DATABASE TRANSACTION

What is a database transaction

A database transaction is a single unit of work that consists of one or more operations.

A classical example of a transaction is a bank transfer from one account to another. A complete transaction must ensure a balance between the sender and receiver accounts. It means that if the sender account transfers `X` amount, the receiver receives `X` amount, no more or no less.

A PostgreSQL transaction is atomic, consistent, isolated, and durable. These properties are often referred to as ACID:

- Atomicity guarantees that the transaction completes in an all-or-nothing manner.
- Consistency ensures the change to data written to the database must be valid and follow predefined rules.
- Isolation determines how transaction integrity is visible to other transactions.
- Durability makes sure that transactions that have been committed will be stored in the database permanently.

```
   ›››  ~  psql bank postgres
psql (14.4)
Type "help" for help.

bank=# SELECT * FROM accounts;
 id |  name   | balance
----+---------+---------
  1 | Udin    | 9000.00
  2 | Kosasih | 6000.00
(2 rows)

bank=# BEGIN;
BEGIN
bank=*# UPDATE accounts
bank-*# SET balance = balance - 2000
bank-*# WHERE id = 1;
UPDATE 1
bank=*# SELECT * FROM accounts;
 id |  name   | balance
----+---------+---------
  2 | Kosasih | 6000.00
  1 | Udin    | 7000.00
(2 rows)

bank=*# UPDATE accounts
bank-*# SET balance = balance + 2000
bank-*# WHERE id = 2;
UPDATE 1
bank=*# SELECT * FROM accounts;
 id |  name   | balance
----+---------+---------
  1 | Udin    | 7000.00
  2 | Kosasih | 8000.00
(2 rows)

bank=*# COMMIT;
COMMIT
bank=#
```

Transactions

Sequelize does not use transactions by default. However, for production-ready usage of Sequelize, you should definitely configure Sequelize to use transactions.

Sequelize supports two ways of using transactions:

1. **Unmanaged transactions:** Committing and rolling back the transaction should be done manually by the user (by calling the appropriate Sequelize methods).

2. **Managed transactions**: Sequelize will automatically rollback the transaction if any error is thrown, or commit the transaction otherwise. Also, if CLS (Continuation Local Storage) is enabled, all queries within the transaction callback will automatically receive the transaction object.

```
// First, we start a transaction from your connection and save it into a variable
const t = await sequelize.transaction();

try {

  // Then, we do some calls passing this transaction as an option:

  const user = await User.create({
    firstName: 'Bart',
    lastName: 'Simpson'
  }, { transaction: t });

  await user.addSibling({
    firstName: 'Lisa',
    lastName: 'Simpson'
  }, { transaction: t });

  // If the execution reaches this line, no errors were thrown.
  // We commit the transaction.
  await t.commit();

} catch (error) {

  // If the execution reaches this line, an error was thrown.
  // We rollback the transaction.
  await t.rollback();

}
```

As shown above, the *unmanaged transaction* approach requires that you commit and rollback the transaction manually, when necessary.

Managed transactions

```
async payOrder(req, res) {
  const t = await sequelize.transaction();
  // 1. cek order nya ada atau ngga berdasarkan id
  // 2. kalo datanya ada order nya sudah dibayar atau belum
  // 3. kalo belum di bayar cek product ada atau ngga
  // 4. kalo productnya ada cek stock
  // 5. kalo stocknya ada kuragin stock
  // 6. update is_paidnya jadi true dan simpan tanggal pembayaran
  // 7. commit
  //

  try {
    const { id } = req.params;
    const order = await Order.findByPk(id);
    if (!order) throw new Error("order tidak ditemukan");
    if (order.is_paid) throw new Error("order sudah dibayar");
    const product = await Product.findByPk(order.product_id);
    if (!product) throw new Error("product tidak ditemukan");
    if (product.stock < order.quantity) throw new Error("stock tidak cukup");

    await Product.update(
      { stock: product.stock - order.quantity },
      {
        where: {
          id: product.id,
        },
        transaction: t,
      },
    );

    await Order.update(
      {
        is_paid: true,
        paid_date: new Date(),
      },
      {
        where: { id: order.id },
      }
    );
```

```
async payOrder(req, res) {
  const t = await sequelize.transaction();
  // 1. cek order nya ada atau ngga berdasarkan id
  // 2. kalo datanya ada order nya sudah dibayar atau belum
  // 3. kalo belum di bayar cek product ada atau ngga
  // 4. kalo productnya ada cek stock
  // 5. kalo stocknya ada kuragin stock
  // 6. update is_paidnya jadi true dan simpan tanggal pembayaran
  // 7. commit
  //

  try {
    const { id } = req.params;
    const order = await Order.findByPk(id, {transaction: t});
    if (!order) throw new Error("order tidak ditemukan");
    if (order.is_paid) throw new Error("order sudah dibayar");
    const product = await Product.findByPk(order.product_id, {transaction: t});
    if (!product) throw new Error("product tidak ditemukan");
    if (product.stock < order.quantity) throw new Error("stock tidak cukup");

    await Product.update(
      { stock: product.stock - order.quantity },
      {
        where: {
          id: product.id,
        },
        transaction: t,
      }
    );

    await Order.update(
      {
        is_paid: true,
        paid_date: new Date(),
      },
      {
        where: { id: order.id },
        transaction: t,
      }
```

```
    try {
      const { order_id: id } = req.params;
      console.log(id, "<<<<<< id");
      const order = await Order.findByPk(id, { transaction: t });
      if (!order) throw new Error("order tidak ditemukan");
      if (order.is_paid) throw new Error("order sudah dibayar");

      const product = await Product.findByPk(order.product_id, {
        transaction: t,
      });
      if (!product) throw new Error("product tidak ditemukan");
      if (product.stock < order.quantity) throw new Error("stock tidak cukup");

      await Product.update(
        { stock: product.stock - order.quantity },
        {
          where: {
            id: product.id,
          },
          transaction: t,
        }
      );

      await Order.update(
        {
          isPaid: true,
          paidDate: new Date(),
        },
        {
          where: { id: order.id },
          transaction: t,
        }
      );

      await t.commit();
      res.send("pembayaran berhasil");
    } catch (error) {
      console.log(error);
      await t.rollback();
```

NORMAL after-lecture-transaction! <lers/index.js javascript utf-8[unix] 83% ln :84/101≡%:2