

07 NoSQL-Datenbanken (MongoDB)

Multimedia Engineering (PS siehe 36609b)

M.Sc. Nils Hellwig

Lehrstuhl für Medieninformatik

FAKULTÄT FÜR INFORMATIK UND DATA SCIENCE



Universität Regensburg

Überblick Themen

- Einleitung NoSQL Datenbanken
- MongoDB
- Mongoose

NoSQL Datenbanken

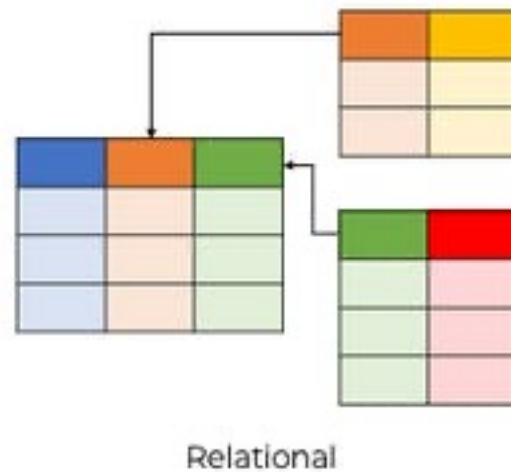
- **NoSQL** steht für "Not Only SQL" und bezeichnet Datenbanken, die nicht auf dem traditionellen relationalen Datenbankmodell basieren.
- Sie sind oft **nicht-relational**, was bedeutet, dass sie keine Tabellen mit festen Beziehungen zwischen den Daten verwenden.
- **Schemalos**: Daten müssen nicht in einem festen Format vorliegen. Sie können flexibel und dynamisch angepasst werden.

Arten von NoSQL-Datenbanken

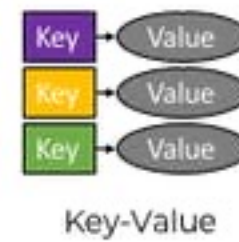
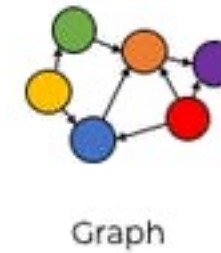
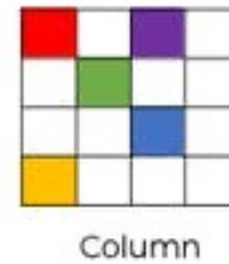
- **Dokumentenorientiert** (z.B. MongoDB): Speichern Daten als Dokumente, typischerweise im JSON-Format.
- **Schlüssel-Wert-Datenbanken** (z. B. Redis): Speichern Daten als Schlüssel-Wert-Paare.
- **Graphdatenbanken** (z. B. Neo4j): Speichern Daten in Form von Knoten und Kanten für komplexe Beziehungen.

Arten von NoSQL-Datenbanken

SQL DATABASES



NoSQL DATABASES

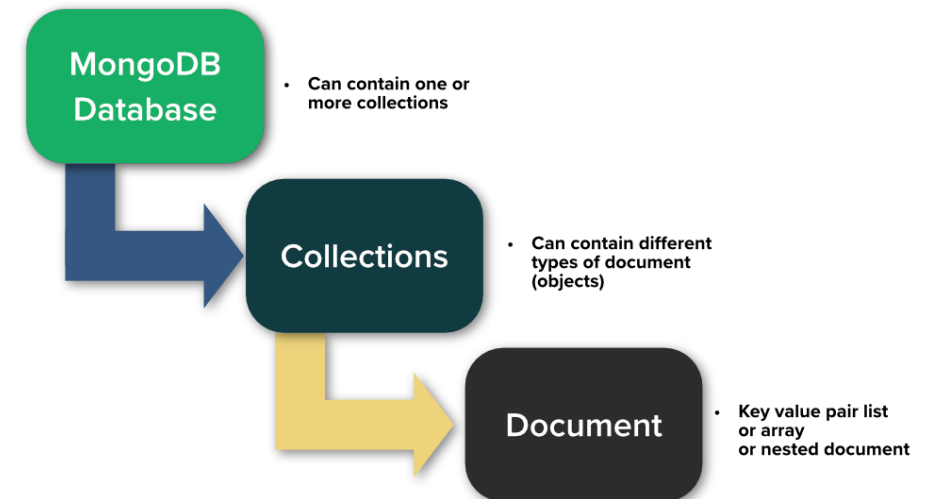


MongoDB - Einleitung

- **Definition:** MongoDB ist eine dokumentenorientierte NoSQL-Datenbank.
- Basiert hauptsächlich auf C++
- Speichert Daten in **Dokumenten** (BSON).
- **BSON** steht für **Binary JSON** (Binäres JSON) und ist ein binäres Format, das von MongoDB verwendet wird, um Daten effizient zu speichern. Es ist eine erweiterte und optimierte Version von JSON (JavaScript Object Notation), das speziell für den Einsatz in einer Datenbankumgebung entwickelt wurde.
- Für uns als Entwickler irrelevant: JSON-Struktur wird in ein kompakteres, schneller verarbeitbares Binärformat umgewandelt.)

MongoDB - Einleitung

- Dokumente werden in **Collections** gespeichert, diese sind vergleichbar mit einer Tabelle in relationalen Datenbanken, aber sie kann Dokumente mit verschiedenen Strukturen enthalten.
- Für die Bedienung und Integration gibt es viele Treiber in verschiedenen Programmiersprachen, z.B. JavaScript (Node.js), Python, Java, C#, Ruby, PHP und viele mehr.



MongoDB – BSON Datentypen

- BSON unterstützt neben den grundlegenden JSON-Datentypen (wie Zahlen, Strings und Arrays) auch zusätzliche Datentypen, die für die Arbeit mit Datenbanken nützlich sind:
 - **Binäre Daten:** Unterstützt die Speicherung von Binärdaten (z.B. Dateien oder Bilder).
 - **Datumsangaben:** BSON bietet einen speziellen Datentyp für Zeitstempel, was für die Speicherung von Datumsangaben und Zeiten von Bedeutung ist.
 - **Objekt-IDs (_id):** MongoDB verwendet BSON, um automatisch generierte Objekt-IDs (_id) zu speichern, die als Primärschlüssel für jedes Dokument dienen.
 - **Reguläre Ausdrücke:** BSON unterstützt auch reguläre Ausdrücke als Datentyp.

MongoDB – BSON Datentypen

MongoDB-Typ	Beschreibung	Beispiel
String (String)	Zeichenkette (UTF-8)	"name": "Lisa"
Integer (Int32, Int64)	Ganzzahl, 32 oder 64 Bit	"alter": 25
Double (Double)	Fließkommazahl	"preis": 19.99
Boolean (Boolean)	Wahr oder falsch	"aktiv": true
Object (Document)	Verschachteltes Objekt	"adresse": { "stadt": "Berlin" }
Array (Array)	Liste von Werten	"hobbys": ["Lesen", "Kochen"]
ObjectId (ObjectId)	Eindeutige ID	"_id": ObjectId("...")
Date (Date)	Datum/Zeit	"geburtstag": ISODate("2000-05-01T00:00:00Z")
Null (Null)	Kein Wert	"telefon": null
Binary Data (BinData)	Binärdaten	z. B. für Bilder
Timestamp (Timestamp)	Interne Zeitmarke für Replikation	selten manuell genutzt
Decimal128 (Decimal128)	Hohe Genauigkeit (z. B. für Geldbeträge)	"preis": NumberDecimal("10.50")
Regular Expression (Regex)	Regulärer Ausdruck	"name": /Lisa/i
MinKey / MaxKey	Kleinster / größter möglicher Wert	Für Vergleiche / Sortierungen

MongoDB – Dokument im BSON Beispiel

```
{
  "_id": ObjectId("60d5ec49f9a1c9d4f8e3b456"),
  "title": "Inception",
  "director": "Christopher Nolan",
  "release_year": 2010,
  "genres": ["Action", "Sci-Fi", "Thriller"],
  "duration_minutes": 148,
  "cast": [
    { "name": "Leonardo DiCaprio", "role": "Dom Cobb" },
    { "name": "Joseph Gordon-Levitt", "role": "Arthur" },
    { "name": "Elliot Page", "role": "Ariadne" }
  ],
  "rating": 8.8,
  "available_on_streaming": true
}
```

In **MongoDB** ist das Feld "_id" ein **automatisch generierter eindeutiger Primärschlüssel**, der **jedem Dokument** zugewiesen wird.

MongoDB – Einige Installationsmöglichkeiten

- **Offizielle MongoDB Pakete**

(1) Pakete online herunterladen von <https://www.mongodb.com/try/download/community>

(2) Package-Manager (Ubuntu/Debian): `sudo apt-get install -y mongodb`

- **MongoDB Atlas (Online-Angebot):** Erstellung eines Accounts auf der offiziellen Internetseite von MongoDB. Dort kann eine Cluster-Instanz eingerichtet werden.
- **Docker mongo:** Bietet Portabilität, schnelle Einrichtung, keine direkte Installation.

MongoDB in Docker

1. MongoDB Image laden:

```
docker pull mongo
```

2. Container ("mongodb-container") auf Basis des Images starten:

```
docker run -d --name mongodb-container -p 27017:27017 mongo
```

MongoDB in Docker – Zugriff auf die MongoDB Shell

- Nun kann man in den laufenden MongoDB-Container eintreten und die MongoDB-Shell

(mongo) ausführen:

```
docker exec -it mongodb-container mongosh
```

- Der Befehl startet die MongoDB-Shell (mongosh) im laufenden Docker-Container mit dem Namen

`mongodb-container`, wobei `-it` für den interaktiven Modus und die Zuweisung eines

Terminals steht.

MongoDB Shell - Commands

```
# Wechsel zur Datenbank (erstellt die Datenbank, wenn sie nicht existiert) -> Wird dennoch erst "richtig"
angelegt, sobald es Daten gibt.
```

```
use myDatabase
```

```
# Zeigt alle Datenbanken an
```

```
show databases
```

```
# Zeigt alle Collections in der aktuellen Datenbank an, die zuvor mit use ausgewählt wurde
```

```
show collections
```

```
# Einfügen eines Dokuments in eine Collection
```

```
db.myCollection.insertOne({ name: "John", age: 30 })
```

```
# Abfragen von Dokumenten aus einer Collection
```

```
db.myCollection.find()
```

MongoDB Shell - Commands

Abfragen mit einer Bedingung

```
db.myCollection.find({ age: { $gt: 25 } })
```

Aktualisieren eines Dokuments

```
db.myCollection.updateOne({ name: "John" }, { $set: { age: 31 } })
```

Löschen eines Dokuments

```
db.myCollection.deleteOne({ name: "John" })
```

Erstellen eines Indexes auf einem Feld

```
db.myCollection.createIndex({ name: 1 })
```

Zeigen der Indizes einer Collection

```
db.myCollection.getIndexes()
```



MongoDB Cheat Sheet
by isaeus via cheatography.com/94031/cs/20684/

Basic Mongo DB	
db	Show name of current database
mongod	Start database
mongo	Connect to database
show dbs	Show databases
use db	Switch to database db
show collections	Display current database collections

Create	
insert (data)	insert document(s) returns write result
insertOne (data, options)	insert one document
insertMany (data, options)	insert many documents
insertMany ([{ }, { }, { }])	needs square brackets

Read	
db.collection.find()	Display documents from collection
find (filter, options)	find all matching documents
findOne (filter, options)	find first matching document

Update	
updateOne (filter, data, options)	Change one document
updateMany (filter, data, options)	Change many documents
replaceOne (filter, data, options)	Replace document entirely

Delete	
deleteOne (filter, options)	Delete one document
deleteMany (filter, options)	Delete many documents

Filters (cont)	
{key: { \$exists: true }}	Matches all documents containing subdocument key
\$eq	Matches values that are equal to a specified value.
\$gt	Matches values that are greater than a specified value.
\$gte	Matches values that are greater than or equal to a specified value.
\$in	Matches any of the values specified in an array
syntax: {key: { \$in: [array of values] }}	
\$lt	Matches values that are less than a specified value.
\$lte	Matches values that are less than or equal to a specified value.
\$ne	Matches all values that are not equal to a specified value.
\$nin	Matches none of the values specified in an array.
\$and	Performs AND operation
syntax: { \$and: [{ }, { }] }	
{key: { \$op: filter }, {filter}}	\$and operator is necessary when the same field or operator has to be specified in multiple expressions
find({ - doc.subdoc - c:v alue })	Filter sub documents

Functions	
.count()	Counts how many results
.sort(filter)	Sort ascend: 1 descend: -1



By isaeus
cheatography.com/isaetus/

Published 3rd October, 2019.
Last updated 3rd October, 2019.
Page 1 of 1.

Sponsored by [Readable.com](https://readable.com)
Measure your website readability!
<https://readable.com>

MongoDB Shell - Commands

Löschen eines Indexes

```
db.myCollection.dropIndex("index_name")
```

Benutzer erstellen

```
db.createUser({ user: "myUser", pwd: "password123", roles: [{ role: "readWrite", db: "myDatabase" }] })
```

Serverstatus anzeigen

```
db.serverStatus()
```


Aufgabe: MongoDB Docker Container starten und Collection anlegen (10 Minuten)

1. Starte einen **MongoDB Docker Container** lokal auf deinem Rechner.
2. Öffne die MongoDB Shell (`mongosh`) im Container.
3. Lege in der Datenbank `test` eine neue **Collection** namens `todos` an.
4. Füge zwei Dokumente (Todos) in die Collection `todos` ein, z. B.:

```
{ task: "Einkaufen", done: false }  und  
{ task: "Hausaufgaben machen", done: true }
```

Mongoose – Verbindung zur MongoDB in Node.js

- Mongoose ist eine **ODM (Object Document Mapper)** - Bibliothek für MongoDB und Node.js.
- Vereinfacht die Interaktion mit MongoDB, indem sie die Arbeit mit MongoDB-Dokumenten und Sammlungen in ein objektorientiertes Modell überführt.
- Statt direkt mit der MongoDB-Abfragesprache zu arbeiten, kannst du mit Mongoose auf eine Art und Weise arbeiten, die dir vertrauter ist, wenn du bereits mit objektorientierter Programmierung und JavaScript arbeitest.

Mongoose – Verbindung zur MongoDB in Node.js

- Installation: `npm i mongoose`
- Nach der Installation kann Mongoose im Node.js-Projekt verwendet werden, um eine Verbindung zur MongoDB-Datenbank herzustellen

```
import mongoose = from 'mongoose'; // Importiere die Mongoose-Bibliothek

// Verbinde dich mit der MongoDB-Datenbank
mongoose.connect('mongodb://localhost/myDatabase', { // URL der MongoDB
  useNewUrlParser: true, // Setzt den neuen Parser für die Verbindung (empfohlen für neuere MongoDB-Versionen)
  useUnifiedTopology: true // Aktiviert die neue Topologie-Bibliothek für eine verbesserte Verbindung
}).then(() => console.log('Datenbank verbunden')) // Wenn die Verbindung erfolgreich ist, gebe 'Datenbank verbunden' aus
.catch(err => console.error('Verbindungsfehler:', err)); //Fehler abfangen
```

Mongoose – Konzepte

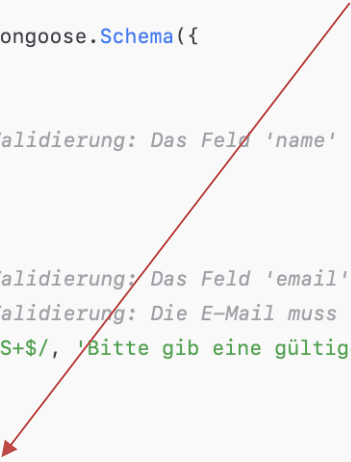
- **Schema:** Ein Schema beschreibt die Struktur von Dokumenten in einer MongoDB-Collection. Es definiert, welche Felder die Dokumente enthalten sollen, welche Datentypen sie haben, und welche Validierungsregeln gelten.
- **Modell:** Ein Modell ist eine Konstruktion, die mit einem **Schema verbunden** ist und als Interface zur **Interaktion mit den Dokumenten der MongoDB-Collection** dient. Modelle ermöglichen es, CRUD-Operationen (Create, Read, Update, Delete) auf den Daten durchzuführen.

Mongoose – Erstellen eines Schemas

- **Feldnamen:** Die Namen der Felder, die in den Dokumenten gespeichert werden.
- **Daten-Typen:** Die Typen der Daten, die jedes Feld speichern kann (z. B. String, Number, Date).
- **Optionen:** Zusätzliche Optionen wie required (ob das Feld erforderlich ist) oder unique (ob der Wert einzigartig sein muss).
- **Validierungen:** Regeln zur Sicherstellung, dass die Daten bestimmte Bedingungen erfüllen.

javascript

```
const mongoose = require('mongoose');  
  
const userSchema = new mongoose.Schema({  
  name: {  
    type: String,  
    required: true, // Validierung: Das Feld 'name' ist erforderlich  
  },  
  email: {  
    type: String,  
    required: true, // Validierung: Das Feld 'email' ist erforderlich  
    unique: true, // Validierung: Die E-Mail muss einzigartig sein  
    match: [/^\S+@\S+\.\S+$/, 'Bitte gib eine gültige E-Mail-Adresse ein'] // Va  
  },  
  age: {  
    type: Number,  
    min: [18, 'Das Alter muss mindestens 18 Jahre betragen'], // Validierung: Da  
    max: [120, 'Das Alter muss unter 120 Jahren liegen'] // Validierung: Das Alt  
  }  
});  
  
const User = mongoose.model('User', userSchema);  
  
// Dokument mit gültigen Daten  
const newUser = new User({  
  name: 'Alice',  
  email: 'alice@example.com',  
  age: 25  
});  
  
newUser.save()  
  .then(() => console.log('Benutzer gespeichert'))  
  .catch(err => console.error('Fehler beim Speichern:', err));
```



Mongoose – Models

- Ein Model ist eine Klasse, die basierend auf einem Schema Dokumente erstellen kann.
- Wenn du `newUser.save()` aufrufst, wird das Dokument in der Datenbank gespeichert.
- Model bietet Methoden für **C**reate**R**ead**U**pdate**D**elese-Operationen
- Beispiele: `save()`, `find()`, `update()`, `delete()`.

```
const User = mongoose.model('User', userSchema);

// Dokument mit gültigen Daten
const newUser = new User({
  name: 'Alice',
  email: 'alice@example.com',
  age: 25
});

newUser.save()
  .then(() => console.log('Benutzer gespeichert'))
  .catch(err => console.error('Fehler beim Speichern:', err));
```

Mongoose – DELETE

javascript

```
// Lösche das erste Dokument, das der Bedingung entspricht
User.deleteOne({ email: 'bob@example.com' })
  .then(result => {
    console.log('Dokument gelöscht:', result);
  })
  .catch(err => {
    console.error('Fehler beim Löschen:', err);
  });
```

Mongoose – UPDATE

javascript

```
// Aktualisiere das Alter eines Benutzers basierend auf der E-Mail-Adresse
User.findOneAndUpdate(
  { email: 'bob@example.com' }, // Bedingung
  { age: 30 },                  // Neue Werte
  { new: true }                 // Gibt das aktualisierte Dokument zurück
)
.then(updatedUser => {
  console.log('Benutzer aktualisiert:', updatedUser);
})
.catch(err => {
  console.error('Fehler beim Aktualisieren:', err);
});
```


Mongoose – GET

javascript

```
// Suche einen Benutzer nach der E-Mail-Adresse  
User.findOne({ email: 'bob@example.com' })  
  .then(user => {  
    console.log('Benutzer gefunden:', user);  
  })  
  .catch(err => {  
    console.error('Fehler beim Suchen:', err);  
  });
```

Mongoose – GET (all)

javascript

```
// Alle Benutzer, die älter als 18 Jahre sind  
User.find({ age: { $gt: 18 } })  
  .then(users => {  
    console.log('Benutzer gefunden:', users);  
  })  
  .catch(err => {  
    console.error('Fehler beim Suchen:', err);  
  });
```

Mongoose – GET (all, Paginierung)

javascript

```
// Begrenze die Anzahl der Ergebnisse auf 5
User.find()
  .limit(5)
  .then(users => {
    console.log('Benutzer gefunden:', users);
  })
  .catch(err => {
    console.error('Fehler beim Suchen:', err);
  });

// Paginierung – überspringe die ersten 10 Benutzer
User.find()
  .skip(10)
  .limit(5)
  .then(users => {
    console.log('Benutzer gefunden:', users);
  })
  .catch(err => {
    console.error('Fehler beim Suchen:', err);
  });
```



User Management mit MongoDB/Mongoose und JWT

Was ist User Management?

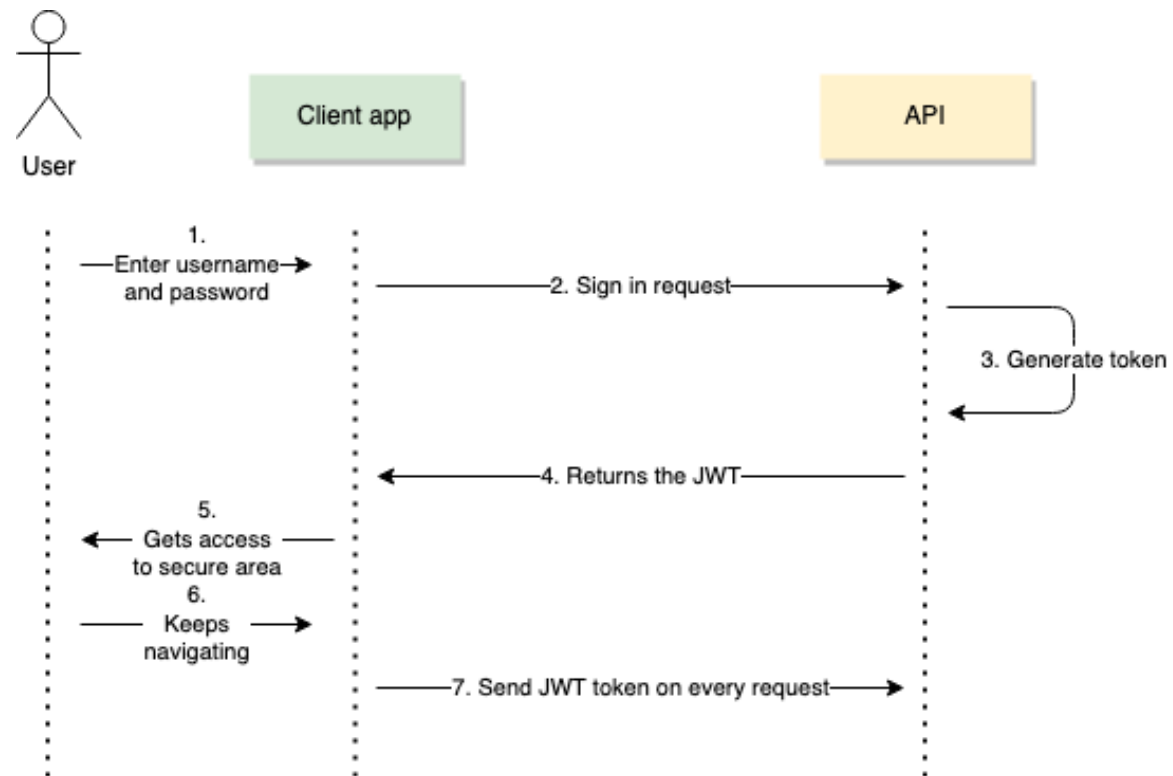
- Verwaltung von Benutzerkonten in einer Webanwendung.
- Funktionen wie Registrierung, Login, Passwortverwaltung und Authentifizierung.
- Ziel: Nur autorisierte Nutzer Zugriff auf bestimmte Ressourcen haben.

User Management mit MongoDB/Mongoose und JWT

- Ein **JWT (JSON Web Token)**: Benutzeridentität zwischen Client und Server kommunizieren.
- Ein **JWT (JSON Web Tokens)**: Benutzeridentität wird in einem **digital signierten Token** gespeichert und zwischen Client und Server übertragen werden kann.
- Beispiel:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikp1eGEgTXVzdGVybWFubGlzInJvbGUiOiJhZG1pbiIsImVudCI6IjE5MjM5NTUyMn0.  
SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

User Management mit MongoDB/Mongoose und JWT



User Management mit MongoDB/Mongoose und JWT

Header: Enthält Metadaten wie Algorithmus (HS256) mit dem der Token signiert wird und Typ (JWT).

Payload: Beinhaltet Benutzerdaten (z. B. Benutzer-ID, Rollen).

Signatur: Verifiziert die Integrität der Daten mit einem SECRET_KEY.



Header und Payload sind codiert (Base64), können einfach z.B. mit Online-Tool ausgelesen werden.

User Management mit MongoDB/Mongoose und JWT

- Der Server nimmt:
 1. den **Header** (Teil 1 des JWT-Tokens)
 2. den **Payload** (Teil 2 des JWT-Tokens)
 3. und einen **geheimen Schlüssel** (String, möglichst anspruchsvoll), den nur der Server kennt.
- Daraus wird eine **Signatur** (Teil 3 des JWT Tokens) mit einer Rechenregel wie „HS256“ berechnet.
- Zusammengefasst: Die **Signatur** ist ein digitales Siegel, das der Server ursprünglich aus Header, Payload und dem geheimen Schlüssel berechnet hat.
- Angenommen, man ändert den Payload, dann würde eine neue, nicht gültige Signatur erstellt werden

HS256 ist ein kryptografischer Algorithmus, der bei JWTs verwendet wird, um eine **digitale Signatur** zu erstellen.

„HS“: **HMAC mit SHA-256**:

HMAC ist eine Methode, die eine Nachricht mit einem geheimen Schlüssel kombiniert, um eine sichere Prüfsumme (Signatur) zu erzeugen.

SHA-256 ist eine Hash-Funktion, die aus beliebigen Daten einen eindeutigen, festen 256-*Bit-Wert* berechnet.

User Management mit MongoDB/Mongoose und JWT: Secret Key

Wie bereits in der vorherigen Folie erwähnt, ist der `SECRET_KEY` ein **geheimer Schlüssel**, der verwendet wird, um JWTs (JSON Web Tokens) zu signieren und zu validieren.

Signierung:

javascript

```
jwt.sign({ userId: 123 }, SECRET_KEY);
```

Verifizierung:

javascript

```
jwt.verify(token, SECRET_KEY);
```

Header Default (3. Argument von `.sign()`): { "alg": "HS256", "typ": "JWT" }

Wo speichert man einen Secret Key? Environment Variables!

- **Environment Variables (Umgebungsvariablen):** Sensible oder konfigurierbare Daten, die außerhalb des Programmcodes in der Laufzeitumgebung definiert sind.
- Diese Variablen werden oftmals in einer Datei `.env` gespeichert.
- **Wichtig:** Veröffentlicht man sein Repository, sollte die `.env` Datei nicht in dem Repository enthalten sein.

Beispiel:

plaintext

```
SECRET_KEY=MeinGeheimerSchlüssel  
DATABASE_URL=mongodb://localhost/myDatabase
```

User Management mit MongoDB/Mongoose und JWT: Wo speichert man einen Secret Key? Environment Variables!

- Variablen aus der .env können mithilfe von dotenv in Node.js geladen werden:

```
bash

npm install dotenv
```

- Integration im Code:

```
javascript

require('dotenv').config();
const secretKey = process.env.SECRET_KEY;
console.log(secretKey);
```

Die Methode `.config()` liest die **.env-Datei** und lädt ihre Inhalte als **Environment-Variablen** in die Laufzeitumgebung der Anwendung.

`process` ist ein globales **Node.js-Objekt**, das Informationen und Funktionen zur Steuerung des aktuellen Node.js-Prozesses bereitstellt.

Demo!

Weitere Ressourcen (Leseempfehlung!)

- Docker compose: <https://docs.docker.com/compose/gettingstarted/> /
<https://www.freecodecamp.org/news/what-is-docker-compose-how-to-use-it/>