Web-Development Backend

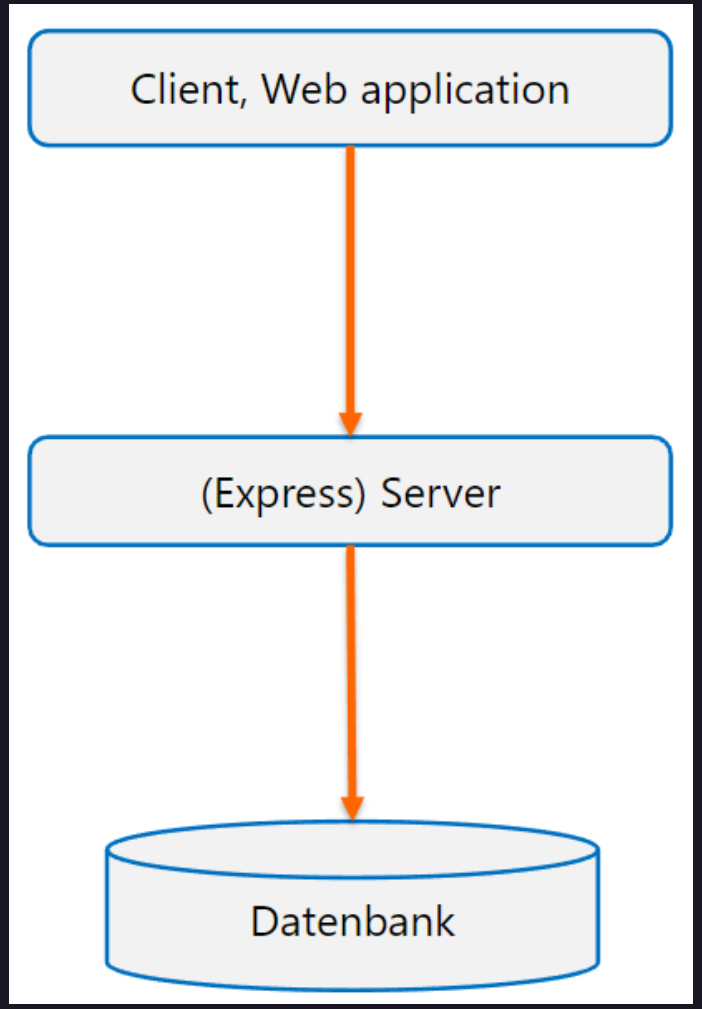
JOI und JWT - Server vor invalidem Input schützen
oder:
Wie mache ich den Server kaputt?

webdev.andreasnicklaus.de - 📁 PDF herunterladen

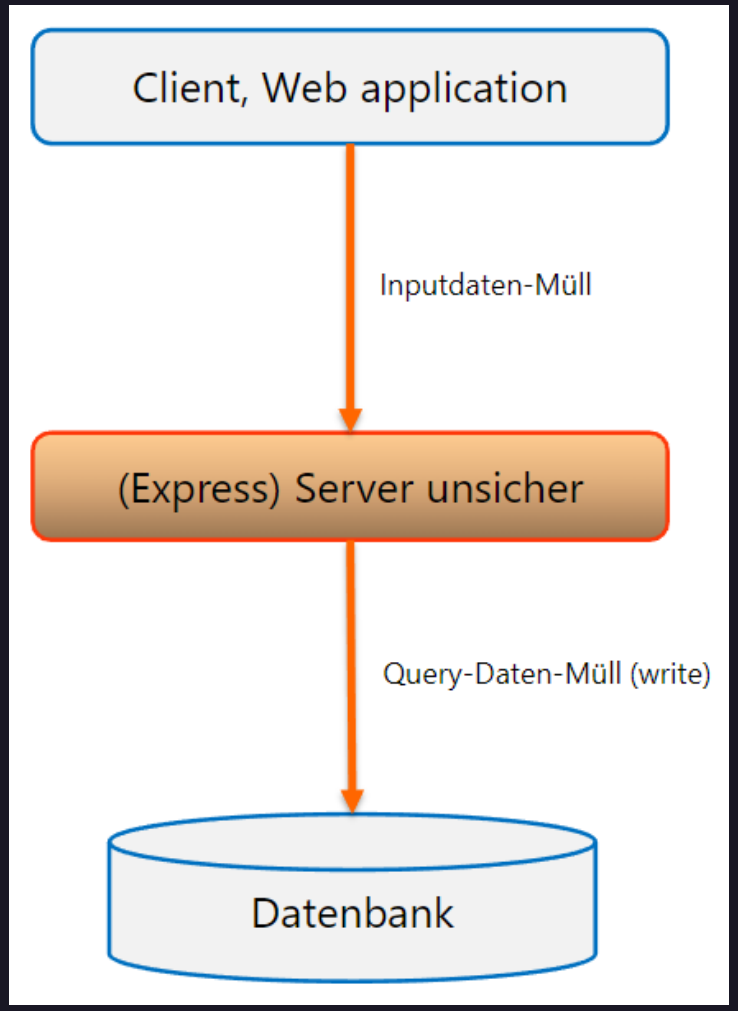
Was haben wir bisher gemacht?

- ✅ HTTP-Server mit Express
- ✅ HTML-Rendering mit EJS
- ✅ Datenbankanbindung mit MongoDB
- ✅ Datenbankverwaltung mit Mongoose
- ✅ 2-Way-Kommunikation mit Websockets
- ⬛ ???
- ⬛ ???

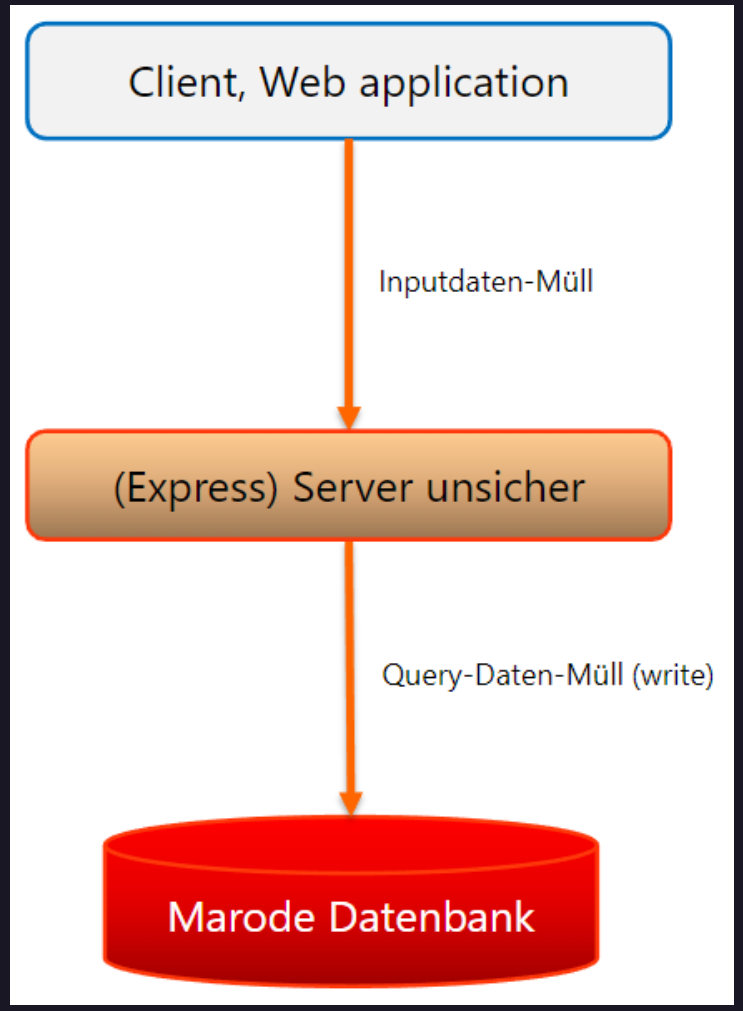
Ungeschützte Architektur



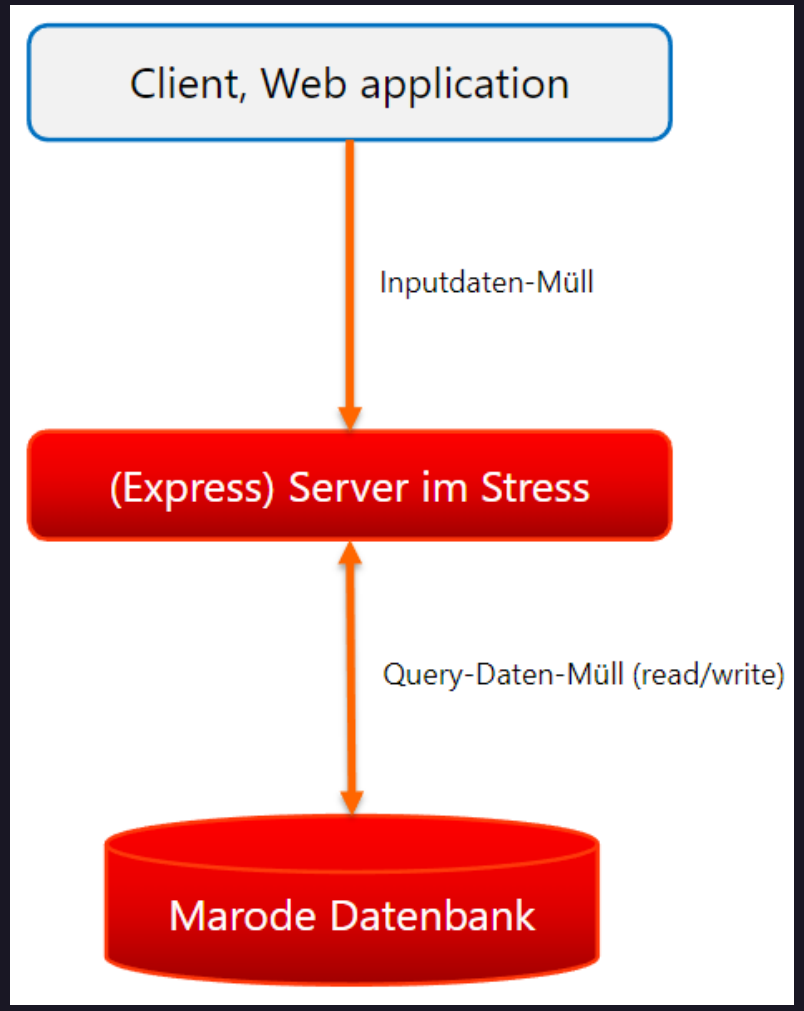
Ungeschützte Architektur



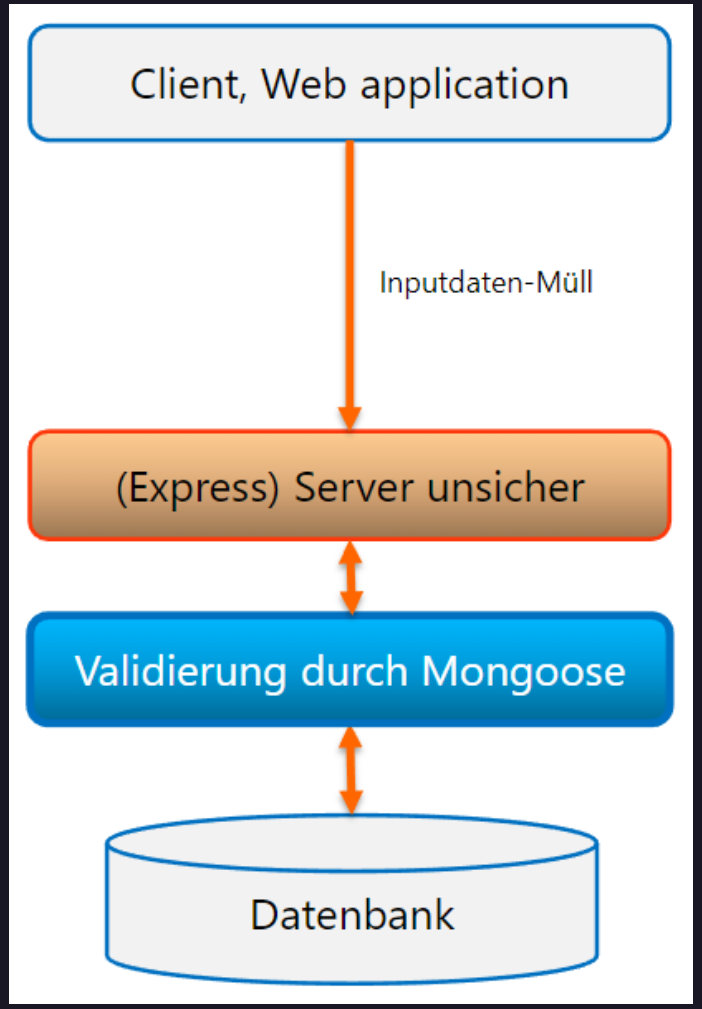
Ungeschützte Architektur



Ungeschützte Architektur



Geschützte Datenbankstruktur



Beispiel Twitter

Welche Arten von schlechten Inputs können wir erfahren?

```
{
  "authorId" : "ElonMusksUserId",
  "content" : "Mark Zuckerberg is a great business man.",
  "creationTime": 1705326917363
}
```

Beispiel Twitter

Schlechte Inputs

```
{
  "authorId" : "iReallyAmElonMusk",
  "content" : "Mark Zuckerberg $§%&/&",
  "creationTime": 000000001
}
```

```
{
  "content" : 123456,
  "creationTime": "1603-01-01T00:00:00.000Z",
}
```

SQL Injection

```
{
  "authorId" : "ElonMusksUserId'; DELETE * from Users; COMMIT;",
  "content" : "123456",
  "creationTime": 1705326917363
}
```

```
SELECT * from Users where id='ElonMusksUserId'; DELETE * from Users; COMMIT;'
```

Schlechte Inputs

- Felder nicht gefüllt
- Felder nicht vorhanden
- Felder haben nicht den richtigen Typ
- Inhalt ist logisch nicht richtig
- Inhalt ist nicht erlaubt

Inputvalidierung mit JOI

```
const Joi = require('joi')

const myTwitterPostSchema = Joi.object({
  authorId: Joi.string().alphanum().required(),
  content: Joi.string().min(1).required(),
  creationTime: Joi.number().integer()
    .min(new Date().valueOf()-60000)
    .max(new Date().valueOf()).required()
})

const data = {
  authorId: "097151d159a0467ea3b45ec37abf771c",
  content: "Twitter was lame. I love X! <3"
}

const result = myTwitterPostSchema.validate(data)
if (result.error) console.error(result.error.message)
```

Weitere Validierungsmöglichkeiten

```
// Joi.object() beschreibt ein JS-Object
const schema = Joi.object({
  // Joi.string() beschreibt ein JS-String
    username: Joi.string().alphanum().min(3).max(30).required(),

    // .pattern() erlaubt eine Regular Expression
    password: Joi.string().pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')),

    // .ref() verweist auf ein anderes Schema
    repeat_password: Joi.ref('password'),

    // Array erlaubt mehrere optionale Typen
    access_token: [Joi.string(), Joi.number()],

    // string.email() defniert den String als E-Mail-Adresse
    email: Joi.string().email({ minDomainSegments: 2, tlds: { allow: ['com', 'net'] } })
})
```

https://github.com/hapijs/joi/blob/v14.3.1/API.md

Beispiel für Einbindung an Express-Server

```
app.post('/post', (req, res, next) => {
  const myTwitterPostSchema = Joi.object({
  authorId: Joi.string().alphanum().required(),
  content: Joi.string().min(1).required(),
  creationTime: Joi.number().integer()
    .min(new Date().valueOf()-60000).max(new Date().valueOf()).required()
})
  const result = myTwitterPostSchema.validate(req.body)

  if (!result.error) {
    res.status(422).json({
      message: 'Invalid request, error: ' + error.message,
      data: req.body
    })
  } else {
    createPost(data).then((createdPost) => {
      res.json({ message: 'Post created', data: createdPost })
    })
  }
})
```

Beispiel für Einbindung als Express-Middleware

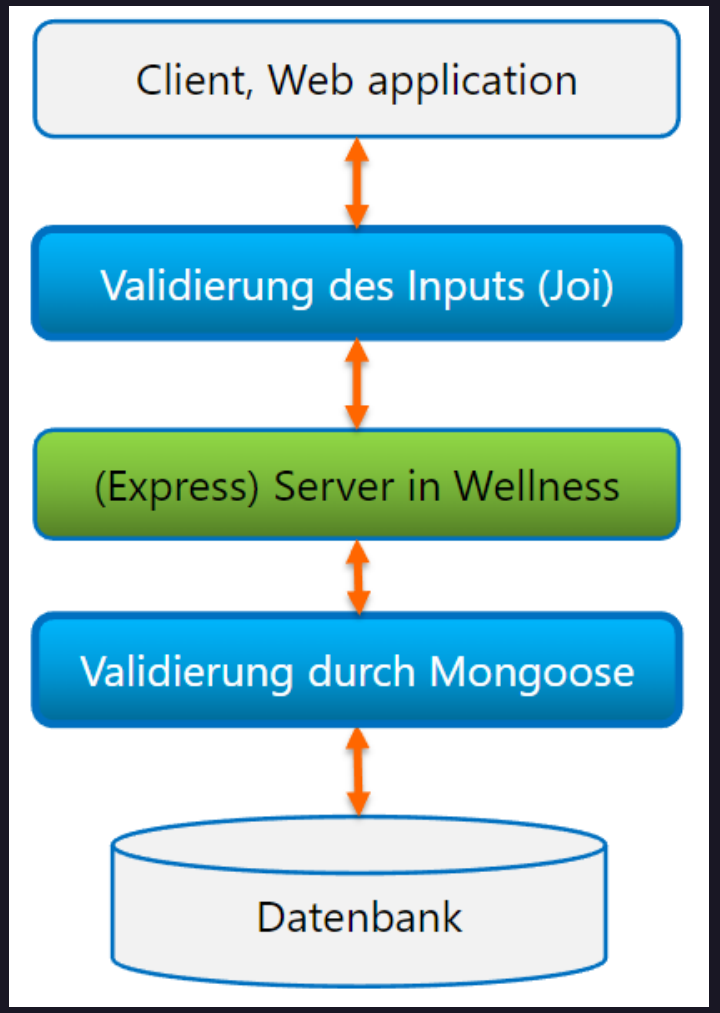
```
validateSchema = function (schema = null, property = null) {
  function middleware(req, res, next) {
    if (!JOI_ENABLED) return next()
    else {
      const { value, error } = schema.validate(req[property])

      if (error) {
        next(error);
        return;
      }
      else next()
    }
  }

  return middleware
}

app.post('/post', validateSchema(myTwitterPostSchema, "body"), => {
  ...
})
```

Geschützter Backend-Server



Was denkt ihr?

1. Ist das einfach?
2. Ist das effektiv?
3. Welche Problemen ergeben sich?

Authentisierung falsch gemacht

```
{
  "myId": "097151d159a0467ea3b45ec37abf771c",
  "post": {
    "content": "This is a funny tweet about spaghetti.",
    "creationTime": 1705326917363
  }
}
```

Ist das sicher?

Authentisierung richtig gemacht

> The HTTP Authorization request header can be used to provide credentials that authenticate a user agent with a server, allowing access to a protected resource.

```
Authorization: <auth-scheme> <authorization-parameters>
```

Hier werden 2 Authentisierungsschemas vorgestellt, es gibt aber noch mehr:

1. Basic
2. Bearer

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization

Basic Authentisierung

`Authorization: Basic <base64('<username>:<password>')>`

- Authentisierungsschema: `Basic`
- Authentisierungsparameter besteht aus dem Base64-enkodiertem String `<username>:<password>`

 Warnung: Base64-encoding kann einfach zum ursprünglichen Namen und Passwort dekodiert werden. Basic Authentisierung ist deshalb vollständig unsicher. HTTPS is immer empfohlen, wenn Authentisierung benutzt wird, aber besonders bei `Basic` Authentisierung.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization#basic

Bearer Authentisierung

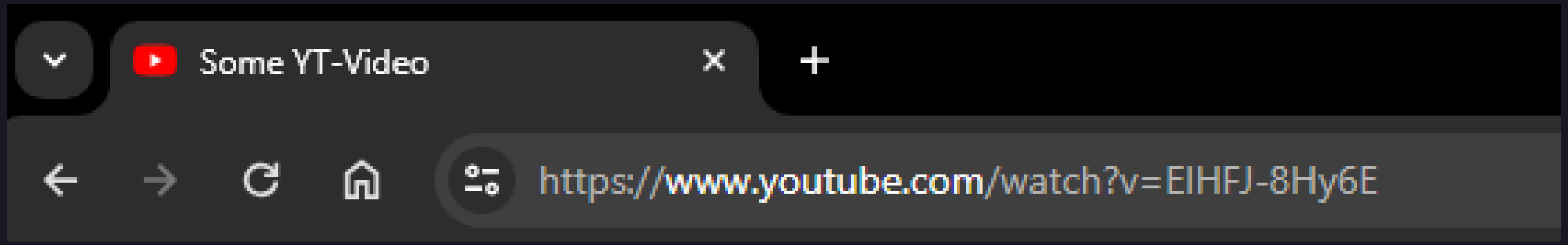
`Authorization: Bearer <Token>`

- Authentisierungsschema: `Bearer`
- Authentisierungsparameter besteht aus einem Token, das der Client nie anfassen will und soll

Bearer Authentisierung erfordert, dass der Token vom Client nicht verändert werden kann

➡️ **JSON Web Tokens**

Intermezzo: IDs



ID-Formate:

- Youtube: `xxxxx-xxxxx` alphanumerisch
- IG Reels: `xxxxxxxxxxx` alphanumerisch
- Twitter/X: `00000000000000000000` numerisch
- IG Stories: `00000000000000000000` numerisch

Intermezzo: IDs

Beispiel Youtube: `https://www.youtube.com/watch?v=ElHFJ-8Hy6E`

Ein paar Fragen zur Auswahl des Formats:

- Welche ID wird als nächste vergeben? Sollte ich dieses Video aufrufen können?
- Wie viele Varianten gibt das Format her?
- Wie groß ist die Wahrscheinlichkeit, eine ID zu erraten?

JSON Web Token (JWT)

- Verschlüsselte Tokens zur Authentifizierung von JSON-Daten

1. Datenintegrität: Signierte Tokens
2. Datenverschlüsselung: Daten werden geheim gehalten

Online Token-Generator: https://jwt.io/

Aufbau eines JWT

`hhhhh.pppppppppppppppppppppp.sssssssssss`

1. Header
2. Payload
3. Signature

JWT Teil 1/3: Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

wird Base64Url kodiert.

JWT Teil 2/3: Payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516171819
}
```

wird Base64Url kodiert.

- Properties des Payloads werden **Claim** genannt
- Registrierte Claims sind `sub` (Subject), `iss` (Issuer), `exp` (Expiration Time), `aud` (Audience, Array of Strings), `iat` (Issued at), `jti` (JWT Id), `nbf` (Not before)
- Neben Public Claims der JSON Web Token Claims Registry sind auch **Private Claims** erlaubt **(Vorsicht vor Kollisionen)**

JWT Teil 3/3: Signature

```
HMAC(Base64Url(header).Base64Url(payload), secret)
```

- Im einfachen Fall werden Header und Payload mit HMAC-Verschlüsselung (hash-base message auth code) symmetrisch Verschlüsselt.
- Payload und Header werden dennoch unverschlüsselt verwendet.

Warum sind JWTs sicher?

- Header und Payload sind Base64-enkodiert ➡️ **Lesbar und veränderbar**
- Signatur enthält Secret und Payload ➡️ **Änderungen sind nachweisbar**

- Änderungen sind nicht reversibel, das Original bleibt unbekannt.
- Keine sensiblen Daten sollten in JWTs verpackt werden.

JWT mit RSA-Verschlüsselung

```
const message =
  RSA_with_publicKey_of_receiver(Base64(Header)) + "." +
  RSA_with_publicKey_of_receiver(Base64(Payload))

const signature = RSA_with_privateKey_of_sender(message)
```

Verwendung von JWTs im Client

```
// Token wird vom letzten Request gespeichert, modifiziert und/oder generiert
const token = getOrGenerateToken()

fetch("http://example.com/path/",
  {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: 'Bearer ' + token
    },
    body: JSON.stringify({...data})
  }
)
```

Generierung von JWTs im Server

```
const jwt = require('jsonwebtoken')

const SECRET = require('crypto').randomBytes(64).toString('hex')
// '09f26e402586e2faa8da4c98a35f1b20d6b033c6097befa8be3486a829587fe2f90a832bd
//  3ff9d42710a4da095a2ce285b009f0c3730cd9b8e1af3eb84df6611'

function generateAccessToken(username) {
  return jwt.sign(username, SECRET, { expiresIn: '1800s' })
}

app.post('/path', (req, res) => {
  const token = generateAccessToken({ username: req.body.username })
  res.json(token);
})
```

JWT-Dekodierung

```
app.post('/path', authenticateToken, (req, res) => {
  // handle request
})

//middleware for authentication
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization']
  const token = authHeader.split(' ')[1]
  if (token == null) return res.sendStatus(401)
  jwt.verify(token, SECRET, (err, user) => {
    if (err)
      return res.sendStatus(403)
    req.user = user
    next()
  })
}
```

Nutzung von JWTs

➡️ Codebeispiel

Was haben wir erreicht?

- ✅ HTTP-Server mit Express
- ✅ HTML-Rendering mit EJS
- ✅ Datenbankanbindung mit MongoDB
- ✅ Datenbankverwaltung mit Mongoose
- ✅ 2-Way-Kommunikation mit Websockets
- ✅ Inputvalidierung mit JOI
- ✅ Authentisierung mit JSON Web Tokens