

execute-transactions



- [Best practices {#best_practices}](#)
 - [Example {#example}](#)
-

You can execute database transactions using an `sql.Tx`, which represents a transaction. In addition to `Commit` and `Rollback` methods representing transaction-specific semantics, `sql.Tx` has all of the methods you use to perform common database operations. To get the `sql.Tx`, you call `DB.Begin` or `DB.BeginTx`.

A [database transaction](#) groups multiple operations as part of a larger goal. All of the operations must succeed or none can, with the data's integrity preserved in either case. Typically, a transaction workflow includes:

1. Beginning the transaction.
2. Performing a set of database operations.
3. If no error occurs, committing the transaction to make database changes.
4. If an error occurs, rolling back the transaction to leave the database unchanged.

The `sql` package provides methods for beginning and concluding a transaction, as well as methods for performing the intervening database operations. These methods correspond to the four steps in the workflow above.

- Begin a transaction.

`DB.Begin` or `DB.BeginTx` begin a new database transaction, returning an `sql.Tx` that represents it.

- Perform database operations.

Using an `sql.Tx`, you can query or update the database in a series of operations that use a single connection. To support this, `Tx` exports the following methods:

- `Exec` and `ExecContext` for making database changes through SQL statements such as `INSERT`, `UPDATE`, and `DELETE`.

For more, see [Executing SQL statements that don't return data](#).

- `Query`, `QueryContext`, `QueryRow`, and `QueryRowContext` for operations that return rows.

For more, see [Querying for data](#).

- `Prepare`, `PrepareContext`, `Stmt`, and `StmtContext` for pre-defining prepared statements.

For more, see [Using prepared statements](#).

- End the transaction with *one* of the following:
 - Commit the transaction using `Tx.Commit`.
If `Commit` succeeds (returns a `nil` error), then all the query results are confirmed as valid and all the executed updates are applied to the database as a single atomic change. If `Commit` fails, then all the results from `Query` and `Exec` on the `Tx` should be discarded as invalid.
 - Roll back the transaction using `Tx.Rollback`.
Even if `Tx.Rollback` fails, the transaction will no longer be valid, nor will it have been committed to the database.

Best practices {#best_practices}

Follow the best practices below to better navigate the complicated semantics and connection management that transactions sometimes require.

- Use the APIs described in this section to manage transactions. Do *not* use transaction-related SQL statements such as `BEGIN` and `COMMIT` directly—doing so can leave your database in an unpredictable state, especially in concurrent programs.
- When using a transaction, take care not to call the non-transaction `sql.DB` methods directly, too, as those will execute outside the transaction, giving your code an inconsistent view of the state of the database or even causing deadlocks.

Example {#example}

Code in the following example uses a transaction to create a new customer order for an album. Along the way, the code will:

1. Begin a transaction.
2. Defer the transaction's rollback. If the transaction succeeds, it will be committed before the function exits, making the deferred rollback call a no-op. If the transaction fails it won't be committed, meaning that the rollback will be called as the function exits.
3. Confirm that there's sufficient inventory for the album the customer is ordering.
4. If there's enough, update the inventory count, reducing it by the number of albums ordered.
5. Create a new order and retrieve the new order's generated ID for the client.
6. Commit the transaction and return the ID.

This example uses `Tx` methods that take a `context.Context` argument. This makes it possible for the function's execution – including database operations -- to be canceled if it runs too long or the client connection closes. For more, see [Canceling in-progress operations](#).

```
// CreateOrder creates an order for an album and returns the new order ID.
func CreateOrder(ctx context.Context, albumID, quantity, custID int) (orderId int64, err
error) {
```

```

// Create a helper function for preparing failure results.
fail := func(err error) (int64, error) {
    return 0, fmt.Errorf("CreateOrder: %v", err)
}

// Get a Tx for making transaction requests.
tx, err := db.BeginTx(ctx, nil)
if err != nil {
    return fail(err)
}
// Defer a rollback in case anything fails.
defer tx.Rollback()

// Confirm that album inventory is enough for the order.
var enough bool
if err = tx.QueryRowContext(ctx, "SELECT (quantity >= ?) from album where id = ?",
    quantity, albumID).Scan(&enough); err != nil {
    if err == sql.ErrNoRows {
        return fail(fmt.Errorf("no such album"))
    }
    return fail(err)
}
if !enough {
    return fail(fmt.Errorf("not enough inventory"))
}

// Update the album inventory to remove the quantity in the order.
_, err = tx.ExecContext(ctx, "UPDATE album SET quantity = quantity - ? WHERE id = ?",
    quantity, albumID)
if err != nil {
    return fail(err)
}

// Create a new row in the album_order table.
result, err := tx.ExecContext(ctx, "INSERT INTO album_order (album_id, cust_id, quantity,
date) VALUES (?, ?, ?, ?)",
    albumID, custID, quantity, time.Now())
if err != nil {
    return fail(err)
}

```

```
// Get the ID of the order item just created.
orderID, err = result.LastInsertId()
if err != nil {
    return fail(err)
}

// Commit the transaction.
if err = tx.Commit(); err != nil {
    return fail(err)
}

// Return the order ID.
return orderID, nil
}
```

#2

2026-10-27 04:26:29 UTC
2026-10-27 04:26:29 UTC