https://github.com/mike-works/sql-fundamentals

Frontend Masters
Mar 27, 2018
Mike North

MIKE WORKS

# Why are you talking to me about a database?

- Lines between frontend and backend continue to blur
- Web apps are becoming "thick clients"
- Mechanical sympathy is valuable for improving performance

# Two SQL Courses

- SQL Fundamentals is a great primer for developers who **use** databases.
  - It mostly sticks to common SQL that's implemented the same way across SQLite, PostgreSQL, MySQL, etc...
- Professional SQL is a deeper course, intended for developers who wish to **design and maintain** a database.
  - It tackles several topics that are treated **very differently** depending on your RDBMS. We'll work with MySQL and PostgreSQL examples.

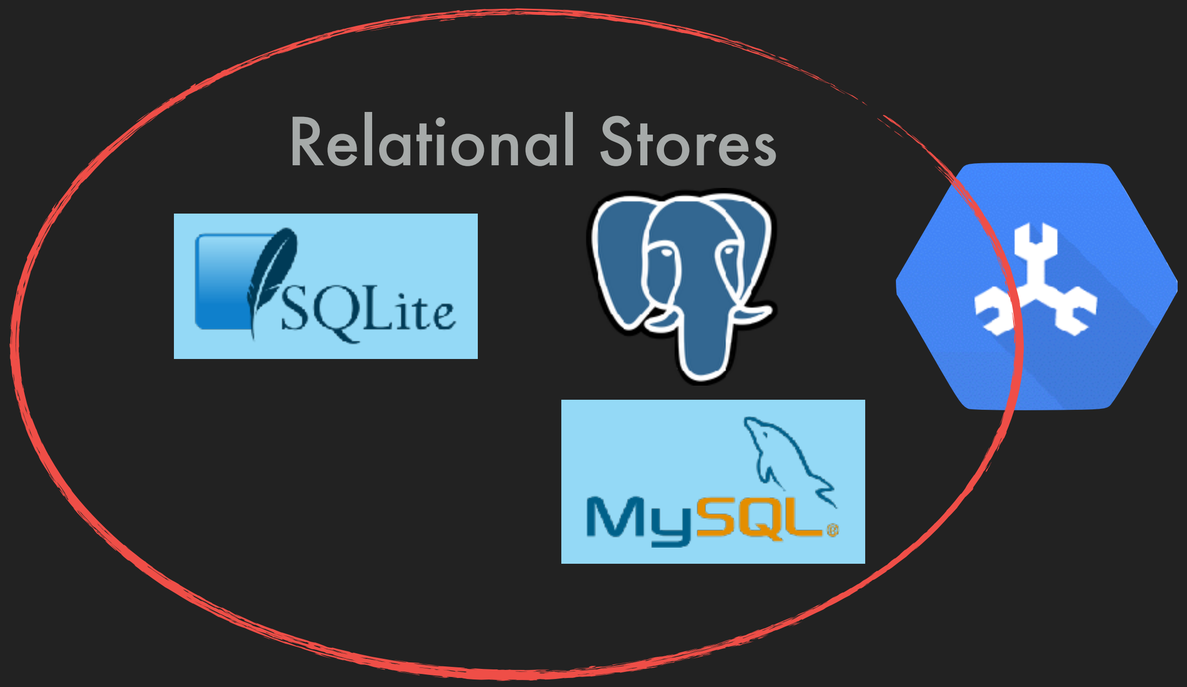
# SQL Fundamentals: Agenda

- Relational Algebra and SQL foundations
- Basic SELECT
- Filtering results with WHERE
- Sorting and paginating
- JOINs
- Aggregate functions and GROUP BY
- Transactions
- Creating/Deleting/Updating Records
- Migrations
- Indices
- Types & Column Constraints

# What makes something a database?

1. Organized collection of persisted data
2. Write something and read it out later

# Databases: a diverse family




Key-Value Stores



Voldemort

Column Stores



Graph Databases


Document Databases

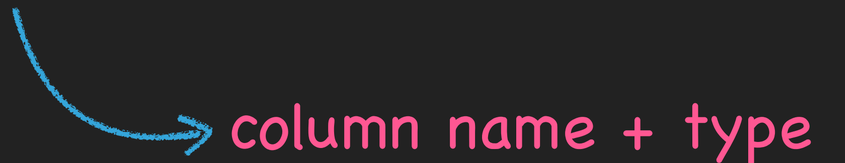
# Our SQL databases


# Codd's Relational Model

- A **relation** is a set of **tuples** ($d_1$, $d_2$, $d_3$, ... $d_n$).


- Each **element in a tuple** $d_j$ is a member of $D_j$, a **data domain**.


- These elements are called **attribute values**, and are associated with an **attribute**.



# Schema

- The "shape" or structure of your data (as opposed to the values themselves)
- Tables, column names, types and constraints
- Strongly connected to your "data domain"

CUSTOMER

| id<br>INTEGER | companyname<br>STRING | contactname<br>STRING |
|---|---|---|
| | | |
| | | |
| | | |

# Relational Algebra vs. Relational Calculus

## Movies Jeff Goldblum is in

### THE RELATIONAL ALGEBRA WAY

- Join Movie and Actor over actorId
- Filter the results to only include those that include Jeff Goldblum
- Select the movieTitle and year columns

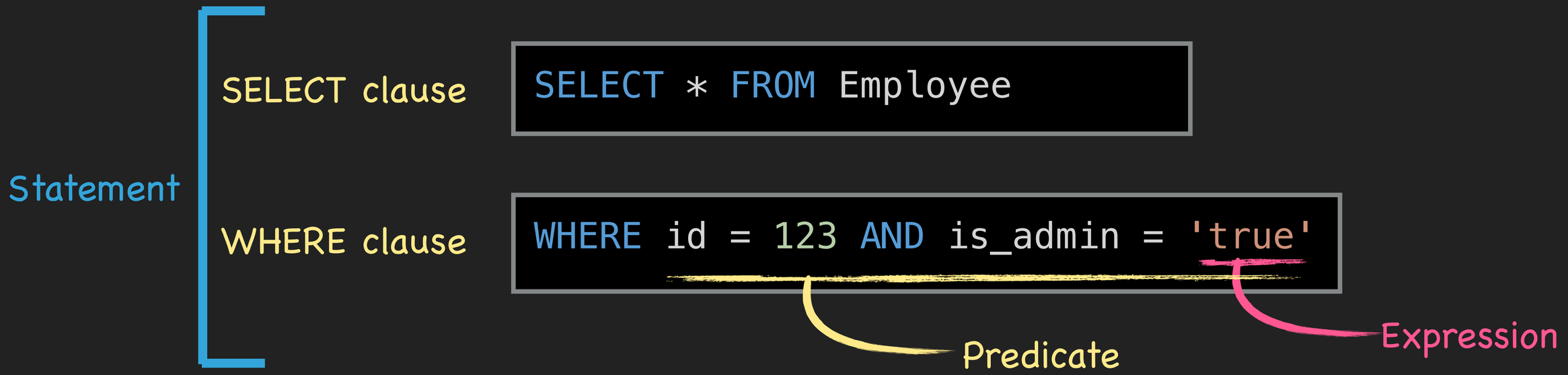
### THE RELATIONAL

- Get movieTitle and year for Movie such that there exists an Actor A who was in the movie and has the name Jeff Goldblum.

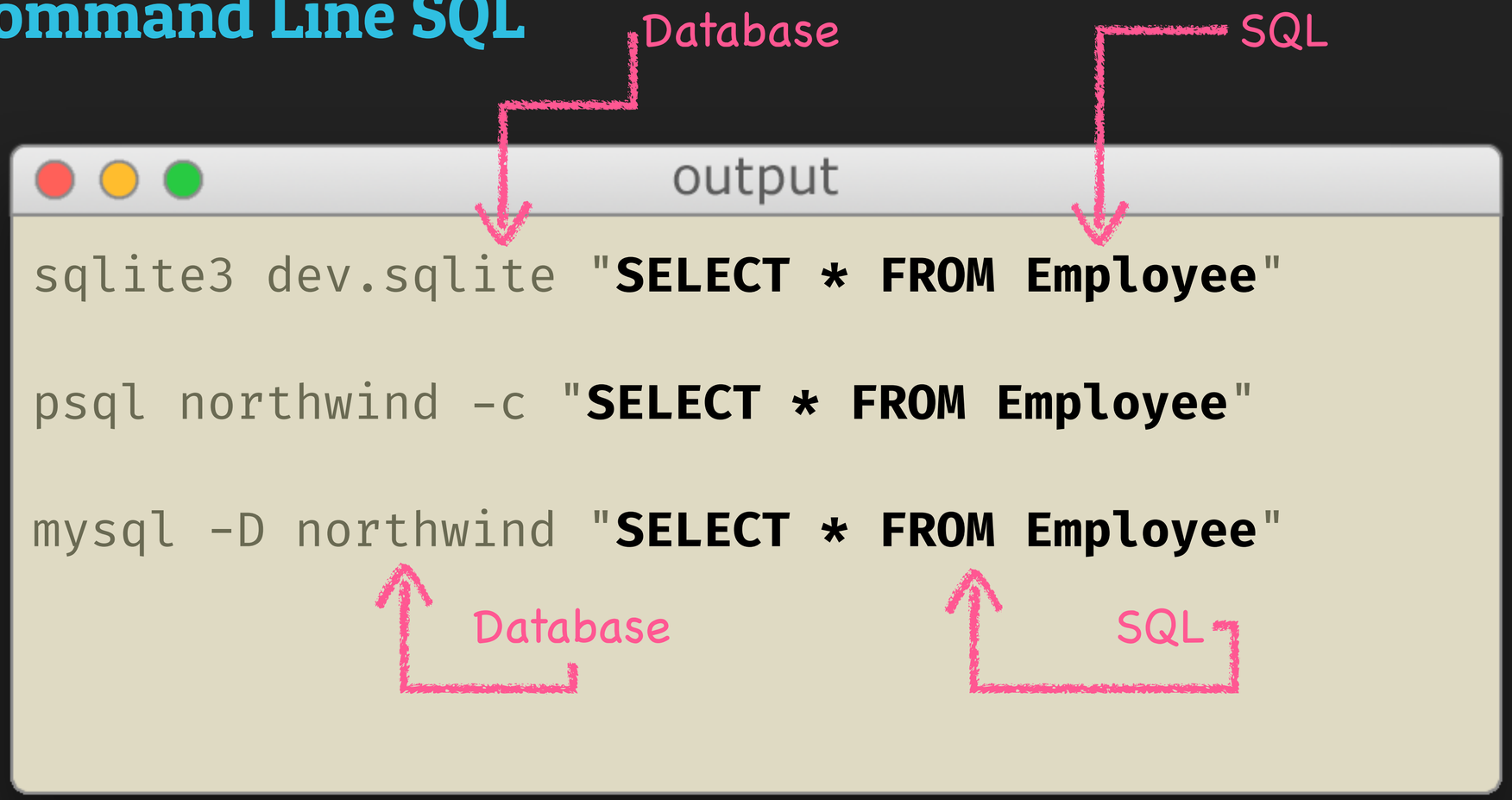
# Structured Query Language

- Used to manage data in a Relational Database Management System (RDBMS)
- Declarative, unlike its predecessors
- Can be procedural too! (PL/SQL)
- Inspired by Codd's Relational Model

# SQL Language Elements



# Command Line SQL



# Creating a Database

output

```
# SQLite db is created automatically
sqlite3 northwind.sqlite

# PostgreSQL
createdb northwind

# MySQL
mysqladmin create 'northwind';
```

# Deleting a Database

output

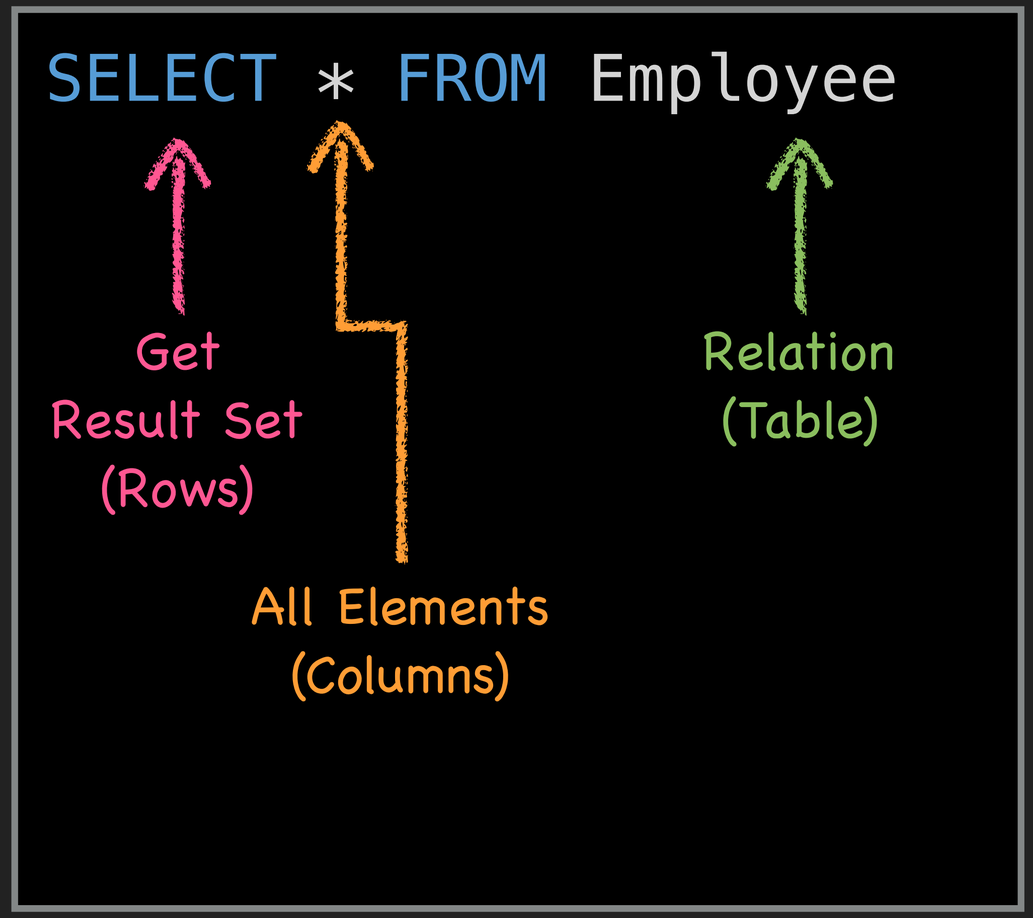
```
# SQLite db is created automatically
rm northwind.sqlite

# PostgreSQL
dropdb northwind

# MySQL
mysqladmin drop 'northwind';
```

# SELECT (very simplified)

- Returns a result set
- For now, assume results are rows
- **FROM** allows one or more elements to be returned from a relation



# SELECT - Choosing Elements

- Elements are returned in the order they're asked for
- Good idea to be explicit because
  - Future schema changes will not affect results
  - Developer intent is clear
  - Less I/O

```
SELECT id, firstname,
lastname
FROM Employee
```

Choose elements (Columns) in order

# Quotes in SQL

- Usually table names, column names and keywords are case insensitive
- 'Single quotes' are used for string literals
- "Double quotes" or `backticks` for words that conflict with SQL keywords, or when case sensitivity is desired

MySQL only

```
SELECT id FROM Employee

SELECT Id FROM Employee

SELECT 'id' FROM Employee

SELECT "Id" FROM Employee
```

output

```
> "SELECT id FROM Employee"
1 2 3 4 5 6 7 8 9
> "SELECT Id FROM Employee"
1 2 3 4 5 6 7 8 9
> "SELECT 'id' FROM Employee"
id id id id id id id id id
> "SELECT \"Id\" FROM Employee"
ERROR:  column "Id" does not exist
```

# Aliases

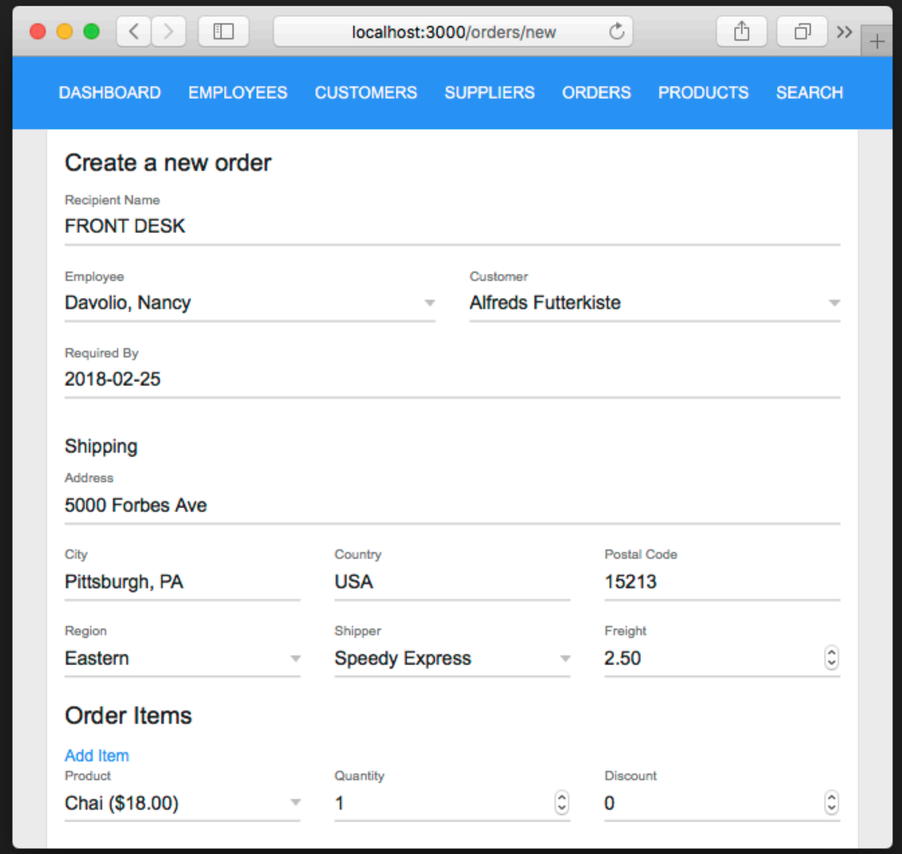
- The AS keyword can be used to give a table or column a local name or "alias"
- Only impacts the current query

```
SELECT
  p.productname AS title
FROM
  Product AS p
```

| title |
| --- |
| Chai |
| Chang |
| Aniseed Syrup |
| Chef Anton's Cajun Seasoning |
| Chef Anton's Gumbo Mix |

# Our Project

- ./src/data - data layer code (SQL queries)
- ./src/routers - Express routers (HTTP handling)
- ./src/db - abstractions around JS database drivers
- ./test - exercise tests
- ./views - handlebars templates
- ./public - static assets



# Our Project

- The SQL tagged template literal can be used to syntax highlight

```
let query = sql`SELECT * FROM Employee`;
```

- Get a database client via `getDb()`

```
import { getDb } from '../db/utils';

let db = await getDb();
// Retrieve a collection of records
let allEmployees = await db.all('SELECT * FROM Employee');
// Retrieve a single record
let product71 = await db.get('SELECT * FROM Product WHERE id = $1', 71);
// Execute a statement, and return the last inserted ID (if applicable)
let { id } = await db.run('INSERT INTO Customer VALUES(...)');
```

# Our Project

- To setup a database

```
npm run db:setup:pg
```

```
npm run db:setup:mysql
```

```
npm run db:setup:sqlite
```

- Run tests that match a filter

```
npm run test EX01
```

```
npm run test:watch EX01
```

- Run an exercise's tests, and all tests from previous exercises

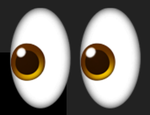
```
npm run test:ex 4
```

```
npm run test:ex:watch 4
```

# Our Project

- To run the project on http://localhost:3000

```
npm run watch
```



- Run tests with a database other than SQLite

```
DB_TYPE=pg npm run watch
```

```
DB_TYPE=mysql npm run watch
```

# SELECTing columns

1

- There are several queries in our app that indiscriminately select all columns in a table.
- Fix this so that we explicitly ask for particular columns we're interested in
- Consider cases where this part of the query may be used multiple times

| Order | Employee | Supplier | Customer |
|---|---|---|---|
| id<br>customerid<br>employeeid<br>shipcity<br>shipcountry<br>shippeddate | id<br>firstname<br>lastname<br>region<br>hiredate<br>title<br>reportsto | id<br>contactname<br>companyname | id<br>contactname<br>companyname |

```
npm run test:ex:watch 1
```

# SELECTing columns

1

- Get Customers

./src/data/customers.js
```
let customers = await getAllCustomers();
```

- Get Employees

./src/data/employees.js
```
let employees = await getAllEmployees();
```

- Get Suppliers

./src/data/suppliers.js
```
let suppliers = await getAllSuppliers();
```

- Get Orders

./src/data/orders.js
```
await getAllOrders();
```

./src/data/orders.js
```
await getCustomerOrders('ALFKI');
```

```
npm run test:ex:watch 1
```

# WHERE

- Used to filter result set with a **condition**
- Mind your quotes!

```
SELECT
  firstname,
  lastname
FROM
  Employee
WHERE
  lastname = 'Leverling'
```

| firstname | lastname |
|---|---|
| "Janet" | "Leverling" |
| | |
| | |
| | |
| | |

# Conditions

- `>`, `<`, `>=`, `<=`, `=`
- Not equal: `<>` or `!=`.
- Within a range
- Member of a set
- String ends with
- String includes
- String length

```
age >= 21
isDeleted != TRUE
temperature BETWEEN 68 AND 75
companyname IN ('Microsoft', 'LinkedIn')
email LIKE '%.gov'
summary LIKE '%spicy%'
billing_state LIKE '__'
```

# AND, OR, NOT

- Boolean (logic) operators
- Operate on conditions
- Parens for clarity

```
SELECT
  productname
FROM
  Product
WHERE (
  unitprice > 60
  AND unitsinstock > 20
)
```

| productname |
| --- |
| Mishi Kobe Niku |
| Carnarvon Tigers |
| Sir Rodney's Marmalade |

# Core Functions

- Each database's set of core functions is slightly different (SQLite, MySQL, PostgreSQL)
- Some of these work across databases (lower, max, min, count, substr, etc...)
- Fine to use them in a comparison

```
SELECT productname
FROM Product
WHERE lower(productname) LIKE '%dried%'
```

- ...or as an additional column

```
SELECT lower(productname) AS label
FROM Product
```

# Debugging conditions

- Conditions can be evaluated directly with a SELECT statement

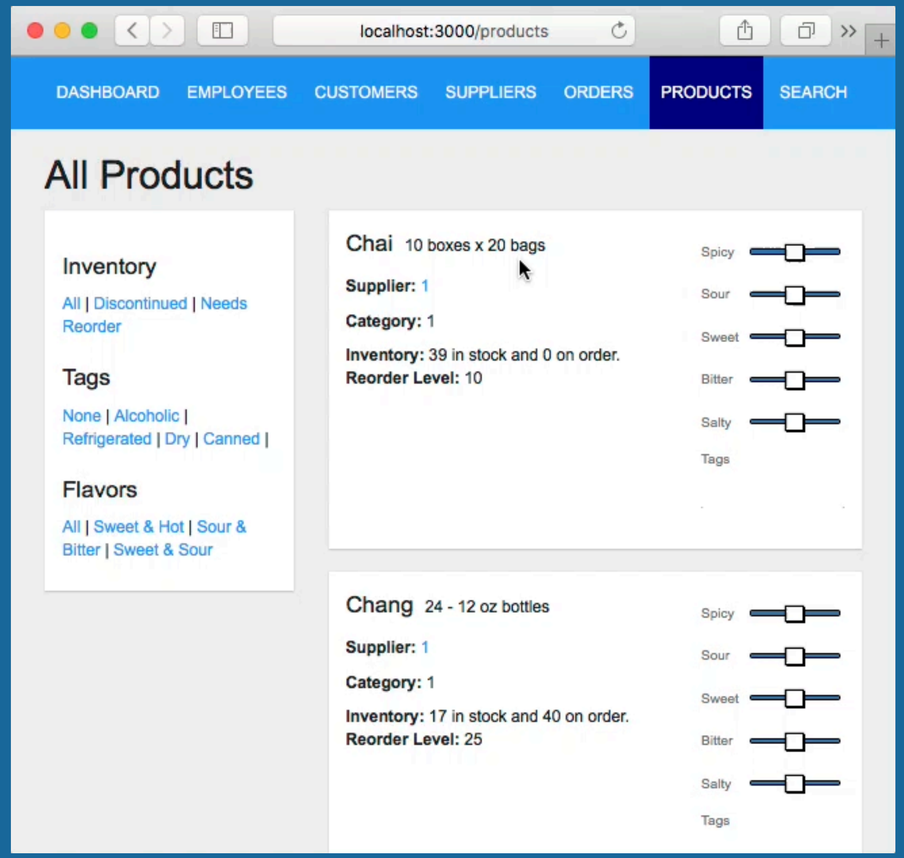
```
SELECT 'mike@example.com' LIKE '%@example.com'; -- TRUE
SELECT 'mike@gmail.com' LIKE '%@example.com'; -- FALSE
```

# Tools Demo

# Filtering with WHERE

2

- On the Products page, the "discontinued" and "needs reorder" links don't work yet.
- In `getAllProducts`, use the options passed to the function to filter the result set appropriately
- `Product.discontinued` is 1 when the product is discontinued
- We need to reorder when `(Product.unitsinstock + Product.unitsonorder) < Product.reorderlevel`




./src/data/products.js

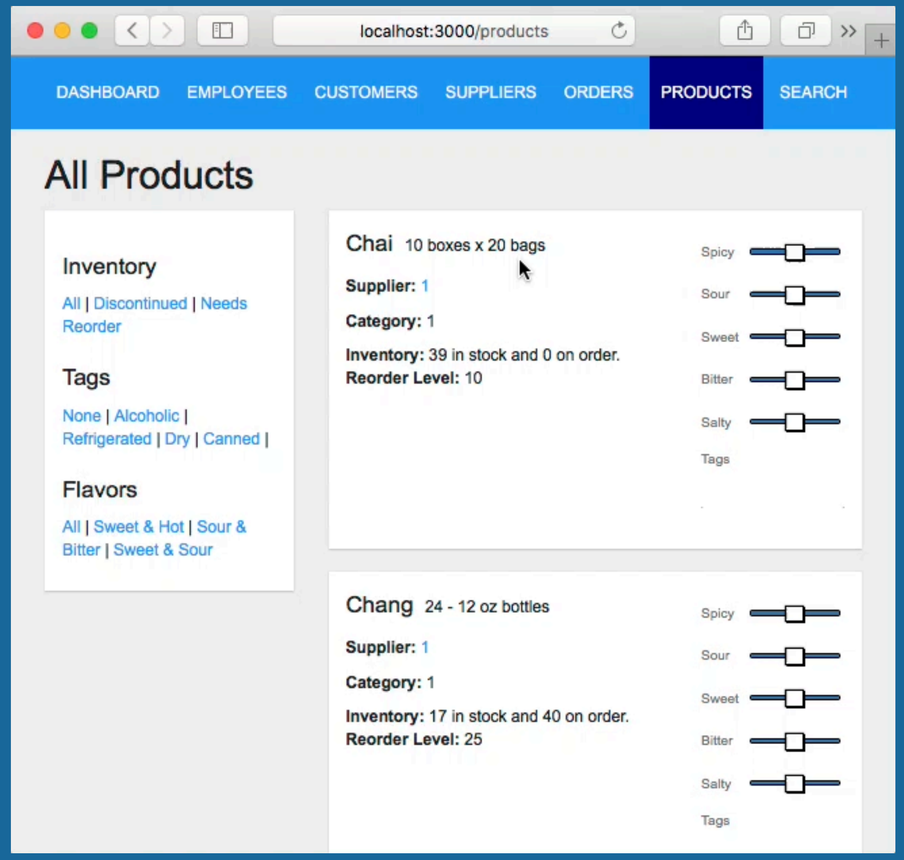
```
getAllProducts({ filter: { inventory: 'discontinued' } });
getAllProducts({ filter: { inventory: 'needs-reorder' } });
```

```
npm run test:ex:watch 2
```

# Filtering with WHERE

2

- On the Customers page, filtering the list by name doesn't work.
- In `getAllCustomers`, use the options passed to the function filter on `Customer.companyName` and `Customer.contactName` columns for case-insensitive values containing the "search filter"




./src/data/customers.js

```
getAllCustomers({ filter: 'Mike'});
```

```
npm run test:ex:watch 2
```

# Sorting

- An `ORDER BY` clause declares the desired sorting of the result set
- Specify sort direction as `ASC` or `DESC`

```
SELECT productname,
unitprice
FROM Product
ORDER BY unitprice DESC
```

| productname | unitprice |
|---|---|
| Côte de Blaye | 263.5 |
| Thüringer | 123.79 |
| Mishi Kobe Niku | 97 |
| Sir Rodney's | 81 |
| Carnarvon Tigers | 62.5 |

# Sorting

- An `ORDER BY` clause declares the desired sorting of the result set
- Specify sort direction as `ASC` or `DESC`
- Multiple sorts and directions may be provided, separated by commas

```
SELECT productname, unitprice
FROM Product
WHERE unitprice BETWEEN 9.6 AND 11
ORDER BY unitprice, productname ASC
```

| productname | unitprice |
|---|---|
| Jack's New England Clam Chowder | 9.65 |
| Aniseed Syrup | 10 |
| Longlife Tofu | 10 |
| Sir Rodney's Scones | 10 |

# Sorting

- An `ORDER BY` clause declares the desired sorting of the result set
- Specify sort direction as `ASC` or `DESC`
- Multiple sorts and directions may be provided, separated by commas

```
SELECT productname, unitprice
FROM Product
WHERE unitprice BETWEEN 9.6 AND 11
ORDER BY unitprice ASC, productname DESC
```

| productname | unitprice |
|---|---|
| Jack's New England Clam Chowder | 9.65 |
| Sir Rodney's Scones | 10 |
| Longlife Tofu | 10 |
| Aniseed Syrup | 10 |

# Limiting

- When dealing with large result sets, sometimes we just want the first N rows
- A `LIMIT` clause allows us to specify how many we want
- Performance can be much higher, **if** the database doesn't have to examine each record for sorting purposes

```
SELECT productname, unitprice
FROM Product
ORDER BY unitprice DESC
LIMIT 3
```

| productname | unitprice |
|---|---|
| Côte de Blaye | 263.5 |
| Thüringer Rostbratwurst | 123.79 |
| Mishi Kobe Niku | 97 |

# Offsetting

- Means "Start with the Nth result"
- We can paginate over a set of results using a `LIMIT` and `OFFSET`

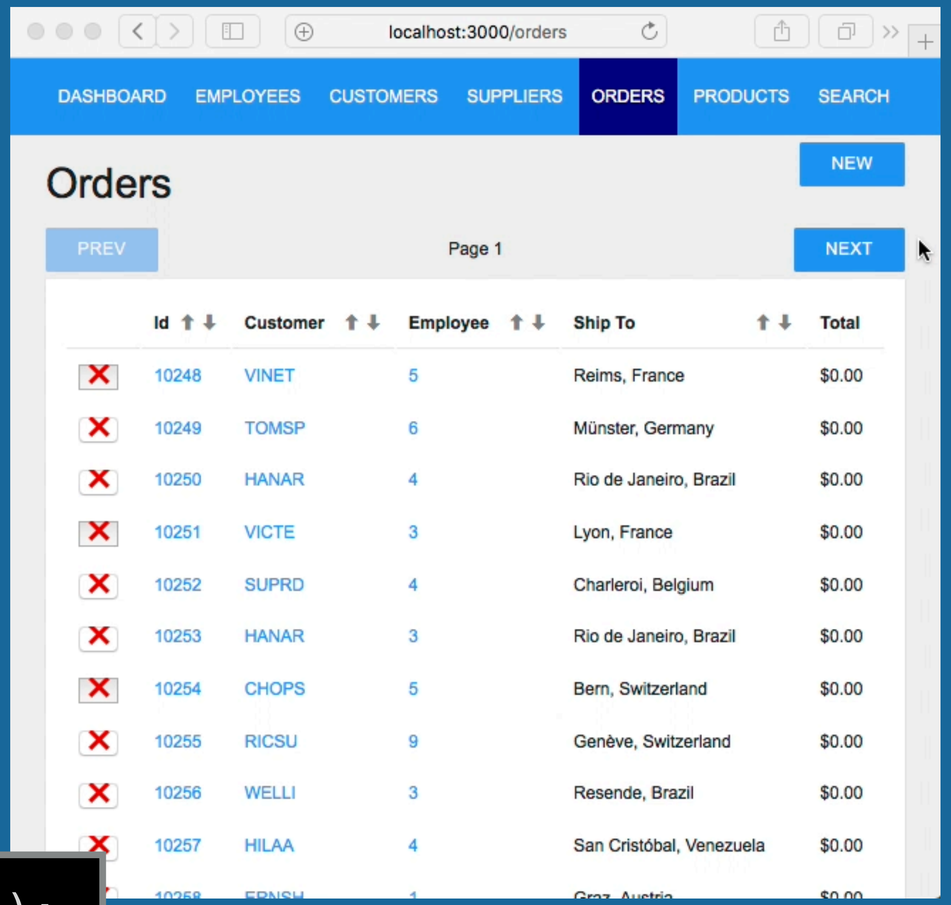

```
SELECT productname,
       unitprice
FROM Product
ORDER BY unitprice DESC
LIMIT 3
OFFSET 3
```

| productname | unitprice |
|---|---|
| Sir Rodney's Marmalade | 81 |
| Carnarvon Tigers | 62.5 |
| Raclette Courdavault | 55 |

# Sorting and Paging

3

- Clicking on a column header should sort orders.
- Update `getAllOrders` / `getCustomerOrders` to take the `opts.sort` and `opts.order` options into account
- Fix `getCustomerOrders` while you're at it, so that it returns only the appropriate orders




./src/data/orders.js

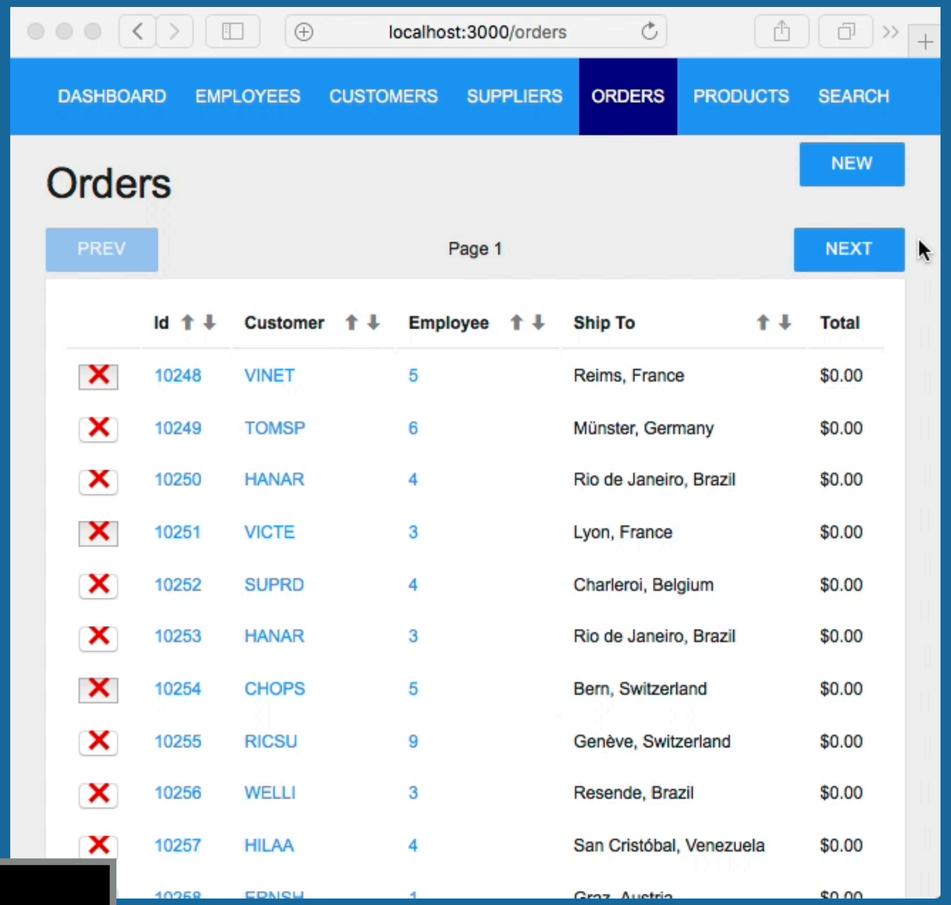
```
getAllOrders({ sort: 'shippeddate', order: 'desc'});
getAllOrders({ sort: 'customerid', order: 'asc' });
```

```
npm run test:ex:watch 3
```

# Sorting and Paging

3

- Pagination buttons are already passing values into `getAllOrders` / `getCustomerOrders` that can be used to offset and limit the result set (`opts.page` and `opts.perPage`).
- Default sorts:
  `getAllOrders` by id, desc
  `getCustomerOrders` by shippeddate asc



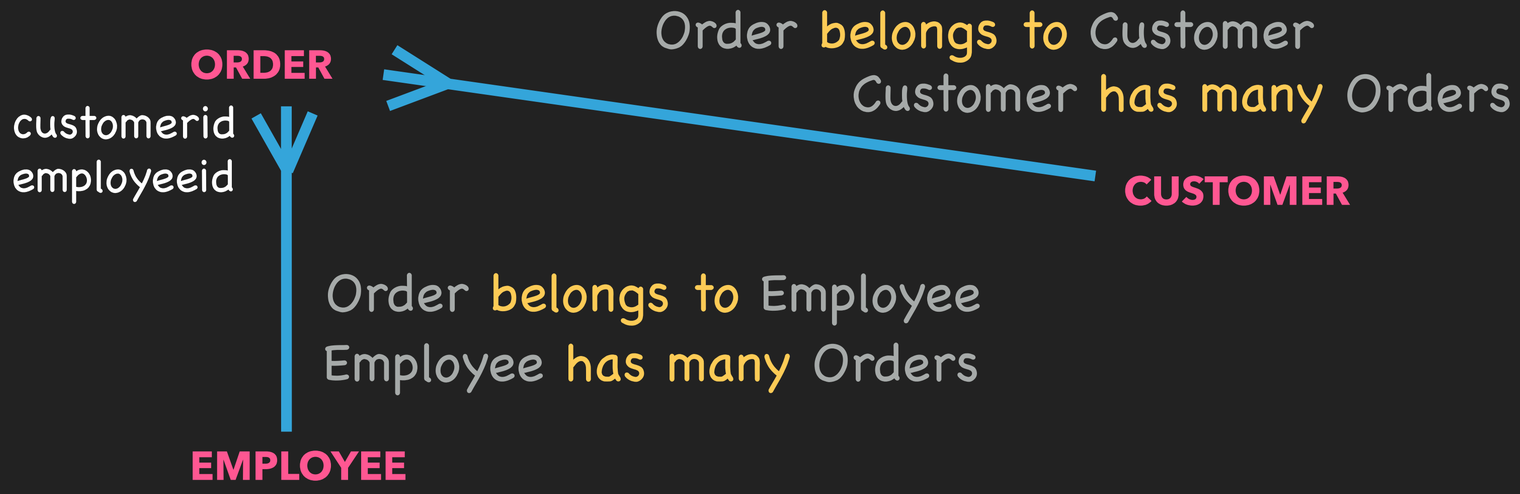

./src/data/orders.js

```
getAllOrders({ page: 3, perPage: 25 });
getCustomerOrders('ALFKI', { page: 5, perPage: 10 });
```

```
npm run test:ex:watch 3
```

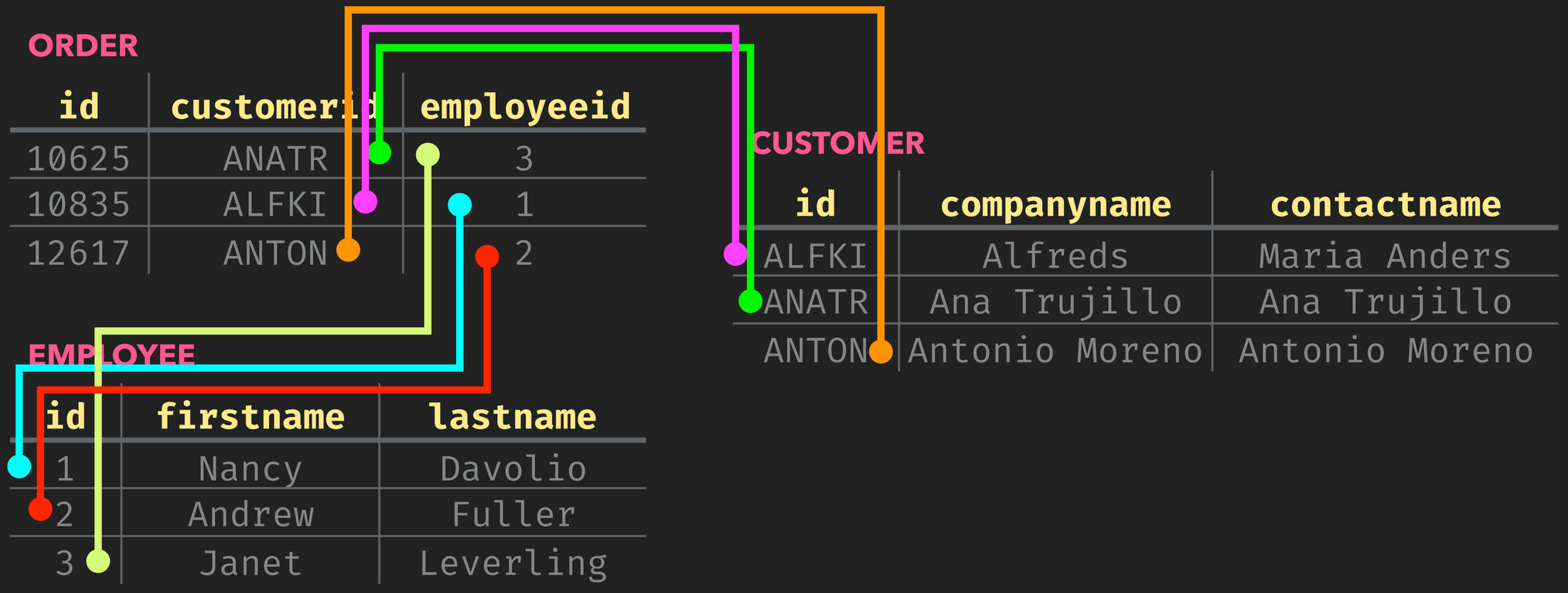
# Relationships

- Reminder: this is **not** where "relational" comes from



# Relationships

- Reminder: this is **not** where "relational" comes from

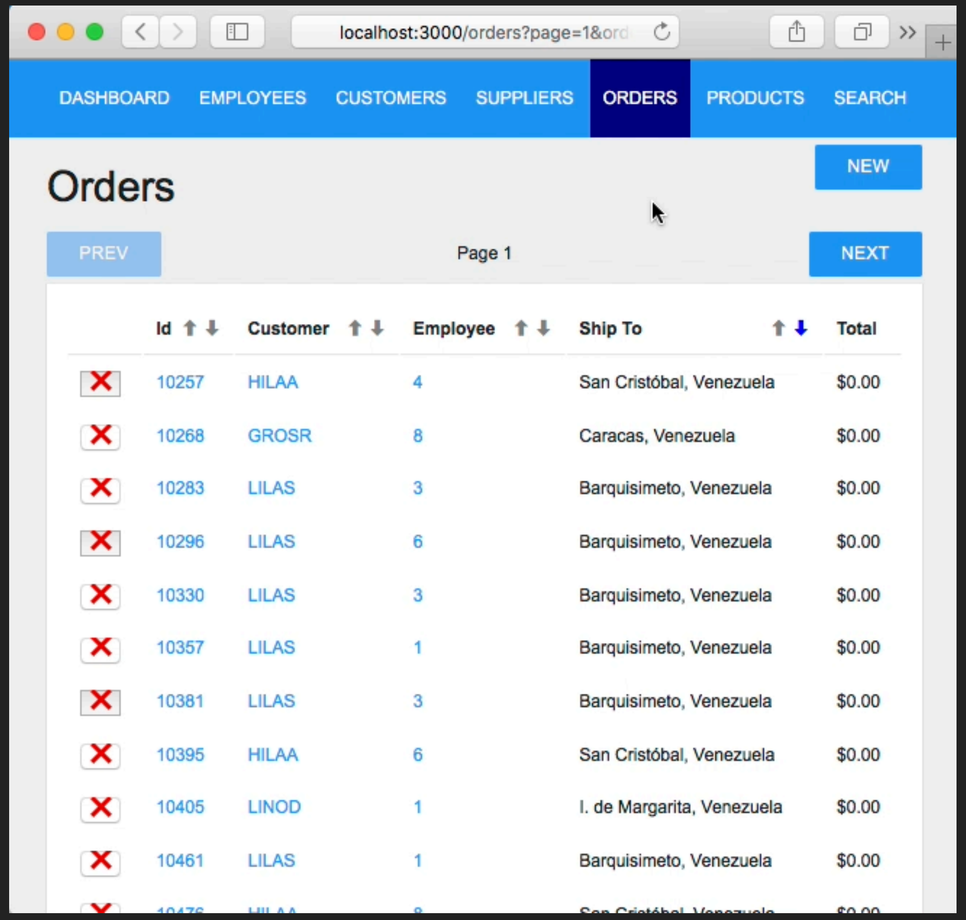


# Resolving Relationships

- Currently we have a result set of objects that look like this

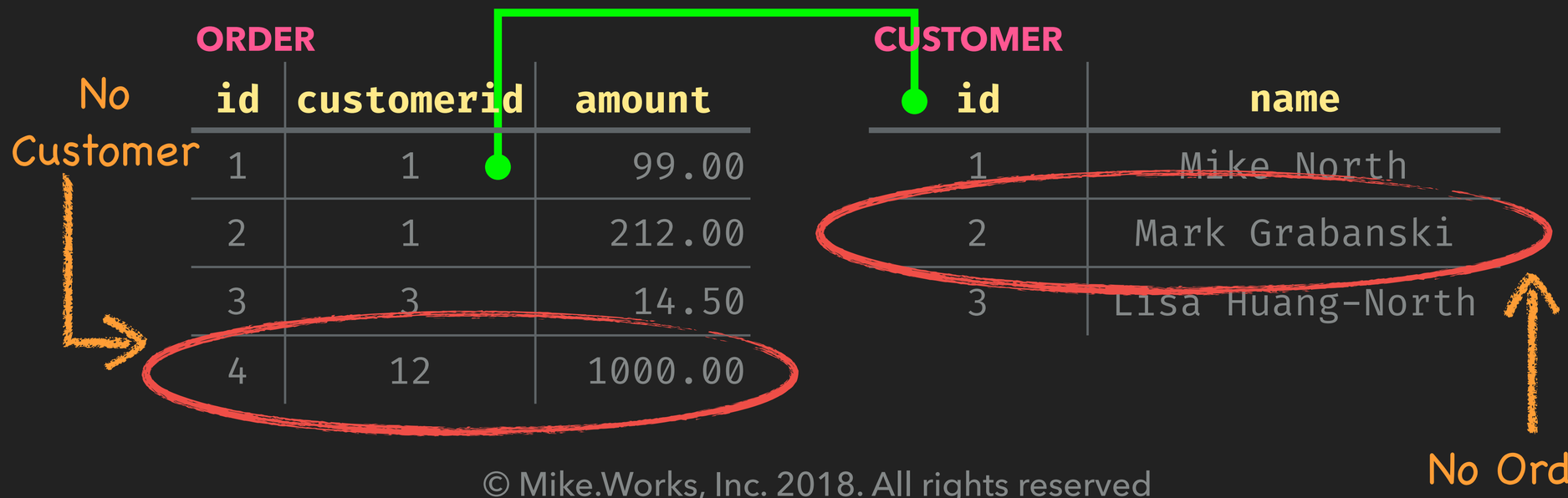
```
[{}, {}, {
  id: 10257,
  customerid: 'HILAA',
  employeeid: 4,
  shipcity: 'San Cristóbal',
  shipcountry: 'Venezuela'
}, ...]
```

- We want to replace `customerid` with the customer's name, but that lives in another table

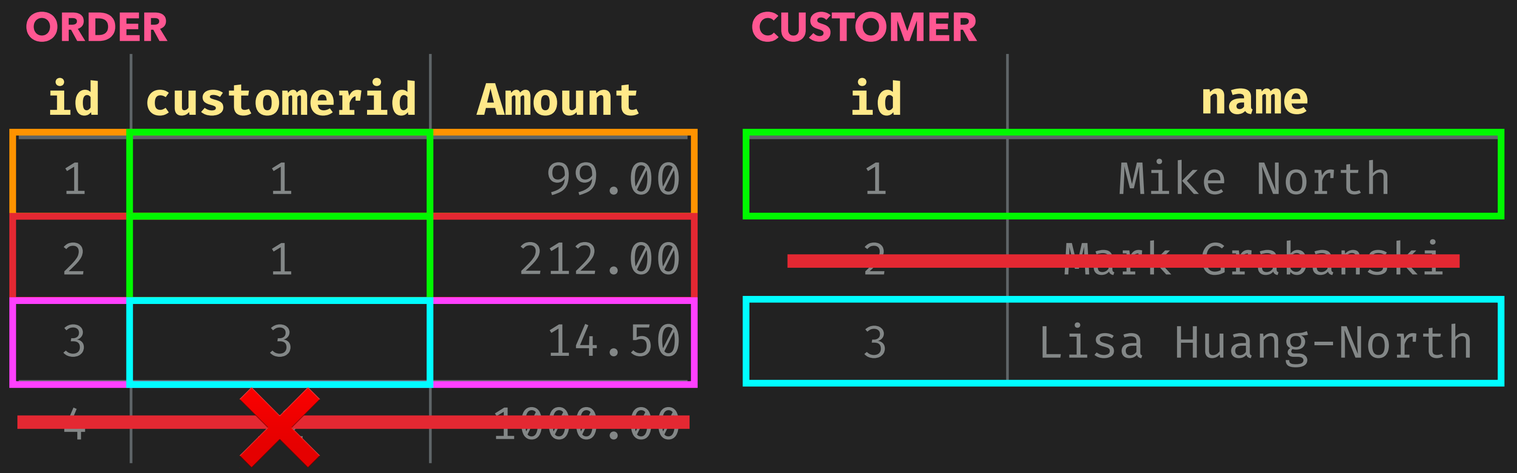


# JOIN: Assemble tables together

- The JOINing is done on a related column between the tables
- Four types of join, that differ in terms of how "incomplete" matches are handled
- Simplified example



# INNER JOIN




- Only rows that have "both ends" of the match will be selected

```
SELECT *
FROM CustomerOrder AS o
INNER JOIN Customer AS c
    ON o.customerid = c.id
```

| id | customerid | amount | id | name |
|---|---|---|---|---|
| 1 | 1 | 99.00 | 1 | Mike North |
| 2 | 1 | 212.00 | 1 | Mike North |
| 3 | 3 | 14.50 | 3 | Lisa Huang-North |

Click for SQLfiddle

# LEFT JOIN

**ORDER**

| id | customerid | Amount |
|---|---|---|
| 1 | 1 | 99.00 |
| 2 | 1 | 212.00 |
| 3 | 3 | 14.50 |
| 4 | 12 | 1000.00 |

**CUSTOMER**

| id | name |
|---|---|
| 1 | Mike North |
| 2 | Mark Grabanski |
| 3 | Lisa Huang-North |

- Rows from LEFT of join will be selected no matter what

```
SELECT *
FROM CustomerOrder AS o
LEFT JOIN Customer AS c
    ON c.id = o.customerid
```

| id | customerid | amount | id | name |
|---|---|---|---|---|
| 1 | 1 | 99.00 | 1 | Mike North |
| 2 | 1 | 212.00 | 1 | Mike North |
| 3 | 3 | 14.50 | 3 | Lisa Huang-North |
| 4 | 12 | 1000.00 | | |

Click for SQLfiddle

# RIGHT JOIN

**ORDER**

| id | customerid | Amount |
|---|---|---|
| 1 | 1 | 99.00 |
| 2 | 1 | 212.00 |
| 3 | 3 | 14.50 |
| ~~4~~ | ~~2~~ | ~~1000.00~~ |

**CUSTOMER**

| id | name |
|---|---|
| 1 | Mike North |
| ~~2~~ | ~~Mark Grabanski~~ |
| 3 | Lisa Huang-North |

- Rows from RIGHT of join will be selected no matter what

```
SELECT *
FROM CustomerOrder AS o
RIGHT JOIN Customer AS c
    ON o.customerid = c.id
```

| id | customerid | amount | id | name |
|---|---|---|---|---|
| 1 | 1 | 99.00 | 1 | Mike North |
| 2 | 1 | 212.00 | 1 | Mike North |
| 3 | 3 | 14.50 | 3 | Lisa Huang-North |
| | | | 2 | Mark Grabanski |

Click for SQLfiddle

# FULL JOIN

**ORDER**

| id | customerid | Amount |
|---|---|---|
| 1 | 1 | 99.00 |
| 2 | 1 | 212.00 |
| 3 | 3 | 14.50 |
| ~~4~~ | ~~12~~ | ~~1000.00~~ |

**CUSTOMER**

| id | name |
|---|---|
| 1 | Mike North |
| 2 | Mark Grabanski |
| 3 | Lisa Huang-North |

- All rows in both tables are selected

```
SELECT *
FROM CustomerOrder AS o
FULL JOIN Customer AS c
    ON o.customerid = c.id
```

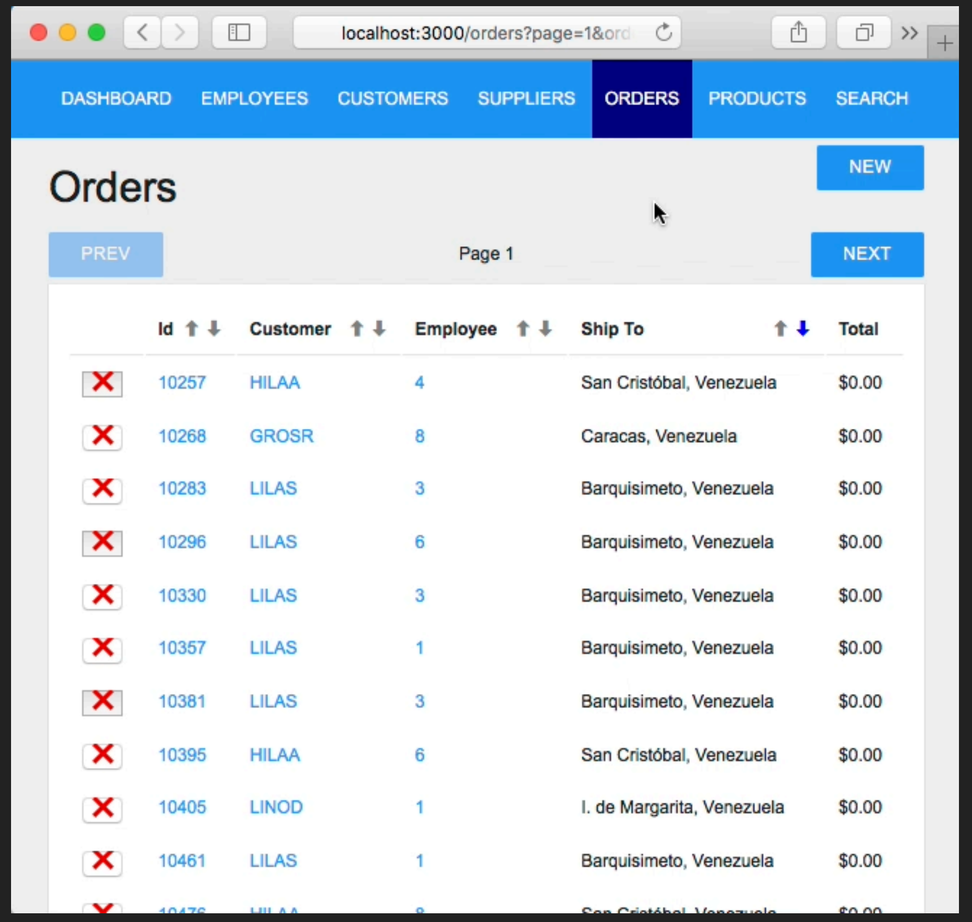
| id | customerid | amount | id | name |
|---|---|---|---|---|
| 1 | 1 | 99.00 | 1 | Mike North |
| 2 | 1 | 212.00 | 1 | Mike North |
| 3 | 3 | 14.50 | 3 | Lisa Huang-North |
| | | | 2 | Mark Grabanski |
| 4 | 12 | 1000.00 | | |

Click for SQLfiddle

# Outer Joins

- `LEFT`, `RIGHT` and `FULL JOIN` are sometimes referred to as `OUTER` joins
- In general, an `OUTER` join allows for the possibility that one or more rows may be partially empty, due to not having a corresponding match in the other table

# Which type of JOIN would we want here?



# JOIN - Selecting Columns

- `SELECT *` is generally a bad idea, and it's particularly dangerous in the context of a `JOIN`
- Choose columns with care, and alias any duplicates

```
SELECT o.id,
       o.customerid,
       o.amount,
       c.name
FROM CustomerOrder AS o
INNER JOIN Customer AS c
    ON o.customerid = c.id
```

| id | customerid | amount | name |
|---|---|---|---|
| 1 | 1 | 99.00 | Mike North |
| 2 | 1 | 212.00 | Mike North |
| 3 | 3 | 14.50 | Lisa Huang-North |

Click for SQLfiddle

# Querying across tables

4

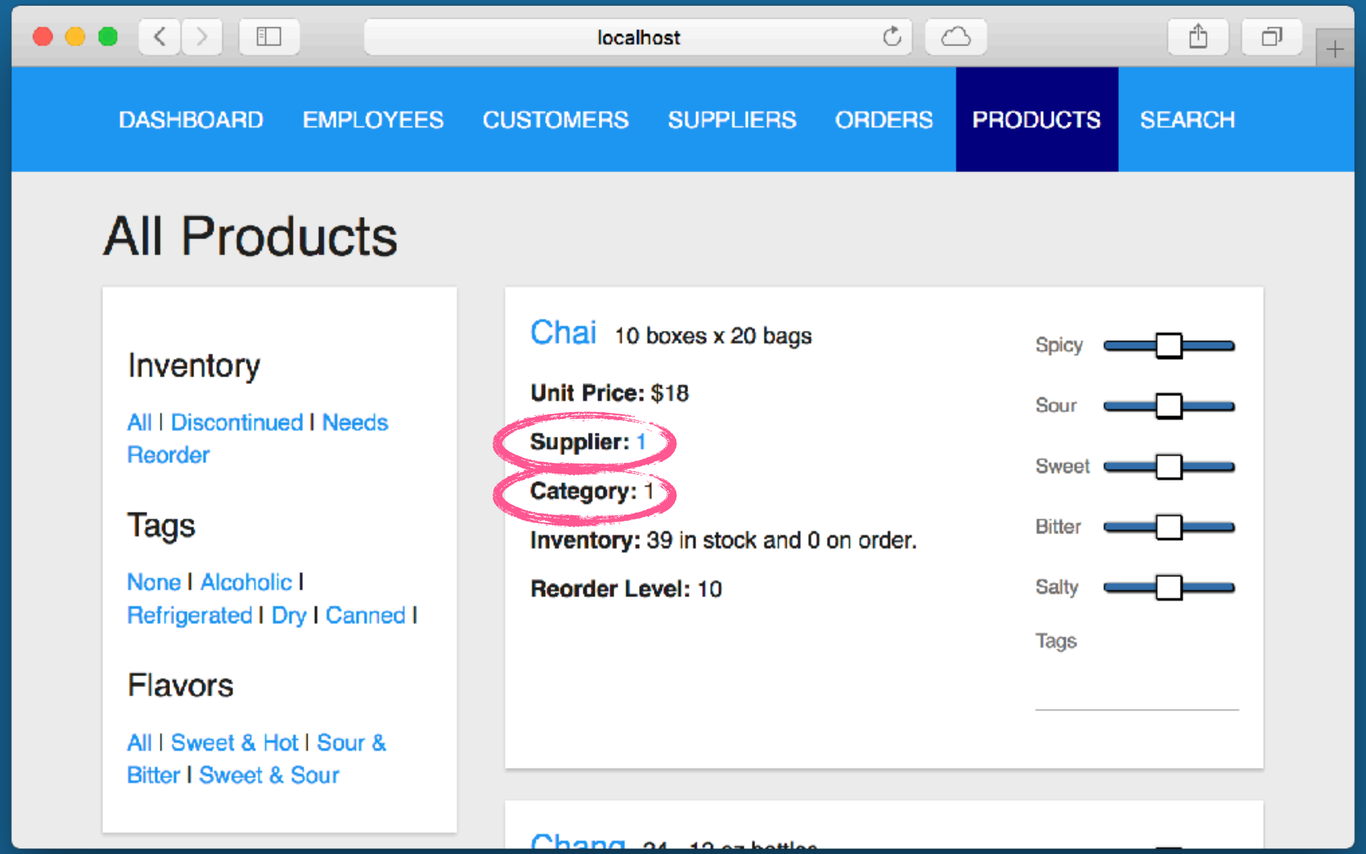
- There are a few places where we're showing Ids
  - Product list: Suppliers, Categories
  - Order list: Customers and Employees
  - Order: Customer, Employee and Products
- Alter the relevant queries to include one or more JOINs so that these Ids are replaced with the relevant human-readable name

```
npm run test:ex:watch 4
```

# Querying across tables

4

- Product list: replace Supplier and Category Ids
- New join columns: suppliername, categoryname




./src/data/products.js

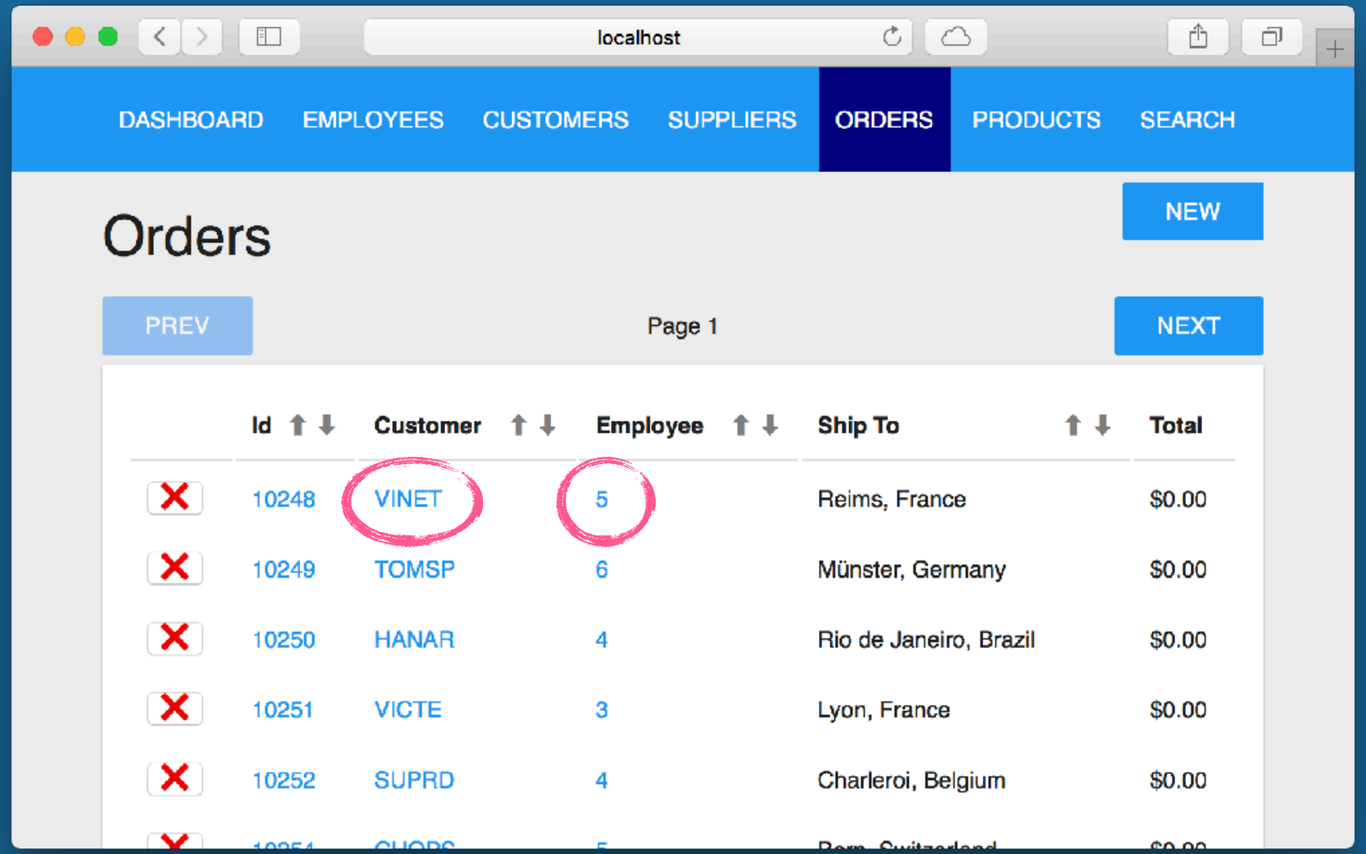
```
getAllProducts();
```

```
npm run test:ex:watch 4
```

# Querying across tables

4

- Order list: replace Customer and Employee Ids
- New join columns: customername, employeename




./src/data/orders.js

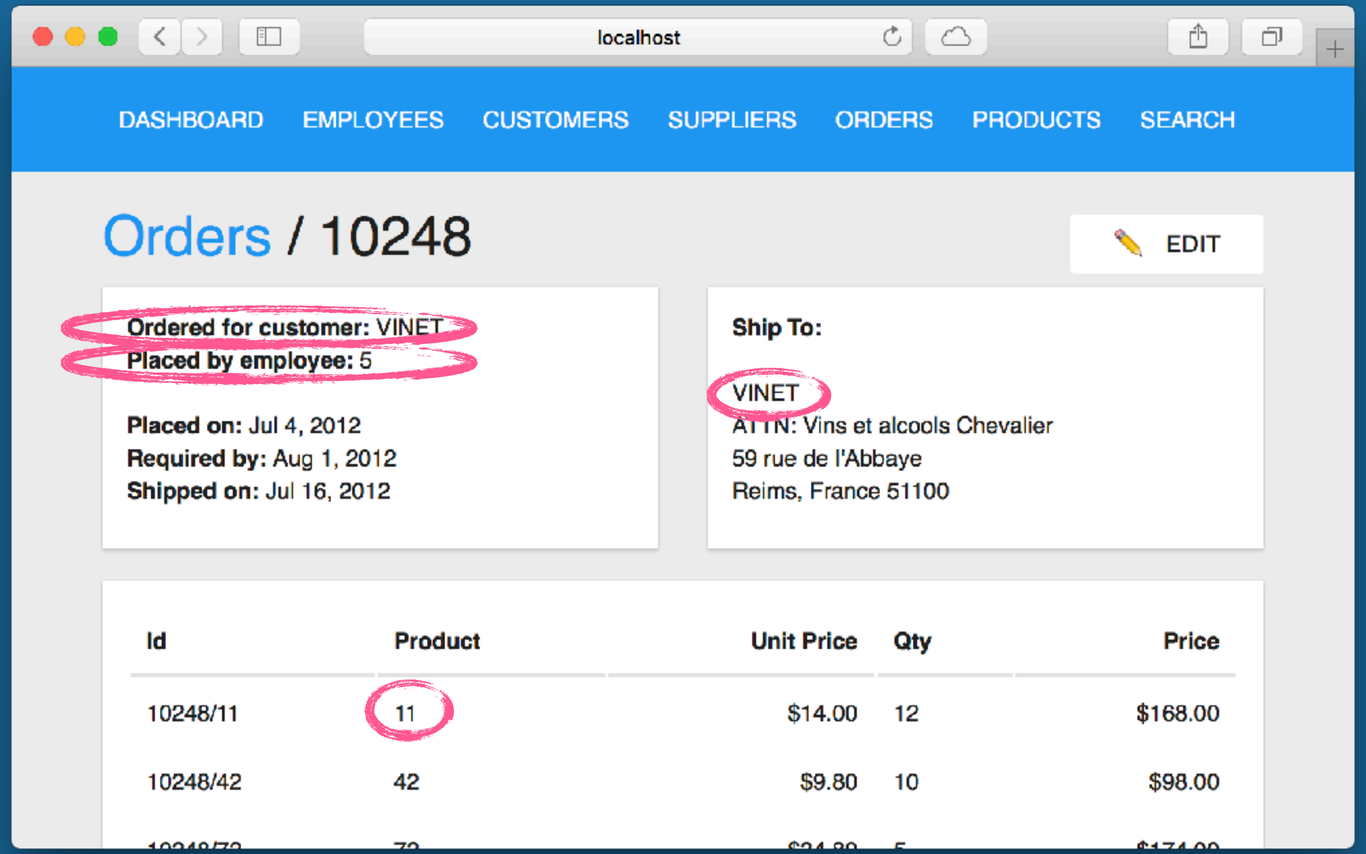
```
getAllOrders();
getCustomerOrders();
```

```
npm run test:ex:watch 4
```

# Querying across tables

4

- Order: replace Customer and Employee Ids
- OrderDetail: replace  Product Ids
- New join columns: customername, employeename, productname



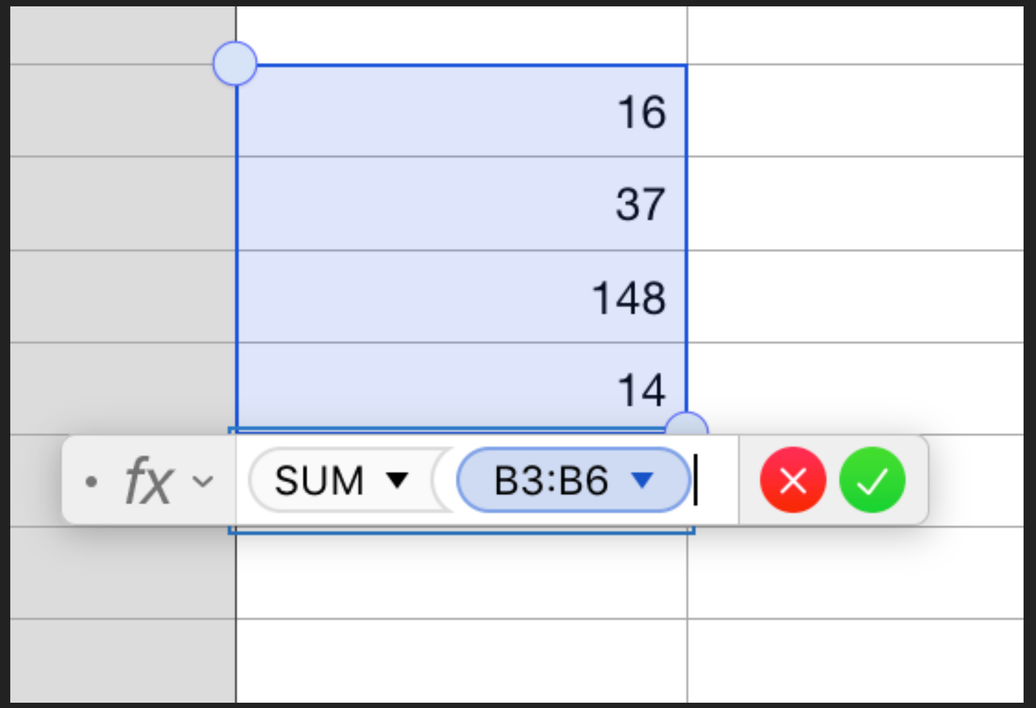

./src/data/orders.js

```
getOrder(12149);
getOrderDetails(12149);
```

```
npm run test:ex:watch 4
```

# Aggregate Functions

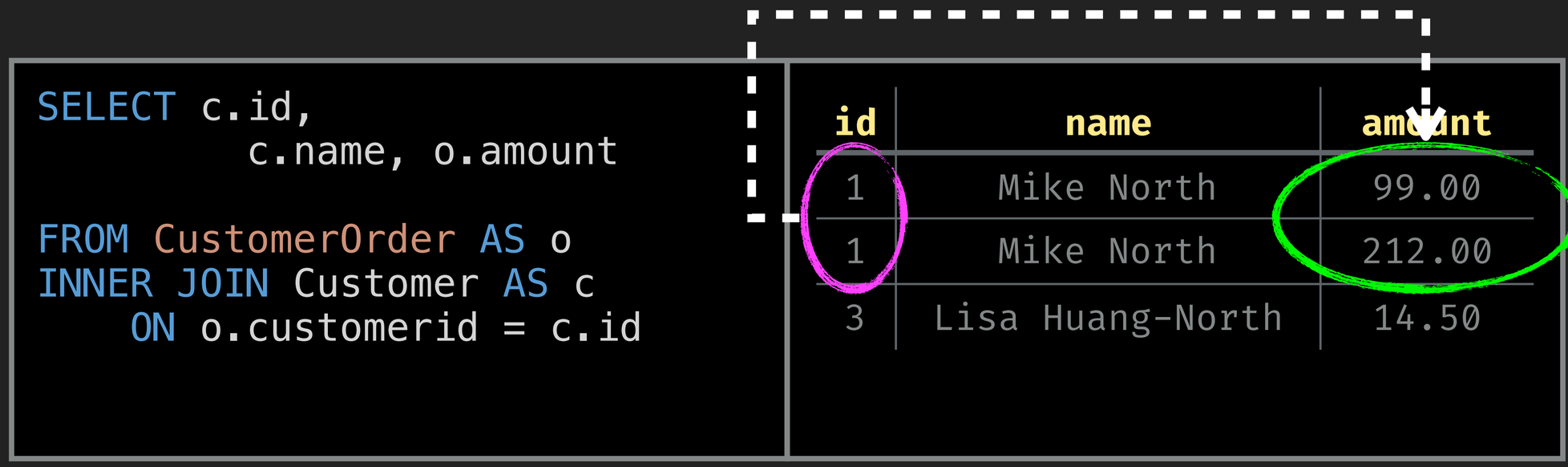
- Perform a calculation on a set of values, to arrive at a single value
- Each RDBMS has its own functions, but they all generally have a foundational set
  - SUM - Adds values together
  - COUNT - Counts the number of values
  - MIN/MAX - Finds the min/max value
  - AVG - Calculates the mean of the values




```
SELECT sum((1 - discount) *
unitprice * quantity)
 AS revenue
FROM OrderDetail;
```

# Aggregate Functions and GROUP BY

- Imagine we have this JOIN query, and we want total amount per customer
- This involves finding the relevant rows to consolidate and summing them up. A relational DB is perfect for these tasks!




Click for SQLfiddle

# Aggregate Functions and GROUP BY

- We can specify which rows to consolidate via a GROUP BY clause
- Then, we can SELECT an aggregate function, describing what to do on the "groups" of rows

```
SELECT c.id,
       c.name,
       sum(o.amount)
FROM CustomerOrder AS o
INNER JOIN Customer AS c
    ON o.customerid = c.id
GROUP BY  c.id
```

| id | name | sum |
|---|---|---|
| 1 | Mike North | 311.00 |
| 3 | Lisa Huang-North | 14.50 |

Click for SQLfiddle

# Aggregate Functions and GROUP BY

- Let's imagine what we could do with GROUP BY in this example
  - Total spent by customerid
  - Total spent by month
  - Total spent by customerid, by month

**CUSTOMER**

| id | name |
|---|---|
| 1 | Mike North |
| 2 | Mark Grabanski |
| 3 | Lisa Huang-North |

**ORDER**

| id | customerid | amount | month |
|---|---|---|---|
| 1 | 1 | 99.00 | 5 |
| 2 | 1 | 212.00 | 3 |
| 3 | 3 | 14.50 | 3 |
| 4 | 12 | 1000.00 | 11 |
| 5 | 1 | 199.00 | 3 |

# Aggregate Functions and GROUP BY

- Total spent by customerid

```
SELECT customerid,
       sum(amount)
FROM CustomerOrder
GROUP BY customerid
```

| id | sum |
|---|---|
| 1 | 510.00 |
| 12 | 1000.00 |
| 3 | 14.50 |

**CUSTOMER**

| id | name |
|---|---|
| 1 | Mike North |
| 2 | Mark Grabanski |
| 3 | Lisa Huang-North |

**ORDER**

| id | customerid | amount | month |
|---|---|---|---|
| 1 | 1 | 99.00 | 5 |
| 2 | 1 | 212.00 | 3 |
| 3 | 3 | 14.50 | 3 |
| 4 | 12 | 1000.00 | 11 |
| 5 | 1 | 199.00 | 3 |

Click for SQLfiddle

# Aggregate Functions and GROUP BY

- Total spent by month

```
SELECT month,
       sum(amount)
FROM CustomerOrder
GROUP BY month
```

| month | sum |
|---|---|
| 3 | 425.50 |
| 5 | 99.00 |
| 11 | 1000 |

CUSTOMER

| id | name |
|---|---|
| 1 | Mike North |
| 2 | Mark Grabanski |
| 3 | Lisa Huang-North |

ORDER

| id | customerid | amount | month |
|---|---|---|---|
| 1 | 1 | 99.00 | 5 |
| 2 | 1 | 212.00 | 3 |
| 3 | 3 | 14.50 | 3 |
| 4 | 12 | 1000.00 | 11 |
| 5 | 1 | 199.00 | 3 |

Click for SQLfiddle

# Aggregate Functions and GROUP BY

- Total spent by customerid, by month

```
SELECT customerid,
       month,
       sum(amount)
FROM CustomerOrder
GROUP BY month, customerid
```

| customerid | month | sum |
|---|---|---|
| 1 | 3 | 411.00 |
| 1 | 5 | 99.00 |
| 3 | 3 | 14.50 |
| 12 | 11 | 1000.00 |

**CUSTOMER**

| id | name |
|---|---|
| 1 | Mike North |
| 2 | Mark Grabanski |
| 3 | Lisa Huang-North |

**ORDER**

| id | customerid | amount | month |
|---|---|---|---|
| 1 | 1 | 99.00 | 5 |
| 2 | 1 | 212.00 | 3 |
| 3 | 3 | 14.50 | 3 |
| 4 | 12 | 1000.00 | 11 |
| 5 | 1 | 199.00 | 3 |

Click for SQLfiddle

# Aggregate Functions and GROUP BY

- What you SELECT, aggregate and GROUP BY must agree

```
SELECT month,
       customerid,
       sum(amount)
FROM CustomerOrder
GROUP BY month
```

```
column
"CustomerOrder.customerid"
must appear in the GROUP BY
clause or be used in an
aggregate function
```

**CUSTOMER**

| id | name |
|---|---|
| 1 | Mike North |
| 2 | Mark Grabanski |
| 3 | Lisa Huang-North |

**ORDER**

| id | customerid | amount | month |
|---|---|---|---|
| 1 | 1 | 99.00 | 5 |
| 2 | 1 | 212.00 | 3 |
| 3 | 3 | 14.50 | 3 |
| 4 | 12 | 1000.00 | 11 |
| 5 | 1 | 199.00 | 3 |

Click for SQLfiddle

# Aggregate Functions

- Different for SQLite, PostgreSQL, MySQL, etc...
- A platform-specific one: string concatenation (PG: `string_agg`, SQLite: `group_concat`)

```
SELECT
  c.id, c.name,
  string_agg(CAST(o.id AS text), ', ') AS orders
FROM Customer AS c
LEFT JOIN CustomerOrder AS o
    ON o.customerid = c.id
GROUP BY  c.id, c.name
ORDER BY c.id
```

pg

| id | name | orders |
|---|---|---|
| 1 | Mike North | "1, 2, 5" |
| 2 | Mark Grabanski | |
| 3 | Lisa Huang-North | "3" |

Click for SQLfiddle

# Concatenating In Aggregate

- Different for SQLite, PostgreSQL, MySQL, etc...
- Most common functions work everywhere: SUM, AVG, MIN, MAX, COUNT
- A platform-specific one: string concatenation (PG: `string_agg`, SQLite: `group_concat`)

```
SELECT
  c.id, c.name,
  string_agg(CAST(o.id AS text), ', ') AS orders
FROM Customer AS c
LEFT JOIN CustomerOrder AS o
    ON o.customerid = c.id
GROUP BY  c.id
ORDER BY c.id
```

Convert integer to string

Concatenate strings

pg

| id | name | orders |
|---|---|---|
| 1 | Mike North | "1, 2, 5" |
| 2 | Mark Grabanski | |
| 3 | Lisa Huang-North | "3" |

# Concatenating In Aggregate

- Concatenating strings together can be done in all of the databases we're studying

MySQL

```
SELECT group_concat(productname ORDER BY productname DESC SEPARATOR ', ')
FROM Product;
```

SQLite

```
SELECT group_concat(productname, ', ')
FROM Product;
```

pg

```
SELECT string_agg(productname, ', ')
FROM Product;
```

# Filtering grouped results

- A `WHERE` clause is used to examine individual rows. It cannot be used with an aggregate function

```
SELECT month,
     sum(amount) AS month_sum
FROM CustomerOrder
WHERE month_sum >= 300
GROUP BY  month
```

```
column "month_sum" does not exist
```

Click for SQLfiddle

# Filtering grouped results

- A `WHERE` clause is used to examine individual rows. It cannot be used with an aggregate function
- A `HAVING` clause is used to examine groups, and include or exclude them from the result set
- It's totally fine to use both in the same query.
  `WHERE` to disregard irrelevant rows
  `HAVING` to disregard irrelevant groups

```
SELECT month,
     sum(amount) AS month_sum
FROM CustomerOrder
WHERE month_sum >= 300
GROUP BY  month
```

```
column "month_sum" does not exist
```

```
SELECT month,
    sum(amount) AS month_sum
FROM CustomerOrder
GROUP BY month
HAVING sum(amount) >= 300
```

Click for SQLfiddle

# Subquery

- A SELECT query can be nested in another query
- Useful when results from the database are needed in order to define "outer" query
- Inner query cannot mutate data
- Sometimes your RDBMS doesn't support ORDER BY within the subquery

```
SELECT *
FROM Product
WHERE categoryid=(
  SELECT id
  FROM Category
  WHERE categoryname='Beverages'
);
```

# If you want to solve exercises using more than one database

```
switch (process.env.DB_TYPE)
{
  case 'mysql' /*...*/:
    break;
  case 'pg' /*...*/:
    break;
  case 'sqlite':
  default:
    /*...*/
    break;
}
```

# Aggregate Functions

5

- Use aggregate functions and GROUP BY to calculate
  - Order: `subtotal`
  - Supplier: `productlist` (concatenate Product.productname strings)
  - Employee: `ordercount`
  - Customer: `ordercount`

```
npm run test:ex:watch 5
```

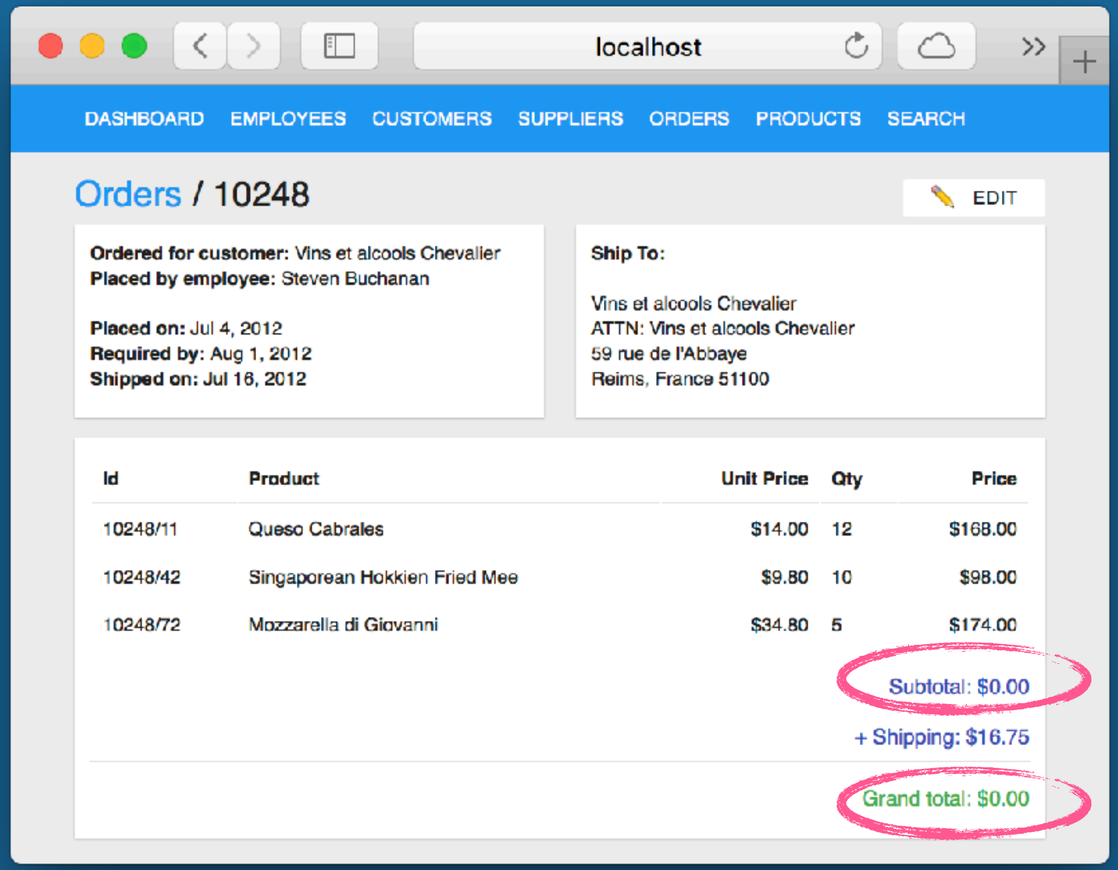
# Aggregate Functions

5

- Order: `subtotal`

./src/data/orders.js

```
getOrderDetails(12149);
```




```
npm run test:ex:watch 5
```

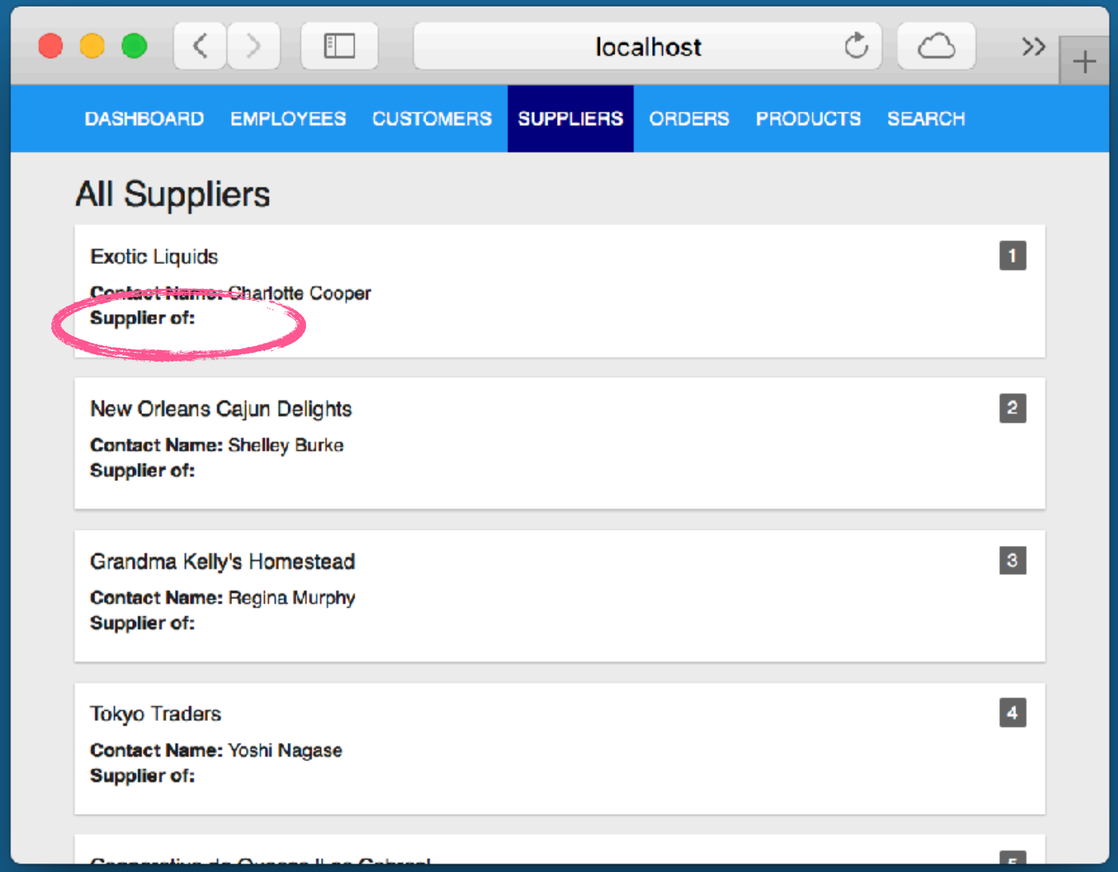
# Aggregate Functions

5

- Supplier: `productlist` (concatenate Product.productname strings)

./src/data/suppliers.js

```
getAllSuppliers();
```




```
npm run test:ex:watch 5
```

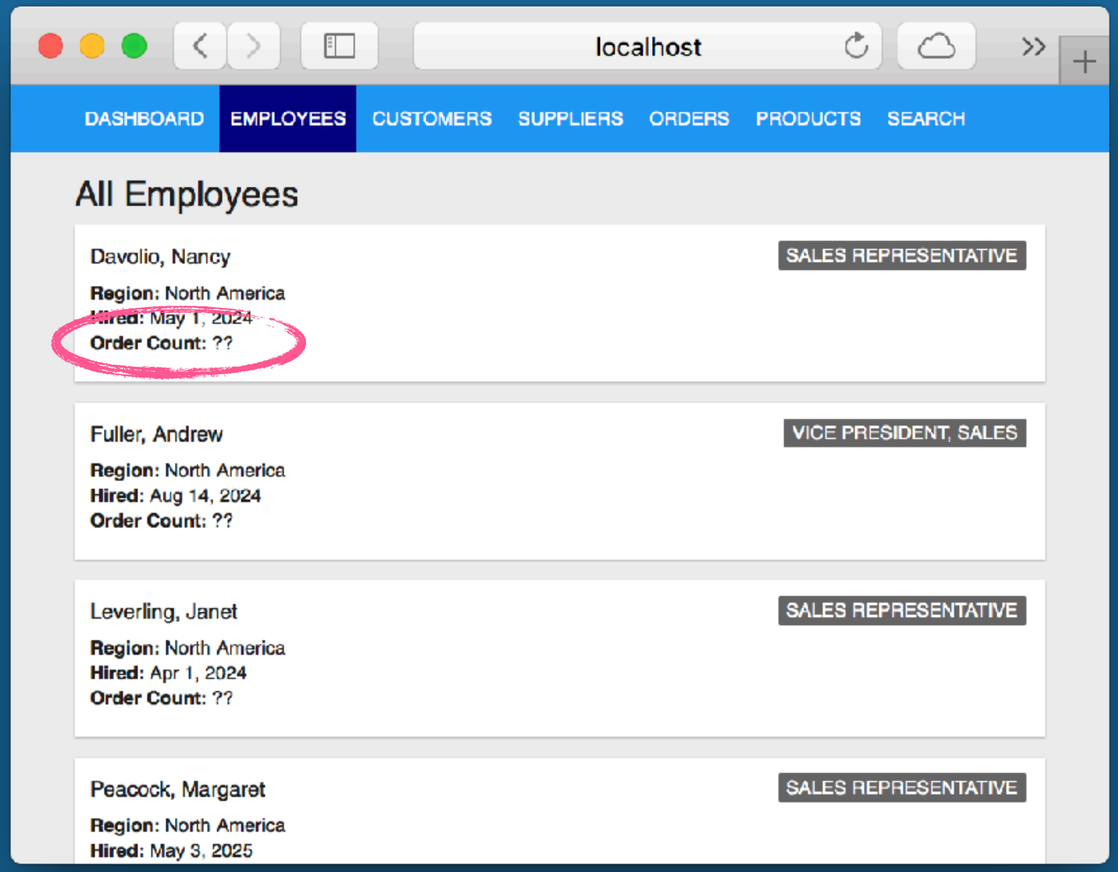
# Aggregate Functions

5

- Employee: `ordercount`

./src/data/employees.js

```
getAllEmployees();
```




```
npm run test:ex:watch 5
```

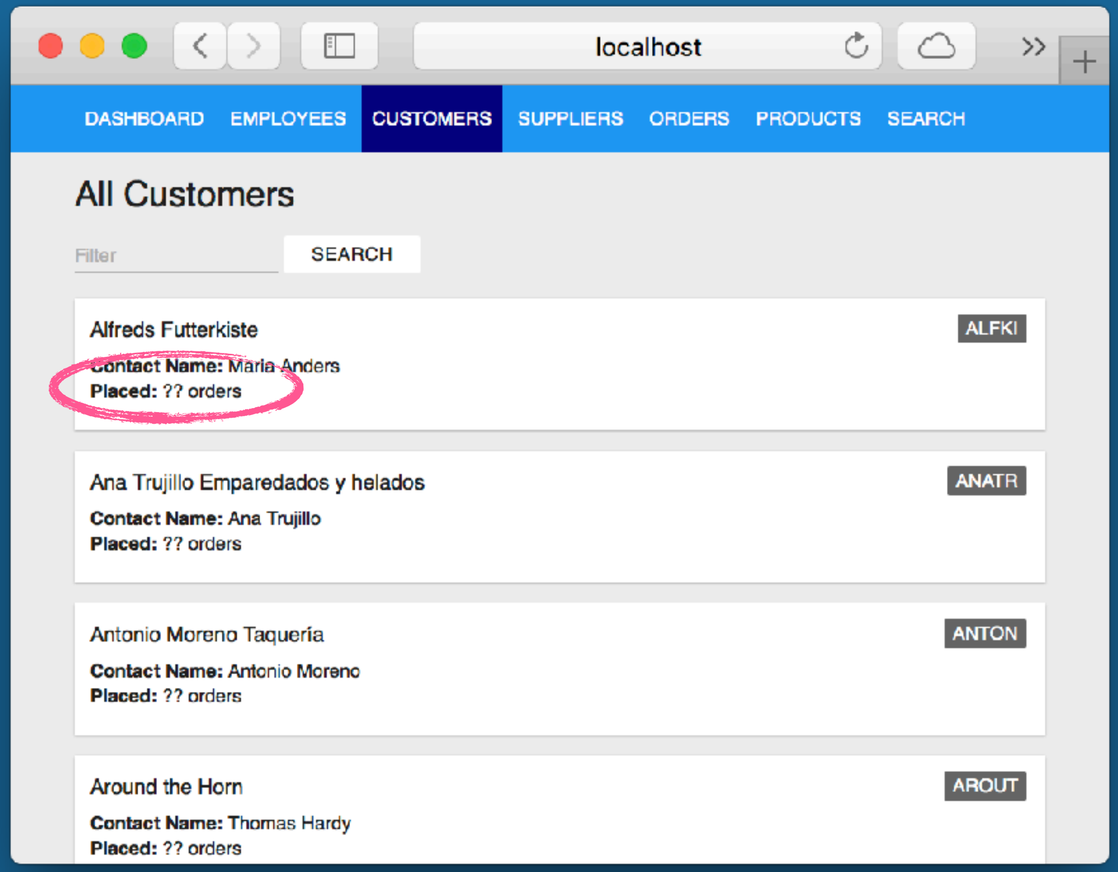
# Aggregate Functions

5

- Customer: `ordercount`

./src/data/customers.js

```
getAllCustomers();
```




```
npm run test:ex:watch 5
```

# Creating Records

- Add new records to a table via an `INSERT INTO` statement, indicating the table to insert into
- This must be followed by a `VALUES` expression, with an array containing the corresponding data, in the order aligned with the table's columns

```
INSERT INTO Customer
VALUES (8424, 'Tanner Thompson', 'Likes scotch');
```

# Creating Records

- Sometimes, columns can have default (or other auto-generated) values and we need only provide data for SOME columns.
- Specify the column names in your `INSERT INTO` statement, and ensure your `VALUES` array is in the respective order.

```
INSERT INTO Customer (name, notes)
VALUES ('Tanner Thompson', 'Likes scotch');
```

# SQL Injection

- Anytime new data is added to your database, it must be appropriately sanitized

```
INSERT INTO Customer (name, notes)
VALUES ('Tanner                          ', 'Tanner notes')
```

Name `Tanner'); DROP TABLE Customer; --`

- Most of the time your SQL driver (library) helps you out

```
db.exec(`INSERT INTO Customer (name, notes)
VALUES ($1, $2)`, ['Tanner', 'Tanner notes']);
```

Sanitized before insertion

# Deleting Records

- Delete a table completely, including column definitions, indices, etc...

```
DROP TABLE Customer;
```

- Delete all records from a table, while leaving the structure and indices intact

```
DELETE FROM Customer;
```

- One or more rows can be deleted while leaving the rest intact, via a `WHERE` clause

```
DELETE FROM Customer
WHERE name = 'Juicero';
```

# Our Project

- The SQL tagged template literal can be used to syntax highlight

```
let query = sql`SELECT * FROM Employee`;
```

- Get a database client via `getDb()`

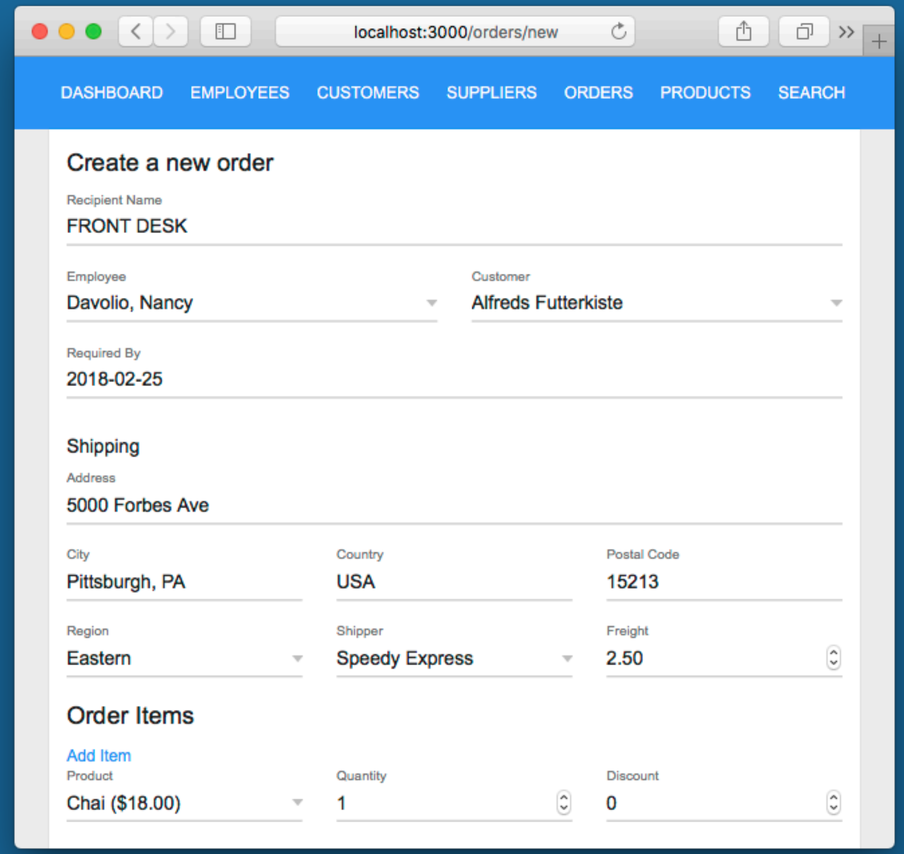
```
import { getDb } from '../db/utils';

let db = await getDb();
// Retrieve a collection of records
let allEmployees = await db.all('SELECT * FROM Employee');
// Retrieve a single record
let product71 = await db.get('SELECT * FROM Product WHERE id = $1', 71);
// Execute a statement, and return the last inserted ID (if applicable)
let { id } = await db.run('INSERT INTO Customer VALUES(...)');
```

# Creating and Destroying

6

- The ❌ button on each row of the order list ultimately calls `deleteOrder`.
- The "New" button on the order page ultimately calls `createOrder`.
- Implement the queries in these two functions so that orders are appropriately created and removed from the database.




./src/data/orders.js

```
deleteOrder(12141);
```

```
npm run test:ex:watch 6
```

# Creating and Destroying

6

./src/data/orders.js

```
createOrder(
  {
    employeeid: 3,
    customerid: 'ALFKI',
    shipcity: 'Minneapolis, MN',
    shipaddress: '60 South 6th St Suite 3625',
    shipname: 'Frontend Masters',
    shipvia: 1,
    shipregion: 1,
    shipcountry: 'USA',
    shippostalcode: '455402',
    requireddate: '2018-03-22T23:38:08.410Z',
    freight: 2.17
  },
  [ { productid: 17, unitprice: 4.11, quantity: 4, discount: 0 },
    { productid: 11, unitprice: 3.37, quantity: 1, discount: 0.10 }]
);
```



npm run test:ex:watch 6

# Transactions

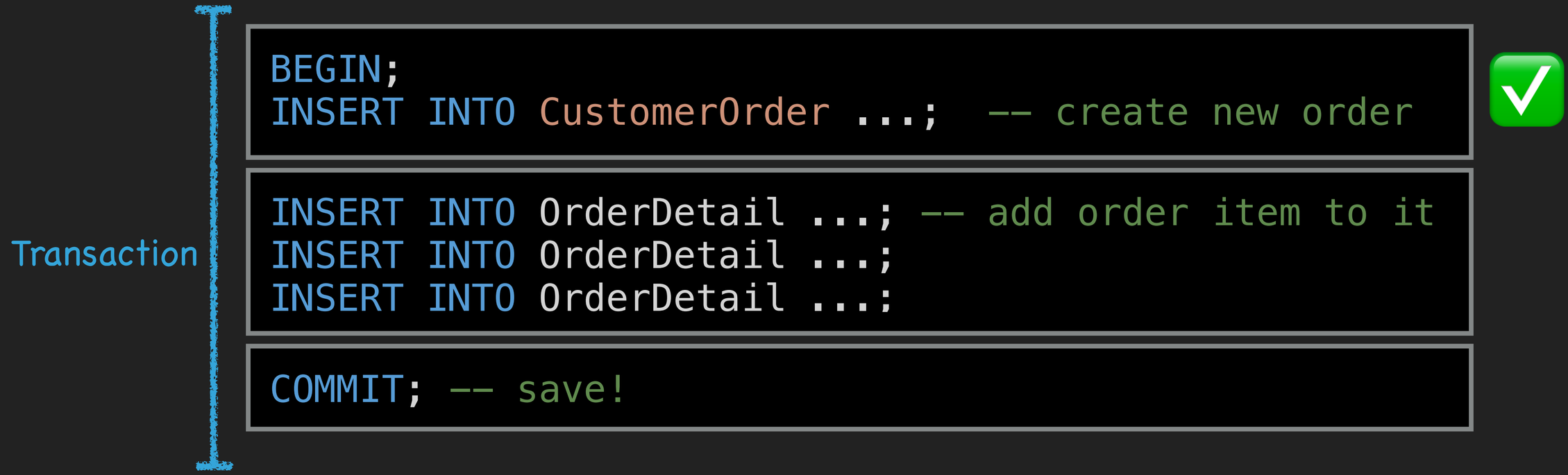
- Sometimes we need to perform multi-statement operations (i.e., create an `Order` and several `OrderDetail` items)
- What would happen if the `Order` was created, but something went wrong with creating one of the `OrderDetails`?

```
INSERT INTO CustomerOrder ...;  -- create new order
```

```
INSERT INTO OrderDetail ...; -- add order item to it
INSERT INTO OrderDetail ...;
INSERT INTO OrderDetail ...;
```

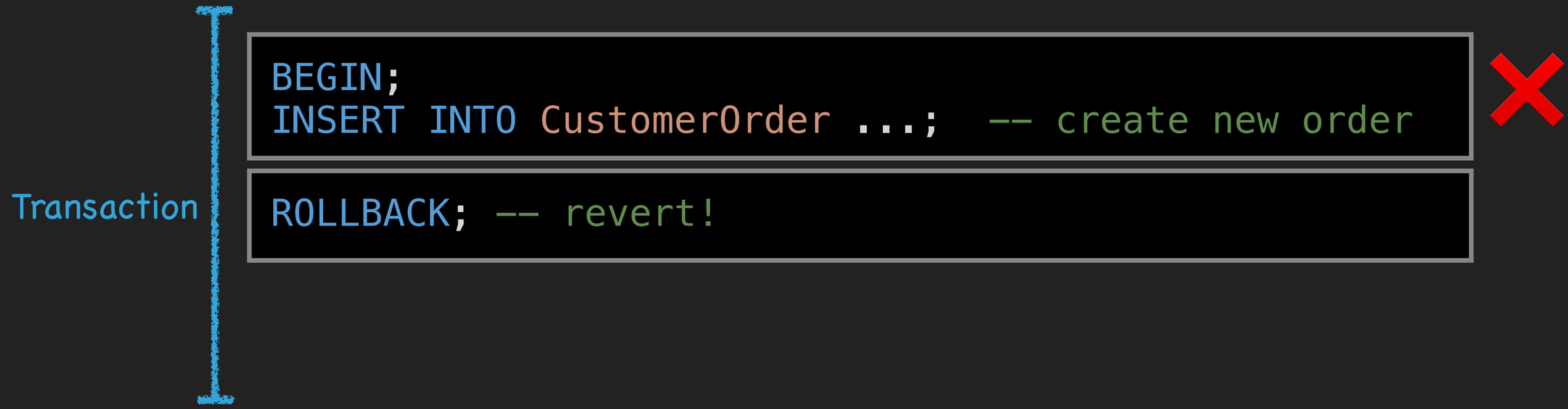
# Transactions

- Surround multiple statements with `BEGIN` and `COMMIT` to include them in a transaction



# Transactions

- Surround multiple statements with `BEGIN` and `COMMIT` to include them in a transaction



# Transaction - ACID

- Principles that guarantee data consistency, even when things go wrong

✅ Atomic - All of the operations are treated as a single "all or nothing" unit

✅ Consistent - Changes from one valid state to another (never halfway)

- Isolated - How visible are particulars of a transaction before it's complete? When? To whom?

✅ Durable - Once it completes, the transaction's results are stored permanently in the system

# Transaction Isolation Levels

- The degree to which other queries to the database can see, or be influenced by an in-progress transaction
- There are levels of isolation, and we as developers have a choice
- Often this is a concurrency vs. safety and consistency trade-off
- First described in the ANSI SQL-92 standard

# Read Phenomena: Dirty Reads

ORDER

| id | customerid | amount | month |
|---|---|---|---|
| 19 | 1 | 99.00 | 5 |
| 20 | 1 | 212.00 | 3 |
| 21 | 3 | 14.50 | 3 |

```
BEGIN;
```

```
SELECT month FROM CustomerOrder WHERE id = 21;
```

3

```
BEGIN;
```

```
UPDATE CustomerOrder SET month=4 WHERE id = 21;
```

```
SELECT month FROM CustomerOrder WHERE id = 21;
```

4

```
ROLLBACK;
```

# Read Phenomena: Dirty Reads

ORDER

| id | customerid | amount | month |
|---|---|---|---|
| 19 | 1 | 99.00 | 5 |
| 20 | 1 | 212.00 | 3 |
| 21 | 3 | 14.50 | 3 |

```
BEGIN;
```

```
SELECT month FROM CustomerOrder WHERE id = 21;
```

3

```
BEGIN;
```

```
UPDATE CustomerOrder SET month=4 WHERE id = 21;
```

```
SELECT month FROM CustomerOrder WHERE id = 21;
```

4

```
ROLLBACK;
```

```
COMMIT;
```

# Transaction Isolation Levels

- Read Uncommitted - Dirty reads are possible.
- Read Committed - Write locks are obtained across the whole transaction, read locks are obtained only for the duration of a SELECT. Dirty reads are not possible

# Read Phenomena: Non-Repeatable Reads

ORDER

| id | customerid | amount | month |
|---|---|---|---|
| 19 | 1 | 99.00 | 5 |
| 20 | 1 | 212.00 | 3 |
| 21 | 3 | 14.50 | 3 |

```
BEGIN;
```

```
SELECT month FROM CustomerOrder WHERE id = 21;
```

3

```
BEGIN;
```

```
UPDATE CustomerOrder SET month=4 WHERE id = 21;
COMMIT;
```

```
SELECT month FROM CustomerOrder WHERE id = 21;
```

4

# Read Phenomena: Non-Repeatable Reads

ORDER

| id | customerid | amount | month |
|---|---|---|---|
| 19 | 1 | 99.00 | 5 |
| 20 | 1 | 212.00 | 3 |
| 21 | 3 | 14.50 | 3 |

```
BEGIN;
```

```
SELECT month FROM CustomerOrder WHERE id = 21;
```

3

```
BEGIN;
```

```
UPDATE CustomerOrder SET month=4 WHERE id = 21;
COMMIT;
```

```
SELECT month FROM CustomerOrder WHERE id = 21;
```

4

# Transaction Isolation Levels

- Read Uncommitted - Dirty reads are possible. Sequence of writes is guaranteed
- Read Committed - Write locks are obtained across the whole transaction, read locks are obtained only for the duration of a SELECT. Dirty reads are not possible
- Repeatable Read - Read and write locks are obtained across the whole transaction.

# Read Phenomena: Phantom Reads

ORDER

| id | customerid | amount | month |
|---|---|---|---|
| 19 | 1 | 99.00 | 5 |
| 20 | 1 | 212.00 | 3 |
| 21 | 3 | 14.50 | 3 |

```
BEGIN;
```

```
SELECT sum(amount) FROM CustomerOrder
WHERE month BETWEEN 19 AND 23;
```

226.50

```
BEGIN;
```

```
INSERT INTO CustomerOrder(customer_id, amount, month)
                    VALUES (22, 40000, 3);
                    COMMIT;
```

```
SELECT sum(amount) FROM CustomerOrder
WHERE month BETWEEN 19 AND 23;
COMMIT;
```

40226.50

# Read Phenomena: Phantom Reads

ORDER

| id | customerid | amount | month |
|---|---|---|---|
| 19 | 1 | 99.00 | 5 |
| 20 | 1 | 212.00 | 3 |
| 21 | 3 | 14.50 | 3 |

```
BEGIN;
```

```
SELECT sum(amount) FROM CustomerOrder
WHERE month BETWEEN 19 AND 23;
```

226.50

```
BEGIN;
```

```
INSERT INTO CustomerOrder(customer_id, amount, month)
                    VALUES (22, 40000, 3);
                    COMMIT;
```

```
SELECT sum(amount) FROM CustomerOrder
WHERE month BETWEEN 19 AND 23;
COMMIT;
```

40226.50

# Transaction Isolation Levels

- Read Uncommitted - Dirty reads are possible. Sequence of writes is guaranteed
- Read Committed - Write locks are obtained across the whole transaction, read locks are obtained only for the duration of a SELECT. Dirty reads are not possible
- Repeatable Read - Read and write locks are obtained across the whole transaction.
- Serializable - Read and write locks are obtained on selected data, range locks are obtained for WHERE clauses.

# Transaction Isolation Levels

- Read Uncommitted - Dirty reads are possible. Sequence of writes is guaranteed
- Read Committed - Write locks are obtained across the whole transaction, read locks are obtained only for the duration of a SELECT. Dirty reads are not possible
- Repeatable Read - Read and write locks are obtained across the whole transaction.
- Serializable - Read and write locks are obtained on selected data, range locks are obtained for WHERE clauses.
- Externally Consistent - See: Google Spanner

# Databases & Storage Engines

- PostgreSQL - MVCC storage engine, and possible changes in v12.
- MySQL
  - InnoDB - supports transactions, and is the best "default choice"
  - MyISAM - no transactions, faster and smaller in some cases
  - Others...
- SQLite

# Transactions

7

- Creating an order involves multiple queries. If some succeed and some fail, we'd be left in an inconsistent state!
- Use a transaction to make this multi-step operation "all or nothing", using best practices to handle errors appropriately
- INSERT INTO must be a statement by its self
- Use try/catch



```
npm run test:ex:watch 7
```

# Updating Records

- One or more records may be updated at a time

```
SELECT id, unitprice, quantity, discount
FROM OrderDetail
WHERE orderid = 10793
```

| id | unitprice | quantity | discount |
|---|---|---|---|
| 10793/41 | 9.65 | 14 | 0 |
| 10793/52 | 7 | 8 | 0 |

```
UPDATE OrderDetail
SET discount = 0.15
WHERE orderid = 10793
```

```
SELECT id, unitprice, quantity, discount
FROM OrderDetail
WHERE orderid = 10793
```

| id | unitprice | quantity | discount |
|---|---|---|---|
| 10793/41 | 9.65 | 14 | 0.15 |
| 10793/52 | 7 | 8 | 0.15 |

# Updating Records

- Sometimes we want to use current values to calculate new ones

```
UPDATE OrderDetail
SET discount = 0.15,
    unitprice = (1 - 0.15) * unitprice
WHERE orderid = 10793
```

```
SELECT id, unitprice, quantity, discount
FROM OrderDetail
WHERE orderid = 10793
```

| id | unitprice | quantity | discount |
|---|---|---|---|
| 10793/41 | 5.037360 | 14 | 0.15 |
| 10793/52 | 3.654044 | 8 | 0.15 |

# Updating Records

Can be re-applied without further altering the value

- Often we want an update to be idempotent

```
UPDATE OrderDetail
SET discount = 0.15,
    unitprice = (1 - 0.15) * ( SELECT unitprice
                               FROM Product
                               WHERE id = OrderDetail.productid )
WHERE orderid = 10793
```

×2

```
SELECT id, unitprice, quantity, discount
FROM OrderDetail
WHERE orderid = 10793
```

| id | unitprice | quantity | discount |
|---|---|---|---|
| 10793/41 | 6.972125 | 14 | 0.15 |
| 10793/52 | 5.0575 | 8 | 0.15 |

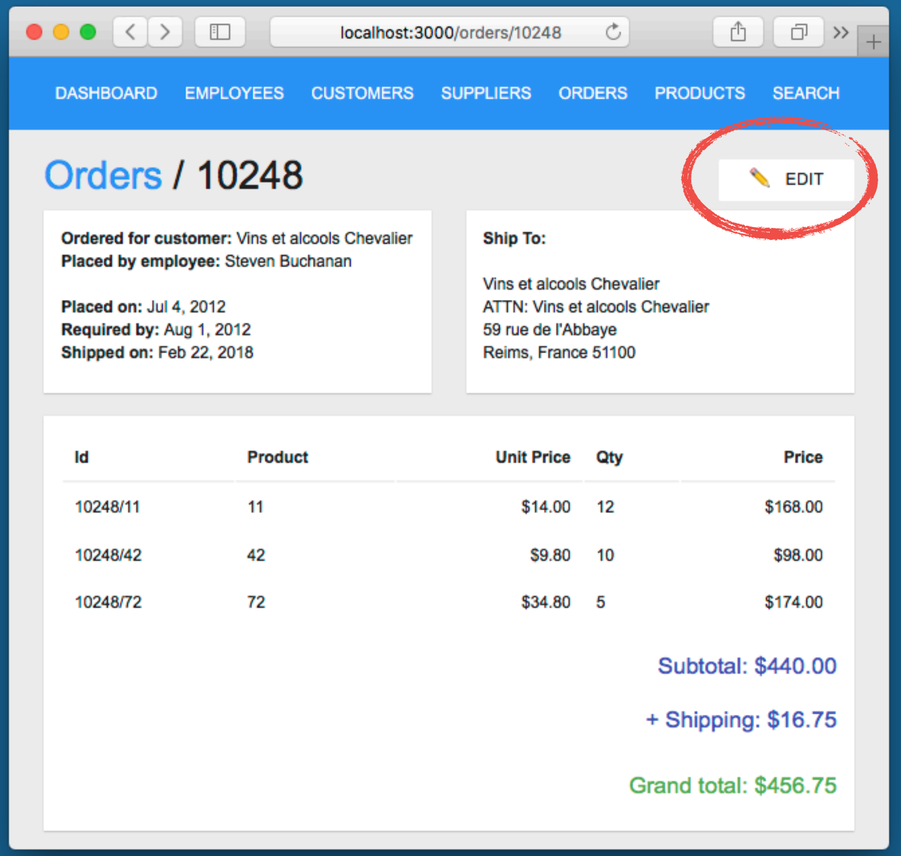
# Insert or Update

- Many RDBMS's allow for some sort of "Insert or Update" command
- Widely different implementations and usage — definitely not standard SQL
- PostgreSQL (10.x) is the most flexible
- SQLite (3.22.x) only allows "Insert or Replace"
- MySQL (4.1) "On conflict"

# Updating Records

8

- Editing an order ultimately calls `updateOrder`.
- Implement the query, using a transaction, such that changes to the order and all order items are persisted appropriately




```
npm run test:ex:watch 8
```

# Updating Records

8

./src/data/orders.js

```
updateOrder(12345, {
    employeeid: 3,
    customerid: 'ALFKI',
    shipcity: 'Minneapolis, MN',
    shipaddress: '60 South 6th St Suite 3625',
    shipname: 'Frontend Masters',
    shipvia: 1,
    shipregion: 1,
    shipcountry: 'USA',
    shippostalcode: '455402',
    requireddate: '2018-03-22T23:38:08.410Z',
    freight: 2.17
  },
  [ { id: '12345/1', productid: 17, unitprice: 4.33, quantity: 4, discount: 0 },
    { id: '12345/2', productid: 11, unitprice: 2.18, quantity: 1, discount: 0.1 }
]
);
```



```
npm run test:ex:watch 8
```

# Changing our schema

- We usually have at least one DB per environment (dev, test, staging, prod)
- Apps that connect to a DB have a hard dependency on a particular schema
- Having reproducible versions includes the database!
- A happy state: a given git commit (or set of commits) = reproducible version

# Migrations

- Incremental changes to a database
- Ideally, reversible
- May start with an initial "seed" script
- "Applied" migrations are stored in a special DB table
- Ideally, backwards compatible

# Migrations in Node

- A simple and popular solution: db-migrate/node-db-migrate
- Check out your project's `./migrations` folder
- Migrations are named, and prefixed by a timestamp (chronological sorting)
- Can be purely JS based, or JS that runs SQL scripts (recommended)
- Uses your `./database.json` to connect to databases of various types
- Env var `DATABASE_URL` takes precedence

```
DATABASE_URL=postgres://mike:password123@db.example.com:5432/customer_db
```

# Migrations in Node - Creating

- Create a new migration using the CLI tool

```
./node_modules/.bin/db-migrate create MyMigration --sql-file
```

- We have a NPM script set up for our project (will always `--sql-file` and generates per RDBMS migration sql scripts)

```
npm run db:migrate:create MyMigration
```

# Migrations in Node - Running Forward

- Attempt to run all migrations not yet applied to a database

```
./node_modules/.bin/db-migrate -e pg up
```

- We have a NPM script set up for our project

```
npm run db:migrate:pg up
```

```
npm run db:migrate:sqlite up
```

```
npm run db:migrate:mysql up
```

# Migrations in Node - Rolling back

- Roll back one migration on a given database

```
./node_modules/.bin/db-migrate -e pg down
```

- We have a NPM script set up for our project

```
npm run db:migrate:pg down
```

```
npm run db:migrate:sqlite down
```

```
npm run db:migrate:mysql down
```

# Indices - A memory-for-time trade-off

- Extra bookkeeping performed on records for quick retrieval later

**Table: People**

| id | first_name | last_name |
|---|---|---|
| 345 | Rick | Sanchez |
| 346 | Mike | North |
| 347 | Mark | Grabanski |
| 348 | Morty | Smith |
| 349 | Jerry | Smith |
| 350 | Summer | Smith |
| 351 | Beth | Smith |

**Index: People(last_name)**


| value | records |
|---|---|
| Grabasnki | 347 |
| North | 346 |
| Sanchez | 345 |
| Smith | 348, 349, 350, 351 |

# Indices - A space-for-time trade-off

- Try grabbing OrderDetails by orderid

```
SELECT * FROM OrderDetail WHERE orderid = 10793
```

55ms

- We can ask our database to describe the work required to get these results

```
EXPLAIN SELECT * FROM OrderDetail WHERE orderid = 10793
```

```
Gather
 (cost = 1000.00..9426.37 rows = 44 width = 33)
Workers Planned: 2
  -> Parallel Seq Scan on orderdetail
       (cost = 0.00..8421.97 rows = 18 width = 33)
       Filter: (orderid = 10793)
```

# Indices - A space-for-time trade-off

- Let's create an index on this column

```
CREATE INDEX orderdetail_oid ON OrderDetail(orderid)
```

```
SELECT * FROM OrderDetail WHERE orderid = 10793
```

28ms

- We can ask our database to describe the work required to get these results

```
EXPLAIN SELECT * FROM OrderDetail WHERE orderid = 10793
```

```
Index Scan using orderdetail_oid on orderdetail
  (cost = 0.42..9.20 rows = 44 width = 33)
   Index Cond: (orderid = 10793)
```

# Indices - A space-for-time trade-off

- Indices can be created on multiple columns

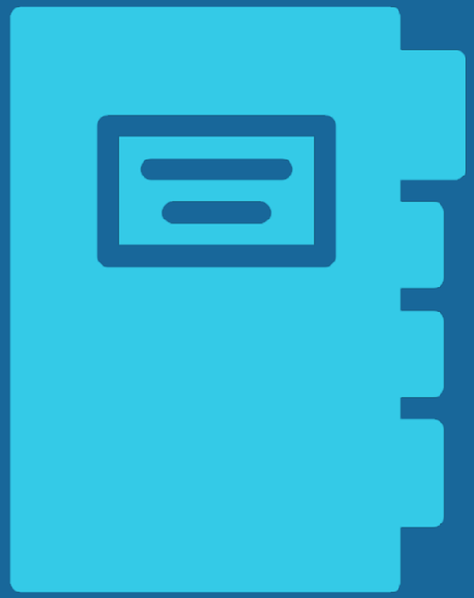
```
CREATE INDEX orderdetail_order_customer
ON OrderDetail(orderid, customerid)
```

- When creating indices, the goal is to reduce (possibly eliminate) the exhaustive searching done for common or perf-sensitive queries

# Migrations and Indices

9

- In Exercise 4, we started combining tables together with JOIN statements, but this has hurt performance
- Identify specific opportunities for adding indices to the database, with the specific goal of reducing database time on
  - The order list
  - The employee list
  - The product list
  - The order page
- Create a new database migration to add these indices to the database



```
npm run test:ex:watch 9
```

# Migrations and Indices

9

- Create a new migration

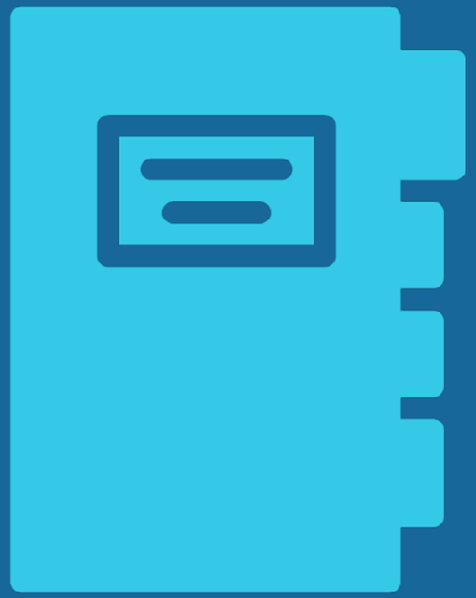
```
npm run db:migrate:create AddEx9Indices
```

- Run the migration forward

```
DB_TYPE=pg npm run db:migrate:up
```

- Roll the migration back

```
DB_TYPE=pg npm run db:migrate:down
```





```
npm run test:ex:watch 9
```

# Column Constraints - Types

- One of the ways we can constrain values placed in particular columns is by types
- Names and properties of these types may vary from system to system
- Choice of column type impacts performance, and options for indexing and filtering

# Column Constraints - Create a Table

- Creating a table gives us the opportunity to define columns as name/type pairs

```
CREATE TABLE UserPreferences (
    id                INT PRIMARY KEY, -- an integer
    favorite_color    CHAR(6), -- Exactly 6 characters
    age               SMALLINT, -- an "small" integer
    birth_date        DATE, -- date (may or may not include time)
    notes             VARCHAR(255), -- up to 255 characters
    is_active         BOOLEAN -- boolean
);
```

# Column Constraints - NOT NULL

- We can forbid null values for one or more columns by adding `NOT NULL`

```
CREATE TABLE UserPreferences (
    id                INT PRIMARY KEY NOT NULL, -- required
    favorite_color    CHAR(6),
    age               SMALLINT,
    birth_date        DATE NOT NULL, -- required
    notes             VARCHAR(255),
    is_active         BOOLEAN NOT NULL -- required
);
```

# Column Constraints - Updating a Table's Definition

- Most RDBMS's allow us to alter tables and column definitions

# Column Constraints - UNIQUE

- A uniqueness constraint may be added to a column via `UNIQUE`

```
CREATE TABLE UserAccount (
    email VARCHAR(255) UNIQUE NOT NULL,
    name VARCHAR(255)
);
```

```
CREATE UNIQUE INDEX u_email ON UserAccount(email);
```

# Column Constraints - PRIMARY KEY

- A combination of a `NOT NULL` and `UNIQUE` constraint

```
CREATE TABLE UserAccount (
    email VARCHAR(255) PRIMARY KEY,
    name VARCHAR(255)
);
```

- Often you'll have one identity column in each that serves as the "id" for each addressable record

# Column Constraints - PRIMARY KEY

- Often we want these IDs to auto-increment

# Column Constraints - PRIMARY KEY

- Often we want these IDs to auto-increment
- In MySQL we have to specify `AUTO_INCREMENT` or we'll get an error

MySQL

```
CREATE TABLE UserAccount (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255)
);
```

```
INSERT INTO UserAccount(name)
VALUES('foo')
```

✅ `d 'id' doesn't have a ult value`

# Column Constraints - PRIMARY KEY

- In PostgreSQL we'll create a `SEQUENCE` as our source of incrementing values

Postgres

```
CREATE SEQUENCE UserAccount_id_seq;
CREATE TABLE UserAccount (
    id INTEGER PRIMARY KEY DEFAULT nextval('UserAccount_id_seq'),
    name VARCHAR(255)
);
ALTER SEQUENCE UserAccount_id_seq OWNED BY UserAccount.id;
```

```
INSERT INTO UserAccount(name) VALUES('foo')
```

# Column Constraints - PRIMARY KEY

- In PostgreSQL we'll create a `SEQUENCE` as our source of incrementing values
- The `SERIAL` type does this for us automatically!

Postgres

```
CREATE TABLE UserAccount (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255)
);
```

```
INSERT INTO UserAccount(name) VALUES('foo')
```

# Column Constraints - PRIMARY KEY

- SQLite has an implicit rowid column, unless we explicitly opt out
- We run into id reuse problems when we delete rows and add new ones

```
CREATE TABLE UserAccount (name TEXT);
INSERT INTO UserAccount(name)
  VALUES ("foo"), ("bar"), (“baz”);
DELETE FROM UserAccount
  WHERE name="baz";
INSERT INTO UserAccount(name)
  VALUES ("new1"), ("new2")
SELECT rowid, name FROM UserAccount
```

SQLite

| rowid | name |
|---|---|
| 1 | foo |
| 2 | bar |
| ~~3~~ | ~~baz~~ |
| 3 | new1 |
| 4 | new2 |

# Column Constraints - PRIMARY KEY

- Even if we create an id column and use PRIMARY KEY, we'll still have the same problem. It's just an alias for rowid!

```
CREATE TABLE UserAccount (
  id INTEGER PRIMARY KEY,
  name TEXT  );
INSERT INTO UserAccount(name)
  VALUES ("foo"), ("bar"), ("baz");
DELETE FROM UserAccount
  WHERE name="baz";
INSERT INTO UserAccount(name)
  VALUES ("new1"), ("new2")
SELECT * FROM UserAccount
```

SQLite

| id | name |
|---|---|
| 1 | foo |
| 2 | bar |
| ~~3~~ | ~~baz~~ |
| 3 | new1 |
| 4 | new2 |

# Column Constraints - PRIMARY KEY

- Only by using AUTOINCREMENT can we avoid reusing ids

```
CREATE TABLE UserAccount (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT  );
INSERT INTO UserAccount(name)
  VALUES ("foo"), ("bar"), (“baz”);
DELETE FROM UserAccount
  WHERE name="baz";
INSERT INTO UserAccount(name)
  VALUES ("new1"), ("new2")
SELECT * FROM UserAccount
```

SQLite

| id | name |
|---|---|
| 1 | foo |
| 2 | bar |
| ~~3~~ | ~~baz~~ |
| 4 | new1 |
| 5 | new2 |

# Column Constraints - Foreign Keys

- Used to enforce the "other end" of a relationship
- Validates against a column in the "foreign" table

```
CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255)
);

CREATE TABLE CustomerOrder (
    id SERIAL PRIMARY KEY,
    customerid INTEGER,
    amount REAL,
    month INTEGER DEFAULT 5 NOT NULL
);
```

**ORDER**

| id | customerid | amount |
|---|---|---|
| 1 | 1 | 99.00 |
| 2 | 1 | 212.00 |
| 3 | 3 | 14.50 |
| 4 | 12 | 1000.00 |

**CUSTOMER**

| id | name |
|---|---|
| 1 | Mike North |
| 2 | Mark Grabanski |
| 3 | Lisa Huang-North |

```
INSERT INTO CustomerOrder
    (customerid, amount, month)
    VALUES (99, 300.50, 3);
```

Click for SQLfiddle

# Column Constraints - Foreign Keys

- Used to enforce the "other end" of a relationship
- Validates against a column in the "foreign" table

ORDER

| id | customerid | amount |
|---|---|---|
| 1 | 1 | 99.00 |
| 2 | 1 | 212.00 |
| 3 | 3 | 14.50 |
| 4 | 12 | 1000.00 |

```
CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255)
);
CREATE TABLE CustomerOrder (
    id SERIAL PRIMARY KEY,
    customerid INTEGER
        NOT NULL REFERENCES customer(id),
    amount REAL,
    month INTEGER DEFAULT 5 NOT NULL
);
```

```
ERROR: insert or update on
table CustomerOrder violates
foreign key constraint
"order_customerid_fkey"
Detail: Key (customerid)=(99)
is not present in table
"customer".
```

```
INSERT INTO CustomerOrder
    (customerid, amount, month)
    VALUES (99, 300.50, 3);
```

Click for SQLfiddle

# Column Constraints

10

- To improve data integrity, add database constraints making it impossible to have multiple OrderDetail records referring to the same orderid and productid
- Create a new table called CustomerOrderTransaction with columns
  - **id** - an auto-incrementing primary key
  - **authorization** - of type string, 255 characters or less, cannot be null
  - **orderid** - of type Integer, cannot be null, must refer to an actual id in the CustomerOrder table

```
npm run test:ex:watch 10
```