補足資料: ブラウザストレージ比較

基本(1分) - ブラウザストレージとは

概要: Webアプリケーションがユーザーのブラウザ内にデータを保存・管理するための仕組みです。これにより、オフラインでの動作、ユーザー設定の記憶、パフォーマンス向上などが可能になります。主に4つのストレージ技術が利用されます。

4つの主要ストレージ:

- `localStorage`: 永続的なキーと値のペアを保存します。
- `sessionStorage`: タブまたはウィンドウのセッション中のみ有効なキーと値のペアを保存します。
- `Cookie`: 少量データを保存し、サーバーとの通信にも利用されます。
- `IndexedDB`: 大量の構造化データを保存できるブラウザ内データベースです。

基本的な使い分けイメージ:

```
// ユーザー設定（例: テーマカラー）を永続的に保存
localStorage.setItem('theme', 'dark');

// 一時的なフォーム入力内容（例: 未送信のコメント）をタブを閉じるまで保存
sessionStorage.setItem('commentDraft', 'これは下書きです...');

// ログイン状態を示すセッショントークンを保存（サーバー送信あり）
// document.cookie = 'sessionId=user123abc; Secure; HttpOnly; SameSite=Strict'; // 設定例

// 大量のアプリケーションデータ（例: オフライン記事）をデータベースに保存
// const request = indexedDB.open('myArticlesDB', 1); // 設定例
```

詳細(10分) - 各ストレージの仕組みと特徴

1. `localStorage`

- **概要**: 文字列ベースのキーと値のペアを永続的に保存します。ブラウザを閉じてもデータは消えません。
- **容量**: 通常 5MB～10MB 程度（ブラウザにより異なる）。

- **ライフサイクル**: JavaScriptで明示的に削除するまで永続します。
- **スコープ**: 同一オリジン（プロトコル + ドメイン + ポート）内でのみ共有されます。
- **API**: 同期的（処理が完了するまで他の処理をブロックする）。シンプルで使いやすいAPIです。
- **データの種類**: 文字列のみ。オブジェクトや配列を保存する場合は、`JSON.stringify()` で文字列に変換し、読み出す際に `JSON.parse()` で元に戻す必要があります。

コード例:

```
// 保存（オブジェクトはJSON文字列に変換）
const userSettings = { notifications: true, language: 'ja' };
localStorage.setItem('userSettings', JSON.stringify(userSettings));

// 取得（JSON文字列をオブジェクトにパース）
const storedSettingsString = localStorage.getItem('userSettings');
const loadedSettings = storedSettingsString ? JSON.parse(storedSettingsString) : { notifications: false, language: 'en' };
console.log('読み込んだ設定:', loadedSettings);

// 特定のアイテムを削除
localStorage.removeItem('userSettings');

// 全てのアイテムを削除（同一オリジン内のlocalStorage全て）
// localStorage.clear();

// 格納されているアイテム数を取得
console.log('localStorageのアイテム数:', localStorage.length);

// n番目のキーを取得（順序は保証されないことが多い）
// const firstKey = localStorage.key(0);
// console.log('最初のキー:', firstKey);
```

セキュリティ上の注意点:

- **XSS（クロスサイトスクリプティング）リスク**: `localStorage` に保存されたデータは、同じオリジンで動作するJavaScriptから容易にアクセス可能です。もしサイトにXSS脆弱性がある場合、悪意のあるスクリプトによって `localStorage` の内容が盗まれたり改ざんされたりする可能性があります。
- **機密情報の非推奨**: パスワードやクレジットカード番号のような機密性の高い情報は `localStorage` に保存すべきではありません。

2. sessionStorage

- **概要**: `localStorage` と同様に文字列ベースのキーと値のペアを保存しますが、データは現在のブラウザタブ（またはウィンドウ）のセッション中のみ有効です。タブを閉じるとデータは消去されます。
- **容量**: 通常 5MB～10MB 程度（`localStorage` と同程度）。

- **ライフサイクル**: ブラウザのタブまたはウィンドウが開いている間のみ。タブを閉じるとデータは消えます。
- **スコープ**: タブごと。同じオリジンのサイトであっても、異なるタブ間で `sessionStorage` は共有されません。
- **API**: `localStorage` と全く同じメソッド（`setItem`, `getItem`, `removeItem`, `clear`, `key`, `length`）を持ち、同様に同期的です。
- **データの種類**: 文字列のみ（`localStorage` と同様にJSON変換が必要）。

コード例:

```
// フォームの一時的な入力内容を保存
const formInput = { name: '田中太郎', email: 'tanaka@example.com' };
sessionStorage.setItem('formData', JSON.stringify(formInput));

// ページ再読み込み時などに復元
const storedFormString = sessionStorage.getItem('formData');
if (storedFormString) {
  const loadedForm = JSON.parse(storedFormString);
  console.log('復元したフォームデータ:', loadedForm);
  // document.getElementById('nameInput').value = loadedForm.name; // 例
}

// タブを閉じると自動的に消えるため、明示的な削除は必須ではないが、不要になった時点で削除も可能
// sessionStorage.removeItem('formData');
```

セキュリティ上の注意点:

- `localStorage` と同様にXSS脆弱性の影響を受けます。機密情報の保存は避けてください。

3. `Cookie` （クッキー）

- **概要**: 小さなテキストデータを保存し、主にサーバーとクライアント間で状態を維持するために使用されます（例: ログインセッション管理）。HTTPリクエストヘッダーに自動的に含まれてサーバーに送信される特徴があります。
- **容量**: 約4KBまでと非常に小さいです。
- **ライフサイクル**: 有効期限（`Expires` または `Max-Age` 属性）を設定でき、永続的なものもセッション限りのものも作成可能です。指定がなければセッション限りです。
- **スコープ**: ドメインとパスによって制御されます。サブドメイン間での共有も可能です。
- **API**: `document.cookie` を介してアクセスしますが、操作はやや煩雑です。通常、ヘルパー関数やライブラリが使われます。
- **データの種類**: 文字列のみ。

コード例（基本的な操作）:

```
// Cookieを設定（名前=値; 属性1=値1; 属性2=値2 ...）
function setCookie(name, value, daysToExpire, path = '/', domain = '') {
  let cookieString = `${encodeURIComponent(name)}=${encodeURIComponent(value)}`;
  if (daysToExpire) {
    const date = new Date();
    date.setTime(date.getTime() + (daysToExpire * 24 * 60 * 60 * 1000));
    cookieString += `; expires=${date.toUTCString()}`;
  }
  cookieString += `; path=${path}`;
  if (domain) {
    cookieString += `; domain=${domain}`;
  }
  // セキュリティ属性の追加を推奨（後述）
  // cookieString += '; Secure; SameSite=Strict';
  document.cookie = cookieString;
  console.log(`Cookieを設定: ${name}`);
}

// Cookieを取得
function getCookie(name) {
  const nameEQ = encodeURIComponent(name) + "=";
  const ca = document.cookie.split(';');
  for (let i = 0; i < ca.length; i++) {
    let c = ca[i];
    while (c.charAt(0) === ' ') c = c.substring(1, c.length);
    if (c.indexOf(nameEQ) === 0) return decodeURIComponent(c.substring(nameEQ.length, c.length));
  }
  return null;
}

// 使用例
setCookie('username', 'JohnDoe', 7); // 7日間有効なCookie
const username = getCookie('username');
console.log('読み込んだCookie (username):', username);

// Cookieを削除（有効期限を過去に設定）
// setCookie('username', '', -1);
```

セキュリティ属性（非常に重要）:

- `HttpOnly`: JavaScriptからの `document.cookie` によるアクセスを禁止します。サーバーサイドでのみCookieを扱う場合に設定し、XSS攻撃によるCookie盗難リスクを大幅に軽減します。
- `Secure`: HTTPS接続の場合のみCookieを送信するようにします。これにより、中間者攻撃によるCookie盗難を防ぎます。
- `SameSite`: CSRF（クロスサイトリクエストフォージェリ）攻撃を緩和します。値には `Strict`（最も厳格）, `Lax`（デフォルトになる傾向）, `None`（`Secure`属性必須）があります。
  - 例: `document.cookie = "sessionId=abc123xyz; Secure; HttpOnly; SameSite=Strict";`

セキュリティ上の注意点:

- **CSRFリスク**: `SameSite`属性を適切に設定しないと、CSRF攻撃の標的となる可能性があります。
- **XSSリスク**: `HttpOnly`属性がない場合、XSS脆弱性があるとCookieが盗まれる可能性があります。
- **情報漏洩**: `Secure`属性がない場合、HTTP通信経由でCookieが盗聴される可能性があります。
- サーバーに毎回送信されるため、不要なデータはCookieに含めないようにし、パフォーマンスへの影響も考慮します。

4. IndexedDB

- **概要**: 大量の構造化データ（オブジェクト、ファイルなど）をブラウザ内に保存できるNoSQLデータベースです。オフラインアプリケーションや複雑なデータ操作に適しています。
- **容量**: 数十MB～数GB、あるいはそれ以上（ユーザーのディスク空き容量やブラウザ設定に依存）。`localStorage`などよりはるかに大容量です。
- **ライフサイクル**: JavaScriptで明示的に削除するまで永続します。
- **スコープ**: 同一オリジン内。
- **API**: 非同期（Promiseベースまたはイベントベース）。`localStorage`などより複雑ですが、トランザクション、インデックス、カーソルといった高度なデータベース機能を提供します。
- **データの種類**: JavaScriptのオブジェクト、配列、プリミティブ値、File/Blobオブジェクトなど、多くのデータ型を直接保存できます（内部的には構造化複製アルゴリズムでシリアライズ）。

コード例（基本的なCRUD操作）:

```
const dbName = 'MyTodoDB';
const dbVersion = 1;
let db;

// 1. データベースを開く（または作成）
function openDB() {
  return new Promise((resolve, reject) => {
    const request = indexedDB.open(dbName, dbVersion);

    request.onerror = (event) => {
      console.error('データベースを開けませんでした:', event.target.error);
      reject(event.target.error);
    };

    request.onsuccess = (event) => {
      db = event.target.result;
      console.log('データベースを開きました。', db);
      resolve(db);
    };

    // データベースのバージョンが古い場合や新規作成時に実行
    request.onupgradeneeded = (event) => {
      const tempDb = event.target.result;
      console.log('データベースのアップグレード/作成中...');
```

```
      if (!tempDb.objectStoreNames.contains('tasks')) {
        // 'tasks' オブジェクトストアを作成（テーブルに相当）
        // 'id' をキーパス（主キー）とし、自動インクリメントを設定
        const taskStore = tempDb.createObjectStore('tasks', { keyPath: 'id', autoIncrement: true });
        // 'dueDate' プロパティにインデックスを作成（検索やソート用）
        taskStore.createIndex('dueDateIndex', 'dueDate', { unique: false });
        console.log(''tasks' オブジェクトストアとインデックスを作成しました。');
      }
    };
  });
}

// 2. データを追加 (Create)
async function addTask(taskObject) {
  if (!db) await openDB();
  return new Promise((resolve, reject) => {
    // 'readwrite' トランザクションを開始
    const transaction = db.transaction(['tasks'], 'readwrite');
    const store = transaction.objectStore('tasks');
    const request = store.add(taskObject); // { text: '買い物', dueDate: '2024-12-31' }

    request.onsuccess = (event) => {
      console.log('タスクを追加しました。ID:', event.target.result);
      resolve(event.target.result); // 追加されたアイテムのキーを返す
    };
    request.onerror = (event) => {
      console.error('タスク追加エラー:', event.target.error);
      reject(event.target.error);
    };
  });
}

// 3. データを取得 (Read)
async function getTask(taskId) {
  if (!db) await openDB();
  return new Promise((resolve, reject) => {
    const transaction = db.transaction(['tasks'], 'readonly');
    const store = transaction.objectStore('tasks');
    const request = store.get(taskId);

    request.onsuccess = (event) => {
      console.log('タスクを取得しました:', event.target.result);
      resolve(event.target.result); // タスクオブジェクトまたはundefined
    };
    request.onerror = (event) => {
      console.error('タスク取得エラー:', event.target.error);
      reject(event.target.error);
    };
  });
}

// 4. データを更新 (Update)
async function updateTask(updatedTaskObject) { // updatedTaskObject は id を含むこと
  if (!db) await openDB();
  return new Promise((resolve, reject) => {
```

```
    const transaction = db.transaction(['tasks'], 'readwrite');
    const store = transaction.objectStore('tasks');
    // put はキーが存在すれば更新、存在しなければ新規追加する
    const request = store.put(updatedTaskObject);

    request.onsuccess = (event) => {
      console.log('タスクを更新しました。ID:', event.target.result);
      resolve(event.target.result);
    };
    request.onerror = (event) => {
      console.error('タスク更新エラー:', event.target.error);
      reject(event.target.error);
    };
  });
}

// 5. データを削除 (Delete)
async function deleteTask(taskId) {
  if (!db) await openDB();
  return new Promise((resolve, reject) => {
    const transaction = db.transaction(['tasks'], 'readwrite');
    const store = transaction.objectStore('tasks');
    const request = store.delete(taskId);

    request.onsuccess = () => {
      console.log(`タスクID: ${taskId} を削除しました。`);
      resolve();
    };
    request.onerror = (event) => {
      console.error('タスク削除エラー:', event.target.error);
      reject(event.target.error);
    };
  });
}

// IndexedDB使用例
(async () => {
  try {
    await openDB();
    const newTaskId = await addTask({ text: '読書をする', dueDate: '2024-08-15', completed: false });
    const task = await getTask(newTaskId);
    if (task) {
      task.completed = true;
      await updateTask(task);
    }
    // await deleteTask(newTaskId);
  } catch (error) {
    console.error('IndexedDB操作中にエラー:', error);
  }
})();
```

セキュリティ上の注意点:

- オリジンによって分離されているため、他のサイトから直接アクセスされることはありません。

- しかし、サイト自体にXSS脆弱性がある場合、そのサイトのオリジンで動作する悪意のあるスクリプトによってIndexedDB内のデータが操作される可能性があります。
- 機密情報は暗号化して保存することを検討してください。

比較表(5分) - 各ストレージ技術の比較

特徴	`localStorage`	`sessionStorage`	`Cookie`	`IndexedDB`
主な用途例	ユーザー設定、テーマ、オフライン時の小規模データ	フォームの一時入力、タブ固有の状態	認証トークン、セッション管理、トラッキング	大量データ、オフラインアプリ、複雑なクエリが必要なデータ
容量	5-10MB	5-10MB	約4KB	数十MB～数GB以上（ディスク空き容量依存）
永続性	永続（明示的削除まで）	セッション中（タブ/ウィンドウを閉じると消滅）	有効期限設定可（設定なければセッション中）	永続（明示的削除まで）
サーバーへの自動送信	なし	なし	あり（HTTPリクエストヘッダーに含まれる）	なし
APIの種類	同期	同期	同期（ただし `document.cookie` の操作は煩雑）	非同期（Promiseベースまたはイベントベース）
データの種類	文字列のみ（JSONでシリアライズ/デシリアライズ）	文字列のみ（JSONでシリアライズ/デシリアライズ）	文字列のみ	構造化データ（オブジェクト、配列、File/Blob等、多くの型を直接保存可能）
スコープ	オリジン単位	タブ単位	ドメイン/パス単位（設定可）	オリジン単位
セキュリティ主な注意点	XSSに注意、機密情報非推奨	XSSに注意、機密情報非推奨	XSS、CSRFに注意。`HttpOnly`, `Secure`, `SameSite`属性が重要。機密情報非推奨	XSSに注意。オリジン分離。機密情報は暗号化検討

特徴	localStorage	sessionStorage	Cookie	IndexedDB
API複雑度	低	低	中（API自体はシンプルだが扱いが煩雑）	高（多機能だが学習コストも高い）
検索/クエリ機能	なし（全件走査）	なし（全件走査）	なし（全件走査）	あり（インデックスを利用した効率的な検索、範囲指定、カーソル）

深掘り（5分） - 発展的なストレージ技術と考慮事項

1. localStorage と storage イベント

localStorage の内容が同じオリジンの他のタブ/ウィンドウで変更された場合、storage イベントが発火します。これを利用して、タブ間で簡易的なデータ同期が可能です。

```
// タブA（データを変更する側）
localStorage.setItem('sharedCounter', parseInt(localStorage.getItem('sharedCounter') || '0') + 1);

// タブB（変更を監視する側）
window.addEventListener('storage', (event) => {
  if (event.key === 'sharedCounter') {
    console.log('sharedCounterが変更されました:');
    console.log('  古い値:', event.oldValue);
    console.log('  新しい値:', event.newValue);
    console.log('  変更元URL:', event.url);
    // document.getElementById('counterDisplay').textContent = event.newValue;
  }
});
```

注意: sessionStorage は storage イベントを発火しません。また、変更を行ったタブ自身ではこのイベントは通常発火しません。

2. IndexedDB のトランザクション

IndexedDB の操作はすべてトランザクション内で行われます。トランザクションは、一連のデータベース操作がすべて成功するか、すべて失敗するか（アトミック性）を保証し、データの一貫性を保ちます。

- **種類**: `readonly`（読み取り専用）、`readwrite`（読み書き可能）。
- **スコープ**: トランザクションが対象とするオブジェクトストアを指定します。
- **ライフサイクル**: `complete`（成功）、`abort`（失敗/明示的中止）、`error`（エラー発生）イベントで管理されます。

```
async function transferFunds(fromAccountId, toAccountId, amount) {
  if (!db) await openDB(); // dbは事前にopenDB()で初期化されているとする

  // 'accounts' オブジェクトストアに対する読み書きトランザクションを開始
  const transaction = db.transaction(['accounts'], 'readwrite');
  const store = transaction.objectStore('accounts');

  return new Promise((resolve, reject) => {
    transaction.oncomplete = () => {
      console.log('資金移動トランザクションが正常に完了しました。');
      resolve();
    };
    transaction.onerror = (event) => {
      console.error('資金移動トランザクションエラー:', event.target.error);
      reject(event.target.error);
    };
    transaction.onabort = () => {
      console.warn('資金移動トランザクションが中止されました。');
      reject(new Error('Transaction aborted'));
    };

    // 1. 送金元口座の残高を取得
    const getFromRequest = store.get(fromAccountId);
    getFromRequest.onsuccess = () => {
      const fromAccount = getFromRequest.result;
      if (!fromAccount || fromAccount.balance < amount) {
        console.error('送金元口座の残高不足または口座が存在しません。');
        transaction.abort(); // トランザクションを中止
        return;
      }

      // 2. 送金先口座の残高を取得
      const getToRequest = store.get(toAccountId);
      getToRequest.onsuccess = () => {
        const toAccount = getToRequest.result;
        if (!toAccount) {
          console.error('送金先口座が存在しません。');
          transaction.abort();
          return;
        }

        // 3. 残高を更新
        fromAccount.balance -= amount;
        toAccount.balance += amount;

        // 4. 更新された口座情報を保存
```

```
      const updateFromRequest = store.put(fromAccount);
      updateFromRequest.onerror = () => transaction.abort(); // エラー時中止

      const updateToRequest = store.put(toAccount);
      updateToRequest.onerror = () => transaction.abort(); // エラー時中止
    };
    getToRequest.onerror = () => transaction.abort();
  };
  getFromRequest.onerror = () => transaction.abort();
});
}
```

3. Cache API と Service Worker (PWA関連)

これらは主にプログレッシブウェブアプリ (PWA) の文脈で、オフライン対応やパフォーマンス向上のために使われます。

- **Cache API**: HTTPリクエストとレスポンスのペアをキャッシュします。静的アセット (HTML, CSS, JS, 画像) のキャッシュに強力です。
- **Service Worker**: ブラウザのバックグラウンドで実行されるスクリプトで、ネットワークリクエストのプロキシ、プッシュ通知、バックグラウンド同期などを行います。Cache APIと連携してオフラインキャッシュ戦略を実装できます。

これらの技術は本資料の主眼である4大ストレージとは少し毛色が異なりますが、高度なWebアプリケーションでは組み合わせて利用されることがあります。詳細はPWA関連の資料を参照してください。

4. ストレージ容量の確認と管理

- **`navigator.storage.estimate()`**: オリジンが使用しているストレージ容量 (`usage`) と利用可能な最大容量 (`quota`) の見積もりを非同期で取得できます。

```
if (navigator.storage && navigator.storage.estimate) {
  navigator.storage.estimate().then(estimate => {
    console.log(`使用容量: ${estimate.usage / 1024 / 1024} MB`);
    console.log(`最大容量: ${estimate.quota / 1024 / 1024} MB`);
  });
}
```

- **容量超過エラー**: `localStorage` や `sessionStorage` で容量制限を超えると `QuotaExceededError` (または同様のエラー) が発生します。`IndexedDB` でも容量関連のエラーが発生し得ます。これらのエラーを適切に `try...catch` で捕捉し、ユーザーに通知したり、古いデータを削除したりする戦略が必要です。

実践応用(5分) - ストレージ技術の組み合わせシナリオ

シナリオ1: ユーザー設定の永続化と一時的なフォーム入力

- **目的**: ユーザーの見た目設定（例: ダークモード）を記憶し、複数ステップのフォーム入力中にタブを閉じても内容が失われないようにする。
- **使用技術**:
  - `localStorage`: テーマ設定（例: `'theme': 'dark'`）を永続保存。
  - `sessionStorage`: フォームの各ステップの入力内容（例: `'step1Data': '{...}'`, `'step2Data': '{...}'`）を一時保存。

```
// --- テーマ設定 (localStorage) ---
const themeToggleButton = document.getElementById('theme-toggle');
let currentTheme = localStorage.getItem('appTheme') || 'light';

function applyTheme(theme) {
  document.body.className = theme + '-theme';
  localStorage.setItem('appTheme', theme);
  currentTheme = theme;
  if (themeToggleButton) themeToggleButton.textContent = theme === 'light' ? 'ダークモードへ' : 'ライトモードへ';
}

if (themeToggleButton) {
  themeToggleButton.addEventListener('click', () => {
    applyTheme(currentTheme === 'light' ? 'dark' : 'light');
  });
}
applyTheme(currentTheme); // 初期テーマ適用

// --- フォーム一時保存 (sessionStorage) ---
const formStep1 = document.getElementById('form-step1-input');
const formStep2 = document.getElementById('form-step2-input');

// ページ読み込み時にsessionStorageから復元
if (formStep1) formStep1.value = sessionStorage.getItem('formStep1Draft') || '';
if (formStep2) formStep2.value = sessionStorage.getItem('formStep2Draft') || '';

// 入力時にsessionStorageに保存
if (formStep1) {
  formStep1.addEventListener('input', () => {
    sessionStorage.setItem('formStep1Draft', formStep1.value);
  });
}
if (formStep2) {
  formStep2.addEventListener('input', () => {
    sessionStorage.setItem('formStep2Draft', formStep2.value);
  });
}
```

```
// フォーム送信成功時にクリア
// document.getElementById('submit-button').addEventListener('click', () => {
//   sessionStorage.removeItem('formStep1Draft');
//   sessionStorage.removeItem('formStep2Draft');
// });
```

選択理由: テーマ設定は永続性が求められるため `localStorage`。フォーム入力はセッション限りでよく、タブ間で共有する必要がないため `sessionStorage` が適しています。

シナリオ2: ToDoリストアプリケーション（データはIndexedDB、設定はlocalStorage）

- **目的**: ToDoアイテムを大量に保存・管理し、表示件数などの設定も記憶する。
- **使用技術**:
  - `IndexedDB`: ToDoアイテム（ID, テキスト, 完了状態,期日など）を構造化データとして保存。
  - `localStorage`: 表示設定（例: 1ページあたりの表示件数、ソート順）を保存。

```
// --- ToDoアイテム管理（IndexedDB - 前述のIndexedDBコード例を参照）---
// (async () => {
//   await openDB(); // IndexedDBの初期化
//   await addTask({ text: '牛乳を買う', dueDate: '2024-08-10', completed: false });
//   const tasks = await getAllTasks(); // IndexedDBから全タスク取得する関数（別途実装）
//   renderTasks(tasks); // タスクを描画する関数（別途実装）
// })();

// --- 表示設定（localStorage）---
const itemsPerPageSelect = document.getElementById('items-per-page');
let itemsPerPage = parseInt(localStorage.getItem('todoSettings_itemsPerPage') || '10');

function applyItemsPerPageSetting(count) {
  itemsPerPage = count;
  localStorage.setItem('todoSettings_itemsPerPage', count.toString());
  // renderTasks(tasks, itemsPerPage); // タスク再描画ロジック
  console.log(`1ページあたり ${count} 件表示に設定しました。`);
}

if (itemsPerPageSelect) {
  itemsPerPageSelect.value = itemsPerPage.toString();
  itemsPerPageSelect.addEventListener('change', (event) => {
    applyItemsPerPageSetting(parseInt(event.target.value));
  });
}

console.log(`初期表示件数: ${itemsPerPage}`);
```

選択理由: ToDoアイテムは数が多くなったり、複雑な検索が必要になったりする可能性があるため、大容量で高機能な `IndexedDB` が適しています。

表示設定のような少量の単純なデータは `localStorage` で手軽に扱えます。

関連する補足資料

- **補足資料: localStorage API詳解** - `localStorage` のより詳細な使い方。
- **補足資料: JSON操作詳解** - 各ストレージでデータを文字列として扱う際の `JSON.parse` / `JSON.stringify` の詳細。
- **補足資料: データ構造設計** - ストレージに保存するデータの設計方法。
- **補足資料: try-catch詳解** - ストレージ操作時のエラーハンドリング。