掲示板システム設計演習 - 授業フロー

前半：基礎復習

これまでの復習

- EC2, Lambda, DynamoDB, S3の役割おさらい
- 各サービスの特徴と使い分け
- 今回作るもの全体像の提示（掲示板システム）

AJAX基礎

1. 同期処理の問題点

```
// 従来型：ページ全体リロード
<form action="/submit" method="POST">
  <button type="submit">送信</button>
</form>
// → 画面が真っ白になってチカチカする
```

2. AJAXとは

- 非同期でサーバーと通信
- ページはそのまま、必要な部分だけ更新
- ユーザー体験が向上

3. fetch API - 2つの書き方

パターン1：Promise（.then）チェーン

```
// GET: データ取得
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
    console.log('レスポンス受信:', response);
    return response.json(); // JSONに変換
  })
  .then(data => {
    console.log('データ:', data);
    document.getElementById('result').textContent = data.title;
  })
  .catch(error => {
    console.error('エラー:', error);
    alert('通信エラーが発生しました');
  });

console.log('この行は先に実行される（非同期）');
```

特徴：

- ストリーミング的に処理が流れる
- `.then()` で処理を繋げていく
- 昔からある書き方、ブラウザ互換性高い

パターン2：async/await

```
// 同じ処理をasync/awaitで
async function getPosts() {
  try {
    console.log('通信開始');

    const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
    console.log('レスポンス受信:', response);

    const data = await response.json();
    console.log('データ:', data);

    document.getElementById('result').textContent = data.title;

  } catch (error) {
    console.error('エラー:', error);
    alert('通信エラーが発生しました');
  }
}

// 関数を呼び出す
getPosts();

console.log('この行は先に実行される（非同期）');
```

特徴：

- 同期処理のように読みやすい
- `await` で結果を待つ
- モダンな書き方、コードが直感的

4. 両者の比較（実演）

```
// ========================================
// パターン1：.then チェーン
// ========================================
function getPostThen() {
  fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => response.json())
    .then(data => {
      console.log('[.then] タイトル:', data.title);
    })
```

```
    .catch(error => {
      console.error('[.then] エラー:', error);
    });
}

// ========================================
// パターン2：async/await
// ========================================
async function getPostAwait() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
    const data = await response.json();
    console.log('[await] タイトル:', data.title);
  } catch (error) {
    console.error('[await] エラー:', error);
  }
}

// どちらも同じ結果
getPostThen();
getPostAwait();
```

どちらを使う？

- 簡単な処理 → どちらでもOK
- 複数の処理を順番に → async/awaitが読みやすい
- イベントハンドラ内 → async/awaitが便利

5. POSTリクエスト（両パターン）

.thenパターン

```
function createPost() {
  fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      title: 'AWSの質問',
      body: 'Lambdaについて教えてください',
      userId: 1
    })
  })
    .then(response => response.json())
    .then(data => {
      console.log('作成成功:', data);
      alert('投稿ID: ' + data.id);
    })
    .catch(error => {
      console.error('エラー:', error);
```

```
    });
}
```

async/awaitパターン

```
async function createPost() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        title: 'AWSの質問',
        body: 'Lambdaについて教えてください',
        userId: 1
      })
    });

    const data = await response.json();
    console.log('作成成功:', data);
    alert('投稿ID: ' + data.id);

  } catch (error) {
    console.error('エラー:', error);
  }
}
```

6. 実践例：複数の処理を順番に（async/awaitの強み）

```
// ========================================
// .thenだとネストが深くなる
// ========================================
function createAndShow() {
  fetch('/posts', { method: 'POST', ... })
    .then(response => response.json())
    .then(newPost => {
      // 作成後に一覧を取得
      return fetch('/posts');
    })
    .then(response => response.json())
    .then(posts => {
      // 画面更新
      displayPosts(posts);
    })
    .catch(error => {
      console.error(error);
    });
}
```

```
// ========================================
// async/awaitだとスッキリ
// ========================================
async function createAndShow() {
  try {
    // 1. 投稿作成
    const createResponse = await fetch('/posts', { method: 'POST', ... });
    const newPost = await createResponse.json();

    // 2. 一覧取得
    const listResponse = await fetch('/posts');
    const posts = await listResponse.json();

    // 3. 画面更新
    displayPosts(posts);

  } catch (error) {
    console.error(error);
  }
}
```

7. 実演デモ（ブラウザで実行）

```
<!DOCTYPE html>
<html>
<body>
  <h2>AJAX実演</h2>

  <button onclick="getPostThen()">パターン1: .then</button>
  <button onclick="getPostAwait()">パターン2: await</button>

  <div id="result"></div>

  <script>
    function getPostThen() {
      fetch('https://jsonplaceholder.typicode.com/posts/1')
        .then(response => response.json())
        .then(data => {
          document.getElementById('result').textContent =
            '[.then] ' + data.title;
        })
        .catch(error => {
          console.error('[.then] エラー:', error);
        });
    }

    async function getPostAwait() {
      try {
        const response = await
fetch('https://jsonplaceholder.typicode.com/posts/1');
```

```
        const data = await response.json();
        document.getElementById('result').textContent =
          '[await] ' + data.title;
      } catch (error) {
        console.error('[await] エラー:', error);
      }
    }
  </script>
</body>
</html>
```

見せるポイント：

- 開発者ツール > Network タブで通信確認
- Console タブでログ確認
- 両方とも同じ結果になることを確認

学生への推奨：

- 今回の演習ではasync/awaitを使いましょう
- 読みやすい、エラー処理がわかりやすい、順次処理が直感的
- でも.thenも理解しておくと、既存コード読むときに役立ちます

DOM操作復習

```
// 取得したデータを画面に表示
async function displayPosts() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts');
    const posts = await response.json();

    const list = document.getElementById('post-list');
    list.innerHTML = ''; // 既存の内容をクリア

    posts.slice(0, 5).forEach(post => {
      const div = document.createElement('div');
      div.className = 'post-item';
      div.innerHTML = `
        <h3>${post.title}</h3>
        <p>${post.body}</p>
      `;
      list.appendChild(div);
    });
  } catch (error) {
    console.error('エラー:', error);
  }
}
```

ポイント：

- `document.getElementById()` で要素取得
- `document.createElement()` で要素作成
- `innerHTML` で内容設定
- `appendChild()` で追加

REST API設計の基本

HTTPメソッドの意味

- `GET`: データ取得（読み取り専用）
- `POST`: データ作成
- `PUT`: データ更新（全体）
- `PATCH`: データ更新（一部）
- `DELETE`: データ削除

URLパス設計の原則

```
良い例：
GET    /posts           # 一覧取得
POST   /posts           # 新規作成
GET    /posts/123       # 特定投稿取得
DELETE /posts/123       # 特定投稿削除

悪い例：
GET    /getPostList
POST   /createNewPost
GET    /getPostById?id=123
DELETE /deletePost?id=123
```

リクエスト/レスポンスの構造

```
// リクエスト例
{
  "title": "タイトル",
  "content": "本文",
  "author": "投稿者"
}

// レスポンス例（成功時）
{
  "status": "success",
  "data": {
    "postId": "abc123",
    "title": "タイトル",
    "createdAt": 1234567890
  }
}

// レスポンス例（エラー時）
```

```
{
  "status": "error",
  "message": "タイトルが空です"
}
```

後半：掲示板システム設計演習

システム全体像の説明

システム概要

- カテゴリ別に投稿できる掲示板
- 画像添付可能
- 複数の静的HTMLページから同じAPIを利用

全体構成図

```
ブラウザ（S3静的ホスティング）
    ↓ AJAX通信
API Gateway
    ↓ トリガー
Lambda関数
    ↓ データ操作
DynamoDB

    ↓ 画像保存
S3バケット（画像用）
```

データの流れ

1. ユーザーがブラウザで操作
2. JavaScriptがfetchでAPI呼び出し
3. API Gatewayが受け取り、Lambda起動
4. LambdaがDynamoDBを操作
5. 結果をJSONで返却
6. JavaScriptがDOM操作で画面更新

各画面の役割

- `index.html`: トップ（カテゴリ選択）
- `list.html`: 投稿一覧
- `create.html`: 新規投稿
- `detail.html`: 投稿詳細

データ設計演習（ワークシート配布）

配布物：データ設計ワークシート

```
# データ設計ワークシート

## DynamoDBテーブル設計

### テーブル名
posts

### パーティションキー（必須）
- 属性名: _____
- データ型: _____
- 用途: _____
- 生成方法: _____

### 属性設計

#### 必須属性
1. title
   - データ型: _____
   - 用途: _____
   - バリデーション: _____

2. content
   - データ型: _____
   - 用途: _____
   - バリデーション: _____

3. author
   - データ型: _____
   - 用途: _____
   - バリデーション: _____

4. timestamp
   - データ型: _____
   - 用途: _____
   - 設定タイミング: _____

#### オプション属性
5. category
   - データ型: _____
   - 用途: _____
   - 選択肢例: _____

6. imageKey
   - データ型: _____
   - 用途: _____
   - 形式: _____

### 検討事項
Q1. postIdはどうやって生成しますか？
回答: _____

Q2. timestampはどのタイミングで設定しますか？
```

回答: _____

Q3. categoryの選択肢はどうしますか？（自由入力？固定？）
回答: _____

Q4. 画像がない投稿の場合、imageKeyはどうしますか？
回答: _____

Q5. 削除機能を実装する場合、物理削除？論理削除？
回答: _____

模範解答例

```
### パーティションキー
- 属性名: postId
- データ型: String
- 用途: 投稿を一意に識別
- 生成方法: UUID（Lambda内でuuid.uuid4()）

### 属性設計
1. title
   - データ型: String
   - 用途: 投稿タイトル
   - バリデーション: 1文字以上100文字以内

2. content
   - データ型: String
   - 用途: 投稿本文
   - バリデーション: 1文字以上1000文字以内

3. author
   - データ型: String
   - 用途: 投稿者名
   - バリデーション: 1文字以上50文字以内

4. timestamp
   - データ型: Number
   - 用途: 投稿日時（UnixTime）
   - 設定タイミング: Lambda内でtime.time()

5. category
   - データ型: String
   - 用途: 投稿カテゴリ
   - 選択肢例: 技術/質問/雑談/お知らせ

6. imageKey
   - データ型: String
   - 用途: S3の画像パス
   - 形式: images/{postId}.jpg

### 検討事項
Q1. Python: uuid.uuid4()、JavaScript: crypto.randomUUID()
```

```
Q2. Lambda関数内で投稿時に自動生成
Q3. 固定選択肢（プルダウン）で統一性を保つ
Q4. 空文字またはnull、フロントで画像表示を分岐
Q5. 物理削除（学習目的のため）
```

API設計演習（ワークシート配布）

配布物：API設計ワークシート

```
# API設計ワークシート

## 基本情報
API GatewayのベースURL: https://xxxxxxxx.execute-api.us-east-1.amazonaws.com/prod

---

## 機能1: 投稿一覧取得

### 設計
- HTTPメソッド: _____
- エンドポイント: _____
- クエリパラメータ: _____（オプション）

### リクエスト例
```

```
### レスポンス例（成功時）
```json
{

}
```

レスポンス例（エラー時）

```
{

}
```

使用するLambda関数名

機能2: 特定投稿取得
```

設計

- HTTPメソッド: _____
- エンドポイント: _____
- パスパラメータ: _____

リクエスト例

```
_____
```

レスポンス例

```
{

}
```

機能3: 新規投稿作成

設計

- HTTPメソッド: _____
- エンドポイント: _____

リクエストBody

```
{

}
```

レスポンス例（成功時）

```
{

}
```

バリデーション

- _____
- _____
- _____

機能4: 投稿削除

設計

- HTTPメソッド: _____
- エンドポイント: _____

リクエスト例

```
_____
```

レスポンス例

```
{

}
```

機能5: 画像アップロード用署名付きURL取得

設計

- HTTPメソッド: _____
- エンドポイント: _____

リクエストBody

```
{

}
```

レスポンス例

```
{

}
```

利用フロー

1.

2. _____
3. _____

CORS設定

必要なヘッダー:

- _____
- _____
- _____

エラーコード設計

HTTPステータス	意味	使用場面
200	_____	_____
201	_____	_____
400	_____	_____
404	_____	_____
500	_____	_____

````
**模範解答例**
```markdown
## 機能1: 投稿一覧取得
- HTTPメソッド: GET
- エンドポイント: /posts
- クエリパラメータ: category（オプション）

リクエスト例:
````

GET /posts?category=技術

````
レスポンス例（成功時）:
```json
{
  "status": "success",
  "posts": [
    {
      "postId": "abc123",
````

```
      "title": "AWSの質問",
      "content": "Lambdaについて...",
      "author": "佐々木",
      "category": "技術",
      "timestamp": 1734518400,
      "imageKey": "images/abc123.jpg"
    }
  ]
}
```

使用するLambda関数名: posts-list

機能2: 特定投稿取得

- HTTPメソッド: GET
- エンドポイント: /posts/{postId}
- パスパラメータ: postId

リクエスト例:

```
GET /posts/abc123
```

レスポンス例:

```
{
  "status": "success",
  "post": {
    "postId": "abc123",
    "title": "AWSの質問",
    "content": "Lambdaについて...",
    "author": "佐々木",
    "category": "技術",
    "timestamp": 1734518400,
    "imageKey": "images/abc123.jpg"
  }
}
```

機能3: 新規投稿作成

- HTTPメソッド: POST
- エンドポイント: /posts

リクエストBody:

```

```

```
{
  "title": "AWSの質問",
  "content": "Lambdaについて教えてください",
  "author": "佐々木",
  "category": "技術",
  "imageKey": "images/xyz789.jpg"
}
```

レスポンス例（成功時）:

```
{
  "status": "success",
  "message": "投稿を作成しました",
  "postId": "abc123"
}
```

バリデーション:

- titleは1文字以上100文字以内
- contentは1文字以上1000文字以内
- authorは必須
- categoryは技術/質問/雑談/お知らせのいずれか

機能4: 投稿削除

- HTTPメソッド: DELETE
- エンドポイント: /posts/{postId}

リクエスト例:

```
DELETE /posts/abc123
```

レスポンス例:

```
{
  "status": "success",
  "message": "投稿を削除しました"
}
```

機能5: 画像アップロード用署名付きURL取得

- HTTPメソッド: POST

- エンドポイント: /upload-url

リクエストBody:

```
{
  "fileName": "photo.jpg",
  "fileType": "image/jpeg"
}
```

レスポンス例:

```
{
  "status": "success",
  "uploadUrl": "https://bucket-name.s3.amazonaws.com/...",
  "imageKey": "images/xyz789.jpg"
}
```

利用フロー:

1. フロントエンドがこのAPIを呼び出してuploadUrlを取得
2. そのURLに対して画像をPUT（fetchまたはXHR）
3. 投稿作成時にimageKeyを含めてPOST /posts

CORS設定

必要なヘッダー:

- Access-Control-Allow-Origin: *
- Access-Control-Allow-Methods: GET, POST, DELETE, OPTIONS
- Access-Control-Allow-Headers: Content-Type

エラーコード設計

HTTPステータス	意味	使用場面
200	成功	GET, DELETE成功時
201	作成成功	POST成功時
400	リクエスト不正	バリデーションエラー
404	見つからない	存在しないpostIdを指定
500	サーバーエラー	Lambda実行エラー

````
---

### 画面・機能設計演習（ワークシート配布）

**配布物：画面設計ワークシート**

```markdown
# 画面設計ワークシート

## 画面1: トップページ（index.html）

### 画面の役割
_____

### 表示する要素
- _____
- _____
- _____

### ユーザー操作
- _____
- _____

### 使用するAPI
なし（静的ページ）

### JavaScript処理
- _____

---

## 画面2: 投稿一覧（list.html）

### 画面の役割
_____

### 表示する要素
1. _____
2. _____
3. _____

### 使用するAPI
- エンドポイント: _____
- HTTPメソッド: _____
- パラメータ: _____

### JavaScript処理フロー
```javascript
// ページ読み込み時の処理
async function loadPosts() {
  // 1. _____
  // 2. _____
````

```
  // 3. _____________
}

// DOM操作
function displayPosts(posts) {
  // 1. _____________
  // 2. _____________
  // 3. _____________
}
```

ポーリング処理

```
// _____________秒ごとに新着チェック
setInterval(() => {
  _____________
}, _____________);
```

カテゴリ絞り込み

- 実装方法: _____________
- UIコンポーネント: _____________

画面3: 新規投稿（create.html）

画面の役割

入力フォーム

1. _____________（必須/任意）
2. _____________（必須/任意）
3. _____________（必須/任意）
4. _____________（必須/任意）
5. _____________（必須/任意）

使用するAPI

投稿作成

- エンドポイント: _____________
- HTTPメソッド: _____________

画像アップロード（オプション）

- エンドポイント1: _____________（署名付きURL取得）
- エンドポイント2: _____________（S3へ直接アップロード）

JavaScript処理フロー

```
// 投稿ボタンクリック時
async function submitPost() {
  // 1. バリデーション
  //    _____________

  // 2. 画像がある場合
  //    _____________

  // 3. 投稿データ送信
  //    _____________

  // 4. 成功時の処理
  //    _____________
}
```

バリデーション

- _____
- _____
- _____

エラーハンドリング

- _____
- _____

画面4: 投稿詳細（detail.html）

画面の役割

URLパラメータ

- パラメータ名: _____
- 取得方法: _____

表示する要素

1. _____
2. _____
3.

4. _____（画像がある場合）
5. _____（削除ボタン）

使用するAPI

投稿取得

- エンドポイント: _____
- HTTPメソッド: _____

投稿削除

- エンドポイント: _____
- HTTPメソッド: _____

JavaScript処理フロー

```
// ページ読み込み時
async function loadPostDetail() {
  // 1. URLからpostId取得
  //    _____

  // 2. API呼び出し
  //    _____

  // 3. 画面表示
  //    _____
}

// 削除ボタンクリック時
async function deletePost() {
  // 1. 確認ダイアログ
  //    _____

  // 2. API呼び出し
  //    _____

  // 3. 一覧画面へ遷移
  //    _____
}
```

共通処理（api.js）

共通関数設計

API呼び出し共通処理

```
// ベースURL定義
const API_BASE_URL = '_____________';

// GET共通処理
async function apiGet(endpoint) {
  // _____________
}

// POST共通処理
async function apiPost(endpoint, data) {
  // _____________
}

// DELETE共通処理
async function apiDelete(endpoint) {
  // _____________
}
```

エラーハンドリング

```
function handleError(error) {
  // _____________
}
```

ローディング表示

```
function showLoading() {
  // _____________
}

function hideLoading() {
  // _____________
}
```

S3静的ホスティング構成

ディレクトリ構造

```
bucket-name/
├── index.html
├── list.html
├── create.html
├── detail.html
├── css/
```

```
│   └── style.css
└── js/
    └── api.js
```

各ファイルの役割

- index.html: _____
- list.html: _____
- create.html: _____
- detail.html: _____
- api.js: _____

模範解答例
```markdown
## 画面2: 投稿一覧（list.html）

### 画面の役割
カテゴリ別に投稿一覧を表示し、新着投稿を自動で取得する

### 表示する要素
1. カテゴリ選択ボタン（全て/技術/質問/雑談/お知らせ）
2. 投稿リスト（タイトル、投稿者、日時、カテゴリ）
3. 新規投稿ボタン

### 使用するAPI
- エンドポイント: GET /posts
- HTTPメソッド: GET
- パラメータ: ?category=技術（オプション）

### JavaScript処理フロー
```javascript
// ページ読み込み時の処理
async function loadPosts(category = '') {
  // 1. ローディング表示
  showLoading();

  // 2. API呼び出し
  const url = category ? `/posts?category=${category}` : '/posts';
  const response = await apiGet(url);

  // 3. 画面表示
  displayPosts(response.posts);
  hideLoading();
}

// DOM操作
function displayPosts(posts) {
  // 1. 既存のリストをクリア
  const list = document.getElementById('post-list');
  list.innerHTML = '';
```
```

```
  // 2. 投稿ごとにdiv作成
  posts.forEach(post => {
    const div = document.createElement('div');
    div.className = 'post-item';
    div.innerHTML = `
      <h3><a href="detail.html?id=${post.postId}">${post.title}</a></h3>
      <p>投稿者: ${post.author} | カテゴリ: ${post.category}</p>
      <p>日時: ${new Date(post.timestamp * 1000).toLocaleString()}</p>
    `;
    list.appendChild(div);
  });

  // 3. 投稿がない場合のメッセージ
  if (posts.length === 0) {
    list.innerHTML = '<p>投稿がありません</p>';
  }
}
```

ポーリング処理

```
// 3秒ごとに新着チェック
setInterval(() => {
  loadPosts(currentCategory);
}, 3000);
```

カテゴリ絞り込み

- 実装方法: ボタンクリックで再度API呼び出し
- UIコンポーネント: ボタングループ

画面3: 新規投稿（create.html）

画面の役割

新しい投稿を作成し、画像をアップロードできる

入力フォーム

1. タイトル（必須、テキストボックス）
2. 本文（必須、テキストエリア）
3. 投稿者名（必須、テキストボックス）
4. カテゴリ（必須、セレクトボックス）
5. 画像（任意、ファイル選択）

使用するAPI

投稿作成

- エンドポイント: POST /posts
- HTTPメソッド: POST

画像アップロード

- エンドポイント1: POST /upload-url（署名付きURL取得）
- エンドポイント2: PUT {署名付きURL}（S3へ直接アップロード）

JavaScript処理フロー

```
async function submitPost() {
  // 1. バリデーション
  const title = document.getElementById('title').value;
  if (!title) {
    alert('タイトルを入力してください');
    return;
  }

  // 2. 画像がある場合
  let imageKey = null;
  const fileInput = document.getElementById('image');
  if (fileInput.files.length > 0) {
    imageKey = await uploadImage(fileInput.files[0]);
  }

  // 3. 投稿データ送信
  const postData = {
    title: title,
    content: document.getElementById('content').value,
    author: document.getElementById('author').value,
    category: document.getElementById('category').value,
    imageKey: imageKey
  };

  const response = await apiPost('/posts', postData);

  // 4. 成功時の処理
  alert('投稿しました');
  location.href = 'list.html';
}

async function uploadImage(file) {
  // 1. 署名付きURL取得
  const urlResponse = await apiPost('/upload-url', {
    fileName: file.name,
    fileType: file.type
  });

  // 2. S3へアップロード
  await fetch(urlResponse.uploadUrl, {
    method: 'PUT',
    body: file,
```

```
    headers: {
      'Content-Type': file.type
    }
  });

  // 3. imageKeyを返す
  return urlResponse.imageKey;
}
```

バリデーション

- タイトル：1文字以上100文字以内
- 本文：1文字以上1000文字以内
- 投稿者名：必須
- カテゴリ：選択必須

エラーハンドリング

- 入力エラー：alertで表示
- 通信エラー：再試行を促す

```
---

### 発表・まとめ

**発表（グループまたは個人）**
- 各自の設計内容を共有（5分×数名）
- 設計の意図や工夫した点を説明
- 質疑応答

**良い点・改善点のフィードバック**
- 講師からのコメント
- 他の学生からの質問や提案

**全体まとめ**
- 設計の重要性の再確認
- API設計の原則
- AJAX通信のポイント

**次回の実装に向けた宿題**
1. 設計の清書（今日のワークシートを整理）
2. Lambda関数の処理フローを疑似コードで記述
3. 実装したい追加機能を考える（例：いいね機能、検索機能など）

---

## 配布資料一覧

### 1. 全体構成図（1枚）
システム全体のアーキテクチャ図
```

```
- ブラウザ、S3、API Gateway、Lambda、DynamoDBの関係
- データの流れ

### 2. AJAX参考資料（1-2枚）
- fetch APIサンプルコード集
- .thenパターンとasync/awaitパターンの比較
- エラーハンドリング例

### 3. データ設計ワークシート（2-3枚）
穴埋め形式でDynamoDBテーブル設計を記入

### 4. API設計ワークシート（3-4枚）
各エンドポイントの設計を記入

### 5. 画面設計ワークシート（4-5枚）
各HTMLファイルの役割と処理フローを記入

### 6. 次回までの宿題シート（1枚）
実装に向けた準備事項

---

## 評価ポイント

### データ設計（30点）
- テーブル構造の妥当性
- 属性の選択と型の適切性
- パーティションキーの設計理由

### API設計（30点）
- RESTful原則への準拠
- エンドポイント設計の一貫性
- エラーハンドリングの考慮

### 画面設計（30点）
- UI/UXの考慮
- AJAX通信の適切な実装計画
- ユーザー操作の流れの明確さ

### 全体整合性（10点）
- データ・API・画面の整合性
- 実装可能性
- 説明の明確さ

---

## 補足：次回以降の展開

### 次回（実装1）
- DynamoDBテーブル作成
- Lambda関数実装（posts-list）
- API Gateway設定
- list.htmlでの一覧表示
```

```
### 3回目（実装2）
- 残りのLambda関数実装
- create.html、detail.html実装
- CRUD操作の完成

### 4回目（機能拡張）
- 画像アップロード機能
- ポーリングによる自動更新
- エラーハンドリング強化
- UI改善

### 5回目（発展課題）
- 検索機能
- いいね機能
- コメント機能
- ページネーション
```