

衛星データ解析技術研究会 技術セミナー（応用編）

Webアプリケーションの開発技術の習得

第四回 2025/07/18

担当講師：田中聡至

前回のあらすじ

最終実装例の確認

<https://github.com/alt9800/2025-RemoteSensingSeminar/tree/main/2025-07-11/handson/app-complete>

擬似データ永続化の例いろいろ

<https://alt9800.github.io/2025-RemoteSensingSeminar/handson/2025-07-11/pseudo-DB/>

MapLibreのいろんな実装

<https://maplibre.org/maplibre-gl-js/docs/examples/>

ホストのポートフォワード

今後の全体像

バックエンド開発①

~~バックエンドとは / サーバ環境の構築~~
サーバ環境の構築などは第2回で済ませた
フロントエンドのみでデータを永続化させる

バックエンド開発② 🖱️今回

~~動的なWebアプリとは~~
認証認可について

バックエンド開発③

フロントエンドとの接続
(この内容も一部今回触れます)

バックエンド開発④

デプロイ

テーマ

認証機能について

念の為事前に確認してもらいます...

Node.js (v14以上) がインストールされていること
npm または yarn が使用できること

一気貫通で実装する機能

ユーザー登録（サインアップ）

ログイン

認証が必要なAPIについて(保護されたルートの作成)

ユーザー情報の取得

(注意)

スライド中のコマンドはLinuxやMacのShellScriptを想定しています。
Powershellやコマンドプロンプトを利用する方は、適宜LLMなどにて 同様のコマンドを
{ご自身のCLI環境}に合わせて変換してください という感じで変換をかけてください。

次回までに完成するディレクトリ構造(npmベアメタル版)

```
project/
├── package.json
├── .env                                # 環境変数（.env.exampleからコピー）
├── .env.example
├── scripts/
│   └── init-db.js                    # データベース初期化スクリプト
├── src/
│   ├── server.js                    # メインサーバーファイル
│   ├── config/
│   │   └── db.js                    # SQLite接続設定
│   ├── middleware/
│   │   └── auth.js                  # JWT認証ミドルウェア
│   └── routes/
│       ├── auth.js                  # 認証関連API
│       ├── posts.js                 # 投稿関連API
│       └── users.js                 # ユーザー関連API
├── public/                          # 静的ファイル（フロントエンド）
├── uploads/                         # アップロードされた画像
└── db/
    └── database.db                  # SQLiteデータベースファイル
```


今回つくる実装

```
express-auth-tutorial/  
├── package.json  
├── .env  
├── src/  
│   ├── server.js           # メインサーバーファイル  
│   ├── config/  
│   │   └── db.js           # データベース設定  
│   ├── middleware/  
│   │   └── auth.js         # 認証ミドルウェア  
│   └── routes/  
│       └── auth.js         # 認証関連のルート  
├── scripts/  
│   └── init-db.js          # DB初期化スクリプト  
├── db/                     # SQLiteデータベース格納  
└── public/                 # 静的ファイル
```

環境構築編

```
npm init -y
```

```
# 基本的なパッケージ
```

```
npm install express dotenv cors helmet
```

```
# 認証関連のパッケージ
```

```
npm install jsonwebtoken bcrypt express-validator
```

```
# データベース関連（SQLite使用）
```

```
npm install sqlite3
```

```
# 開発用パッケージ
```

```
npm install --save-dev nodemon
```

```
mkdir -p src/{config,middleware,routes}  
mkdir -p db scripts public
```

.envを設定

```
# データベース設定
DB_PATH=./db/database.db

# JWT設定
JWT_SECRET=your-jwt-secret-key-change-in-production

# アプリケーション設定
NODE_ENV=development
PORT=3000
```

データベース編

scripts/init-db.js

```
const sqlite3 = require('sqlite3').verbose();
const path = require('path');
const fs = require('fs');
const dotenv = require('dotenv');

// 環境変数の読み込み
dotenv.config();

// データベースディレクトリの作成
const dbDir = path.join(__dirname, '../db');
if (!fs.existsSync(dbDir)) {
  fs.mkdirSync(dbDir, { recursive: true });
}

// データベースパス
const dbPath = process.env.DB_PATH || path.join(dbDir, 'database.db');

// データベースの初期化
const db = new sqlite3.Database(dbPath);

console.log('Initializing database...');

db.serialize(() => {
  // ユーザーテーブル
  db.run(`
    CREATE TABLE IF NOT EXISTS users (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      user_id VARCHAR(50) UNIQUE NOT NULL,
      email VARCHAR(255) UNIQUE NOT NULL,
      password_hash VARCHAR(255) NOT NULL,
      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
  `);

  console.log('Database initialization completed!');
});

db.close((err) => {
  if (err) {
    console.error('Error closing database:', err);
  } else {
    console.log('Database connection closed.');
  }
});
```

src/config/db.js

```
const sqlite3 = require('sqlite3').verbose();
const path = require('path');
const fs = require('fs');

// データベースディレクトリの作成
const dbDir = path.join(__dirname, '../db');
if (!fs.existsSync(dbDir)) {
  fs.mkdirSync(dbDir, { recursive: true });
}

// データベースパス
const dbPath = process.env.DB_PATH || path.join(dbDir, 'database.db');

// データベースインスタンスの作成
const db = new sqlite3.Database(dbPath, (err) => {
  if (err) {
    console.error('Error opening database:', err);
  } else {
    console.log('Connected to SQLite database');
  }
});

// Promise ベースのクエリ関数
const query = (sql, params = []) => {
  return new Promise((resolve, reject) => {
    if (sql.trim().toUpperCase().startsWith('SELECT')) {
      db.all(sql, params, (err, rows) => {
        if (err) reject(err);
        else resolve({ rows });
      });
    } else {
      db.run(sql, params, function(err) {
        if (err) reject(err);
        else resolve({
          rows: [{
            id: this.lastID,
            changes: this.changes
          }]
        });
      });
    }
  });
};

module.exports = {
  query,
  db
};
```


DBの初期化

```
node scripts/init-db.js
```

```
$ node scripts/init-db.js  
[dotenv@17.2.0] injecting env (4) from .env (tip: ⚙ load multiple .env files with { path: ['.env.local', '.env'] })  
Initializing database...  
Database initialization completed!  
Database connection closed.
```

認証ミドルウェアの作成

ミドルウェアとは？

リクエスト → [ミドルウェア1] → [ミドルウェア2] → [ルートハンドラ] → レスポンス

の様な処理がある際に、リクエストとレスポンスの間の処理を行うものをWebにおけるミドルウェアとして扱います。

(ミドルウェアという語がいろいろなところで使われてややこしいかもしれません。)

src/middleware/auth.js : JWT認証

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

const JWT_SECRET = process.env.JWT_SECRET || 'your-jwt-secret-key-change-in-production';

// JWTトークンの生成
const generateToken = (user) => {
  return jwt.sign(
    {
      id: user.id,
      user_id: user.user_id,
      email: user.email
    },
    JWT_SECRET,
    { expiresIn: '7d' }
  );
};

// 認証ミドルウェア
const authenticateToken = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1]; // Bearer TOKEN

  if (!token) {
    return res.status(401).json({ message: '認証トークンが必要です' });
  }

  jwt.verify(token, JWT_SECRET, (err, user) => {
    if (err) {
      return res.status(403).json({ message: 'トークンが無効です' });
    }
    req.user = user;
    next();
  });
};

module.exports = {
  generateToken,
  authenticateToken
};
```

認証ルートの実装

src/routes/auth.js：認証ルートの作成 非常に長いのでコピペしてみていきましょう

```
const express = require('express');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const { body, validationResult } = require('express-validator');
const db = require('../config/db');
const { generateToken } = require('../middleware/auth');

const router = express.Router();

// サインアップ
router.post('/signup', [
  body('email').isEmail().normalizeEmail(),
  body('password').isLength({ min: 8 }), withMessage('パスワードは8文字以上必要です'),
  body('user_id').isLength({ min: 3 }).withMessage('idは3~20の1桁'),
  body('password').isLength({ min: 8 }).withMessage('パスワードは8文字以上の英数字、ハイフン、アンダースコアのみ使用可能です'),
], async (req, res) => {
  try {
    //バリデーションチェック
    const errors = validationResult(req);
    if (errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    const { email, password, user_id } = req.body;

    // ユーザーの重複チェック
    const existingUser = await db.query(
      'SELECT id FROM users WHERE email = ? OR user_id = ?',
      [email, user_id]
    );

    if (existingUser.rows.length > 0) {
      return res.status(400).json({
        message: 'このメールアドレスまたはユーザーIDは既に使用されています'
      });
    }

    //パスワードのハッシュ化
    const hashedPassword = await bcrypt.hash(password, 10);

    // ユーザーの作成
    const result = await db.query(
      'INSERT INTO users (email, password_hash, user_id) VALUES (?, ?, ?)',
      [email, hashedPassword, user_id]
    );

    const userId = result.rows[0].id;

    // DBにユーザー情報を追加
    const newUser = await db.query(
      'SELECT id, email, user_id FROM users WHERE id = ?',
      [userId]
    );

    const user = newUser.rows[0];
    const token = generateToken(user);

    res.status(201).json({
      message: 'ユーザー登録が完了しました',
      token,
      user: {
        id: user.id,
        email: user.email,
        user_id: user.user_id
      }
    });
  } catch (error) {
    console.error('Signup error:', error);
    res.status(500).json({ message: 'サーバーエラーが発生しました' });
  }
});

// ログイン
router.post('/login', [
  body('user_id').isLength({ min: 3 }).withMessage('idは3~20の1桁'),
  body('password').isLength({ min: 8 }).withMessage('パスワードは8文字以上の英数字、ハイフン、アンダースコアのみ使用可能です'),
], async (req, res) => {
  try {
    const errors = validationResult(req);
    if (errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    const { user_id, password } = req.body;

    // ユーザーの検索
    const result = await db.query(
      'SELECT id, email, user_id, password_hash FROM users WHERE user_id = ?',
      [user_id]
    );

    if (result.rows.length === 0) {
      return res.status(401).json({ message: 'ユーザーIDまたはパスワードが正しくありません' });
    }

    const user = result.rows[0];

    //パスワードの検証
    const isValid = await bcrypt.compare(password, user.password_hash);
    if (!isValid) {
      return res.status(401).json({ message: 'ユーザーIDまたはパスワードが正しくありません' });
    }

    const token = generateToken(user);

    res.json({
      message: 'ログインに成功しました',
      token,
      user: {
        id: user.id,
        email: user.email,
        user_id: user.user_id
      }
    });
  } catch (error) {
    console.error('Login error:', error);
    res.status(500).json({ message: 'サーバーエラーが発生しました' });
  }
});

// トークン検証
router.get('/verify', (req, res) => {
  const authHeader = req.headers.authorization;
  const token = authHeader && authHeader.split(' ')[1];

  if (!token) {
    return res.status(401).json({ valid: false });
  }

  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) {
      return res.status(401).json({ valid: false });
    }
    res.json({ valid: true, user });
  });
});

module.exports = router;
```

エントリーポイントの作成

src/server.js : メインサーバーファイルの作成

```
const express = require('express');
const path = require('path');
const helmet = require('helmet');
const cors = require('cors');
const dotenv = require('dotenv');

// 環境変数の読み込み
dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

// ミドルウェアの設定
app.use(helmet({
  contentSecurityPolicy: false,
  crossOriginEmbedderPolicy: false,
}));

app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// 静的ファイルの配信
app.use(express.static(path.join(__dirname, '../public')));

// ルーターのインポート
const authRoutes = require('./routes/auth');

// APIルートの設定
app.use('/api/auth', authRoutes);

// エラーハンドリング
app.use((err, req, res, next) => {
  console.error('Error:', err.stack);
  res.status(err.status || 500).json({
    message: err.message || 'Internal Server Error',
    ...(process.env.NODE_ENV === 'development' && { stack: err.stack })
  });
});

// 404ハンドラー
app.use((req, res) => {
  res.status(404).json({ message: 'Not Found' });
});

// サーバー起動
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
  console.log(`Environment: ${process.env.NODE_ENV}`);
});
```

package.jsonの設定

```
{  
  "scripts": {  
    "start": "node src/server.js",  
    "dev": "nodemon src/server.js",  
    "init-db": "node scripts/init-db.js"  
  }  
}
```

テスト

```
npm run dev
```

👉 <http://localhost:3000> にアクセスしてみましょう！

💡 **Curlリクエストをかける都合上、もう一つCLI(コマンドプロンプトやPowerShell)をもう一つ立ち上げます。**

Curlの基本形は

```
curl -X [GET|POST|PUT|DELETE] [URL] [-H "Content-Type: application/json"] [-d '{"key": "value"}']
```

です

curlの-Hオプションで指定できる主なContent-Typeは以下のようなものがあります

アプリケーション関連

```
application/json: JSONデータ
application/xml: XMLデータ
application/x-www-form-urlencoded: フォームデータ
application/octet-stream: バイナリデータ
application/pdf: PDFファイル
application/zip: ZIPファイル
```

テキスト関連

```
<div class="tiny-text">
text/plain: プレーンテキスト
text/html: HTMLデータ
text/csv: CSVファイル
text/javascript: JavaScriptファイル
```

マルチパート関連

multipart/form-data: ファイルアップロード時によく使用
multipart/mixed: 複数のコンテンツタイプを含む

メディア関連

```
<div class="tiny-text">  
image/jpeg: JPEGイメージ  
image/png: PNGイメージ  
image/gif: GIFイメージ  
audio/mpeg: MP3音声  
video/mp4: MP4動画  
</div>
```



```
curl -X POST http://localhost:3000/api/auth/signup \  
-H "Content-Type: application/json" \  
-d '{  
  "email": "test@example.com",  
  "password": "password123",  
  "user_id": "testuser"  
'
```

レスポンス例

```
{  
  "message": "ユーザー登録が完了しました",  
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",  
  "user": {  
    "id": 1,  
    "email": "test@example.com",  
    "user_id": "testuser"  
  }  
}
```

ログイン

```
curl -X POST http://localhost:3000/api/auth/login \  
  -H "Content-Type: application/json" \  
  -d '{  
    "user_id": "testuser",  
    "password": "password123"  
  }'
```

トークンの生成プロセス

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

// 1. シークレットキー（サーバーだけが知っている）
const JWT_SECRET = 'your-super-secret-key-12345';

// 2. トークンに含めるデータ
const userData = {
  id: 1,
  user_id: 'testuser',
  email: 'test@example.com'
};

// 3. トークンの生成
const token = jwt.sign(
  userData,           // ペイロードデータ
  JWT_SECRET,         // シークレットキー
  { expiresIn: '7d' } // オプション（7日間有効）
);

console.log(token);
// eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

同じユーザーでも、ログインするたびに異なるトークンが生成されます。

```
{  
  "id": 1,  
  "user_id": "testuser",  
  "iat": 1705123456,  // ← これが毎回変わる（現在時刻）  
  "exp": 1705728256  // ← これも毎回変わる（期限）  
}
```

重要な概念として、ステートレス(状態を持たない)というものがあります。

- サーバーにセッション情報を保存しない
- トークン自体に必要な情報が含まれている

ちなみに、トークンは平文で送信されるため、必ずHTTPS通信を使用する必要があります。(httpだと改竄可能性があります)

よりAPIとして使いやすくする

`http://localhost:3000/api/users/me` の情報をhtmlで読み込むようにすると、(ログインしているユーザごとに)動的にユーザ情報をページとして表示できるようになります。

src/routes/users.js

```
const express = require('express');
const db = require('../config/db');
const { authenticateToken } = require('../middleware/auth');

const router = express.Router();

// 現在のユーザー情報を取得（認証必要）
router.get('/me', authenticateToken, async (req, res) => {
  try {
    const query = `
      SELECT id, user_id, email, created_at
      FROM users
      WHERE id = ?
    `;

    const result = await db.query(query, [req.user.id]);

    if (result.rows.length === 0) {
      return res.status(404).json({ message: 'ユーザーが見つかりません' });
    }

    const user = result.rows[0];
    res.json({
      user: {
        id: user.id,
        user_id: user.user_id,
        email: user.email,
        created_at: user.created_at
      }
    });
  } catch (error) {
    console.error('Get user info error:', error);
    res.status(500).json({ message: 'サーバーエラーが発生しました' });
  }
});

module.exports = router;
```

src/server.js

```
// ルーターのインポート
const authRoutes = require('./routes/auth');
const usersRoutes = require('./routes/users'); // 追加

// APIルートの設定
app.use('/api/auth', authRoutes);
app.use('/api/users', usersRoutes); // 追加
```

テスト

```
# ログインしてトークンを取得（この部分エラーが吐かれないので改善の余地があるかも）
TOKEN=$(curl -s -X POST http://localhost:3000/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"user_id": "testuser", "password": "password123"}' \
  | grep -o '"token":"[^"]*"' | grep -o '["^"]*$')

# 認証付きでユーザー情報を取得
curl -X GET http://localhost:3000/api/users/me \
  -H "Authorization: Bearer $TOKEN"
```

public/login.html ログイン用のフロントエンド作ってみる

```
<!DOCTYPE html>
<html lang="ja">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>ログイン</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      max-width: 400px;
      margin: 50px auto;
      padding: 20px;
    }
    .form-group {
      margin-bottom: 15px;
    }
    label {
      display: block;
      margin-bottom: 5px;
    }
    input {
      width: 100%;
      padding: 8px;
      border: 1px solid #ddd;
      border-radius: 4px;
    }
    button {
      width: 100%;
      padding: 10px;
      background-color: #007bff;
      color: white;
      border: none;
      border-radius: 4px;
      cursor: pointer;
    }
    button:hover {
      background-color: #0056b3;
    }
    .error {
      color: red;
      margin-top: 10px;
    }
    .success {
      color: green;
      margin-top: 10px;
    }
  </style>
</head>
<body>
  <h2>ログイン</h2>
  <form id="loginForm">
    <div class="form-group">
      <label for="user_id">ユーザーID</label>
      <input type="text" id="user_id" required>
    </div>
    <div class="form-group">
      <label for="password">パスワード</label>
      <input type="password" id="password" required>
    </div>
    <button type="submit">ログイン</button>
  </form>
  <div id="message"></div>
</body>
<script>
  const API_BASE_URL = '/api';

  document.getElementById('loginForm').addEventListener('submit', async (e) => {
    e.preventDefault();

    const userId = document.getElementById('user_id').value;
    const password = document.getElementById('password').value;
    const messageDiv = document.getElementById('message');

    try {
      const response = await fetch(`${API_BASE_URL}/auth/login`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          user_id: userId,
          password: password
        })
      });

      const data = await response.json();

      if (response.ok) {
        // トークンをローカルストレージに保存
        localStorage.setItem('auth_token', data.token);
        localStorage.setItem('user_info', JSON.stringify(data.user));
        messageDiv.innerHTML = '<p class="success">ログイン成功！</p>';

        // 1秒後にリダイレクト
        setTimeout(() => {
          window.location.href = '/';
        }, 1000);
      } else {
        messageDiv.innerHTML = '<p class="error">${data.message}</p>';
      }
    } catch (error) {
      messageDiv.innerHTML = '<p class="error">ネットワークエラーが発生しました</p>';
    }
  });
</script>
</html>
```

講義中に触れたAPI Callを便利にするツールはこの辺り

実際にバックエンド開発ではよく使われています。

[HTTPIe – API testing client that flows with you](#)

[Postmanをダウンロード | 無料で始める](#)

原始的に、セッションはどのように管理されていたか (Apacheとサーブレットを例に)

サーバーサイドで管理する仕組み

ユーザー情報をサーバー側でDBなどで保存しておいて、エージェントなどの識別を行う

クライアント側に情報を持たせて通信しよう

JWTとは (この講義では基本的にはこれを利用してアプリを組んでいく)

ブラウザ (Client)

サーバー (Express.js)

mypage.htmlをチェック
→ 認証が必要なページ

```
<--- 2. 200 OK (mypage.html) -----  
      ただしJavaScriptで認証チェック
```

```
if (!localStorage.getItem('auth_token')) {
  window.location.href = '/login/';
}
```

```
3. GET /login/ ----->
   (リダイレクト)
```

```
<--- 4. 200 OK (login.html) -----
```



```
=====
2. ログイン処理
=====

ブラウザ (Client)                                サーバー (Express.js)

[ユーザーがフォームに入力]
- user_id: testuser
- password: password123

5. POST /api/auth/login -----> |
Content-Type: application/json
Body: {
  "user_id": "testuser",
  "password": "password123"
}

[サーバー処理]
1. リクエストボディを解析
2. データベースでユーザー検索
   SELECT * FROM users
   WHERE user_id = 'testuser'
3. パスワード検証
   bcrypt.compare(password, hash)
4. JWT生成
   jwt.sign({
     id: 1,
     user_id: 'testuser',
     email: 'test@example.com'
   }, SECRET, {expiresIn: '7d'})

<--- 6. 200 OK (JSON) ----- |
{
  "message": "ログインに成功しました",
  "token": "eyJhbGciOiJIUzI1NiIs... ",
  "user": {
    "id": 1,
    "user_id": "testuser",
    "email": "test@example.com"
  }
}

[ブラウザでトークン保存]
localStorage.setItem('auth_token', response.token);
localStorage.setItem('user_info', JSON.stringify(response.user));

7. GET / -----> |
(リダイレクト to メインページ)

<--- 8. 200 OK (index.html) ----- |
```

```
=====
3. 認証が必要なAPIへのアクセス
=====

ブラウザ (Client)                                サーバー (Express.js)
|                                                    |
[JavaScriptでAPI呼び出し]                          |
const token = localStorage.getItem('auth_token');  |
|                                                    |
9. GET /api/users/me -----> |
   Authorization: Bearer eyJhbGciOiJIUzI1NiIs...  |
|                                                    |
[認証ミドルウェア処理]                             |
authenticateToken()                               |
1. ヘッダーからトークン取得                       |
   Bearer eyJhbGci...                             |
2. トークンを検証                                 |
   jwt.verify(token, SECRET)                      |
3. 検証成功                                       |
   req.user = {                                   |
     id: 1,                                       |
     user_id: 'testuser',                       |
     email: 'test@example.com',                 |
     iat: 1705123456,                           |
     exp: 1705728256                           |
   }                                             |
4. next() で次の処理へ                           |
|                                                    |
[ルートハンドラ処理]                             |
router.get('/me', ...)                          |
req.user が使える!                             |
|                                                    |
<--- 10. 200 OK (JSON) ----- |
    {                                           |
      "user": {                                |
        "id": 1,                               |
        "user_id": "testuser",                 |
        "email": "test@example.com",           |
        "created_at": "2024-01-15T10:30:00.000Z" |
      }                                         |
    }                                         |
|                                                    |
[JavaScriptでDOM更新]                             |
document.getElementById('username').textContent = user.user_id; |
```

OAuth認証とは？

他のサービスのアカウントでログインする仕組みです。

「Googleでログイン」「GitHubでログイン」などがこれです。

メリット

パスワードを新しく作らなくていい

信頼できるサービスに認証を任せられる

ユーザーが使いやすい

OAuth認証の流れ

「Googleでログイン」ボタンを押す

Googleのログイン画面に移動

Googleで認証する

元のサービスに戻ってくる

ログイン完了

ユーザーのパスワードは元のサービスには渡りません。

類似技術 OAuth2.0

サービスのOAuthログイン自体には関わりませんが、APIの認証などで、一度ユーザ側のログインを行い、トークンをサーバーに送ることで認証を行う方法があります。

Twitter(X)のAPIはこの仕組みを利用しています。

詳しくはこちらの同人誌が実装のイメージが沸きやすいと思います。

<https://www.authya.com>

ログインOuthのサービスとしてはSNSだと

- Google
- Twitter(X)
- Facebook
- に加えて、
- Apple
- Yahoo!ID
- GitHub
- Line
- Azure AD
- instagram

などがよく見かけると思います。

Twitter(X)はAPIの料金が高くなり、気軽に「Twitterでログイン」は設けにくくなってい

ログインOuthでは、そのサービスから情報を引っ張って来れるパターンもあります。
Google ログイン を行ってもらった上でメールアドレスを取得しておいて、ユーザの初
回ログイン時にはメールも設定されている、
というようなサービス構成も作れます。
その上でパスワード設定も念の為お願いするようにするとユーザも管理側も便利なロ
グイン周りを構築しやすいでしょう。

BASIC認証とは？

最もシンプルな認証方式です。

ユーザー名とパスワードを毎回送信します。

特徴

HTTPヘッダーに認証情報を含める

Base64でエンコード（暗号化ではない） → HTTPSを利用する必要があります。

単純な方法で実装も簡単ですが、現在では安全性に問題があります。

JWTはどこに持つか？

ブラウザの場合

localStorage - JavaScriptで簡単に扱える

sessionStorage - タブを閉じると消える

Cookie - サーバーに自動送信される

curlの場合

シェル変数に保存します

ブラウザはないので、メモリ上で管理

sessionStorageの使い方としては、フォームに入力した情報を保存することが多いです。

一般的な申し込みフォームに加え、チケット購入システムなどでも便利に使われます。

2FA(二段階認証)の種類

1. SMS認証

電話番号にコードを送信

2. 認証アプリ（TOTP）

Google AuthenticatorやAuthyなど

30秒ごとに変わる6桁の数字

(オフラインでも使える)

3. ハードウェアキー

YubiKeyなど物理的なデバイス

最も安全だが、コストが高い

4. Passkey

ユーザーの生体認証情報を使って認証

? 認証と認可の違い

建物に例えると

認証 = 入館証で本人確認

認可 = どの部屋に入れるかの権限確認

入館証をかざす（認証）

3階の会議室に入れるか確認（認可）


段階と、ユーザ側とシステム側どちらが主体かが異なります。

ユーザの情報は必要がなければできるだけ取得をしない方針で考えた方が良いでしょう。

個人情報三原則

個人情報を

- 作らない
- 取得しない
- 保持しない

 クレジットカードの情報は特に法令に則ったサービスに委託するのがよいでしょう。

決済が必要な場合は Pay.jpやStripeなどを利用しましょう。

Pay.jpはgRPCに対応しているので、WebAPIとしてのセキュリティが高めで講師はお気に入りです。(HTTP/2を利用します)

DBの閲覧ソフトウェアを導入すると便利

- MySQL Workbench
- SQLite Browser
- DB Browser
- DBeaver
- Metabase
- pgAdmin (ブラウザでDBを確認する / PHPの開発で人気)
- Grafana(BI / ダッシュボードツール)
- Tbls (リレーション可視化ツール)

一部、TCP/IPやsshを理解する必要があります。

次回の話題

- 実際にログインユーザごと表示されるものが変わる例
- ログインユーザでないとデータが表示されない例
- 画像(マルチメディアファイル)のアップロードをどうするか
- 三層アーキテクチャとWebサーバの存在

先週のご相談

Q1 オフライン調査ツール

Webアプリケーションを使った現地調査ツールで、電波が入らない場所での調査が必要なことが多い。(理想としては一度ダウンロードしておくで現地で開いた時に記録などができるツール)

Q2 Webサーバを立ち上げずに処理できるサービス

自治体の職員さんなどに使ってもらうときに、インストールやWebへの接続がなく使えるツールを提供したい。

A1

理論的には、各種ライブラリ、pbfをキャッシュしておけばオフラインでリロードしても一定範囲の地図は読み込めるはず...

→ Service Workerで行けるかも??

(次点PWAとか：スターバックスのアプリケーションなどがこれを採用)

MacのSafariの例ではあるが、試験的な機能の有効化が必要な場合もあるかもしれません。



A2

FileAPIが行けるのでは...?

HTTP(S)で使わずに、ブラウザで `file://` を使ってhtmlを表示してローカルデータを触る方法。

あるいはローカルアプリとしても.exeとしてバンドルすることでGUIを使ったりと幅の広い提案ができそう。

紹介した書籍

Docker&仮想サーバー完全入門 Webクリエイター&エンジニアの作業がはかどる開発環境構築ガイド - インプレスブックス

作りながら学ぶ Webシステムの教科書 | 日経BOOKプラス

JavaScript コードレシピ集 | 技術評論社