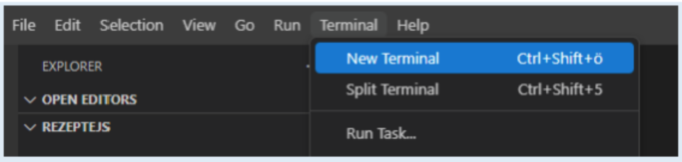
Neuen Projektordner erstellen, diesen in VSCode öffnen und dort das Terminal öffnen



Projekt initialisieren

```
npm init -y
```

Frameworks/Module installieren, in diesem Fall

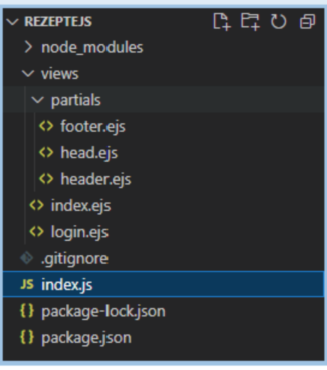
- Express
- EJS

```
npm install express
npm install ejs
```

Diese Module werden im Unterordner node_modules abgelegt. Es empfiehlt sich, diesen Ordner **nicht** mit ins GitHub Repository aufzunehmen. Dies erreicht man mit einer Datei mit dem Namen *.gitignore*.
Klont man das Repo lokal neu, kann man mit `npm install` schnell alle Abhängigkeiten neu installieren, da diese in *package.json* hinterlegt sind.

.gitignore
/node_module

Projektstruktur anlegen, Ordner und Dateien siehe Screenshot



Controller

Inhalt **Index.js**

app.* "routet" die Anwendung
Möglich u.a. sind app.**get**, app **post**
Weitere Infos dazu:
Express-Middleware verwenden (expressjs.com)

res.render() gibt die jeweilige View aus.

Hier kommt das Prinzip der asynchronen Programmierung zum tragen mit sogenannten "Callback Functions" => Node.js — JavaScript Asynchronous Programming and Callbacks (nodejs.org)

index.js

```
const express = require('express');
const app = express();
const PORT = 3000;

// Set EJS as the view engine
app.set('view engine', 'ejs');

// Routes
app.get('/', (req, res) => {
    res.render('index');
});
// Start server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Inhalt **index.ejs**

Wir geben zunächst erst einmal unsere statische Rezepteseite aus.

Verwenden Sie hier Ihre eigene Lösung aus den vorherigen Unterrichtsstunden

index.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- Latest compiled and minified CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"
rel="stylesheet">
    <!-- Latest compiled JavaScript -->
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js">
</script>
    <title>Rezepte</title>
</head>
<body>
    <div class="container p-3 mt-3 bg-primary text-white text-center">
        <h1 id="rezepte">Rezepte</h1>
    </div>
    <div class="container p-3">
        <div class="row">
            <div class="col-lg-4 mt-3">
                <div class="card">
                    <div class="card-header">
                        <h2>Pizza</h2>
                    </div>
                    <div class="card-body">
                        <ul>
```

Server starten

Samstag, 9. November 2024   05:46

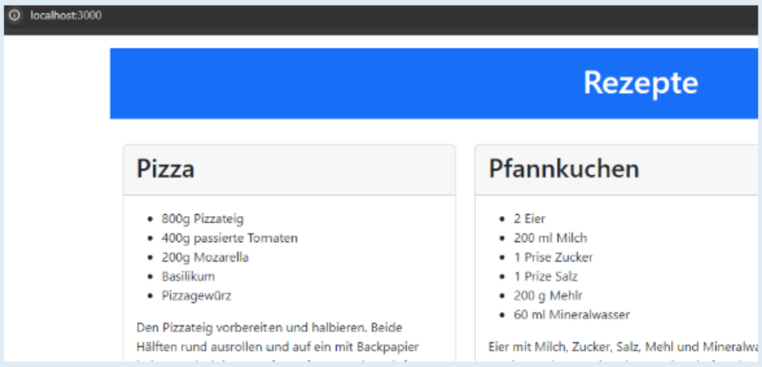
Server zum ersten Mal starten

Im Terminal:

```
node ./index.js
```

```
Server is running on http://localhost:3000
```

Im Browser:



localhost:3000

Rezepte

Pizza

- 800g Pizzateig
- 400g passierte Tomaten
- 200g Mozarella
- Basilikum
- Pizzagewürz

Den Pizzateig vorbereiten und halbieren. Beide Hälften rund ausrollen und auf ein mit Backpapier

Pfannkuchen

- 2 Eier
- 200 ml Milch
- 1 Prise Zucker
- 1 Prize Salz
- 200 g Mehlr
- 60 ml Mineralwasser

Eier mit Milch, Zucker, Salz, Mehl und Mineralwa

Mit "partials" arbeiten, um Meta-Tags, den Header und den Footer, welche ja in allen Seiten identisch sind, in alle Views einzubinden

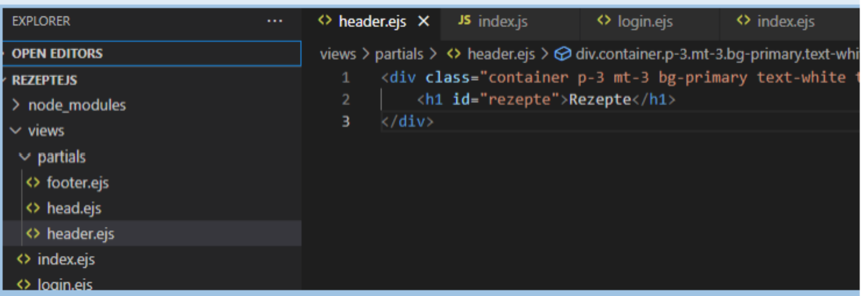
Inhalte des <head> auslagern in **paritals/head.ejs**

Hier könnte man auch weitere Tags wie <style> ergänzen

head.ejs

```html
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<!-- Latest compiled and minified CSS -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="style
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" inte
YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz" crossorigin="anonymous"></scr
<title>Rezepte</title>
```

header.ejs und footer.ejs dementsprechend mit Inhalt füllen



```html
<div class="container p-3 mt-3 bg-primary text-white t
    <h1 id="rezepte">Rezepte</h1>
</div>
```

Das Einbinden in die Views erfolgt dann wie hier beispielhaft an der index.ejs gezeigt

index.ejs

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <%- include('partials/head'); %>
</head>
<body>
    <%- include('partials/header'); %>
    <div class="container p-3">
        <div class="row">
            <div class="col-lg-4 mt-3">
```

Model

Prisma installieren und initialisieren

```
npm install prisma
npx prisma init
```

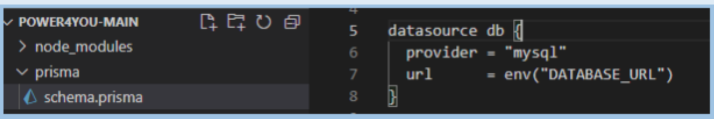
Im root des Projektes befindet sich nun die Datei *.env*

Dort werden die Verbindungsinformationen hinterlegt

.env

```
DATABASE_URL="mysql://d041e67c:wit31rezepte@w012ac34.kasserver.com:3306/d041e67c?schema=public"
```

Den Provider in *./prisma/schema.prisma* auf ***mysql*** setzen



```
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}
```

Das Datenbankschema in ein Prisma-Schema umwandeln und einen Prisma-Client generieren

```
npx prisma db pull
npx prisma generate
```

In der Datei */prisma/schema.prisma* finden sich nun alle Datenbanktabellen als Prisma-Schema

```
model user {
  user_id  Int       @id
  username String    @db.VarChar(255)
  password String    @db.VarChar(255)
  rezepte  rezepte[]
}
```

In der index.js kann nun der Client initialisiert werden

Index.js

```js
const { PrismaClient } = require("@prisma/client");
const prisma = new PrismaClient();
```

In der index.js kann nun der Client initialisiert werden

Index.js

```
const { PrismaClient } = require("@prisma/client");
const prisma = new PrismaClient();
```

Über das Prisma-Objekt können nun Abfragen erstellt werden.

WICHTIG: Das await-Keyword kann nur innerhalb einer async-Function verwendet werden!

==> https://www.w3schools.com/js/js_async.asp

Beispielabfrage

```
app.get('/', async (req, res) => {
  const users= await prisma.user.findMany();
  console.log(users); //Gibt alle Daten der Tabelle User aus
  ...
```

Die Rückgabe erfolgt direkt als JSON-**Object:**

```
[
  { user_id: 1, username: 'Karl', password: 'Karl' },
  { user_id: 2, username: 'Carla', password: 'Carla' }
]
```

Bsp.:
`users[1].username` ergibt

```
Carla
```

Legen Sie sich in der Datenbank einen eigenen Benutzer in der Tabelle *user* an **(Passwort md5 !!!)**
https://w012ac34.kasserver.com/mysqladmin/
d041e67c:wit31rezepte

Ein Template für die Login-View: https://getbootstrap.com/docs/5.3/examples/sign-in/

In der index.js muss folgendes ergänzt werden, damit der Post der Loginseite ausgelesen werden kann.

index.js

```
var bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: false }));
```

In der Middleware kann dann `app.post` genutzt werden:

Index.js

```
app.post('/login', async (req, res) => {

     // Beispiel Zufriff auf Postdaten:
     var username = req.body.username;

}
```

Für die Logik der Benutzerprüfung eignet sich folgendes Konstrukt:

Index.js

```
try {
    var user = await prisma.user.findFirstOrThrow({
        //WHERE etc...
    })
    //Login erfolgreich. Redirect Hauptseite.
  } catch (error) {
    //Login nicht erfolgreich. Zurück zum login
  }
```

How to Implement Session Management in Node.js Applications

Session management is a crucial aspect of web application development, as it ensures that user data and preferences are stored securely and accurately. In this article, we will explore how to implement session management in Node.js applications.

What is session management?

Session management is the process of managing user sessions within a web application. A session is a period of time in which a user interacts with an application, typically starting when the user logs in and ending when they log out. Session management ensures that user data, preferences, and session-related information are securely stored and managed.

Implementing session management in Node.js applications

To implement session management in Node.js applications, you need to use a session management middleware. A middleware is a function that sits between the client and the server, processing requests and responses.

Installing and configuring session middleware

The first step in implementing session management in Node.js applications is to install and configure the session middleware. There are several session middleware options available for Node.js, including express-session, cookie-session, and session-file-store. You can install and configure these middleware options using npm.

To install express-session, we can run the following command:

```
npm install express-session
```

Code

```
const express = require('express');
const session = require('express-session');

const app = express();

app.use(session({
    secret: 'secret-key',
    resave: false,
    saveUninitialized: false,
  }));
```

In the above code sample, we have initialized the express-session middleware with the following configuration options:

- **secret:** This option is used to set a secret key for the session. The secret key is used to sign the session ID cookie to prevent tampering.
- **resave:** This option determines whether the session should be saved to the store on every request. Setting this option to false can improve performance.
- **saveUninitialized:** This option determines whether to save uninitialized sessions. Setting this option to false can improve performance

Initializing the session middleware

Once you have installed and configured the session middleware, the next step is to initialize it. Initialization involves creating a session object that stores user data and preferences. You can initialize the session middleware in your application's entry point, such as app.js or server.js.

Code

```
const session = require('express-session');
const app = express();
app.use(session({
  secret: 'secret-key',
  resave: false,
  saveUninitialized: false,
}));
app.get('/', (req, res) => {
  const sessionData = req.session;
// Access session data
});
```

In the above code sample, we have initialized the session middleware and accessed the session data using the req.session object.

Storing session data

The session middleware stores session data in the server's memory or a separate session store, such as a Redis database. When a user logs in, the session middleware creates a session object and assigns it a unique ID. The session ID is then stored in a cookie on the user's browser. The session middleware uses the session ID to retrieve the session data from the server or session store.

Code

```
app.post('/login', (req, res) => {
    const { username, password } = req.body;
  // Authenticate user
    if (isValidUser(username, password)) {
      req.session.isLoggedIn = true;
      req.session.username = username;
  res.redirect('/dashboard');
    } else {
      res.redirect('/login');
    }
  });
```

In the above code sample, we have stored session data for an authenticated user using the req.session object.

Managing session timeouts

To ensure that session data is not stored indefinitely, it is essential to manage session timeouts. Session timeouts determine how long a session can remain idle before it is invalidated. You can set a timeout for a session by configuring the session middleware. When a session timeout occurs, the session middleware deletes the session data from the server or session store.
We can set the session timeout using the maxAge option when initializing the session middleware. The maxAge option is expressed in milliseconds and determines the maximum age of a session.

Code

```
app.use(session({
    secret: 'secret-key',
    resave: false,
    saveUninitialized: false,
    cookie: { maxAge: 60000 } // session timeout of 60 seconds
  }));
```

In the above code sample, we have set the session timeout to 60 seconds using the maxAge option.

Destroying Sessions

When a user logs out or the session expires, we need to destroy the session to ensure that session data is not stored indefinitely. We can destroy a session using the req.session.destroy() method.

Code

```
app.get('/logout', (req, res) => {
    req.session.destroy((err) => {
      if (err) {
        console.log(err);
      } else {
        res.redirect('/login');
      }
    });
  });
```

In the above code sample, we have destroyed the session using the req.session.destroy() method.

Retrieving Session Data

To retrieve session data, we can access the req.session object. The req.session object is an object that contains session data.

Code

```
app.get('/dashboard', (req, res) => {
    const isLoggedIn = req.session.isLoggedIn;
    const username = req.session.username;
  if (isLoggedIn) {
      res.render('dashboard', { username });
    } else {
      res.redirect('/login');
    }
  });
```

In the above code sample, we have retrieved session data using the req.session object.