

Semesterarbeit Webarchitekturen: Thema -> Next.JS

Ersteller: Lorant Gulyas

#####

- Semesterarbeit Webarchitekturen: Thema -> Next.JS
 - Ersteller: Lorant Gulyas
 - Vorwort
 - Technische Voraussetzungen in der Entwicklungsumgebung
 - Betriebssystem
 - Benötigte Accounts
 - Initialisierung eines neuen Projektes
 - Start den Webserver
 - Wir haben es geschafft, wir haben unsere erste eigene Next.Js App angelegt
 - Wir können die Webseite öffnen, indem wir die Localhost Adresse in unseren Browser öffnen
 - Datenbank Verbindung erzeugen
 - Datenmodel erzeugen
 - API zu MongoDB für Abfrage der Daten
 - API Schnittstelle zum hochladen von Daten.
 - Die Startseite
 - Schlusswort

Vorwort

Diese Dokumentation hat zum Ziel die Installation einer kleinen To-do-Liste als Webseite zu begleiten, hierbei werden alle wichtigen Schritte beschrieben, um an Ende eine funktionierende Anwendung mit Datenbank-Anbindung zu haben. Diese Dokumentation wurde auf einem macOS Betriebssystem erstellt. Die Befehle können auf einer Linux System leicht abweichen. Die Logik der Befehle ändert sich jedoch nicht.

Die Finale Seite kann man unter folgenden URL sehen:

[Todo Liste für Web Architekturen](#)

Den dazugehörigen Source Code findet man auf Github:

[Github](#)

#####

Technische Voraussetzungen in der Entwicklungsumgebung

Betriebssystem

Es wird eine Linux oder Mac OS X Umgebung benötigt. Sofern man auf einem Mac oder Linux arbeitet, sind alle notwendigen Tools nativ verfügbar.

Sofern Windows verwendet wird, muss entweder eine virtuelle Maschine mit einer Linux Instanz installiert werden, oder ein WSL System in der Shell angelegt werden. Unter folgenden Links findet man eine Anleitung zum Installieren einer virtuellen Maschine sowie WSL Subsystem.

[Ubuntu als Virtuelle Maschine installieren](#)

[WSL auf Windows installieren](#)

Node:

eine Instanz des Node Framework muss am System installiert sein.

Um zu prüfen, ob wir Node installiert haben, können wir den folgenden Befehl absetzen.

```
node --version
```

Sofern Node bereits installiert ist, erhalten sie eine Information über die Version. Diese sollte mindestens mit einer 18 beginnen. Sollte Node nicht installiert sein, wird eine Fehlermeldung erscheinen, dass der Befehl nicht erkannt wurde.

In diesem Fall muss Node erst installiert werden. Das kann man etwas über Homebrew bei macOS machen.

```
brew install node
```

Bei Linux kann man die Pakete großteils mit Paketmanagern installieren,

zB.

```
yum install nodejs14
```

oder

```
pacman -S node npm
```

VsCode:

Der Code Editor ist unabdingbar, um den Source Code der Seite zu schreiben. Das Tool ist Gratis und kann unter folgenden URI heruntergeladen werden.

[Vs Code Editor](#)

Einfach installieren, und danach starten. Im IDE der Software hat man die Möglichkeit Plugins zu installieren, sowie Accounts zu verknüpfen. Wir werden diese Funktion nutzen, um VsCode mit Github zu verbinden, um das Update der App zu erleichtern. Links unten im Editor Fenster kann man ein Avatar sehen, über den man sich auf Github anmelden kann, sofern das entsprechende Plugin installiert wurde.

Verwendete Plugins:

- GitHub Pull Request
- Git History
- GitLens
- Git Graph

Benötigte Accounts

Github:

Dieser Account wird die Schaltzentrale für unsere Versionsverwaltung. Wir werden unsere App auf Github laden, von wo es sich mit unseren Webserver synchronisiert wird. Hierdurch ist es möglich, Änderungen direkt von unserem Editor bis auf die Webseite per Knopfdruck einzuspielen.

Um uns mit Vscode verbinden zu können, müssen wir uns zuerst registrieren.

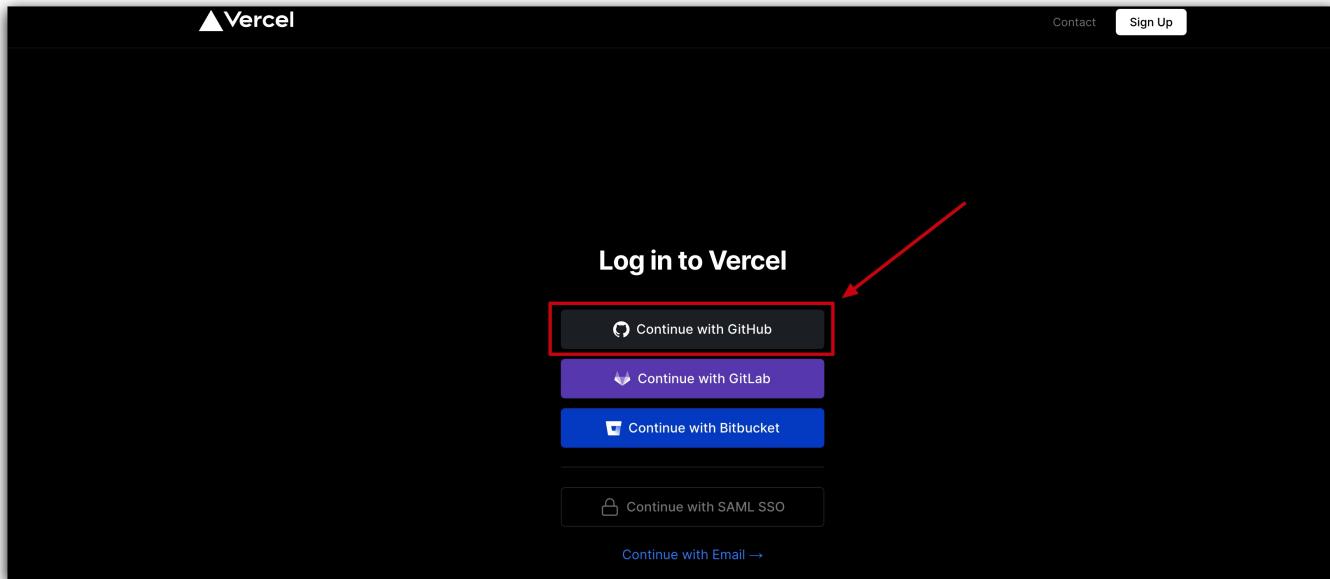
[Github Login Seite](#)

Hier einfach rechts oben auf Sign Up klicken und ein neues Konto anlegen. Eine Empfehlung ist es, ein schon bestehenden Account zu verwenden. Hierzu bietet sich z.B. ein Google Account.



Vercel:

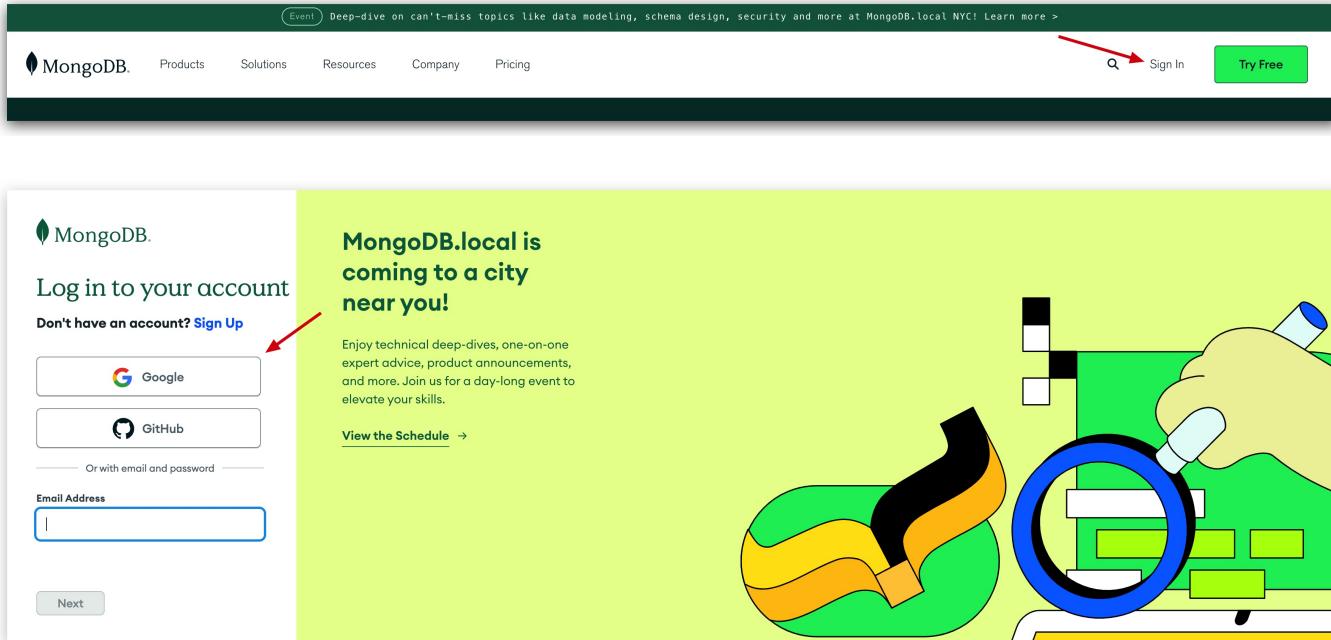
Zusätzlich zu Github benötigen wir auch einen Account auf Vercel. Das gute ist, dass man hier direkt seinen Github Zugang verwenden kann. Einfach beim Registrieren Github wählen, und die Verbindung bestätigen.



Sobald diese beiden Accounts angelegt wurden, kann auch schon mit der Erstellung der Webseite begonnen werden.

MongoDB:

Als Letztes benötigen wir noch einen Account bei MongoDB, das ist eine dokumentenbasierte Datenbank, auf der wir uns eigene Strukturen ganz leicht anlegen können, damit wir diese später über die Webseite abrufen können. Hier haben wir auch die Möglichkeit, und einen kostenlosen Account anzulegen. Auch hier haben wir die Möglichkeit uns mit unseren Google Account anzumelden.



Nachdem wir uns angemeldet haben, werden wir auf das Dashboard geleitet. Von hier aus können wir unsere Datenbanken verwalten, indem wir links im Menü auf Database klicken.

Auf der neu geladenen Seite haben wir die Möglichkeit, eine neue Datenbank zu erstellen. Dazu einfach im rechten Bereich auf Create klicken. Hier haben wir die Möglichkeit für unsere Testzwecke eine kostenlose Datenbank anzulegen. Hierzu einfach die Shared Methode wählen. Zusätzlich zur Shared Option können wir auch eine Serverless oder Dedicated Variante erstellen. Diese sind aber kostenpflichtig, und für unsere Testzwecke nicht vonnöten, daher werden sie in dieser Anleitung auch nicht näher erläutert. Beim Erstellen der Instanz lohnt es sich darauf zu achten, dass wir einen Server auswählen, der möglichst nah an der geografischen Ort der Webseite gelegen ist. Zusätzlich gibt es die Möglichkeit zwischen den 3 großen Anbietern einen auszuwählen. Für unser Projekt haben wir keine fixe Vorgabe. Später kann es aber aufgrund von z. B. DSGVO zu Einschränkungen in der Auswahl kommen.

LORANT'S ORG - 2023-03-04 > SHOOT

Database Deployments

Find a database deployment...

Monitoring for Photogulasch is Paused
Monitoring will automatically resume when you connect to your cluster.
[Visit the documentation](#) for more info.

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED APP SERVICES	ATLAS SQL	ATLAS SEARCH
6.0.6	AWS / N. Virginia (us-east-1)	MD Sandbox (General)	Replica Set - 3 nodes	Inactive	Multiple applications linked	Connect	Create Index

CLUSTERS > CREATE A SHARED CLUSTER

Create a Shared Cluster

Serverless Dedicated **Shared**

For learning and exploring MongoDB in a sandbox environment. Basic configuration controls.
No credit card required to start. Upgrade to dedicated clusters for full functionality.
Explore with sample datasets. Limit of one free cluster per project.

Cloud Provider & Region

AWS, Frankfurt (eu-central-1)

NORTH AMERICA **EUROPE** **AUSTRALIA**

- ★ Recommended region ⓘ ⓘ Dedicated tier region ⓘ
- N. Virginia (us-east-1)** ★
- Stockholm (eu-north-1) ★
- Sydney (ap-southeast-2) ★
- Oregon (us-west-2) ★
- Paris (eu-west-3) ★
- Melbourne (cp-southeast-4) ★
- Ohio (us-east-2) ★ ⓘ
- Ireland (eu-west-1) ★
- Seoul (ap-northeast-2) ★
- N. California (us-west-1) ⓘ
- Frankfurt (eu-central-1)
- London (eu-west-2) ★ ⓘ
- Mumbai (ap-south-1) ★
- Montreal (ca-central-1) ★ ⓘ
- Singapore (ap-southeast-1) ★

FREE Free forever! Your MD cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Create Cluster

Wenn unsere Datenbank erstellt wurde, erscheint sie in der Auflistung auf der vorherigen Seite unter dem Punkt „Database Deployments“

Als Nächstes müssen wir den Netzwerkzugang sicherstellen. Das ist die Möglichkeit den Zugriff einzugrenzen, z. B. auf bestimmte Geräte, von denen man auf die Datenbank zugreifen kann. Hierzu gehen wir im linken Menü auf „Network Access“ (**1**) und klicken im rechten Bereich auf „Add IP Address“ (**2**). Hier geben wir die IP 0.0.0.0/0 an, damit der Zugriff von jeder IP-Adresse möglich ist. Wenn wir später einen eigenen Server betreiben, benötigt die Datenbank nur Zugriff vom Webserver. Mit diesem Menü haben wir eine sehr gute Möglichkeit die Sicherheit unserer Installation zu erhöhen, indem wir den Zugriff auf einen bestimmten Server begrenzen.

Nachdem wir den Netzwerkzugriff überprüft und angepasst haben, müssen wir einen Benutzer erstellen, mit dem wir später über die API auf die Datenbank zugreifen werden. Hierbei ist es wichtig, dass es verschiedene Ebenen des Zugriffs gibt. Allgemein kann man sagen, dass Schreibrechte nur für Benutzer benötigt werden, die auch aktiv Inhalt verändern und ergänzen werden. Sofern der Zugriff rein zur Abfrage dient, reicht der Lesezugriff.

Die Einstellung der Zugriffsrechte erfolgt über den Menüpunkt "Database Access" (1), wo wir eine Liste aller schon angelegter Benutzer sehen. Hier können wir dann rechts auf "add new Database User" (2) klicken, um einen neuen Zugang zu erstellen. Als Methode wählen wir "Password" aus (3) und tragen danach Benutzernamen und Passwort für den Benutzer ein. Zum Abschluss müssen wir noch die Rechte definieren. (4) Hierzu ist es am einfachsten eine Vorlage auf dem sich öffnenden Menü zu wählen. Für unsere Aufgabe benötigen wir "read and write to any database" als Zugriffsrecht.

Wenn wir alles eingestellt haben, können wir am Ende der Seite auf "Add User klicken".

Jetzt haben wir die Möglichkeit links im Menü weiter oben auf "Database" zu klicken und in der Liste unserer Datenbanken auf "Connect" zu klicken, um den Verbindungsstrang zu erhalten.

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with various options like Overview, Deployment, Database (which is selected and highlighted in green), Data Lake, Services, Triggers, Data API, Data Federation, Search, Security, Quickstart, Backup, Database Access, Network Access, and Advanced. In the main content area, it says 'Database Deployments' and 'LORANT'S ORG - 2023-03-04 > SHOOT'. There's a search bar 'Find a database deployment...'. Below that is a 'Connect' button with a red box around it. Other buttons include 'View Monitoring', 'Browse Collections', and '...'. To the right of the 'Connect' button, there are sections for 'Visualize Your Data', 'Connections', 'Network Activity', and 'Data Size'. At the bottom, there's a table with columns: VERSION (6.0.6), REGION (AWS N. Virginia (us-east-1)), CLUSTER TIER (M0 Sandbox (General)), TYPE (Replica Set - 3 nodes), BACKUPS (Inactive), LINKED APP SERVICES (Multiple applications linked), ATLAS SQL (Connect), and ATLAS SEARCH (Create Index). A 'FREE' badge is on the right.

Im neu erscheinenden Menü wählen wir als Erstes die Verbindungsmöglichkeit, die wir nutzen möchten. Hier klicken wir auf "Drivers", um eine direkte API (application programming interface) Verbindung aufzubauen zu können. Im nächsten Fenster müssen wir dann Node.JS als Framework wählen, und die letzte Version anwählen. Weiter unten im Bild erscheint dann unser Link, wo wir nur noch den bei Network Access eingestellten Benutzernamen und das Passwort an der markierten Stelle einsetzen müssen. In der URL sehen wir auch den Namen unserer Datenbank. Dieser befinden sich direkt nach dem @ in der Zeile. Im beigefügten Screenshot wäre das "photogulasch". Der Aufbau hierbei ist immer gleich: Benutzer:password@Datenbank. Wichtig ist hierbei das / Zeichen, da es das Dokument innerhalb der Datenbank zeigt. In unserem Fall ist es Weba

Die Vollständige Verbindung URI würde lauten, später werden wir als **MONGO_URI** in der *Environment Variable* auf diesen String referenzieren:

```
mongodb+srv://vercel-admin-user:  
<password>@photogulasch.j841hex.mongodb.net/Weba?retryWrites=true&w=majority
```

Hier einfach "vercel-admin-user", "password", und Dokument Name tauschen und durch eigene Werte ersetzen.

The screenshot shows the MongoDB Atlas connection setup process. Step 1: Set up connection security (checkmark). Step 2: Choose a connection method (highlighted by a red arrow pointing to the 'Drivers' section). Step 3: Connect.

Connect to your application

Drivers Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.) >

Access your data through tools

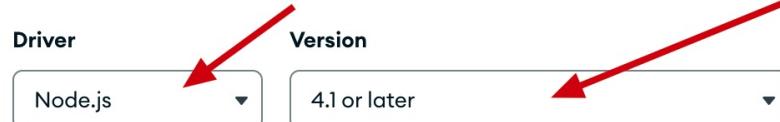
Compass Explore, modify, and visualize your data with MongoDB's GUI >

Shell Quickly add & update data using MongoDB's Javascript command-line interface >

Connecting with MongoDB Driver

1. Select your driver and version

We recommend installing and using the latest driver version.



2. Install your driver

Run the following on the command line

```
npm install mongodb@4.1
```



[View MongoDB Node.js Driver installation instructions.](#)

3. Add your connection string into your application code

View full code sample

```
mongodb+srv://<username>:<password>@photogulasch.j841hex.mongodb.net/?retryWrites=true&w=majority
```



Replace **<password>** with the password for the **<username>** user. Ensure any option params are [URL encoded](#).

#####
#####

Initialisierung eines neuen Projektes

Um ein neues Projekt zu starten, müssen die folgenden Schritte berücksichtigt werden.

Starten wir die virtuelle Maschine oder öffnen das WSL Terminal.

Wechseln wir in den Arbeitsordner, in dem das Projekt angelegt werden soll. Das kann z.B. der Ordner sein, in dem alle Github Projekte abliegen.

Hier verwenden wir den folgenden Befehl:

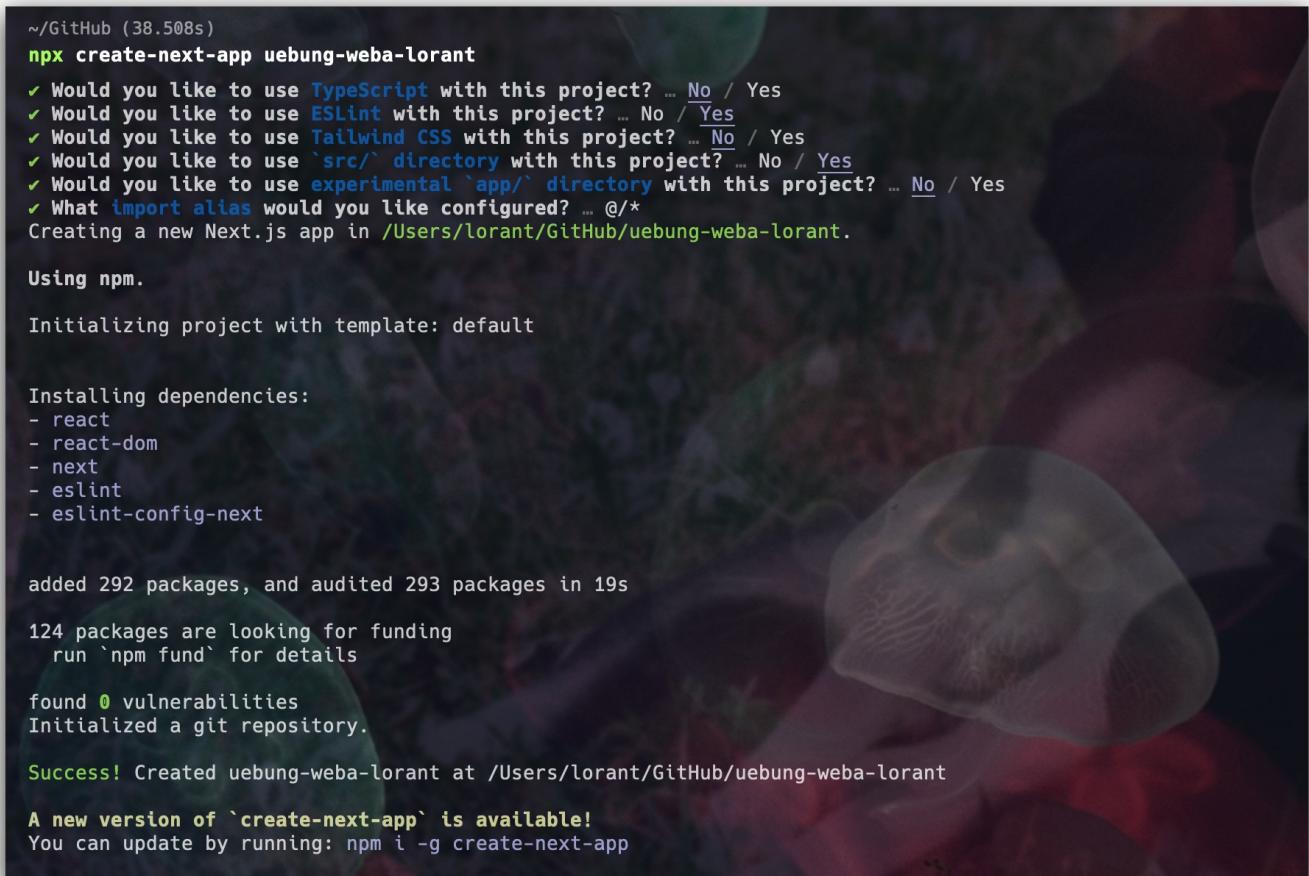
```
cd "xx"/Github/
```

Wobei xx für die Ordnerstruktur steht, wo der Ordner abgelegt ist.

Im Ordner führen wir den folgenden Befehl aus.

```
npx create-next-app weba-lorant-weba
```

Hierbei ist **npx** der Befehl, **create-next-app** der Parameter und "**weba-uebung-lorant**" der Projektnname. Nachdem Ausführen des Befehls werden einige Fragen zur Art des Projektes gestellt. Hierbei wird z.B. abgefragt, ob das Projekt Typescript verwenden soll, Tailwind Css verwendet wird und ähnliches. Im folgenden Bild sind die passenden Einstellungen für das aktuelle Projekt abgebildet. Hier sieht man auch die Ausgabe des Befehls in der Konsole.



```
~/GitHub (38.508s)
npx create-next-app uebung-weba-lorant
✓ Would you like to use TypeScript with this project? ... No / Yes
✓ Would you like to use ESLint with this project? ... No / Yes
✓ Would you like to use Tailwind CSS with this project? ... No / Yes
✓ Would you like to use `src/` directory with this project? ... No / Yes
✓ Would you like to use experimental `app/` directory with this project? ... No / Yes
✓ What import alias would you like configured? ... @/*
Creating a new Next.js app in /Users/lorant/GitHub/uebung-weba-lorant.

Using npm.

Initializing project with template: default

Installing dependencies:
- react
- react-dom
- next
- eslint
- eslint-config-next

added 292 packages, and audited 293 packages in 19s

124 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
Initialized a git repository.

Success! Created uebung-weba-lorant at /Users/lorant/GitHub/uebung-weba-lorant

A new version of `create-next-app` is available!
You can update by running: npm i -g create-next-app
```

Nachdem das Projekt erstellt wurde, kann auch schon das passende Git Repository erstellt werden. (*Wir haben die erweiterte Version von Git verwendet. (Die Erweiterung Git Flow ist hierbei aber Optional. Sie dient dazu verschiedene Stränge für die Entwicklung bereitzustellen.)*)

Zuerst mit `cd "Projektname"` in das entsprechende Verzeichnis wechseln und den folgenden Befehl ausführen.

```
git flow init
```

Anbei die Ausgabe des Befehls, mit den möglichen Parametern, die man angeben kann.

```
~/GitHub (0.045s)
cd uebung_lorant_weba/

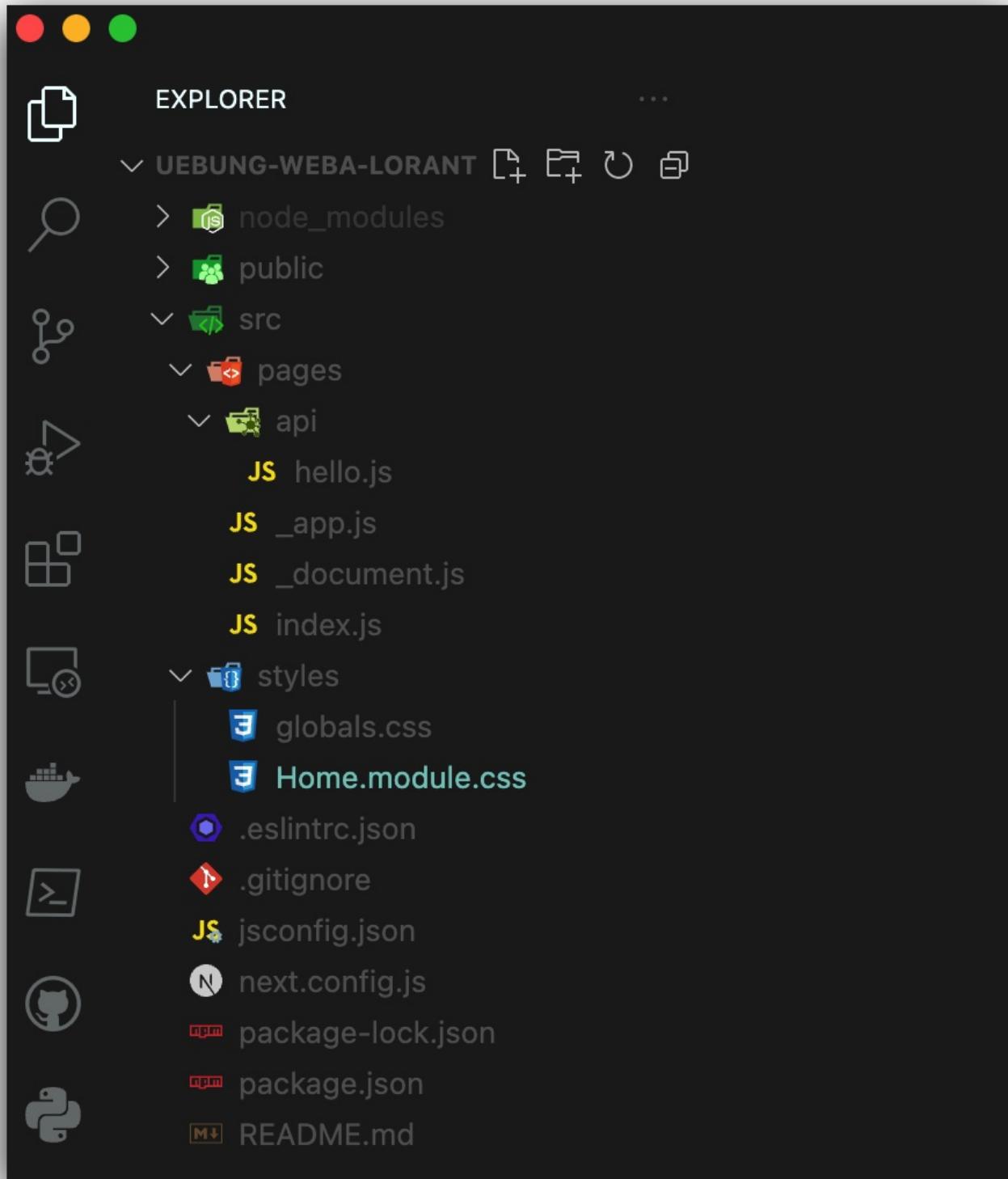
~/GitHub/uebung_lorant_weba git:(main) (19.822s)
git flow init

Which branch should be used for bringing forth production releases?
- main
Branch name for production releases: [main]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []

~/GitHub/uebung_lorant_weba git:(develop)
```

Wenn alles erfolgreich geklappt hat, dann sollten folgende Dateien zu sehen sein, wenn man den neu angelegten Ordner mit Visual Studio Code öffnet.



Wenn wir dieses Bild sehen, können wir die Applikation zum ersten mal starten.

Start den Webserver

Um den Webserver zu starten müssen wir sicherstellen, dass wir im Projektordner sind. Das können wir ganz leicht mit: `pwd` prüfen.

Sollten wir wiedererwarten nicht im richtigen Ordner sein, bitte zuerst in den Projektordner wechseln.

Wenn das erledigt ist, dann kann man folgenden Befehl ausführen:

npm run dev

Hiermit wir eine Entwicklungsinstanz auf unseren Rechner gestartet. Wenn der Befehl ausgeführt wurde, bekommen wir eine Benachrichtigung, dass der Server auf <http://localhost:3000> gestartet wurde. Und ggf. Fehlermeldungen, die eine Ausführung behindern.

```
~/GitHub/uebung-weba-lorant git:(main)
npm run dev

> uebung-weba-lorant@0.1.0 dev
> next dev

- ready started server on 0.0.0.0:3000, url: http://localhost:3000
- info Loaded env from /Users/lorant/GitHub/uebung-weba-lorant/.env.local
- event compiled client and server successfully in 401 ms (171 modules)
- wait compiling...
- wait compiling /_error (client and server)...
- event compiled client and server successfully in 229 ms (173 modules)
```

Wenn alles richtig gemacht wurde, bekommen wir in der letzten Zeile die Nachricht, dass "compiled client and server successfully"

Wir haben es geschafft, wir haben unsere erste eigene Next.Js App angelegt

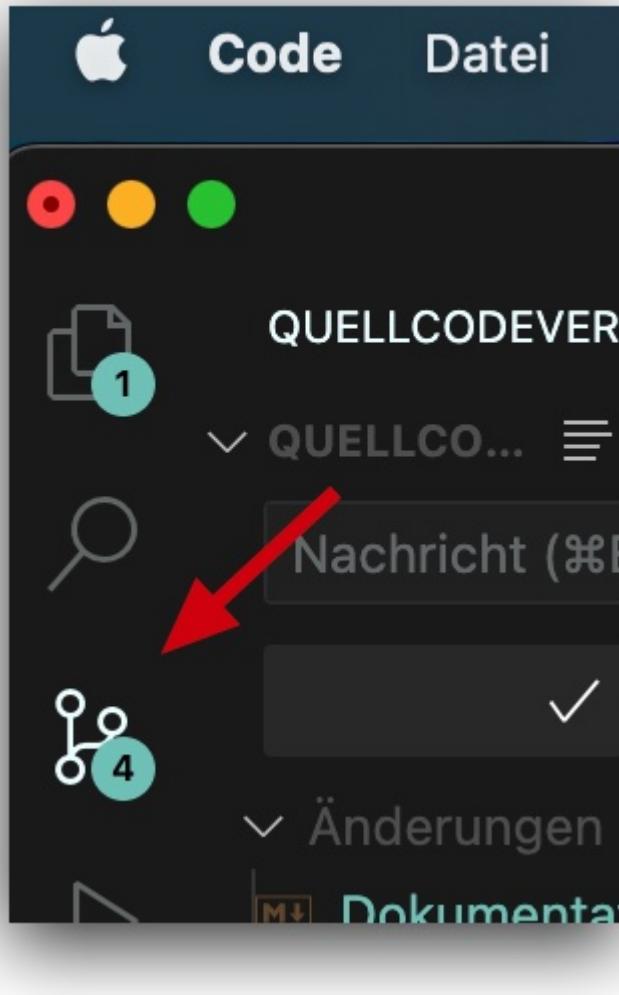
Wir können die Webseite öffnen, indem wir die Localhost Adresse in unseren Browser öffnen

Bevor wir mit den Anpassungen der Seite beginnen, sollten wir zuerst die Verbindung mit Vercel anlegen, damit später die Aktualisierung der App direkt aus VsCode passieren kann.

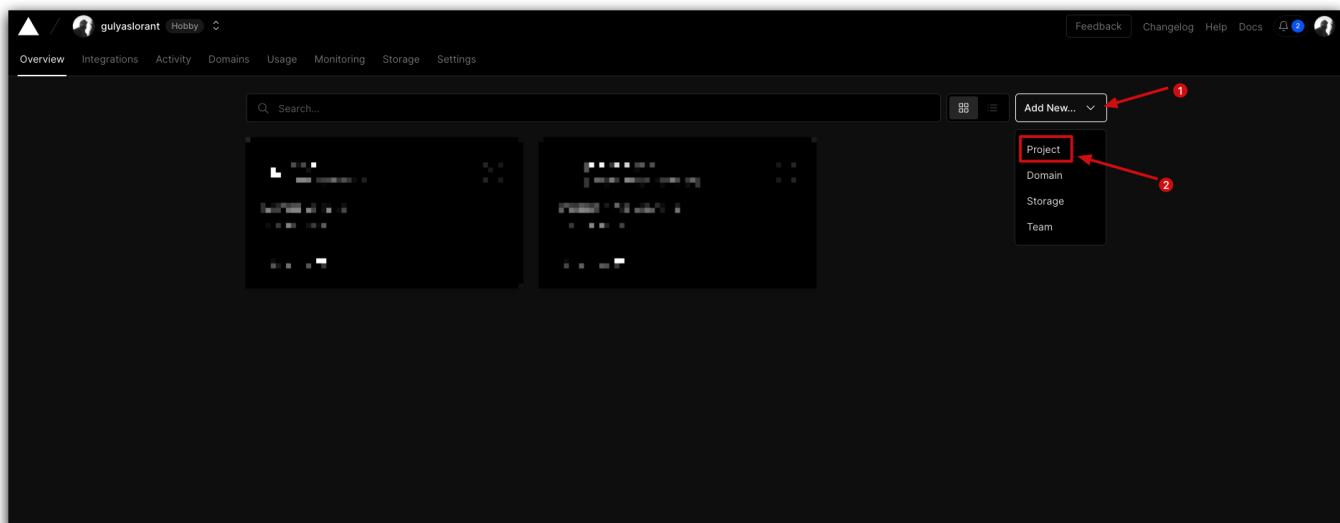
Da die genaue Verbindung von Github und Vs Code ein eigenes Kapitel darstellen würde, wird hier die offizielle Anleitung verlinkt, wo genau beschrieben ist, welche Schritte man durchführen muss.

[Github Verbindung mit Vs Code](#)

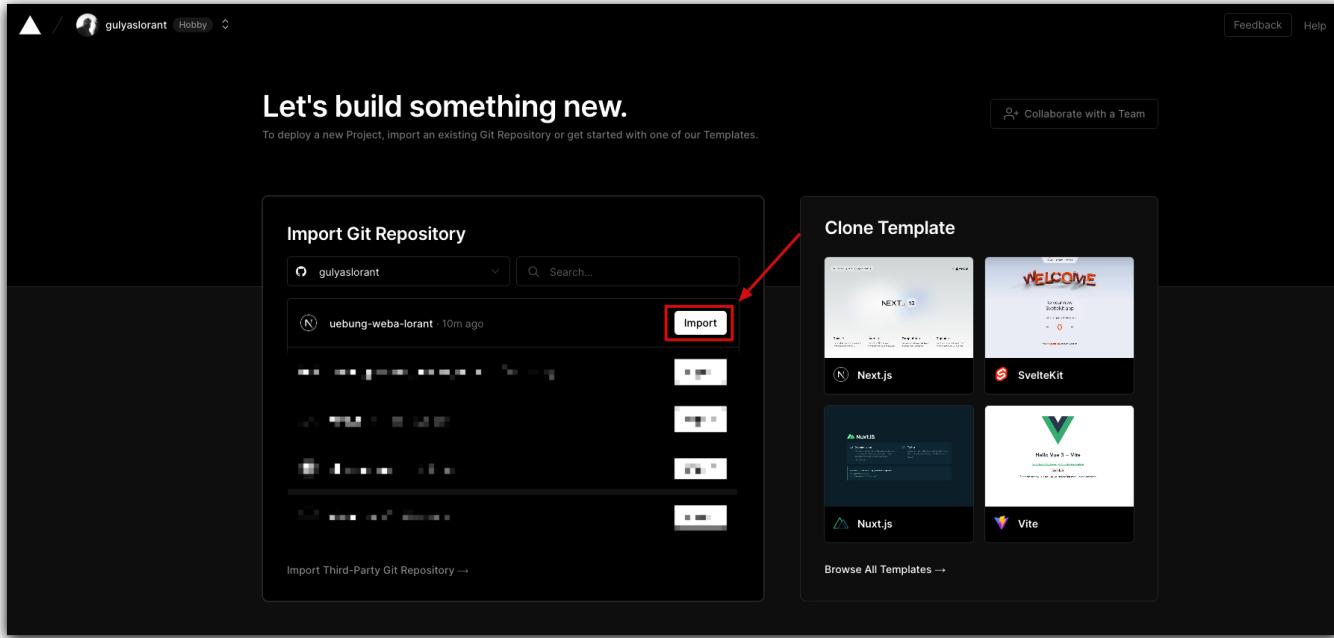
Wenn alles gepasst hat, sollten wir auf der linken Seite, beim Punkt Quellencode Verwaltung eine Zahl sehen -> Diese zeigt an, wie viele Dateien mit dem Server zu synchronisieren sind. Den Menüpunkt anklicken, eine Nachricht verfassen, um für später das Update zuordnen zu können, und dann auf Commit klicken. Damit wurde das Paket mit Github synchronisiert.



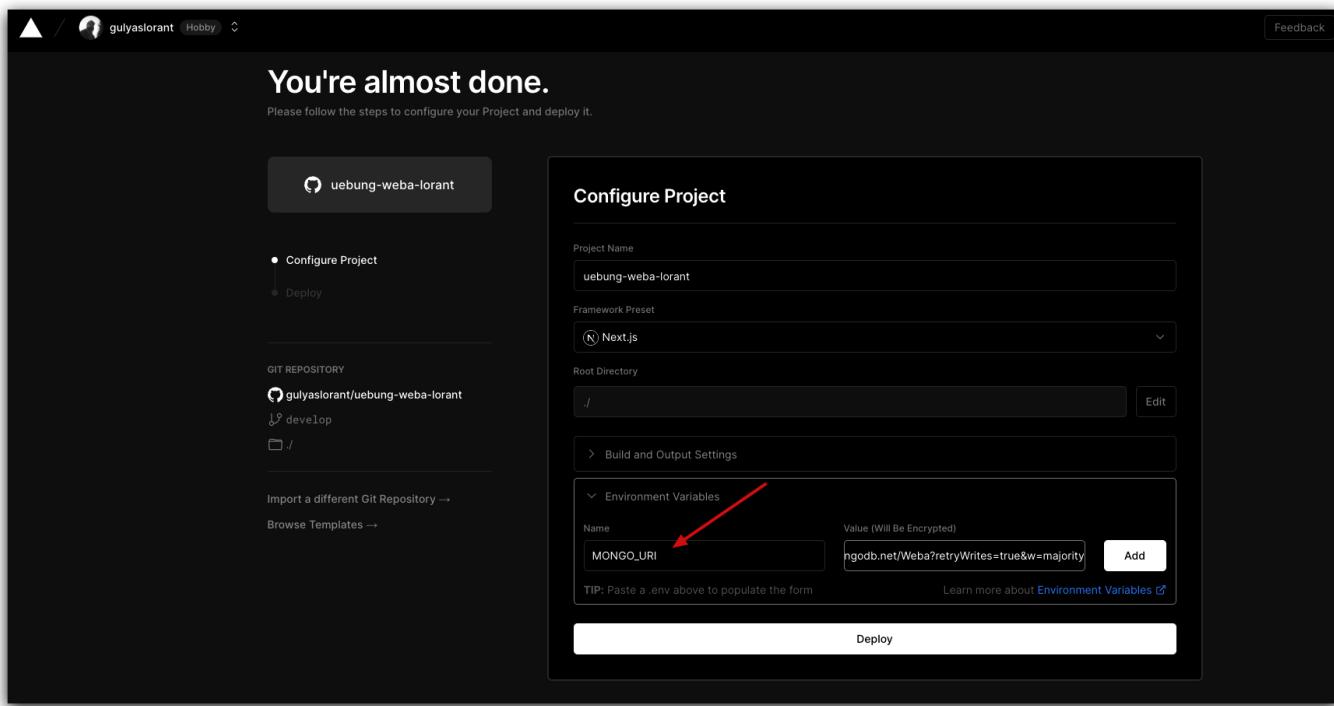
Nachdem Sync mit Github, müssen wir das Projekt noch auf Vercel veröffentlichen. Sofern wir bei der Accountregistrierung alles richtig gemacht haben, können wir das Github Projekt sehr leicht anlegen. Im Dashboard von Vercel klicken wir hierzu auf **Add New -> Project** im rechten Bereich des Bildschirmes.



Auf dem nächsten Bildschirm erscheint eine Liste unserer Github Repository, mit jeweils kleinen Symbolen, sofern der passende Typ gefunden wurde. Hier erscheint jetzt unser neues Projekt, mit einem kleinen (N) Logo als Symbol für ein Next.JS System. Einfach auf Import neben dem Namen klicken, um das Deployment zu starten.



Am folgenden Bildschirm können wir einige notwendige Details einstellen, und auch Environment Variables anlegen. Diese benötigen wir später auch für die Datenbank Anbindung. Hierzu einfach in den Punkt "Build and Output Settings" auf Environment Variables klicken. Hier können wir neue hinzufügen. Unsere Variable hat den Namen "MONGO_URI" und der Wert ist der weiter oben bei MongoDB definierte ZugriffsURL. Danach einfach auf Add klicken, um diese dem Projekt hinzuzufügen.



Bevor wir weitermachen, sollten wir einen kurzen Blick auf die wichtigsten Komponenten des Projektes werfen.

- **/src/pages/index.js** --> Das ist unsere Startseite, die im Browser geöffnet wird, wenn wir die URL aufrufen.
- **/src/styles/global.css** --> Hier werden alle Designangaben getätigt, die sich Global auswirken sollen
- **package.json** --> Hier werden alle Komponenten gelistet, die in der Instanz laufen und benötigt werden.

- **/public** --> Hier sind alle Dateien, die frei zugänglich sein sollen.

Unser Ziel ist es, eine eigenständige Anwendung zu erlangen, und hierdurch auch ein besseres Verständnis für den Aufbau einer Next.JS Seite zu bekommen.

Unser Ziel ist es eine einfach Todo Liste zu erstellen, also eine Liste, auf der über ein Eingabefeld neue Elemente eintragen können, und diese dann direkt auf der Webseite angezeigt werden. Die angelegten Daten, die wir später auch anzeigen möchten, werden hierfür auf MongoDB gespeichert, und von hier abgerufen.

Datenbank Verbindung erzeugen

Wir starten den Aufbau unserer Seite mit dem erstellen der Verbindung zu MongoDB.

Hierzu erstellen wir zuerst einen Ordner **libs** im src Ordner unseres Projektes. In diesem Ordner erstellen wir eine Datei, die wir MongoConnect.js nennen. Diese Datei wird beim aufrufen der Seite die Verbindung zum Server herstellen und halten.

Folgender Code muss in die Datei eingefügt werden.

```
/** @format */

import mongoose from "mongoose";

export const connectMongoDB = async () => {
  if (mongoose.connection.readyState === 1) {
    return mongoose.connection.asPromise();
  }
  return await mongoose.connect(process.env.MONGO_URI);
};
```

Wenn wir den Code 1:1 in die Datei kopieren, werden wir eine Fehlermeldung bekommen, dass mongoose nicht gefunden werden konnte. Mongoose ist ein Paket, welches wir verwenden, um eine Verbindung zum Mongo Server zu ermöglichen.

Um Mongoose zu installieren müssen wir im Terminal den folgenden Befehl im Projektordner absetzen.

```
npm install mongoose
```

Wenn die Installation abgeschlossen ist, können wir unseren Arbeitsbereich aktualisieren, um direkt mongoose verwenden zu können.

Hierzu einfach **Shift+Strg+P** oder **Command+Shift+P** drücken, oder auch alternativ im Menü Anzeigen->Befehlspalette anwählen und

```
reload
```

tippen. Hiermit haben wir den Arbeitsbereich neu geladen, und die Module werden auch erkannt.

In unserer Datei erstellen wir eine Variable, die als Wert den Status der Verbindung haben wird. Das ganze wird über eine Asynchrone Funktion abgerufen. Hierbei überwachen wir die Verbindung. Wenn diese als

Status ===1 hat, also wahr ist, dann wird sie weitergeben, ansonsten mongoose.connect mit der Umgebungsvariable **MONGO_URI** Wir erinnern uns, beim erstellen der Mongo Datenbank haben wir den Verbindungstext erzeugt. Diesen werden hier jetzt verwenden.

Wir erstellen eine Datei im Hauptverzeichnis des Projektes. Diese benennen wir **.env.local**

Diese Datei wird nur eine einzige Zeile enthalten.

MONGO_URI="mongodb+srv://..." (Hier bitte den Vollständigen String eintragen)

Wir können jetzt die Datei speichern und schliessen.

Datenmodel erzeugen

Bei MongoDB werden Datenmodelle definiert, mit deren Hilfe wir strukturiert Daten in unserer Datenbank ablegen können. Das ist quasi die Grundlage der späteren Operationen zum auslesen und hochladen der Daten.

Hierzu erstellen wir in unseren *src* Ordner einen *models* Unterordner. Hier erstellen wir dann eine Datei mit dem Namen **taskModel.js**

Folgenden Code kopieren wir in die Datei

```
/** @format */

import mongoose from "mongoose";

const taskSchema = new mongoose.Schema({
  task: {
    type: String,
    required: true,
  },
});

const Task = mongoose.models.Task || mongoose.model("Task", taskSchema);

export default Task;
```

In dieser Datei verwenden wir wieder Mongoose als Verbindungsmodul.

Erstellen ein Variable mit dem Namen taskSchema, das beinhaltet ein mongoose.Schema, mit dem Wert *task*, welches vom Typ ein String ist, und verpflichtend angegeben werden muss.

Aus dem Schema erstellen wir dann eine Variable *Task* welches unser Model sein wird. Als erstes definieren wir, dass *Task* ein Model ist, um danach anzugeben, dass das Task model aus dem taskSchema besteht.

Zum Schluss exportieren wir das Model aus Rückgabewert.

Nachdem wir auch unser Model definiert haben, können wir unsere API Routen erstellen.

API zu MongoDB für Abfrage der Daten

Unsere API enthält 2 Routen, die wir anlegen müssen. Sowohl die Abfrage als auch Eintrag muss natürlich dargestellt werden.

Fangen wir als erstes mit der Abfrage an. Hierzu erstellen wir in `/src/pages/` einen neuen Unterordner mit dem Namen `api`. In diesem Ordner erstellen wir eine neue Datei mit dem Namen `get_task.js`

```
/** @format */

import { connectMongoDB } from "@/libs/MongoConnect";
import Task from "@/models/taskModel";

export default async function handler(req, res) {
  if (req.method !== "GET") {
    res.status(405).send({ msg: "Only GET requests are allowed" });
    return;
  }
  const { task } = req.body;

  try {
    await connectMongoDB();
    const tasks = await Task.find();
    res.status(200).send(tasks);
  } catch (err) {
    console.log(err);
    res.status(400).send({ err, msg: "Something went wrong" });
  }
}
```

In dieser Datei prüfen wir die Art der Anfrage, die über das Frontend gesendet werden, um senden entsprechende Statusmeldungen weiter. In diesem Beispiel prüfen wir, ob der gesendete Status "GET" ist. Sofern es ein anderer Wert ist, sendet die API eine Rückmeldung, dass nur GET Befehl gestattet sind.

Sofern der Korrekte Befehl empfangen wurde, wird die Variable `tasks` erstellt, mit den Werten der Datenbank. Hierzu wird der ein Status 200 mit `tasks` versendet. Wegen der asynchronen Funktnio muss alles in Try & Catch verschachtelt sein.

Wenn wir unsere API für die Abfrage erstellt haben, können wir das gleiche für das Hochladen von Dateien machen.

API Schnittstelle zum hochladen von Daten.

Ähnlich, wie bei der Abfrage werden wir auch hir einen Status an die Datenbank senden, nur in diesem Fall verpackt mit dem Inhalt, den wir hinterlegen möchten. Hierbei hilft uns das Datenmodel, dass wir schon erstellt haben, denn hierdurch können die Daten entsprechend hinterlegt werden. Als erstes erstellen wir eine neue Datei in `/src/pages/api/`, die sein wird `set_task.js`

Folgender Code muss in der Datei hinterlegt werden.

`src/pages/api/set_task.js` 1:

```
/** @format */

import { connectMongoDB } from "@/libs/MongoConnect";
import Task from "@/models/taskModel";

export default async function handler(req, res) {
  if (req.method !== "POST") {
    res.status(405).send({ msg: "Only POST requests are allowed" });
    return;
  }
  const { task } = req.body;

  try {
    await connectMongoDB();
    Task.create({ task }).then((data) => {
      console.log(data);
      res.status(201).send(data);
    });
  } catch (err) {
    console.log(err);
    res.status(400).send({ err, msg: "Something went wrong" });
  }
}
```

Hier haben wir eine ähnliche Aufstellung, wie bei der vorherigen Api. Der Unterschied ist hierbei nur, dass diesmal auf den Status *POST* geprüft wird. Sofern der passende Status gesendet wurde, wird innerhalb der Funktion der Status 201 an den Server gesendet, was es uns ermöglicht, Daten mitzusenden, die eingefügt werden können. Der Rest der API ist ähnlich aufgebaut. Eine Asynchrone Funktion, die auf eine Eingabe wartet, danach eine in Try & Catch verpackte Statusübermittlung.

Die Startseite

Nachdem wir alle Vorbereitungen getroffen haben, können wir uns jetzt der Startseite widmen. Das ist die Seite, die der Besucher der Seite sehen wird. Auf dieser Seite haben wir ein Bild, einen Titel, ein Eingabefeld, sowie die Liste der Aufgaben, die wir schon eingetragen haben. Zum Schluss werden wir auch noch eine kleine Animation einfügen, damit die Seite etwas Dynamischer wirkt.

Wir bearbeiten jetzt die Datei `/src/pages/index.js`

Mit der Erstellung der App wurde ja schon eine Standard Startseite angelegt. In dieser können wir jetzt getrost alle Zeilen löschen. Und mit folgenden Code ersetzen.

`src/pages/index.js` 1:

```
/** @format */
```

```
import Head from "next/head";
import { useEffect, useState } from "react";
import axios from "axios";
import AnimatedText from "@/components/AnimatedText";
import Bild from "../../public/Bild01.jpg";
import Image from "next/image";

export default function Home() {
  const [input, setInput] = useState("");
  const [task, setTask] = useState([]);

  const handleSubmit = (e) => {
    e.preventDefault();

    axios
      .post(`/api/set_task`, { task: input })
      .then(() => {
        console.log(res);
        setInput("");
      })
      .catch((err) => console.log(err));
  };

  useEffect(
    () => {
      axios.get(`/api/get_task`).then((res) => {
        setTask(res.data);
        console.log(res.data);
      });
    }
  );
  return (
    <>
      <Head>
        <title>Todo Liste für Web Architekturen</title>
        <meta name="description" content="Todo Listen Einträge mit Next.JS" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
      </Head>
      <main>
        <AnimatedText text="Das ist eine kleine To Do Liste, auf der man einträge posten kann" />
        <Image src={Bild} alt="Moderne Todo" />
        <form onSubmit={handleSubmit}>
          <input
            type="text"
            value={input}
            onChange={(e) => setInput(e.target.value)}
          />
          <button type="submit">Send</button>
        </form>
        <ul>

```

```

        {task.map((t) => (
            <li key={t._id}>{t.task}</li>
        )));
    </ul>
</main>
</>
);
}
}

```

Hierbei fallen einige neuen Befehle auf, die wir hier kurz erläutern müssen. Zuerst die Module, die importiert werden müssen.

- **import Head from "next/head";** -> Hiermit Importieren wir die Kopfzeile der HTML Seite, wo wir Informationen, wie Titel und Meta hinterlegen können.
- **import { useEffect, useState } from "react";** -> Wird benötigt, um den Zustand von Variablen zu ändern.
- **import axios from "axios";** -> Benötigen wir zum Verbinden mit der Datenbank
- **import AnimatedText from "@/components/AnimatedText";** -> Der Animationseffekt, die wir später detailliert beschreiben.
- **import Bild from "../public/Bild01.jpg";** -> Das Bild, welcher wir auf der Startseite darstellen werden.
- **import Image from "next/image";** -> Das Bildmodul zum darstellen von Bildern.

Das Modul Animatedtext werden wir selber erstellen. Axios müssen wir installieren. Zusätzlich müssen wir noch die /src/styles/globals.css anpassen, und die gewünschte Optik zu erlangen.

Hierzu verwenden den schon bekannten npm install Befehl.

```
npm install axios
```

```

> npm install axios

added 9 packages, and audited 323 packages in 1s

126 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

```

Die Index Datei besteht nach den Imports am Anfang, danach definieren wir die Funktion **Home()**, welche wir später auch an den Webserver übermitteln werden.

Danach definieren wir die Variabln, welche wir für die Datenübermittlung benötigen. Hier kommt auch Axios zum Einsatz, eine vereinfachte Methode, um Daten an MongoDB zu übermitteln. das ist der **handleSubmit** Abschnitt.

Hier werden wir nach Druck des "Senden" Buttons die Daten an die API übergeben, und zugleich das Eingabefenster wieder leeren.

Zusätzlich definieren wir eine Funktion, um die Einträge abzufragen. Die so erstellte Array übermitteln wir dann an die Seite.

Die dargestellte Seite an sich finden wir nach dem Wort **return** zwischen den **<> </>** Zeichen.

Hier haben wir die Head Sektion, wo wir Titel und Meta Daten haben. Danach kommt der *main* Abschnitt.

In diesen Abschnitt wird die Seite abgebildet. Hier haben die Funktion Animatedtext mit dem Textparameter übergeben. Hierzu etwas später mehr. Danach kommt das Bild, welches wir darstellen.

Hier müssen wir beim parameter **src=""** das Bild angeben. Dieses haben wir ja schon als Variable beim Import deklariert und können hier einfach den Namen aus dem Import verwenden. Bitte hierbei immer beachten, dass man die entsprechenden Pfade bei den Imports angibt. Darum sehen wir hier auch **..../public** Das ist, weil wir aus dem /src/pages/ Ordner um 2 Ebenen weiter rauf müssen um in den Public ordner zu kommen. Zusätzlich haben wir hier noch die Möglichkeit eine Beschriftung mit dem **alt=""** Parameter mitzugeben. Die Größe des Bildes werden wir später über globals.css einstellen. Wichtig ist es, dass hier eine eigene Datei eingetragen wird.

Als nächstes Element haben wir die Form, welche wir verwenden, um den Text abzufragen, welchen wir übermitteln werden. In der CSS Datei haben wir das Aussehen definiert. Wir benötigen ein Eingabefeld sowie einen Button, welchen wir drücken können. Beim Drücken des Buttons werden die Daten übermittelt und die Axios Funktion aktiviert.

Nach der Form kommt eine Abfrage der schon bestehenden Einträge. Hierzu lassen wir eine Schleife beim der Task variable laufen, welche wir bereits definiert haben, mit Map lassen wir die Daten einzeln in eine Variable übergeben, welche wir dann darstellen können.

Besonders wichtig hierbei ist, dass wir im li Element eine key="t._id" angeben. Damit können wir sicherstellen, dass unsere Schleife sieht, ob der Eintrag schon abgebildet wurde.

Wenn wir alles erstellt haben, sollten wir unsere Seite schon Abbilden können, auch wenn sie noch nicht besonders schön ist.

Jetzt können wir unsere Animation erstellen.

Hierzu müssen wir eine neue Datei erstellen. Zuerst erstellen wir einen Unterordner im src Ordner, den wir **components** nennen. In dem neu erstellten Ordner werden wir eine Datei anlegen. Diese nennen wir **AnimatedText.js**

Folgender Code können wir direkt in die Datei kopieren.

src/components/AnimatedText.js 1:

```
/** @format */

import React from "react";
import { motion } from "framer-motion";
```

```
const quote = {
  initial: {
    opacity: 1,
  },
  animate: {
    opacity: 1,
    transition: {
      delay: 0.5,
      staggerChildren: 0.2,
    },
  },
};

const singleWord = {
  initial: {
    opacity: 0,
    y: 50,
  },
  animate: {
    opacity: 1,
    y: 0,
    transition: {
      duration: 1,
    },
  },
};

const AnimatedText = ({ text }) => {
  return (
    <div>
      <motion.h1
        variants={quote}
        initial="initial"
        animate="animate"
      >
        {text.split(" ").map((word, index) => (
          <motion.span
            key={word + "-" + index}
            variants={singleWord}
          >
            {word}&nbsp;
            </motion.span>
          )));
      </motion.h1>
    </div>
  );
};

export default AnimatedText;
```

Für diese Animation müssen wir das framer-motion Paket installieren.

```
npm install framer-motion
```

Wenn das Paket installiert ist, können wir den Arbeitsbereich neu laden, und das Modul verwenden. Das besondere an dem Modul ist, dass wir hiermit die normalen Html Elemente animieren können, wenn wir motion. vor die Blöcke geben. So z. B. motion.span. Um die Effekte richtig verwenden zu können, müssen wir das verhalten der Animation anhand vo Parametern definieren. Die genauen Parameter kann man dem Code entnehmen. Wichtig ist es immer, dass wir einen Initial Wert definieren, also was der Status ist, welches das Element hat, wenn die Animation gestartet wird. Danach definieren wir, was animiert werden soll, und können auch z. B. angeben, welche Zeit die Animation beansprucht.

In unserem Beispiel animieren wir den Text an sich, und dann noch verschachtelt das versetze bewegen der einzelnen Wörter.

Wer genauere Details zu framer motion haben möchte, kann die offizielle Dokumentation zu Rate ziehen.

[Framer Motion Dokumentation](#)

Wenn wir alles richtig eingestellt haben, können wir die Funktion auf unserer Index Seite direkt aufrufen, und den zu animierenden Text als Information weitergeben. Diese liefert dann den Text mit einer netten Animation.

Als letztes müssen wir noch die globals.css Datei in dem Ordner /src/styles editieren, damit die Darstellung auch entsprechend aussieht.

Die standardmäßig definierten Einstellungen können wir komplett löschen, da wir eigene Elemente haben, die wir anpassen müssen.

Folgende Werte werden wir benötigen:

src/styles/globals.css 1:

```
html {  
  background-color: rgb(251, 228, 198);  
}  
main {  
  max-width: 1000px;  
  margin: 0 auto;  
  display: flexbox;  
  background-color: hsl(39, 49%, 66%);  
  justify-items: center;  
}  
form {  
  display: flexbox;  
  gap: 20px;  
  font-size: 20px;  
}  
input, button {  
  padding: 10px 20px;  
}  
  
li {  
  padding: 10px 10px;
```

```

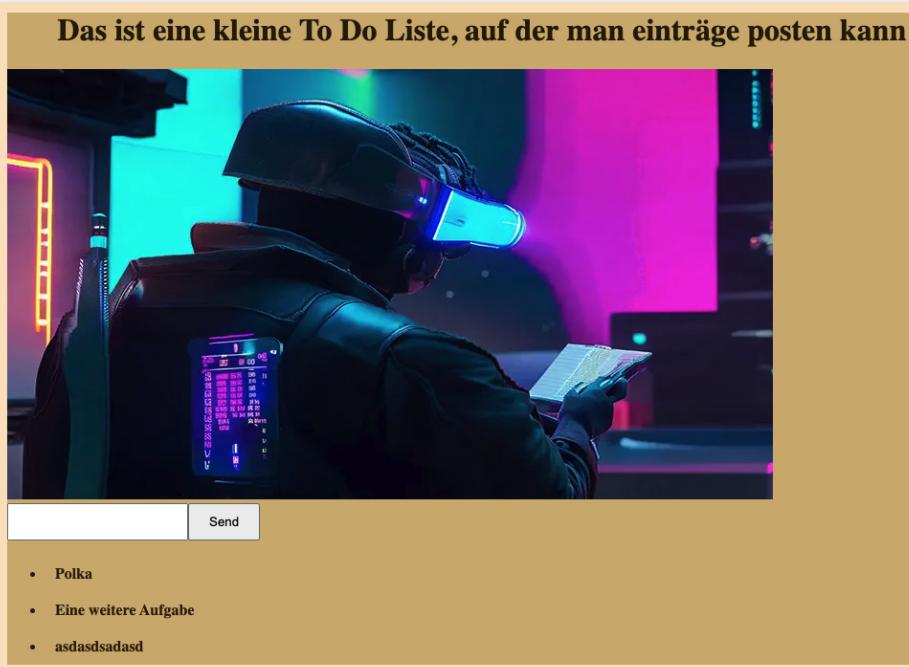
font-weight: 800;
color:rgb(54, 35, 6);
}

h1 {
  color:rgb(46, 32, 5);
  align-items: center;
  justify-content: center;
  text-align: center;
}
Image {
  width: auto;
  height: auto;
  max-width: 500;
}

```

Hier definieren wir das Aussehen für die `main`, die `form`, `input`, `button`, `li`, `h1`, und `Image` Komponenten. Alle hier eingestellten Änderungen werden direkt auf alle Seiten übernommen.

Wenn alles geklappt hat, sollte eine Seite wie folgt aussehen.



Sofern wir alle Verbindungen richtig erstellt haben, können wir die Daten einfach mit Github über Quellenverwaltung synchronisieren. Sobald die aktuelle Version bei Github aufliegt, wird es auch automatisch an Vercel gepusht. Vercel erstellt dann automatisiert ein neues Build, welches dann dem Endnutzer bereitgestellt wird.

Schlusswort

Der besondere Vorteil dieses Frameworks ist die dynamische Abbildung am Server. Der Endnutzer braucht sehr wenig Voraussetzungen am Browser selber. Die Komponenten werden beim generieren der Anwendung am Server hinterlegt. Da vorwiegend Javascript verwendet wird, kann sehr leicht eine

Interaktive Seite aufgebaut werden, welche die Besucher zum mitmachen animieren kann. Das kann man auf vielen Anwendungsgebieten gut verwenden. Besonders im UI/UX Bereich wird es immer häufiger verwendet. Zusätzlich sind viele große Plattformen, wie **Twitch**, **Tiktok** oder auch **Netflix** auf Basis von Node aufgesetzt.

Eventuelle Fehler bitte im Issues Abschnitt auf der Github Repo hinterlassen.