

# 仕様書

24G1007 網中洲

2025 年 01 月 7 日

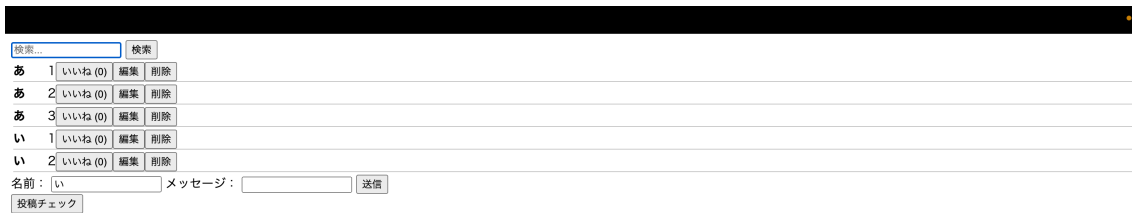
## 1 概要

このアプリケーションは、シンプルな掲示板（BBS）システムを実装している。ユーザーは以下の操作を行うことができる。

1. 投稿（名前、メッセージの送信）。
2. 投稿一覧の取得（リアルタイム更新）。
3. 投稿の検索（キーワード検索）。
4. 投稿の編集および削除。
5. 投稿への「いいね」。

## 2 利用者向けの説明

画面のレイアウトは、以下の図 1 のようになる。「送信」ボタンを押すと、名前とメッセージが送信され、「投稿チェック」ボタンを押すと、投稿された名前とメッセージを閲覧できる。さらに、投稿されたメッセージごとに「いいね」ボタン、「編集」ボタン、「削除」ボタンがある。これらは、投稿されたメッセージの評価、編集、削除ができる。また、上部に「検索」ボタンがある。これは、検索したいメッセージの一部を入力することで、見たいメッセージを検索できる。



検索...		検索
あ	1	いいね (0) 編集 削除
あ	2	いいね (0) 編集 削除
あ	3	いいね (0) 編集 削除
い	1	いいね (0) 編集 削除
い	2	いいね (0) 編集 削除

名前: い      メッセージ:      送信

投稿チェック

図 1 画面のレイアウト

### 3 管理者向けの説明

このサーバーを立ち上げる手順以下になる。

1. ターミナルを開き、‘node app8.js’ と入力（ホスト名は ‘localhost’，ポート番号は ‘8080’）。
2. 別のウィンドウでターミナルを開き，‘telnet localhost 8080’ と入力。
3. Web ブラウザの URL 欄に ‘http://localhost:8080/public/bbs.html’ と入力し，ページを表示する。

### 4 開発者向けの説明

#### 4.1 サーバー側プログラムの構成と説明 (app8.js)

役割として，サーバーは，クライアントからのリクエストに応じて投稿データを操作し，レスポンスを返す。主なエンドポイントは，投稿作成，一覧取得，いいね，検索，編集，削除などを提供する。投稿データは bbs 配列に保存される（一時的なメモリ管理）。また，図 2 は，本システムのシーケンス図である。

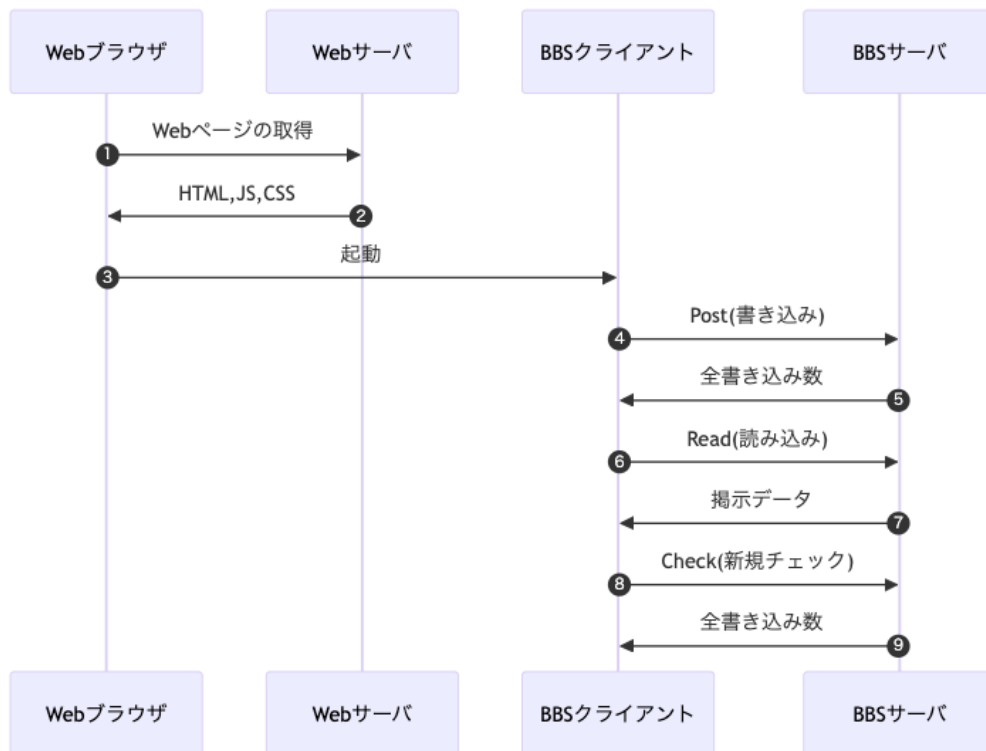


図2 本システムのシーケンス図

#### 4.1.1 初期設定

```

1 const express = require("express");
2 const app = express();
3 app.use(express.urlencoded({ extended: true }));
4 app.use("/public", express.static(__dirname + "/public"));
  
```

##### プログラムの追加説明

- express はルーティングとリクエストの処理を簡略化する.
- express.urlencoded はフォームデータを解析するミドルウェア.

#### 4.1.2 投稿データの管理

```

1 let bbs = [];
  
```

- 投稿データをメモリ内で管理.
- 配列を簡易的なデータストアとして使用.

#### 4.1.3 投稿作成エンドポイント

```
1 app.post("/post", (req, res) => {
2   const name = req.body.name;
3   const message = req.body.message;
4   console.log([name, message]);
5   bbs.push({ name: name, message: message, likes: 0 });
6   res.json({ number: bbs.length });
7 });
```

##### —— 処理内容 ——

- 1 req.body から name と message を取得.
- 2 投稿データに初期値 likes: 0 を追加.
- 3 bbs 配列に保存.
- 4 新しい投稿数をレスポンスとして返却.

#### 4.1.4 投稿一覧取得エンドポイント

```
1 app.post("/read", (req, res) => {
2   const start = Number(req.body.start);
3   const messages = start === 0 ? bbs : bbs.slice(start);
4   res.json({ messages });
5 });
```

##### —— 処理内容 ——

- 1 start インデックスから投稿を取得.
- 2 投稿全体または一部をレスポンスとして返却.

#### 4.1.5 いいねエンドポイント

```
1 app.post("/like", (req, res) => {
2   const id = Number(req.body.id);
3   if (id >= 0 && id < bbs.length) {
4     bbs[id].likes = (bbs[id].likes || 0) + 1;
5     res.json({ likes: bbs[id].likes });
6   } else {
7     res.status(400).json({ error: "Invalid ID" });
8   }
9 });
```

— 処理内容 —

- 1 id に対応する投稿を検索.
- 2 該当する投稿の likes をインクリメント.
- 3 新しい「いいね」数を返却.

#### 4.1.6 検索エンドポイント

```
1 app.post("/search", (req, res) => {
2   const keyword = req.body.keyword.toLowerCase();
3   const results = bbs.filter(
4     (post) =>
5       post.name.toLowerCase().includes(keyword) ||
6       post.message.toLowerCase().includes(keyword)
7   );
8   res.json({ messages: results });
9 });
```

— 処理内容 —

- 1 キーワードで投稿をフィルタリング.
- 2 投稿者名またはメッセージにキーワードが含まれる投稿をレスポンスとして返却.

#### 4.1.7 編集エンドポイント

```
1 app.put("/edit/:id", (req, res) => {
2   const id = Number(req.params.id); // URL パラメータ ":id" を取得し数値に変換
3   const newMessage = req.body.message; // リクエストボディから "message" を取得
4   if (id >= 0 && id < bbs.length) { // ID が有効な範囲内か確認
5     bbs[id].message = newMessage; // 投稿内容を新しいメッセージに更新
6     res.json({
7       success: true,
8       updatedMessage: bbs[id] // 更新後の投稿データを返却
9     });
10  } else {
11    res.status(400).json({ error: "Invalid ID" }); // 無効なID の場合エラーを返却
12  }
13 });
```

— 処理内容 —

- 1 クライアントがリクエストした投稿の id (投稿の配列インデックス) を URL パラメータから取得.
- 2 クライアントがリクエストボディで送信した message を取得.
- 3 id が bbs 配列の範囲内 (0 以上、bbs.length - 1 以下) かを確認. 条件を満たさない場合、ステータスコード 400 とエラーメッセージを返却.
- 4 該当する投稿 (bbs[id]) の message を新しいメッセージで上書き
- 5 更新が成功した場合, success: true と更新後の投稿データ (bbs[id]) を JSON 形式で返却
- 6 無効な id (例: 配列範囲外のインデックス) に対してエラーを返却

#### 4.1.8 削除エンドポイント

```
1 app.delete("/delete/:id", (req, res) => {
2   const id = Number(req.params.id); // URL パラメータ ":id" を取得し数値に変換
3   if (id >= 0 && id < bbs.length) { // ID が有効な範囲内か確認
4     bbs.splice(id, 1); // 指定された投稿を削除
5     res.json({ success: true }); // 成功レスポンスを返却
6   } else {
7     res.status(400).json({ error: "Invalid ID" }); // 無効なID の場合エラーを返却
8   }
9 });
```

— 処理内容 —

- 1 クライアントがリクエストした投稿の id (投稿の配列インデックス) を URL パラメータから取得.
- 2 id が bbs 配列の範囲内 (0 以上、bbs.length - 1 以下) かを確認. 条件を満たさない場合、ステータスコード 400 とエラーメッセージを返却.
- 3 bbs.splice(id, 1) を使用して、該当する投稿を削除.splice は指定したインデックスの要素を削除し、配列を更新.
- 4 削除が成功した場合、success: true を JSON 形式で返却.
- 5 無効な id (例: 配列範囲外のインデックス) に対してエラーを返却

## 4.2 クライアント側プログラムの構成と説明 (bbs.js)

### 4.2.1 初期設定

```
1 let number = 0;
2 const bbs = document.querySelector('#bbs');
```

— プログラムの追加説明 —

- number はサーバーから取得した投稿数を追跡.
- bbs は投稿を表示する HTML 要素.

#### 4.2.2 投稿送信処理

```
1 document.querySelector('#post').addEventListener('click', () => {
2   const name = document.querySelector('#name').value;
3   const message = document.querySelector('#message').value;
4   const params = new URLSearchParams({ name, message });
5   fetch("/post", {
6     method: "POST",
7     body: params,
8     headers: { 'Content-Type': 'application/x-www-form-urlencoded' }
9   })
10  .then((response) => response.json())
11  .then(() => {
12    document.querySelector('#message').value = "";
13  });
14 });
```

##### 処理内容

- 1 フォーム入力値を取得.
- 2 fetch を使用してサーバーに非同期リクエストを送信.
- 3 レスポンスを受け取った後、入力フィールドをクリア.

#### 4.2.3 投稿一覧の取得

```
1 document.querySelector('#check').addEventListener('click', () => {
2   fetch("/check", { method: "POST" })
3   .then((response) => response.json())
4   .then((response) => {
5     if (number !== response.number) {
6       fetch("/read", {
7         method: "POST",
8         body: 'start=${number}',
9         headers: { 'Content-Type': 'application/x-www-form-urlencoded' }
10      })
11      .then((response) => response.json())
12      .then((response) => {
13        number += response.messages.length;
14        response.messages.forEach((mes) => {
15          const cover = document.createElement('div');
16          cover.className = 'cover';
17          cover.innerHTML = `
18            <span class="name">${mes.name}</span>
19            <span class="mes">${mes.message}</span>
20          `;
```

```
21         bbs.appendChild(cover);
22     });
23 });
24 }
25 });
26 });
```

---

— 処理内容 —

- 1 サーバーに現在の投稿数を問い合わせ.
- 2 新しい投稿があれば、それを取得して画面に追加表示.

#### 4.2.4 いいね操作

---

```
1 let like_button = document.createElement('button');
2 like_button.innerText = いいね' (${mes.likes || 0})';
3 like_button.addEventListener('click', () => {
4     fetch('/like', {
5         method: "POST",
6         body: 'id=${mes.id}',
7         headers: { 'Content-Type': 'application/x-www-form-urlencoded' }
8     })
9     .then((response) => response.json())
10    .then((data) => {
11        like_button.innerText = いいね' (${data.likes})';
12    });
13 });
```

---

— 処理内容 —

- 1 ボタンを動的に生成し, 現在の「いいね」数を表示するテキストを設定.
- 2 ボタンがクリックされたとき, サーバーに対象投稿の「いいね」操作をリクエスト.
- 3 /like エンドポイントに POST リクエストを送信. リクエストには投稿 ID を含むデータを送る.
- 4 サーバーから新しい「いいね」数を受け取り, ボタンの表示を更新