SQL

Frontend Masters
March 28, 2018
Mike North

MIKE
WORKS

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.

Today's MySQL/PG Compared to 5y ago

- A **lot** more capable
- "Exotic" and risky features are ready for prime time
- "Obsoleting" other system components
- Customization is common

Prerequisites

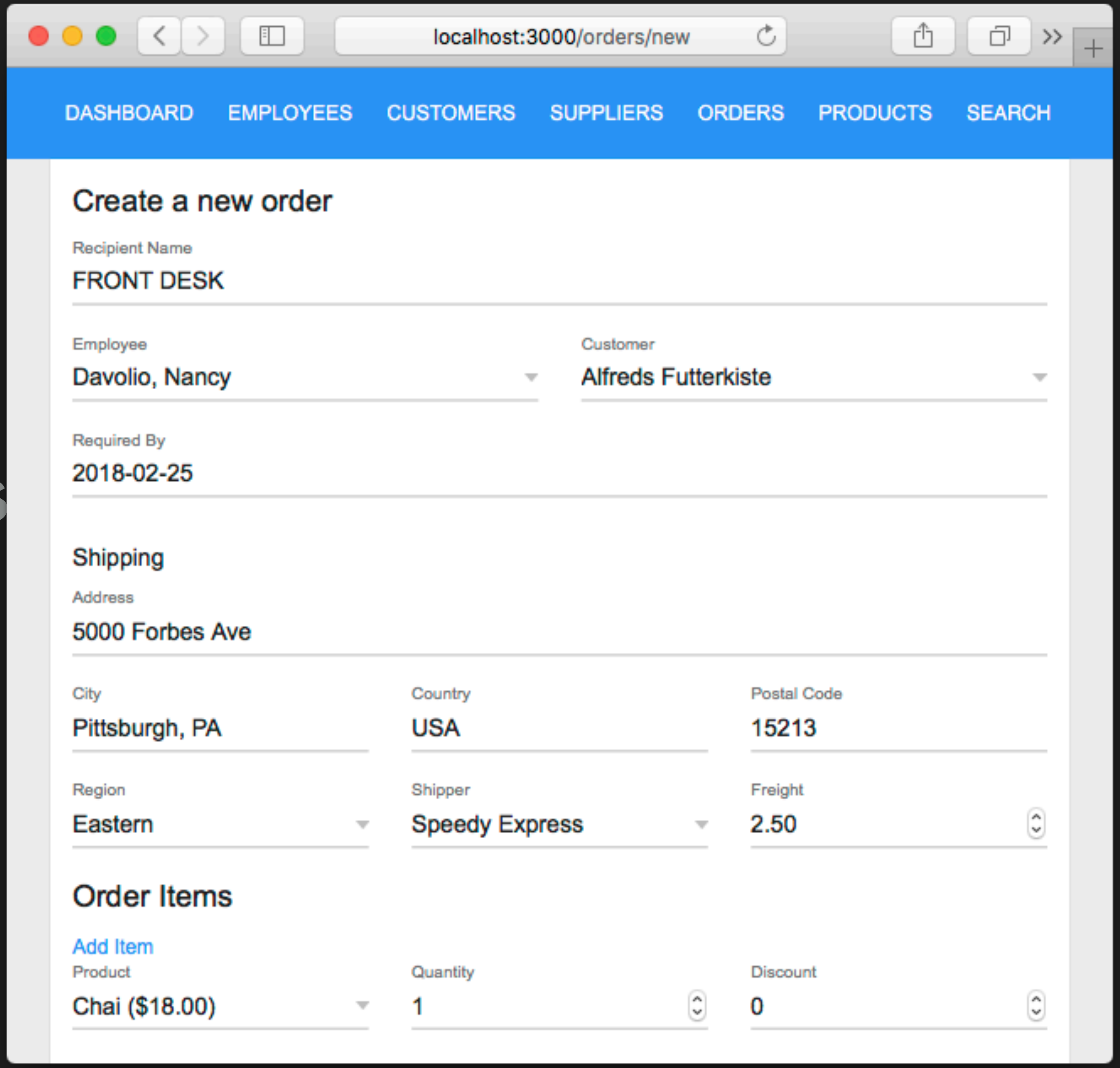
- Relational Algebra and SQL foundations
- Basic SELECT
- Filtering results with WHERE
- Sorting and paginating
- JOINs
- Aggregate functions and GROUP BY
- Transactions
- Creating/Deleting/Updating Records
- Migrations
- Indices
- Types & Column Constraints

Professional SQL: Agenda

- Triggers & Stored Procedures
- Prepared Statements
- Views (Materialized and regular)
- JSON and Array Columns
- Full Text Search
- PubSub
- Database Maintenance & Optimization

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



Create a new order

Recipient Name
FRONT DESK

Employee
Davolio, Nancy

Customer
Alfreds Futterkiste

Required By
2018-02-25

Shipping

Address
5000 Forbes Ave

City
Pittsburgh, PA

Country
USA

Postal Code
15213

Region
Eastern

Shipper
Speedy Express

Freight
2.50

Order Items

Product
Chai ($18.00)

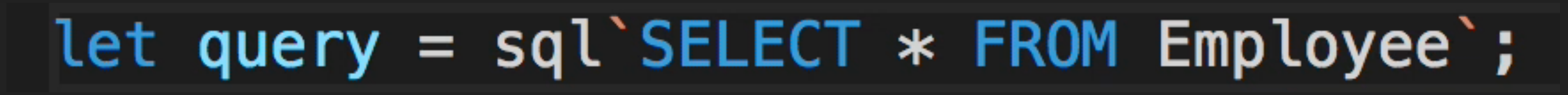
Quantity
1

Discount
0

Professional SQL Begins With Branch: `femasters/begin-pro`

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
```

👀

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
```

🏁

Triggers

- A function stored in the database, bound to the table it operates on
- Executed either before or after several specific events (Insert, Update, Delete, etc…)
- Can run on a per-statement or per-row basis
- ⚠️ May be mysterious if database consumers don't know a trigger exists

MySQL Triggers

- Kind of looks like a transaction with "kick off" instructions

MySQL

```
CREATE TRIGGER LogPermissionsChange
    AFTER UPDATE ON User
    FOR EACH ROW
BEGIN
    IF OLD.permissions != NEW.permissions THEN
        INSERT INTO UserPermissionsLog (userid, before, after,
changed_at) VALUES (OLD.id, OLD.permissions, NEW.permissions,
NOW());
    END IF;
END;
```

```
CREATE TRIGGER LogPermissionsChange
    AFTER UPDATE ON User
    FOR EACH ROW
BEGIN
    IF OLD.permissions != NEW.permissions THEN
        INSERT INTO UserPermissionsLog (userid, before, after,
changed_at) VALUES (OLD.id, OLD.permissions, NEW.permissions,
NOW());
    END IF;
END;
```

```
CREATE TRIGGER LogPermissionsChange
    AFTER UPDATE ON User
    FOR EACH ROW
```

```
BEGIN
    IF OLD.permissions != NEW.permissions THEN
        INSERT INTO UserPermissionsLog (userid, before, after,
changed_at) VALUES (OLD.id, OLD.permissions, NEW.permissions,
NOW());
    END IF;
END;
```

```
CREATE TRIGGER LogPermissionsChange
    AFTER UPDATE ON User
    FOR EACH ROW EXECUTE PROCEDURE log_user_permissions();
```

```
CREATE FUNCTION log_user_permissions()
  RETURNS trigger AS
$$
BEGIN
    IF OLD.permissions != NEW.permissions THEN
        INSERT INTO UserPermissionsLog (userid, before, after,
changed_at) VALUES (OLD.id, OLD.permissions, NEW.permissions,
NOW());
    END IF;
END;
$$
LANGUAGE 'plpgsql';
```

PostgreSQL Triggers

- Make use of **stored procedures** (functions) for use across several triggers

pg

```
CREATE FUNCTION log_user_permissions()
  RETURNS trigger AS
$$
BEGIN
    IF OLD.permissions != NEW.permissions THEN
        INSERT INTO UserPermissionsLog (userid, before, after,
changed_at) VALUES (OLD.id, OLD.permissions, NEW.permissions, NOW());
     END IF;
     RETURN NEW;
END;
$$
```

PostgreSQL Triggers

- Now that we have a stored procedure, we can use it across many triggers

pg

```
CREATE TRIGGER LogPermissionsChange
    AFTER UPDATE ON User
    FOR EACH ROW EXECUTE PROCEDURE log_user_permissions();
```

pg

```
CREATE TRIGGER LogPermissionsChange
    AFTER INSERT ON User
    FOR EACH ROW EXECUTE PROCEDURE log_user_permissions();
```

PostgreSQL Stored Procedures

- Stored procedures can be used for way more than just triggers

pg

```
CREATE FUNCTION addition(a INTEGER, b INTEGER) RETURNS INTEGER AS
$$
BEGIN
   RETURN a + b;
END;
$$
LANGUAGE ‘plpgsql';
```

```
SELECT add_numbers(3, 5) as the_answer;
```

8

Triggers

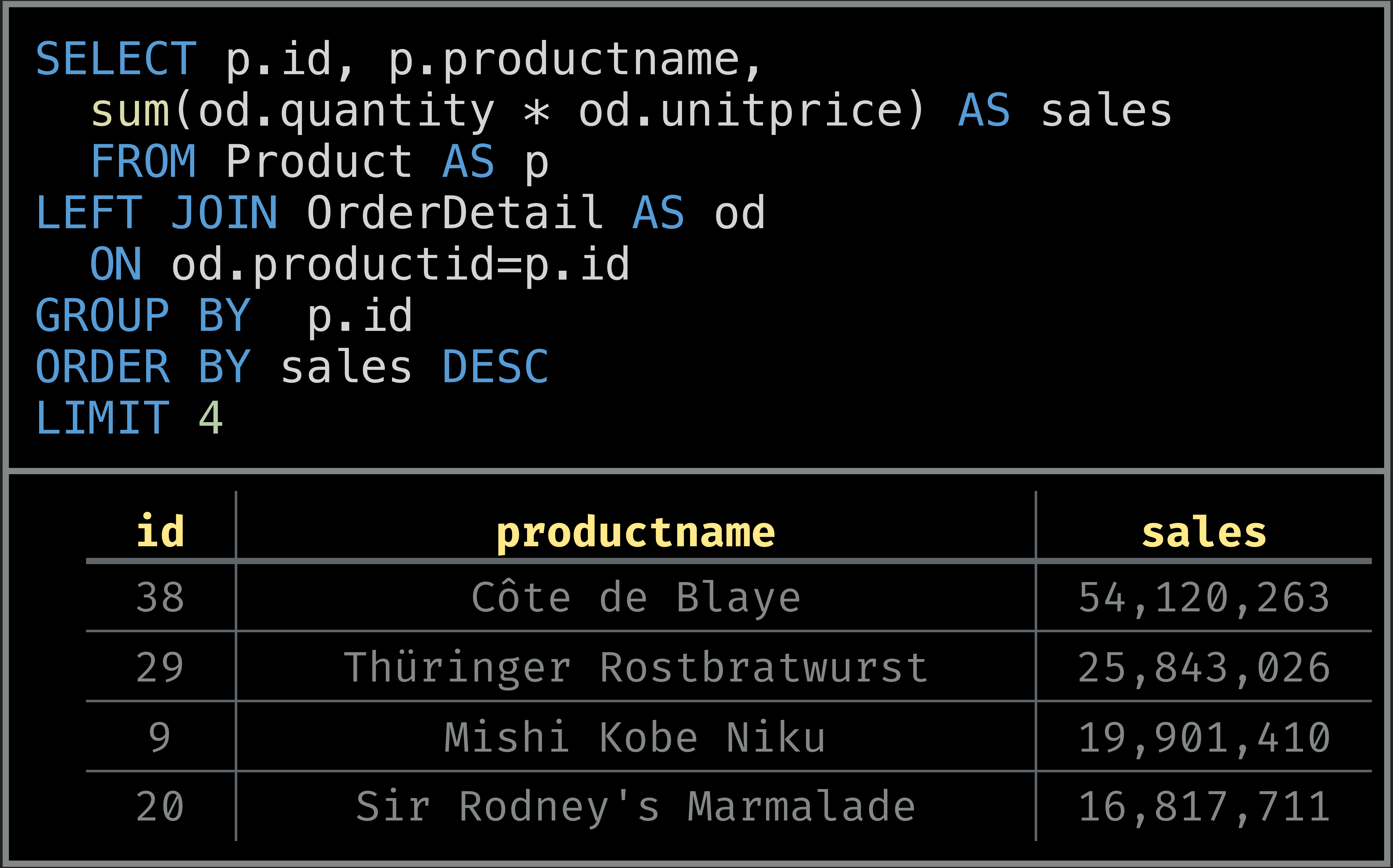
1

- We wish to keep track of Product pricing changes via triggers
- Create a new table called `ProductPricingInfo` w/
  `id` - auto-incrementing primary key
  `fromprice` - decimal
  `toprice` - decimal not null
  `changedate` - date/text not null
  `productid` - integer not null
- Setup triggers (`ProductPricingUpdate` and `ProductPricingInsert`) to log pricing changes in the event new products are created, or product prices are changed.

```
npm run test:ex:watch 11
```

Views

- Sometimes we have SELECT queries that are used over and over
- DRY principle works for databases too!
- Consistency!



```sql
SELECT p.id, p.productname,
  sum(od.quantity * od.unitprice) AS sales
  FROM Product AS p
LEFT JOIN OrderDetail AS od
  ON od.productid=p.id
GROUP BY  p.id
ORDER BY sales DESC
LIMIT 4
```

id	productname	sales
38	Côte de Blaye	54,120,263
29	Thüringer Rostbratwurst	25,843,026
9	Mishi Kobe Niku	19,901,410
20	Sir Rodney's Marmalade	16,817,711

Views

- Named SELECT queries, stored in the DB
- Re-calculated each time you run them
- User access can be restricted

```
CREATE VIEW top_product_sales AS
  SELECT p.id, p.productname,
    sum(od.quantity * od.unitprice) AS sales
    FROM Product AS p
  LEFT JOIN OrderDetail AS od ON od.productid=p.id
  GROUP BY  p.id
  ORDER BY sales DESC LIMIT 4
```

Views

```sql
CREATE VIEW top_product_sales AS
  SELECT p.id, p.productname,
    sum(od.quantity * od.unitprice) AS sales
    FROM Product AS p
  LEFT JOIN OrderDetail AS od ON od.productid=p.id
  GROUP BY  p.id
  ORDER BY sales DESC LIMIT 4
```

```sql
SELECT * FROM top_product_sales
```

id	productname	sales
38	Côte de Blaye	54,120,263
29	Thüringer Rostbratwurst	25,843,026
9	Mishi Kobe Niku	19,901,410
20	Sir Rodney's Marmalade	16,817,711

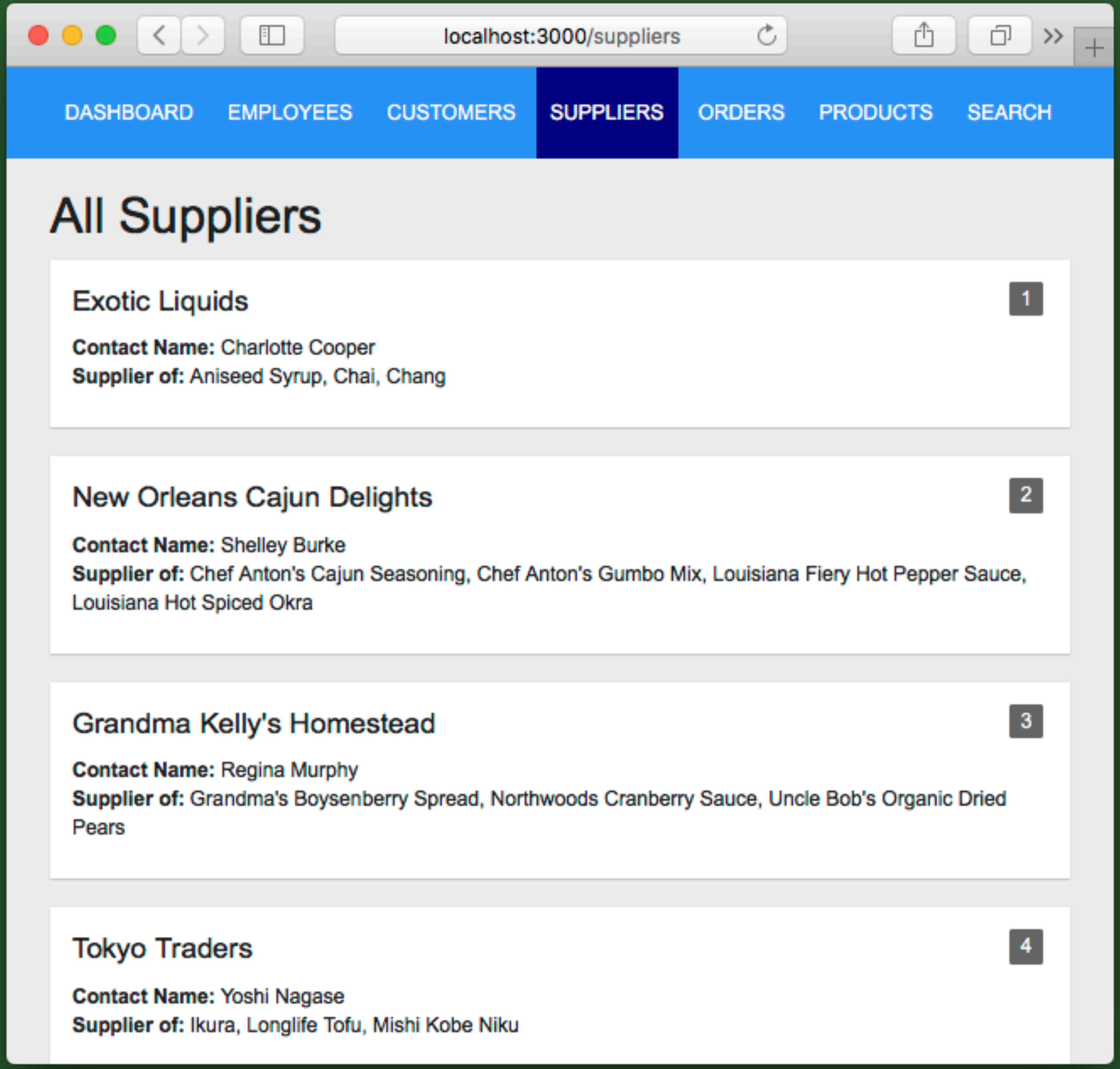
Views

2

- Create a new migration that adds a view SupplierList_V for the supplier list. The query results should remain exactly the same.

./src/data/suppliers.js

```
getAllSuppliers();
```



All Suppliers

Exotic Liquids 1

Contact Name: Charlotte Cooper
Supplier of: Aniseed Syrup, Chai, Chang

New Orleans Cajun Delights 2

Contact Name: Shelley Burke
Supplier of: Chef Anton's Cajun Seasoning, Chef Anton's Gumbo Mix, Louisiana Fiery Hot Pepper Sauce, Louisiana Hot Spiced Okra

Grandma Kelly's Homestead 3

Contact Name: Regina Murphy
Supplier of: Grandma's Boysenberry Spread, Northwoods Cranberry Sauce, Uncle Bob's Organic Dried Pears

Tokyo Traders 4

Contact Name: Yoshi Nagase
Supplier of: Ikura, Longlife Tofu, Mishi Kobe Niku

```
npm run test:ex:watch 12
```

Prepared Statements

- Often DB-driven apps involve many very similar calls to a database
- Each of these has to be parsed as if it's a completely new query!

```
SELECT id, productname from Product WHERE id=1
SELECT id, productname from Product WHERE id=2
SELECT id, productname from Product WHERE id=3
SELECT id, productname from Product WHERE id=4
SELECT id, productname from Product WHERE id=5
SELECT id, productname from Product WHERE id=6
SELECT id, productname from Product WHERE id=7
```

Prepared Statements

- All three of our SQL clients permit us to pre-parse a statement without values
- It’s sent to the database in binary format for direct use, along with values
- 🔐 Security benefit: queries are parsed without involving untrusted values

```
SELECT id, productname from Product WHERE id=?
SELECT id, productname from Product WHERE id=?
SELECT id, productname from Product WHERE id=?
SELECT id, productname from Product WHERE id=?
SELECT id, productname from Product WHERE id=?
SELECT id, productname from Product WHERE id=?
SELECT id, productname from Product WHERE id=?
```

```
[ 1 ]
[ 2 ]
[ 3 ]
[ 4 ]
[ 5 ]
[ 6 ]
[ 7 ]
```

Prepared Statements

- We'll be working with a client-side prepared statement, which must be re-established each time we make a new connection to our databases

./src/db/prepared.ts

```
async function setupPreparedStatements(db) {
  let stmt = await db.prepare('SELECT * FROM Customer');
  return {
    getCustomers: stmt
  };
}
```

```
let customers = await db.statements.getCustomers.all();
```

Prepared Statements

3

- Create a prepared statement in `setupPreparedStatements` for retrieving data about an individual order.

./src/db/prepared.ts

```
setupPreparedStatements(db);
```

- Use this prepared statement in `getOrder` as follows

```
return await db.statements.getOrder.get(id);
```

./src/data/orders.js

```
getOrder();
```

```
npm run test:ex:watch 13
```

Materialized Views

- Like all views, the query that defines it lives in the database
- However, results are not re-calculated on each query
- Mental model (and MySQL hack)
  - temporary read-only table
  - containing the view's result set
  - ...and you have to deliberately refresh it

PostgreSQL - Materialized Views

- Officially supported
- Syntax for creation is nearly identical to non-materialized view

pg

```
CREATE MATERIALIZED VIEW MV_ExampleAccounts AS
  SELECT * FROM UserAccount WHERE lower(email) LIKE '%example.com';
```

pg

```
SELECT email FROM MV_ExampleAccounts;
```

```
       email
-----------------
mike@example.com
-----------------
marc@example.com
```

PostgreSQL - Materialized Views

- Refreshing a materialized view is usually done in a trigger

pg

```
REFRESH MATERIALIZED VIEW MV_ExampleAccounts;
```

MySQL - Materialized Views

- Not officially supported
- However, if we think about what a materialized view really is, we can build our own from scratch
  - Query is stored in the DB
  - Results are cached and can be queried against
  - We can run some SQL to update the cached data

MySQL - Materialized Views

- Step 1: Define a regular view to store the query

MySQL

```
CREATE IF NOT EXISTS VIEW V_CustomerStats AS
  SELECT c.id, round(avg(os.subtotal + os.freight), 2) AS avgOrderSpend,
               round(sum(os.subtotal + os.freight), 2) AS totalOrderSpend,
               round(sum(os.totalDiscount), 2) AS lifetimeDiscount,
               count(os.id) AS numOrders
  FROM Customer AS c
  JOIN (SELECT  o.id,
                o.customerid,
                o.freight,
                sum((od.unitprice * (1 - od.discount)) * od.quantity) AS subtotal,
                sum((od.unitprice * od.discount) * od.quantity) AS totalDiscount
      FROM CustomerOrder AS o
      JOIN OrderDetail AS od
        ON o.id=od.orderid
      GROUP BY o.id) AS os
    ON os.customerid=c.id
  GROUP BY c.id;
```

1730ms

MySQL - Materialized Views

- Step 2: Create a new table based on the view’s result set

MySQL

```
CREATE TABLE  MV_CustomerStats AS SELECT * from V_CustomerStats;
CREATE INDEX MV_CustomerStatsId ON MV_CustomerStats(id);
```

- Step 3: Get the table’s result set

MySQL

```
SELECT * FROM MV_CustomerStats;
```

0.6ms

MySQL - Materialized Views

- Step 4: Refresh the data in the table
- Very important to use a single RENAME query as shown below

MySQL

```
-- Create a new table for the updated result set
CREATE TABLE  MV_CustomerStats_new AS SELECT * from V_CustomerStats;
CREATE INDEX MV_CustomerStatsId ON MV_CustomerStats_new(id);

-- Move the new table into position and the old one out of position
- in the same RENAME query
RENAME TABLE MV_CustomerStats TO MV_CustomerStats_old,
             MV_CustomerStats_new TO MV_CustomerStats;

-- Get rid of the old data
DROP TABLE MV_CustomerStats_old;
```

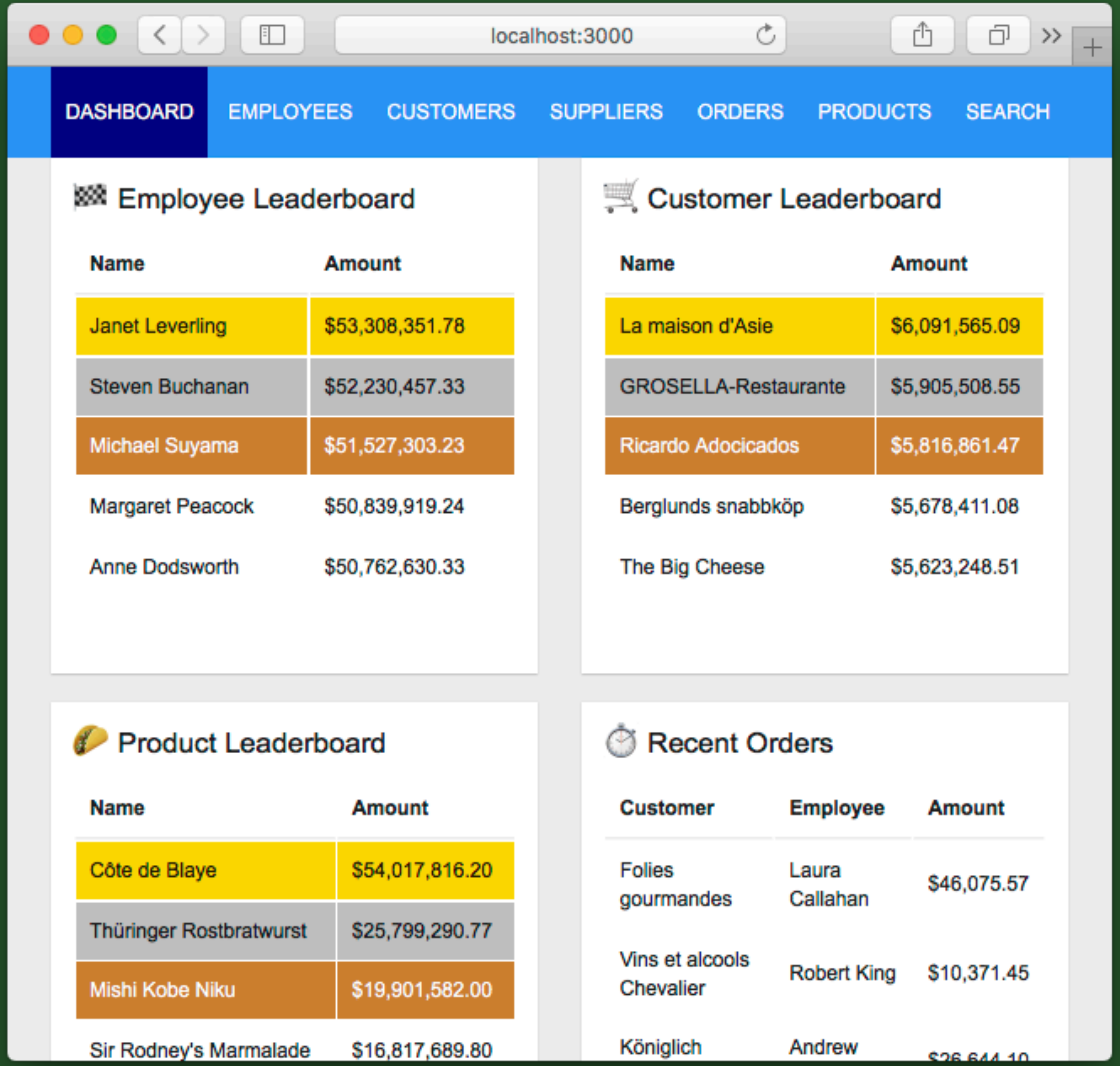
Materialized Views

4

- The database has to do some heavy lifting to obtain the data required for the dashboard page.
- Replace every dashboard query with materialized views
- Ensure that new orders trigger re-calculation of the appropriate materialized views
- Measure the difference in page load time and database time that this improvement makes

./src/data/dashboard.js

```
getEmployeeSalesLeaderboard();
getProductSalesLeaderboard();
getRecentOrders();
getReorderList();
```



Employee Leaderboard

Name	Amount
Janet Leverling	$53,308,351.78
Steven Buchanan	$52,230,457.33
Michael Suyama	$51,527,303.23
Margaret Peacock	$50,839,919.24
Anne Dodsworth	$50,762,630.33

Customer Leaderboard

Name	Amount
La maison d'Asie	$6,091,565.09
GROSELLA-Restaurante	$5,905,508.55
Ricardo Adocicados	$5,816,861.47
Berglunds snabbköp	$5,678,411.08
The Big Cheese	$5,623,248.51

Product Leaderboard

Name	Amount
Côte de Blaye	$54,017,816.20
Thüringer Rostbratwurst	$25,799,290.77
Mishi Kobe Niku	$19,901,582.00
Sir Rodney's Marmalade	$16,817,689.80

Recent Orders

Customer	Employee	Amount
Folies gourmandes	Laura Callahan	$46,075.57
Vins et alcools Chevalier	Robert King	$10,371.45
Königlich	Andrew	$26,644.10

```
npm run test:ex:watch 15
```

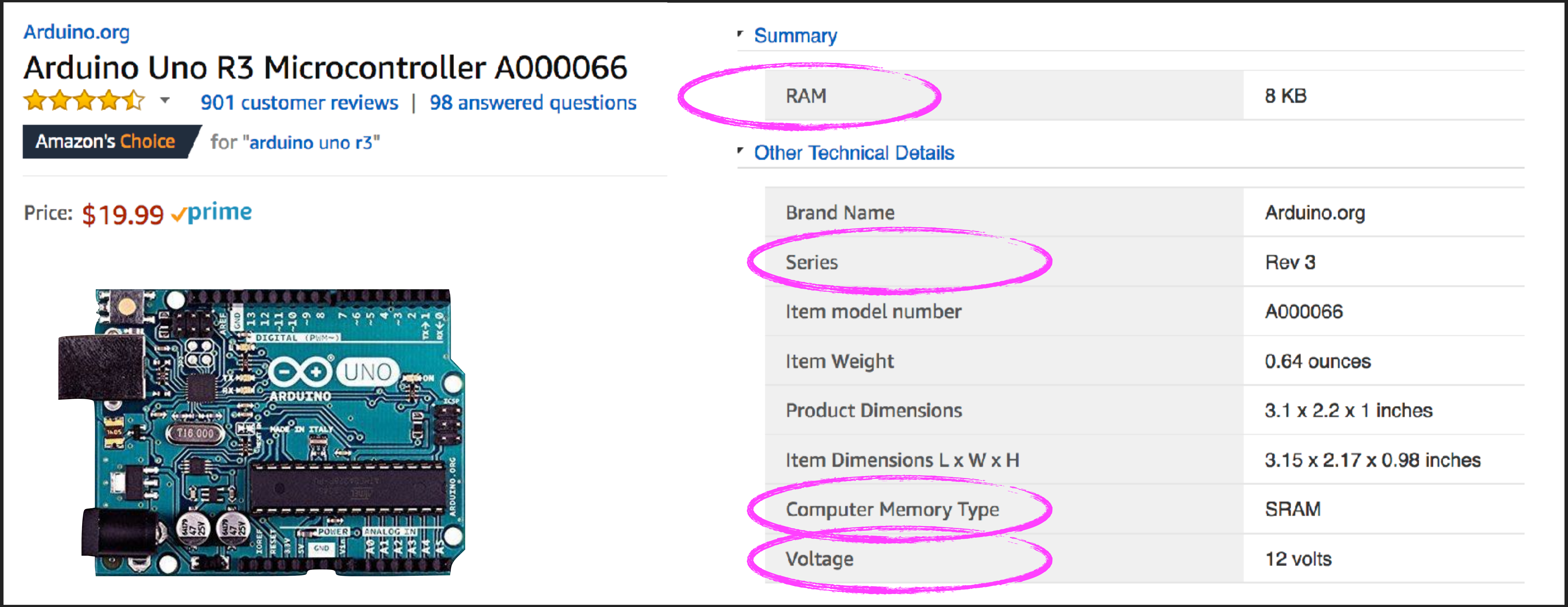
NoSQL

- A very broad category of databases
- Many arose from limitations in relational databases
- Includes things like wide-column stores, key-value stores, document stores
- Often sacrifice consistency in favor of availability, cluster-ability and speed

NoSQL: Document Stores

- Instead of tuples being stored in relations, we have documents in stores
- Common examples: IndexedDB, MongoDB, CouchDB
- What's a document? JSON++ (binary values are usually ok)
- Often a much more efficient way to store sparse data

Case Study: amazon.com items for sale



Arduino.org

Arduino Uno R3 Microcontroller A000066

★★★★☆ 901 customer reviews | 98 answered questions

Amazon's Choice for "arduino uno r3"

Price: $19.99 ✓prime

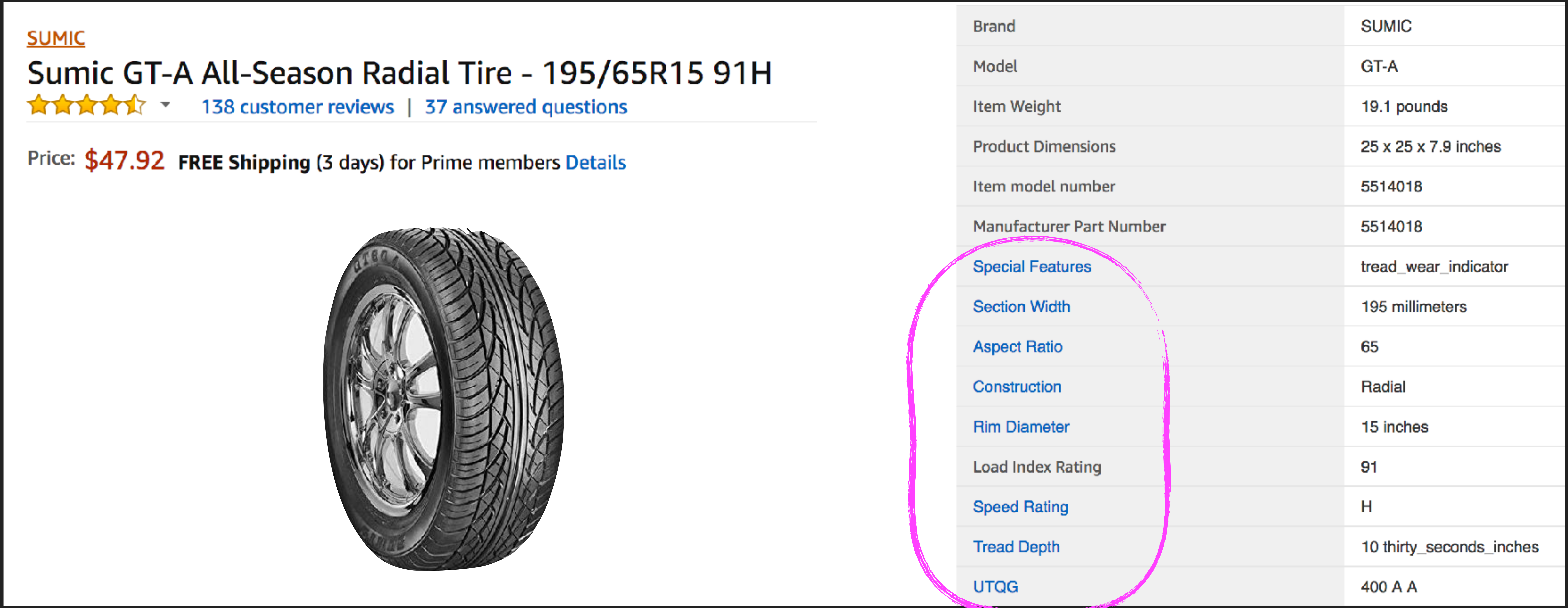
Summary

RAM	8 KB

Other Technical Details

Brand Name	Arduino.org
Series	Rev 3
Item model number	A000066
Item Weight	0.64 ounces
Product Dimensions	3.1 x 2.2 x 1 inches
Item Dimensions L x W x H	3.15 x 2.17 x 0.98 inches
Computer Memory Type	SRAM
Voltage	12 volts

Case Study: amazon.com items for sale



SUMIC

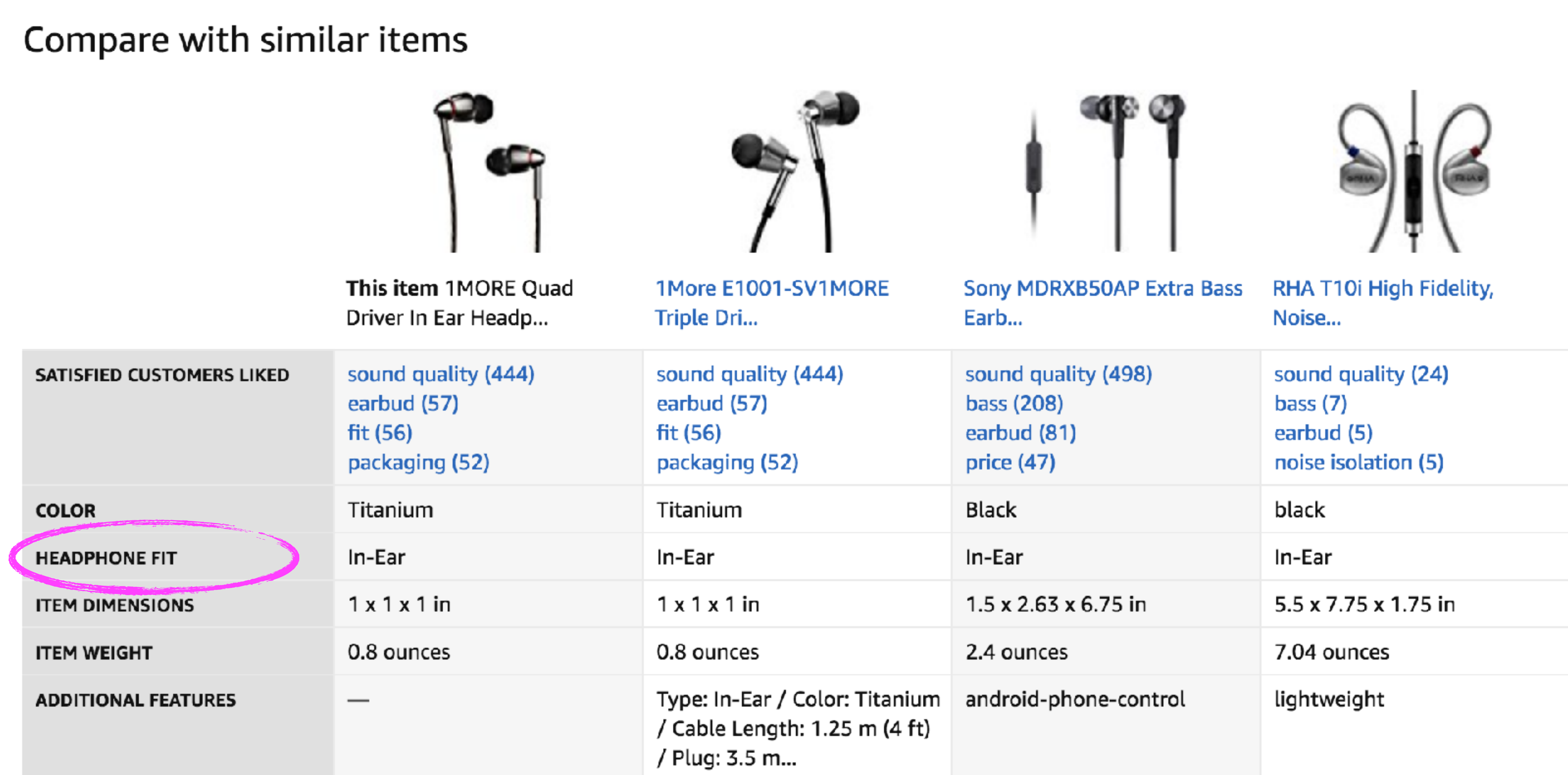
Sumic GT-A All-Season Radial Tire - 195/65R15 91H

★★★★☆ ▾ 138 customer reviews | 37 answered questions

Price: $47.92 **FREE Shipping** (3 days) for Prime members Details

Brand	SUMIC
Model	GT-A
Item Weight	19.1 pounds
Product Dimensions	25 x 25 x 7.9 inches
Item model number	5514018
Manufacturer Part Number	5514018
Special Features	tread_wear_indicator
Section Width	195 millimeters
Aspect Ratio	65
Construction	Radial
Rim Diameter	15 inches
Load Index Rating	91
Speed Rating	H
Tread Depth	10 thirty_seconds_inches
UTQG	400 A A

Case Study: amazon.com items for sale



Compare with similar items

	This item 1MORE Quad Driver In Ear Headp...	1More E1001-SV1MORE Triple Dri...	Sony MDRXB50AP Extra Bass Earb...	RHA T10i High Fidelity, Noise...
SATISFIED CUSTOMERS LIKED	sound quality (444) earbud (57) fit (56) packaging (52)	sound quality (444) earbud (57) fit (56) packaging (52)	sound quality (498) bass (208) earbud (81) price (47)	sound quality (24) bass (7) earbud (5) noise isolation (5)
COLOR	Titanium	Titanium	Black	black
HEADPHONE FIT	In-Ear	In-Ear	In-Ear	In-Ear
ITEM DIMENSIONS	1 x 1 x 1 in	1 x 1 x 1 in	1.5 x 2.63 x 6.75 in	5.5 x 7.75 x 1.75 in
ITEM WEIGHT	0.8 ounces	0.8 ounces	2.4 ounces	7.04 ounces
ADDITIONAL FEATURES	—	Type: In-Ear / Color: Titanium / Cable Length: 1.25 m (4 ft) / Plug: 3.5 m...	android-phone-control	lightweight

Document Data

- How would we represent this data in a relational DB?
  - Use a “type” column with a Single Table Inheritance strategy?
  - Store this data as text?
  - Lots and lots of relationships?
- We really just want to represent some of this information using some sparse hierarchical format like JSON

JSON and Array Column Types

- Much more than just stringified JSON stored in a column
- Ability to query deep into hierarchical objects
- Depending on RDBMs, deep indexing may be possible
- PostgreSQL 9.2+ support: VERY GOOD
- MySQL 5.7.8+ support: OK
- SQLite w/ JSON1 Extension: MEH

Creating a JSON column

- PostgreSQL allows a default value, MySQL does not
- NOT NULL and UNIQUE apply as usual

```sql
CREATE TABLE IF NOT EXISTS StoreItem (
  label TEXT NOT NULL,
  colors JSON
);
```

INSERTing a JSON value (MySQL and PostgreSQL)

- Single quotes around a JSON value

```
CREATE TABLE IF NOT EXISTS StoreItem (
  label TEXT NOT NULL,
  colors JSON
);
```

```
INSERT INTO StoreItem(label, colors)
  VALUES(‘Hats', '{ "small": ["red", "blue"], "medium": ["green"], "large": [] }');
INSERT INTO StoreItem(label, colors)
  VALUES('Shirts', '{ "small": ["purple"], "medium": [], "large": ["white"] }');
INSERT INTO StoreItem(label, colors)
  VALUES('Socks', '{ "small": ["red"], "medium": ["red"], "large": ["red"] }');
INSERT INTO StoreItem(label, colors)
  VALUES('Flash Drives', '{ "capacity64g": ["silver"] }');
```

MySQL and JSON: Creating JSON values

- Lots of JSON functions for constructing, manipulating, searching and serializing JSON
- When figuring things out, use hard-coded values and experiment with simple operations

MySQL

```
-- Create a JSON array ["one", "two", "three"]
SELECT JSON_ARRAY('one', 'two', 'three');

– Create a JSON object {"a": "First", "b": "Second"}
SELECT JSON_OBJECT('a', 'First', 'b', 'Second');
```

MySQL and JSON: Searching

- JSON_SEARCH can be used for finding a value within a JSON object

MySQL

```
SELECT JSON_SEARCH( -- Check for the presence of a value within an array
   -- ["foo", "bar", "baz"]
  JSON_ARRAY('foo', 'bar', 'baz'),
  'all', -- find "one" or "all" results?
  'ba%' -- thing to find. wildcard % is allowed
); -- ["$[1]", "$[2]"]

SELECT JSON_SEARCH( -- Check for the presence of a value within an object
  -- { "properties": ["foo", "bar", "baz"] }
  JSON_OBJECT('properties', JSON_ARRAY('foo', 'bar', 'baz')),
  'all', -- find "one" or "all" results?
  'ba%' -- thing to find. wildcard % is allowed
); -- ["$.properties[1]", "$.properties[2]"]
```

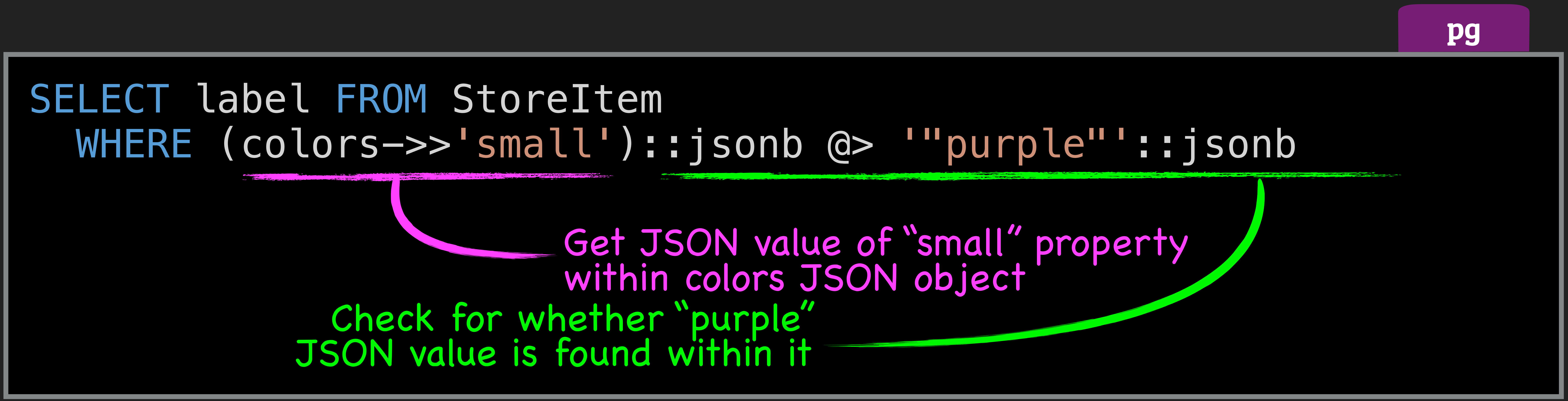
MySQL and JSON

MySQL

```sql
CREATE TABLE IF NOT EXISTS StoreItem (
  label TEXT NOT NULL,
  colors JSON
);

INSERT INTO StoreItem(label, colors)
  VALUES('Jeans', '{"small": ["red", "blue"], "medium": ["green"], "large": [] }');
INSERT INTO StoreItem(label, colors)
  VALUES('Shirts', '{"small": ["purple"], "medium": [], "large": ["white"] }');
INSERT INTO StoreItem(label, colors)
  VALUES('Socks', '{"small": ["red"], "medium": ["red"], "large": ["red"] }');
INSERT INTO StoreItem(label, colors)
  VALUES('Flash Drives', '{"capacity64g": ["silver"] }');


SELECT * FROM StoreItem
WHERE JSON_SEARCH(lower(colors->"$.small[*]"), 'one', lower('red')) IS NOT NULL;
```

PostgreSQL and JSON

- Using the JSONB type improves I/O performance and compressibility
- It also allows use of some important operators!
- Lots of JSON functions for your data processing pleasure!



```sql
SELECT label FROM StoreItem
  WHERE (colors->>'small')::jsonb @> '"purple"'::jsonb
```

Get JSON value of "small" property within colors JSON object

Check for whether "purple" JSON value is found within it

PostgreSQL - JSON Operators

Operator	Right Operand	Description	Example	Example Result
->	int	JSON array element	'[{"a":"foo"},{"b":"bar"}, {"c":"baz"}]'::json->2	{ "c": "baz" }
->	text	JSON field by key	'{"a": {"b":"foo"}}'::json->'a'	{ "b": "foo" }
->>	int	JSON array element as text	'[1,2,3]'::json->>2	3
->>	text	JSON field as text	'{"a":1,"b":2}'::json->>'b'	2
#>	text[]	JSON object at path	'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'	{ "c": "foo" }
#>>	text[]	JSON object at path as text	'{"a":[1,2,3],"b": [4,5,6]}'::json#>>'{a,2}'	3

PostgreSQL - JSONB Operators

Operator	Right Operand	Description	Example
@>	jsonb	Does the left JSON value contain within it the right value?	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb
<@	jsonb	Is the left JSON value contained within the right value?	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb
?	text	Does the key/element string exist within the JSON value?	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	Do any of these key/element strings exist?	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']
?&	text[]	Do all of these key/element strings exist?	'["a", "b"]'::jsonb ?& array['a', 'b']
@>	jsonb	Does the left JSON value contain within it the right value?	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb

PostgreSQL - Arrays

- Multidimensional array type independent of JSON
- Works with existing types like INTEGER, VARCHAR, etc…
- Much simpler to work with than arbitrary JSON values

pg

```
CREATE TABLE UserAccount (
  email VARCHAR(255) PRIMARY KEY,
  names TEXT[] NOT NULL,
  locations REAL[2][]
);
```

PostgreSQL - Creating Array Values

- Create values as strings

pg

```
INSERT INTO UserAccount
  VALUES('mike@example.com',
         '{"Mike", "North"}',
         '{{39.0968, 120.0324}, {37.3861, 122.0839}}');
```

- Or using the ARRAY type

pg

```
INSERT INTO UserAccount
  VALUES('mike@example.com',
         ARRAY['Mike', 'North'],
         ARRAY[[39.0968, 120.0324],
               [37.3861, 122.0839]]);
```

PostgreSQL - SELECTing Array Values

- Use square brackets to access elements by id in an array

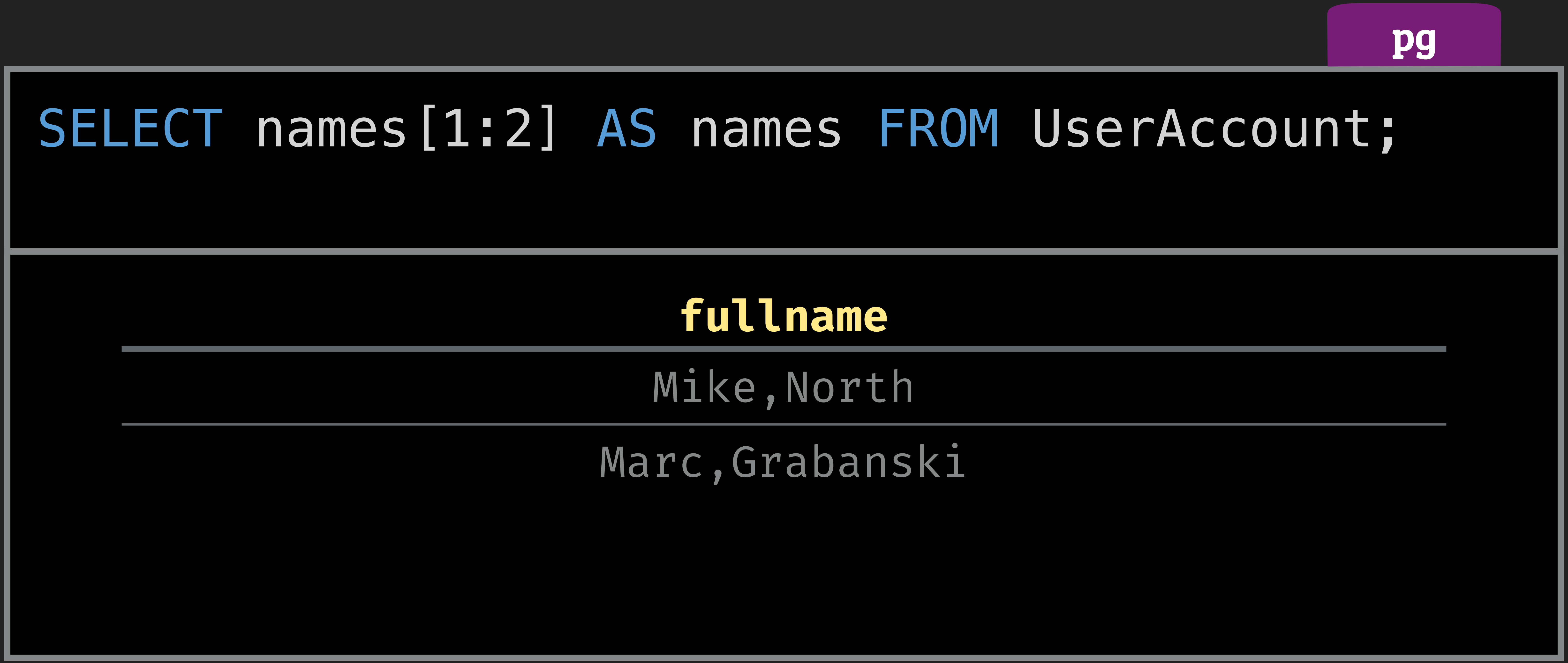
pg

```
SELECT (names[1] || ' ' || names[2]) AS fullname
FROM UserAccount;
```

```
   fullname
-------------
 Mike North
-------------
Marc Grabanski
```

PostgreSQL - SELECTing Array Values

- Use square brackets to access elements by id in an array

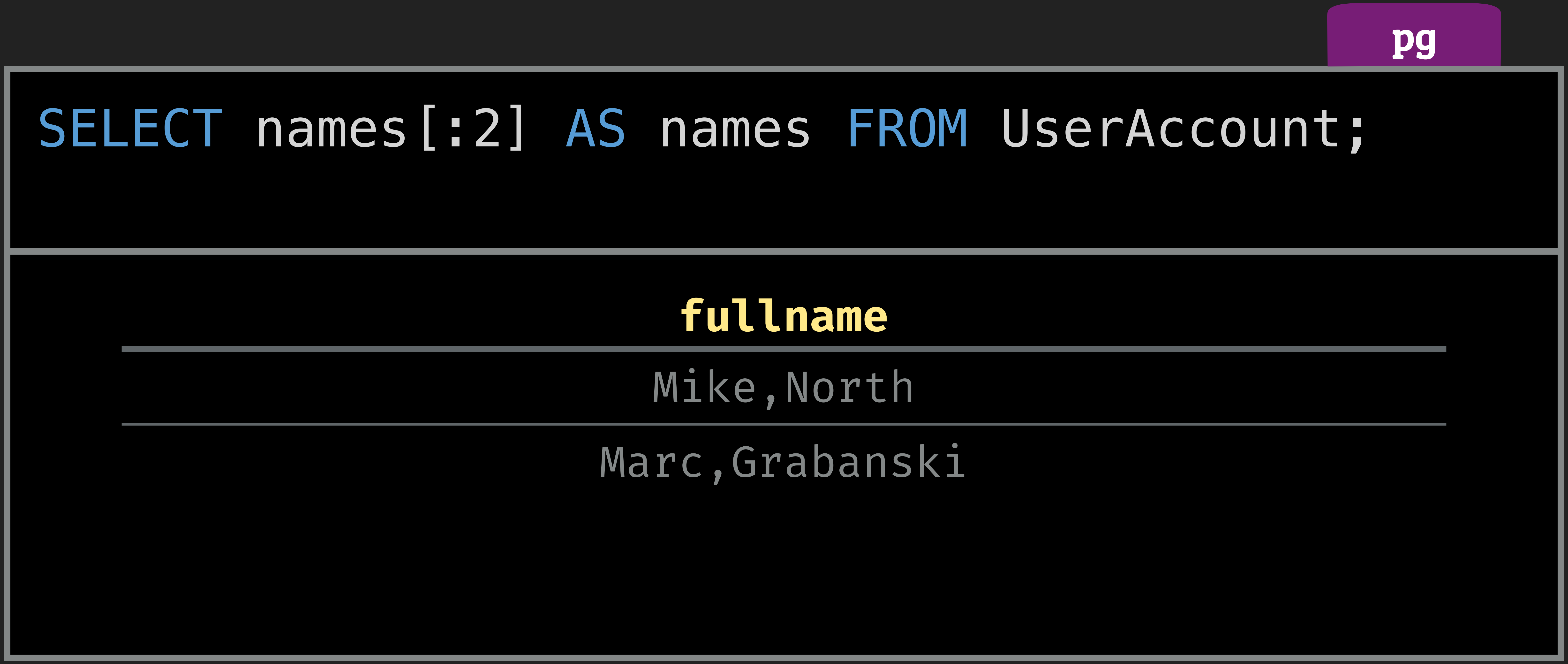


```sql
SELECT names[1:2] AS names FROM UserAccount;
```

```
      fullname
-------------------
   Mike,North
-------------------
 Marc,Grabanski
```

PostgreSQL - SELECTing Array Values

- Use square brackets to access elements by id in an array



```sql
SELECT names[:2] AS names FROM UserAccount;
```

```
      fullname
----------------
  Mike,North
----------------
Marc,Grabanski
```

PostgreSQL - Array Inclusion Check

- Use the ANY keyword to check whether a given value is present in an array

pg

```
SELECT email FROM UserAccount WHERE 'North' = ANY (names);
```

```
      email
-----------------
mike@example.com
```

- Use the && operator to check for an overlap between arrays

```
SELECT ARRAY[4, 5, 6] && ARRAY [2, 6, 9, 16]; -- true
```

JSON and Array Columns

5

- Create a DB migration to add metadata and tags columns to the Product table
  - **metadata** - should be of type `jsonb`, with a default value of

```
{ flavor: { spicy: -1, sweet: -1, sour: -1, salty: -1, bitter: -1 } }
```

  - **tags** - should be of type `string[]`, with a default value of

```
[]
```

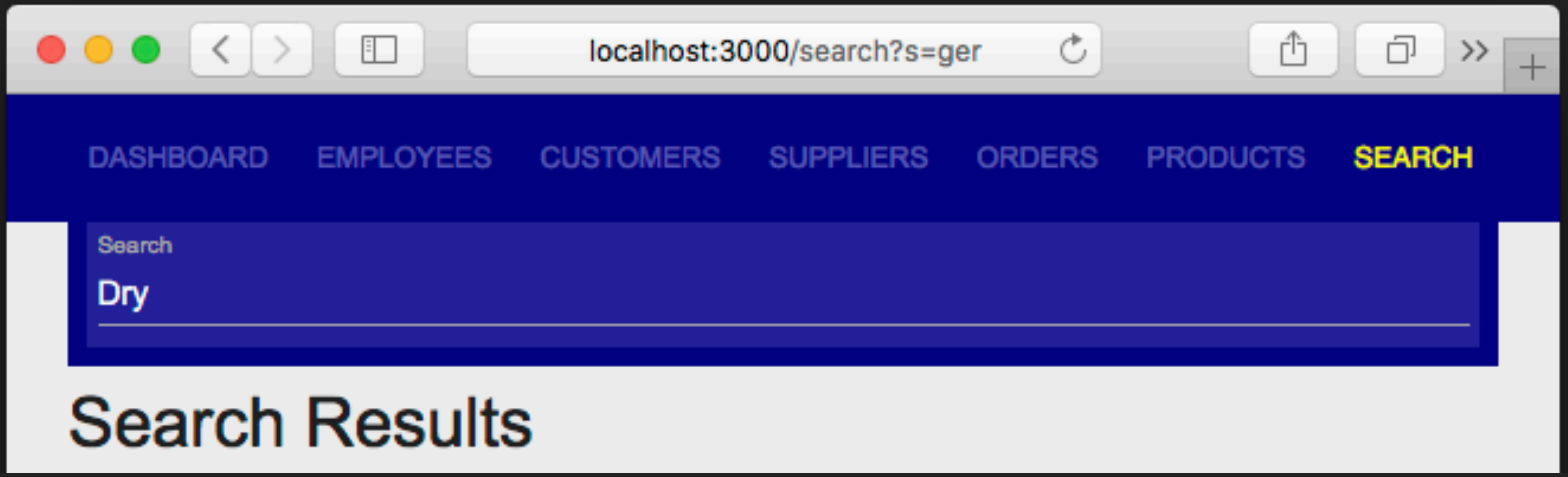
- In `getAllProducts` make use of the filter property to create a query that reflects the requested tag/flavor constraints

./src/data/products.js

```
getAllProducts({
  requiredTags: ['alcoholic'],
  flavor: [{ flavorName: 'sweet', type: 'greater-than', level: 2 }]
});
```

```
npm run test:ex:watch 14
```

Let's consider our search feature



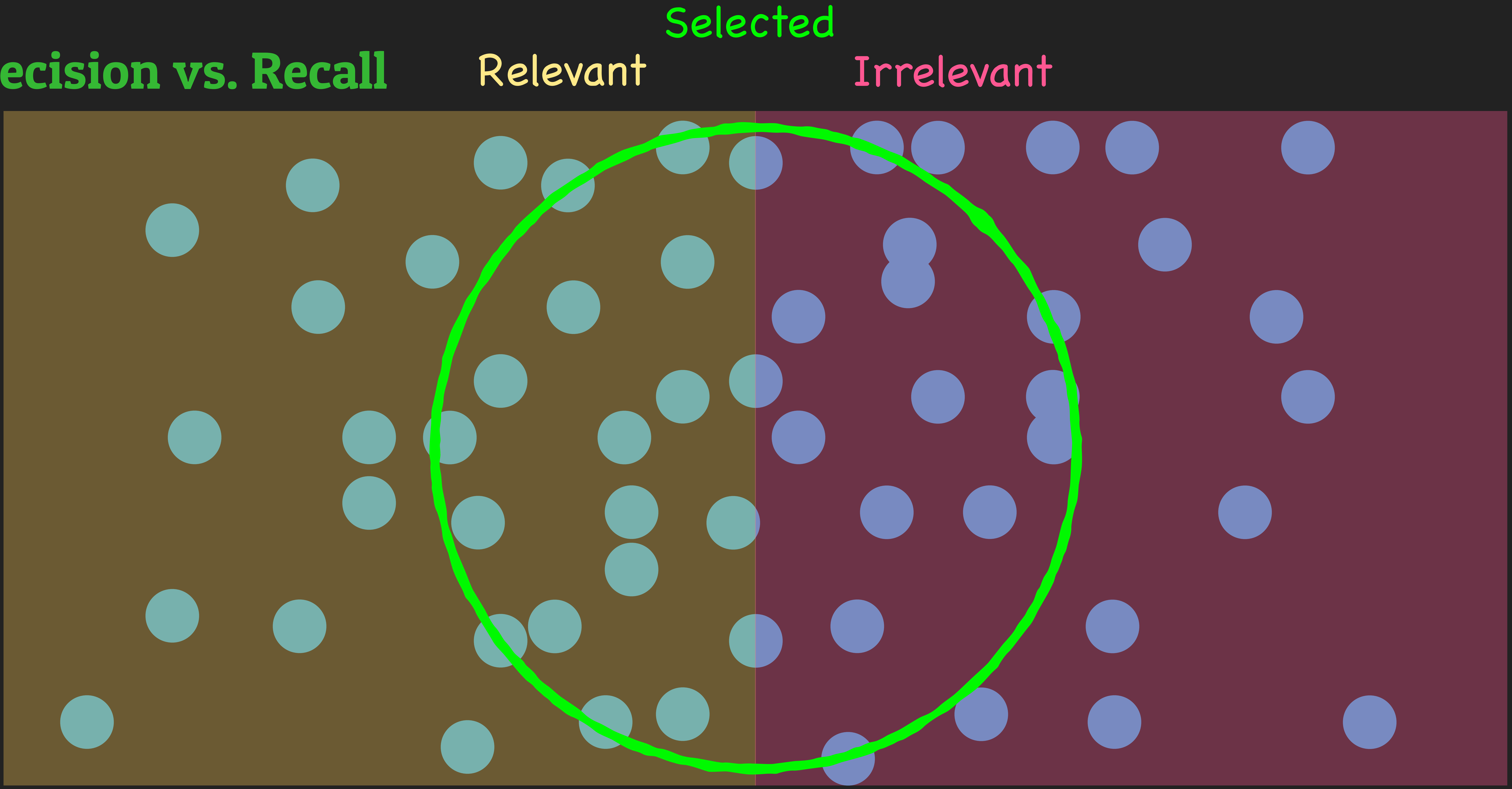
Search

Dry

Search Results

- A `WHERE "build" LIKE task` search is ok, but many relevant results will be omitted. We want...
  - Multiple words from the same root to be treated as a single concept ("build", "built", "building", "builds")
  - Omit stop words ("the", "and", "is", etc...)
  - Indexing instead of full-table scanning

Precision vs. Recall

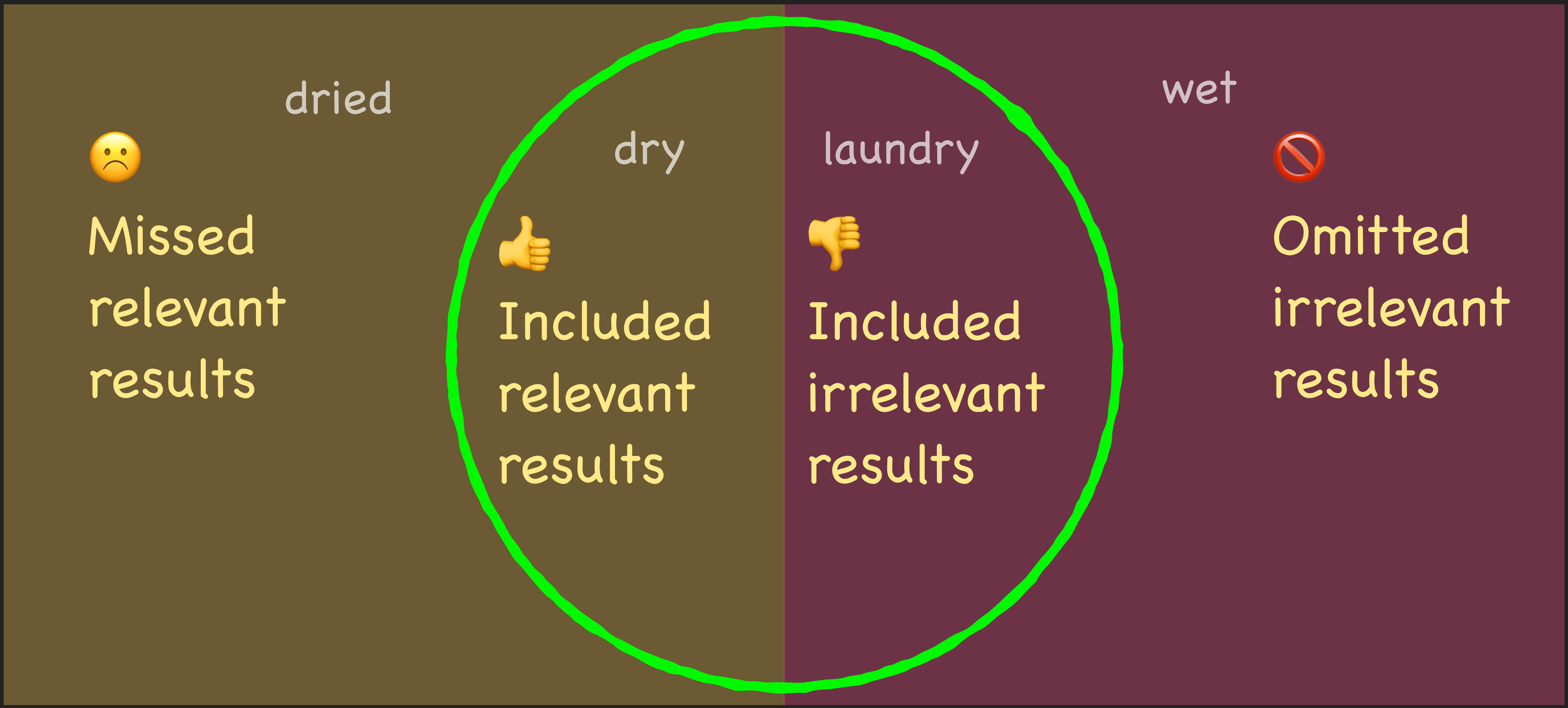


Precision vs. Recall

Selected

Relevant

Irrelevant



dried

☹️

Missed relevant results

dry

👍

Included relevant results

laundry

👎

Included irrelevant results

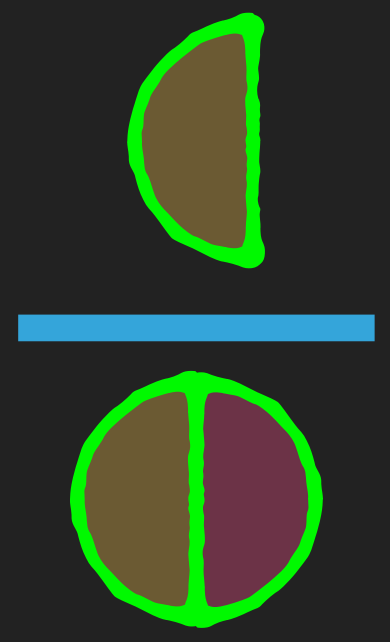
wet

🚫

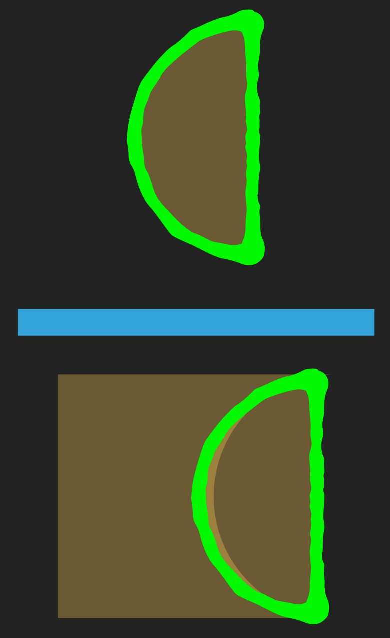
Omitted irrelevant results

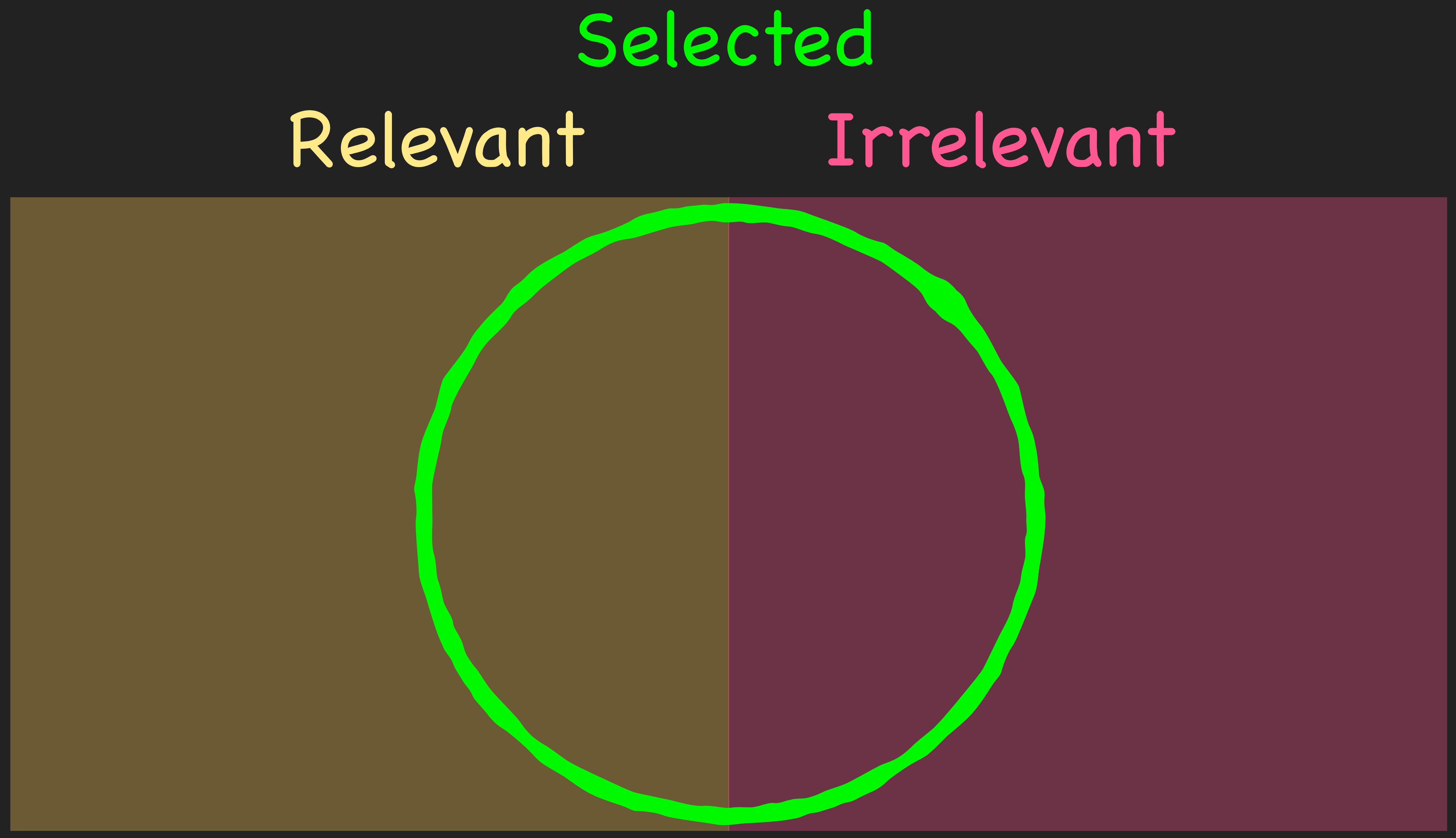
Precision vs. Recall

- **Precision** is the ratio of selected results that are relevant



- **Recall** is the ratio of relevant results that are selected





Full-Text Search

- A set of features collectively known as "full text search" will help!
- PostgreSQL support: GREAT, MySQL support: OK, SQLite support: NO
- You may not need Lucene, Solr or Sphinx anymore!
- Based around the idea of a reverse index

FORWARD INDEX

document	words
Document 1	css, styling, grid, selector
Document 2	function, variable, selector

REVERSE INDEX

word	documents
css	Document 1
styling	Document 1
grid	Document 1
selector	Document 1, Document 2
function	Document 2
variable	Document 2

Full-Text Search: Normalized Terms

- We'll want to keep track normalized keywords associated with each record
  - MySQL: does this automatically in "caching tables"
  - PostgreSQL: I recommend creating a new column

```
CREATE TABLE Whiskey(
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  notes TEXT
);
```

pg

```
ALTER TABLE Whiskey ADD COLUMN whiskey_fts tsvector;
UPDATE Whiskey SET whiskey_fts = to_tsvector('english', name || ' ' || notes);
```

Full-Text Search: Indices

- In MySQL, create a FULLTEXT index on exactly the keys you'll search against

MySQL

```
CREATE FULLTEXT INDEX whiskey_fts_idx ON Whiskey(name, notes);
```

- In PostgreSQL, create a GIN index on a `tsvector`, concatenating where appropriate

pg

```
CREATE INDEX whiskey_fts_idx ON Whiskey
USING GIN (whiskey_fts);
```

Full-Text Search: SELECTing Relevant Results

- In MySQL 5.7.6+, we can use the MATCH … AGAINST( ) syntax

MySQL

```
SELECT * FROM Whiskey
WHERE MATCH (name, notes) AGAINST ('sweet' IN BOOLEAN MODE );
```

- NATURAL LANGUAGE MODE is the default
- BOOLEAN MODE allows for operators: +Android -Samsung
- Minimum search term length: 3 characters

Full-Text Search: SELECTing Relevant Results

- In PostgreSQL, we'll just query against our special whiskey_fts column
- Use the to_tsquery( ) function to convert a search term to a tsquery type
- The @@ operator checks for a match between a tsvector and tsquery

pg

```sql
SELECT * FROM Whiskey
WHERE whiskey_fts @@ to_tsquery('dry');
```

- & (AND), | (OR) and ! (NOT) operators may be used in search terms

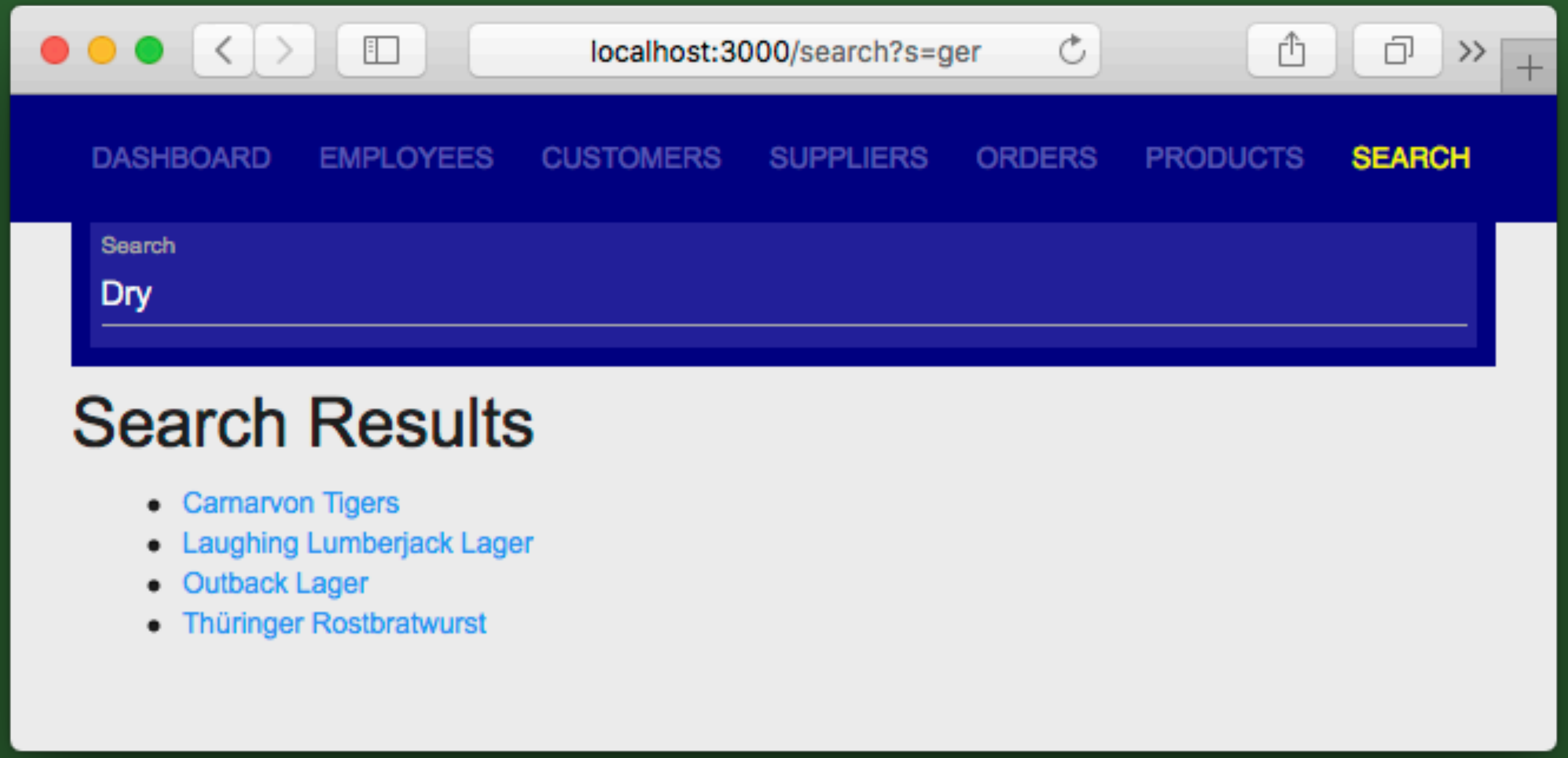
Full Text Search

6

- The global search feature of this app is pretty unimpressive, as it's based around a LIKE comparison.
- Update the query in `getSearchResults` to use PostgreSQL's full-text search capabilities
- Create appropriate indices to keep text search results speedy

./src/data/search.js

```
getSearchResults('foo');
```



Dry

Search Results

- Carnarvon Tigers
- Laughing Lumberjack Lager
- Outback Lager
- Thüringer Rostbratwurst

```
npm run test:ex:watch 16
```

PubSub Messaging Pattern

- Publisher and Subscriber have no direct knowledge of each other
- Subscribe to, and publish to a "topic"
- Different from Observable/Observer, in that neither side has knowledge of the other
- Results in improved scalability and loose coupling
- If you're not subscribed, you miss the message

PubSub Support

- PostgreSQL: GREAT, MySQL: NO, SQLite: NO

/src/db/postgres-pubsub.ts

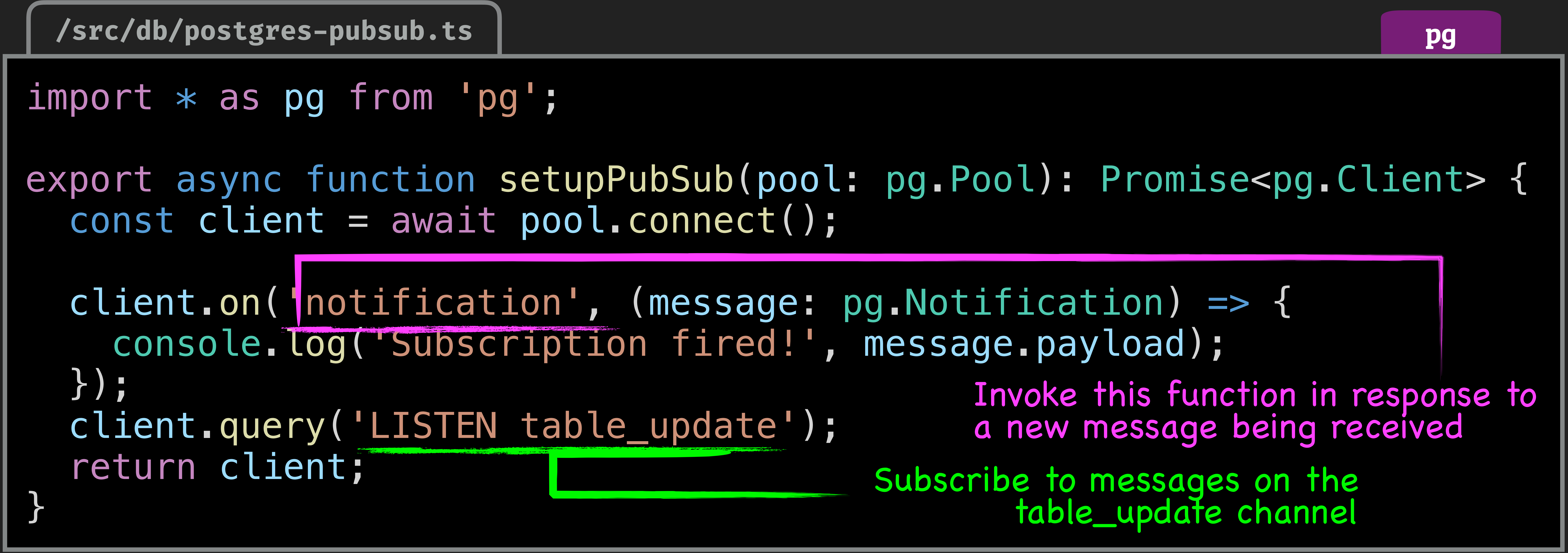
```
import * as pg from 'pg';

export async function setupPubSub(pool: pg.Pool): Promise<pg.Client> {
  const client = await pool.connect();

  client.on('notification', (message: pg.Notification) => {
    console.log('Subscription fired!', message.payload);
  });
  client.query('LISTEN table_update');
  return client;
}
```

Invoke this function in response to a new message being received

Subscribe to messages on the table_update channel

PubSub Support

- PostgreSQL: GREAT, MySQL: NO, SQLite: NO



/src/db/postgres-pubsub.ts

pg

```
import * as pg from 'pg';

export async function setupPubSub(pool: pg.Pool): Promise<pg.Client> {
  const client = await pool.connect();

  client.on('notification', (message: pg.Notification) => {
    console.log('Subscription fired!', message.payload);
  });
  client.query('LISTEN table_update');
  return client;
}
```

Invoke this function in response to a new message being received

Subscribe to messages on the table_update channel

PubSub Support

- Firing a message is done via the NOTIFY command

pg

```
NOTIFY 'message_channel', 'this is the message';
```

- Or by using the pg_notify(<channel>, <message>) function

pg

```
PERFORM pg_notify('message_channel', 'this is the message');
```

- Payloads should be small. Push a small signal and then pull something more substantial in response (if necessary)

Control Flow in SQL

- MySQL has some limited support for control flow
- PostgreSQL has its own procedural language, very similar to Oracle's PL/SQL
- Use this in stored procedures and/or triggers

```
IF p.age >= 21 THEN SELECT * FROM Drinks WHERE hasAlcohol=false
ELSE SELECT * FROM Drinks
END IF
```

PubSub

7

- Add real-time refreshing to this app, using your database's PubSub system.
- Create a custom stored procedure `table_update_notify` that is invoked by triggers `order_notify_update`, `order_notify_insert`, `order_notify_delete` to publish a notification to the `table_update` channel.
- Subscribe for notifications to the `table_update` channel, and call refreshAllClients() to trigger a page reload on the dashboard.

```
import wsm from '../ws';
// Notify all browsers via websocket
wsm().refreshAllClients();
```

```
npm run test:ex:watch 17
```