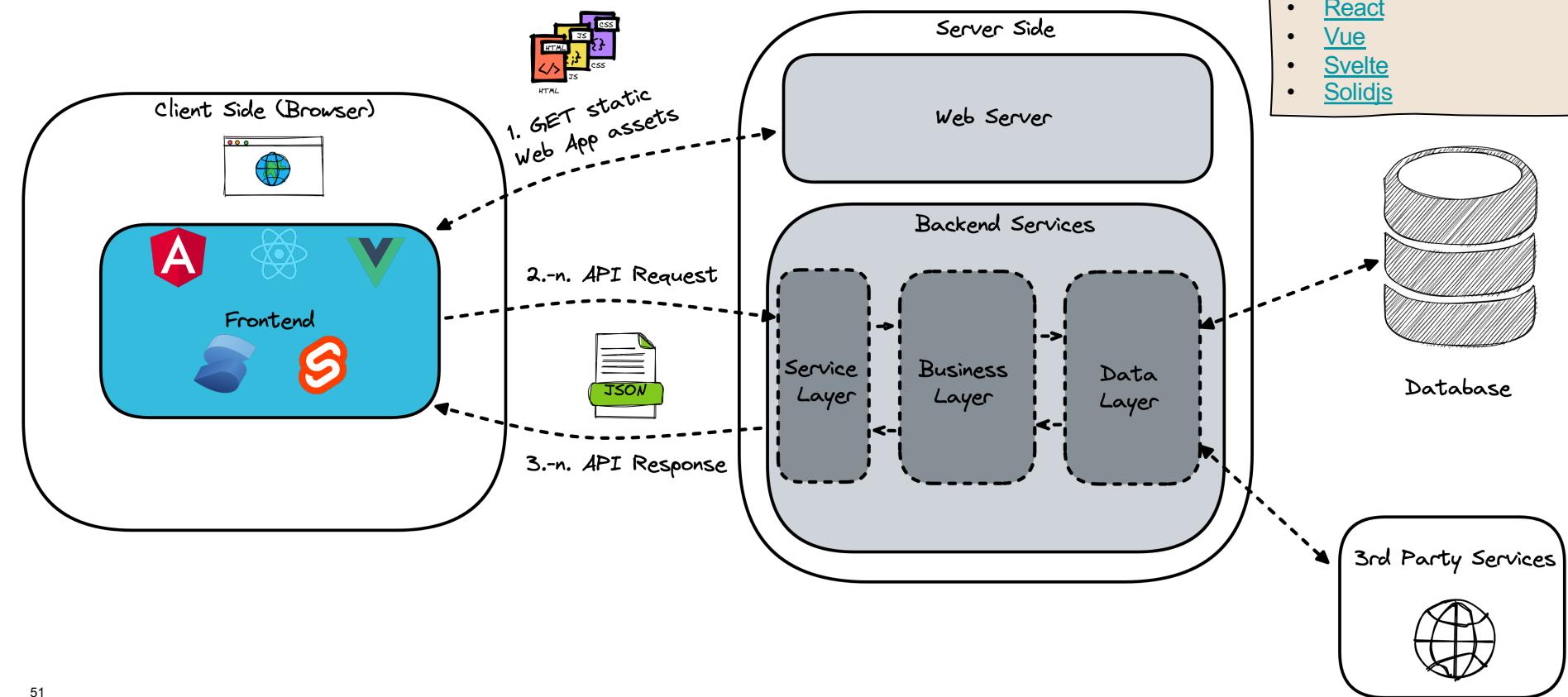
Single Page Application (SPA)



Single Page Application (SPA)



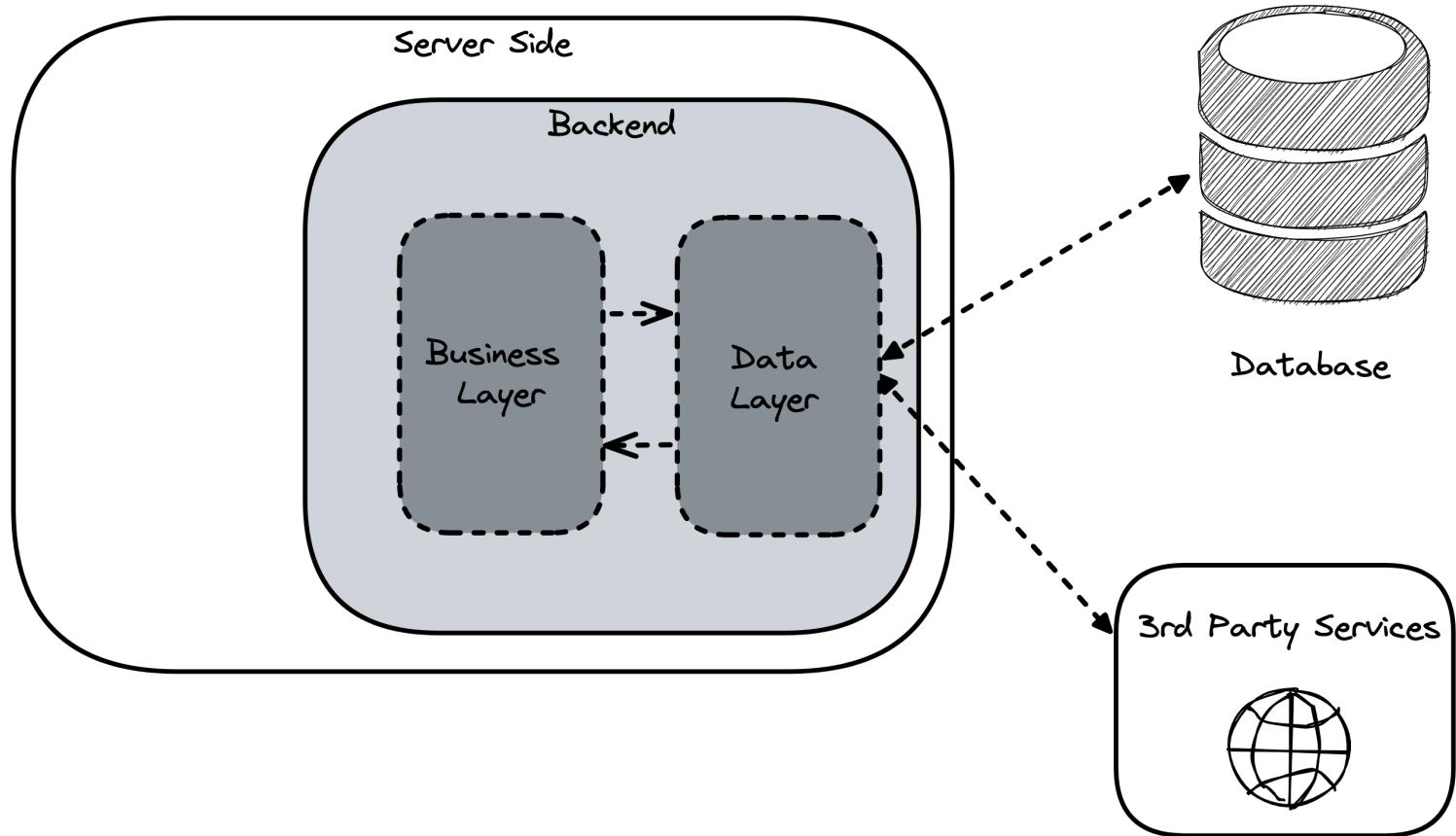
Single Page Application (SPA)



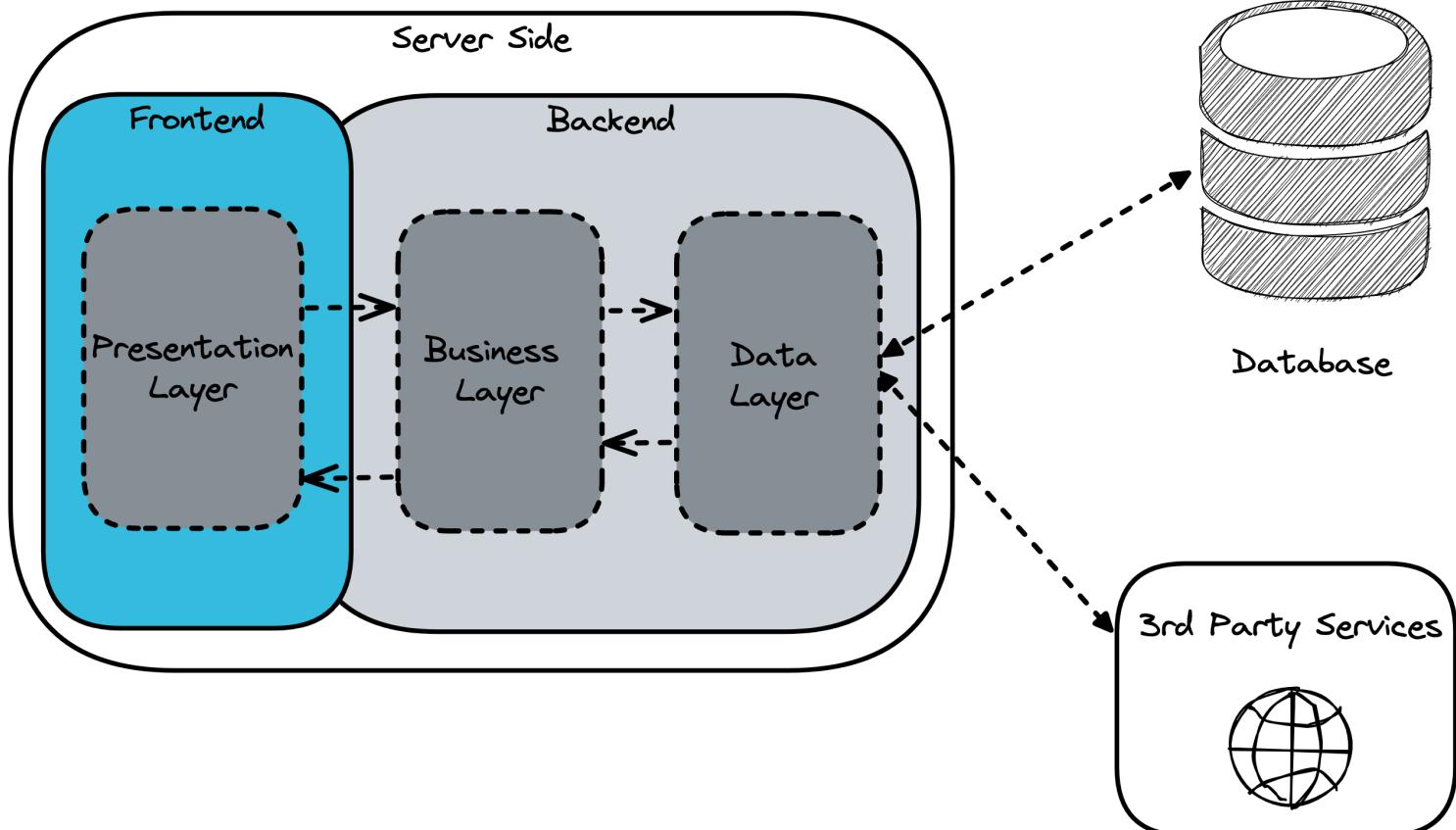
Single Page Application (SPA)

- **Clientseitiges Rendering**, d.h. Single Page Applications werden im Browser durch JavaScript Code generiert.
- Serverseitig gerenderetes HTML (mehrheitlich) ist überflüssig.
- Die Aufgabe des Servers besteht bei SPAs hauptsächlich beim Ausliefern von Daten (= statischer Inhalt & APIs).
 - + User Experience (Nutzung der Client-Rechenpower)
 - + Testbarkeit resp. Entkopplung vom Server
 - + Serverseitige HTTP APIs vorhanden
 - SEO
 - Komplexität & Technologieheterogenität
 - Rechnerzeit auf Client

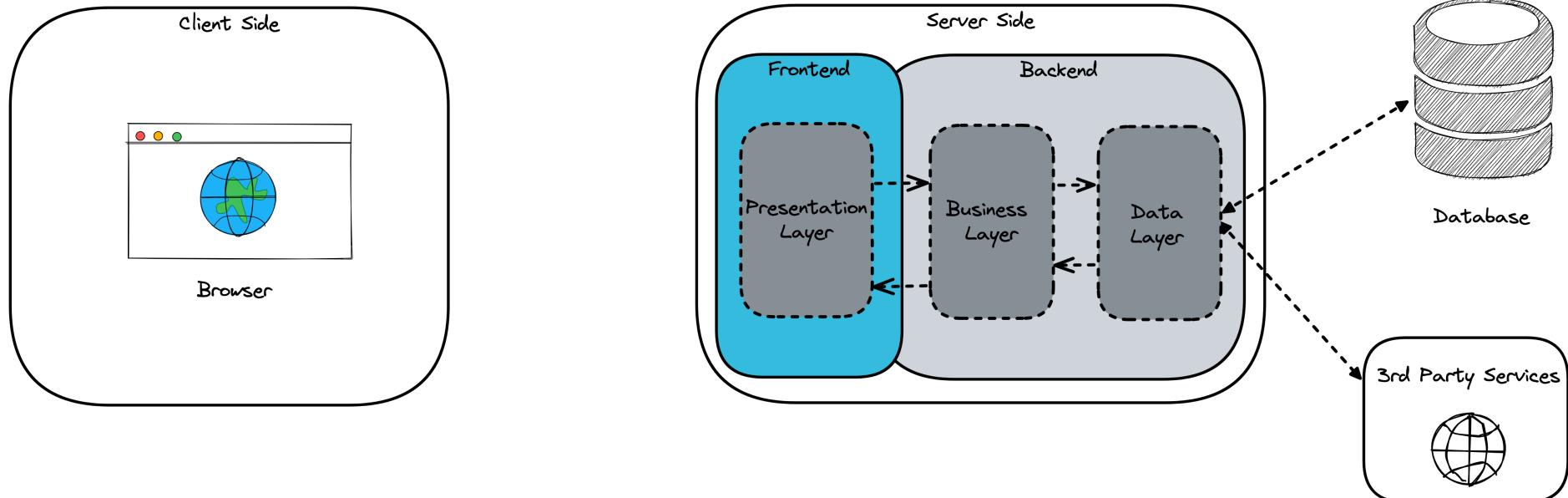
Server Rendering



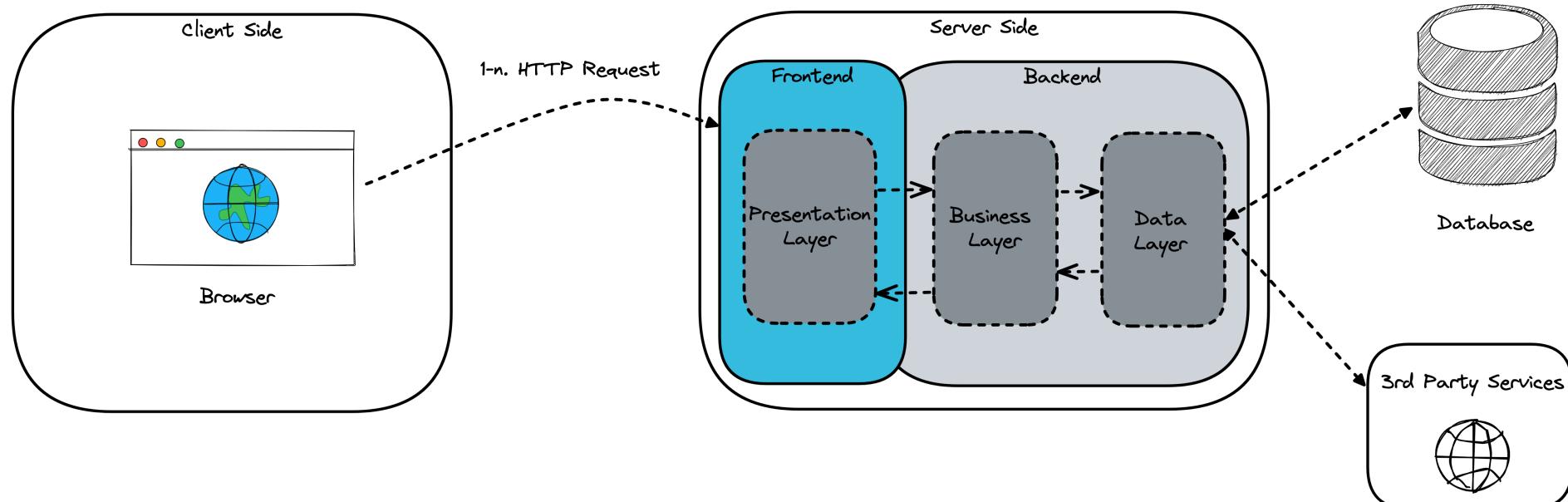
Server Rendering



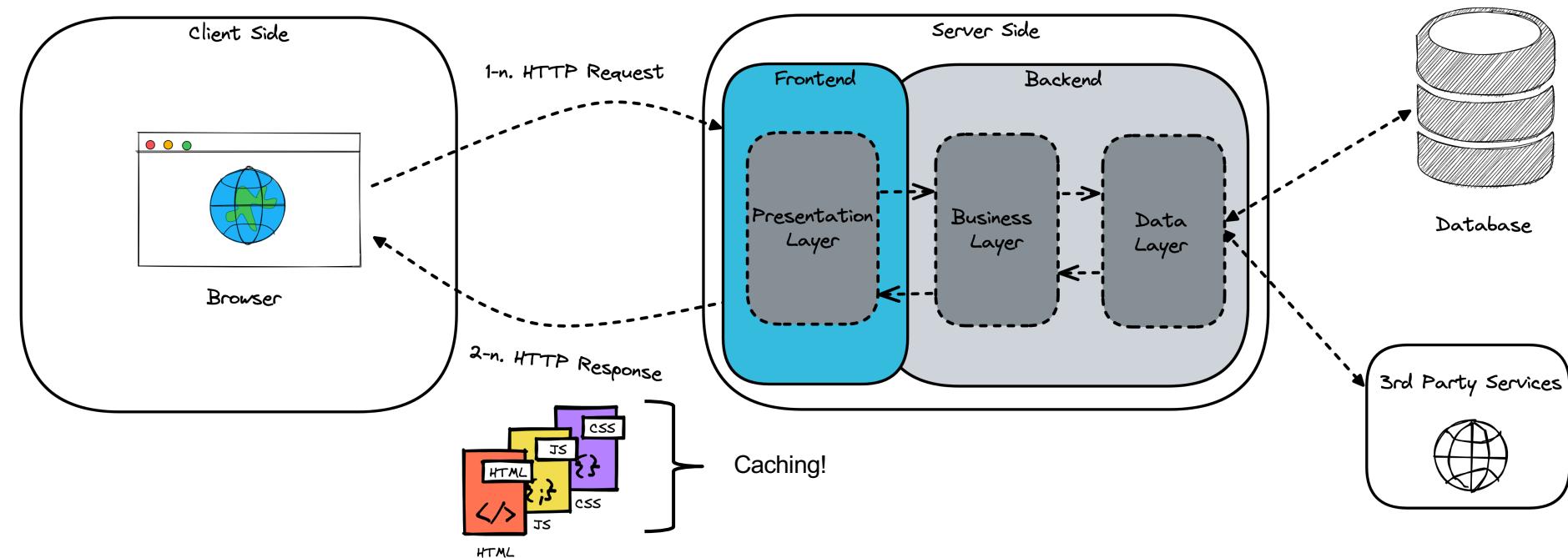
Server Rendering



Server Rendering



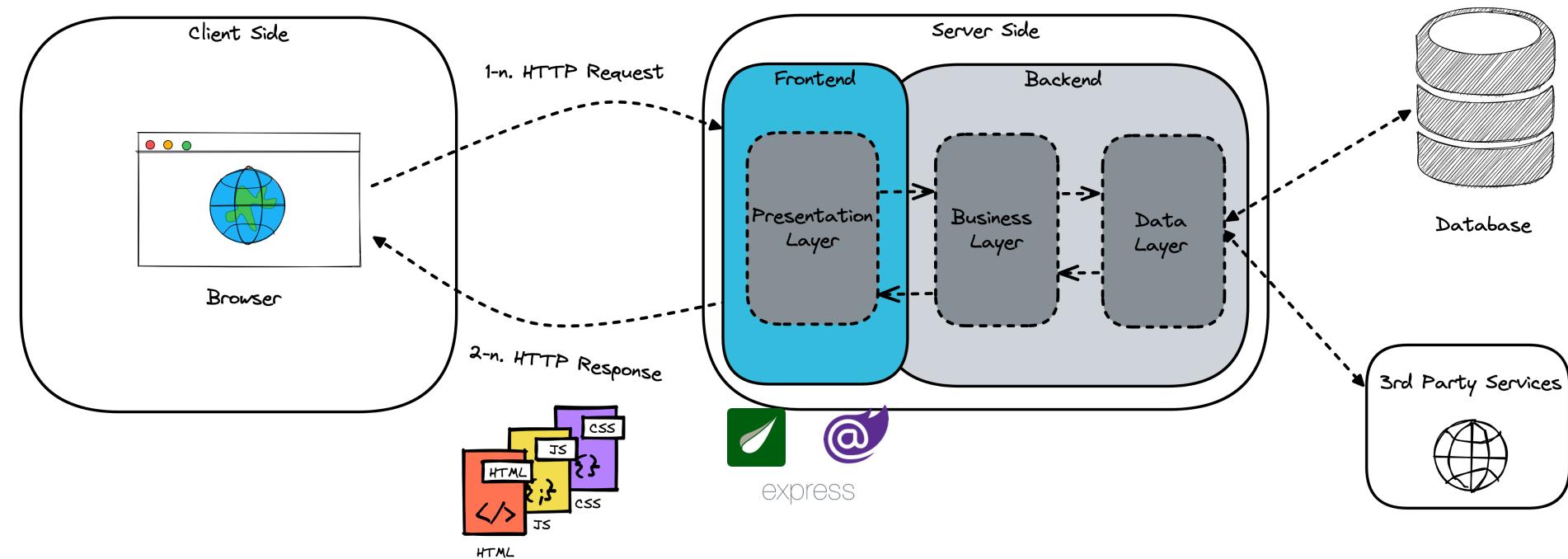
Server Rendering



Server Rendering

Example Server Rendering Frameworks:

- express.js, i.e. with template engines like Handlebars
- Spring MVC with Thymeleaf
- ASP.NET Core Blazor



Resource Oriented Client Architecture (ROCA)

- Prinzipien:
 - Der Server hält sich an REST. Alle Ressourcen haben eine eindeutige URL.
 - Durch URLs identifizierte Ressourcen können auch andere Repräsentationen haben.
 - Alle Logik (inkl. Rendering) ist auf dem Server.**
 - JavaScript dient lediglich zur Optimierung der Benutzeroberfläche.**
 - Es darf keine serverseitige Session geben. (Einige Ausnahme: Auth-Informationen)
 - Browser-Bedienelemente wie Back-, Forward- und Refresh Button sollen funktionieren.
 - Das HTML enthält keine Layout-Informationen.
 - Die Anwendung **soll auch ohne JavaScript** genutzt werden können.
 - Logik darf nicht auf dem Client- und Server redundant implementiert werden.**

Resource Oriented Client Architecture (ROCA)

Vorteile

- ✓ Logik ist nur auf dem Server → Release lediglich vom Server
 - ✓ Wenig Bandbreite & schnell, da kaum mehr als HTML übertragen werden muss
 - ✓ Fehler im JavaScript Code oder bei der Übertragung von JS-Code führt lediglich dazu dass die Anwendung nicht so gut benutzbar ist; sie steht aber immer noch zur Verfügung
- Der ROCA Ansatz ist der Gegenentwurf zu SPAs, bei denen der Browser v.a. genutzt wird, um JavaScript-Code auszuführen.

Beispielhafte [ROCA Implementierung](#)

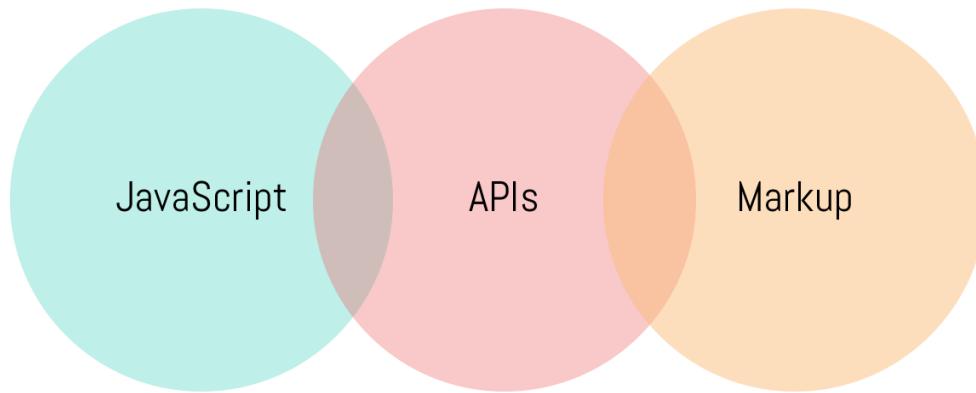
JavaScript, APIs and Markup - Jamstack

"A modern web development architecture based on client-side JavaScript, reusable APIs, and prebuild Markup."

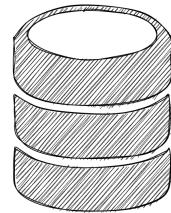
Matt Biilmann (CEO & Co-founder of [netlify](#)).



JavaScript, APIs and prerendered Markup - JAM



JAM Stack

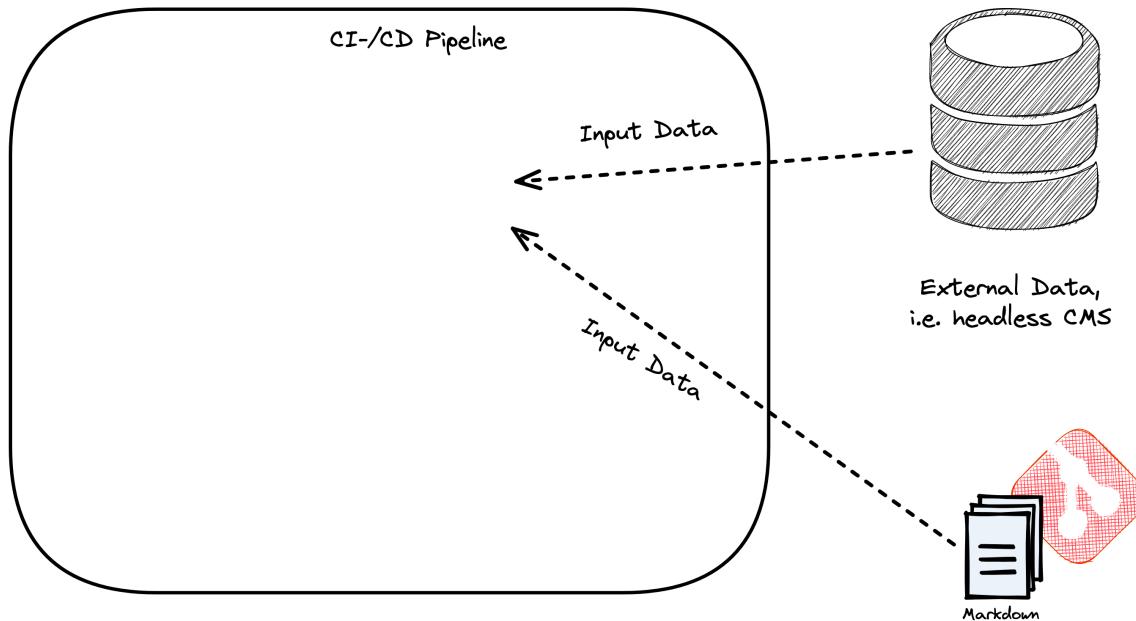


External Data,
i.e. headless CMS

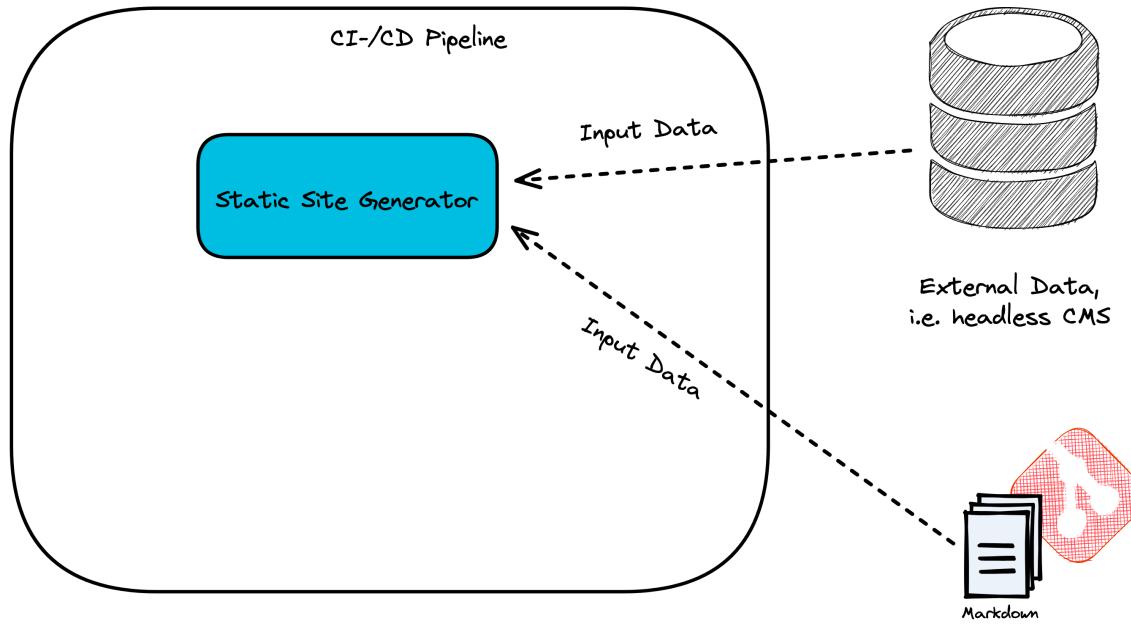


Markdown

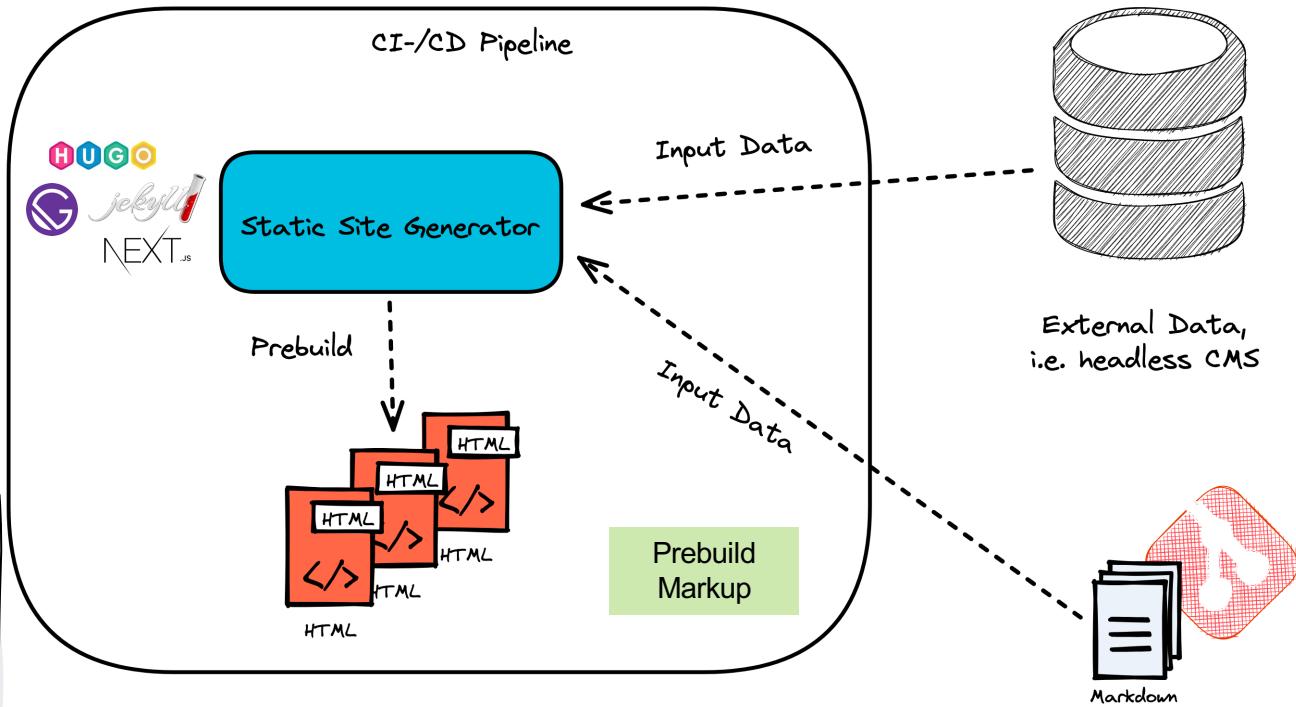
JAM Stack



JAM Stack

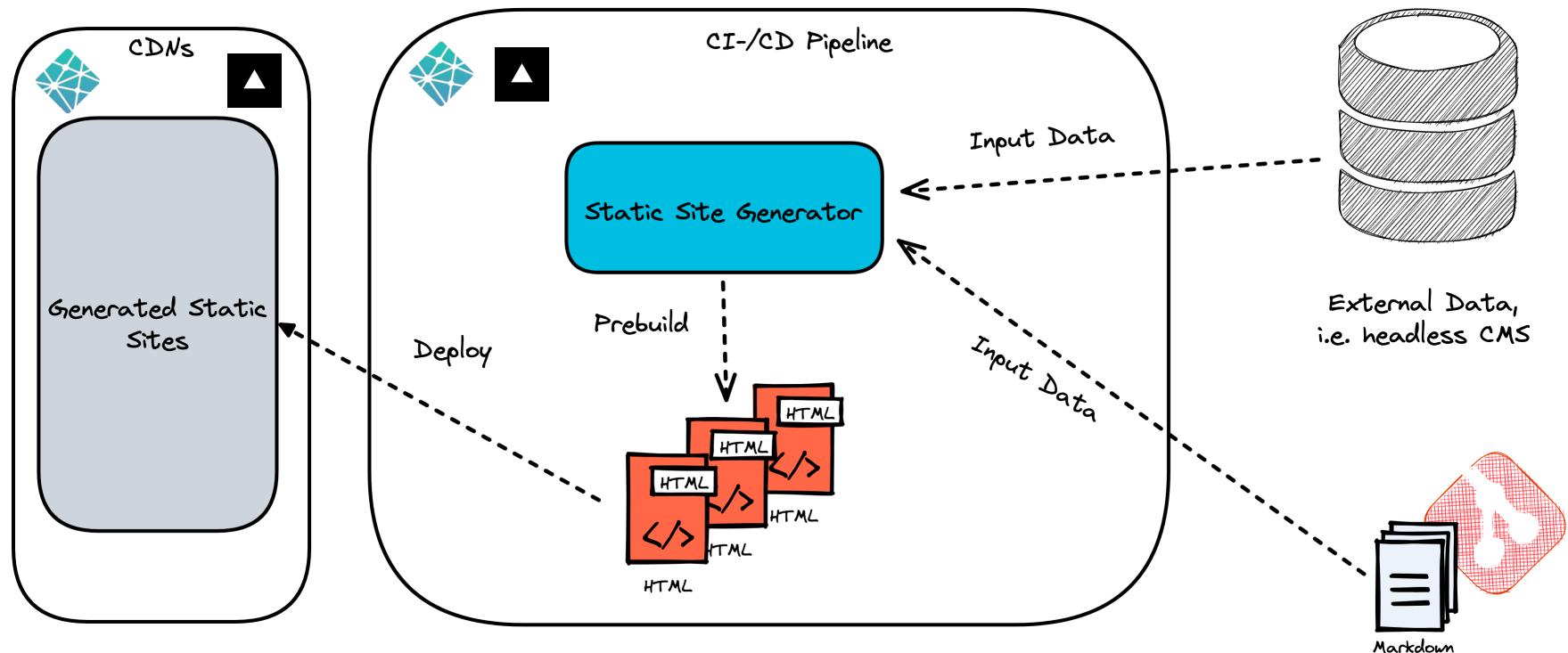


JAM Stack

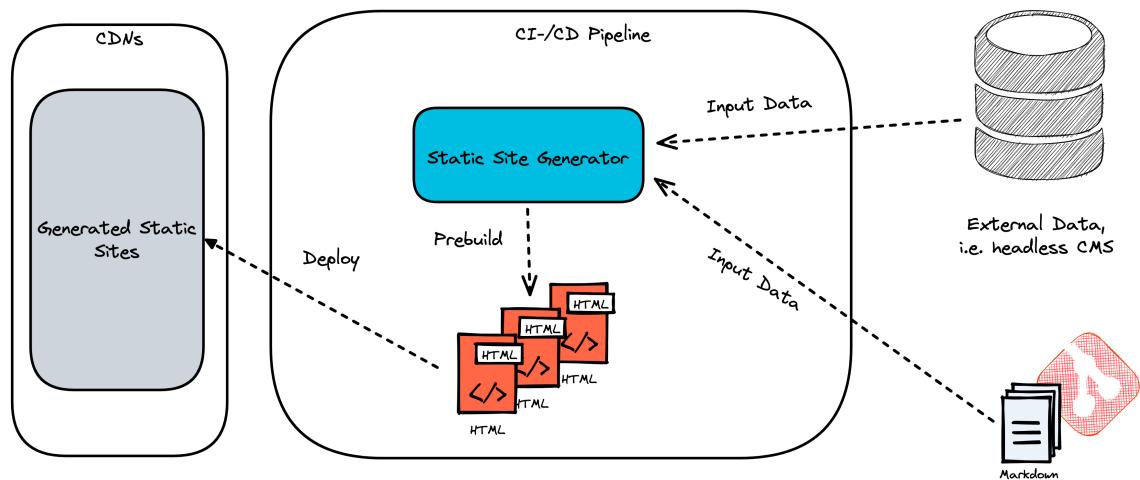
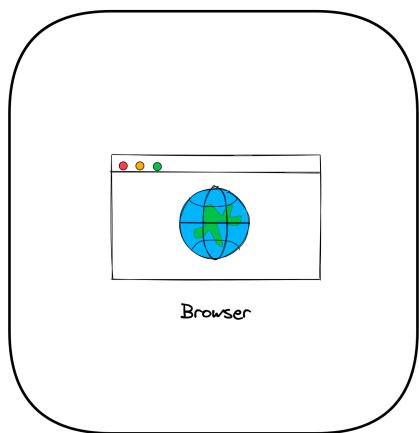


JAM Stack

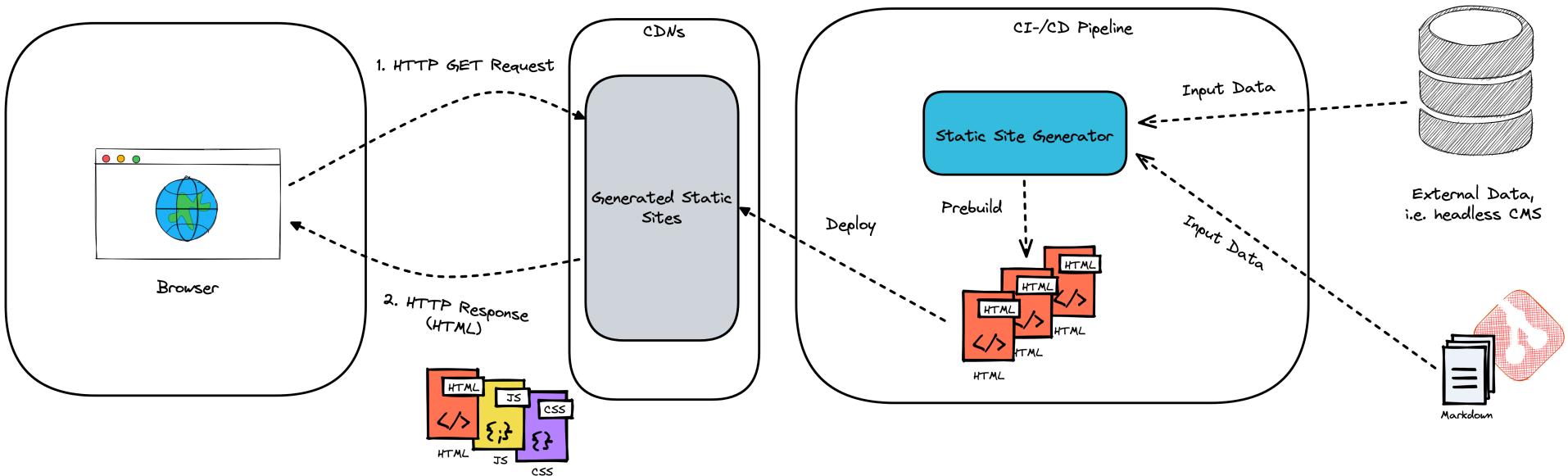
CDNs & Build Platforms for SSG
Scenarios:
• [Netlify](#)
• [Vercel](#)



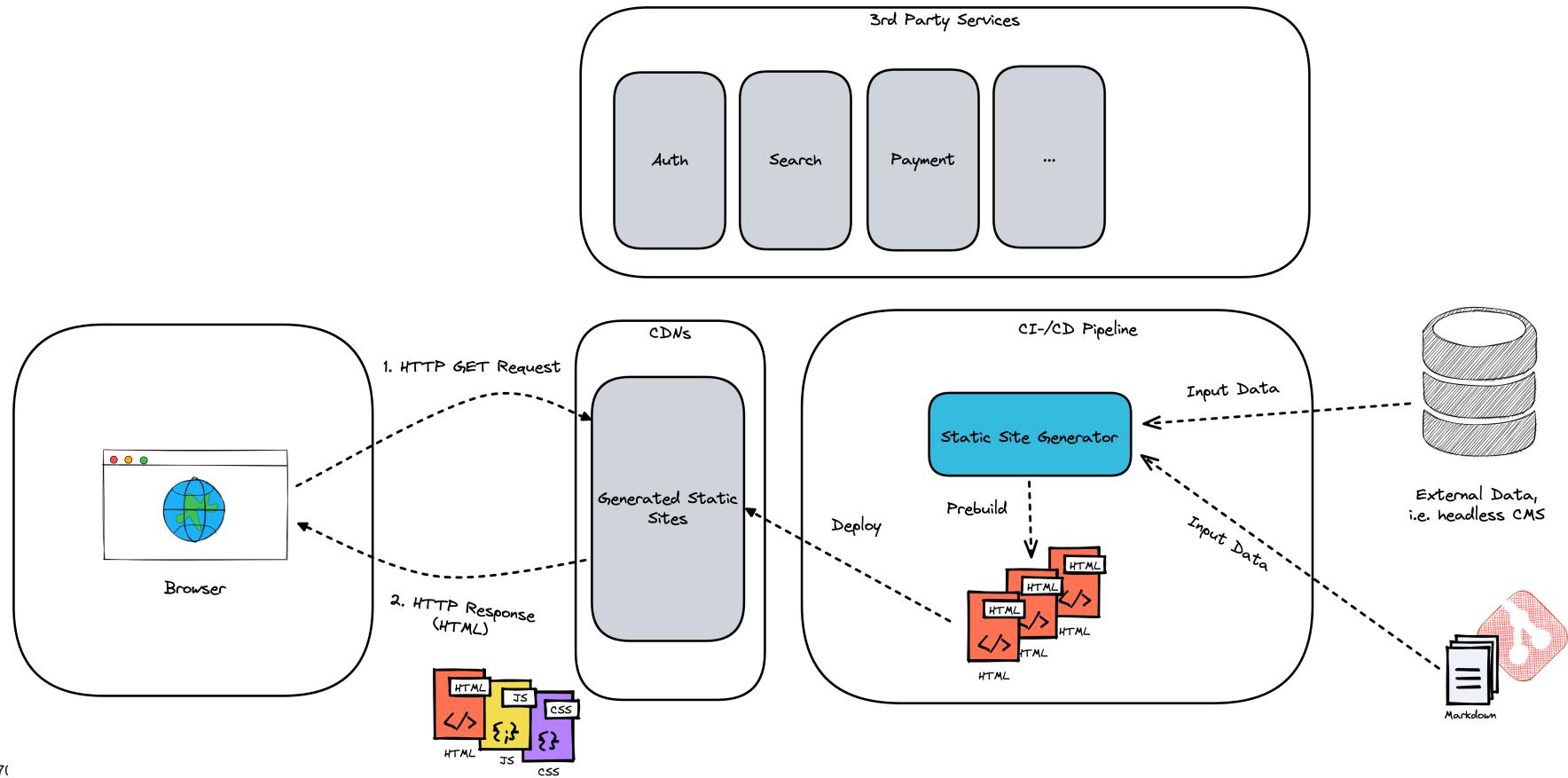
JAM Stack



JAM Stack



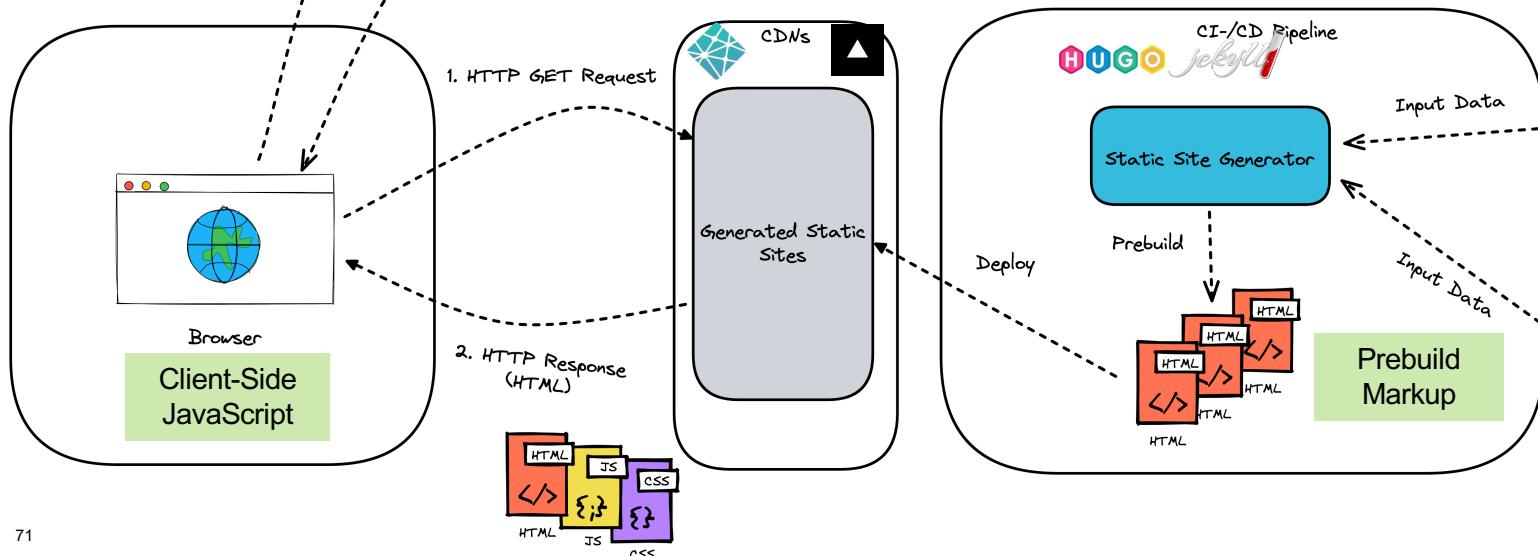
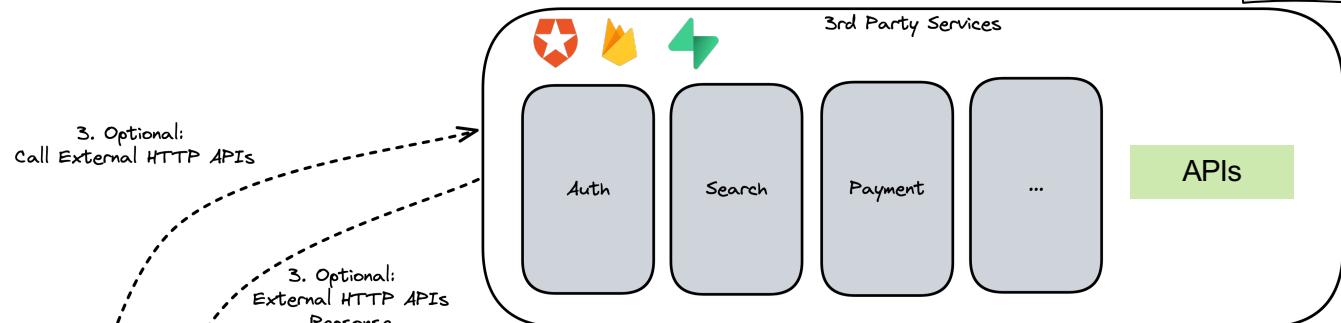
JAM Stack



JAM Stack

Example 3rdParty Services:

- [Auth0](#)
- [Firebase](#)
- [Supabase](#)



JavaScript, APIs and prerendered Markup - JAM

JavaScript: Dynamic functionalities are handled by JavaScript. There is no restriction on which framework or library you must use.

APIs: Server side operations are abstracted into reusable APIs and accessed over HTTPS with JavaScript. These can be third party services or your custom function.

Markup: Websites are served as static HTML files. These can be generated from source files, such as Markdown, using a Static Site Generator.

- <https://jamstack.org/>
- <https://jamstack.wtf/>

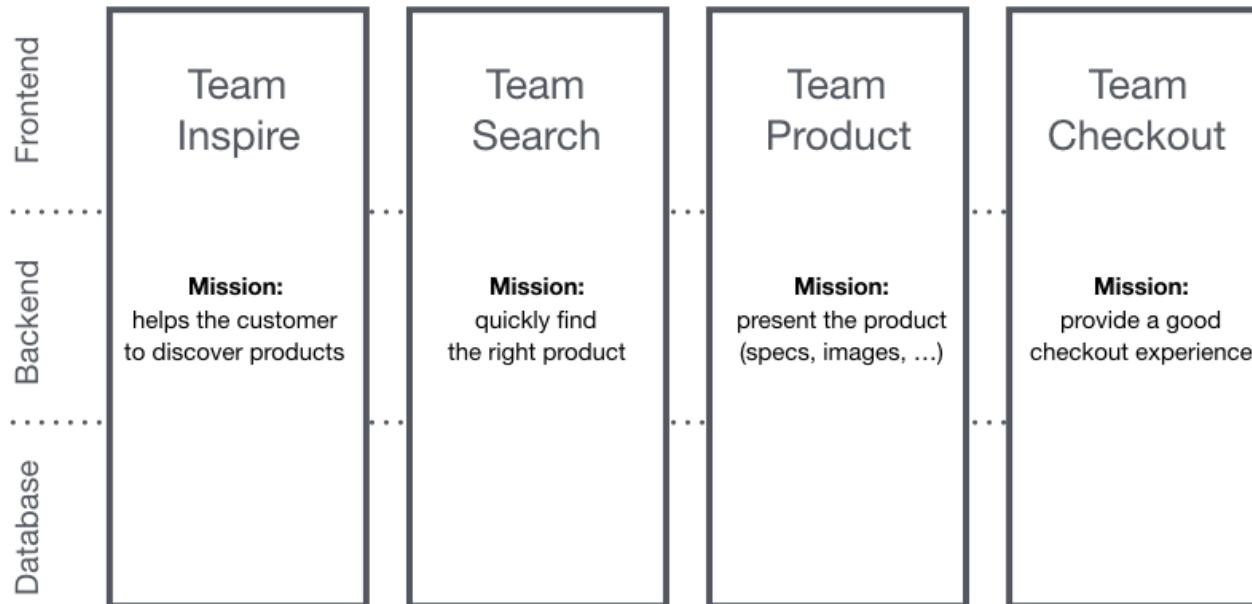
Microfrontends

The Monolith



Microfrontends

End-to-End Teams with Micro Frontends



Microfrontends

The Model Store

basket: 0 item(s)



Tractor Porsche-Diesel Master
419



buy for 66,00 €

Related Products



Team Product ⚡

Team Checkout 💎

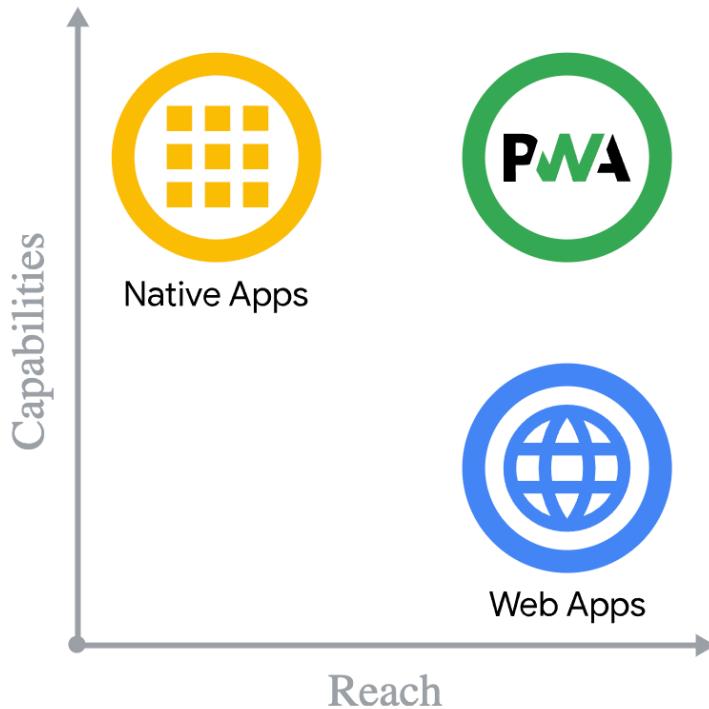
Team Inspire 🌻

<https://micro-frontends.org/>

Progressive Web Apps

Beispiel PWA:

<https://squoosh.app/>



<https://web.dev/what-are-pwas/>

Progressive Web Apps

Progressive Web Apps sind **Web Anwendungen**, die so konzipiert wurden, dass sie:

- Leistungsfähig (z.B. Push-Notifikationen, Geolocation, [WebRTC](#), etc. nutzen)
- Zuverlässig (z.B. offline-verfügbar resp. benutzbar sind)
- Installierbar (z.B. als Standalone Applikation auf der Zielplattform)

...sind

Im Grundsatz sind es Web Anwendungen die sich entlang von der Möglichkeit der Laufzeit anpassen.

Diesen Ansatz vertiefen wir am Freitag morgen!

Zusammenfassung

- Nicht funktionale Anforderungen als Architekturtreiber
- Backend- und Infrastrukturansätze Architekturansätze
 - > Microservices
 - > Self Contained Systems
- Client-Architekturstile (siehe auch [Blog Post](#))
 - > Single Page Applications (Client Rendering)
 - > Resource Oriented Client Architecture (Server Rendering)
 - > JAMStack (Static Rendering)
 - > Microfrontends
 - > Progressive Web Apps

Übung «Architektur Technologie-Radar»

Ihr dürft einen Technologie-Radar umsetzen. Folgende Anforderungen sind bekannt:

- Technologien können in einem geschützten Bereich erfasst, geändert, publiziert und gelöscht werden.
- Publizierte Technologien erscheinen in der Technologie-Radar Ansicht.
- Besucher des Technologie-Radars sehen die publizierten Technologien und können diese kommentieren.
- Die Technologien auf dem Radar müssen schnell geladen werden.
- Die Verwaltung der Technologien muss sicher sein. D.h. nur bestimmte Benutzer (Rolle CTO / Tech-Lead) dürfen die Technologien verwalten.
- Der Technologie-Radar soll auf allen Bildschirmgrößen lesbar sein.

Aufgaben (20') [Link auf das Mural-Board](#)

- Identifiziert mögliche nicht funktionale Anforderungen
- Optional: Diskutiert mögliche bekannte Architekturansätze und Technologie-Einsätze des Technologie-Radars
- 2-3er Gruppen, 1 – 2 Vorstellungen der Gruppe

JavaScript Sprachkonzepte I

Eigenschaften, History und Versionen, Setup Dev Umgebung

re: Oh, the dilemma! Should you learn React or Angular or Vue or "Another framework".js [VIEW POST](#)

[TOP OF THREAD](#)[FULL DISCUSSION](#)

re: I agree so much. As a Dev still learning alot of JS, I initially fell prey to the shininess of React, Vue and all of the other fra...



Kenny Whyte [Twitter](#) [GitHub](#) [Author](#)

Dec 8 '19 [...](#)

I agree. I made the **HUGE** mistake to skip learning vanilla JS and jumped into Angular, then the 'magic' got me confused when I was getting errors that I couldn't figure out. Things got a lot better when I stopped...then started over from scratch with just JS.



2

[REPLY](#)



Coner [GitHub](#)

Dec 8 '19 [...](#)

That's the current process I'm going through. I've found I can build stuff in ReactJS because I understand how to use **it's framework specific language**. But, now I'm looking at **expressJS** I'm lost again. So back to basics it is.



2

[REPLY](#)

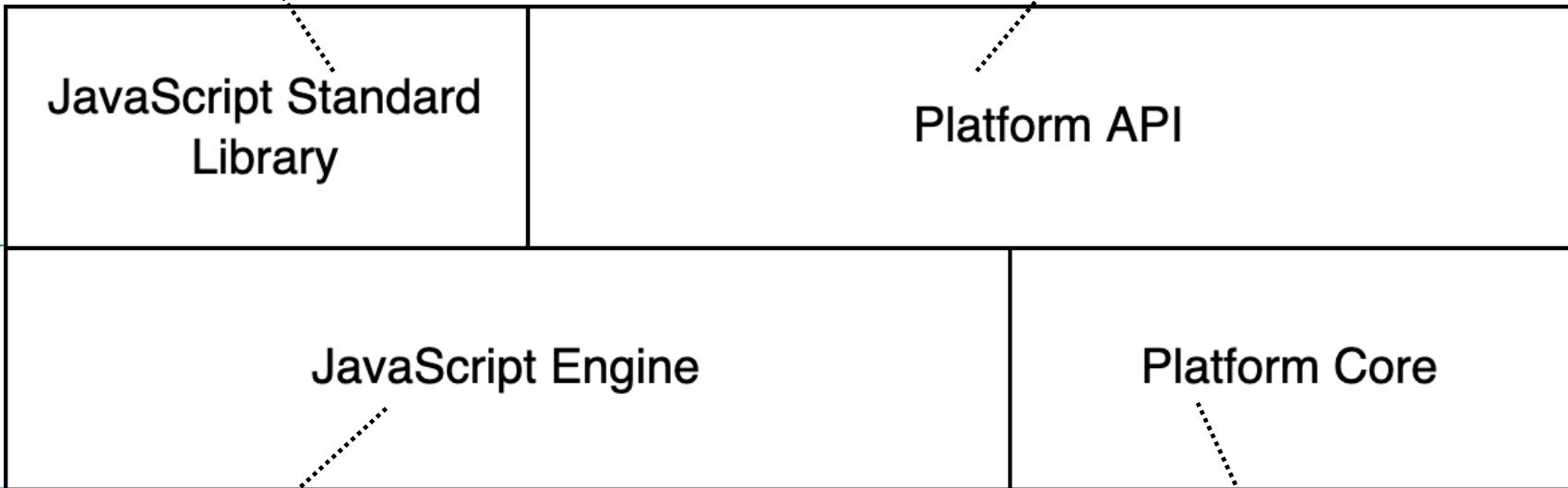
JavaScript

- **Populärste** Programmiersprache der Welt
- Erste und einzige Programmiersprache, die **nativ** von Webbrowsern unterstützt wird.
- Wird auch **ausserhalb** des Browsers verwendet.
 - Backend / Cloud-Applikationen ([Node.js](#), [Deno](#), [Bun](#))
 - Mobile Applikationen ([Ionic](#))
 - Desktop Applikationen ([Electron](#))
 - Datenbanken ([CouchDB](#))
 - Serverless Computing (z.B. [Azure Functions](#))
 - Infrastruktur (“Infrastructure as Code”) ([Pulumi](#))

JavaScript Platform Integration

Sprachspezifika von JavaScript,
wie z.B. Datentypen, Functions,
etc.

Foundation Layer



Interpretiert das JavaScript,
z.B. V8 von Chrome

Plattformspezifische Funktionalität, z.B.
LocalStorage im Browser; Lesen und
Schreiben von Dateien in Node.js

JavaScript Eigenschaften

- **High Level**

Dank JavaScript können die Details der Laufzeitplattform ignoriert werden. Das Memory wird automatisch über einen Garbage Collector gemanaget, so dass man sich auf den Code fokussieren kann.
- **Dynamisch**

JavaScript führt viele Dinge zur Laufzeit aus, welche eine statische Programmiersprache zur Kompilierzeit macht. Dies hat Vor- und Nachteile. Es gibt interessante Features wie z.B. dynamic typing.

JavaScript Eigenschaften

- **Schwach typisiert**

Eine Variable ist an den initialen Typ nicht gebunden. Dies ermöglicht bei der Programmierung mehr Flexibilität, jedoch mindert dies die Typensicherheit resp. verlieren den Typencheck zur Laufzeit.
- **Interpretiert**

Das bedeutet, es braucht keinen Kompilierungsschritt bevor das Programm laufen kann, wie dies z.B. bei C# oder Java der Fall ist. Browsers kompilieren JavaScript bevor der Programmcode ausgeführt wird - jedoch ist kein zusätzlicher Schritt involviert.

JavaScript Eigenschaften

- **Mehrere Paradigmen**

JavaScript forciert kein bestimmtes Programmierparadigma, wie z.B. die Objektorientierung wie von Java. JavaScript kann z.B. in einem objektorientierten oder funktionalen Stil verwendet werden.

Wichtig: Man sollte JavaScript nicht mit der Programmiersprache Java verwechseln. Die beiden Programmiersprachen haben eine sehr unterschiedliche Syntax, Semantik und Verwendung.

JavaScript Versionen und History

ECMAScript ist die **Spezifikation** auf welcher JavaScript basiert. Neben JavaScript gibt es auch noch andere Sprachen, welche die Spezifikation implementiert haben:

- ActionScript (Flash Script Sprache)
- JScript (Microsoft Scripting Dialekt)

→ JavaScript ist die populärste Implementation der ECMAScript Spezifikation.

JavaScript Versionen und History

Name	Offizieller Name	Publiziert
ES13	ES2022	Juni 2022
ES12	ES2021	Juni 2021
ES11	ES2020	Juni 2020
ES10	ES2019	Juni 2019
ES9	ES2018	Juni 2018
ES8	ES2017	Juni 2017
ES7	ES2016	Juni 2016
ES6	ES2015	Juni 2015
ES5.1	ES5.1	Juni 2011
ES5	ES5	Dezember 2009
ES4	ES4	Entwurf, abgekündigt
ES3	ES3	Oktober 1999
ES2	ES2	Juni 1998
ES1	ES1	Juni 1997

Evolving Specification

JavaScript Versionen und History

- Ecma International ist ein schweizer Verein der für die Standardisierung von Informations- und Kommunikationssysteme verantwortlich ist.
 - TC39 ist ein technisches Komitee von Ecma International und ist Verantwortlich für die ECMAScript Spezifikation.
- ECMAScript 2022 Language Spezifikation ([Link](#))
→ Proposals aus dem technischen Komittee (Staging-Prozess):
<https://github.com/tc39/proposals>



Übungsfragen im Plenum (5')

1. Auf welchen Plattformen wird JavaScript verwendet resp. kann verwendet werden?
2. Nenne mindestens drei Eigenschaften von JavaScript und erkläre diese.
3. Finde heraus, welche Firmen die ECMAScript Spezifikation massgeblich mitprägen (Chairgroup).

Let's dive into code...

Setup Editor (10')

👉 <https://github.com/web-programming-lab/javascript-sprachkonzepte>

- **10' im Breakout Room**
 - > VS Code installiert, Node.js installiert, Repo geklont (siehe Anleitung)
 - > Auf der Konsole wird 'Hello Web Programming Lab 👍 ' angezeigt (index.js).
 - > Debugging möglich (Breakpoint setzen)
- **5' Im Plenum**
 - > Kurz durch die wichtigsten Elemente des Repos gehen

JavaScript Sprachkonzepte II

Variablen, primitive Datentypen, Operatoren

Variablen und Datentypen

- **const** vs. **let** vs. **var**

```
const studentId = 50;  
const studentId = 49; // Fehler  
const student; // Fehler
```

snippet-const.js

- Variablen welche mit **const** deklariert wurden, können nicht mehr verändert werden. Sie müssen direkt initialisiert werden.

Variablen und Datentypen

- const vs. let vs. var
→ “Block Scoped”

```
function letTest() {  
    let x = 31;  
  
    if(true) {  
        let x = 71; // andere Variable  
        let y = 80;  
        console.log(x); // 71  
    }  
  
    console.log(x); // 31  
    console.log(y); // Error  
}  
  
letTest();
```

Variablen und Datentypen

- const vs. let vs. **var**
→ “Function Scoped”

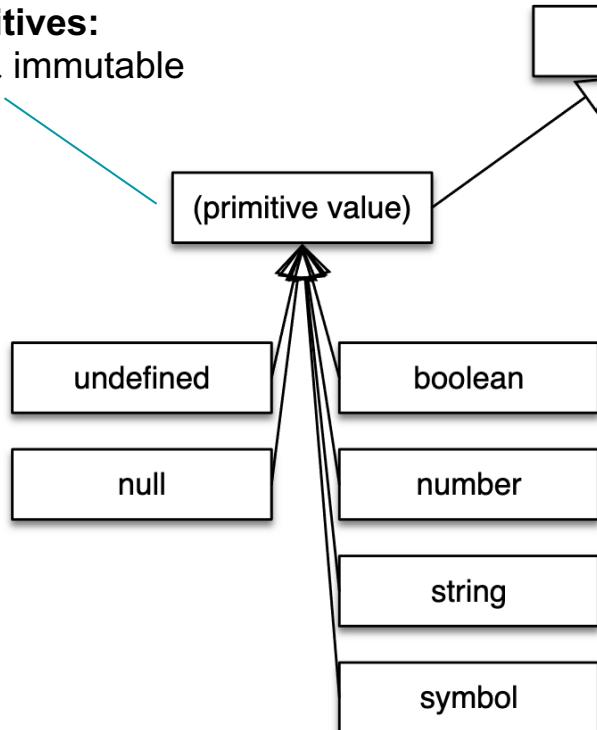
```
function varTest() {  
    var x = 31;  
  
    if(true) {  
        var x = 71; // gleiche Variable x, wird nach oben gueltig!  
        var y = 80;  
        console.log(x); // 71  
    }  
  
    console.log(x); // 71  
    console.log(y); // 80  
}  
  
varTest();
```

Variablen und Datentypen

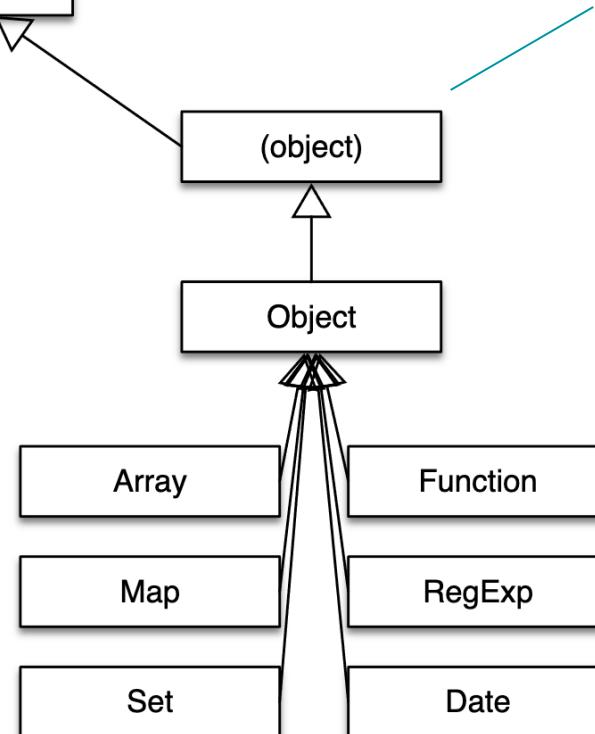
- **const** vs. **let** vs. **var**
 - > Immer wenn möglich **const** verwenden
 - > Wenn **const** nicht passt, **let** verwenden
(**const** ist auch block-scoped)
 - > **var** vermeiden

Datentypen

Primitives:
By value & immutable

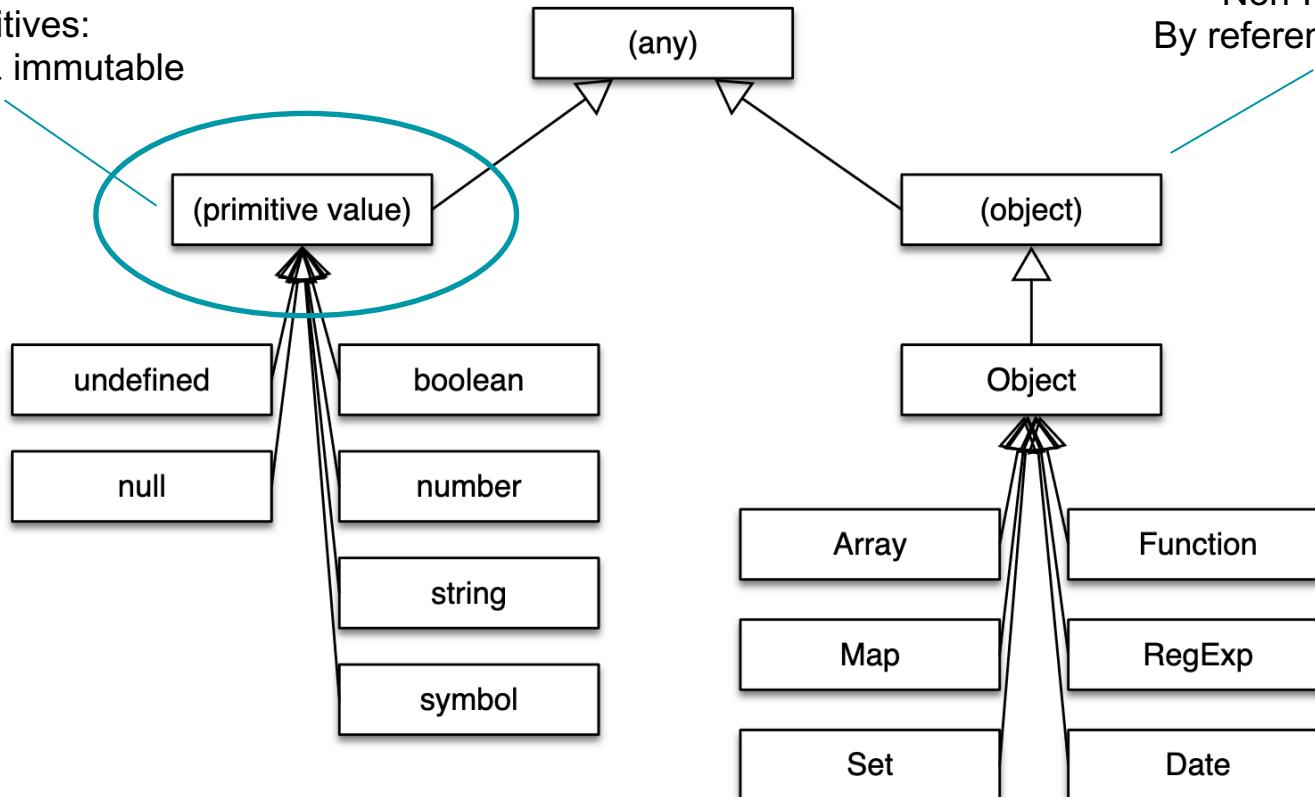


Non-Primitives:
By reference & mutable



Datentypen

Primitives:
By value & immutable



Non-Primitives:
By reference & mutable

Primitive Datentypen - boolean

```
const myBoolean = true;  
const myBoolean2 = false;  
const isGreater = 5 > 2;  
console.log(isGreater); // true
```

snippet-boolean.js

Primitive Datentypen - number

```
let n = 123; // number (integer)  
n = 12.345; // number (float)  
n = n / 0; // number (Infinity)
```

snippet-number.js

- Mittels number Datentyp werden Integer- und Floating Point Werte abgebildet.
Alle number Werte werden als double, d.h. als 64-bit Floating Point implementiert (IEEE 754).
- Neben regulären Werten können "spezielle" numerische Werte vergeben werden, wie z.B. `Infinity`, `-Infinity` und `NaN`.
 - > `Infinity` repräsentiert das "mathematische" ∞ (grosser als Number)
 - > `NaN` repräsentiert einen Berechnungsfehler.

Primitive Datentypen - string

```
const myString = 'Hallo';
const myString2 = 'Web Programming Lab';
const myStringWithBackTicks = `embed strings: ${myString}, ${myString2}`;
console.log(myStringWithBackTicks);
```

snippet-string.js

Primitive Datentypen – «null» / «undefined»

```
const myNullValue = null;
const myUndefinedValue = undefined;
console.log(myNullValue); // null
console.log(myUndefinedValue); // undefined

// Nullish coalescing operator (ES2020)
console.log(myNullValue ?? 'example-value'); // example-value
console.log(myUndefinedValue ?? 'example-value'); // example-value
```

snippet-null-undefined.js

- `null` ist ein spezieller Wert, der “**nichts**” oder “**leer**” markiert
- `undefined` bedeutet “Wert wurde noch nicht zugewiesen”
(z.B. Variable) oder “inexistent” (z.B. Property auf einem Objekt)
- **Best Practice:** “`null`” statt “`undefined`” um ein “leer” oder “unbekannt” zu markieren. ”`undefined`“ in Checks zu verwenden, ob bereits ein Wert hinzugefügt wurde oder nicht.

Weitere primitive Datentypen

- **BigInt** ist ein neuer primitiver Datentyp (ES2020) für Integer-Werte, die zu gross sein können für number. BigInts haben keine fixe Speichergrösse in bits; ihre Grösse werden entlang dem abzubildenden Integer Wert adaptiert.

```
const myBigInt = 123n + 12n;  
console.log(myBigInt); // 135n  
console.log(typeof(myBigInt)); //bigint
```

snippet-bigint.js

- **symbol** werden dazu verwendet, um unique Property Keys oder Werte für Konstanten abzubilden.

Datentyp ermitteln mittels `typeof` vs. `instanceof`

- Eine Variable in JavaScript kann verschiedene Arten von Werte resp. Datentypen annehmen.
- Der Typ kann sich während der Laufzeit verändern (= schwache Typisierung / Dynamic Variable Typing).
- Der Datentyp kann mittels `typeof` oder `instanceof` zur Laufzeit ermittelt werden.

```
let myVar = 'hello';
console.log(typeof myVar); // string
myVar = 1234;
console.log(typeof myVar); // number
```

snippet-typeof.js

[typeof Operator](#) für primitive Werte

```
const arr = [];
console.log(arr instanceof Array); // true
```

snippet-instanceof.js.

[instanceof Operator](#) für Objekte

typeof() Operator

```
typeof(1); // ??  
typeof(true); // ??  
typeof('hallo'); // ??  
typeof({}); // ??  
typeof(null); // ??  
typeof(undefined); // ??  
typeof(NaN); // ??
```

snippet-typeof-II.js

Mittels typeof Operator können Typen von bestimmten Operanden beschrieben werden.

Type conversions

```
// boolean to string
let value = true;
value = String(value);
console.log(typeof(value));

// string to number
console.log('8' / '4'); // Output?
let myString = '123';
let myNumber = Number(myString);
console.log(typeof(myNumber)); // number
```

Operatoren – Gleicheitsoperator == / ===

```
const var1 = 1;
const var2 = '1';
console.log(var1 == var2); //Output?
const var3 = 1;
const var4 = 1;
console.log(var3 === var4); //Output?
console.log(var3 === var2); //Output?
```

snippet-equality.js

Übung I (ca. 5')

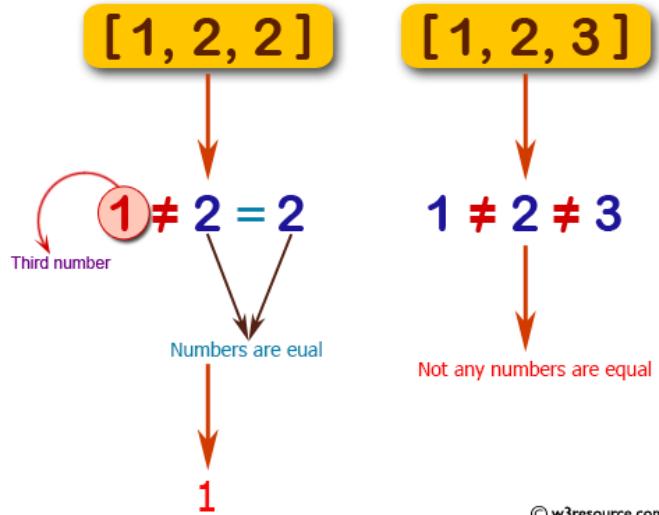
1. Was ist der Unterschied zwischen `let` und `var`?
2. Für was wird `typeof()` verwendet?
3. Was bedeutet schwach typisiert?
4. Was ist der Unterschied zwischen `==` und `===`?
5. Was ist der Unterschied zwischen Primitives und Non-Primitives?

Übung II (5')

```
console.log('' + 1 + 0); // ?  
console.log('' - 1 + 0 ); // ?  
console.log(true + false); // ?  
console.log(6 / '3'); // ?  
console.log('2' * '3' ); // ?  
console.log(7 / 0); // ?  
console.log(null + 1); // ?  
console.log(undefined + 1); // ?  
console.log(21 == '21'); // ?  
console.log(undefined == null); // ?  
console.log(undefined === null); // ?  
console.log(21 === 21); // ?
```

Übung III (10')

1. Gegeben sind drei Nummern. Davon sind zwei gleich. Finde die Dritte.



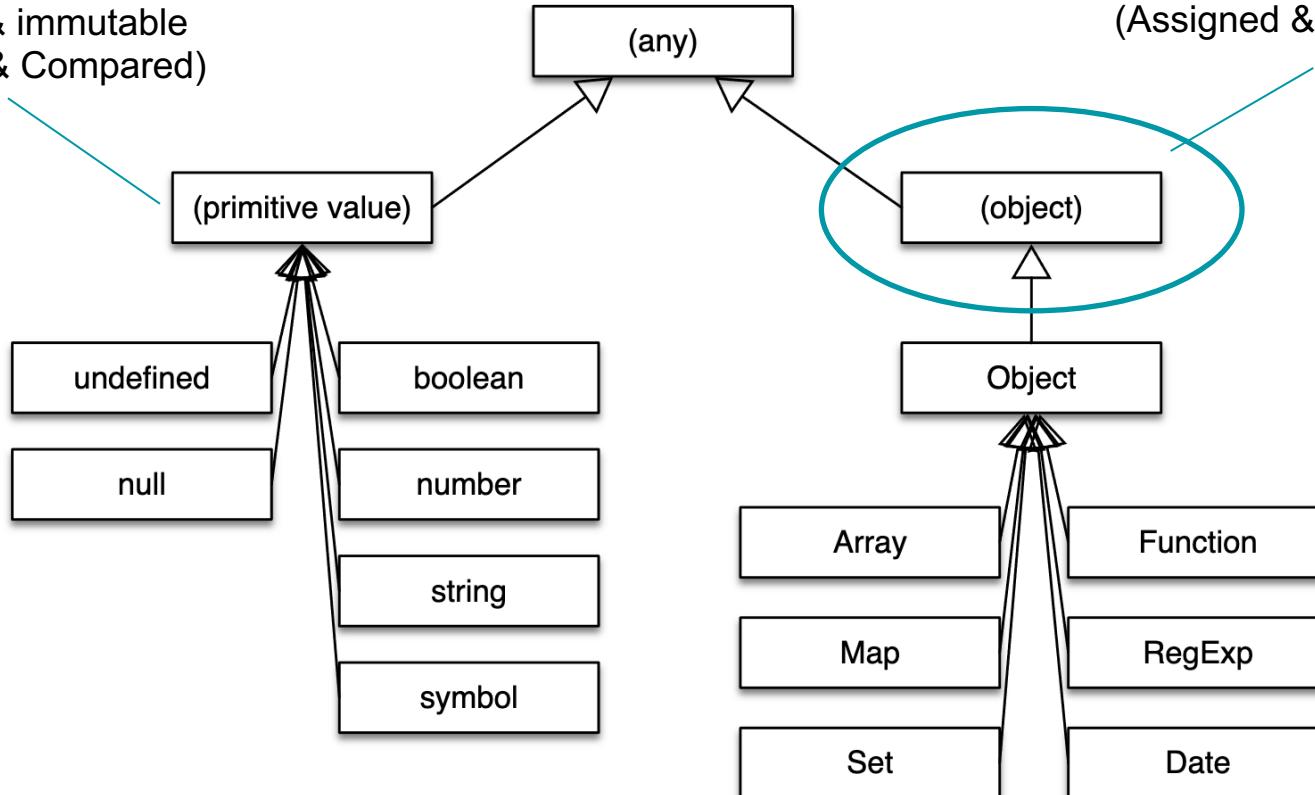
2. Erweitern Sie die Übung so, dass lediglich Ganzzahlen eingegeben werden können (Rückgabe 'null', falls keine Ganzzahlen).

JavaScript Sprachkonzepte III

Objects, Arrays, Functions, Prototype Chains, Classes

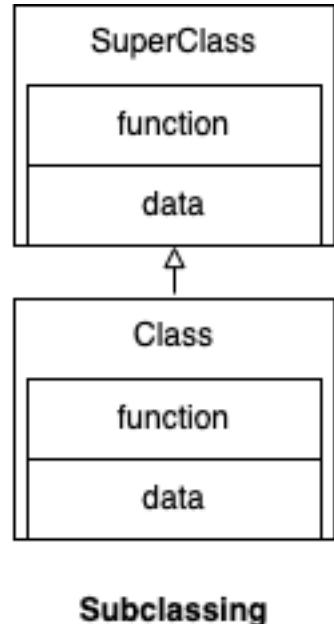
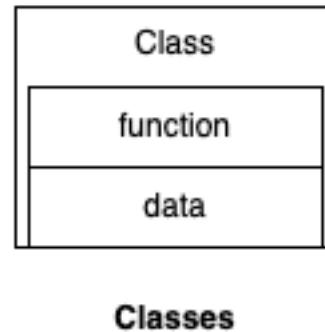
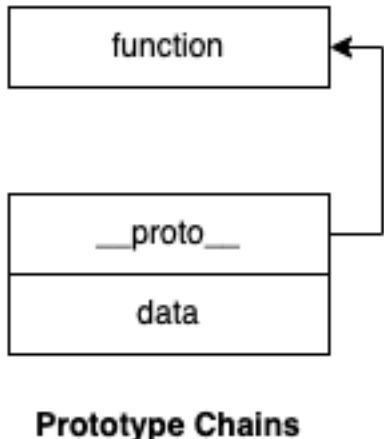
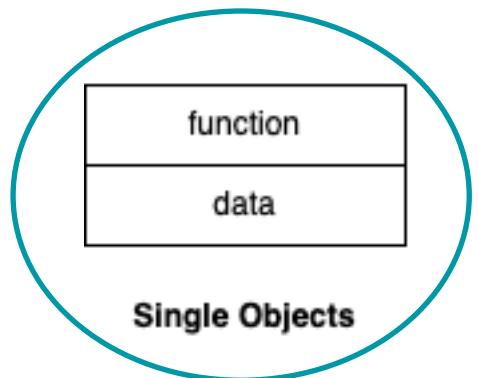
Grundlagen – Datentypen

By value & immutable
(Assigned & Compared)



By reference & mutable
(Assigned & Compared)

Datentypen – Objects



Datentypen – Objects - Erstellung

snippet-object-literal.js

```
const demoObject = { // creating an object with Object literal
    property1: 1,
    myMethod() {
        return 2;
    },
    set myAccessor() {
        return this.property1;
    },
    set myAccessor(value) {
        this.property1 = value;
    },
};
// Accessing object
console.log(demoObject.property1); // 1
console.log(demoObject.myMethod()); // 2
console.log(demoObject.myAccessor); // 1
demoObject.myAccessor = 5;
console.log(demoObject.myAccessor); // 5
```

Datentypen – Objects - Erstellung

snippet-object-constructor.js

```
const demoObject2 = new Object(); // Creation via new Object Constructor
demoObject2.property1 = 1;
demoObject2.myMethod = () => {
    return 2;
};
Object.defineProperty(demoObject2, 'myAccessor', {
    get() {
        return this.property1;
    },
    set(value) {
        this.property1 = value;
    },
});
// Accessing object
console.log(demoObject2.property1); // 1
console.log(demoObject2.myMethod()); // 2
console.log(demoObject2.myAccessor); // 1
demoObject2.myAccessor = 5;
console.log(demoObject2.myAccessor); // 5
```

Datentypen – Objects – Zugriff auf Properties

snippet-access-properties.js

```
const user = {  
    name: 'patrick',  
    age: 34,  
};  
  
console.log(user.name); // Patrick  
console.log(user['name']); // Patrick  
  
const { name, age } = user; // Destructuring assignment  
console.log(name, age); // Patrick 34  
  
let fruit = 'apple';  
  
// Computed Properties  
let bag = {  
    [fruit]: 5,  
};  
console.log(bag.apple); // 5
```

Datentypen – Objects – Properties prüfen

```
const user = {};  
  
console.log(user.mysuperduperproperty === undefined);  
  
const myUser = {  
    name: "Patrick", age: 35  
};  
  
// Properties prüfen  
console.log('name' in myUser);  
console.log('key' in myUser);
```

Datentypen – Objects – Properties ermitteln

```
const myUser = {  
    name: 'Patrick', age: 33  
};  
  
// Properties durchlaufen  
for (const key in myUser) {  
    console.log(key);  
}  
  
// Properties mit Werten durchlaufen  
for (const [key, value] of Object.entries(myUser)) {  
    console.log(` ${key}, ${value}`);  
}
```

Datentypen – Objects - Referenzen

```
const myUser = {  
    name: 'Patrick', age: 33  
};  
  
const myUser2 = myUser; // Zuweisen der Referenz  
myUser2.name = 'Andreas';  
console.log(myUser.name); // Andreas
```

snippet-object-references.js

- Ein grosser **Unterschied** zwischen Primitives und Objects:
 - > Bei Primitives (wie z.B. number, string, etc.) wird der komplette Wert zugewiesen.
 - > Bei Non-Primitives werden die Referenzen zugewiesen.

Datentypen – Objects – Shallow Copy

snippet-object-clone.js

```
const myExampleObject = {  
    name: 'Max',  
    age: 30,  
    hobbies: ['Sports', 'Cooking'],  
    address: {  
        street: 'Mainstreet 1',  
        city: 'Berlin',  
    }  
};  
  
const assign = Object.assign({}, myExampleObject); // shallow copy !  
const spread = { ...myExampleObject }; // shallow copy !
```

Shallow Copy: https://developer.mozilla.org/en-US/docs/Glossary/Shallow_copy

Datentypen – Objects – Deep Copy

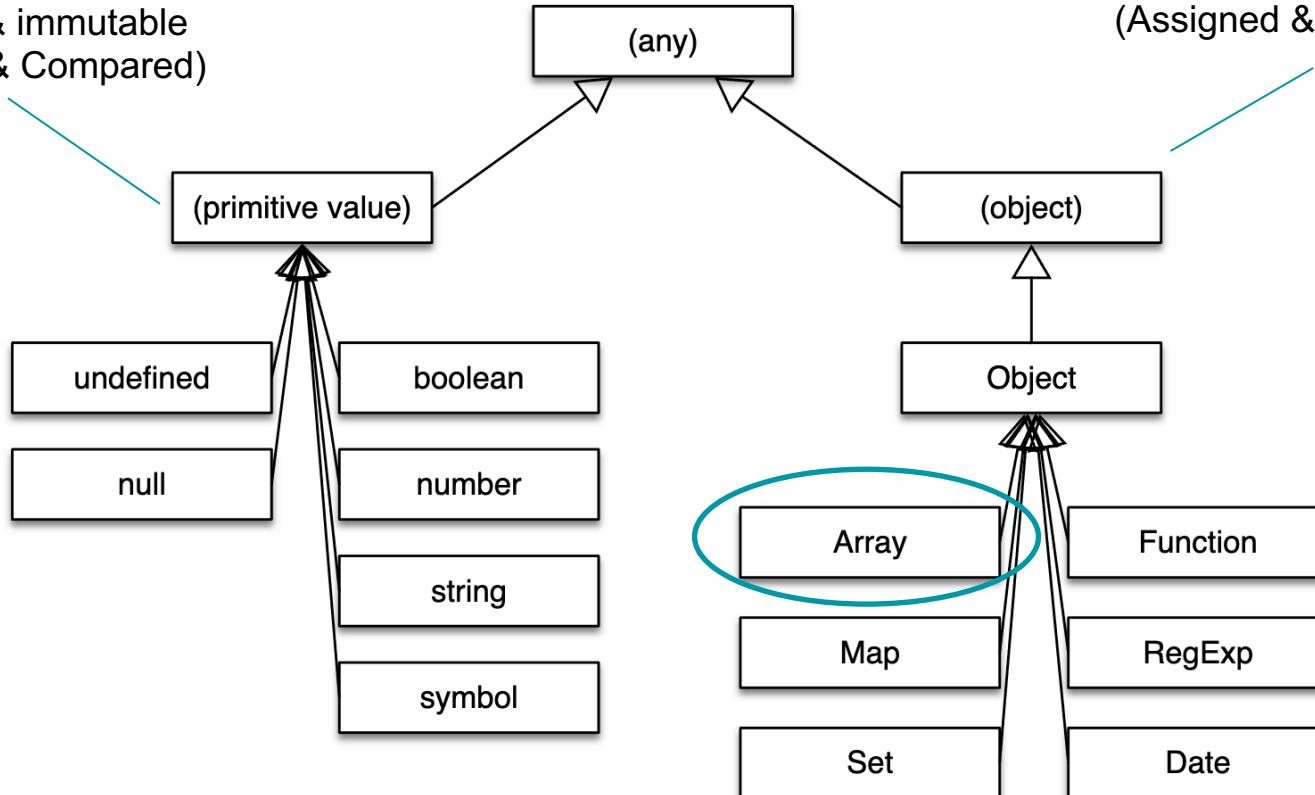
snippet-object-clone.js

```
const myExampleObject = {  
    name: 'Max',  
    age: 30,  
    hobbies: ['Sports', 'Cooking'],  
    address: {  
        street: 'Mainstreet 1',  
        city: 'Berlin',  
    },  
};
```

```
const myExampleDeepClone = deepClone(myExampleObject); // self-made  
const myExampleStructuredClone = structuredClone(myExampleObject);  
const myExampleJsonParse = JSON.parse(JSON.stringify(myExampleObject));
```

Grundlagen – Datentypen

By value & immutable
(Assigned & Compared)



By reference & mutable
(Assigned & Compared)

Datentypen – Objects – Array I

```
const myArray = ['a', 'b', 'c']; // Array literal
console.log(myArray[0]); // 'a'
myArray[0] = 'x';
console.log(myArray[0]); // 'x'
console.log(myArray.length); // 3
myArray.length = 1;
console.log(myArray); // ['x']
myArray.push('y');
console.log(myArray); // ['x', 'y'];
myArray.push(...['z', '1']);
console.log(myArray); // ['x', 'y', 'z', '1'];
console.log(typeof myArray); // object
```

Datentypen – Objects – Array II

snippet-array-methods.js

```
const myArray2 = ['a', 'b', 'c']; // Array literal

for (const [index, value] of myArray2.entries()) {
    console.log(index, value);
}

// Output
// 0 a
// 1 b
// 2 c

const myArray3 = myArray2.slice(1, 2);
console.log(myArray3); // ['b']

const myArray4 = myArray2.map((item) => item + 'x');
console.log(myArray4); // ['ax', 'bx', 'cx']

const myArray5 = myArray4.filter((item) => item === 'bx');
console.log(myArray5); // ['bx']
```

Datentypen – Objects – Array III

snippet-array.js

```
const myArray2 = ['a', 'b', 'c']; // Array literal
let a, bc;
[a, ...bc] = myArray2;
console.log(a, bc); // 'a' ['b', 'c']
```

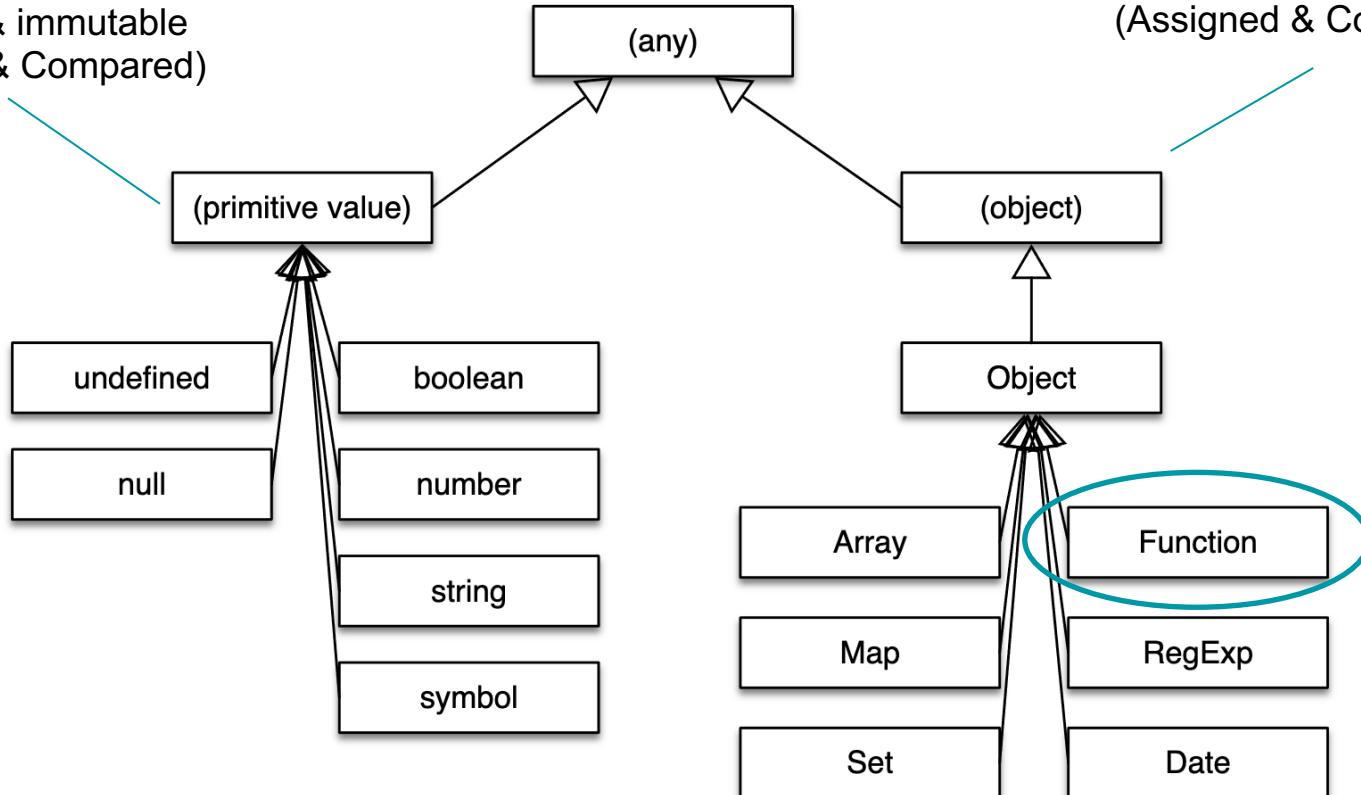
Loops - Überblick

- While
- Do-While
- For-Loop
- For-In → Loopt durch die Properties von einem Objekt
- For-Of → Mit dem **for...of statement** können sogenannte iterable objects durchlaufen werden ([Array](#), [Map](#), [Set](#), das [arguments](#) Objekt und weitere eingeschlossen)

```
const arr = [3, 5, 7];
arr.foo = 'hallo';
for (const i in arr) {
  console.log(i); // logs "0", "1", "2", "foo"
}
for (const i of arr) {
  console.log(i); // logs "3", "5", "7"
}
```

Grundlagen – Datentypen

By value & immutable
(Assigned & Compared)



By reference & mutable
(Assigned & Compared)

Function Declaration

snippet-functions-var-scope.js

```
const newMessage = 'test'; // globale Variable
function printSth() { // function declaration
    const myMessage = 'hello'; // lokale Variable
    console.log(myMessage);
    console.log(newMessage);
}
printSth();
console.log(myMessage); // error, da lokale Variable
console.log(newMessage); // Kein Fehler, da globale Variable
```

- Eine Funktion kann auf eine Variable ausserhalb der Funktion zugreifen.
- Die äussere Variable (= globale Variable) kann verwendet werden, wenn es keine Variable innerhalb des Blocks gibt.

Functions – (default) Parameter

```
function printSth(message, sender='Patrick') {  
    console.log(` ${message} ${sender}`);  
}  
  
printSth('hallo');  
printSth('hallo', 'Andreas');
```

snippet-functions-default-parameter.js

- Funktionen können default Parameter besitzen.
- Ein default Parameter können auch durch eine separate Funktion zusammengesetzt werden.

Functions – Function Expressions

```
const myConsolePrint = function (message) {  
    console.log(message);  
};  
  
myConsolePrint('this is a message');
```

snippet-function-expression.js

Diese Deklaration einer Function von `myConsolePrint` nennt man **function expression**.

Functions – Callback Functions

```
function display(result) {  
  console.log(`Result: ${result}`);  
}  
  
function sum(num1, num2, callback) {  
  const result = num1 + num2;  
  callback(result);  
}  
  
sum(1, 2, display);
```

snippet-callback-function.js

Functions – Arrow Functions

```
const sum = (a, b) => a + b;  
console.log(sum(1,2));
```

snippet-arrow-function.js

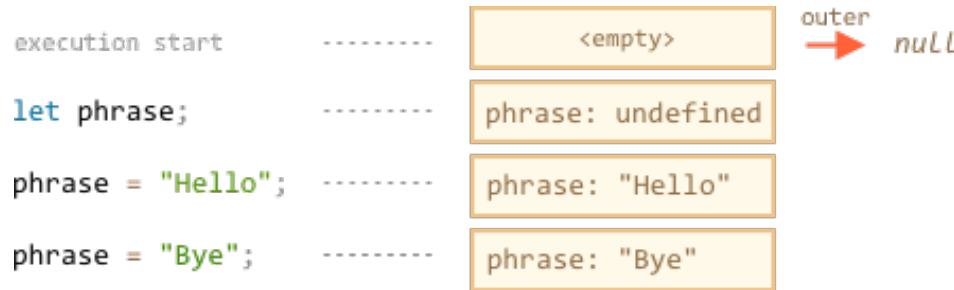
Functions - Lexical Scope

- In JavaScript, jede lauffähige `function` hat einen **lexikalalen Scope**.
- Der Scope besteht aus zwei Aspekten
 - > **Environment Record**, ein Objekt, welche alle lokalen Variablen und seine Properties speichert.
 - > **Eine Referenz auf den äusseren lexikalalen Scope**.
- D.h. eine Variable ist lediglich ein Property von diesem Environment Record.

Functions - Lexical Scope

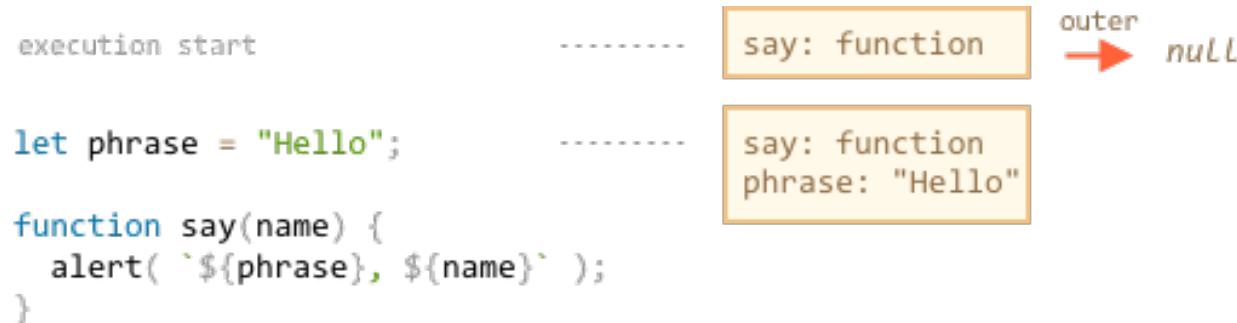


Functions - Lexical Scope



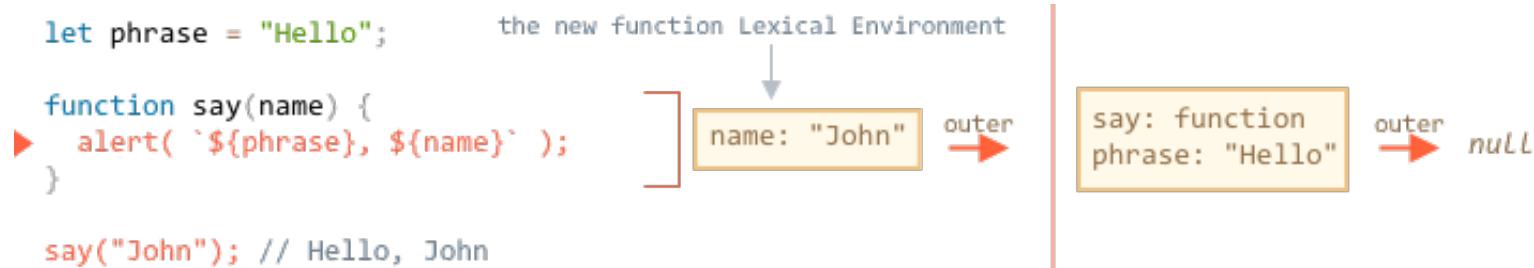
Functions - Lexical Scope

- Function Declarations werden sobald der Lexical Scope erstellt wird, innerhalb des Scopes erstellt.
- Function expressions resp. Arrow functions werden bei der Erstellung nicht innerhalb des Scopes erstellt!



Functions - Lexical Scope – Access outer var

- Zwei lexikale Scopes: Der Innere (function call) und der äussere (global)
- Sobald der Code den Zugriff auf eine Variable sucht, prüft er den inneren lexikalischen Scope zuerst und erst dann den Äusseren (bis zum Globalen).



Functions - Lexical Scope - Closure

- Eine Closure ist eine function, welche sich an äussere Variablen erinnert und auf diese entsprechend zugreifen kann.
- Deshalb können in JavaScript alle Functions Closures sein.
- Exemplarische Closure als “self-invoking function”:

snippet-closure.js

```
let add = (function () {
    let counter = 0;
    return function () {counter += 1; return counter;};
})();

add(); //1
add(); //2
add(); //3
```

- Mit diesem Beispiel kann die Counter Variable von aussen nicht zugänglich gemacht werden.

Functions - Zusammenfassung

- Zwei Arten von lexical scopes: Die **Inneren** (function call) und der **Äussere** (global). Der innere Scope zeigt auf einen Äusseren.
- Sobald der Code den Zugriff auf eine Variable sucht, prüft er den inneren lexikalischen Scope zuerst und erst dann der Äussere (bis zum Globalen).
- Pro ausgeführte Funktion wird ein lexikaler Scope erstellt.
- Der lexikale Scope ist ein Spezifikationsobjekt und kann nicht direkt manipuliert werden. JavaScript Engines optimieren diese (entfernen von nicht benutzten Variablen, um Memory zu optimieren, etc.)

Übung I (10')

```
// Vor dem Funktionsaufruf
let menu = { width: 200, height: 300, title: "Titel" };
multiProperties(menu);
// Nach dem Funktionsaufruf
menu = { width: 400, height: 600, title: "Titel" };
```

Schreibe eine Funktion, welche alle number Properties um 2 multipliziert (generisch!).

Übung: exercises/exercise-multiproperties.js

→ Beachte, dass die Funktion nichts zurückgibt!

Übung II (10')

```
const salaries = {  
    patrick: 100000,  
    andreas: 110000,  
    gwendolin: 91000,  
    nayoona: 45000  
};
```

Schreibe eine Funktion, welche alle Löhne des Objektes 'salaries' aufsummiert (generisch!).

Übung: exercises/exercise-salaries.js

Zusatz: Übungsfrage

```
const name = 'Patrick';
console.log(`hello ${1}`); // ?
console.log(`hello ${"name"}`); // ?
console.log(`hello ${name}`); // ?
```

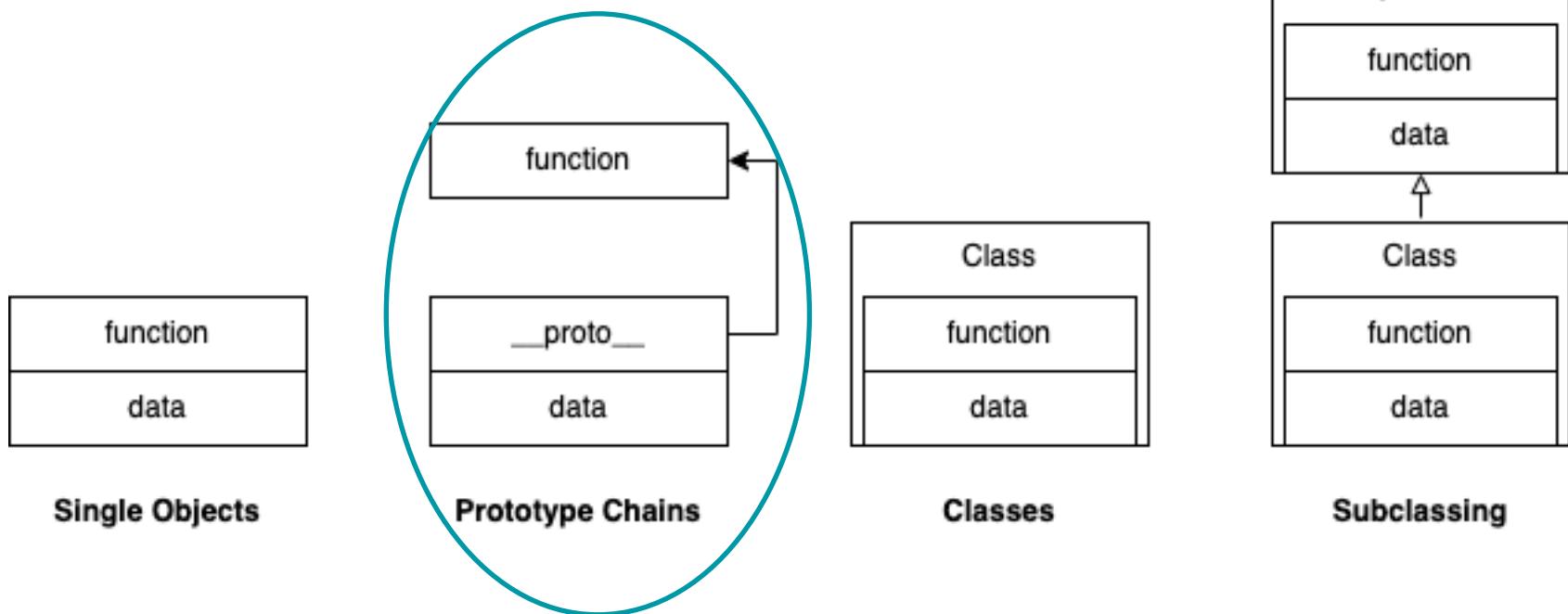
// Funktioniert das? Warum?

```
const user = { name: "John" };
user.name = "Pete";
```

JavaScript Sprachkonzepte IV

Prototype Chains und “this” Keyword

Objects - Prototype Inheritance



Objects - Prototype Inheritance

- Bei der Programmierung wollen wir oft bestimmte Dinge erweitern und wiederverwenden.
- Prototypische Vererbung ist ein JavaScript Language Feature, welches uns dies ermöglicht.

Objects - Prototype Inheritance

snippet-prototype-inheritance.js

```
const vehicle = {  
    hasEngine: true  
};  
  
let car = {  
    hasFourWheels: true  
};  
  
car = Object.setPrototypeOf(car, vehicle);  
console.log(car.hasEngine); //Output: true  
console.log(car.hasFourWheels); //Output: true
```

Prototype Object



Objects - Prototype Inheritance

Prototype Object

```
const car2 = Object.create(vehicle, {hasFourWheels: {value: true}});
console.log(car2.hasEngine); //Output: true
console.log(car2.hasFourWheels); //Output: true
```

```
//Prototype Chain
console.log(car2.__proto__ === vehicle); // Output: true
console.log(vehicle.__proto__ === Object.prototype); // Output: true
console.log(Object.prototype.__proto__ === null); // Output: true
```

Objects - Prototype Inheritance

- Wenn wir ein Property von einem Objekt lesen möchten und dieses nicht vorhanden ist, wird es vom Prototype Objekt gelesen. Dieses Pattern nennt man “**prototypal inheritance**”.
- Das Property `[[Prototype]]` ist intern, aber kann via `Object.get|setPrototypeOf` / `__proto__` abgefragt resp. gesetzt werden. ([siehe MDN Dokumentation](#))

Objects - Verwendung von «this»

snippet-this-keyword.js

```
const car = {  
    name: 'my car',  
    age: 10,  
    start() {  
        console.log(`start ${this.name}`);  
    }  
}
```

Um den Zugriff auf das Objekt zu erhalten, kann man das `this` Keyword verwenden.

Objects - Verwendung von «this»

snippet-this-keyword.js

```
const car = {  
    name: 'my car',  
    age: 10,  
    start() {  
        console.log(`start ${this.name}`);  
    }  
}
```

Tipp um **this** im JS Kontext besser zu verstehen:

Man stellt sich **this** einfach als impliziter Parameter im Function Call vor.

Im Gegensatz zu anderen Programmiersprachen ist das **this** Keyword nicht gebunden, d.h. **this** wird während der Laufzeit auf Basis des entsprechenden Kontext aufgebaut. Es kann alles sein.

Objects - Verwendung von «this»

snippet-this-context.js

```
let car1 = {name: 'car1'};  
let car2 = {name: 'car2'};  
  
function start() {  
    console.log(`start ${this.name}`);  
}  
  
car1.start = start;  
car2.start = start;  
  
car1.start(); // Output: start car1  
car2.start(); // Output: start car2
```

Objects - Verwendung von «this» - call()

snippet-this-call.js

```
const car = {  
    name: 'my car',  
    age: 10,  
    start() {  
        console.dir(this);  
        console.log(`start ${this.name}`);  
    },  
};  
  
const car2 = {  
    name: 'my car 2',  
};  
  
car.start.call(car2); // 'start my car 2'
```

Functions haben auch Functions. `.call()` macht den normalerweise impliziten `this` parameter explizit.

Objects - Verwendung von «this» - bind()

snippet-this-bind.js

```
const car = {  
    name: 'my car',  
    age: 10,  
    start() {  
        console.log(this);  
        console.log(`start ${this.name}`);  
    },  
};  
  
const car2 = {  
    name: 'my car 2',  
};  
  
const start = car.start.bind(car2);  
start(); // 'start my car 2'
```

Mittels `.bind()` kann man aus `start()` eine funktionsfähige standalone Function machen, indem man ihr einen Objektkontext mitgibt.

Objects - «new» & constructor functions

snippet-constructor-functions.js

```
function Student(name) {  
    this.name = name;  
}  
  
const student = new Student('Patrick');  
console.log(student.name);
```

Konventionen:

- Erster Buchstabe der Constructor Function gross
- Sollte nur mittels dem "new" Operator ausgeführt werden.

Objects - «new» & constructor functions

snippet-closure.js

```
function Student(name) {  
    // this = {}; -> implizit  
    this.name = name;  
    // return this; -> implizit  
}  
  
let student = new Student('Patrick');  
console.log(student.name);
```

1. Es wird ein neues leeres Objekt erstellt und zu this zugewiesen.
2. Der function Body wird ausgeführt. Modifiziert this in dem das entsprechende Property angehängt wird.
3. Der Wert von this wird zurückgegeben.

Übungsaufgabe I

exercise-prototype.js

```
let user = {
  doSth() {
    if (!this.isPausing) {
      console.log('I am doing sth.');
    }
  },
  pause() {
    this.isPausing = true;
  }
};

let admin = {
  name: 'Admin User',
  __proto__: user
};

admin.pause();
console.log(admin.isPausing); //? Wieso?
console.log(user.isPausing); //? Wieso?
```

Übungsaufgabe II - Erstelle einen Calculator (20')

- Erstelle ein Calculator mit folgenden drei Methoden:
 - > `read()` → einlesen von zwei Werten und speichere diese als Objekt Properties
 - > `sum()` → summieren der zwei gespeicherten Werte
 - > `mul()` → multiplizieren der zwei gespeicherten Werte
- Erweitere den Rechner so, dass nicht nur 2 Werte eingelesen werden können sowie dass dieser auch mit ungültigen Werten umgehen kann.
- Übung: exercises/exercise-calculator.js

```
const calculator = {  
    // ... your code ...  
};  
calculator.read(1, 2);  
console.log( calculator.sum() );  
console.log( calculator.mul() );
```

Übungsaufgabe II - «Chaining» (10')

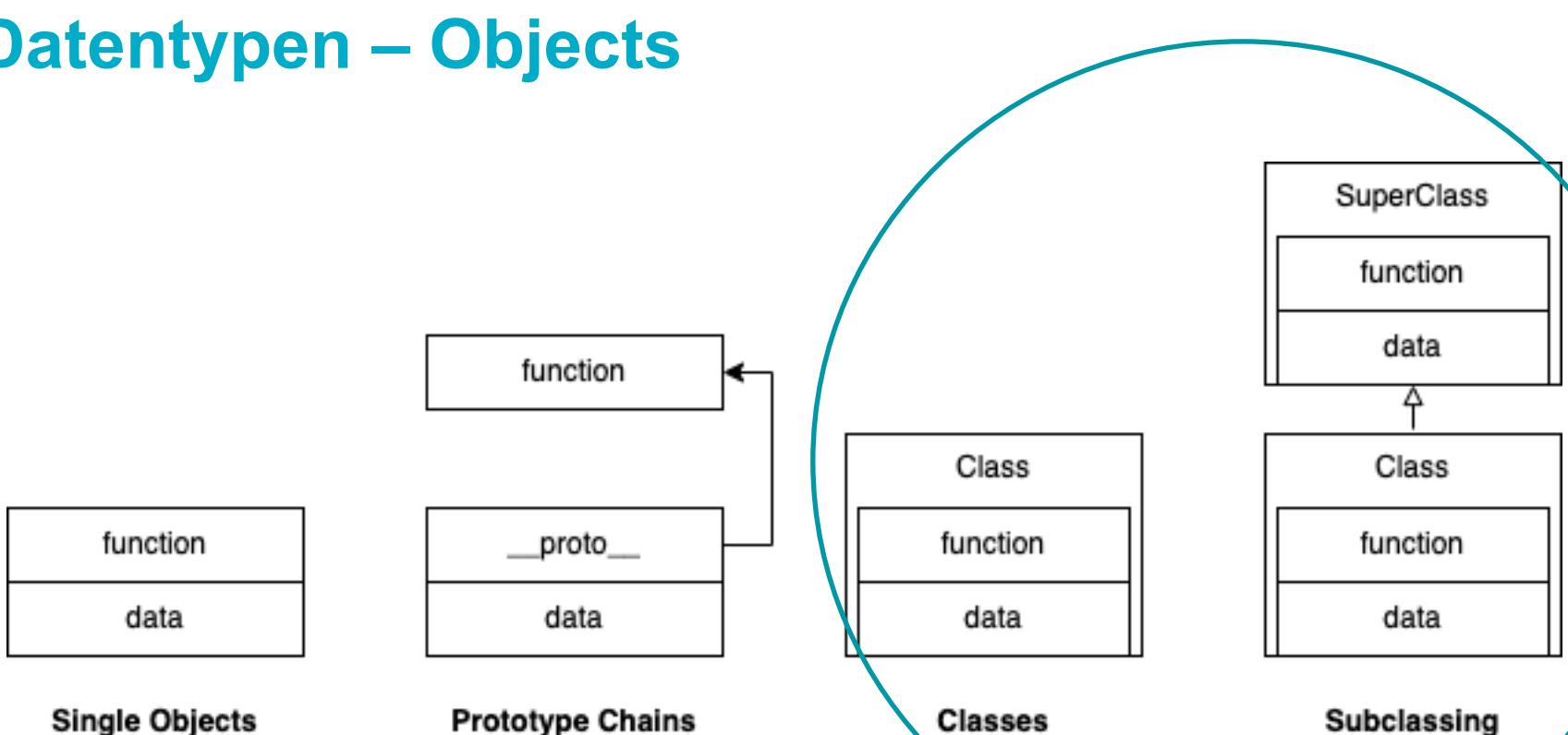
- Implementiere die Leiter und modifiziere den Leiter Code, dass er "chainable" ist.
- Übung: ./exercises/exercise-ladder.js

```
ladder.up().up().up().down().currentStep(); // 2
```

JavaScript Sprachkonzepte V

Classes

Datentypen – Objects



Classes

- In JavaScript ab ES6 gibt es das `class` Konstrukt, das “syntactic sugar” für die objektorientierte Programmierung ist.

snippet-class.js

```
class MyClass {  
    constructor() {}  
  
    method1() {}  
    method2() {}  
}  
  
const myObj = new MyClass(); // Neues Objekt
```

Classes - Beispiel

snippet-class-method.js

```
class Person {  
  
    constructor(name) {  
        this.name = name;  
    }  
  
    sayHello() {  
        console.log(`hello ${this.name}`);  
    }  
}  
  
console.log(new Person('Patrick').sayHello());
```

Classes – Was ist eine Klasse?

- In JavaScript entspricht eine Klasse einer speziellen Art function.

```
class Person {  
  
    constructor(name) {  
        this.name = name;  
    }  
  
    sayHello() {  
        console.log(`hello ${this.name}`);  
    }  
}  
  
console.log(typeof(Person));  
console.dir(Person);
```

Classes – Was ist eine Klasse?

Das `class` Konstrukt macht die folgenden Dinge:

1. Erstellt eine function namens Person
2. Speichert die Methode sayHello im Person.prototype

```
console.log(typeof Person);
console.log(Person === Person.prototype.constructor);
console.log(Person.prototype.sayHello);
console.log(Object.getOwnPropertyNames(Person.prototype));
console.log(Object.getPrototypeOf(new Person()));
```

snippet-class-typeof.js

Classes – getter & setters

```
class Person {  
    constructor(name) {  
        this._name = name;  
    }  
  
    get name() {  
        return this._name;  
    }  
  
    set name(value) {  
        this._name = value;  
    }  
}
```

snippet-class-property.js

Mit ECMAScript 2022 kam das Feature “private class fields” welches ermöglicht Instanzvariablen von äusseren Zugriff zu schützen ([Proposal](#)).

```
class Person {  
#age; // Private class fields  
constructor(name, age) {  
    this.#age = age;  
}  
  
get age() {  
    return this.#age;  
}  
}
```

snippet-class-property.js

Classes – Vererbung

snippet-class-extends.js

```
class Person {  
    constructor(name) {this.name = name;}  
}  
  
class Admin extends Person {  
    doAdminTasks () {  
        console.log(`${this.name} doing admin tasks`);  
    }  
}  
  
const myAdmin = new Admin('Patrick');  
myAdmin.doAdminTasks();
```

Classes – Methoden überschreiben

snippet-class-method-override.js

```
class Person {  
    constructor(name) {this.name = name;}  
    sayHello() {  
        console.log(`hello ${this.name}`);  
    }  
}  
  
class Admin extends Person {  
    doAdminTasks () {  
        console.log('doing admin tasks');  
    }  
    sayHello() {  
        super.sayHello();  
        console.log('hello from admin');  
    }  
}  
  
let myAdmin = new Admin('Patrick');  
myAdmin.sayHello();
```

Übungsaufgabe – Classes (5')

- Schreibe die Person Klasse mit puren functions.

```
const person = new Person('Patrick');
person.sayHello(); // Hello Patrick
```

Zusatz: Übungsaufgabe (20')

- Schreibe eine Uhr die tickt (nach OOP-Prinzipien).

```
const clock = new Clock('h:m:s');
clock.start();
```

- Schreibe eine erweiterte Clock (extends), bei welcher die Präzision (Anzahl ms) mitgegeben werden kann.

```
const extendedClock = new ExtendedClock('h:m:s', 2000);
extendedClock.start();
```

- Tipp: [setInterval](#) anschauen

JavaScript Sprachkonzepte VI

Error Handling & Promises, `async` & `await`

Error Handling mit try...catch

snippet-try-catch.js

```
try {  
    // code  
} catch(error) {  
    // error handling  
}
```

- `try...catch` funktioniert lediglich für Laufzeitfehler (z.B. funktioniert nicht für Syntaxfehler)
- `try..catch` funktioniert nur in einem synchronen Kontext (z.B. funktioniert das `try..catch` um ein `setInterval()` nicht)

Error Handling – das Error Objekt

snippet-try-catch.js

```
try {  
    // code  
} catch(error) {  
    // error handling  
}
```

- Das Error Objekt besteht aus folgenden zwei Properties:
 - name: Name / Typ des Fehlers
 - message: Textuelle Beschreibung des Fehlers

Error Handling – eigene Fehler werfen

Snippet-try-throw-catch-error.js

```
try {  
    // code  
    throw new Error('Das ist ist ein Fehler');  
} catch(error) {  
    // error handling  
}
```

- Es gibt verschiedene [Error Typen](#)
- Error → kann als Basis Klasse für benutzerdefinierte Fehler verwendet werden

Error Handling – finally

snippet-try-finally.js

```
try {  
    // code  
} catch(error) {  
    // error handling  
} finally {  
    // execute always  
}
```

- Der finally-Block wird immer ausgeführt.

Error Handling – Zusammenfassung

- Der `try...catch...finally` Mechanismus erlaubt es Laufzeitfehler zu behandeln.
- Folgende Properties gibt es im Fehlerfall typischerweise
 - message
 - name
 - (stack → nicht standard)
- In einem Catch-Block muss nicht unbedingt das Fehlerobjekt verwendet werden (→ catch ohne Error Objekt)

Promises - Problem

- Viele Aktionen in JavaScript sind asynchron.
- Für die Behandlung können Callback-Funktionen verwendet werden.

snippet-promise-problem-chaining.js

```
first(function (result) {
    second(result, function (newResult) {
        third(newResult, function (finalResult) {
            console.log(`Got the final result: ${finalResult}`);
        }, failureCallback);
    }, failureCallback);
}, failureCallback);
```

Promises - Intro

- Typischerweise haben wir einen Code der Inhalte produziert und Code, der Inhalte konsumiert.

```
first(function (result) {  
    second(result, function (newResult) {
```

snippet-promise-problem-chaining.js

- Das Promise Konstrukt bringt die beiden Aspekte zusammen.
- Der produzierende Code braucht für die Produktion entsprechend Zeit und die Promise stellt den Output entsprechend dem konsumierenden Code zur Verfügung - quasi als “**Versprechen**” für die Zukunft.

Promises - Syntax

snippet-promise.js

```
const promise = new Promise((resolve, reject) => {
  // der Code, der sobald ausgeführt
  // wird, wenn die Promise erstellt wird.
  setTimeout(() => resolve('its done.'), 1000);
});

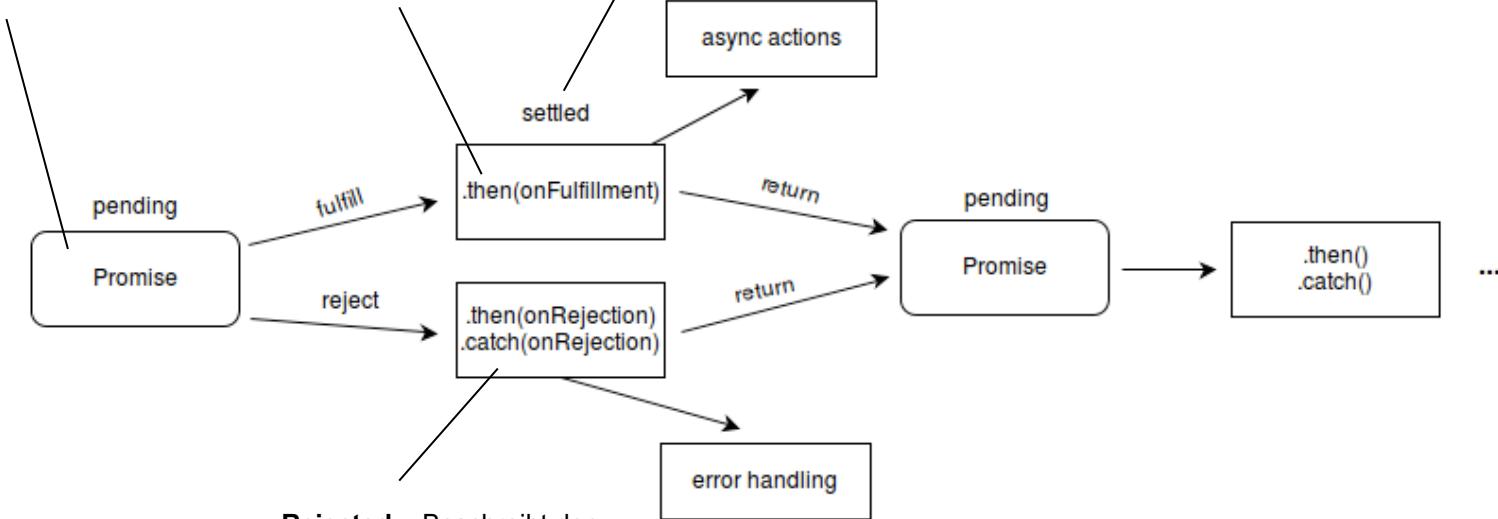
promise.then((result) => {
  console.log(result); // It's done.
});
```

Promises - Intro

Pending = schwiegend,
d.h. initialer Status, weder
fulfilled noch rejected

Fulfilled = erfüllt, d.h. die
Operation wurde
erfolgreich abgeschlossen

Settled = Beschreibt den
Zustand erledigt, aber nicht
zwingend erfolgreich



Rejected = Beschreibt den
Zustand erledigt, aber nicht
zwingend erfolgreich

Promises

Was ist der Output vom folgenden Code?

snippet-promise-multiple-resolve.js

```
const myPromise = new Promise((resolve, reject) => {
    resolve(1);
    setTimeout(() => resolve(2), 1000);
});

myPromise.then(console.log);
```

Promises - Chaining

snippet-promise-chaining.js

```
new Promise(function(resolve, reject) {
    // producer
    setTimeout(() => resolve(1), 1000);
}).then(function(result) {
    // then 1
    console.log(result);
    return ++result;
}).then(function(result) {
    // then 2
    console.log(result);
});
```

Promises – Chaining – Was passiert hier?

snippet-promise-then.js

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
});

promise.then((result) => {
  console.log(result);
  return ++result;
});

promise.then((result) => {
  console.log(result);
  return ++result;
});
```

Promises – Chaining – Mehrwert

snippet-promise-chaining-starting-example.js

```
first(function (result) {  
    second(result, function (newResult) {  
        third(newResult, function (finalResult) {  
            console.log(`Got the final result: ${finalResult}`);  
        }, failureCallback);  
    }, failureCallback);  
}, failureCallback);
```



```
first()  
  .then(second)  
  .then(third)  
  .then((result) => console.log(`Got the final result ${result}`))  
  .catch(error);
```

Async / Await

- `async & await` (ES8) ist «Syntactic Sugar» auf Basis von Promises.
- `async & await` macht es einfacher asynchronen Code zu schreiben und zu lesen.

```
function fetchJsonViaPromises(url) {  
    return fetch(url) // async  
        .then((request) => request.text()) // async  
        .then((text) => JSON.parse(text)) // sync  
        .catch((error) => {  
            console.error(error);  
        });  
}
```

snippet-promises-vs-async.js

```
async function fetchJsonViaAsync(url) {  
    try {  
        const request = await fetch(url); // async  
        const text = await request.text(); // async  
        return JSON.parse(text); // sync  
    } catch (error) {  
        console.error(error);  
    }  
}
```

Übungsfrage (5')

- Gibt es hier einen Unterschied? Wenn ja, welchen?

```
promise.then(fn1).catch(fn2);
```

// vs.

```
promise.then(fn1, fn2);
```

Übungsaufgabe – Promises (15')

- Erstelle eine Promise, welche nach einer angegeben Zeit in ms den übergebenen String wieder zurück gibt. Der Code soll wie folgt anwendbar sein:

```
delay(2000, 'hallo').then(console.log);
```

Übung: ./exercises/exercise-promise.js

Anhang