Couchbase NoSQL Developer Workshop

ラボハンドブック

ラボ **3:** その他のキーバリュー操作

手順

このラボの目的は、注文の保存、更新、および取得を可能にするロジックを作成することです。JavaScript SDK を利用して、K/V の取得、挿入/アップサート、および置換操作を実行します。 K/V 操作の使用の詳細については、SDK ドキュメントを参照してください。

ステップ 1: API にロジックを追加する

ドキュメント

- ドキュメントの取得
- ドキュメントの作成
- ドキュメントの更新

API の戻り値に関する注意 ***

K/V get 操作の結果オブジェクトには、要求されたドキュメントの他、メタデータおよび API コールのパフォーマンスメトリックが含まれます。 *result.value* は、通常結果として期待されるドキュメントです。

API を設計する際には、設計を最適化するための数多くの選択肢があり、「正しい」選択肢はありませんが、時には「間違った」選択肢は、あります。 また、統一性が最優先であると判断する場合があります。

- getOrder ()
- saveOrder ()
- updateOrder ()
- deleteOrder ()

これらは同じ値を返します。ここで、疑問があります。戻り値は何であるべきでしょうか?

- ドキュメント?

  Node.js のような非同期・コールバック指向の言語では、渡されるコールバックファンクションに必要なすべての情報が含まれるように、ドキュメント全体を返す必要があります。 しかし、これらのコールバック関数がドキュメントにあまりアクセスしない場合、クライアント が使用しない情報に対して大量のネットワーク トラフィック (および場合によってはクラウドからの データ出力コスト) が発生する可能性があります。

- ID?

  これはドキュメント全体よりもはるかに小さいため、効率的でネットワークの競合が少なく、データの出力コストが小さくなる可能性があります。また、ダウンストリームでコールバックに id を渡し 、必要に応じてドキュメント全体を取得するのは、コールバックの迅速かつ簡単な操作です。

- ブール値?

  データパケットのサイズを最適化する場合、ブール値は最小だと思いますか? いいえ。何も返さなければ、さらに小さくなります。 また、適切に設計されたエラー処理フレームワーク(try/catch)がある場合は、チェックする必要があるブール値を返すと、醜い不要なコードが発生します。

では、 このラボでは、何を最適化したのでしょうか。それは、教育的であることです。 提供するサンプル コードでは、それぞれのケース毎に異なる方法で、値を返す方法を示しています。選択はあなたです。 このサンプルは API の整合性のために最適化されていません - :)

注文の取得

API repository.js のファイルを開きます(付録の API のプロジェクト構造を参照)。 *getOrder( )*メソッドを検索します。 注文ドキュメントを更新 (または置換) するために必要なロジックを追加して、*getOrder*() メソッドを編集します。

getOrder() input:

- orderId: 文字列 - 注文ドキュメントのドキュメントキー
- コールバック

getOrder() 出力:

- エラー オブジェクト (該当する場合)
- 注文ドキュメント:サンプル注文ドキュメントの付録を参照してください。

*getOrder()*メソッドの実装については、以下のコード スニペットを参照してください。 これは、または同様のソリューションを使用して、*getOrder()*メソッド ロジックを実装できます。

注意: NOP コード行を、コメントアウトするか、削除します。

```
0:  async getOrder(orderId) {
1:    try {
2:      /**
3:       * Lab 3:  K/V operation(s):
4:       *  1.  get order:  collection.get(key)
5:       *
6:       */
7:      let result = await this.collection.get(orderId);
8:      return { order: result.value ? result.value : null, error: null };
9:    } catch (err) {
10:     //Optional - add business logic to handle error types
11:     outputMessage(err, "repository.js:getOrder() - error:");
12:     return { order: null, error: err };
13:   }
14: }
```

コードに関する注意事項:

- 非同期/待機構文を使用します。
- 7 行目: K/V 操作は、3.x SDK の収集レベルで行われます。
    - K/V は操作パラメータを取得します。
        - ドキュメント キー
- 8 行目: エラーがない場合は結果の値のみを返します。 ラボでは、ドキュメントのコンテンツを返すだけで済みます。
- *outputMessage()*:コンソールに情報を簡単に出力するためのヘルパーメソッドは、/library ディレクトリにあります (付録で詳しく説明されている API のプロジェクト構造を参照してください)。

- - try/catch & err オブジェクトの処理は、一般的な方法でポーズ的に行われます。 ラボ参加者は、エラーを処理するさまざまな方法をテストするために、それに応じてロジックを自由に追加できます。

完了したら、*repository.js* ファイルが保存されていることを確認します。 API *Docker* コンテナは API の作業ディレクトリにマップされるため、API コードに対して行われたすべての更新はコンテナに反映される必要があります。 コードを保存すると、特定の注文を取得する機能をテストできます。 以下の手順に従って、*スワッガー UI* ページを使用して *getOrders()*ロジックを確認します。

注 ***

getOrder() を使用するには承認が必要です。

1. *[Swagger UI]*ページに移動します: http://localhost:3000/api-docs/
2. */user/getOrder* エンドポイントをクリックします。
3. パネルが展開されたら、[*Try it out*]ボタンをクリックします。
4. 指定されたテキスト入力フィールドに orderId を入力します。
    a. 例: order_1000
5. [*Execute*] ボタンをクリックします。
6. 応答 *コード*は 200 である必要があり、*応答*本文には、手順 #4 で入力した注文キーのドキュメントの内容を含むデータプロパティが含まれている必要があります。

*応答*本文のサンプル

```
{
  "data": {
   ... sample order document JSON ...
  },
  "message": "Successfully retrieved order.",
  "error": null,
  "authorized": true,
  "requestId": null
}
```

注文の保存

API リポジトリ ディレクトリで *repository.js* ファイルを開きます(付録で詳しく説明されている API のプロジェクト構造を参照)。 *saveOrder( )*メソッドを検索します。 saveOrder( )メソッドを編集するには、注文ドキュメントを挿入 (または upsert)するために必要なロジックを追加します。

考える点がいくつかあります。

1. 新しい注文の値を設定する方法 Id ?
    a. ヒント: 付録に用意されているヘルパー メソッドを 参照してください。
2. 挿入/アップサートメソッドは何を 返すのですか?
3. 新しく挿入された注文文書を返す方法は?
    a. ヒント: 最近作成したメソッドを使用します。

saveOrder() 入力:

- 注文ドキュメント:サンプル注文ドキュメントの付録を参照してください。

- コールバック

saveOrder() 出力:

- エラー オブジェクト (場合はエラー オブジェクト)
- 注文ドキュメント:サンプル注文ドキュメントの付録を参照してください。

*saveOrder()*メソッドの実装については、以下のコード スニペットを参照してください。

注意: NOP コード行を、コメントアウトするか、削除します。

```
0:  async saveOrder(order) {
1:    try {
2:      /**
3:       * Lab 3:  K/V operation(s):
4:       *  1.  generate key:  order_<orderId>
5:       *  2.  insert order:  collection.insert(key, document)
6:       *  3.  IF successful insert, GET order
7:       *
8:       */
9:
10:     let orderId = await this.getNextOrderId();
11:     let key = `order_${orderId}`;
12:
13:     order._id = key;
14:     order.orderId = orderId;
15:     order.doc.created = Math.floor(new Date() / 1000);
16:     order.doc.createdBy = order.custId;
17:
18:     let savedDoc = await this.collection.insert(key, order);
19:     if (!savedDoc) {
20:       return { order: null, error: null };
21:     }
22:     let result = await this.collection.get(key);
23:
24:     return { order: result.value ? result.value : null, error: null };
25:   } catch (err) {
26:     //Optional - add business logic to handle error types
27:     outputMessage(err, "repository.js:saveOrder() - error:");
28:     return { order: null, error: err };
29:   }
30: }
31:
32:
33:
34:
35:
```

コードに関する注意事項:

- 非同期/待機構文を使用します。
- 10 行目: ヘルパー メソッドを使用して *getNextOrderId()*を使用して新しい orderId 値をシードします。
- 13-16 行: 提供された注文ドキュメントにドキュメントコンポーネントを追加します。
- 18 行目: K/V 操作は、3.x SDK の収集レベルで行われます。
    - K/V 挿入操作パラメータ:

        - ドキュメント キー
        - ドキュメント
- 22 行目: 挿入/アップサート K/V 操作はドキュメントを回転させないので、前回の K/V 取得操作を利用して、最近作成された順序を返します。 もう 1 つの方法は、挿入/アップサート操作に渡されたドキュメントを返す方法です 。
- *outputMessage()*: コンソールに情報を簡単に出力するために使用されるヘルパー メソッドは、/library ディレクトリにあります (付録で詳しく説明されている API のプロジェクト構造を参照してください) 。
- try/catch & err オブジェクトの処理は、汎用的な方法で意図的に行われます。 ラボ 参加者は、エラーを処理するさまざまな方法をテストするために、それに応じてロジックを自由に追加できます。

完了したら、*repository.js* ファイルが保存されていることを確認します。 API *Docker* コンテナは API の作業ディレクトリにマップされるため、API コードに対して行われた更新はすべてコンテナに反映されます。 コンテナーの状態の詳細については、*docker logs api* コマンドを使用します。 コードを保存すると、保存順序機能が Web UI 内でアクティブになる必要があります。 Web UI を使用して *saveOrder()*ロジックを検証するには、次の手順に従います。

注 ***

saveOrder() を使用するには承認が必要です。

1. http://localhost:8080 に移動します。
2. ログインしていない場合は、ログインします。
3. *ホームページにいない場合(検索ボックス*が表示されていない)
    a. 左上隅にある*[Couchbase|NoEQUAL]*画像をクリックして*ホーム*画面に移動します。
4. *検索ボックス*に製品を入力し、*虫眼鏡*をクリックして検索を実行します。
5. 在庫品 の 左下隅のドロップダウン矢印をクリックすると、製品数を増減し、商品をカートに追加する機能とともに、製品の詳細が表示されます。
6. [+] ボタンをクリックします。
7. [*Add to Cart]をクリックします。*
8. 右上隅のカート アイコンの数字が、追加された製品の数だけ増加します。

注文の更新

API リポジトリ ディレクトリで *repository.js* ファイルを開きます(付録で詳しく説明されている API のプロジェクト構造を参照)。 *replaceOrder( )*メソッドを検索します。 注文ドキュメントを更新 (または置換) するために必要な logic を追加して *replaceOrder*() メソッドを編集します。

置換順序() 入力:

- 注文ドキュメント:サンプル注文ドキュメントの付録を参照してください。
- コールバック

置換 Order() 出力:

- エラー オブジェクト (該当する場合)
- 注文ドキュメントキー

*replaceOrder()*メソッドの実装については、以下のコード スニペットを参照してください。

注意: NOP コード行を、コメントアウトするか、削除します。

```
0:   async replaceOrder(order) {
1:     try {
2:       /**
3:        * Lab 3:  K/V operation(s):
4:        *  1.  generate key:  order_<orderId>
5:        *  2.  replace order:  collection.replace(key, document)
6:        *
7:        */
8:       let key = `order_${order.orderId}`;
9:       order.doc.modified = Math.floor(new Date() / 1000);
10:      order.doc.modifiedBy = order.custId;
11:      let result = await this.collection.replace(key, order);
12:      return { success: result != null, error: null };
13:    } catch (err) {
14:      //Optional - add business logic to handle error types
15:      outputMessage(err, "repository.js:replaceOrder() - error:");
16:      return { success: false, error: err };
17:    }
18:  }
```

コードに関する注意事項:

- 非同期/待機構文を使用します。
- 8-10 行: 注文ドキュメントのキーを設定し、監査プロパティを更新します。
- 11 行目: K/V 操作は、3.x SDK の収集レベルで実行されます。
    - K/V 置換操作パラメータ:
        - ドキュメント キー
        - ドキュメント
- 12 行目: 成功した場合に注文ドキュメントを返します。
- *outputMessage()*: コンソールに情報を簡単に出力するためのヘルパーメソッドは、/library ディレクトリにあります (付録で詳しく説明されている API のプロジェクト構造を参照してください)。
- try/catch & err オブジェクトの処理は、汎用的な方法で意図的に行われます。 ラボ参加者は、エラーを処理するさまざまな方法をテストするために、それに応じてロジックを自由に追加できます。

完了したら、*repository.js* ファイルが保存されていることを確認します。API *Docker* コンテナは API の作業ディレクトリにマップされるため、API コードに対して行われたすべての更新はコンテナに反映される必要があります。 コンテナーの状態の詳細については、*docker logs api* コマンドを使用します。 コードが保存されると、置換注文機能は Web UI 内でアクティブにする必要があります。 以下の手順に従って、*replaceOrder()* ロジックを確認します。

注 ***

次の手順は、注文が既に保存されている場合にのみ機能し、それ以外の場合は新しい注文が作成されます。 前のセクションで新しい order が既に作成されているはずです。 また、 *replaceOrder()* を使用するには承認が必要です。

1. http://localhost:8080 に移動
2. ログインしていない場合は、ログインします。

3. ホーム画面にない場合(検索バーが表示されないなど)
    a. 左上隅にある*[Couchbase|NoEQUAL]*画像をクリックしてホーム画面に移動する。
4. 検索ボックスに製品名(例: shirt)を入力し、虫眼鏡アイコンをクリックして検索を実行します。
5. 在庫製品で、左下隅のドロップダウン矢印をクリックすると、製品の詳細が表示され、製品の数を増減し、商品をカートに追加する機能が表示されます。
6. [+] *ボタンをクリックします。*
7. [*Add to Cart]をクリックします。*
8. 右上隅のカート アイコンの数字が 、追加された製品の数だけ増加します。

注文の削除

API リポジトリ ディレクトリで *repository.js* ファイルを開きます(付録で詳しく説明されている API のプロジェクト構造を参照)。 *deleteOrder ( )*メソッドを検索します。 注文ドキュメントを削除するために必要なロジックを追加して、*deleteOrder*() メソッドを編集します。

deleteOrder () 入力:

- orderId: 文字列 - 注文ドキュメントのドキュメントキー
- コールバック

deleteOrder () 出力:

- エラー オブジェクト (該当する場合)
- 成功: ブール値(削除が成功した場合は true)

*deleteOrder()*メソッドの実装については、以下のコード スニペットを参照してください。

注意: NOP コード行を、コメントアウトするか、削除します。

```
0:  async deleteOrder(orderId) {
1:    try {
2:      /**
3:       * Lab 3:  K/V operation(s):
4:       *  1.  delete order:  cluster.remove(key)
5:       *
6:       */
7:      let result = await this.collection.remove(orderId);
8:      return { success: result != null, error: null };
9:    } catch (err) {
10:     //Optional - add business logic to handle error types
11:     outputMessage(err, "repository.js:deleteOrder() - error:");
12:     return { success: false, error: err };
13:   }
14: }
```

コードに関する注意事項:

- 非同期/待機構文を使用します。
- 7 行目: K/V 操作は、3.x SDK の Collection レベルで行われます。
    - K/V 除去操作パラメータ:
    - ドキュメント キー
- 8 行目: 操作の成功に基づいてブール値を返すか、該当する場合はエラー オブジェクトを返します。

- *outputMessage()*: コンソールに情報を簡単に出力するためのヘルパーメソッドは、/library ディレクトリにあります (付録の API のプロジェクト構造を参照してください)
- try/catch & err オブジェクトの処理は、汎用的な方法で意図的に行われます。 ラボ参加者は、エラーを処理するさまざまな方法をテストするために、それに応じてロジックを自由に追加できます。

完了したら、*repository.js* ファイルが保存されます。 API *Docker* コンテナは API の作業ディレクトリにマップされるため、API コードに対して行われたすべての更新はコンテナに反映される必要があります。コンテナの状態に関する詳細については、*docker logs api* コマンドを使用します。
以下の手順に従って、*deleteOrder()* ロジックを確認します。

注 ***

次の手順は、新しい注文または保留中の注文がある場合にのみ機能します。 前のセクションで新しい注文が既に作成されているはずです。 また、*deleteOrder()* を使用するには承認が必要です。

1. http://localhost:8080 に移動します。
2. ログインしていない場合は、ログインします。
3. 右上のカート アイコンをクリックします。
    a. または、ユーザー アイコンの横にある *[Hello {名}]*をクリックし、ドロップダウン メニューの[*カート*] をクリックし、*カート*ページにリダイレクトする事ができます。
4. [*Cart*]ページの *[Item]*セクションで、アイテムの数の横にある*[Delete*] ボタンをクリックします。 アイテムがカートから消えるはずです。
    a. すべてのアイテムがカートから削除された場合、Web UI はホームページにリダイレクトする必要があります。
    b. *カート*ページに戻ると、「カートに品目がありません No Item in cart.」というメッセージが表示されます。

ヘルパー メソッド

次のヘルパー メソッドは *repository.js* ファイルに既に実装されており、API にロジックを追加するときに使用できます。

次の注文 ID の取得

*getNextOrderId()*メソッドは、カウンタ ドキュメントとアトミック操作を利用して、カウンタ ドキュメントをインクリメントして次の orderId を取得します。

```
0:  getNextOrderId(callback) {
1:    this.bucket.counter(
2:      this.counterIds["order"],
3:      1,
4:      { initial: 5000 },
5:      function (err, res) {
6:        if (err) {
7:          return callback(err, null);
8:        }
9:        callback(err, res.value);
10:     });
11: }
```

次の顧客 ID の取得

getNextCustomerId() メソッドは、カウンタ ドキュメントとアトミック操作を利用して、カウンタ ドキュメントをインクリメントして次の customerId を取得します。

```
0:  getNextCustomerId(callback) {
1:    this.bucket.counter(
2:      this.counterIds["customer"],
3:      1,
4:      { initial: 1000 },
5:      function (err, res) {
6:        if (err) {
7:          return callback(err, null);
8:        }
9:        callback(err, res.value);
10:     }
11:   );
12: }
```

次のユーザー ID の取得

getNextUserId() メソッドは、カウンタ ドキュメントとアトミック操作を利用して、カウンタ ドキュメントをインクリメントして次の userId を取得します。

```
0:  getNextUserId(callback) {
1:    this.bucket.counter(
2:      this.counterIds["user"],
```

```
      1,
      { initial: 1000 },
      function (err, res) {
        if (err) {
          return callback(err, null);
        }
        callback(err, res.value);
      }
    );
  }
```