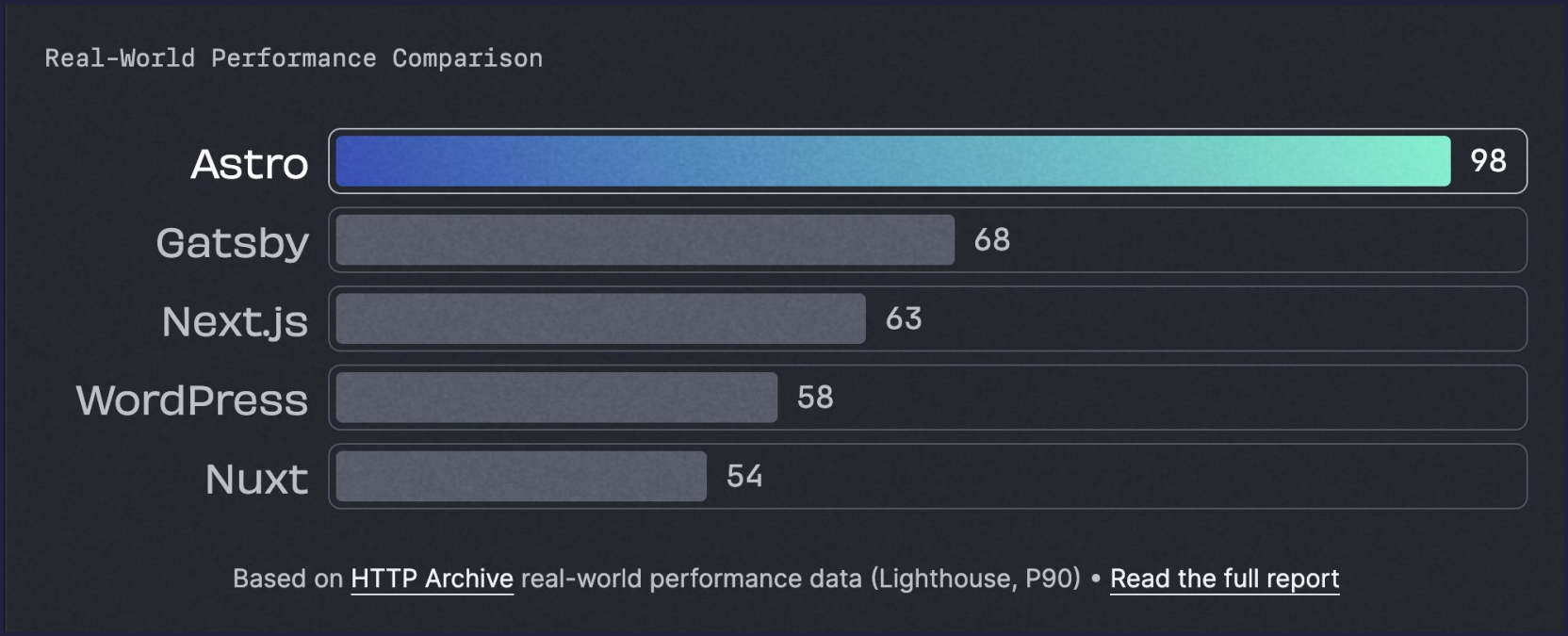

astro
Build the Web you want

“Astro is the **all-in-one** web framework designed for **speed**. Pull your **content** from anywhere and deploy everywhere, all powered by **your favorite UI components** and libraries.”


Astro
SSR w/
Fred!


Real-World Performance Comparison
Astro
98
Gatsby
68
Next.js
63
WordPress
58
Nuxt
54
Based on HTTP Archive real-world performance data (Lighthouse, P90) • Read the full report

March 7, 2023

2023 Web Framework Performance Report

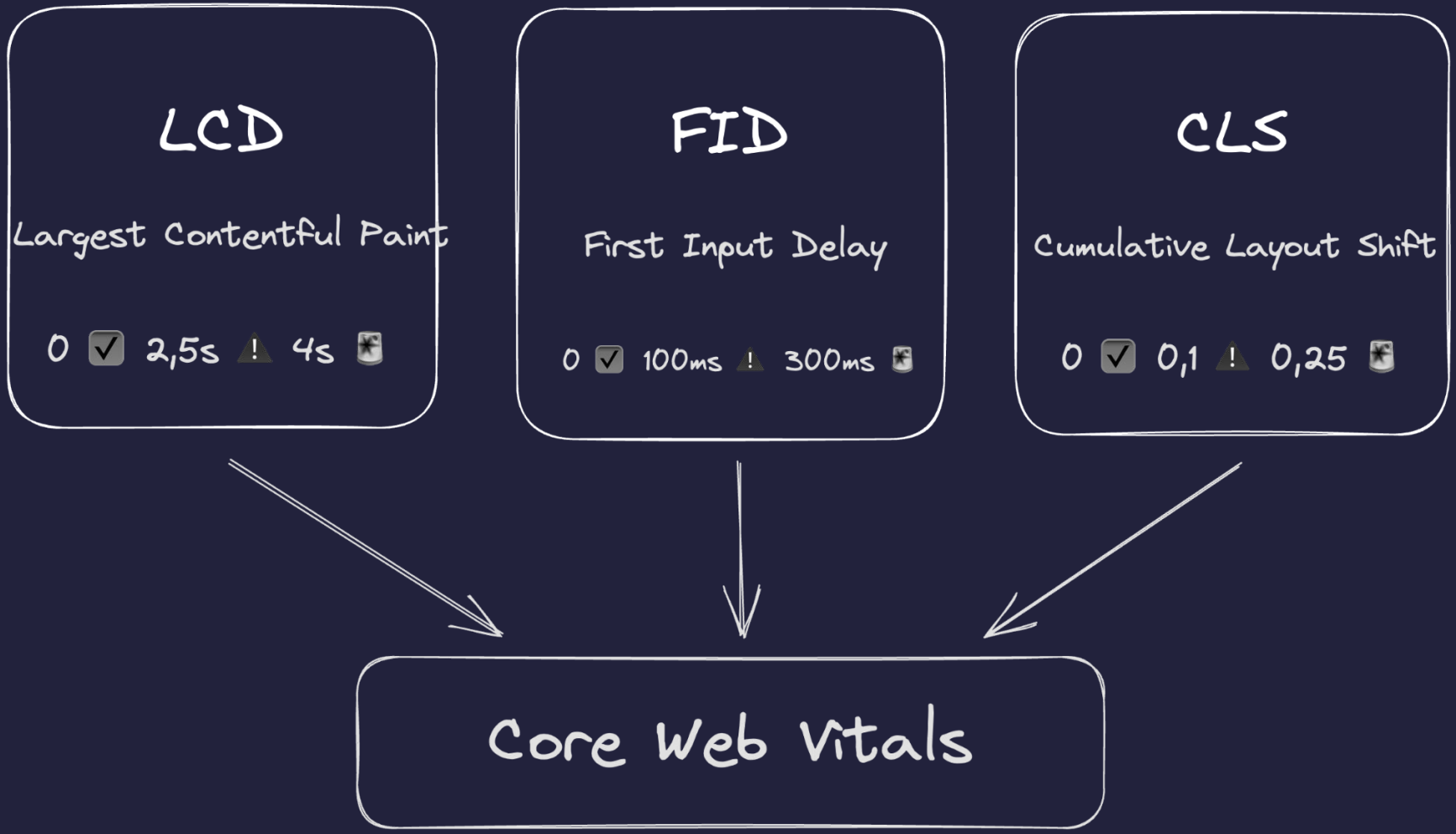
By  Fred Schott

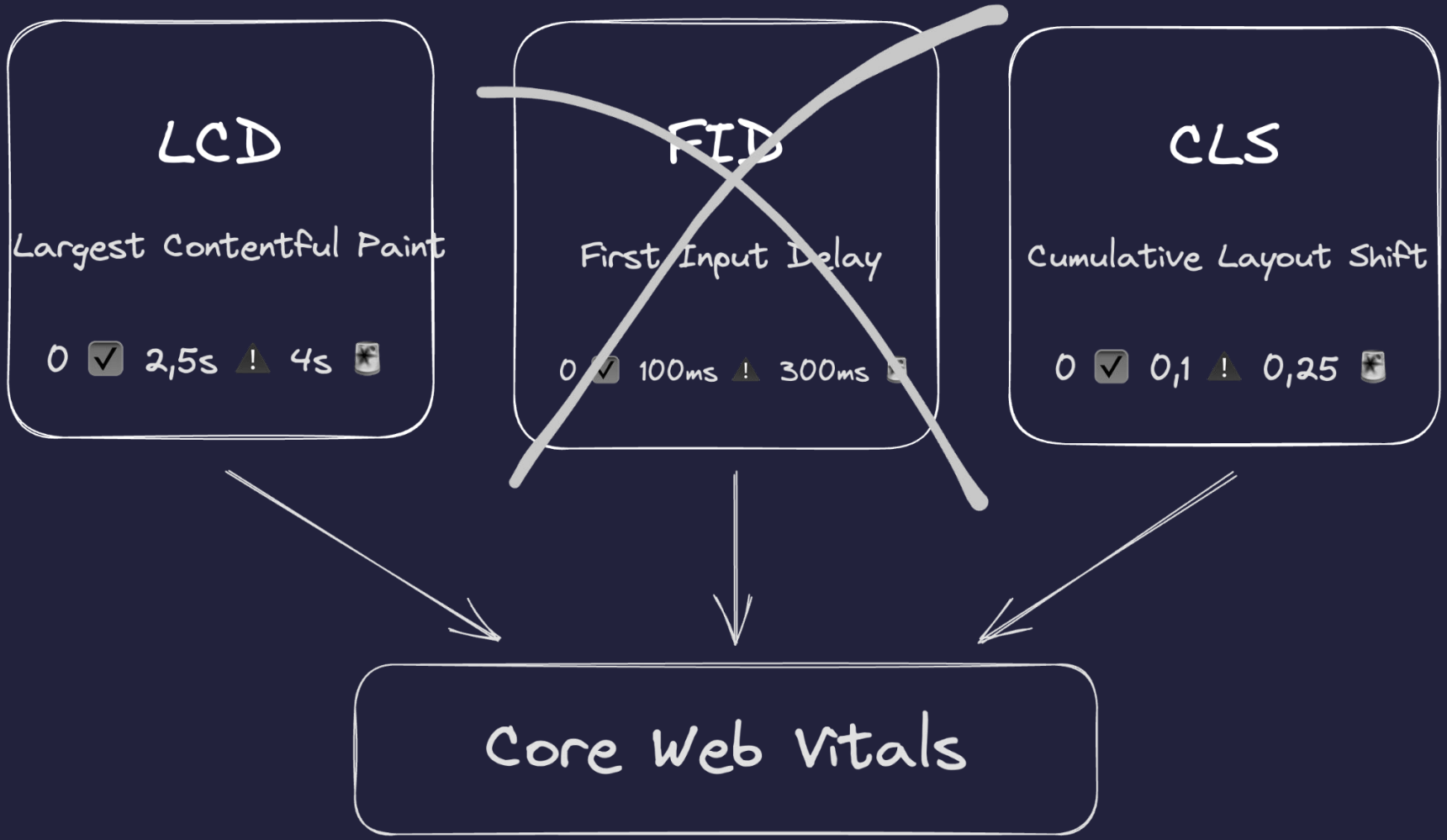
Worum geht´s?

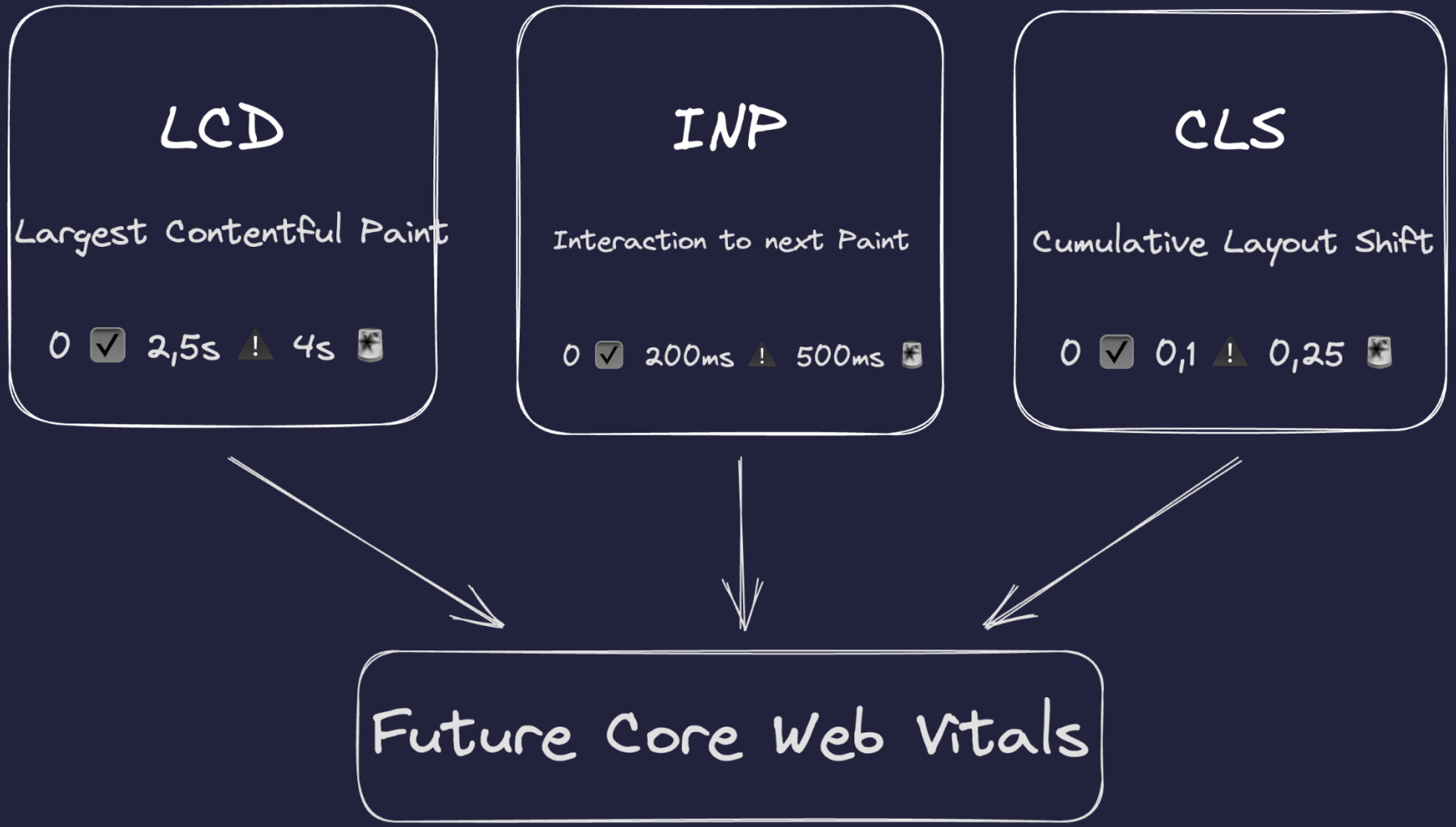
- Bewertung der realen Nutzung und Leistung moderner Frameworks
- Bewertung durch Core Web Vitals Metriken
- Rolle von Javascript

Web Vitals

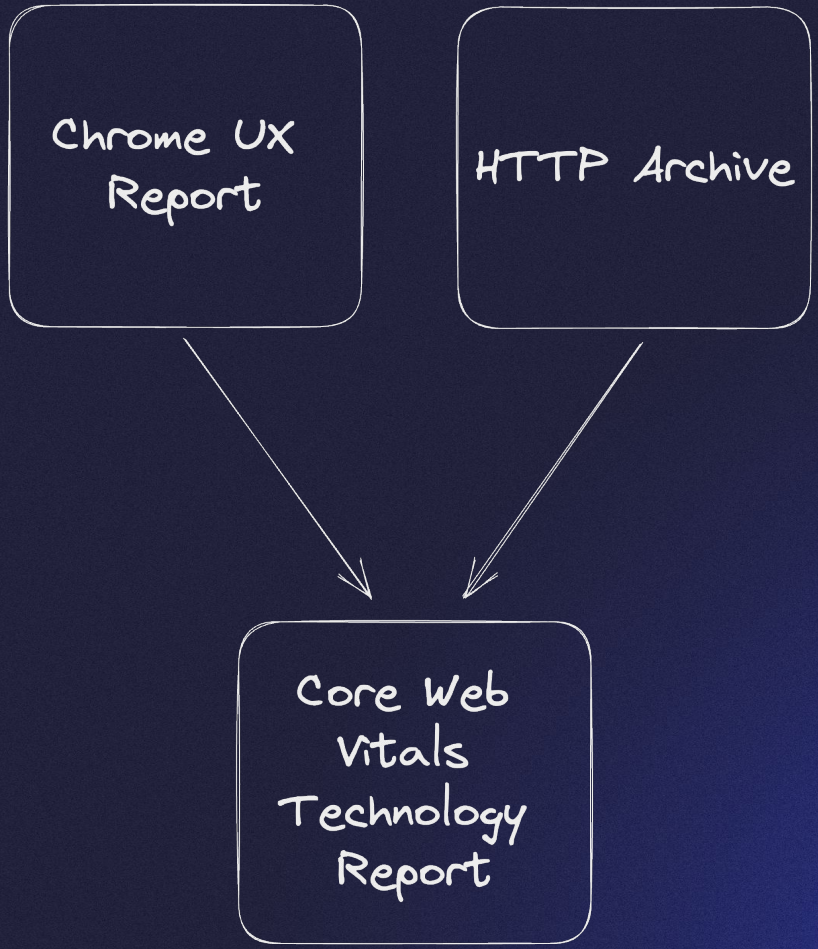
- Metriken zur Messung der User Experience im Web
- Von Google
- Beeinflussen SEO stark
- Je besser die Metrik, desto besser das Ranking in Google Search (und anderen Search Engines)
- Zufriedenere Kunden, besseres Image
- Kunden bleiben eher auf der Seite
- Höhere Conversion Rate
- Einfach wichtig

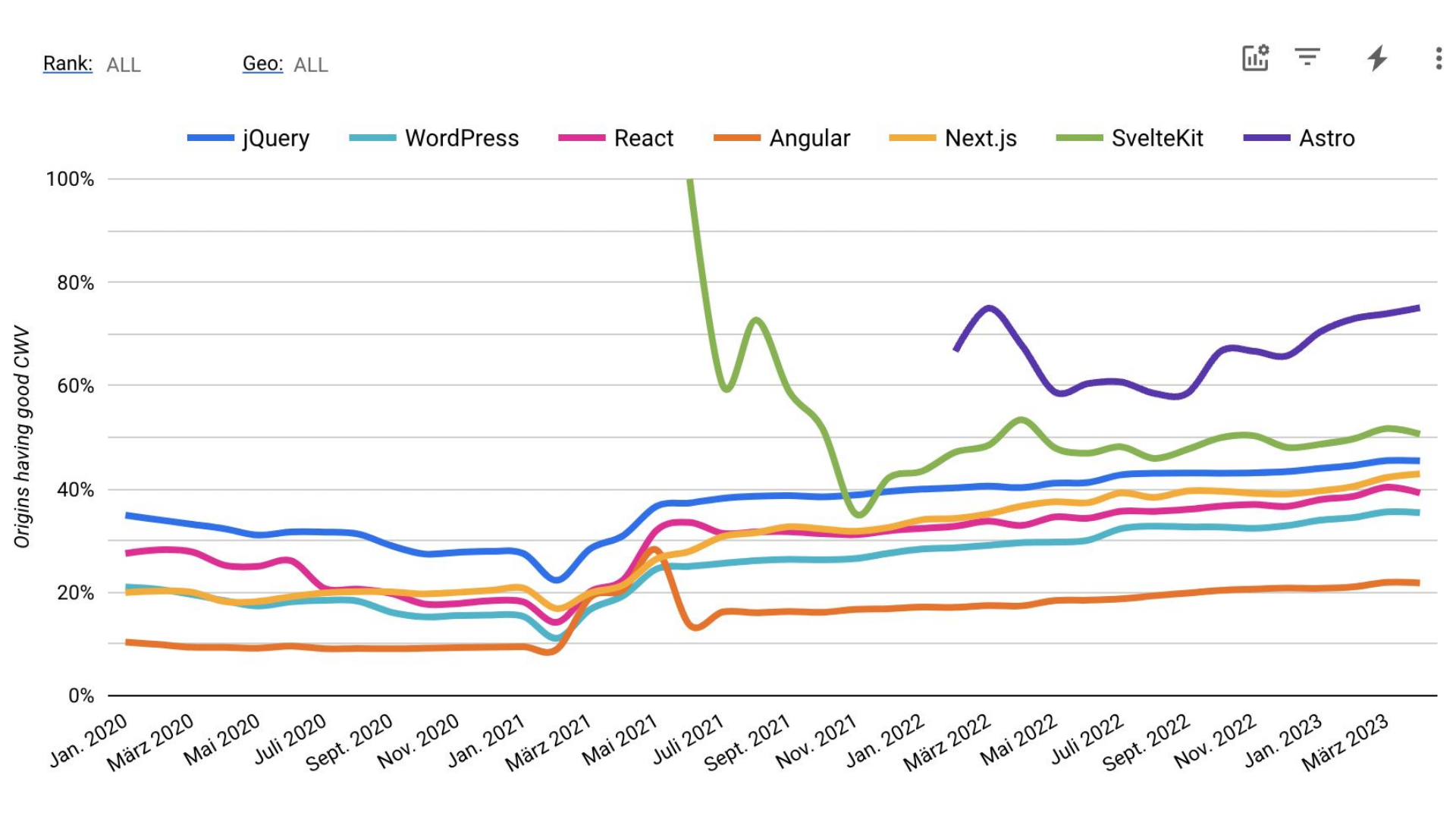

LCD
Largest Contentful Paint
0 2,5s 4s
FID
First Input Delay
0 100ms 300ms
CLS
Cumulative Layout Shift
0 0,1 0,25
Core Web Vitals

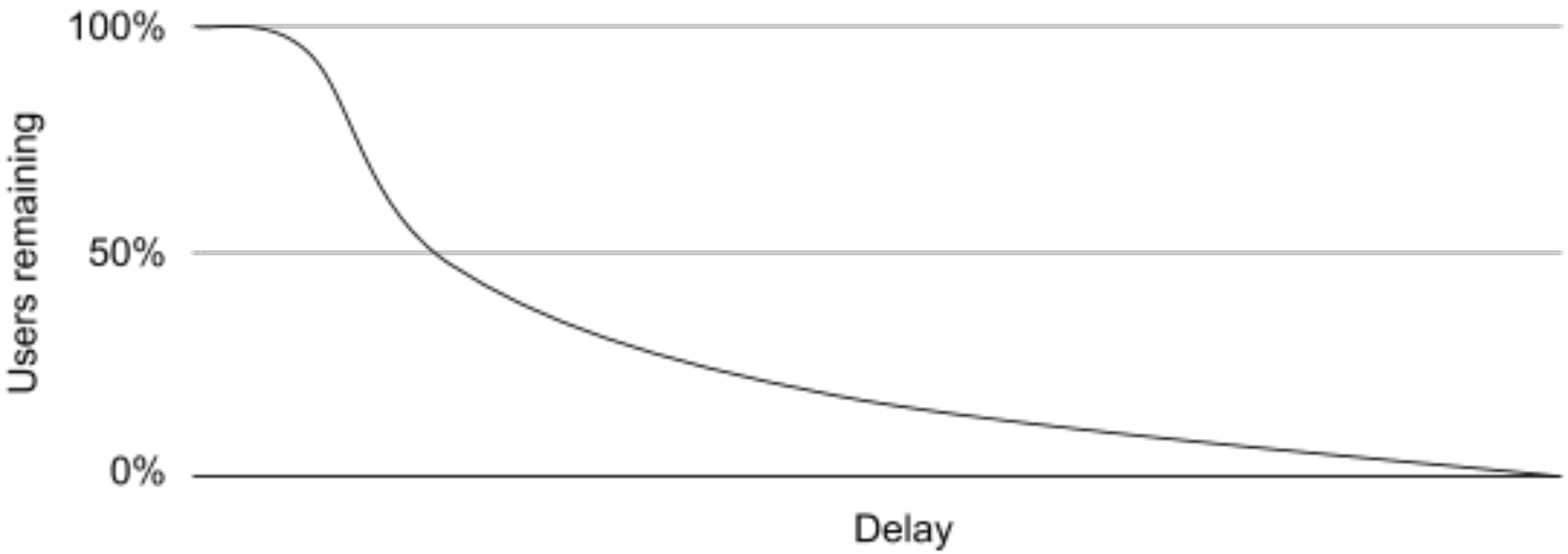

LCD
Largest Contentful Paint
0
2,5s
4s
FID
First Input Delay
0
100ms
300ms
CLS
Cumulative Layout Shift
0
0,1
0,25
Core Web Vitals


LCD
Largest Contentful Paint
0 2,5s 4s
INP
Interaction to next Paint
0 200ms 500ms
CLS
Cumulative Layout Shift
0 0,1 0,25
Future Core Web Vitals

Die Daten


Chrome UX Report
HTTP Archive
Core Web Vitals Technology Report


Rank: ALL
Geo: ALL
jQuery
WordPress
React
Angular
Next.js
SvelteKit
Astro
Origins having good CWV
100%
80%
60%
40%
20%
0%
Jan. 2020
März 2020
Mai 2020
Juli 2020
Sept. 2020
Nov. 2020
Jan. 2021
März 2021
Mai 2021
Juli 2021
Sept. 2021
Nov. 2021
Jan. 2022
März 2022
Mai 2022
Juli 2022
Sept. 2022
Nov. 2022
Jan. 2023
März 2023


100%
50%
0%
Users remaining
Delay

Wer benutzt Astro?











Wird von über 100.000 Entwicklern und Teams auf der ganzen Welt genutzt

Partner













Vergleichbare Frameworks





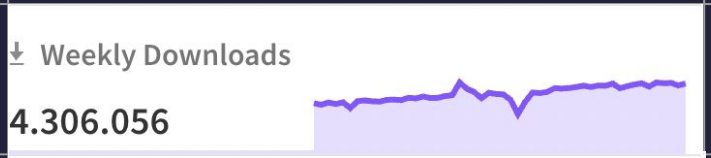
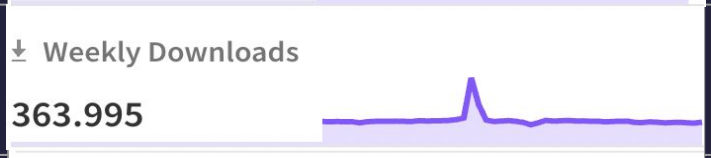
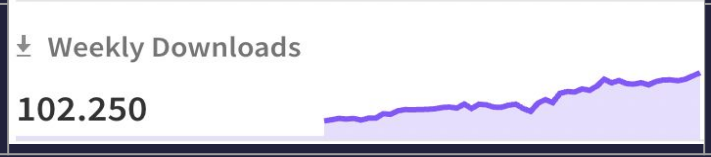


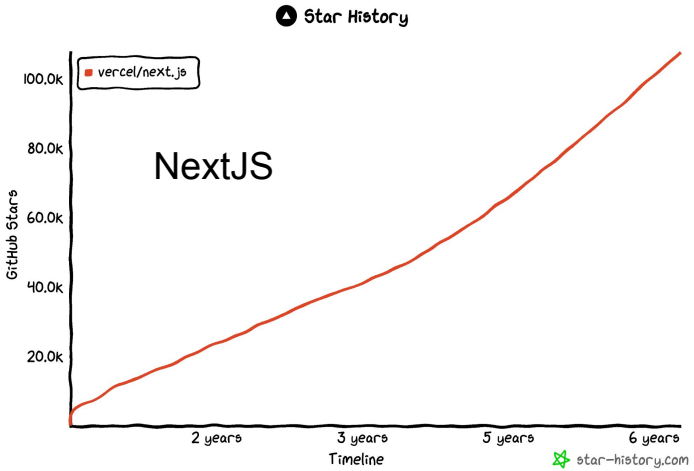


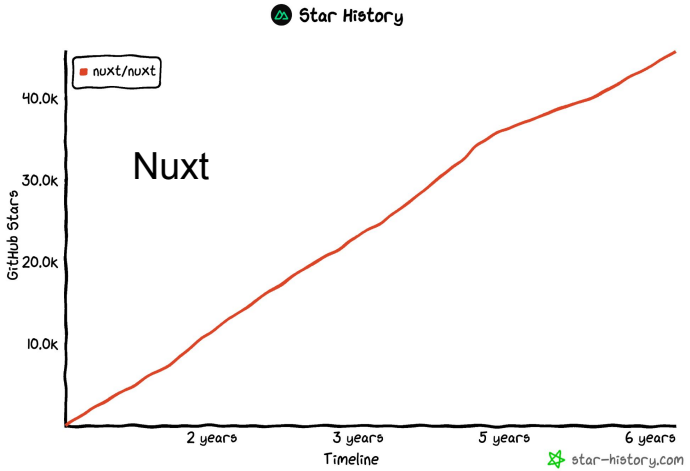


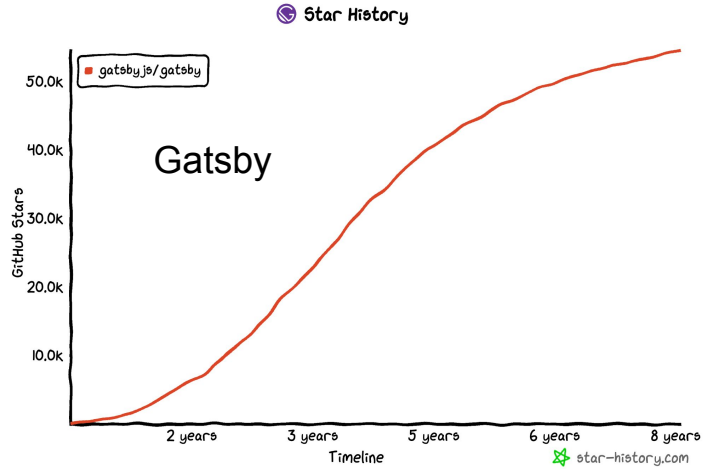


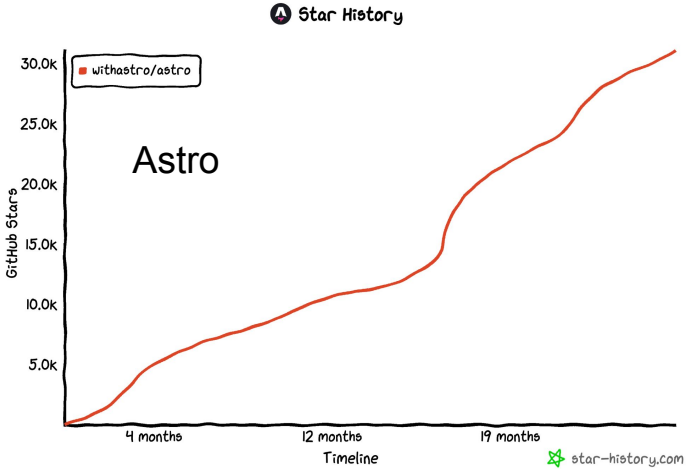
Vergleichbare Frameworks

Framework	NPM Downloads (w)	GitHub-Stars
Next.js		107.000
Nuxt.js		45.600
Gatsby		54.500
Astro		30.700


Star History
vercel/next.js
NextJS
100.0k
80.0k
60.0k
40.0k
20.0k
GitHub Stars
2 years
3 years
5 years
6 years
Timeline
star-history.com

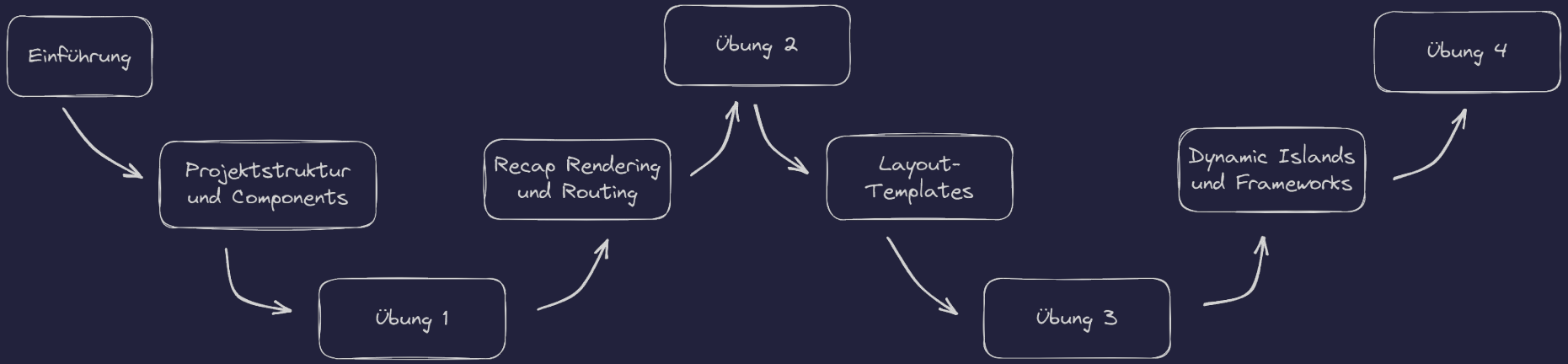

Star History
nuxt/nuxt
Nuxt
40.0k
30.0k
20.0k
10.0k
GitHub Stars
2 years
3 years
5 years
6 years
Timeline
star-history.com


Star History
gatsbyjs/gatsby
Gatsby
50.0k
40.0k
30.0k
20.0k
10.0k
GitHub Stars
2 years
3 years
5 years
6 years
8 years
Timeline
star-history.com


Star History
withastro/astro
Astro
30.0k
25.0k
20.0k
15.0k
10.0k
5.0k
GitHub Stars
4 months
12 months
19 months
Timeline
star-history.com

Roadmap

Was euch im Workshop erwartet


Einführung
Projektstruktur und Components
Übung 1
Recap Rendering und Routing
Übung 2
Layout-Templates
Übung 3
Dynamic Islands und Frameworks
Übung 4

Einführung

- Open-Source Web Framework (The Astro Technology Company)
- Wurde 2021 gelauncht, August 2022 Release der 1.0 Version
- 2.0 Version im Januar 2023, mittlerweile 2.6
- Grundzüge eines Static Site Generators kombiniert mit Ansatz als Meta-Framework
- Komponentenarchitektur, bei der die Webseiten aus wiederverwendbaren Elementen aufgebaut werden

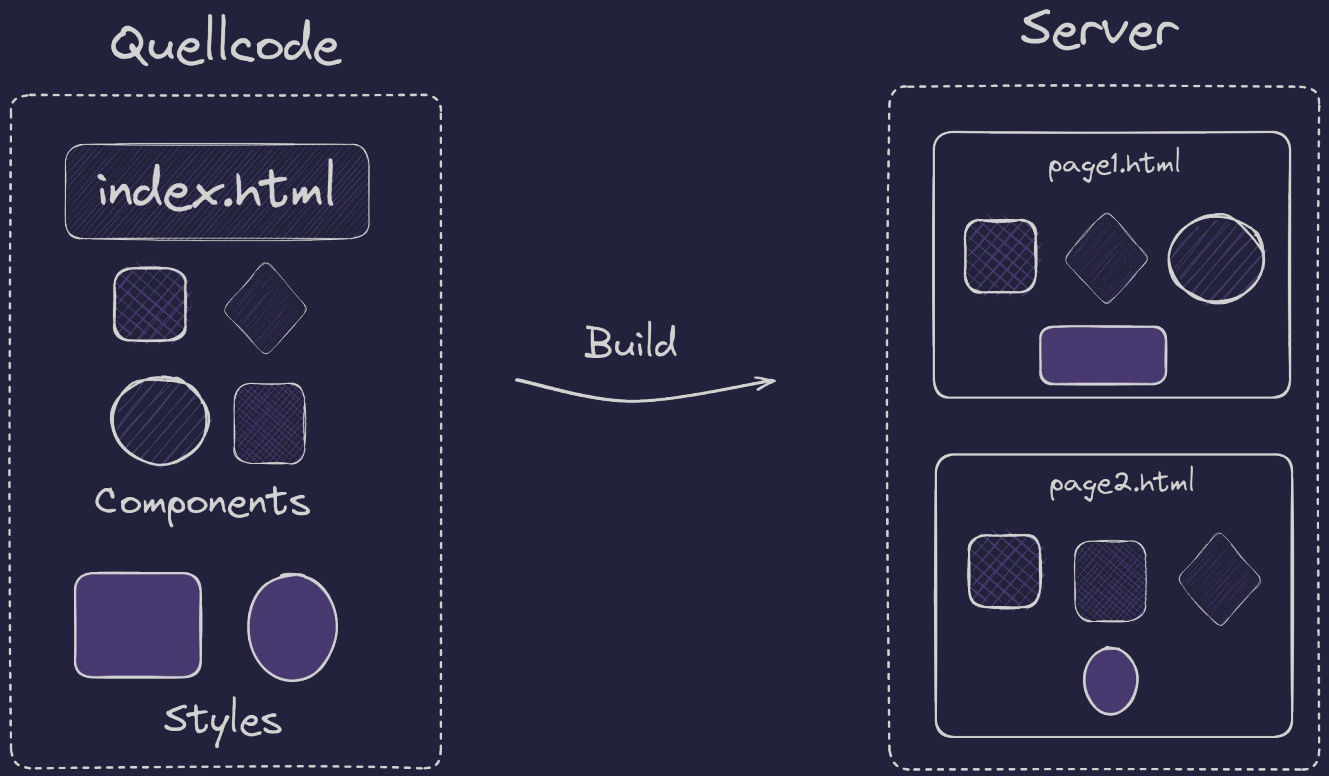
Einführung

- Komponenten aus verschiedenen Frontend-Frameworks wie React, Vue.js, Svelte verwendbar
- Seite wird während dem Build-Prozess in statisches HTML umgewandelt und alles JavaScript entfernt
- Interaktivität wird auf Wunsch ermöglicht und nur die entsprechende Komponente geladen, der Rest bleibt statisch
- Interne Build Engine wird von Vite betrieben

Idee und Motivation

- Web Development der letzten Jahre geprägt durch SPA auf Basis von JavaScript-Frameworks
- *“Too often we see teams blindly accepting the complexity of SPAs by default even when the business needs don’t justify it.”*
- Content-Focused Websites
- Performantere Webseiten durch weniger bzw. kein JavaScript
- Superset von HTML für einfache Syntax
- “*Bring your own Framework*” - Ansatz

Was versteht man unter dem Build-Prozess?



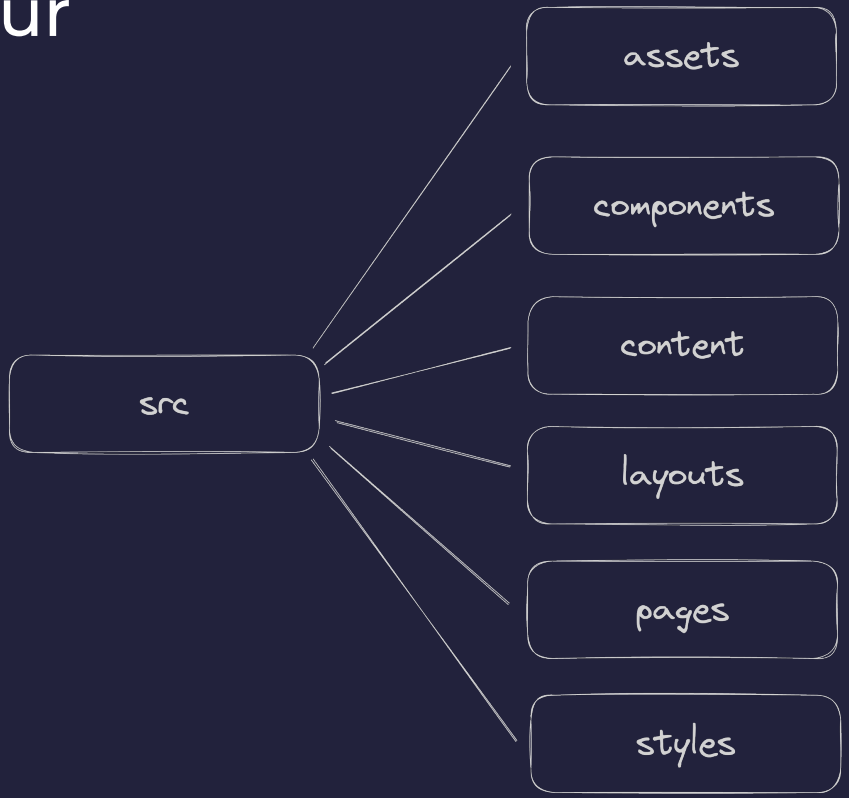
Grundlagen 1

Projektstruktur
Komponentensyntax

Projektstruktur

- Setup via `create astro`
- Vielzahl an Templates
- Vorgegebenes Ordner-Layout
  - `src/*` - Source code (components, pages, styles, etc.)
  - `public/*` - Non-code / Assets (fonts, icons, etc.)
  - `package.json` - Alle Dependencies und möglichen commands
  - `astro.config.mjs` - Hier können Frameworks importiert werden (React etc.)
  - `tsconfig.json` - TypeScript config file (recommended)

Projektstruktur



Astro Component

- `.astro`
- Enthält oft wiederverwendbare UI
- Simple Syntax die jedem vertraut ist
- Frontmatter: `--- //JavaScriptCode ---`
- Wird nicht beim Client gerendert, sondern im Build-Prozess zu HTML kompiliert (bzw. on-demand bei SSR).

Astro Component

Component Structure

```
---
//Komponentenskript (serverseitiges JS)
const title = "Welt"
---
<!-- Komponententepmlate (HTML + JS-Ausdrücke) -->
<h1>Hallo {title}</h1>

```

- Imports von Elementen, Frameworks, fetchen und definierte Variablen
- Code Ausführung

Astro Component Props

- Eine Astro-Komponente kann Props definieren und verwenden
- Diese Props werden dann für das Component Template zum Rendern von HTML verfügbar gemacht
- Props sind in der Variable Astro.props im Frontmatter-Teil verfügbar
- Default-Wert kann angegeben werden

Astro Component Props

```
---
//Abrufen der Properties, "Hello" als Default-Wert
const {greeting = "Hello", myname} = Astro.props
---
<!-- Einsatz der Properties im Template -->
<h1>{greeting}, {myname}!</h1>

```

```
---
import Greetingcard from "../components/Greetingcard.astro";
const name = "Astro"
---
<p>Greet someone here</p>
<!-- Übergeben von Werten an die Astro-Komponente -->
<Greetingcard greeting ="Hi" myname={name}/>
```

Astro Component Props Interface

- Es kann optional ein Interface definiert werden
- Astro erkennt automatisch, dass es sich um die Props für die Komponente handelt
- Ermöglicht Intellisense und Typsicherheit

```
5
6   export interface Props {
7     title: string;
8     description: string;
9     image?: string;
10  }
```

Astro Component vs React Component

```astro
---
const {name} = Astro.props
---

<h1>Hallo {name}</h1>

```

```jsx
export default function MyReactComponent (props) {

    const myname = props.name

    return(
        <div>Hello {myname}</div>
    )
}

```

CSS-Styling

- Styling mit `<style>`-Tags direkt innerhalb der Components/ Pages-Files möglich
- Diese CSS-Regeln sind standardmäßig automatisch *scoped* und gelten nur für HTML, das innerhalb der gleichen Komponente geschrieben wurde

```
7   <h1>Aktuelle Beiträge</h1>
8   <Card />
9
10  <style>
11      /*Wird nur auf H1-Tags der Komponente angewendet*/
12      h1{
13          color: red;
14      }
15  </style>
16
```

CSS-Styling

- Scoped Styles sind empfohlen
- Mit dem Attribut *<style is:global>* kann man das automatische CSS-Scoping deaktivieren
- global()

```
9
10  <style is:global>
11      /*Wird auf alle H1-Tags der Seite angewendet*/
12      h1{
13          color: red;
14      }
15  </style>
16
```

Ein paar hilfreiche Shortcuts (WIP)

Befehl	**macOS**	Windows / Linux
Open File (fuzzy search)	⌘ + P	Ctrl + P
Preview	⇧ + ⌘ + V	⇧ + Ctrl + V
Toggle Console	⌘ + J	Ctrl + J
Toggle Sidebar	⌘ + B	Ctrl + B
Search	⇧ + ⌘ + F	⇧ + Ctrl + F
Explorer	⇧ + ⌘ + E	⇧ + Ctrl + E
Close Active Tab	⌘ + W	Ctrl + W
Reopen last tab(s)	⌘ + ⇧ + T	Ctrl + ⇧ + T
Switch tabs (left/right)	⌘ + Option + LEFT/RIGHT	Ctrl + Alt + Arrow keys

Ein paar Worte noch zum Repo



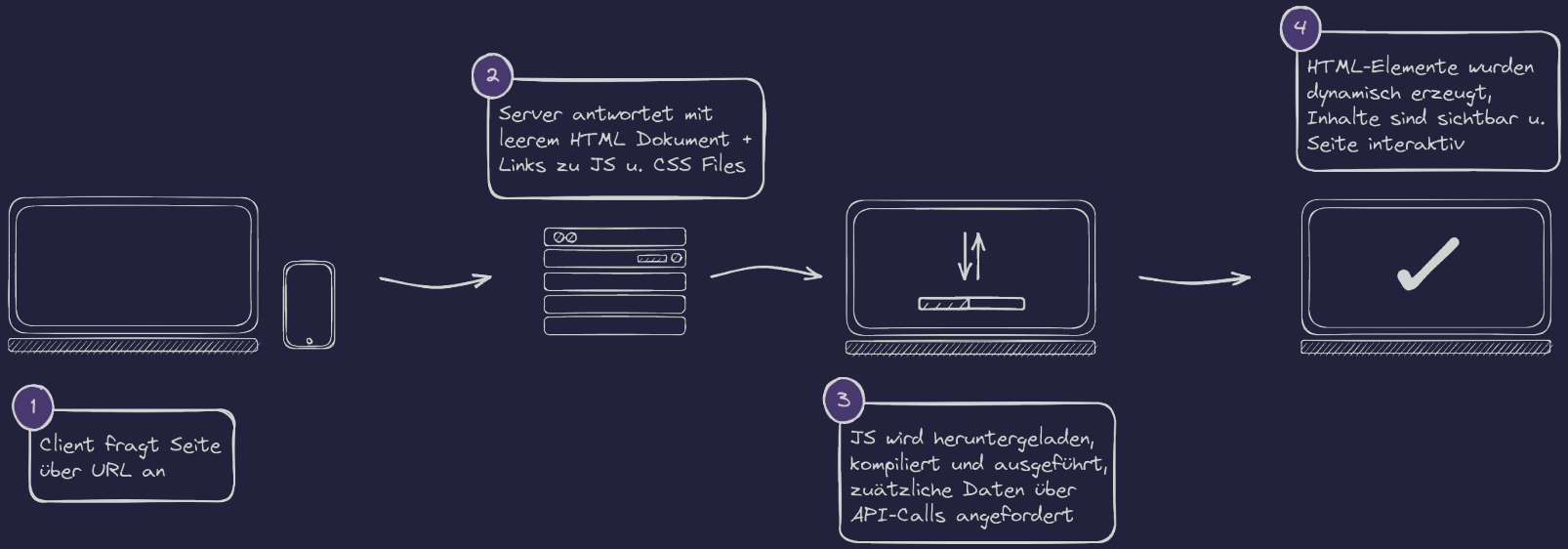
Übung 1

github.com/BingeCode/workshop-hdm-astro

Renderinganssätze

Wiederholung des letzten Workshops

Client Side Rendering



Server Side Rendering



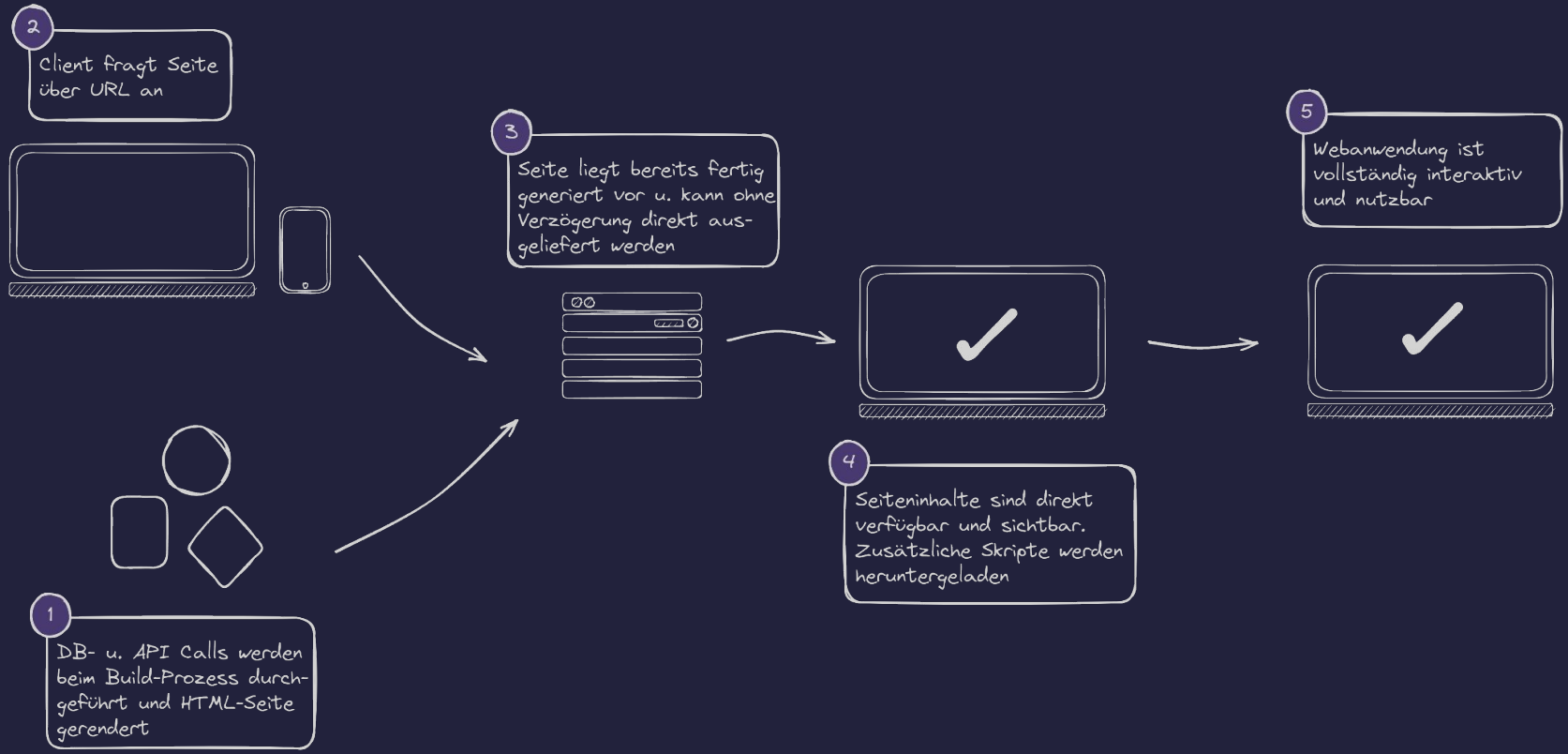
1 Client fragt Seite über URL an

2 Server führt API u. DB-Aufrufe aus, verarbeitet die Daten u. generiert HTML

3 Seiteninhalte sind direkt verfügbar und sichtbar. Zusätzliche Skripte werden heruntergeladen

4 Webanwendung ist vollständig interaktiv und nutzbar

Static Site Generation



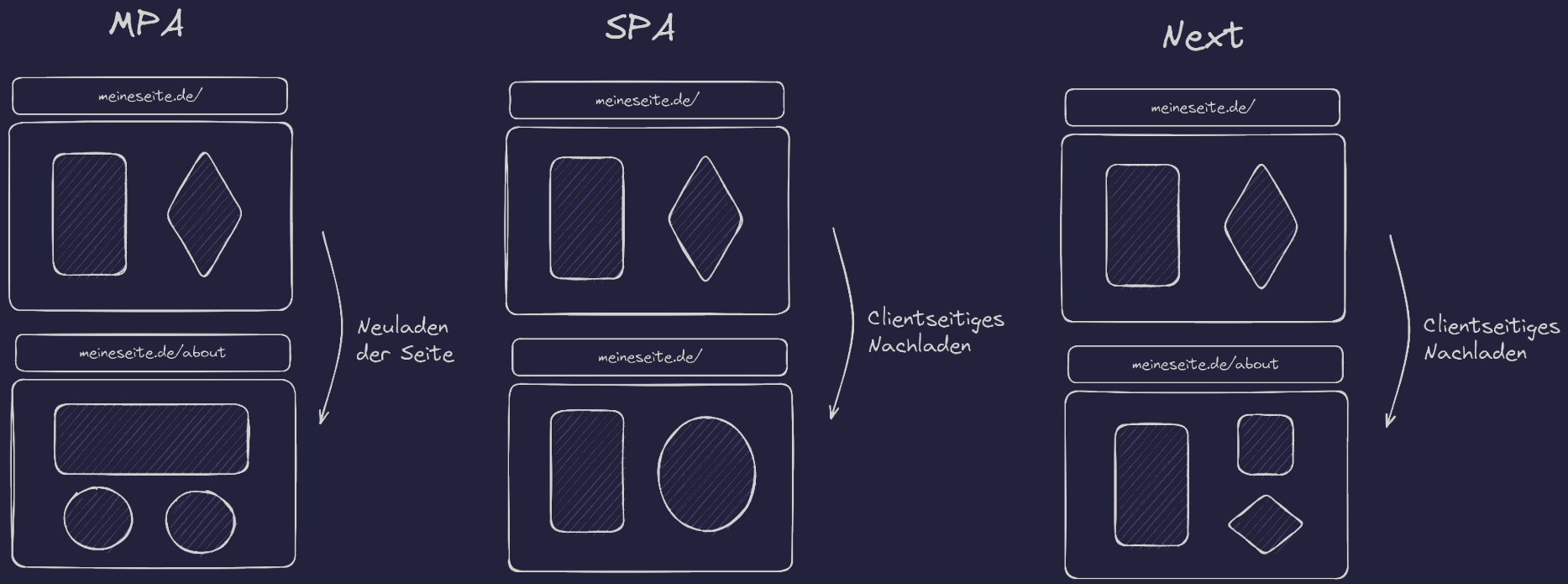
SPA vs MPA

Exkurs

Single Page vs. Multi Page Application

- SPA besteht aus einer HTML-Seite, typischerweise clientseitig gerendert
- Inhalte werden dynamisch nachgeladen und aktualisiert, ohne dass die Seite neu geladen werden muss, keine URL-Wechsel
- MPA besteht aus mehreren HTML-Seiten, die separat geladen werden und meist auf Server gerendert (o. statisch generiert) werden
- Wenn Nutzer zu einer neuen Seite navigiert, wechselt URL und Browser fordert neue HTML-Seite vom Server an

Single Page vs. Multi Page Application

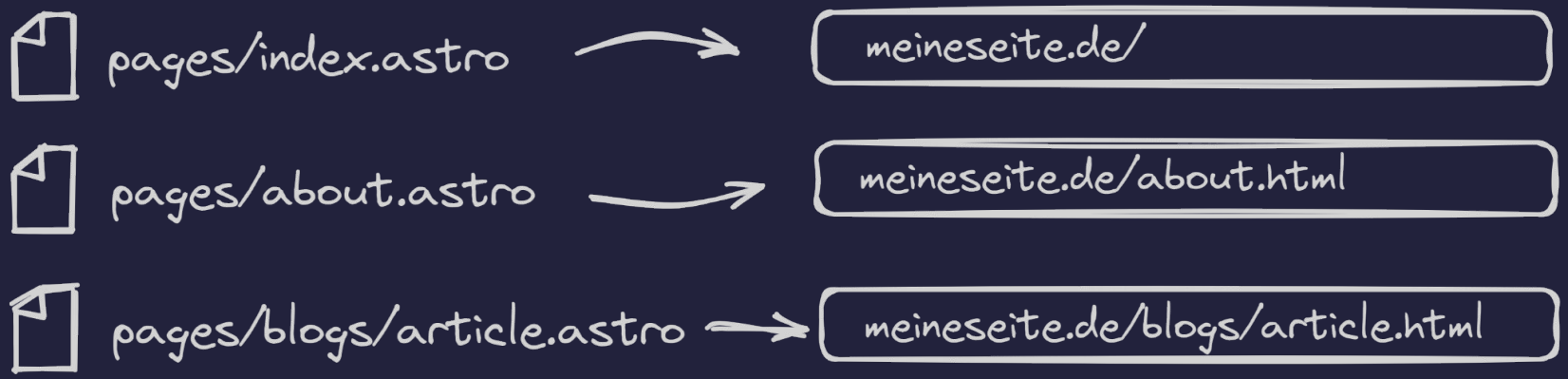


Grundlagen 2

Routing

Basics of Routing in Astro

- Astro-Komponenten (.astro) und Markdown-Dateien (.md) im Verzeichnis `src/pages` werden automatisch zu Seiten auf deiner Website. Die Route jeder Seite entspricht ihrem Pfad und Dateinamen im Verzeichnis `src/pages`.



Basics of Routing in Astro

- Astro verwendet standardmäßige `<a>`-HTML-Elemente, um zwischen einzelnen Routen zu navigieren
- Es gibt keine Framework-spezifische `<Link>`-Komponente

```
4
5   <nav>
6     <ul>
7       <li><a href="index.html">Home</a></li>
8       <li><a href="about.html">About</a></li>
9       <li><a href="blog.html">Blog</a></li>
10    </ul>
11  </nav>
12
```

Übung 2

Grundlagen 3

Layouts

Templating mit Astro Layouts

- Layouts sind besondere Astro-Komponenten, die nützlich für die Erstellung wiederverwendbarer Seitenvorlagen sind.
- In die Layout-Komponente werden üblicherweise die Header und Footer Komponenten integriert. Diese dienen als Seiten-Gerüst.
- Dadurch wird der Code übersichtlicher und leichter zu pflegen.
- Allerdings muss dennoch die ganze Seite neu geladen werden. SPA-Anwendungen können hingegen differential updates durchführen

Astro Layout Beispiel

```astro
---
import Header from "../components/Header.astro";
import Footer from "../components/Footer.astro"
---
<!--Header, Footer und HTML-Head werden auf jeder Seite
eingebunden, auf die das Layout angewendet wird-->
<!DOCTYPE html>
<html lang="en">
    <head>
        <link rel="icon" type="image/svg+xml" href="/favicon.svg" />
        <meta name="generator" content="Mein Blog" />
        <title>Mein Blog</title>
    </head>
    <body>
        <Header/>
        <!--Inhalte werden automatisch in Slot Komponente eingefügt,
        diese dient im Layout als Platzhalter -->
        <slot />
        <Footer/>
    </body>
</html>
```

<slot/>-Element

- Das `<slot/>`-Element ist ein Platzhalter für externe HTML-Inhalte, der es dir ermöglicht, untergeordnete Elemente aus anderen Dateien in deine Komponentenvorlage einzubinden.
- In unserem Beispiel ist: `<p>Mein Seiteninhalt… </p>` der Platzhalter von `<slot/>`

```
1   ---
2   import MeinLayout from "../layouts/MeinLayout.astro";
3   ---
4
5   <MeinLayout>
6       <!--Inhalte innerhalb der Layout-Komponente
7       werden automatisch in de Slot eingefügt-->
8       <h1>Mein Seitenihlat umgeben von einem Layout</h1>
9   </MeinLayout>
10
```

Übung 3

Grundlagen 4

Island-Architektur
Progressive Hydration
Externe Frameworks

Zero JavaScript by default

- Standardmäßig wird jede Astro-Seite ohne JavaScript ausgeliefert
- Sorgt für optimale Performance beim Laden
- Interaktivität bestimmter Komponenten würde fehlen

Exkurs: Was heißt Hydrierung?

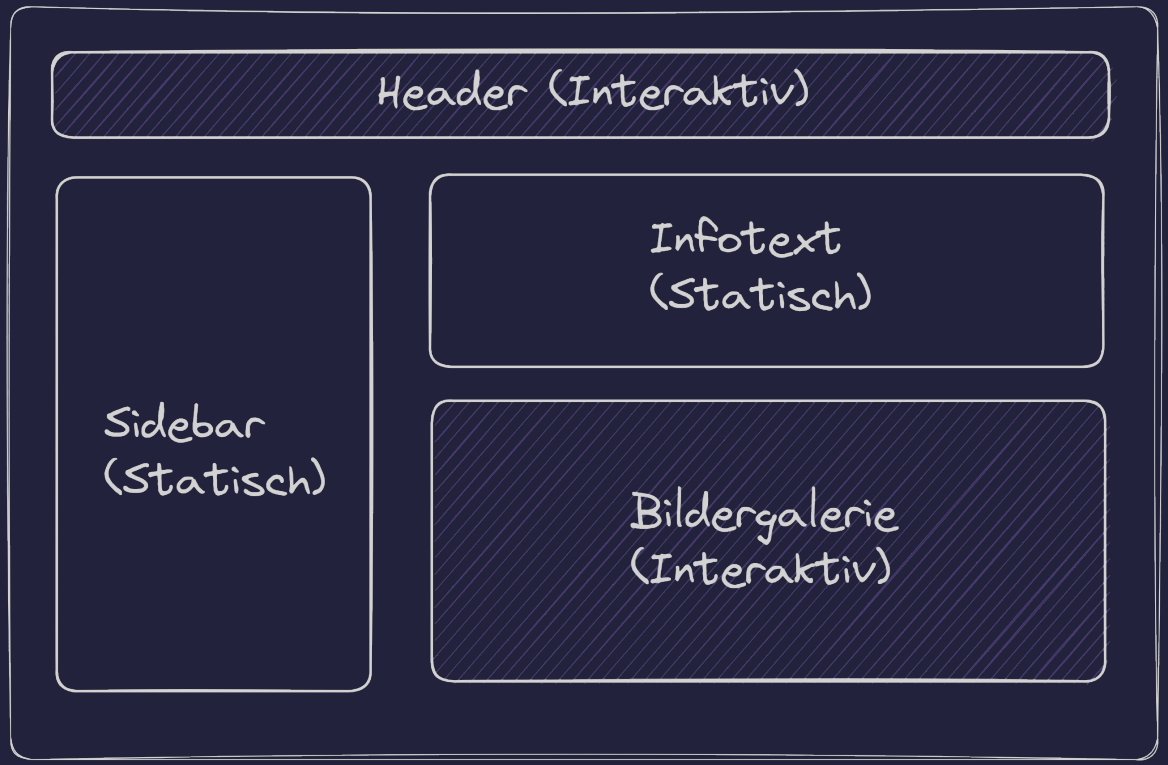


- Statische HTML-Seite wird von Server gesendet
- Zusätzliches JS-Bundle auf Client-Seite heruntergeladen und ausgeführt
- Elemente werden “hydriert” und sind wieder interaktiv

Astro Islands

- Astro setzt auf “*Dynamic Island*” - Architektur mit progressive/partial Hydration
- Idee dahinter ist, dass die Seite ihre einzelnen Bereiche nach und nach, also fortschreitend (progressive) hydriert
- Manche Bereiche der Seite z.B statische Komponenten müssen gar nicht hydriert werden (partial)
- Astro-Seiten sind aus interaktiven Inseln, die in statische Content-Bereiche eingebettet sind, aufgebaut

Astro Islands



Clientseitiges JavaScript

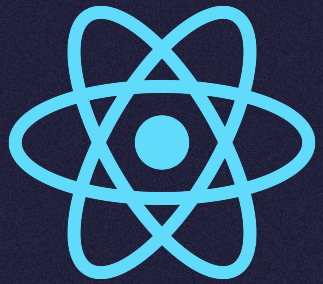
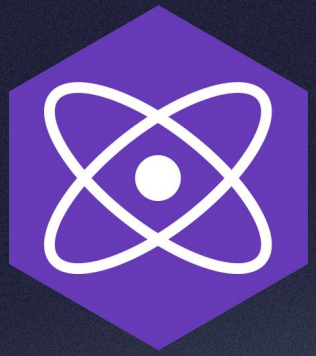
- Der Code innerhalb des Frontmatters (`---`) wird immer nur serverseitig ausgeführt
- Auf diese Weise keine Interaktivität möglich!
- Interaktivität z.B. EventListener trotzdem realisierbar
- `<script>`-Tag in deiner Astro-Komponentenvorlage verwenden und JavaScript an den Browser senden, das dort ausgeführt wird

Clientseitiges JavaScript

```
7   <h1>Aktuelle Beiträge</h1>
8   <Card />
9
10  <script>
11      //Clientseitiges JavaScript im Browser augeführt
12      //Gebundelt und TypeScript-Support
13      console.log("Ich benötige interaktive Skripts")
14  </script>
15
```

```
10
11  <!--Clientseitiges Script über relativen Pfad einbinden-->
12  <script src="../scripts/local.js"></script>
13
```

Integration von UI-Frameworks






React Components

- Setup-Helper via `npx astro add react` aufrufen
- Dependencies werden installiert
- `astro.config` wird angepasst.

```
1  import { defineConfig } from 'astro/config';
2
3  import react from "@astrojs/react";
4
5  // https://astro.build/config
6  export default defineConfig({
7    integrations: [react()]
8  });
```

React Components

- Alle Funktionalitäten wie State-Hooks können genutzt werden

```jsx
import { useState } from "react";

export default function Showcase () {

    useState [myname, setName] = useState("Brad")

    return(
        <div>Hello {myname}</div>
    )
}

```

```astro
---
import Showcase from "../components/Showcase.jsx";
import Layout from "../layouts/Layout.astro";
---
<Layout title="Homepage">
    <Showcase/>
</Layout>

```

Directives

Direktive	Funktion
client:load	Die Komponente wird sofort beim Laden der Seite hydriert
client:idle	Die Komponente wird hydriert, wenn der Browser freie Ressourcen hat. Unterstützt der Browser dieses Verhalten nicht, wird die Komponente beim Laden der Seite hydriert.
client:visible	Die Komponente wird hydriert, wenn sie in den sichtbaren Bereich gescrollt wird.
client:media	Die Komponente wird abhängig von einer CSS Media Query hydriert, z. B. abhängig von der Fenstergröße.
client:only	Die Komponente wird nicht auf dem Server gerendert, sondern nur clientseitig. Das Hydrieren erfolgt wie client:load

Directives

```
---
import Showcase from "../components/Showcase.jsx"
import ReactButton from "../components/Reactbutton.jsx"
---

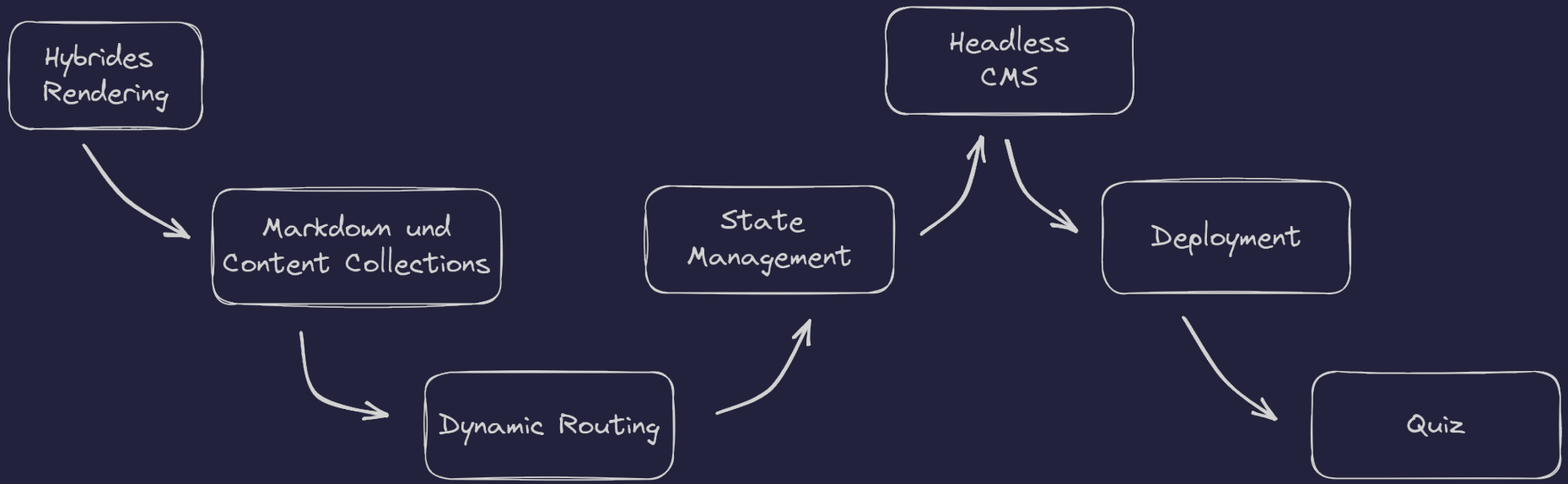
<h1>Über uns</h1>
<!--Komponente wird direkt beim Laden interaktiv -->
<Showcase client:load/>
<!--Komponente wird erst interaktiv, wenn Nutzer
    in sichtbaren Bereich scrollt -->
<ReactButton client:visible/>

```

Übung 4

Ausblick

Was euch nächste Woche erwartet


Hybrides Rendering
Markdown und Content Collections
Dynamic Routing
State Management
Headless CMS
Deployment
Quiz



Workshop - Part 2

Noch alles in Erinnerung?

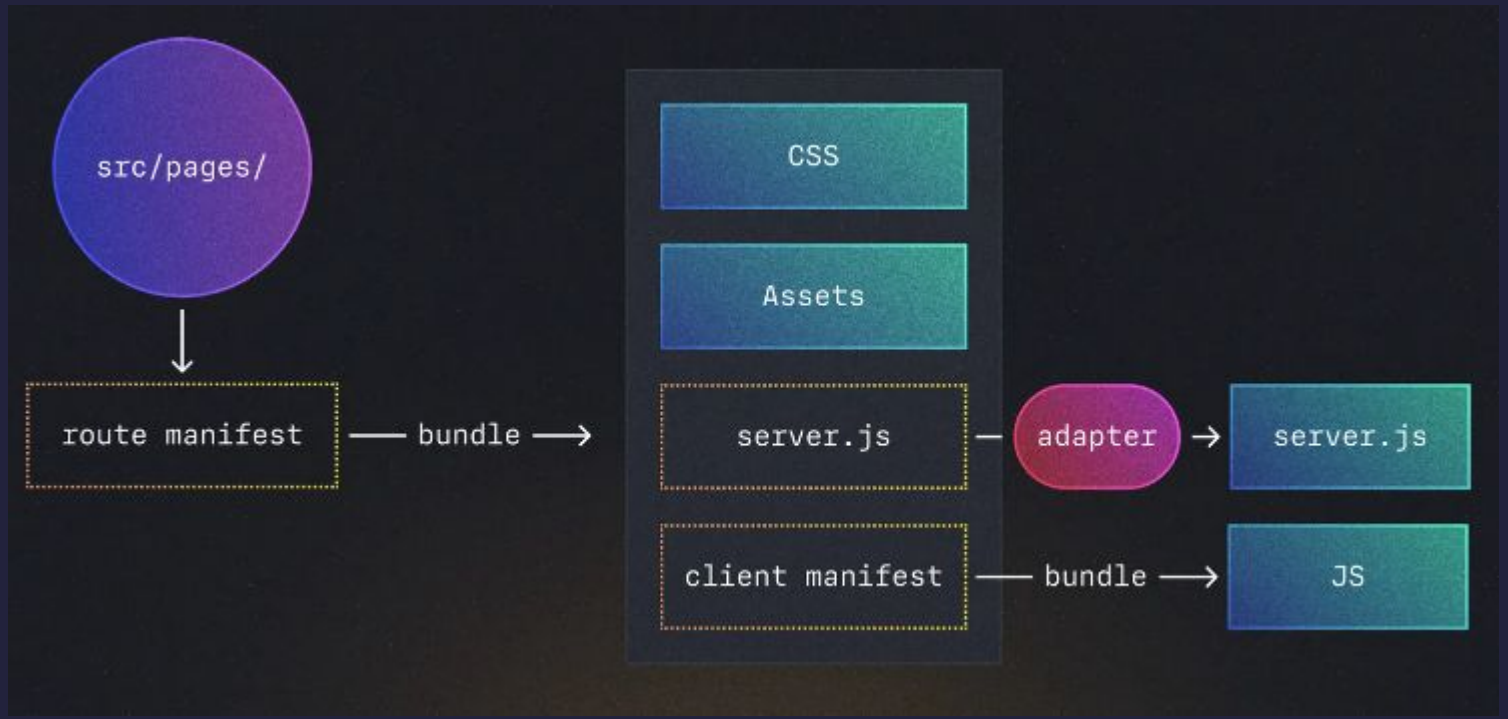


Astro build prozess und rendering

- Vor Astro 2.0 musste bei jedem Build Prozess eine Entscheidung getroffen werden.
- SSR oder SSG?
- output: ‘static’ verwerft das serverseitige JavaScript
- output: ‘server’ serverseitiges JS wird an einen Adapter weitergegeben

```
1  import { defineConfig } from "astro/config";
2  import meinAdapter from "@astrojs/meinAdapter"
3
4  // https://astro.build/config
5  export default defineConfig({
6    output: 'server',
7    adapter: meinAdapter()
8  });
```

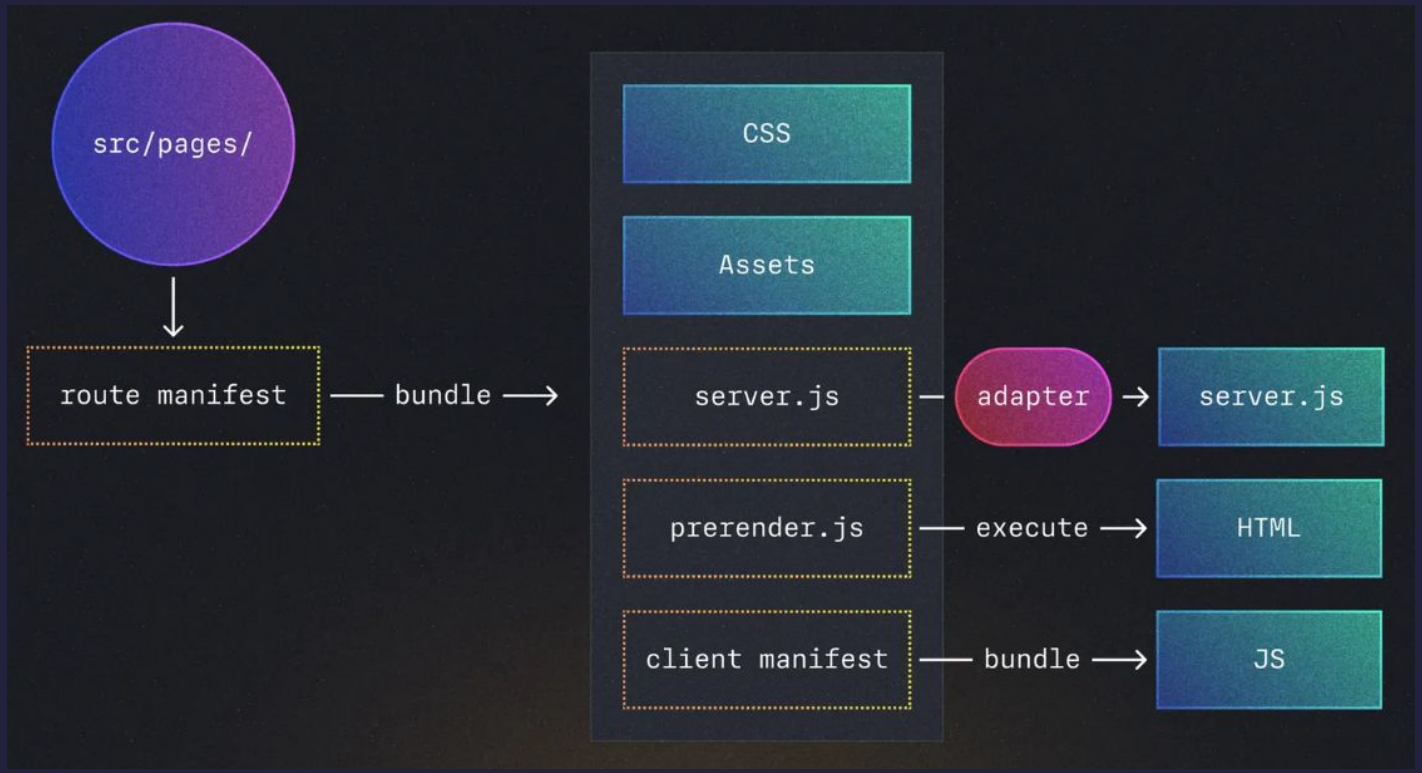
Astro build prozess und rendering



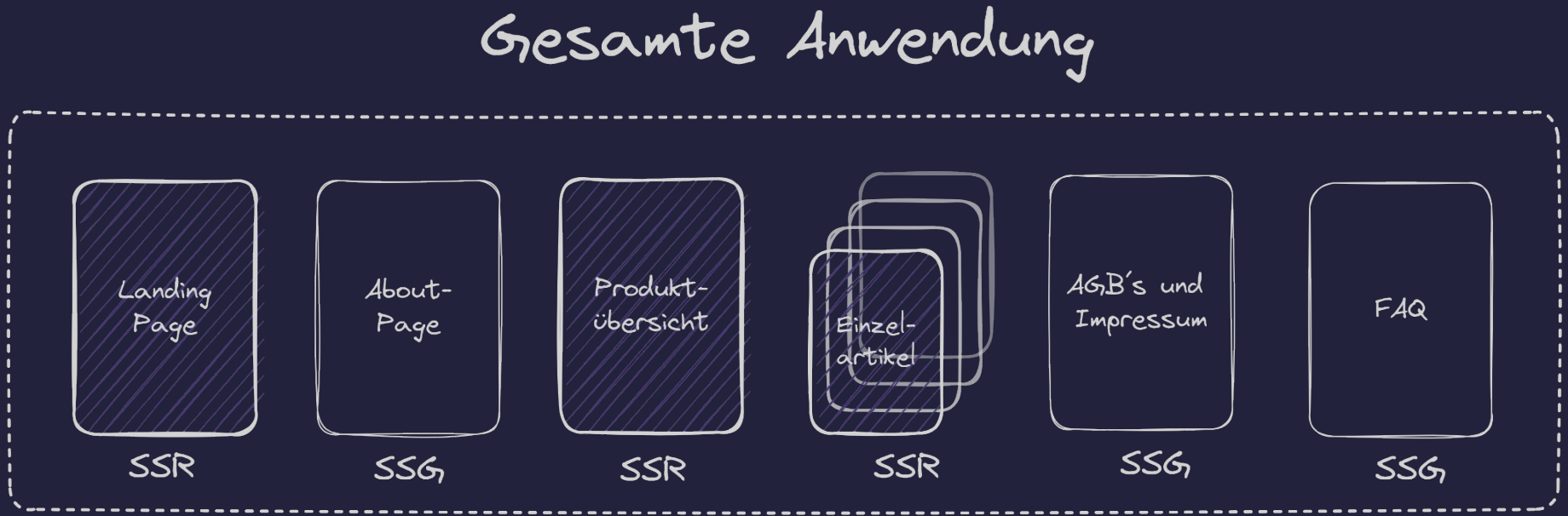
Server Side & Hybrides Rendering

- Astro 2.0 führte hybrides Rendering ein.
- Entwickler entscheiden, welche Seiten beim Build statisch erzeugt werden und welche erst auf Client-Anfrage serverseitig gerendert werden.
- Dies ermöglicht es uns die Routen aufzuteilen, je nachdem, wann und wo sie gerendert werden sollen

Hybrides Rendering



Hybrides Rendering



Server Side & Hybrides Rendering

- Um SSR zu aktivieren, muss Adapter verwendet werden
- Durch die Installation des Adapters erhält Astro Zugriff auf die entsprechende API und kann ein Skript ausgeben, das dein Projekt auf diesem Server ausführt
- Für Hybrides Rendering muss Astro in der Lage sein beim Build zu erkennen, welche Seiten direkt statisch gerendert werden müssen und welche nicht

Hybrides Rendering

- Default SSR und bei Bedarf SSG

```
import { defineConfig } from 'astro/config';

export default defineConfig({
  output: 'server'
});

```

```
---
export const prerender = true
---
<html>
    <!--Static HTML -->
</html>

```

- Default SSG und bei Bedarf SSR

```
import { defineConfig } from 'astro/config';

export default defineConfig({
  output: 'hybrid'
});

```

```
---
export const prerender = false
---
<html>
    <!--Server Rendered HTML -->
</html>

```



Übung 5

github.com/BingeCode/workshop-hdm-astro

Grundlagen 6

Markdown

Markdown

- 2004 entwickelt von John Gruber und Aaron Swartz
- “[...] to be used as a format for **writing for the web**“ (John Gruber)
  “Easy to read, easy to write”
- Alternative zu HTML und RTF
- Deutlich weniger Formatierungsanweisungen als Rich Text Format (RTF) oder HTML.
- Im Kontrast zu WYSIWYG Editoren wie Word, Pages, etc.

Output


My Simple Web Article
A Quick Guide
This is a simple web article written to demonstrate different formatting options available when writing web articles. Please follow this link to learn more.
Click here

RTF 🤔

```
{\rtf1\ansi\deff0
{\header My Simple Web Article \line A Quick Guide}
This is a simple web article written to demonstrate different
formatting options available when writing web articles.
Please follow this link to learn more. \line
{\field{\*\fldinst{HYPERLINK "http://www.example.com"}}{\fldrslt Click here}}
}
```

HTML 😕

```html
<!DOCTYPE html>
<html>
  <head>
    <title>My Simple Web Article</title>
  </head>
  <body>
    <h1>My Simple Web Article</h1>
    <h2>A Quick Guide</h2>
    <p>
      This is a simple web article written to demonstrate different formatting
      options available when writing web articles. Please follow this link to
      learn more.
    </p>
    <a href="http://www.example.com">Click here</a>
  </body>
</html>
```

Markdown 😌

```
# My Simple Web Article

## A Quick Guide

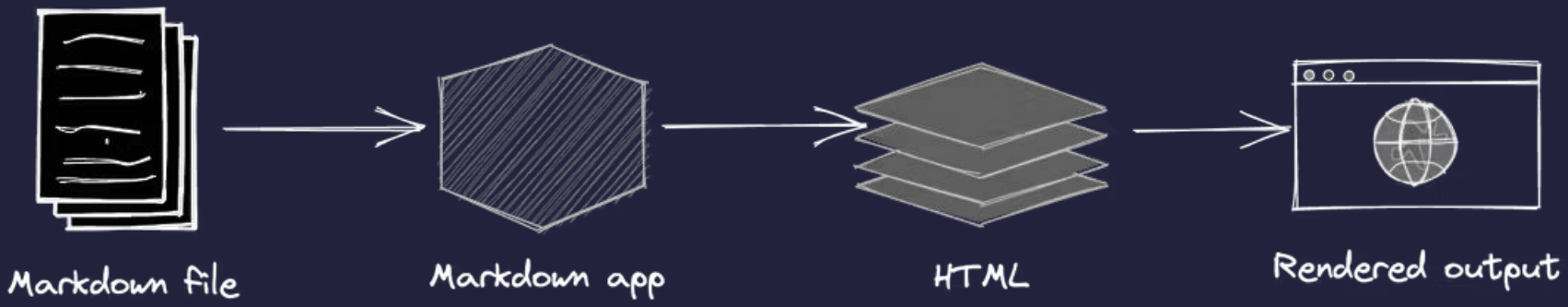
This is a simple web article written to demonstrate different
formatting options available when writing web articles.
Please follow this link to learn more.

[Click here](http://www.example.com)
```

Warum Markdown?

- Jeder kann Markdown schreiben
- Einfach erweiterbar mit Plugins
- Man kann fast alles damit schreiben
    - Blogs, Webseiten, Bücher, Präsentationen, Emails, Notizen
- Plattformunabhängig und zukunftssicher, immer lesbar

Wie funktioniert Markdown?



Markdown

- Viele Dialekte (Flavors) von Markdown
- Uneindeutiges Verhalten zwischen Flavors
- Seit 2014 eindeutige CommonMark Spezifikation
- Github Flavored Markdown seit 2017 (Superset von CommonMark)
    - Task-Listen, Durchstreichen, Tabellen, etc.
- Markdown wird von vielen Anwendungen genutzt (Github, Slack, StackOverflow, Microsoft Teams, Discord, Reddit, etc.)
    - Allerdings unterschiedliche Flavors

Astro und Markdown

- Astro unterstützt Markdown out-of-the-box
- `.md` Files sitzen normalerweise in `src/content/`
  - Später: Content Collections
- `.md` Files in `src/pages` werden automatisch zu Seiten konvertiert



Astro und Markdown

- Zusätzlich kann Frontmatter YAML verwendet werden (Metadaten)
- Kein JS, nur YAML
- `draft`: true verhindert das automatische Rendern bei Build

```
---
layout: ../../layouts/BaseLayout.astro
title: My Blog Post
draft: true
---

This is my in-progress blog post.

No page will be built for this post.

To build and publish this post:

- update the frontmatter to `draft: false` or
- remove the `draft` property entirely.

But, this page _will_ be returned by any matching `Astro.glob()` request.

```

Frontmatter

- Das zu verwendende Layout des MD-Artikels über `layout` definieren
- Layout Komponente erhält dann Zugriff auf die Frontmatter Props des `.md` Files

```
---
layout: ../../layouts/BlogPostLayout.astro
title: Astro in brief
author: Himanshu
description: Find out what makes Astro awesome!
---
This is a post written in Markdown.
```

```
---
const { frontmatter } = Astro.props;
---

<html>
  <!-- ... -->
  <h1>{frontmatter.title}</h1>
  <h2>Post author: {frontmatter.author}</h2>
  <p>{frontmatter.description}</p>
  <slot />
  <!-- Markdown content is injected here -->
  <!-- ... -->

</html>
```

Weitere Markdown Layout Props

```
const props = {
  file: '/home/user/projects/.../file.md',
  url: '/en/guides/markdown-content/',
  frontmatter: {
    /** Frontmatter from a blog post */
    title: 'Astro 0.18 Release',
    date: 'Tuesday, July 27 2021',
    author: 'Matthew Phillips',
    description: 'Astro 0.18 is our biggest release since Astro launch.',
    /** Generated values */
    file: '/home/user/projects/.../file.md',
    url: '/en/guides/markdown-content/',
  },
  headings: [
    /* ... */
  ],

  /** Available in Markdown only */
  rawContent: () =>
    '# Astro 0.18 Release\nA little over a month ago, the first public beta [...]',
  compiledContent: () =>
    '<h1>Astro 0.18 Release</h1>\n<p>A little over a month ago, the first public beta [...]</p>',
};

console.log(props);
```

Kleine Besonderheit: Heading IDs

- Astro konvertiert Headings automatisch zu Headings mit ID
- Ermöglicht interne Verlinkung

```
---
title: My page of content
---

## Introduction

I can link internally to [my conclusion](#conclusion) on the same page when writing Markdown.

## Conclusion

I can use the URL `https://my-domain.com/page-1/#introduction` to navigate directly to my Introduction on the page.

```

Übung 6


MDX
Markdown
JSX

MDX: Markdown extended

- Oder auch MD + JSX = MDX
- “Markdown for the component era”
- Unterstützt JSX variables, JS expressions und components

```
---
layout: ../layouts/BaseLayout.astro
title: About me
---

import Button from '../components/Button.astro'
import ReactCounter from '../components/ReactCounter.jsx'

I live on **Mars** but feel free to <Button title="Contact me" />.

PI equals {Math.PI}.

<ReactCounter client:load />
```

MDX: Markdown Syntax mit Komponente überschreiben

- Mittels eigener Komponenten kann der Default Style von Markdown “Elementen” überschrieben werden
- Zum Beispiel das `blockquote` HTML Element (>)

```
import Blockquote from '../components/Blockquote.astro';
export const components = {blockquote: Blockquote}

> This quote will be a custom Blockquote
```

```
---
const props = Astro.props;
---

<blockquote {...props} class="bg-blue-50 p-4">
  <span class="text-4xl text-blue-600 mb-2">“</span>
  <slot />
  <!-- Be sure to add a `<slot/>` for child content! -->
</blockquote>
```

MDX: Markdown Syntax mit Komponente(n) überschreiben

- Das gleiche kann man auch in einer Astro-Komponente machen
- Es können gleich mehrere Custom Components definiert werden
- Auch die Custom Components aus dem `.mdx` File werden übernommen

```
---
import { Content, components} from '../content.mdx';
import Heading from './Heading.astro';
---
<!-- Creates a custom <h1> for the # syntax,
_and_ applies any custom components defined in `content.mdx` -->
<Content components={{...components, h1: Heading }} />

```

Markdown Extensions

- Astro nutzt remarkjs unter der Haube als Markdown Prozessor
- remark verfügt über ein weitreichendes Plugin-Ökosystem (150+)
- Beispiele:
  - Inhaltsverzeichnis automatisch generieren
  - Emojis accessible machen
  - Support für GFM
  - Support für MDX
  - Support für Frontmatter
  - etc.
- Astro benutzt GMN standardmäßig (Frontmatter inkludiert)

Übung 7

Grundlagen 7

Content Collections

Content Collections

- Hinzugefügt in Astro 2.0
- Jeder Ordner in src/content/ ist eine Content Collection
- Z.B.: src/content/newsletter und src/content/authors usw.
- Die Einträge (Content Entries) können sogenannte authoring formats wie .md und .mdx oder data formats wie .yaml und .json sein
- data (authors) und authoring (newsletter) formats sollten ihre eigenen Ordner haben



Collection Schema

- Collection Schemas erzwingen konsistentes Frontmatter innerhalb einer Collection
- Wenn ein Collection Entry gegen das Schema verstößt, verweist Astro mit einem Fehler darauf
- -> Mehr Typsicherheit und Predictability

Defining a Collection

```
// 1. Import utilities from `astro:content`
import { z, defineCollection } from 'astro:content';

// 2. Define a `type` and `schema` for each collection
const blogCollection = defineCollection({
  type: 'content', // v2.5.0 and later
  schema: z.object({
    title: z.string(),
    tags: z.array(z.string()),
    image: z.string().optional(),
  }),
});

// 3. Export a single `collections` object to register your collection(s)
export const collections = {
  'blog': blogCollection,
};
```

Defining multiple Collections

```ts
import { z, defineCollection } from 'astro:content';

const blogCollection = defineCollection({
  type: 'content',
  schema: z.object({ /* ... */ })
});
const newsletter = defineCollection({
  type: 'content',
  schema: z.object({ /* ... */ })
});
const authors = defineCollection({
  type: 'data',
  schema: z.object({ /* ... */ })
});

export const collections = {
  'blog': blogCollection,
  'newsletter': newsletter,
  'authors': authors,
};
```

```ts
// Example: A cheatsheet of many common Zod datatypes
import { z, defineCollection } from 'astro:content';

defineCollection({
  schema: z.object({
    isDraft: z.boolean(),
    title: z.string(),
    sortOrder: z.number(),
    image: z.object({
      src: z.string(),
      alt: z.string(),
    }),
    author: z.string().default('Anonymous'),
    language: z.enum(['en', 'es']),
    tags: z.array(z.string()),
    // An optional frontmatter property. Very common!
    footnote: z.string().optional(),
    // In frontmatter, dates written without quotes around them are interpreted as Date
    publishDate: z.date(),
    // You can also transform a date string (e.g. "2022-07-08") to a Date object
    // publishDate: z.string().transform((str) => new Date(str)),
    // Advanced: Validate that the string is also an email
    authorContact: z.string().email(),
    // Advanced: Validate that the string is also a URL
    canonicalURL: z.string().url(),
  })
})
```

Defining a Schema

Referencing a Schema

```ts
import { defineCollection, reference, z } from 'astro:content';

const blog = defineCollection({
  type: 'content',
  schema: z.object({
    title: z.string(),
    // Reference a single author from the `authors` collection by `id`
    author: reference('authors'),
    // Reference an array of related posts from the `blog` collection by `slug`
    relatedPosts: z.array(reference('blog')),
  })
});

const authors = defineCollection({
  type: 'data',
  schema: z.object({
    name: z.string(),
    portfolio: z.string().url(),
  })
});

export const collections = { blog, authors };
```

Referencing a Schema

```
---
title: "Welcome to my blog"
author: ben-holmes # references `src/content/authors/ben-holmes.json`
relatedPosts:
- about-me # references `src/content/blog/about-me.md`
- my-year-in-review # references `src/content/blog/my-year-in-review.md`
---
```

Getting a collection

```
---
import { getCollection, getEntry } from 'astro:content';

// Get all entries from a collection.
// Requires the name of the collection as an argument.
// Example: retrieve `src/content/blog/**`
const allBlogPosts = await getCollection('blog');

// Get a single entry from a collection.
// Requires the name of the collection and either
// the entry `slug` (content collections) or `id` (data collections)
// Example: retrieve `src/content/blog/my-first-post.md`
const graceHopperProfile = await getEntry('blog', 'my-first-post');
---
```

Link Komponente

- Astro verwendet standardmäßige *<a>-HTML-Elemente*, um zwischen einzelnen Routen zu navigieren.
- Es gibt **noch** keine Framework-spezifische *<Link>*-Komponente.
- Diese kann aber eigenständig aufgebaut werden. Vergleich mit Next.js

```
<Link href="/about/" style="secondary" text="👋 About Me" />
```

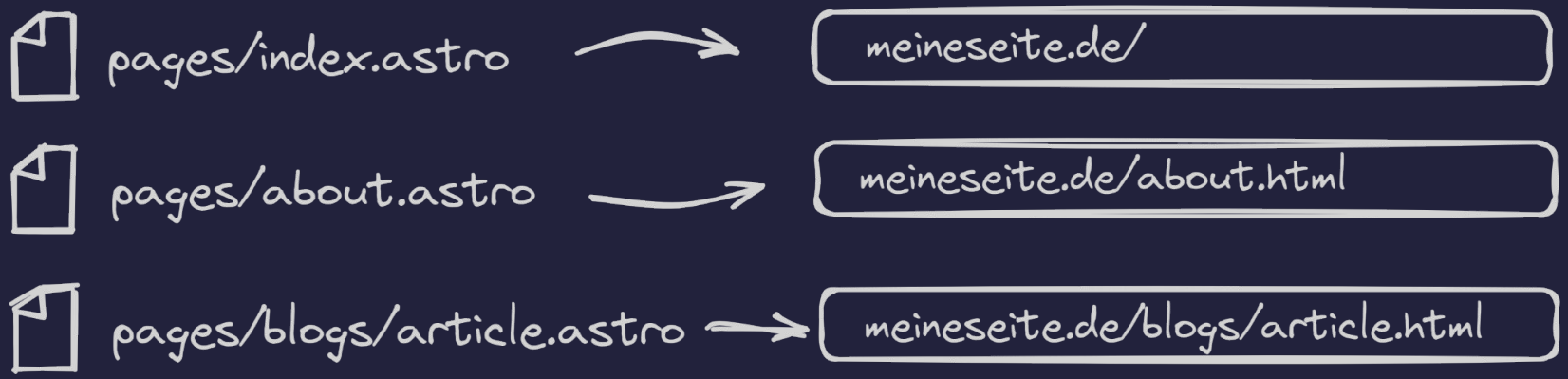
Showcase

Grundlagen 8

Dynamic Routing
Data Fetching

Recap: Statische Routen

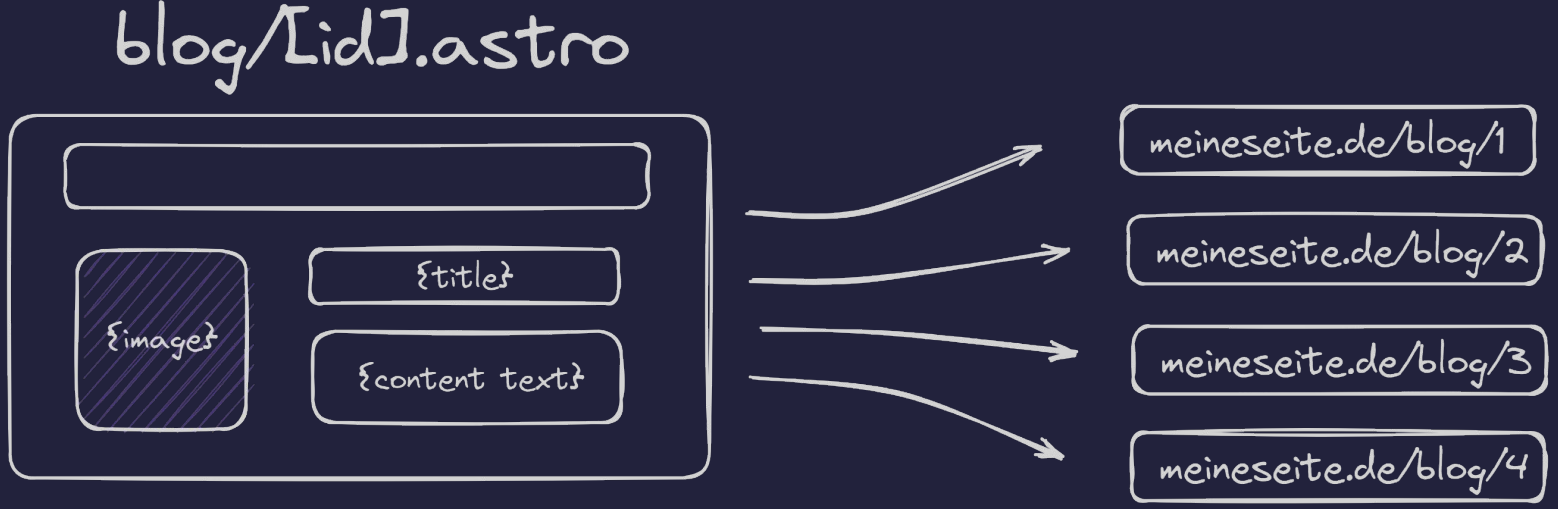
- Neue `.astro` oder `.md`-Datei im `pages`-Folder wird automatisch als neue Seite erstellt unter ist unter Dateinamen als URL erreichbar



Dynamisches Routing

- Verschiedene Routen und Seiten basierend auf variablen oder dynamischen Parametern zu generieren
- Möglichkeit anstatt jede einzelne Route manuell zu definieren
- Effizientere Erstellung von Inhalten
- Parameter, um Teile der URL zu variabilisieren
- `[parameter].astro`

Dynamisches Routing



Dynamisches Routing

- *getStaticPaths()*-Funktion wird benötigt
- Gibt Array von Objekten zurück mit `params`-Property
- Aus jedem Objekt wird dann eine entsprechende Route generiert
- Über `Astro.params` kann die Seite dann auf das Objekt zugreifen
- Um zusätzliche Daten an jede generierte Page zu geben, kann zudem `props`-Objekt genutzt werden

Dynamisches Routing

```
1   ---
2   export function getStaticPaths() {
3     return [
4       {params: {product: 'iphone'}},
5       {params: {product: 'surface'}},
6       {params: {product: 'pixel'}},
7     ];
8   }
9
10  const { product } = Astro.params;
11  ---
12  <div>Produktübersicht {product}!</div>
```

Data Fetching

- Alle Astro-Komponenten und Seiten haben Zugriff auf die `fetch`-Funktion
- Auch in Framework-Components verfügbar
- HTTP-Requests zu API-Endpoints
- Fetch wird standardmäßig zur Build-Time durchgeführt
- Bei aktiviertem SSR erst zur Laufzeit

Data Fetching

- Daten stehen dann dem Komponenten-Template zur Verfügung
- Top-Level-Await

```
---
import Contact from '../components/Contact.jsx';

const response = await fetch('https://randomuser.me/api/');
const data = await response.json();
const randomUser = data.results[0];
---
<!-- Data fetched at build can be rendered in HTML -->
<h1>User</h1>
<h2>{randomUser.name.first} {randomUser.name.last}</h2>

<!-- Data fetched at build can be passed to components as props -->
<Contact client:load email={randomUser.email} />
```

Fetching und Dynamic Routing in Kombination

```
---
export async function getStaticPaths() {
  const data = await fetch('...').then(response => response.json());

  return data.map((post) => {
    return {
      params: { id: post.id },
      props: { post },
    };
  });
}

const { id } = Astro.params;
const { post } = Astro.props;
---
<h1>{id}: {post.name}</h1>
```

Dynamisches Routing und Content Collections

- Content Collections liegen außerhalb vom Pages-Folder
- Wird `getCollection()`–Funktion in `getStaticPaths()` abegrufen, können daraus dynamische Pages erstellt werden
- Jeder Content-Eintrag beseitzt ein `slug`-Property, entspricht Dateinamen (falls nicht anders definiert)
- Über `slug`-Property kann Astro dann die URLs generieren

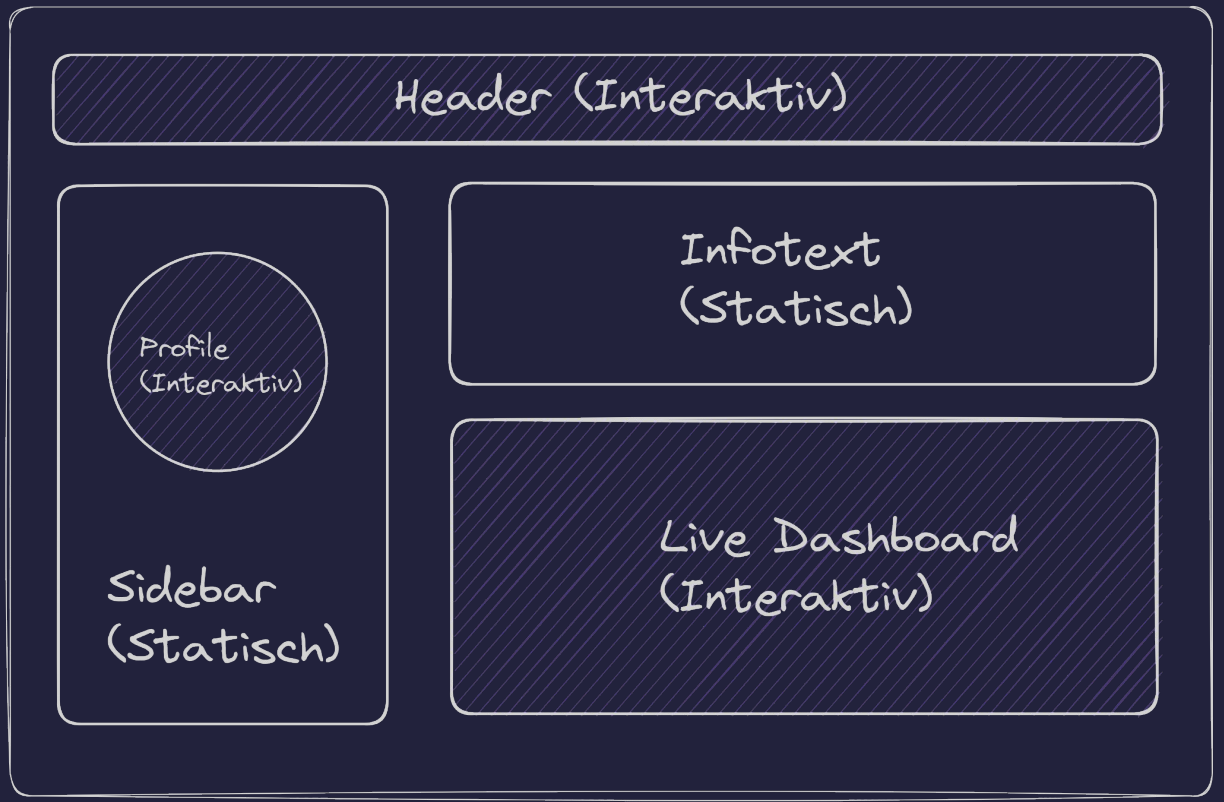
```
16
17  return articles.map(article => ({
18      params: { slug: article.slug }
19    }));
20
```

Übung 8

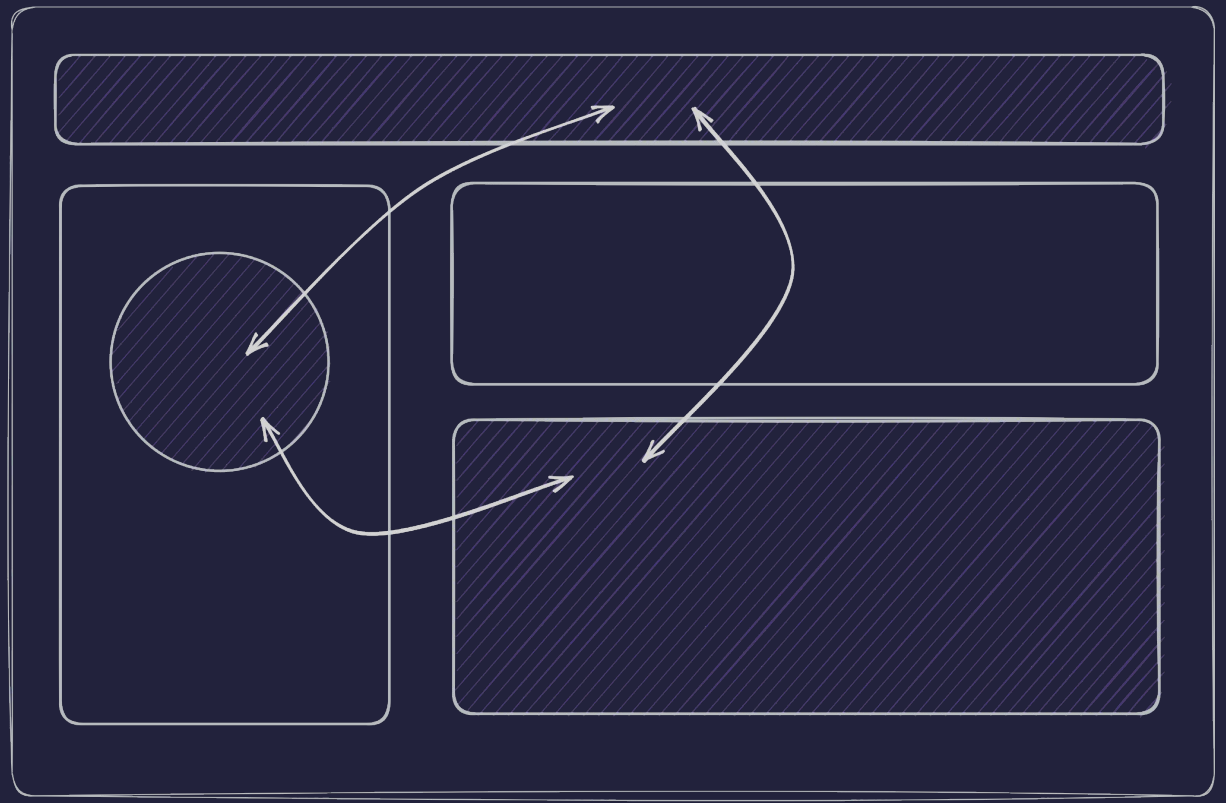
State Management

NanoStores

Astro Islands



State zwischen Islands teilen?



Nano Stores

- Globales State Management implementieren
- Jede Komponente (Astro Island) kann damit interagieren
- Leichtgewichtig: Weniger als 1kb JavaScript und keine Dependencies
- Framework-Agnostisch: State Sharing zwischen Komponenten verschiedener Frameworks funktioniert nahtlos
- TypeScript-Support
- `npm install nanostores @nanostores/[framework]`

Nano Stores

- Zuerst neue `.js` oder `.ts` - Datei erstellen, die den Store enthält
- Hier wird dann ein `atom` von NanoStores genutzt um einen neuen State in einer Konstante zu speichern und initialen Wert zu übergeben

```
1   import { atom } from 'nanostores';
2
3   export const counter = atom(0);
4
```

Nano Stores

- Jetzt kann der Store in jede Komponentendatei importiert werden, die ihn lesen oder ändern muss
- `useStore`-Funktion um Store abzufragen und bei Änderungen die Komponente neu zu rendern

```
1  import { useStore } from '@nanostores/react';
2  import { counter } from '../store/cartStore';
3
4  export default function CartButton() {
5    // read the store value with the `useStore` hook
6    const number = useStore(counter);
7
```

Nano Stores

- Helper-Funktionen: Über `store.get()` kann der Wert gelesen werden, über `store.set(nextvalue)` kann der Wert geändert werden
- `useStore()` verwenden wenn zusätzlich auch neu gerendert werden soll, am Besten wenn Wert in UI angezeigt wird

```
10    // write to the imported store using `.set`
11    return (
12      <button onClick={() => counter.set(counter.get()+1)}>Add</button>
13    )
```

Kann ich Nano Stores in .astro-Dateien verwenden?

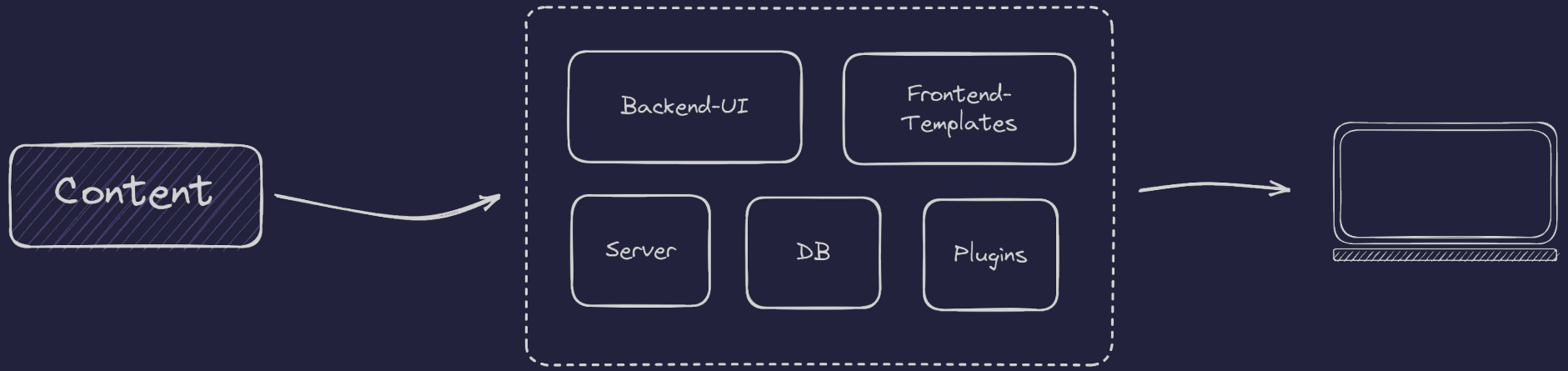
- Rein theoretisch möglich, jedoch nicht empfohlen
- Schreiben in einen Store aus einer `.astro`-Datei Komponente hat keine Auswirkung auf den Wert
- Nano Store kann nicht als “prop” an clientseitige Komponenten übergeben werden
- Kein Re-rendering von Astro-Komponenten

Übung 9

Headless CMS

Abgrenzung zu traditionellen CMS
Funktionsweise
Mögliche Ansätze

Traditionelles CMS



Vor- und Nachteile: Traditionelles CMS

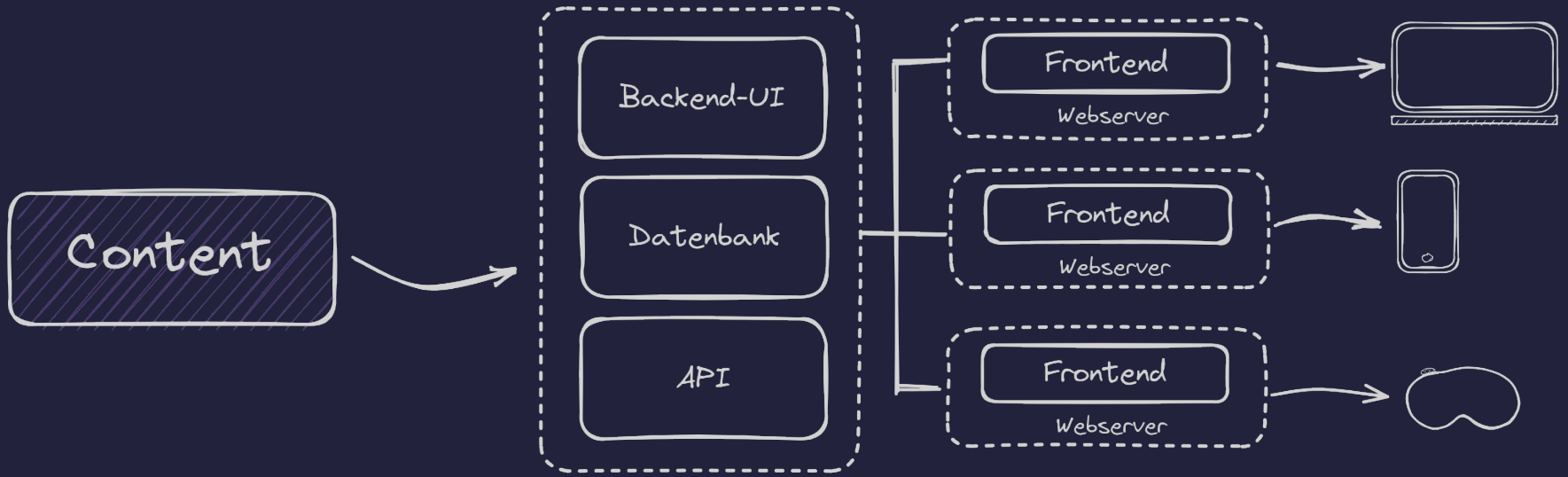


- Ohne Programmier-kenntnisse Webseiten und Anwendungen erstellen
- Geringer Aufwand bei Installation und Einrichtung
- Vielzahl an Themes und Plugins



- Inhalte i.d.R. an einziges Frontend gebunden
- Limitierte Wahl an Technologien
- Spezielle Features und Anforderungen nur schwer umsetzbar

Headless CMS



Vor- und Nachteile: Headless CMS



- Über einzige zentrale Schnittstelle Inhalte auf mehreren Plattformen ausspielen
- Flexiblere Wahl der Technologie
- Bessere Skalierbarkeit und Sicherheit



- Frontend muss selber programmiert und gestaltet werden
- I.d.R. keine WYSIWYG-Funktion
- Mehr Zeit- und Ressourcenaufwand bei Einrichtung

Vertreter



Storyblok



strapi



Contentful



DATO CMS

SANITY



ButterCMS

Headless CMS und Astro - eine perfekte Kombination?

- Einfache Entwicklung von modernen, performanten Webseiten und Anwendungen
- Headless CMS ermöglichen die flexible Verwaltung von Inhalten u. Bereitstellung über Schnittstelle
- Inhalte zur Build-Zeit o. auf Request abgerufen und in die HTML-Templates eingebettet
- Moderne Entwicklungsumgebung, die die Zusammenarbeit zwischen Entwicklern und Inhaltsautoren verbessert und gleichzeitig eine optimale Performance/UX bietet

Showcase

Suchmaschinen-optimierung

SEO-Komponente
Sitemaps

SEO

- Vorm Deployment bietet es sich an, eine SEO-Komponente zu erstellen, um SEO in unseren Blogs bzw. unsere Seiten miteinzubeziehen.
- Aufgabe dieser Komponente ist es, alle Eigenschaften der aufgerufenen Seite in Metatags umzuwandeln und weiterzuleiten.

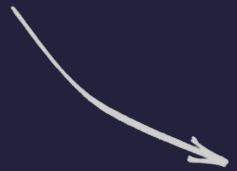
SEO

```
<!-- SEO -->
<link rel="canonical" href={url} />

<!-- Open Graph -->
<meta property="og:site_name" content="My Astro Blog" />
<meta property="og:title" content={title} />
<meta property="og:description" content={description} />
<meta property="og:url" content={url} />
<meta property="og:image" content={image?.src || siteData.image.src} />
<meta property="og:image:url" content={image?.src || siteData.image.src} />
<meta
  property="og:image:secure_url"
  content={image?.src || siteData.image.src}
/>
<meta property="og:image:type" content="image/jpeg" />
<meta property="og:image:width" content="1200" />
<meta property="og:image:height" content="600" />
<meta property="og:image:alt" content={image?.alt || siteData.image.alt} />
```

SEO

```
<meta charset="utf-8">
<link rel="icon" type="image/svg+xml" href="/favicon.svg">
<meta name="viewport" content="width=device-width">
<meta name="generator" content="Astro v1.6.13">
<title>My Blog</title>
<meta name="description" content="My musings about the Astro framework">
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Atkinson+Hyperlegible:ital,wght@0,400;0,700;1,400;1,700&display=swap" rel="styleshe
et">
<script type="module" src="/C:/Users/yassi/.vscode/astro-blog-tutorial/src/components/Footer.astro?astro&type=script&index=0&lang.ts">
</script>
<script type="module" src="/C:/Users/yassi/.vscode/astro-blog-tutorial/src/components/Nav.astro?astro&type=script&index=0&lang.ts">
</script>
<script type="module" src="/@vite/client"></script>
<script type="module" src="/@fs/C:/Users/yassi/.vscode/astro-blog-tutorial/node_modules/.vite/deps/astro_runtime_client_hmr__js.js?v=2c2
4733d"></script>
<script type="module" src="/src/styles/global.css"></script>
<style type="text/css" data-vite-dev-id="C:/Users/yassi/.vscode/astro-blog-tutorial/src/styles/global.css"> == $0
```



```
▼ <head> == $0
    <meta charset="utf-8">
    <link rel="icon" type="image/svg+xml" href="/favicon.svg">
    <meta name="viewport" content="width=device-width">
    <meta name="generator" content="Astro v1.6.13">
    <title>My Blog</title>
    <meta name="description" content="My musings about the Astro framework">
    <!-- SEO -->
    <link rel="canonical" href="http://localhost:3000/">
    <!-- Open Graph -->
    <meta property="og:site_name" content="My Astro Blog">
    <meta property="og:title" content="My Blog">
    <meta property="og:description" content="My musings about the Astro framework">
    <meta property="og:url" content="http://localhost:3000/">
    <meta property="og:image" content="/images/image-default.jpg">
    <meta property="og:image:url" content="/images/image-default.jpg">
    <meta property="og:image:secure_url" content="/images/image-default.jpg">
    <meta property="og:image:type" content="image/jpeg">
    <meta property="og:image:width" content="1200">
    <meta property="og:image:height" content="600">
    <meta property="og:image:alt" content="My Astro Blog">
```

Good to know: Sitemaps

- Eine Sitemap ist eine XML-Datei, die alle Seiten, Videos und Dateien auf Ihrer Website auflistet. Suchmaschinen wie Google lesen diese Datei, um die Website effizienter zu crawlen.
- Es wird empfohlen, für sämtliche Mehrseiten-Websites eine Sitemap zu erstellen, insbesondere wenn das Ziel darin besteht, eine hohe Besucherzahl zu erzielen.
- Mit Astro kann eine Sitemap innerhalb kürzester Zeit erstellt werden.

Good to know: Sitemaps

1. npm install @astrojs/sitemap
2. Definieren der Site Eigenschaft:
3. npm run build

```
import { defineConfig } from 'astro/config';
import sitemap from '@astrojs/sitemap';

export default defineConfig({
  // ...
  integrations: [sitemap()],
})

```

Sitemaps filter() Funktion

- Standardmäßig werden alle Seiten in der Sitemap aufgenommen.
- Sollte die Filterfunktion hinzugefügt werden, können die aufgenommenen Seiten nach URL gefiltert werden.

```
sitemap({
  filter: (page) =>
    page !== "https://astro-blog-cip/blog-1" &&
    page !== "https://astro-blog-cip/blog-2" &&
    page !== "https://astro-blog-cip/blog-3" &&
    page !== "https://astro-blog-cip/about",
})]
```

Grundlagen 9

Deployment

Deployment

- Astro bietet eine detaillierte Deployment Anleitung für 20 unterschiedliche Services u.a.:



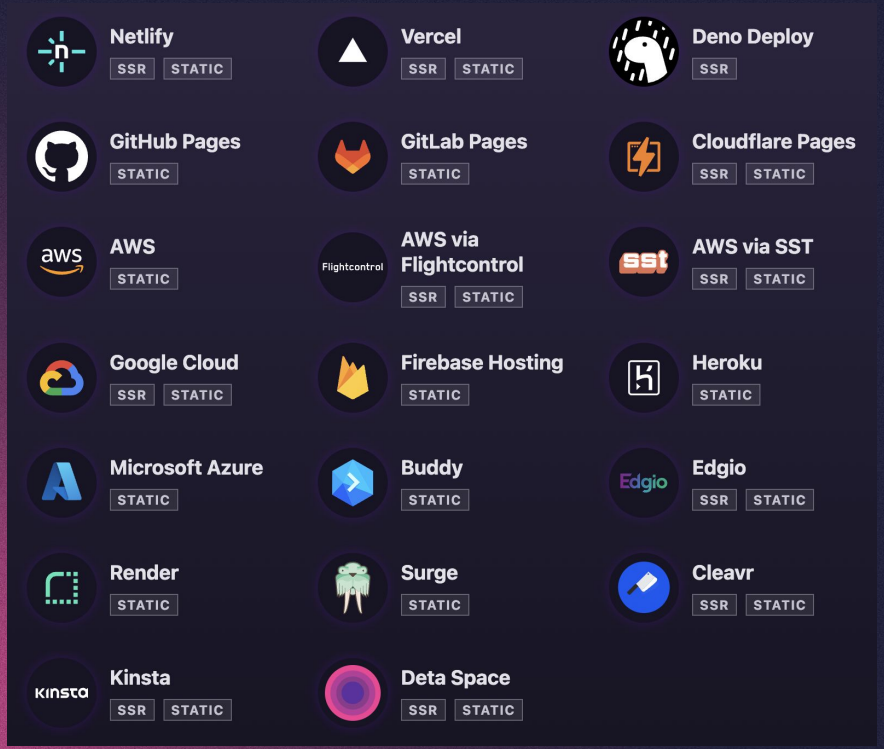






Google Cloud

Deployment Dienstleister



Deployment

- Für ein schnelles und intuitives Deployment können wir unseren bereits bestehenden Git repository mit einem Host verbinden.
- Da wir uns ein kontinuierliches Deployment wünschen, entscheiden wir uns für netlify, die diese Dienstleistung unterstützen.



Deployment Prozess

1. npm install @astro/netlify
2. Konfigurieren des Outputs und Adapters.

```
import { defineConfig } from 'astro/config';
import netlify from '@astrojs/netlify/functions';

export default defineConfig({
  output: 'server',
  adapter: netlify(),
});

```

Ihr wollt noch mehr wissen?

- https://astro.build/blog/
- https://docs.astro.build/en/getting-started/
- https://entwickler.de/javascript/island-architecture-astro
- https://bejamas.io/blog/practical-guide-to-astro-js-framework/
- https://blog.logrocket.com/understanding-astro-islands-architecture
- https://github.com/nanostores/nanostores
- https://www.sitecore.com/de-de/knowledge-center/digital-marketing-resources/what-is-a-headless-cms

Abschlussquiz

