



Vorlesung 3

05 Grundlagen

5.1. Routing

5.2. Data fetching

06 Fortsetzung Grundlagen

6.1. State management

6.2. Nitro

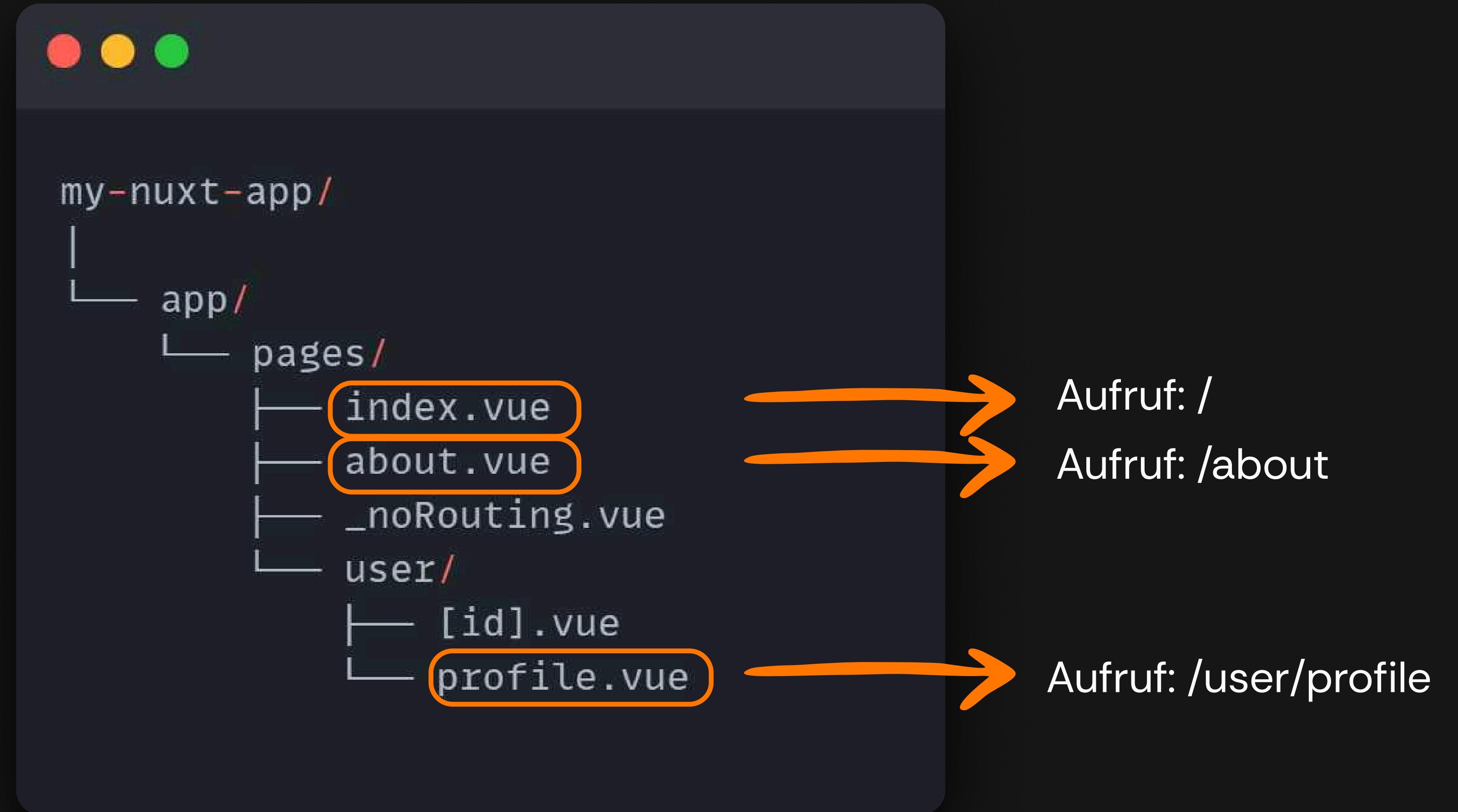


Wiederholung Ordnerstruktur

```
my-nuxt-app/
|
|   app/
|     components/      # Globale Vue-Komponenten
|     layouts/
|       default.vue    # Standard-Layout
|
|     pages/           # Definiert automatisch das Routing-System
|       index.vue      # Homepage
|
|     app.vue          # Haupteinstiegspunkt (root layout)
|
|   server            # Nitro-Komponenten
```



Routing





Routing

```
my-nuxt-app/
|
└── app/
    └── pages/
        ├── index.vue
        ├── about.vue
        ├── _noRouting.vue
        └── user/
            ├── [id].vue
            └── profile.vue
```



wird nicht geroutet



Routing

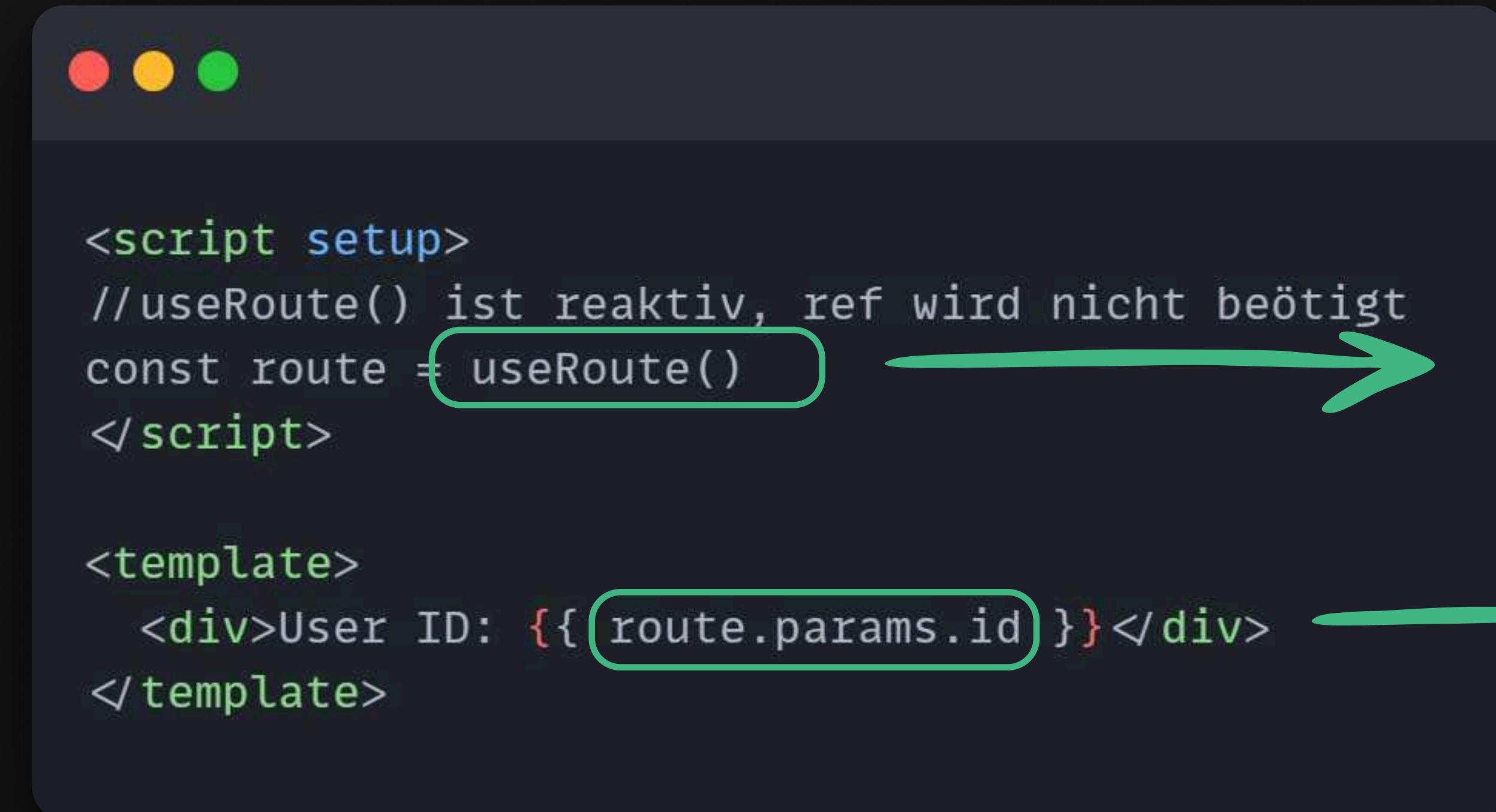
```
my-nuxt-app /  
|  
└── app /  
    └── pages /  
        ├── index.vue  
        ├── about.vue  
        └── _noRouting.vue  
    └── user /  
        ├── [id].vue  
        └── profile.vue
```

dynamisches Routing

Aufruf: /user/123



Routing



```
<script setup>
//useRoute() ist reaktiv, ref wird nicht benötigt
const route = useRoute() → Zugriff auf Routendetails
</script>

<template>
  <div>User ID: {{ route.params.id }}</div> → Parameter lesen
</template>
```



Aufgabe 11

10 min

11 – Routing

Ziel

In dieser Übung lernst du, wie du mit **Nuxt 3 Routing** arbeitest, dynamische Seitenparameter nutzt und Navigation über die Sidebar steuerst.
Am Ende kannst du zwischen Startseite, Profilen und Einstellungen wechseln – mit echten Nuxt-Routen.

Ausgangspunkt

Dein Projekt besteht bereits aus folgenden Seiten:

- `pages/index.vue` → Startseite
- `pages/profile/[username].vue` → Profilseite (dynamisch)
- `pages/settings.vue` → Einstellungen

In der Sidebar ist aktuell noch kein Routing aktiv. Das sollst du jetzt ändern.

Aufgabe

1. Sidebar mit Navigation erweitern

Öffne die Datei `components/Sidebar.vue` und ersetze die bisherigen statischen Elemente durch **NuxtLinks**:

Diese Links führen:

- zur Startseite "/"
- zum eigenen Profil (Standard-Username: `hdm.stuttgart`) "/profile"
- zur Einstellungsseite "/settings"

Die Sidebar bleibt durch das globale Layout (`layouts/default.vue`) auf allen Seiten sichtbar.

2. Dynamisches Profil-Routing verstehen

In Nuxt 3 erzeugt eine Datei wie `pages/profile/[username].vue` automatisch eine **dynamische Route**, die z. B. unter `/profile/hdm.stuttgart` oder `/profile/max.mustermann` erreichbar ist.

Du musst in dieser Datei also nur noch den Parameter auslesen.



Data fetching

Methode, um Daten zu fetchen	Use Case
<code>\$fetch</code>	Rohe HTTP-Anfrage
<code>useFetch()</code>	Einfaches Laden
<code>useAsyncData()</code>	Flexibles Laden, Laden mit Logik



Data fetching

\$fetch

- einfachster Weg für Netzwerkanfrage
- built-in library (fetch)
- bei Verwenden in setup-Funktion einer Komponente:
 - Könnte doppelt aufgerufen werden (Rendering auf Server und während Hydration)
 - Kann zu Problemen führen
- nicht reaktiv (kann es manuell einstellen)
- für clientseitige Interaktionen geeignet



Data fetching

\$fetch

```
<script setup lang="ts">
async function addTodo () {
  const todo = await $fetch('/api/todos', {
    method: 'POST',
    body: {
      // My todo data
    },
  })
}
</script>
```



Data fetching

`useFetch()`

- baut auf `$fetch` auf
- ladet einmalig während des SSR
- reaktiv im Browser, wenn Parameter sich ändern
- verhindert doppeltes Laden derselben Anfrage → Daten werden gecached
- einfache Syntax, speziell für HTTP-Requests optimiert



Data fetching

useFetch()

Rückgabewerte	Typ	Beschreibung
data	Ref<DataT undefined>	enthält geladene Daten
pending	Ref<boolean>	true, solange Anfrage läuft
error	Ref<ErrorT undefined>	Error, falls Fehler
status	Ref<'idle' 'pending' 'success' 'error'>	Status der Anfrage
refresh/execute	() => Promise<void>	Manuelles Nachladen
clear	() => void	Löscht gespeicherte Daten aus Cashe



Data fetching

useFetch()



```
<script setup lang="ts">
const { data: count } = await useFetch('/api/count')
</script>

<template>
  <p>Page visits: {{ count }}</p>
</template>
```



Data fetching

useAsyncData()

- lädt Daten asynchron, bevor die Seite gerendert wird (SSR)
- reaktiv, falls Parameter sich ändern
- deutlich umfangreicher als useFetch()
- mehrere fetches können gleichzeitig ausgeführt werden
- verhindert doppeltes Laden derselben Anfrage → Daten werden gecached
- Unterstützt beliebige asynchrone Funktionen, nicht nur HTTP-Requests
- Muss einen key und einen handler übergeben



Data fetching

useAsyncData()



```
<script setup lang="ts">
const route = useRoute()
const userId = computed(() => `user-${route.params.id}`)

// When the route changes and userId updates, the data will be automatically refetched
const { data: user } = useAsyncData(
  userId,
  () => fetchUserById(route.params.id),
)
</script>
```



Data fetching

useAsyncData()

```
//fetching ohne HTTP
const { data } = useAsyncData(async () => {
  const { data, error } = await supabase.from("countries").select();
  return data;
});

//mehrere fetches parallel ausführen
const { data } = await useAsyncData(() => {
  return Promise.all([
    $fetch("https://dummyjson.com/api/items/1"),
    $fetch("https://dummyjson.com/api/reviews?item=1"),
  ]);
});
```



Aufgabe 12

10 min



* Lab 12: Data Fetching in Vue/Nuxt

Ziel der Übung

In dieser Übung lernst du, wie du Kommentare dynamisch über eine API lädst, anstatt sie lokal zu speichern. Du erweiterst die Komponente `PostOverlay.vue`, damit sie Kommentare direkt aus der API <https://dummyjson.com/comments/post/{postId}> lädt.

Ausgangspunkt

In deiner `PostOverlay.vue` sind Kommentare aktuell lokal gespeichert:

```
const comments = ref([
  { id: 1, text: 'Toller Beitrag!' },
  { id: 2, text: 'Freue mich schon auf mehr Posts 🎉' }
])
```

Diese sollen nun durch echte Kommentare aus der API ersetzt werden.

Aufbau der API Response

Beim Aufruf von <https://dummyjson.com/comments/post/1> erhältst du folgende JSON-Struktur:

```
{
  "comments": [
    {
      "id": 1,
      "body": "This is some awesome thinking!",
      "postId": 1,
      "user": {
        "id": 63,
        "username": "emilys"
      }
    },
    {
      "id": 2,
      "body": "What terrific math skills you're showing!",
      "postId": 1,
      "user": {
        "id": 12,
        "username": "tomhanks"
      }
    }
  ],
  "total": 5,
```



Vorlesung 3

05 Grundlagen

5.1. Routing

5.2. Data fetching

06 Fortsetzung Grundlagen

6.1. State management

6.2. Nitro



State Management

- Globaler, reaktiver Daten-Speicher
- über useState eingebaut, kein extra Store nötig
- Zugriff überall über gleichen Schlüssel
- Änderungen sind sofort in allen Komponenten sichtbar

- + Einfach
- + SSR-kompatibel
- + Reaktiv



State Management

```
<template>
  <div>
    <h2>Homepage</h2>
    <p>Aktueller Wert: {{ counter }}</p>
    <button @click="increment">Erhöhe</button>
    <button @click="decrement">Verringere</button>
  </div>
</template>
<script setup lang="ts">
import { useState } from '#app'

const counter = useState<number>('counter', () => 0)

function increment() {
  counter.value++
}

function decrement() {
  counter.value--
}
</script>
```



Neuer State wird erstellt



State Management

```
<template>
  <div>
    <h2>About</h2>
    <h3>Count: {{ counter }}</h3>
  </div>
</template>
<script setup lang="ts">
  const counter = useState<number>('counter')
</script>
```

→ Zugriff über ID



State Management

auch Objekte können gespeichert werden



```
<template>
  <div>
    <h2>Homepage</h2>
    <p>Aktueller Wert: {{ counter.age }}</p>
    <button @click="increment">Erhöhe</button>
    <button @click="decrement">Verringere</button>
    <input
      type="text"
      v-model="counter.name"
      placeholder="Gib deinen Namen ein"
    />
  </div>
</template>
<script setup lang="ts">
import { useState } from '#app'

const counter = useState<{ age: number; name: string }>('counter', () => ({
  age: 0,
  name: ""
}))
```

```
function increment() {
  counter.value.age++
}

function decrement() {
  counter.value.age--
}
```

```
</script>
```



Aufgabe 13

10 min



⚙️ Lab 13: State Management in Nuxt/Vue

Ziel der Übung

In dieser Übung lernst du, wie du bestehende Eingabefelder in der Seite `/settings/profile` so umbaust, dass sie globale Zustände mit `useState()` verwenden. Dadurch werden Änderungen an den Profildaten automatisch auch im Profil (`/profile/[username]`) sichtbar – ohne die Seite neu zu laden.

Ausgangssituation

In der Seite `/settings/profile` sind bereits Eingabefelder vorhanden, die lokale Refs verwenden, z. B.:

```
const username = ref('hdm.stuttgart')
const fullname = ref('Hochschule der Medien (HdM)')
const description = ref('Offizieller Instagram-Account der #hdmstuttgart 📸')
```

Diese sollen nun durch globale States mit `useState()` ersetzt werden.

Aufgabenstellung

1. Ersetze alle lokalen Refs durch globale States:

```
const username = useState('username', () => 'hdm.stuttgart')
const fullname = useState('fullname', () => 'Hochschule der Medien (HdM)')
const description = useState('description', () => 'Offizieller Instagram-Account der #hdmstuttgart 📸')
```

2. Verwende dieselben States auch in der Profilseite (`/profile/[username].vue`):

So sind beide Seiten automatisch miteinander synchronisiert.

3. Teste die Reaktivität:

- Öffne `/settings/profile`
- Ändere den Namen oder die Beschreibung
- Wechsle zu `/profile/[username]`
- Die Änderungen sind sofort sichtbar



Nitro

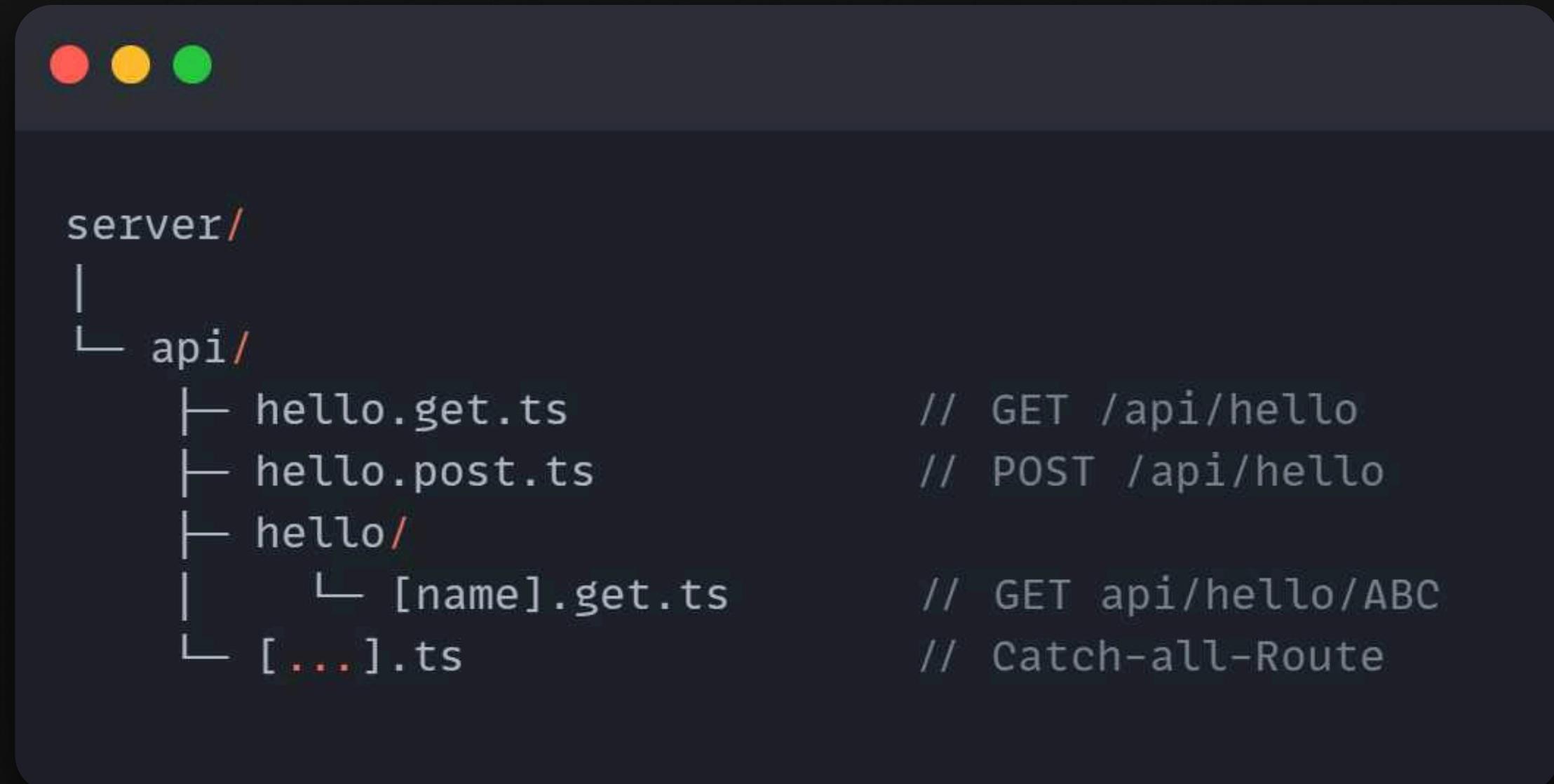
- Server-Engine von Nuxt
- Zuständig für SSR und API-Routing
- hat ebenfalls Ordnerstruktur

```
server/
  |
  +-- api/
        +-- hello.get.ts    // /api/hello
  |
  +-- routes/
        +-- bonjour.ts     // /bonjour
  |
  +-- middleware/
        +-- logger.ts      // log all requests
  |
  +-- plugins/
        +-- init.ts         // plugins
  |
  +-- utils/
        +-- db.ts           // custom handler
```



Nitro - Routing

- funktioniert über server/api und server/routes
- Prinzipien gelten wie bei Nuxt
- Jedes File braucht einen Eventhandler
- HTTP-Methode wird in Filename geschrieben
- Mithilfe von Funktionen kann auf Parameter, Query und Body zugegriffen werden



```
server/
└── api/
    ├── hello.get.ts          // GET /api/hello
    ├── hello.post.ts         // POST /api/hello
    └── hello/
        └── [name].get.ts      // GET api/hello/ABC
    └── [...].ts              // catch-all-Route
```



Nitro - API



server/api/[name].get.ts

```
export default defineEventHandler(async (event) => {
```

→ Standard Eventhandler-Methode

```
  const name = getRouterParam(event, 'name')
```

→ Zugriff auf Parameter

```
  const query = getQuery(event) as QueryParams
```

→ Zugriff auf Querys

```
  const body = await readBody(event) as RequestBody
```

→ Zugriff auf Body (async)

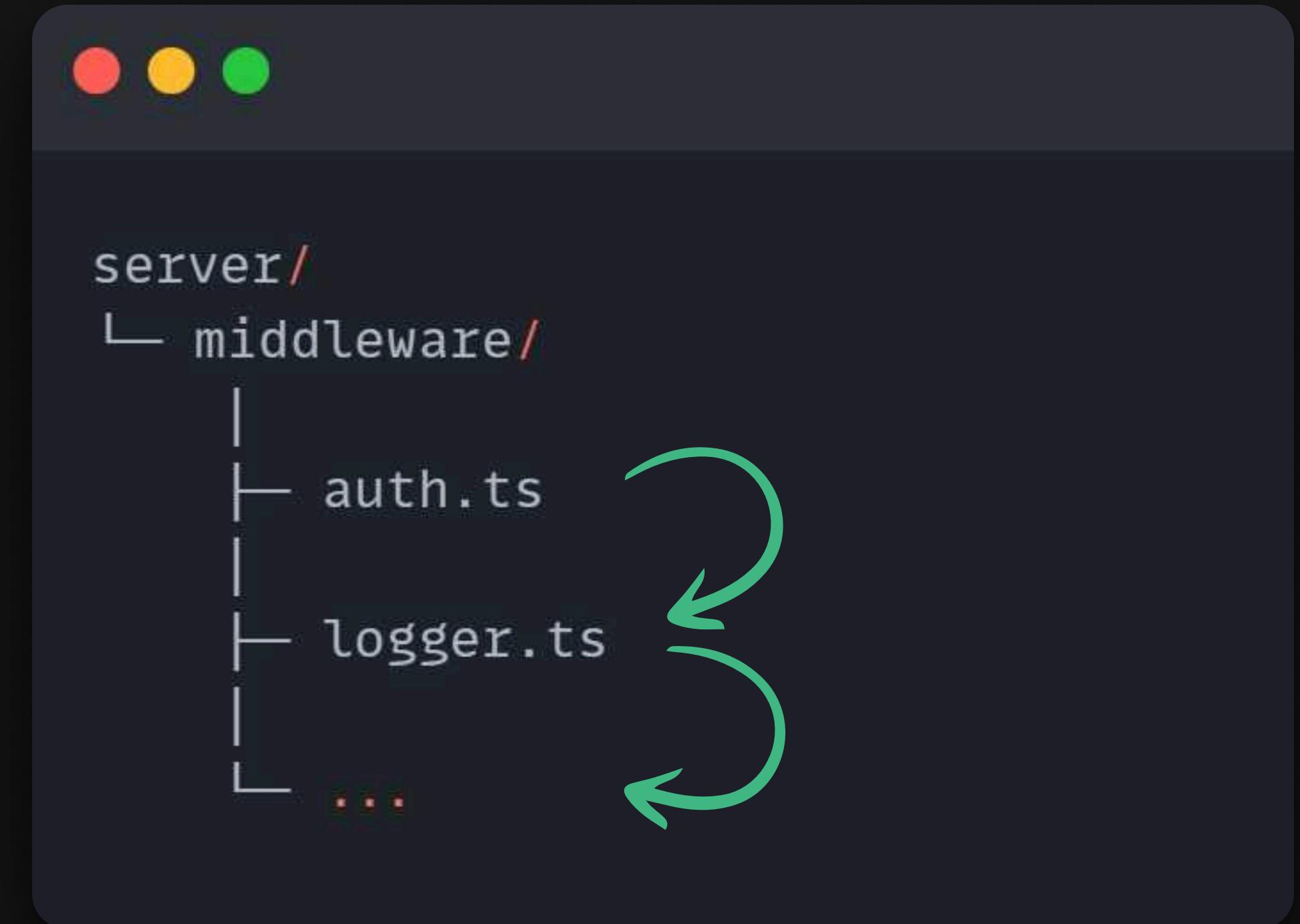
```
  return {
    hello: "New Post was created",
  }
})
```

→ Return als JSON oder Promise



Nitro - Middleware

- befinden sich im server/middleware Ordner
- benötigen ebenfalls Eventhandler
- werden bei jedem Aufruf ausgeführt
 - Konditionen können manuell hinzugefügt werden
 - Werden in Reihenfolge des Verzeichnisses ausgeführt





Nitro - Middleware



```
export default defineEventHandler((event) => {
  console.log('New request: ' + getRequestURL(event))
})
```



Standard Eventhandler-Methode



Nitro - Middleware

```
export default defineEventHandler((event) => {
  // Will only execute for /auth route
  if (getRequestURL(event).pathname.startsWith('/auth')) {
    event.context.user = { name: 'Nitro' }
  }
})
```



Kondition



Aufgabe 14

10 min



* Lab 12: Data Fetching in Vue/Nuxt

Ziel der Übung

In dieser Übung lernst du, wie du Kommentare dynamisch über eine API lädst, anstatt sie lokal zu speichern. Du erweiterst die Komponente `PostOverlay.vue`, damit sie Kommentare direkt aus der API <https://dummyjson.com/comments/post/{postId}> lädt.

Ausgangspunkt

In deiner `PostOverlay.vue` sind Kommentare aktuell lokal gespeichert:

```
const comments = ref([
  { id: 1, text: 'Toller Beitrag!' },
  { id: 2, text: 'Freue mich schon auf mehr Posts 🎉' }
])
```

Diese sollen nun durch echte Kommentare aus der API ersetzt werden.

Aufbau der API Response

Beim Aufruf von <https://dummyjson.com/comments/post/1> erhältst du folgende JSON-Struktur:

```
{
  "comments": [
    {
      "id": 1,
      "body": "This is some awesome thinking!",
      "postId": 1,
      "user": {
        "id": 63,
        "username": "emilys"
      }
    },
    {
      "id": 2,
      "body": "What terrific math skills you're showing!",
      "postId": 1,
      "user": {
        "id": 12,
        "username": "tomhanks"
      }
    }
  ],
  "total": 5,
```



Checkerfragen

