The Magical Marvels of MongoDB

Example Application

Potion Reviews

Invisibility

Score: 70

Taste: 4 Strength: 1

More Info

Love

Score: 84

Taste: 3 Strength: 5

More Info

Shrinking

Score: 94

Taste: 2 Strength: 3

More Info

THE MAGICAL MARVELS OF MONGODB

Course Outline

01 Conjuring MongoDB

02 Mystical Modifications

03 Materializing Potions

04 Morphing Models

05 Aggregation Apparitions

Conjuring MongoDB

Level 1 – Section 1

Introducing MongoDB

What Is MongoDB?

- Open-source NoSQL database
- Document-oriented
- Great for unstructured data, especially when you have a lot of it

Catch-all term for databases that generally aren't relational and don't have a query language like SQL

Began developing MongoDB as part of their stack

2007

Open-sourced

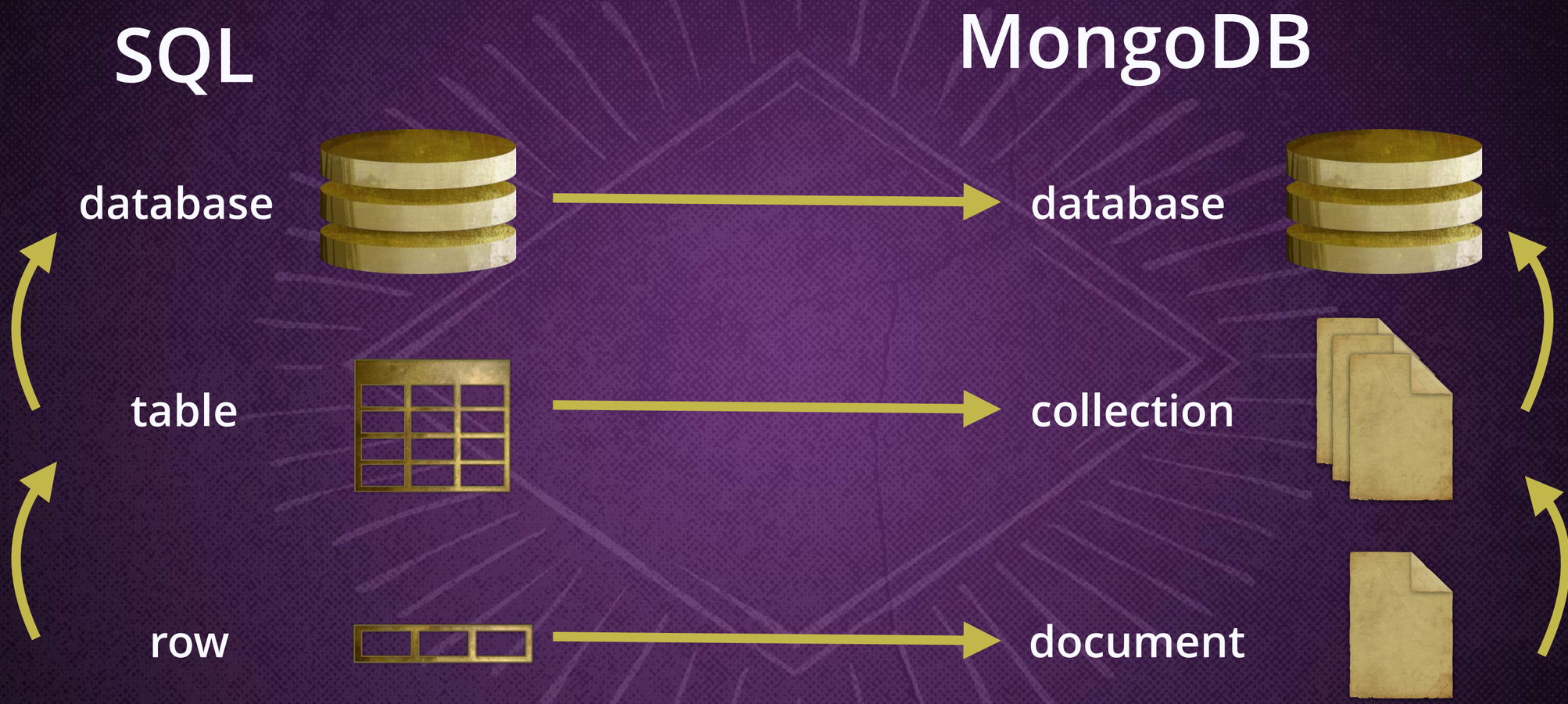
2009

Name comes from the word "humongous"

Renamed to MongoDB

2013

MongoDB Comparison to SQL



The main difference? SQL is *relational* and MongoDB is *document-oriented.*

THE MAGICAL MARVELS OF MONGODB

Relational vs. Document-oriented

Relational database management systems save data in rows within tables. MongoDB saves data as documents within collections.

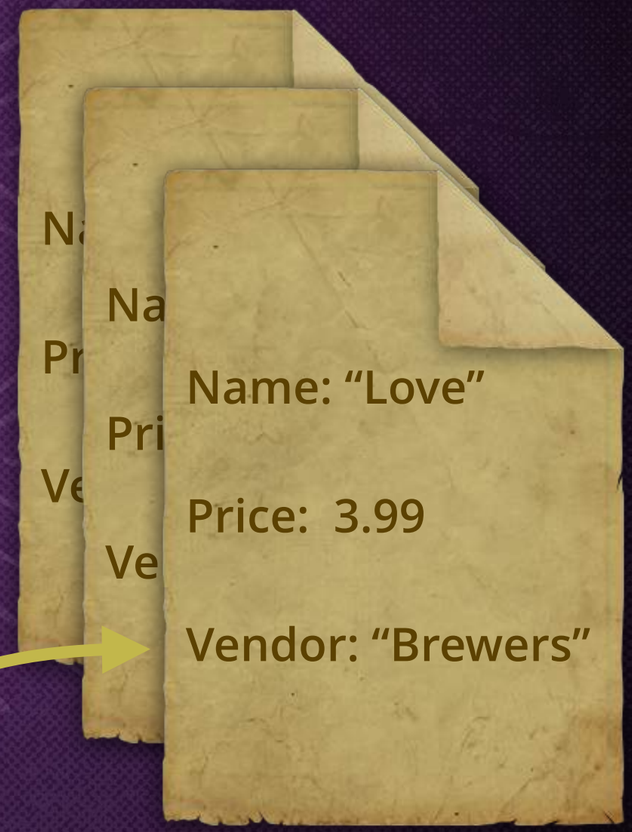
Potions Table

potion_id	name	price	vendor_id
1	“Love”	3.99	2
2	“Invisibility”	15.99	1
3	“Shrinking”	9.99	1

Vendors Table

vendor_id	name
1	“Kettlecooked”
2	“Brewers”

Potions Collection



All data is grouped within documents

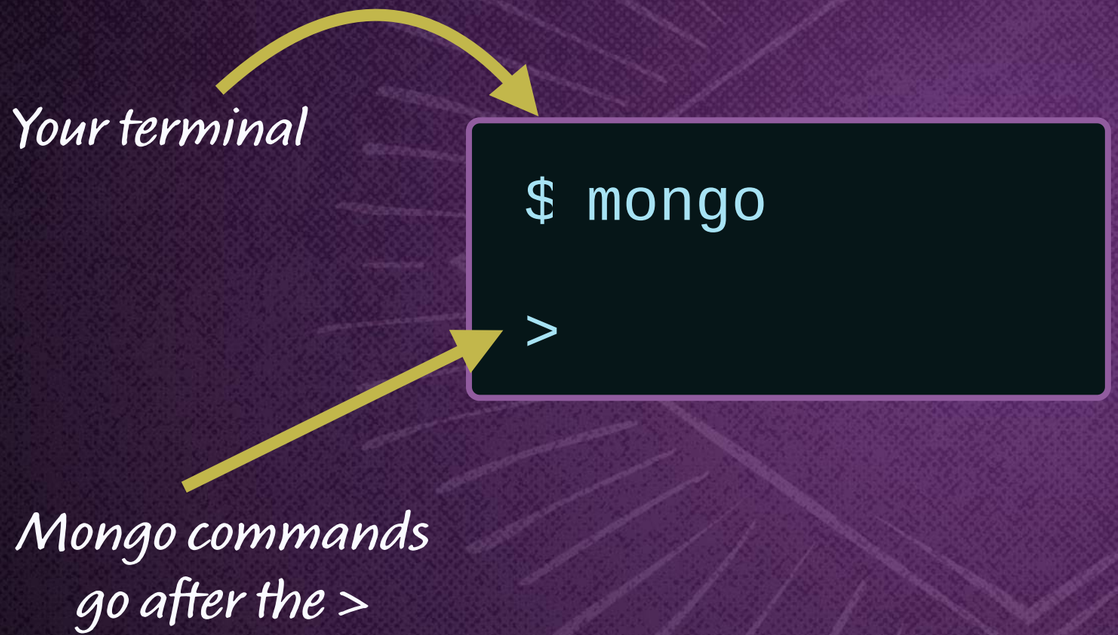
Collections Group Documents

Collections are simply groups of documents. Since documents exist independently, they can have different fields.

This is referred to as a "dynamic schema."

Starting the Shell

We can access MongoDB through the terminal application. If you want try this out locally, follow the link below for MongoDB installation instructions.



Download MongoDB here:
http://go.codeschool.com/download-mongodb

How Do We Interact With MongoDB?

All instances of MongoDB come with a command line program we can use to interact with our database using JavaScript.

SHELL

Regular JavaScript variable assignment

Access the variable to see the contents

Get a response back

```
> var potion = {
    "name": "Invisibility",
    "vendor": "Kettlecooked"
  }
> potion
{
  "name": "Invisibility",
  "vendor": "Kettlecooked"
}
```

This is all just normal JavaScript!

Documents Are Just JSON-like Objects

Here's what a simple document looks like.

SHELL

```
...
{
  "name": "Invisibility",
  "vendor": "Kettlecooked"
}
```

Field

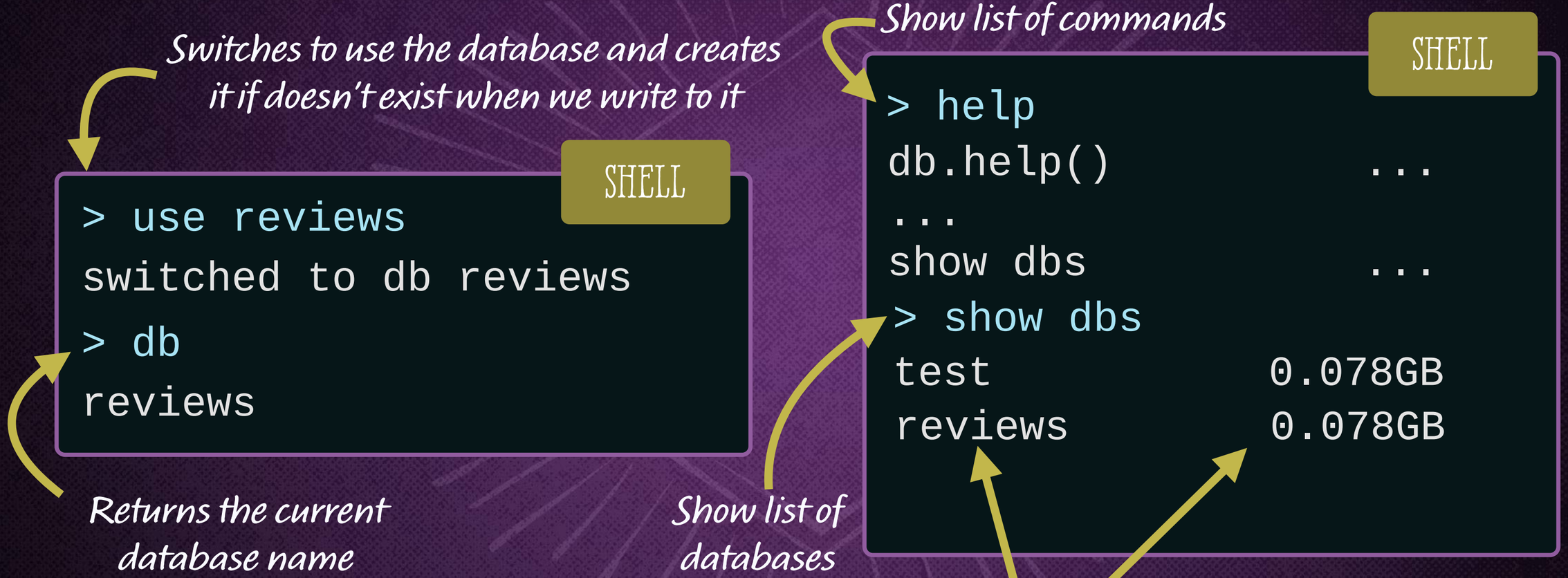
Value

Separated by a comma

Surrounded by curly braces

Using the Shell

MongoDB comes with helper methods to make it easy to interact with the database.



THE MAGICAL MARVELS OF MONGODB

Documents Need to Be Stored in Collections

Documents are always stored in collections within a database.

Potion Document

```
{
  "name": "Invisibility",
  "vendor": "Kettlecooked"
}
```

Document must be placed in a collection

Potions Collection

Inserting a Document Into a Collection

We can use the ***insert()*** collection method to save a potion document to the potions collection.

This collection doesn't exist yet, so it will automatically be created

To write to the database, we specify the collection and the operation to perform

Potion document as a parameter of the insert method

SHELL

```
> db.potions.insert(
  {
    "name": "Invisibility",
    "vendor": "Kettlecooked"
  }
)

WriteResult({ "nInserted": 1 })
```

What's a WriteResult?

Whenever we write to the database, we'll always be returned a WriteResult object that tells us if the operation was successful or not.

SHELL

```
> db.potions.insert(
  {
    "name": "Invisibility",
    "vendor": "Kettlecooked"
  }
)
WriteResult({ "nInserted": 1 })
```

1 document successfully inserted

Finding All Potions

We can use the ***find()*** collection method to retrieve the potion from the inventory collection.

All collection methods must end with parentheses

SHELL

```
> db.potions.find()
 {
    "_id": ObjectId("559f07d741894edebdd8aa6d"),
    "name": "Invisibility",
    "vendor": "Kettlecooked"
 }
```

Unique id that gets automatically generated

Using Find to Return All Documents in a Collection

SHELL

```
> db.potions.insert(...)
  WriteResult({ "nInserted": 1 })
> db.potions.insert(...)
  WriteResult({ "nInserted": 1 })
> db.potions.find()
  { "name": "Invisibility" ... }
  { "name": "Love" ... }
  { "name": "Shrinking" ... }
```

Let's add 2 more potions

Now find returns a total of 3 potions

THE MAGICAL MARVELS OF MONGODB

Conjuring MongoDB

Level 1 – Section 2

Queries and Data Types

ObjectIds Make Documents Unique

Every document is required to have a unique ***_id*** field. If we don't specify one when inserting a document, MongoDB will generate one using the ObjectId data type.

SHELL

```
> db.potions.find()
{
  "_id": ObjectId("559f07d741894edebdd8aa6d"),
  "name": "Invisibility",
  "vendor": "Kettlecooked"
}
```

It's common to let MongoDB handle _id generation.

Finding a Specific Potion With a Query

We can perform a query of equality by specifying a field to query and the value we'd like.

Queries are field/value pairs

SHELL

```
> db.potions.find({"name": "Invisibility"})
{
  "_id": ObjectId("559f07d741894edebdd8aa6d"),
  "name": "Invisibility",
  "vendor": "Kettlecooked"
}
```

Queries will return all the fields of matching documents

Queries That Return Multiple Values

More than 1 document matches the query

SHELL

```
> db.potions.find({"vendor": "Kettlecooked"})
{
  "_id": ObjectId("55d232a5819aa726..."),
  "name": "Invisibility",
  "vendor": "Kettlecooked"
}
{
  "_id": ObjectId("55c3b9501aad0cb0..."),
  "name": "Shrinking",
  "vendor": "Kettlecooked"
}
```

Two separate documents are returned

Queries are case sensitive.

THE MAGICAL MARVELS OF MONGODB

What Else Can We Store?

Documents are persisted in a format called BSON.

BSON is like JSON, so you can store:

Strings	`"Invisibility"`
Numbers	`1400` `3.14`
Booleans	`true` `false`
Arrays	`["newt toes", "pickles"]`
Objects	`{"type" : "potion"}`
Null	`null`

BSON comes with some extras.

ObjectID	`ObjectId(...)`
Date	`ISODate(...)`

Learn more at:
http://go.codeschool.com/bson-spec

Building Out Our Potions

Now that we have a better grasp on documents, let's build out our potion document with all the necessary details.

THE MAGICAL MARVELS OF MONGODB

Adding Price and Score

We can store both integers and floats in a document.

```
{
  "name": "Invisibility",
  "vendor": "Kettlecooked",
  "price": 10.99,
  "score": 59
}
```

MongoDB will preserve the precision of both floats and integers

Adding a tryDate

Dates can be added using the JavaScript Date object and get saved in the database as an ISODate object.

```
{
  "name": "Invisibility",
  "vendor": "Kettlecooked",
  "price": 10.99,
  "score": 59,
  "tryDate": new Date(2012, 8, 13)
}
```

Reads as September 13, 2012, since JavaScript months begin at 0

Dates get converted to an ISO format when saved to the database

```
"tryDate": ISODate("2012-09-13T04:00:00Z")
```

Adding a List of Ingredients

Arrays are a great option for storing lists of data.

```
{
  "name": "Invisibility",
  "vendor": "Kettlecooked",
  "price": 10.99,
  "score": 59,
  "tryDate": new Date(2012, 8, 13),
  "ingredients": ["newt toes", 42, "laughter"]
}
```

We can store any data type within an array

Adding a Potion's Ratings

Each potion has 2 different ratings, which are scores based on a scale of 1 to 5.

```
{
  "name": "Invisibility",
  "vendor": "Kettlecooked",
  "price": 10.99,
  "score": 59,
  ...
}
```

Each rating will have 2 fields

```
{
  "strength": 2,
  "flavor": 5
}
```

MongoDB supports embedded documents so we can simply add this to our potion document

Embedded Documents

We embed documents simply by adding the document as a value for a given field.

```
{
  "name": "Invisibility",
  "vendor": "Kettlecooked",
  "price": 10.99,
  "score": 59,
  "tryDate": new Date(2012, 8, 13),
  "ingredients": ["newt toes", 42, "laughter"],
  "ratings": {"strength": 2, "flavor": 5}
}
```

An embedded document doesn't require an id since it's a child of the main document

Inserting Our New Potion

We've cleared out the inventory collection — now let's add our newly constructed potion!

SHELL

```
> db.potions.insert(
  {
    "name": "Invisibility",
    "vendor": "Kettlecooked",
    "price": 10.99,
    "score": 59,
    "tryDate": new Date(2012, 8, 13),
    "ingredients": ["newt toes", 42, "laughter"],
    "ratings": {"strength": 2, "flavor": 5}
  }
)
WriteResult({ "nInserted": 1 })
```

Finding Potions by Ingredients

Array values are treated individually, which means we can query them by specifying the field of the array and the value we'd like to find.

Same format as basic query for equality

SHELL

```
> db.potions.find({"ingredients": "laughter"})
 {
   "_id": "ObjectId(...)",
   "name": "Invisibility",
   ...
   "ingredients": ["newt toes","secret", "laughter"]
 }
```

Potion contains the right ingredient

Finding a Potion Based on the Flavor

We can search for potions by their ratings using dot notation to specify the embedded field we'd like to search.

```
{
  "_id": "ObjectId(...)",
  "name": "Invisibility",
  ...
  "ratings": {"strength": 2, "flavor": 5}
}
```

We can easily query embedded documents

```
db.potions.find({"ratings.flavor": 5})
```

What About Insert Validations?

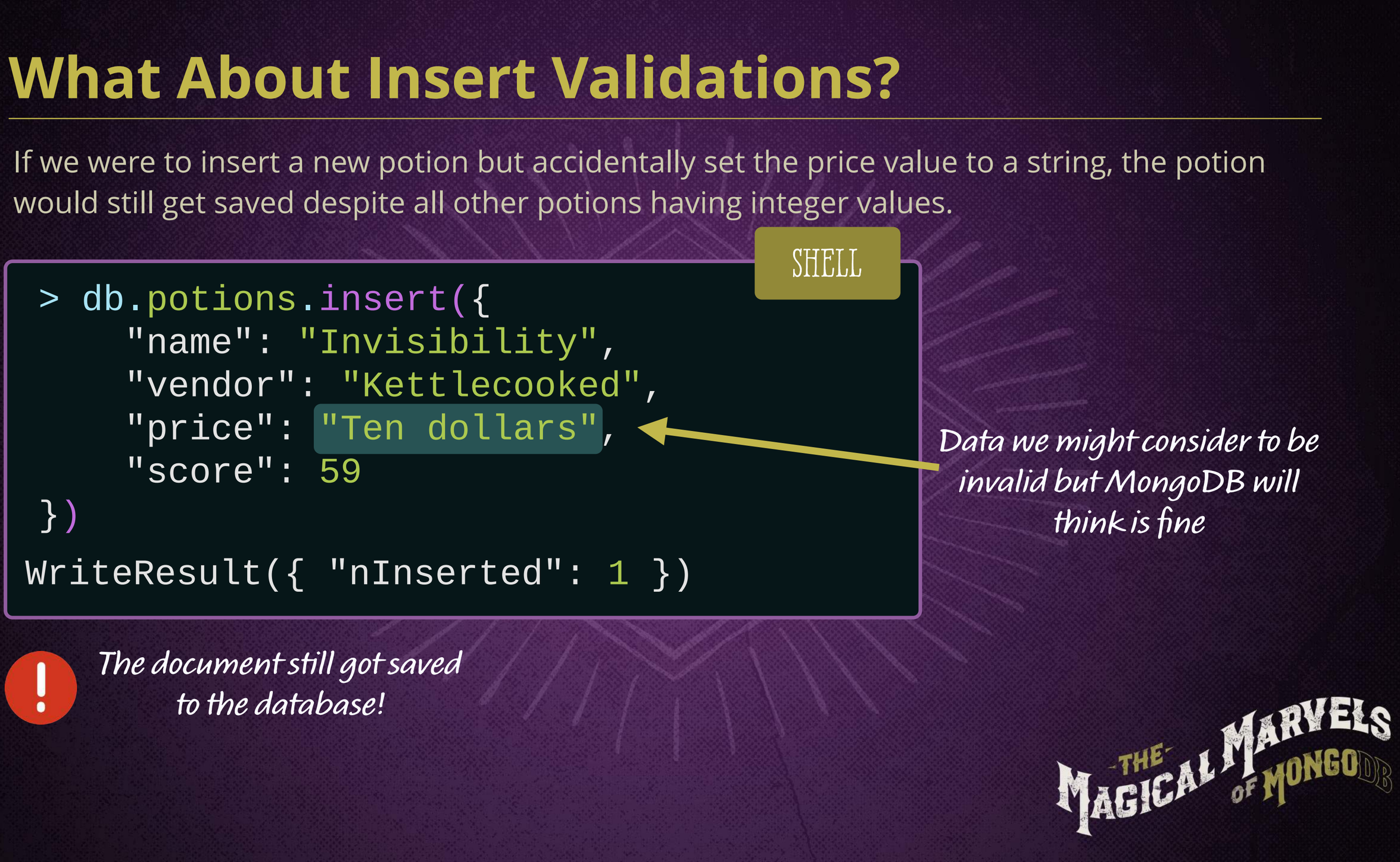
If we were to insert a new potion but accidentally set the price value to a string, the potion would still get saved despite all other potions having integer values.

SHELL

```
> db.potions.insert({
    "name": "Invisibility",
    "vendor": "Kettlecooked",
    "price": "Ten dollars",
    "score": 59
})
WriteResult({ "nInserted": 1 })
```

Data we might consider to be invalid but MongoDB will think is fine

The document still got saved to the database!



Validations Supported by MongoDB

MongoDB will only enforce a few rules, which means we'll need to make sure data is valid client-side before saving it.

```
{
  "_id": ObjectId("55c3b9561..."),
  "name": "Invisibility",
  "vendor": "Kettlecooked",
  "price": 10.99
}
{
  "_id": ObjectId("55d232a51..."),
  "name": "Shrinking",
  "vendor": "Kettlecooked",
  "price": 9.99
}
```

- No other document shares same _id
- No syntax errors
- Document is less than 16mb

Validations Supported by MongoDB

MongoDB will only enforce a few rules, which means we'll need to make sure data is valid client-side before saving it.

```
{
  "_id": ObjectId("55c3b9561..."),
  "name": "Invisibility",
  "vendor": "Kettlecooked",
  "price": 10.99
,
{
  "_id": ObjectId("55d232a51..."),
  "name": "Shrinking",
  "vendor": "Kettlecooked",
  "price": 9.99
}
```

Missing end bracket

- No other document shares same _id
- No syntax errors
- Document is less than 16mb

Validations Supported by MongoDB

MongoDB will only enforce a few rules, which means we'll need to make sure data is valid client-side before saving it.

```
{
  "_id": 1,
  "name": "Invisibility",
  "vendor": "Kettlecooked",
  "price": 10.99
},
{
  "_id": 1,
  "name": "Shrinking",
  "vendor": "Kettlecooked",
  "price": 9.99
}
```

Duplicate _id

- No other document shares same _id
- No syntax errors
- Document is less than 16mb

THE MAGICAL MARVELS OF MONGODB

Mystical Modifications

Level 2 – Section 1

Removing and Modifying Documents

Potion Catastrophe

Uh-oh — we sneezed while performing a spell and ruined some potions in our database!

Need a way to remove the affected potions from our collection

Delete a Single Document

The ***remove()*** collection method will delete documents that match the provided query.

Ruined Potions

SHELL

```
> db.potions.remove(
    {"name": "Love"}
  )
WriteResult({ "nRemoved": 1 })
```

Query matches single document

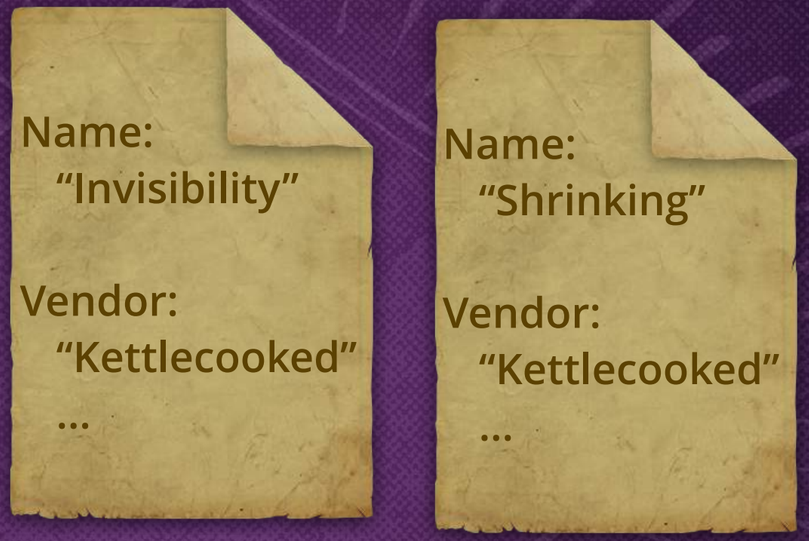
1 document successfully removed

THE MAGICAL MARVELS OF MONGODB

Delete a Single Document

The ***remove()*** collection method will delete documents that match the provided query.

Ruined Potions



SHELL

```
> db.potions.remove(
    {"name": "Love"}
  )
WriteResult({ "nRemoved": 1 })
```

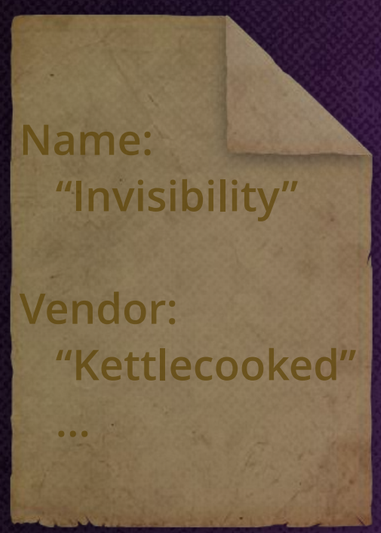
Query matches single document

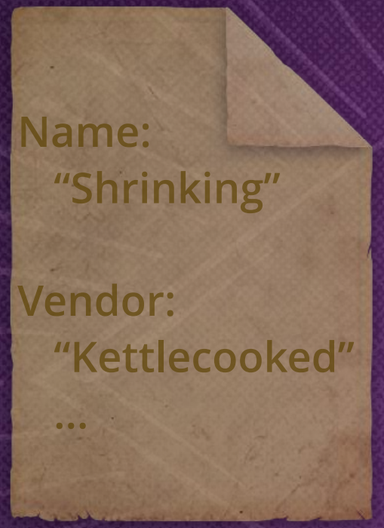
1 document successfully removed

THE MAGICAL MARVELS OF MONGODB

Delete Multiple Documents

If our query matches multiple documents, then ***remove()*** will delete all of them.





Passing {} as the query would delete all documents in the collection.

SHELL

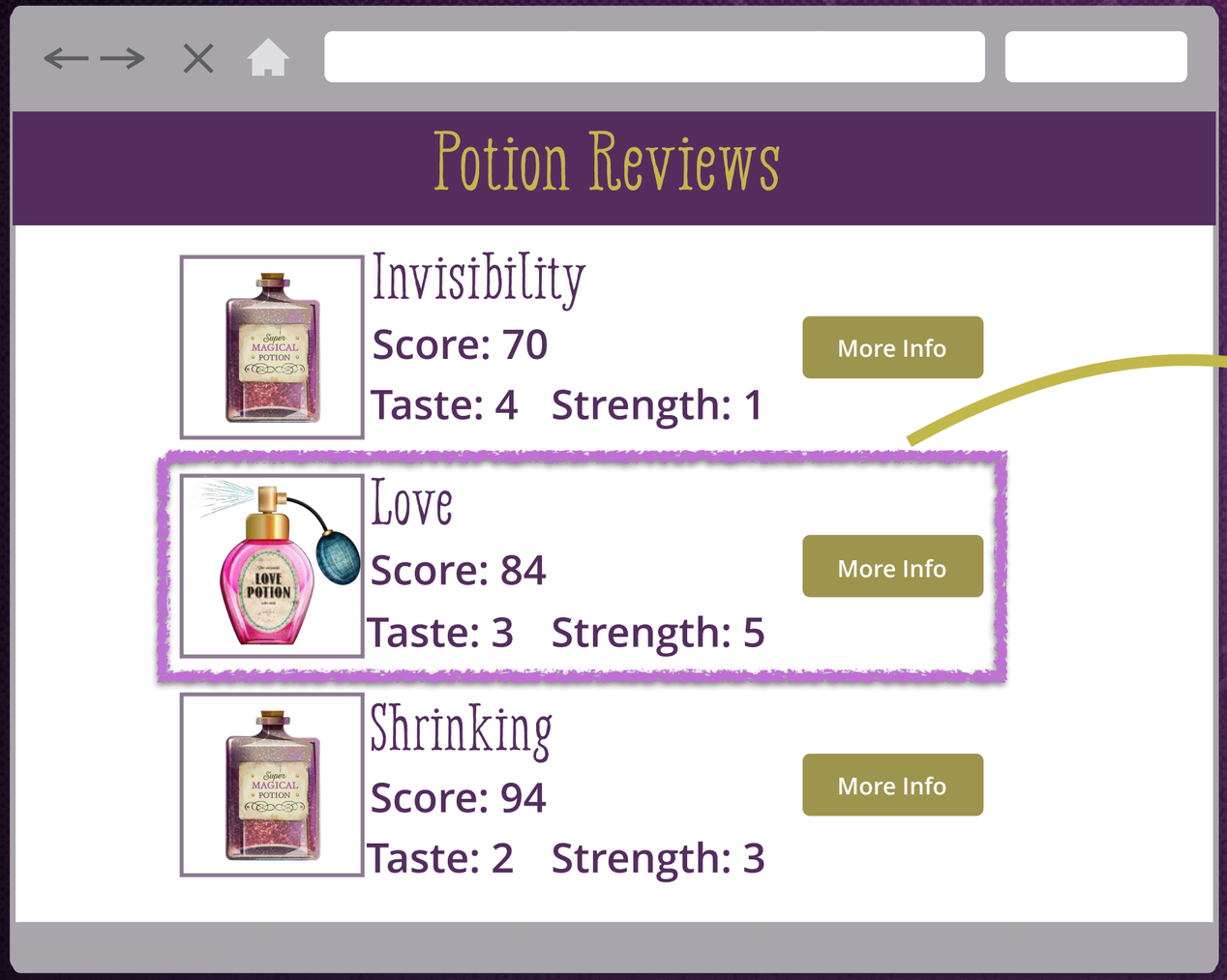
```
> db.potions.remove(
    {"vendor": "Kettlecooked"}
  )
WriteResult({ "nRemoved": 2})
```

Query matches both documents

Removed 2 documents

Update the Price of a Single Potion

We made a typo while inserting our love potion, so let's update the price.

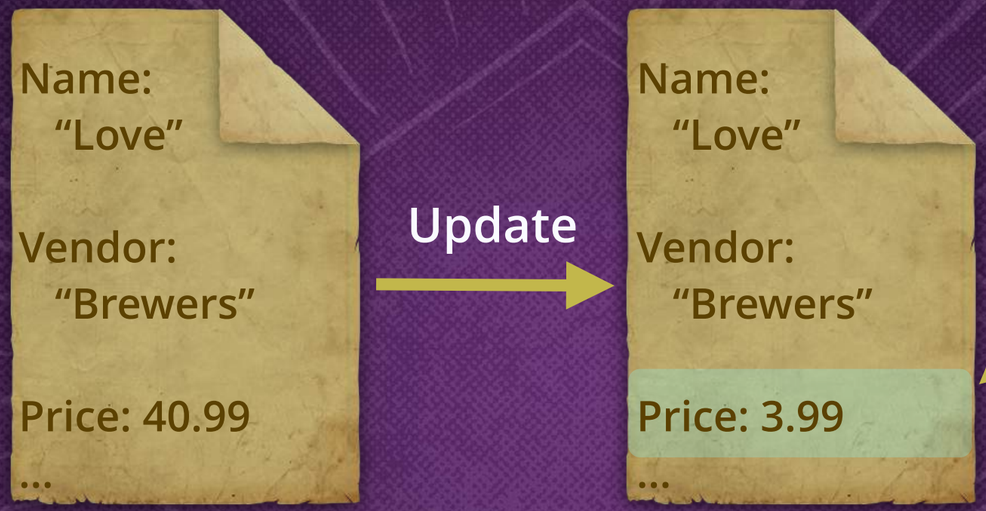


Needs to be updated with the correct price

THE MAGICAL MARVELS OF MONGODB

Updating a Document

We can use the ***update()*** collection method to modify existing documents.



Price is updated!

Update only applied to first matching document

SHELL

```
> db.potions.update(
    {"name": "Love"},
    {"$set": {"price": 3.99 }}
  )
```

Query parameter

Update parameter

Update operators always begin with a $

THE MAGICAL MARVELS OF MONGODB

Understanding the Update WriteResult

The WriteResult gives a summary of what the ***update()*** method did.

SHELL

```
> db.potions.update(
    {"name": "Love"},
    {"$set": {"price": 3.99 }}
  )
WriteResult({
              "nMatched": 1,
              "nUpserted": 0,
              "nModified": 1
            })
```

Number of documents matched

Number of documents that were created

Number of documents modified

Update Without an Operator

If the update parameter consists of only field/value pairs, then everything but the ***_id*** is replaced in the matching document.

Document replaced with the update parameter

No operator, just field/value pair

SHELL

```
> db.potions.update(
    {"name": "Love"},
    {"price": 3.99 }
  )
```

Useful for importing data

THE MAGICAL MARVELS OF MONGODB

Updating Multiple Documents

The update method can take a third parameter for options.

SHELL

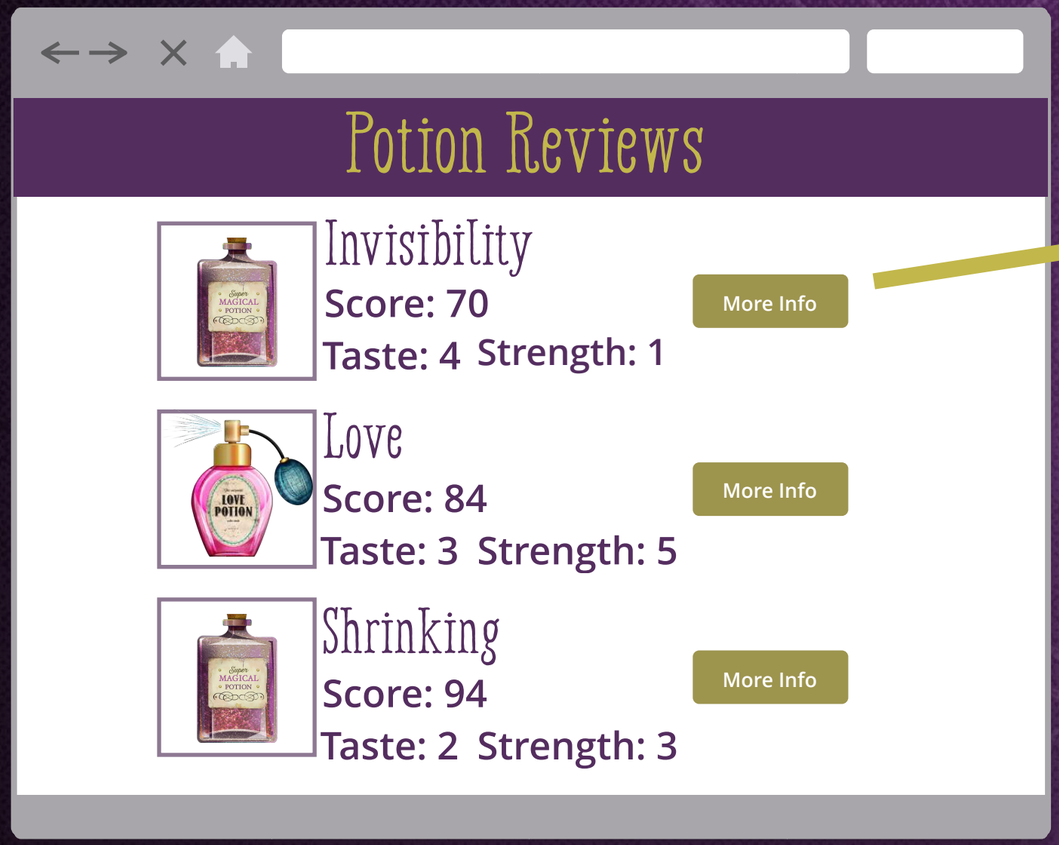
```
> db.potions.update(
    {"vendor": "KC"},
    {"$set": { "vendor": "Kettlecooked" }},
    {"multi": true}
  )
  WriteResult({
                  "nMatched": 4,
                  "nUpserted": 0,
                  "nModified": 4
                })
```

When multi is true, the update modifies all matching documents

4 documents matched and modified

Recording Potion Views

Time to start analyzing which potions are viewed the most. To do this, we need to record each potion page view.



Create or update existing log document

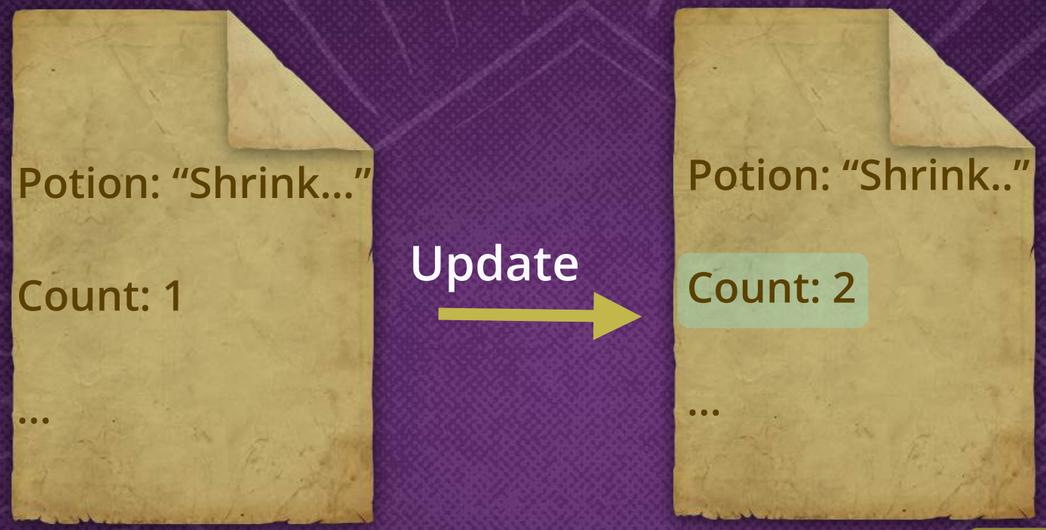
Logs Collection

```
{
  "_id": ObjectId(...),
  "potion": "Frog Tonic",
  "count": 1
}
```

We'll update count with each click

Update a Document's Count

We can use the ***$inc*** operator to increment the count of an existing log document.



Count incremented by 1!

Increments field by specified value

We can increment by any number, positive or negative!

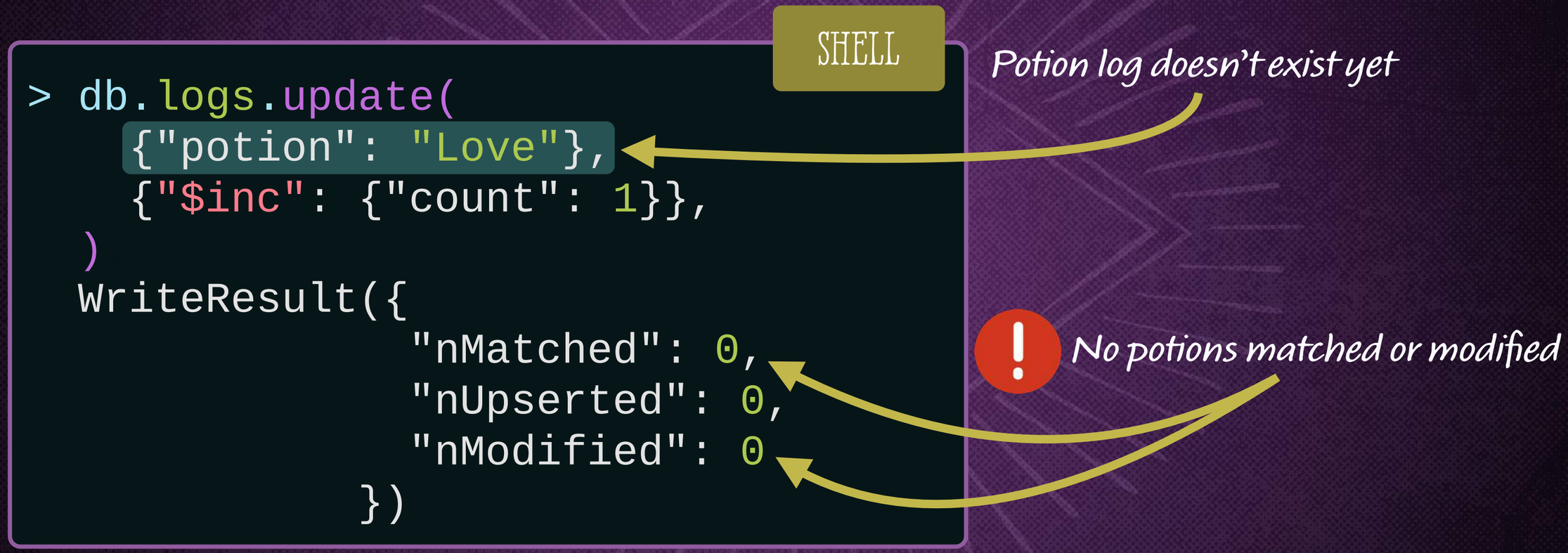
SHELL

```
> db.logs.update(
    {"potion": "Shrinking"},
    {"$inc": {"count": 1}}
  )
```

If field doesn't exist, it gets created with the value.

Update a Non-existing Potion

If we run the update on a potion that doesn't exist, then nothing will happen.



Find or Create With Upsert

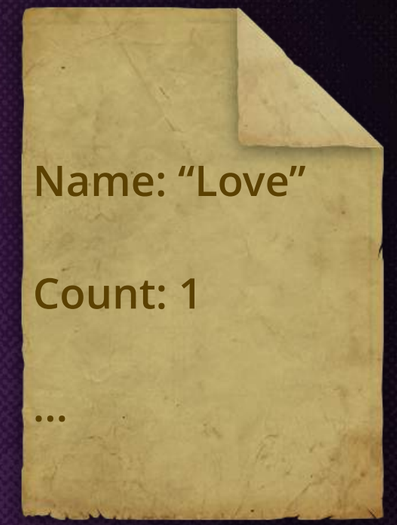
The upsert option either updates an existing document or creates a new one.

If the field doesn't exist, it gets created with the value

SHELL

```
> db.logs.update(
    {"potion": "Love"},
    {"$inc": {"count": 1}},
    {"upsert": true}
  )
  WriteResult({
                "nMatched": 0,
                "nUpserted": 1,
                "nModified": 0
              })
```

Results in new document



Creates a document using the values from the query and update parameter

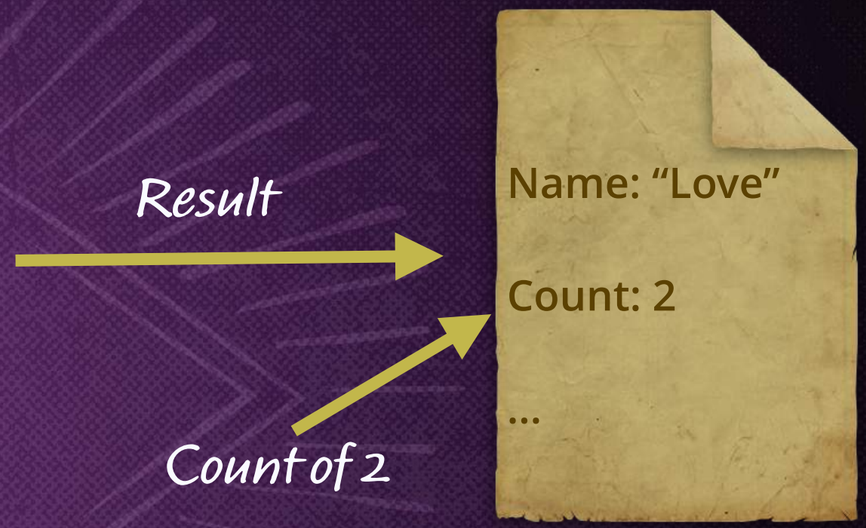
1 document created

Updating Once More

If we run the same update again, the update will act normally and ***upsert*** won't create another document.

SHELL

```
> db.logs.update(
    {"potion": "Love"},
    {"$inc": {"count": 1}},
    {"upsert": true}
  )
  WriteResult({
                "nMatched": 1,
                "nUpserted": 0,
                "nModified": 1
              })
```



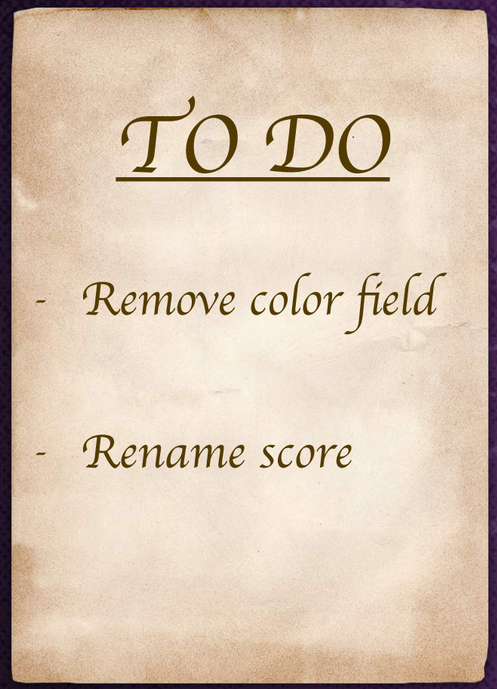
Mystical Modifications

Level 2 – Section 2

Advanced Modification

Improving Potions

We rushed a bit during development and need to fix up our potions. Luckily, we keep a to-do list for what to work on.

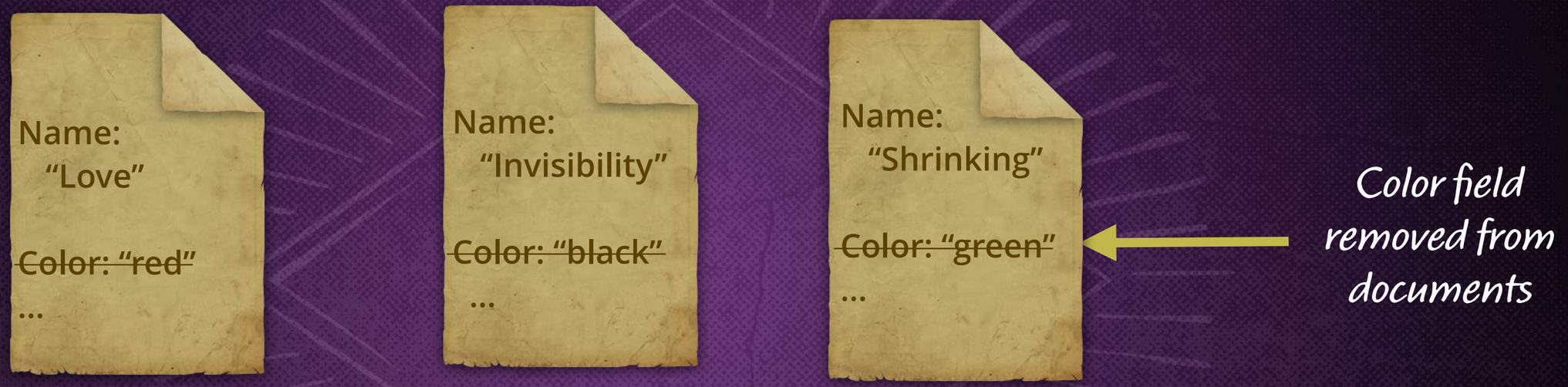


We can accomplish these tasks with update operators!

THE MAGICAL MARVELS OF MONGODB

Removing Fields From Documents

We initially thought we'd need a potion's color, but we never use it. The ***$unset*** operator can be used to remove specified fields.



Query for all potions

Update all potions

SHELL

```
> db.potions.update(
    {},
    {"$unset": {"color": ""}},
    {"multi": true}
  )
```

The value we pass doesn't impact the operation

Updating a Field Name With $rename

We can use ***$rename*** to change field names.

```
{
 "_id": ObjectId(...),
 "name": "Love",
 "score": 84,
 ...
}
```

```
{
 "_id": ObjectId(...),
 "name": "Love",
 "grade": 84,
 ...
}
```

Renamed to grade!

SHELL

```
> db.potions.update(
    {},
    {"$rename": {"score": "grade"}},
    {"multi": true}
  )
```

New field name

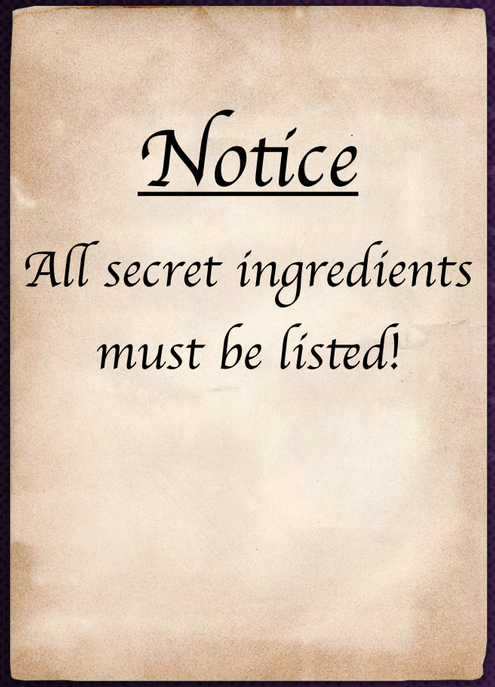
Field to rename

Renames specified field

THE MAGICAL MARVELS OF MONGODB

Potion Ingredient Regulation

The Magical Council has passed a new regulation requiring us to list all ingredients in a potion. No more secret ingredients!

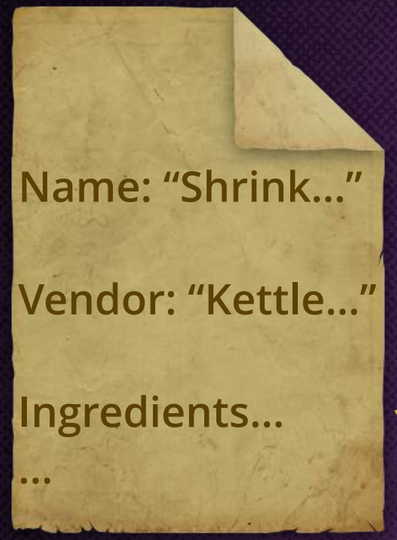



```
{
  "_id": ObjectId(...),
  "name": "Shrinking",
  ...
  "ingredients": ["hippo", "secret", "mouse feet"]
}
```

Need to update with actual ingredient: 42

THE MAGICAL MARVELS OF MONGODB

The Dilemma of Updating an Array




```
"ingredients": ["hippo", "secret", "mouse feet"]
```

SHELL

```
> db.potions.update(
    {"ingredients": "secret"},
    {"$set": {"ingredients": "42"}}
  )
```

Would overwrite the entire array and set it as 42

```
"ingredients": 42
```

Updating Array Values by Location

Since array values are treated individually, we can update a single value by specifying its location in the array using **dot notation**.

```
{
  "_id": ObjectId(...),
  "name": "Shrinking",
  "vendor": "Kettlecooked",
  "score": 94,
  ...
  "ingredients": ["hippo", "secret", "mouse feet"]
}
```

ingredients.0 *ingredients.1* *ingredients.2*

BSON arrays start with an index of 0

Updating Single Array Value

The ***$set*** operator will update the value of a specified field.

The secret ingredient!

SHELL

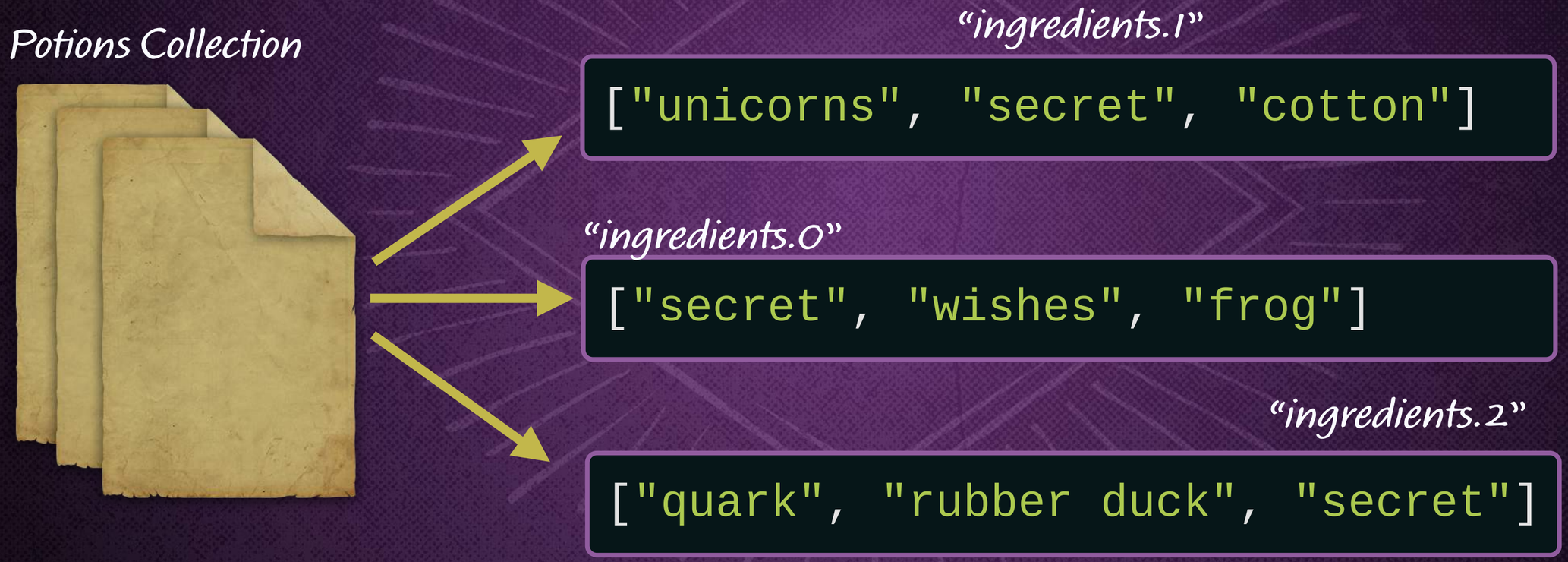
```
> db.potions.update(
    {"name": "Shrinking"},
    {"$set": {"ingredients.1" : 42}}
  )
WriteResult({"nMatched": 1,"nUpserted": 0,"nModified: 1})
```

```
{
  "_id": ObjectId(...),
  "name": "Shrinking",
  ...
  "ingredients": ["hippo", 42, "mouse feet"]
}
```

Successful update

Updating Multiple Arrays

We need to change “secret” in multiple documents, but the location isn’t always the same for every potion.



Updating Values Without Knowing Position

The positional operator is a placeholder that will set the proper position for the value specified in the query parameter.

SHELL

```
> db.potions.update(
    {"ingredients": "secret"},
    {"$set": {"ingredients.$" : 42}},
    {"multi": true}
  )
```

Query for the value we want to change

The $ is a placeholder for the matched value

Multi is true to make the change to all documents

Only updates the first match per document

The Result of Using the Positional Operator to Update

```
...
  {"ingredients": "secret"},
  {"$set": {"ingredients.$" : 42}},
...
```

Potions Collection

"ingredients.1"

```
["unicorns", 42, "cotton"]
```

"ingredients.0"

```
[42, "wishes", "frog"]
```

"ingredients.2"

```
["quark", "rubber duck", 42]
```

Shrunken Conundrum

Uh-oh — the shrinking potion hasn't worn off, and we keep on shrinking! We better update that strength rating.

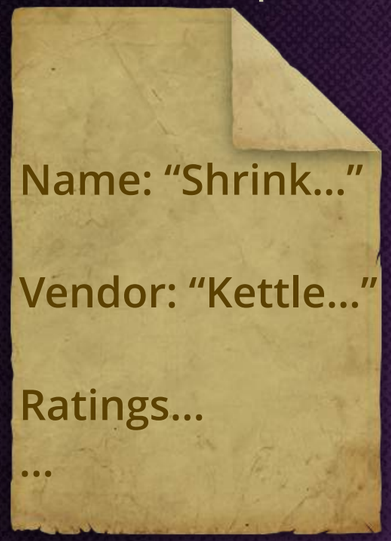

```
{
  "_id": ObjectId(...),
  "name": "Shrinking",
  ...
 "ratings": {
                "strength": 1,
                "flavor": 5
              }
}
```

Update strength to 5

THE MAGICAL MARVELS OF MONGODB

Updating an Embedded Value

We can update using the dot notation to specify the field we want to update.




```
"ratings": {
            "strength": 1,
            "flavor": 5
           }
```

ratings.strength

SHELL

```
> db.potions.update(
    {"name": "Shrinking"},
    {"$set": {"ratings.strength" : 5}}
  )

WriteResult({"nMatched": 1,"nUpserted": 0,"nModified": 1})
```

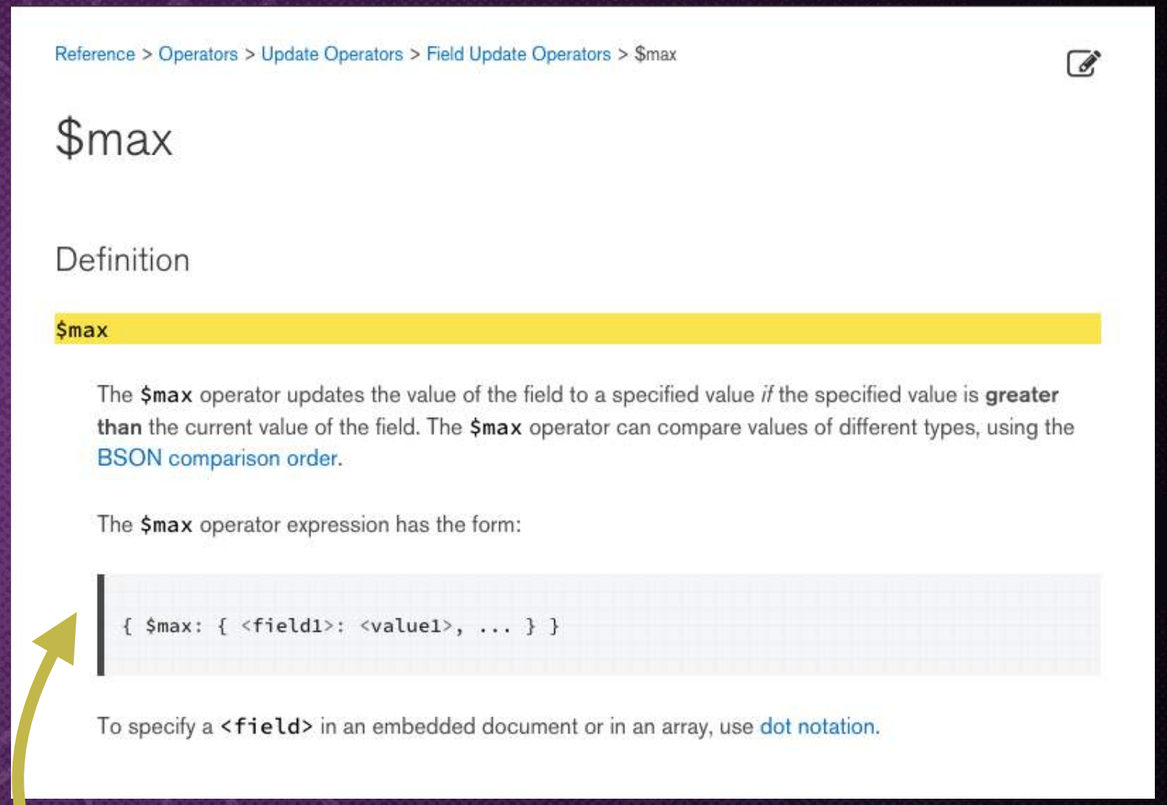
Useful Update Operators

MongoDB provides a variety of ways to modify the values of fields.

`$max` — *Updates if new value is greater than current or inserts if empty*

`$min` — *Updates if new value is less than current or inserts if empty*

`$mul` — *Multiplies current field value by specified value. If empty, it inserts 0.*



Reference > Operators > Update Operators > Field Update Operators > $max

$max

Definition

$max

The **$max** operator updates the value of the field to a specified value *if* the specified value is **greater than** the current value of the field. The **$max** operator can compare values of different types, using the BSON comparison order.

The **$max** operator expression has the form:

```
{ $max: { <field1>: <value1>, ... } }
```

To specify a **<field>** in an embedded document or in an array, use dot notation.

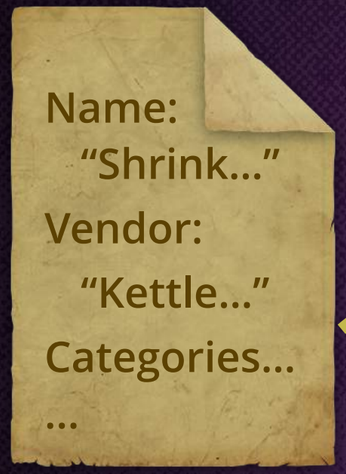
Operator documentation:
http://go.codeschool.com/update-operators

MongoDB's documentation is great

THE MAGICAL MARVELS OF MONGODB

Modifying Arrays

We've added categories to the potions but need a way to easily manipulate the values.

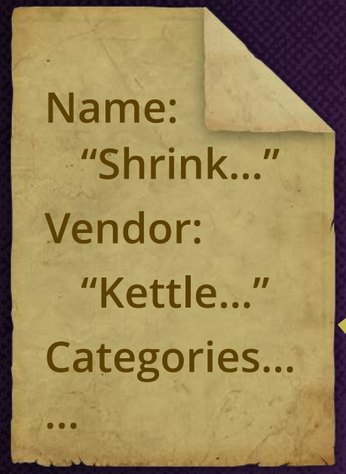


List of categories for the potion

```
"categories": ["tasty", "effective"]
```

Removing the First or Last Value of an Array

The ***$pop*** operator will remove either the first or last value of an array.




```
"categories": ["tasty", "effective"]
```

Doesn't return the value — only modifies the array

```
> db.potions.update(
   {"name": "Shrinking"},
   {"$pop": {"categories": 1}})
```

-1	*Removes the first element*
1	*Removes the last element*

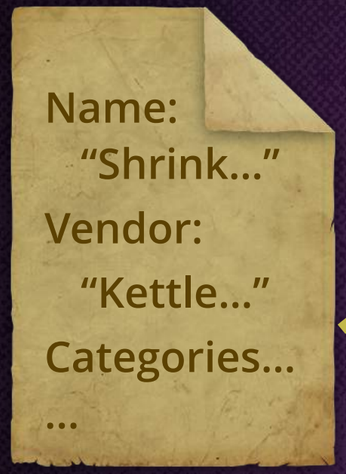
Result

```
"categories": ["tasty"]
```

THE MAGICAL MARVELS OF MONGODB

Adding Values to the End of an Array

The ***$push*** operator will add a value to the end of an array.




```
"categories": ["tasty"]
```

```
> db.potions.update(
   {"name": "Shrinking"},
   {"$push": {"categories": "budget"}})
```

Added to the end

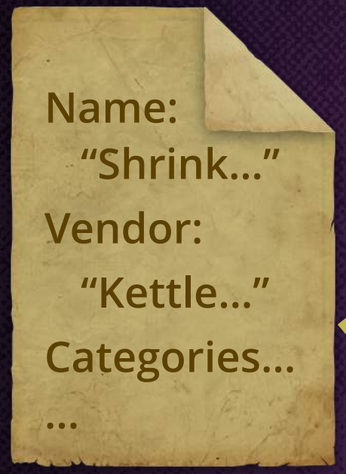
Result

```
"categories": ["tasty", "budget"]
```

THE MAGICAL MARVELS OF MONGODB

Adding Unique Values to an Array

The ***$addToSet*** operator will add a value to the end of an array unless it is already present.




```
"categories": ["tasty", "budget"]
```

```
> db.potions.update(
   {"name": "Shrinking"},
   {"$addToSet": {"categories": "budget"}})
```

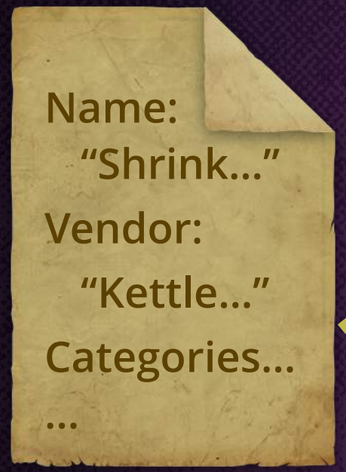
Value already exists, so it doesn't get added again

Result

```
"categories": ["tasty", "budget"]
```

Removing Values From an Array

The ***$pull*** operator will remove any instance of a value from an array.




```
"categories": ["tasty", "budget"]
```

```
> db.potions.update(
  {"name": "Shrinking"},
  {"$pull": {"categories": "tasty"}})
```

Result

```
"categories": ["budget"]
```

If value isn't unique, then all instances will be removed from the array.

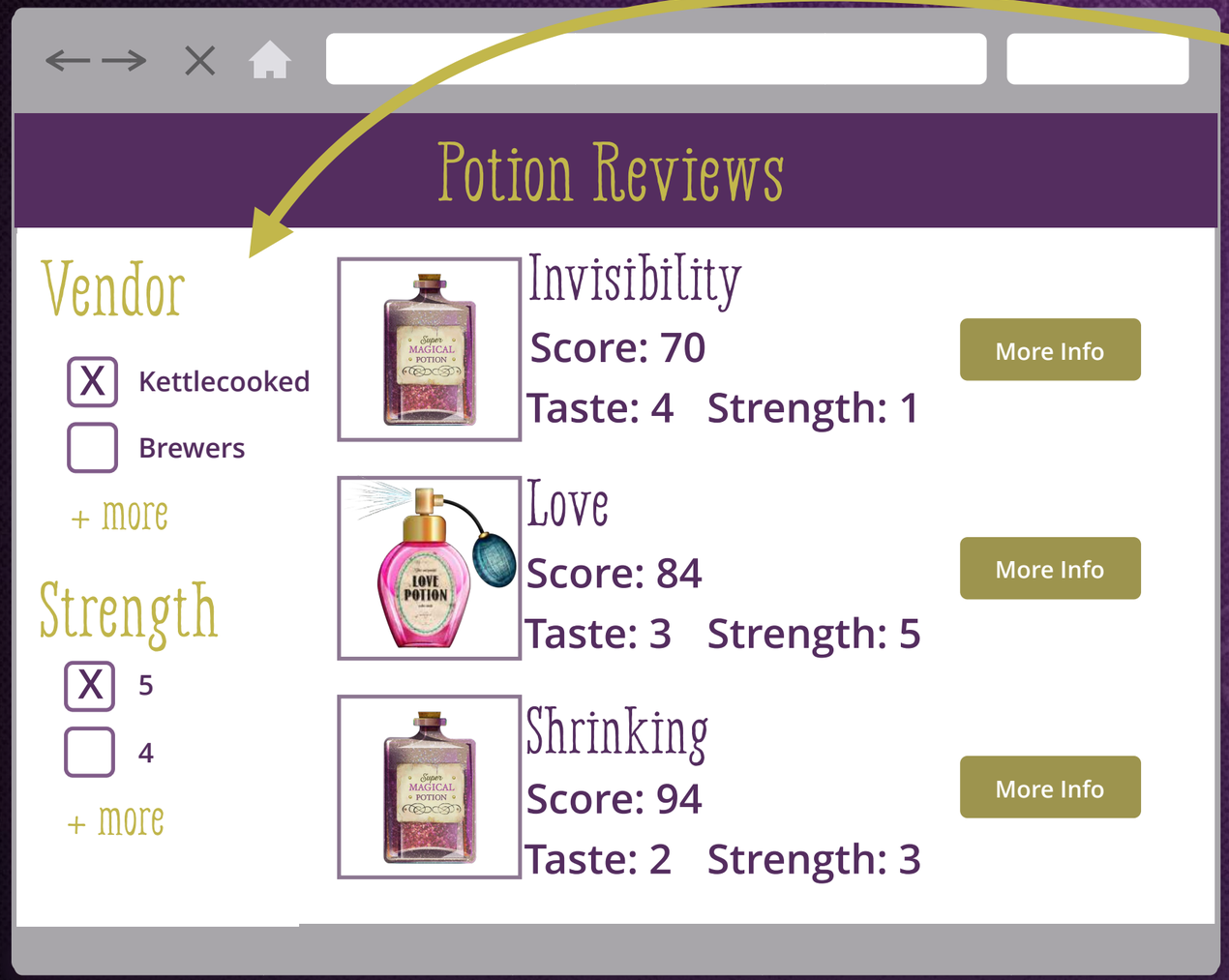
Materializing Potions

Level 3 – Section 1

Query Operators

Adding a Filter for Potions

We've received a new feature request to allow users to filter potions based on multiple criteria.

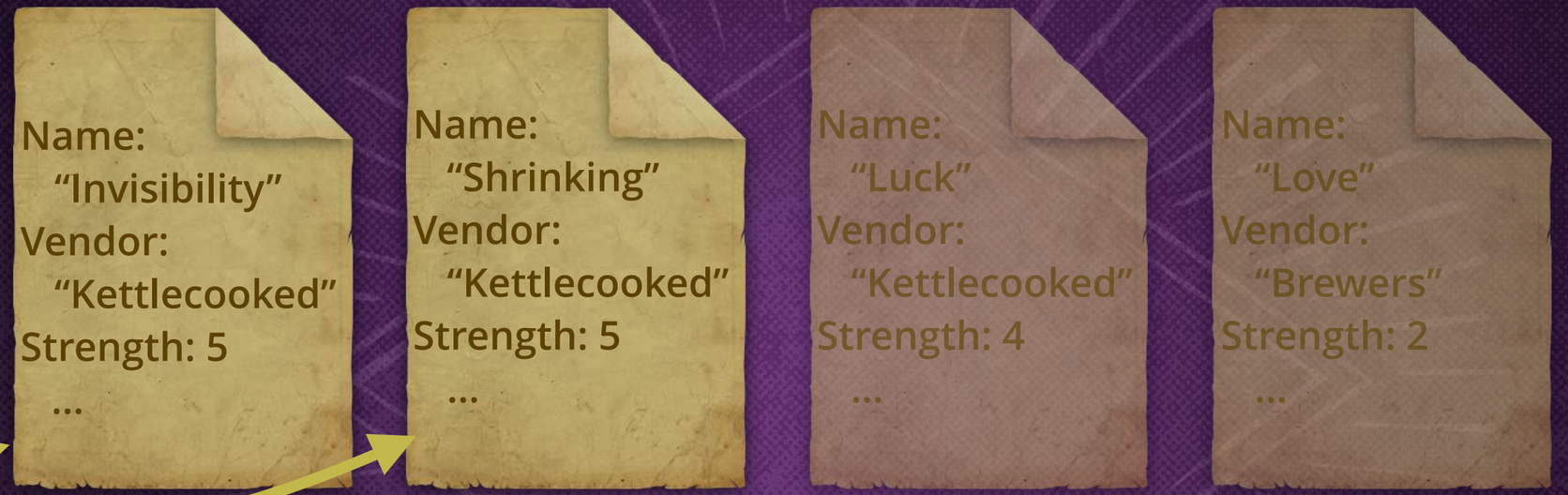


Only show potions made by Kettlecooked that have a strength of 5

THE MAGICAL MARVELS OF MONGODB

Querying With Multiple Criteria

We can query based on multiple criteria by passing in comma-separated queries.



Matches both documents

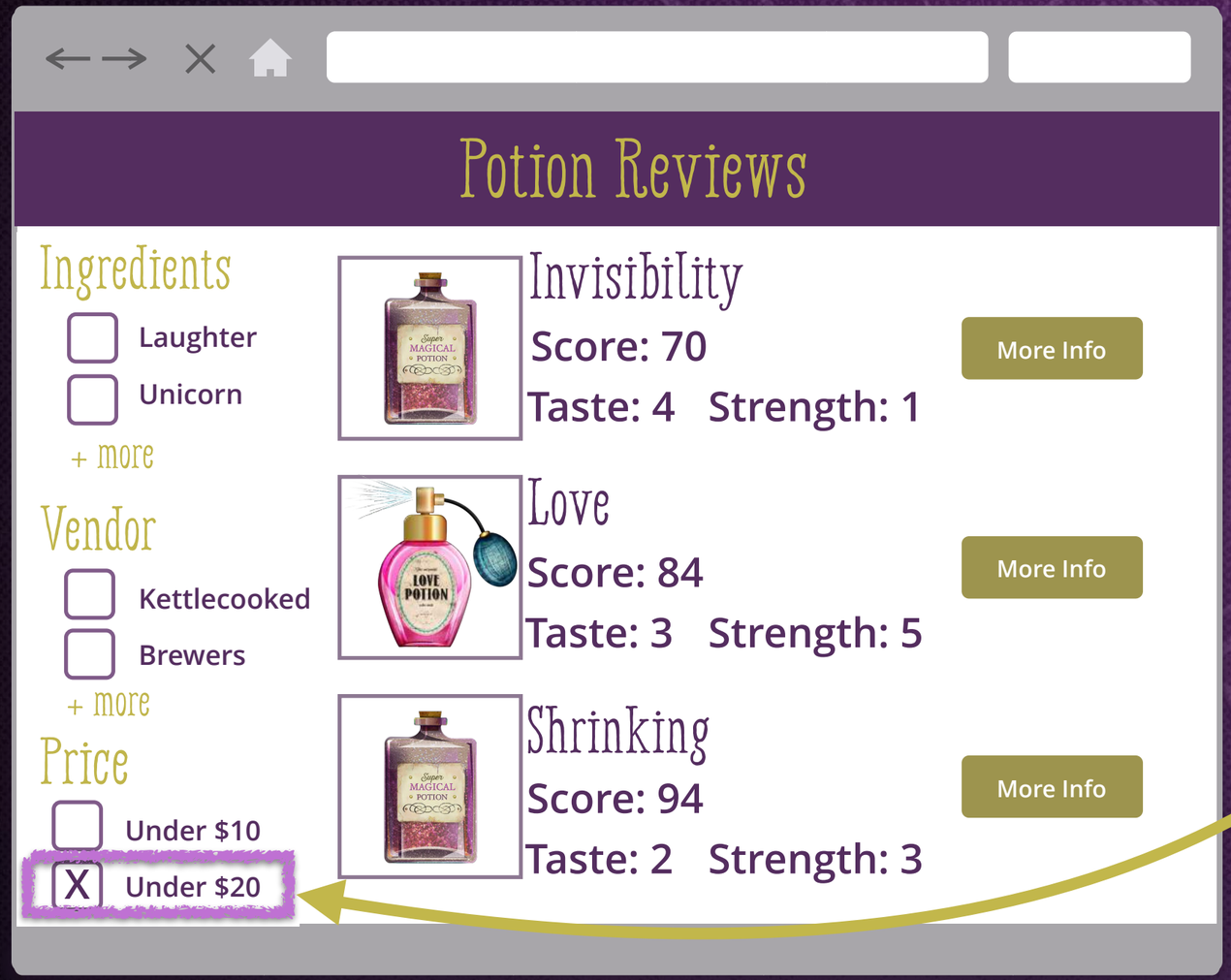
We can pass in more than 1 query

SHELL

```
> db.potions.find(
    {
     "vendor": "Kettlecooked",
     "ratings.strength": 5
    }
  )
```

Finding Potions Based on Conditions

Queries of equality are great, but sometimes we'll need to query based on conditions.



Search for potions with a price less than 20

THE MAGICAL MARVELS OF MONGODB

Comparison Query Operators

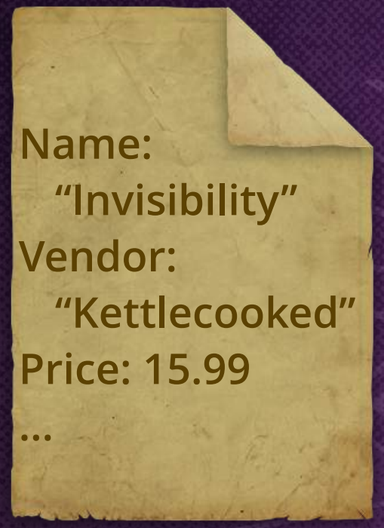
We can use comparison query operators to match documents based on the comparison of a specified value.

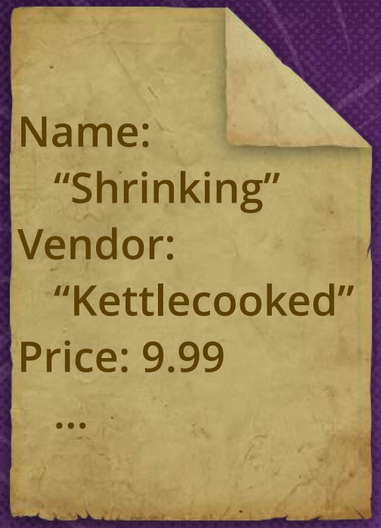
Common Comparisons

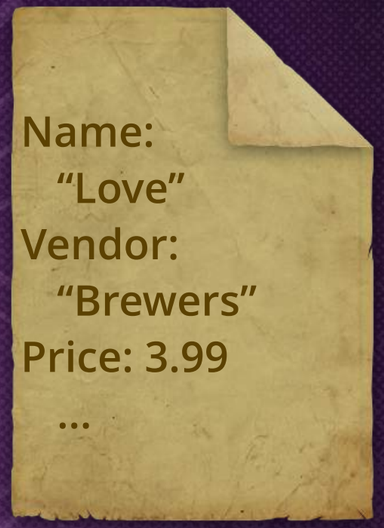
`$gt`	*greater than*	`$lt`	*less than*
`$gte`	*greater than or equal to*	`$lte`	*less than or equal to*
`$ne`	*not equal to*		

Finding Potions That Are Less Than $20

We can match the appropriate documents by using the ***$lt*** comparison operator.







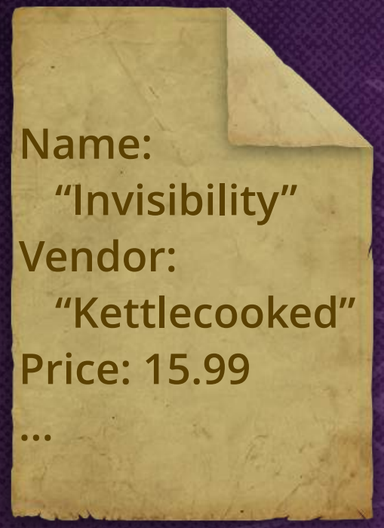
SHELL

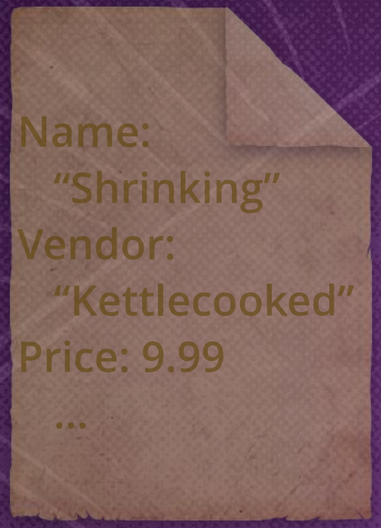
```
> db.potions.find({"price": {"$lt": 20}})
```

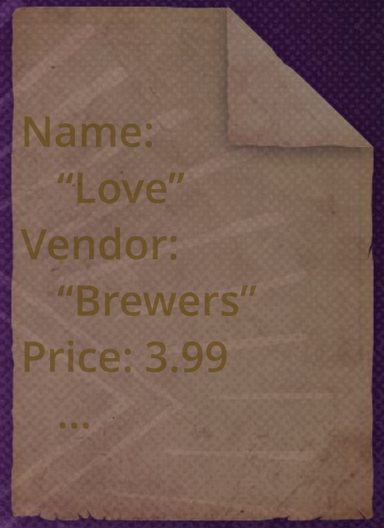
Price less than 20

Finding Potions Between Prices

We can query with a range by combining comparison operators.







SHELL

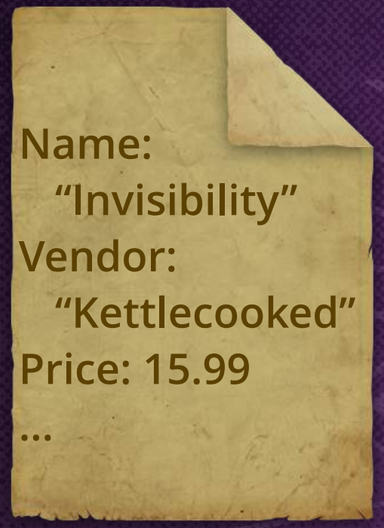
```
> db.potions.find({"price": {"$gt":10, "$lt": 20}})
```

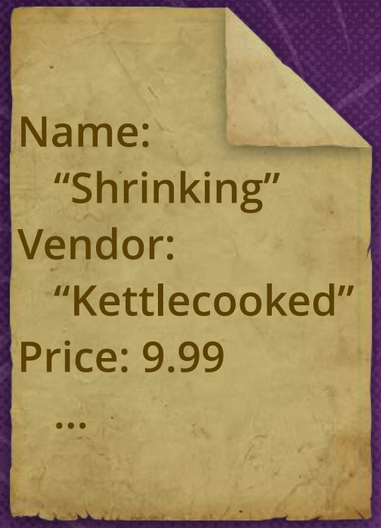
Price greater than 10 and less than 20

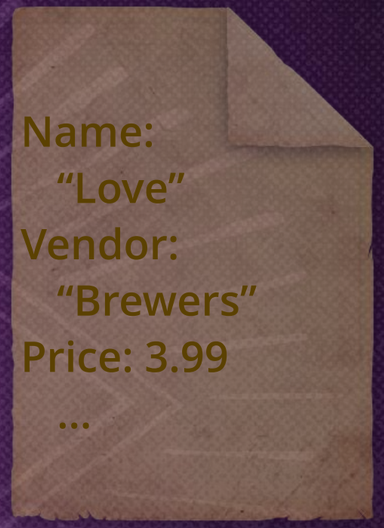
THE MAGICAL MARVELS OF MONGODB

Queries of Non-equality

We can use the ***$ne*** operator to find potions with fields that don't equal the specified value.







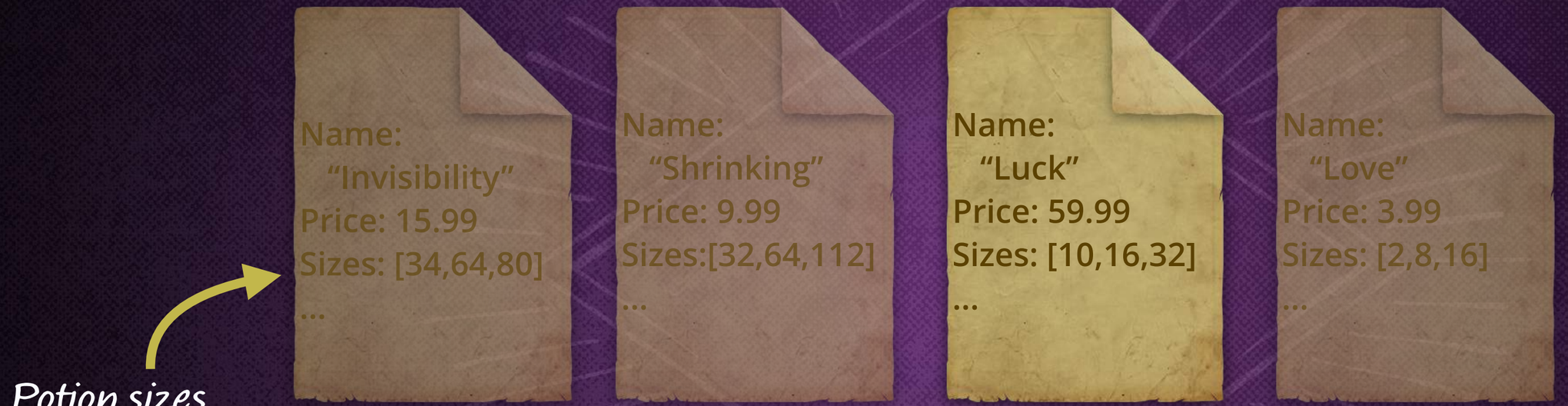
SHELL

```
> db.potions.find({"vendor": {"$ne": "Brewers"}})
```

Vendor not equal to "Brewers"

Range Queries on an Array

Each potion has a size field that contains an array of available sizes. We can use ***$elemMatch*** to make sure at least 1 element matches all criteria.



SHELL

```
> db.potions.find(
  {"sizes" : {"$elemMatch": {"$gt": 8, "$lt": 16}}}
 )
```

Be Careful When Querying Arrays With Ranges

What happens when we try to perform a normal range query on an array?

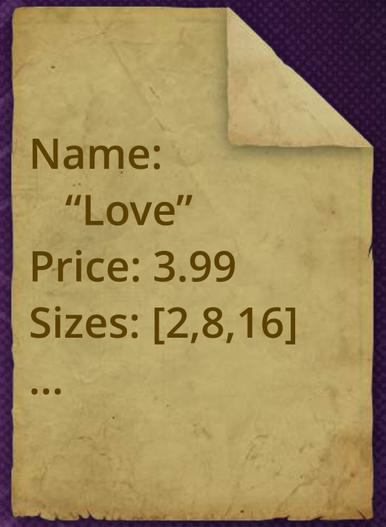
SHELL

```
> db.potions.find(
  {"sizes" : {"$gt": 8, "$lt": 16}}
 )
```

Doesn't contain any matching sizes, so why did it match?

Be Careful When Querying Arrays With Ranges

What happens when we try to perform a normal range query on an array?

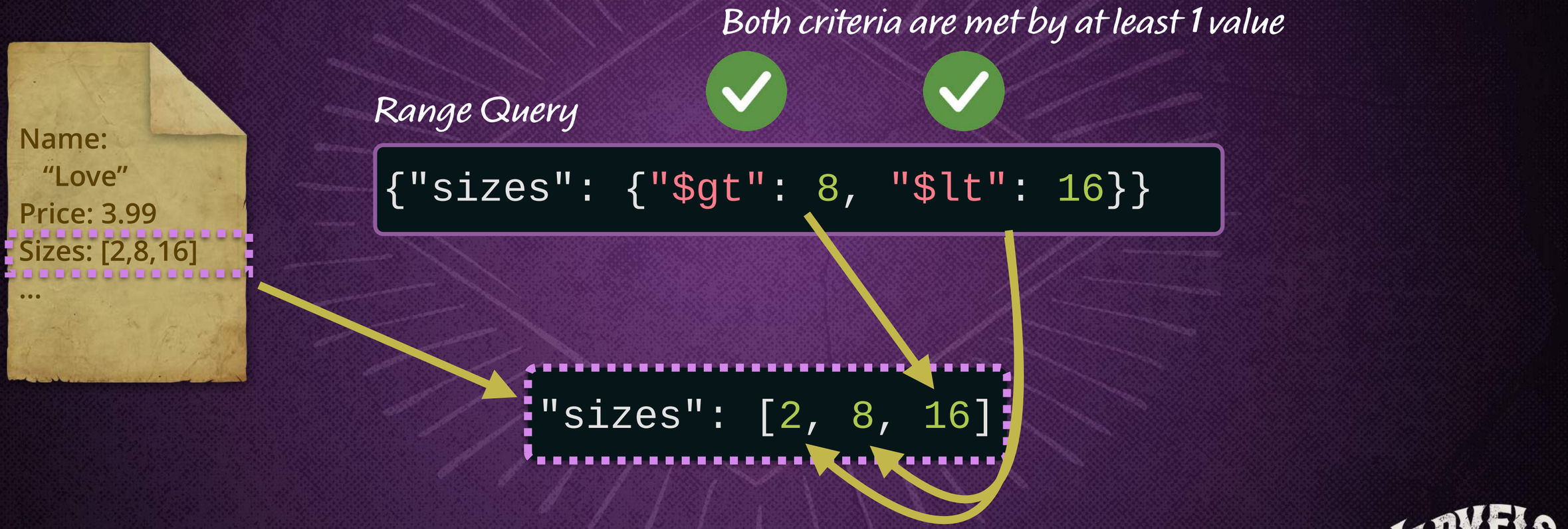


SHELL

```
> db.potions.find(
  {"sizes" : {"$gt": 8, "$lt": 16}}
 )
```

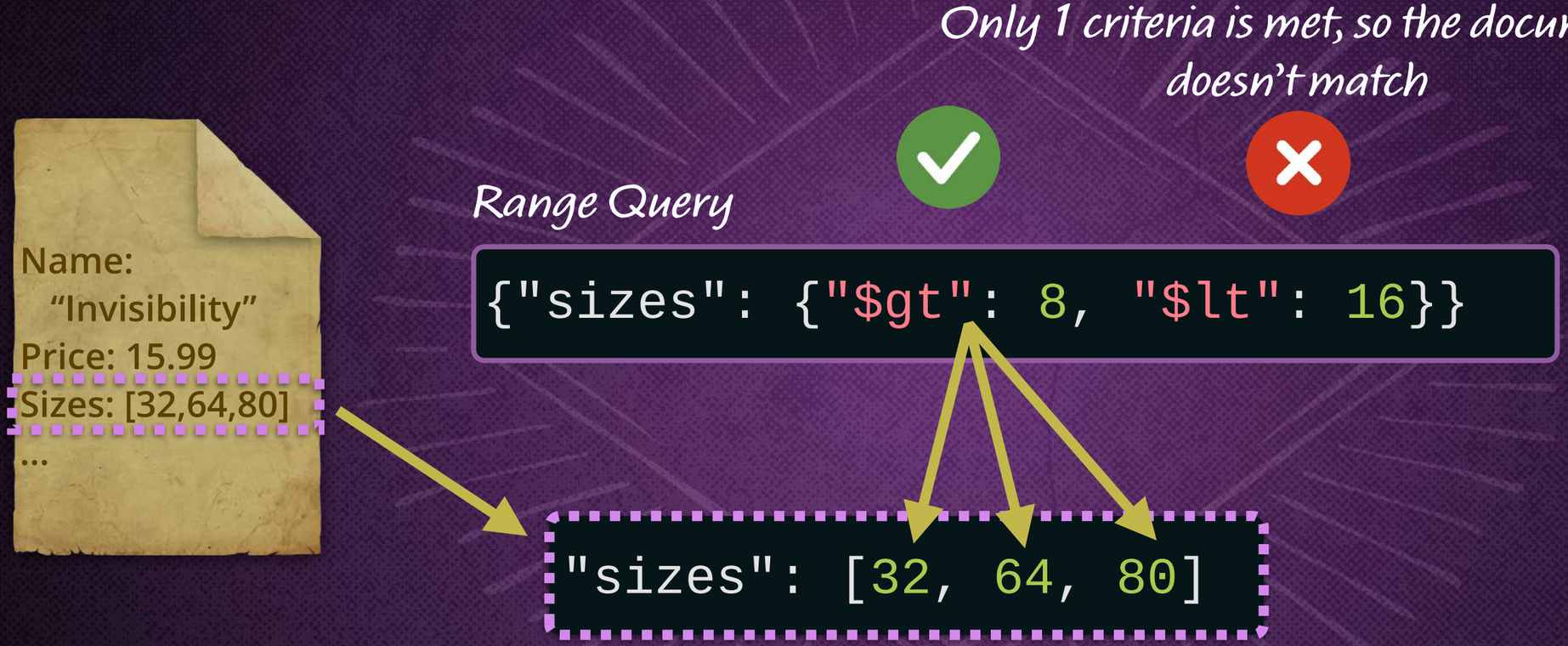
Why Did the Document Match?

Each value in the array is checked individually. If at least 1 array value is true for each criteria, the entire document matches.



Not Matching a Document

Conversely, the document will not match if only 1 criteria is met.



Materializing Potions

Level 3 – Section 2

Customizing Queries

Listing Our Best Potions

We're putting together a list of the best potions we've used. Let's find potions with a grade equal to or greater than 80.

Potions Collection

Need the name and vendor of potions with a high grade

Best potions

_____

_____

_____

THE MAGICAL MARVELS OF MONGODB

Introducing Projections

find() takes a second parameter called a “projection” that we can use to specify the exact fields we want back by setting their value to true.

SHELL

```
> db.potions.find(
  {"grade": {"$gte": 80}},
  {"vendor": true, "name": true}
)
{
  "_id": ObjectId(...),
  "vendor": "Kettlecooked",
  "name": "Shrinking"
}
...
```

When selecting fields, all other fields but the _id are automatically set to false

Only retrieve what’s needed

Excluding Fields

Sometimes you want all the fields except for a few. In that case, we can exclude specific fields.

SHELL

```
> db.potions.find(
  {"grade": {"$gte": 80}},
  {"vendor": false, "price": false}
)
{
  "_id": ObjectId(...),
  "name": "Shrinking",
  "grade": 94,
  "ingredients": [...],
  ...
}
```

When excluding fields, all fields but those set to false are defaulted to true

Great for removing sensitive data

Excluding the _id

The ***_id*** field is always returned whenever selecting or excluding fields. It's the only field that can be set to false when selecting other fields.

SHELL

```
> db.potions.find(
  {"grade": {"$gte": 80}},
  {"vendor": true, "price": true, "_id": false}
)
{
  "vendor": "Homebrewed",
  "price": 9.99
}
```

The only time we can mix an exclusion with selections

Removing the id is common when preparing data reports for non-developers.

THE MAGICAL MARVELS OF MONGODB

Either Select or Exclude Fields

Whenever projecting, we either select or exclude the fields we want — we don't do both.

SHELL

```
> db.potions.find(
  {"grade": {"$gte": 80}},
  {"name": true, "vendor": false}
)
```

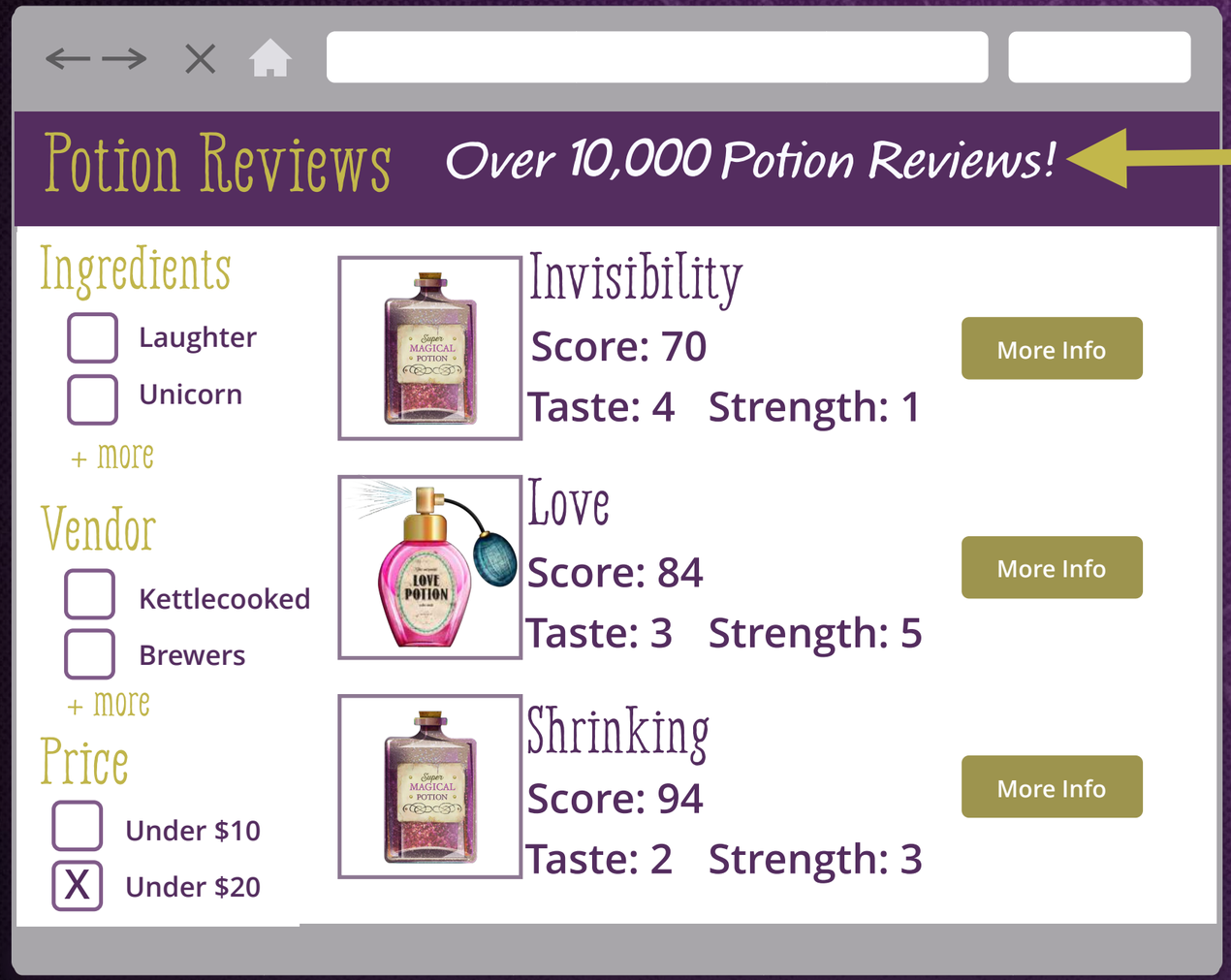
Causes an error to be raised

ERROR

```
"$err": "Can't canonicalize query: BadValue
Projection cannot have a mix of inclusion
and exclusion."
```

Counting Our Potions

Time to advertise our expertise and list the total number of potions we've reviewed.



Need to count the total number of potions in the potions collection

THE MAGICAL MARVELS OF MONGODB

Introducing the Cursor

Whenever we search for documents, an object is returned from the find method called a "cursor object."

SHELL

```
> db.potions.find({"vendor": "Kettlecooked"})
{"_id": ObjectId(...), ... }
{"_id": ObjectId(...), ... }
{"_id": ObjectId(...), ... }
...
```

First 20 documents

By default, the first 20 documents are printed out

Iterating Through the Cursor

When there are more than 20 documents, the cursor will iterate through them 20 at a time.

```
db.potions.find()
```

Sends 20 documents

SHELL

```
...
{"_id": ObjectId(...), "name": ... }
{"_id": ObjectId(...), "name": ... }
{"_id": ObjectId(...), "name": ... }
type "it" for more
```

THE MAGICAL MARVELS OF MONGODB

Continuing to Iterate Through the Cursor

Typing “it” will display the next 20 documents in the cursor.

```
db.potions.find()
```

Next batch sent

Iterates the cursor

We’ll continue being prompted until no documents are left

SHELL

```
> it
{"_id": ObjectId(...), "name": ... }
{"_id": ObjectId(...), "name": ... }
...
type "it" for more
```

THE MAGICAL MARVELS OF MONGODB

Cursor Methods

Since the cursor is actually an object, we can chain methods on it.

Method on cursor that returns the count of matching documents

Returns cursor object

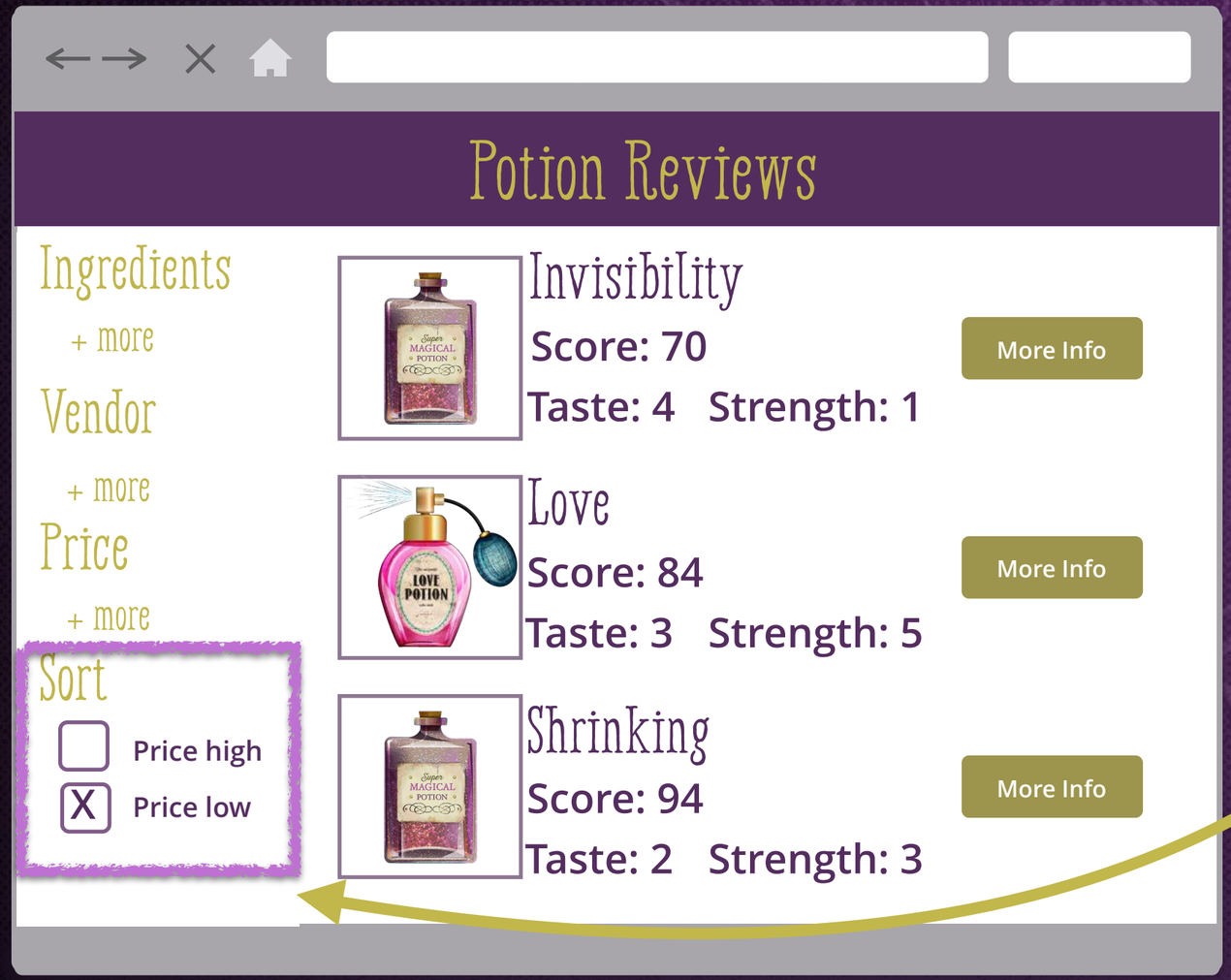
SHELL

```
> db.potions.find().count()
80
```

Cursor methods always come after find() since it returns the cursor object.

Sort by Price

We want to implement a way for users to sort potions by price.

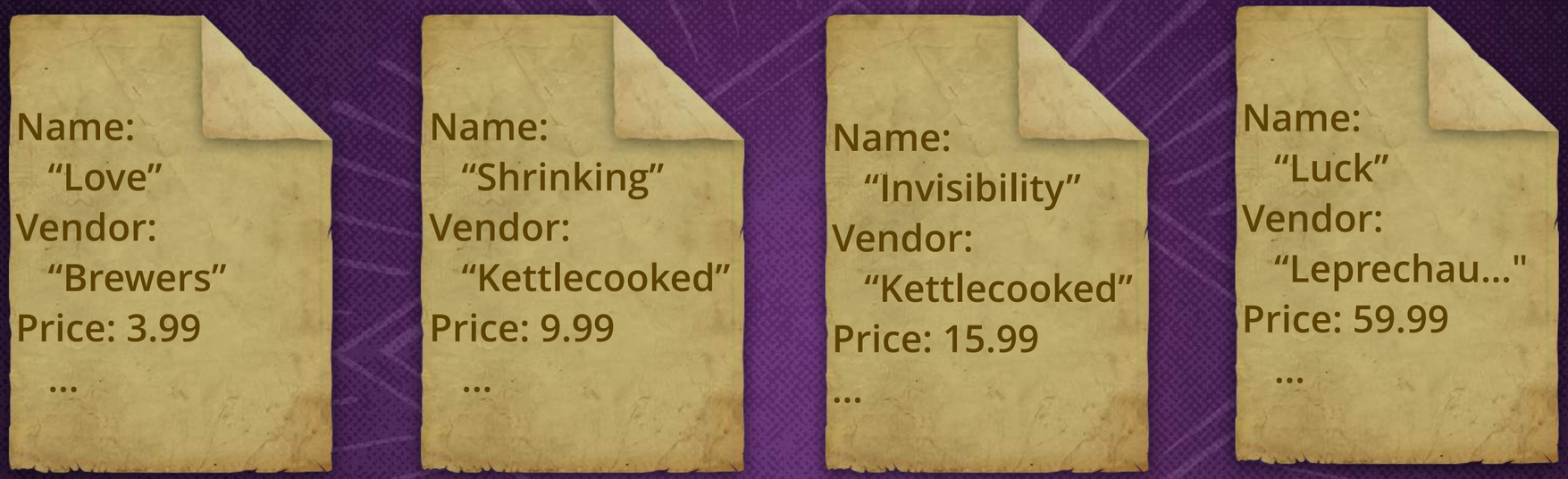


Sort potions with the lowest price first

THE MAGICAL MARVELS OF MONGODB

Sorting Potions

We can use the ***sort()*** cursor method to sort documents.



SHELL

```
> db.potions.find().sort({"price": 1})
```

Field to sort

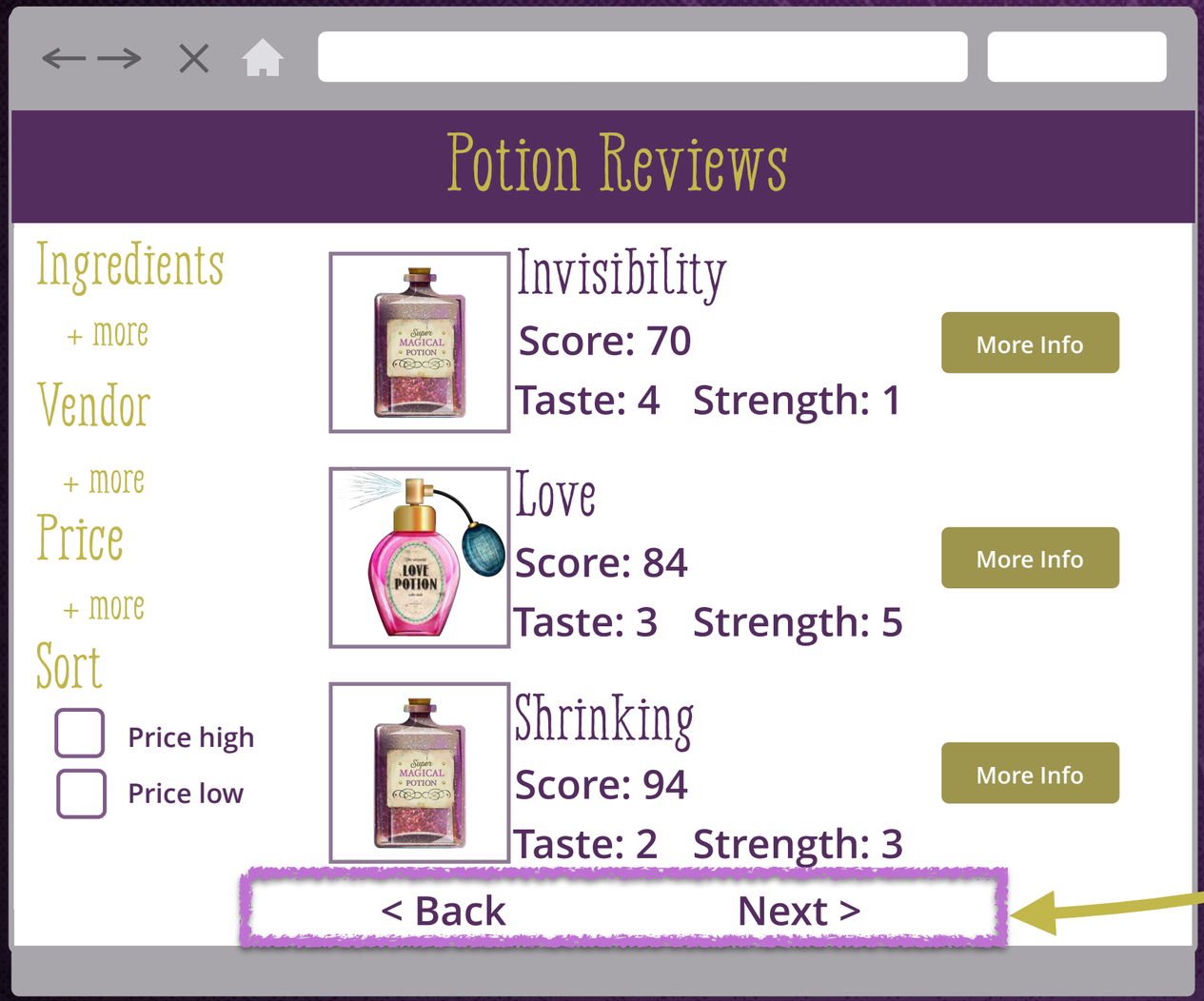
-1 to order descending

1 to order ascending

THE MAGICAL MARVELS OF MONGODB

Paginating the Potions Page

We only want to show 3 potions per page. Time to implement pagination!



Paginate results so we only see 3 potions on each page

Basic Pagination

We can implement basic pagination by limiting and skipping over documents. To do this, we'll use the ***skip()*** and ***limit()*** cursor methods.

Page 1

Skip 0, Limit 3

SHELL

```
> db.potions.find().limit(3)
```

Since we're not skipping, we can leave off the skip method and just limit 3

THE MAGICAL MARVELS OF MONGODB

Basic Pagination

We can implement basic pagination by limiting and skipping over documents.

SHELL

```
> db.potions.find().skip(3).limit(3)
```

THE MAGICAL MARVELS OF MONGODB

Basic Pagination

We can implement basic pagination by limiting and skipping over documents.

SHELL

```
> db.potions.find().skip(6).limit(3)
```

This approach can become really expensive with large collections.

THE MAGICAL MARVELS OF MONGODB

Morphing Models

Level 4 – Section 1

Data Modeling

Introducing User Profiles

We've added a user profile page to the site and want to allow users to enter up to 3 of their favorite potions.

User Document

```
{
    "_id": ObjectId(...),
    "username": "Azrius",
    "email": "azrius@example.com",
    "favorites": "Luck"
}
```

Need a way to store up to 3 potions

THE MAGICAL MARVELS OF MONGODB

Storing Favorites Within a User

Users and their favorites are strongly related and will be used together often.

User Document

```
{
    "_id": ObjectId(...),
    "username": "Azrius",
    "email": "azrius@example.com",
    "favorites": [
      "Newt Tonic",
      "Sleeping",
      "Love"
    ]
}
```

Use an array to hold multiple values

THE MAGICAL MARVELS OF MONGODB

Adding Vendor Information

We'd like to add more vendor information to our potions so users can be more informed about where they get their potions.

Vendor

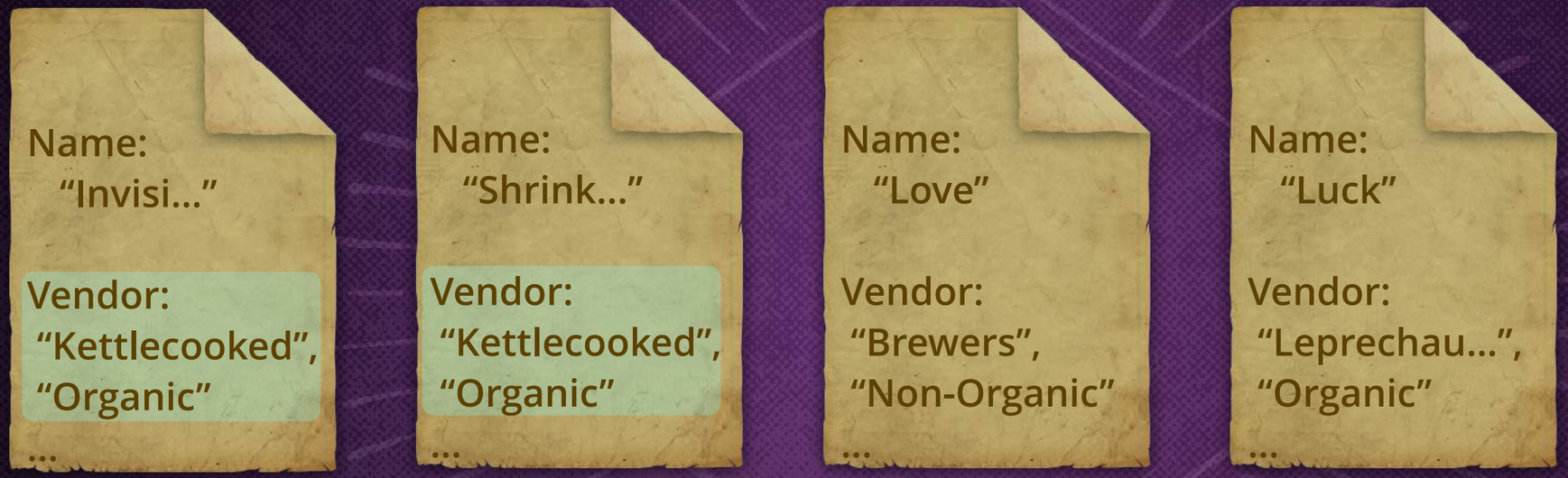
Inserting the data as an embedded document

Potion Document

```
{
  "_id": ObjectId(...),
  "name": "Invisibility",
  ...
  "vendor": {
    "name": "Kettlecooked",
    "phone": 5555555555,
    "organic": true
  }
}
```

Storing Vendor Information

Each potion now contains its vendor information, which means we have vendor information repeated throughout our documents.

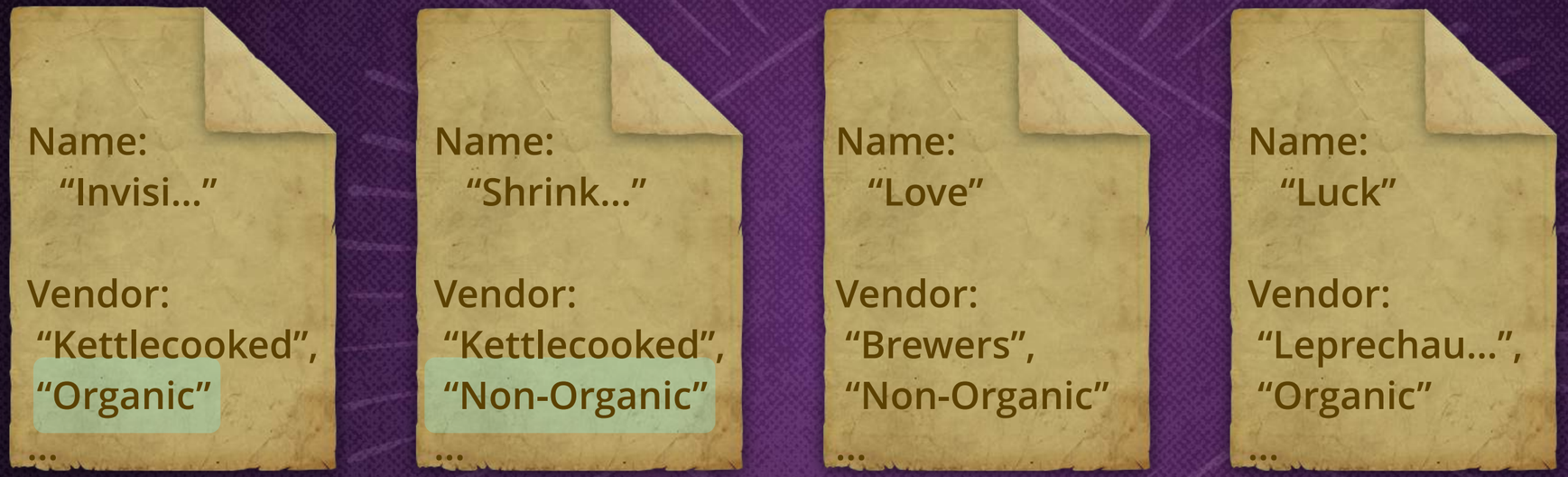


We're going to have duplicate vendor information for each potion.

THE MAGICAL MARVELS OF MONGODB

Dangers of Duplication

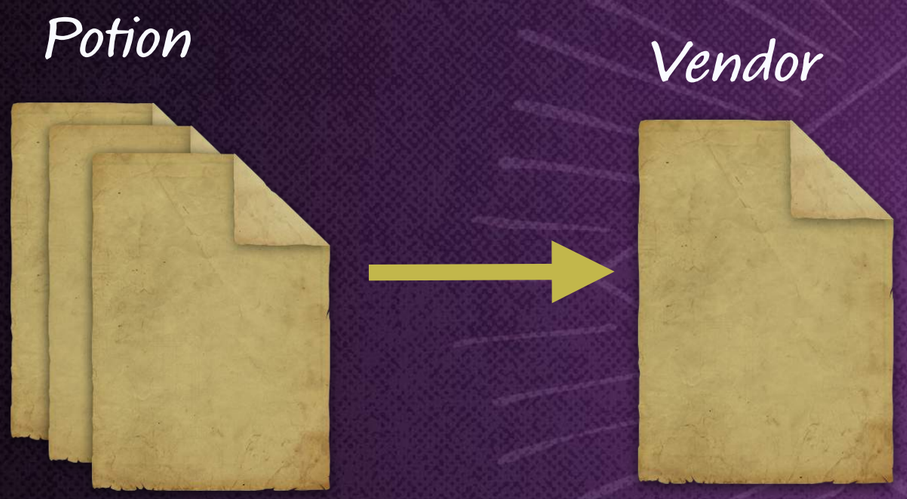
Duplicate data can be hard to keep consistent throughout the database.



If 1 potion gets updated with new information and the rest don't, our data is no longer correct

Referencing Vendor Information

Instead of embedding the vendor information, we can create a vendors collection and reference the vendor document in each potion document.



Vendor names are unique and don't change

Potion

```
{
  "_id": ObjectId(...),
  "name": "Invisibility",
  "vendor_id": "Kettlecooked",
  ...
}
```

Vendor

```
{
  "_id": "Kettlecooked",
  "phone": 5555555555,
  "organic": true
}
```

Inserting Referenced Documents

SHELL

```
> db.vendors.insert({
    "_id": "Kettlecooked",
    "phone": 5555555555,
    "organic": true
})
```

We can specify the unique _id of document

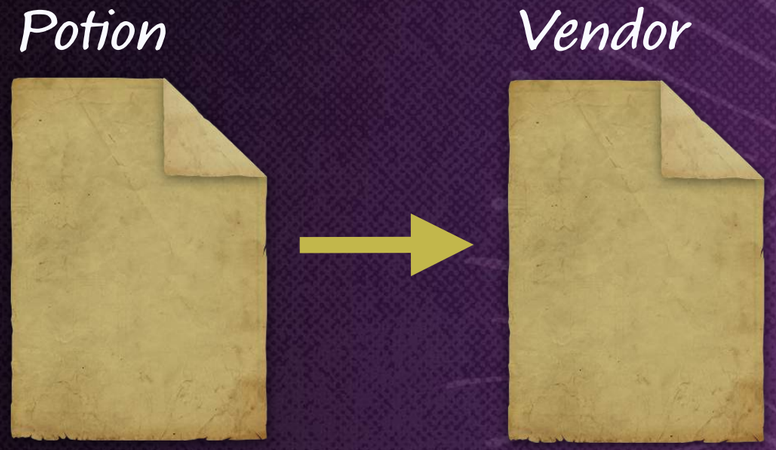
SHELL

```
> db.potions.insert({
    "name": "Invisibility",
    "vendor_id": "Kettlecooked"
    ...
})
```

Referenced document

Querying a Referenced Document

In order to pull a potion document and the vendor information, we must first query for the potion to get the vendor_id and then query once more to get their vendor information.




```
{
  "_id": ObjectId(...),
  "name": "Invisibility",
  "vendor_id": "Kettlecooked",
  ...
}
```

First, query to retrieve potion information

```
db.potions.find({"name": "Invisibility"})
```

Second, query to retrieve vendor information

```
db.vendors.find({"_id": "Kettlecooked"})
```

We get the vendor _id from the first query

Some Features of Embedded Documents

With a single query, we can grab a user's email and their addresses

Data readily available

SHELL

```
db.users.find({},{"email": true, "favorites": true})
```

```
{
    "_id": ObjectId(...),
    "email": "azrius@example.com"
    "favorites": [
      "Newt Tonic",
      "Sleeping",
      "Love"
     ]
}
```

More Features of Embedded Documents

Data readily available

SHELL

```
db.users.find({},{"email": true, "favorites": true})
```

Atomic write operations

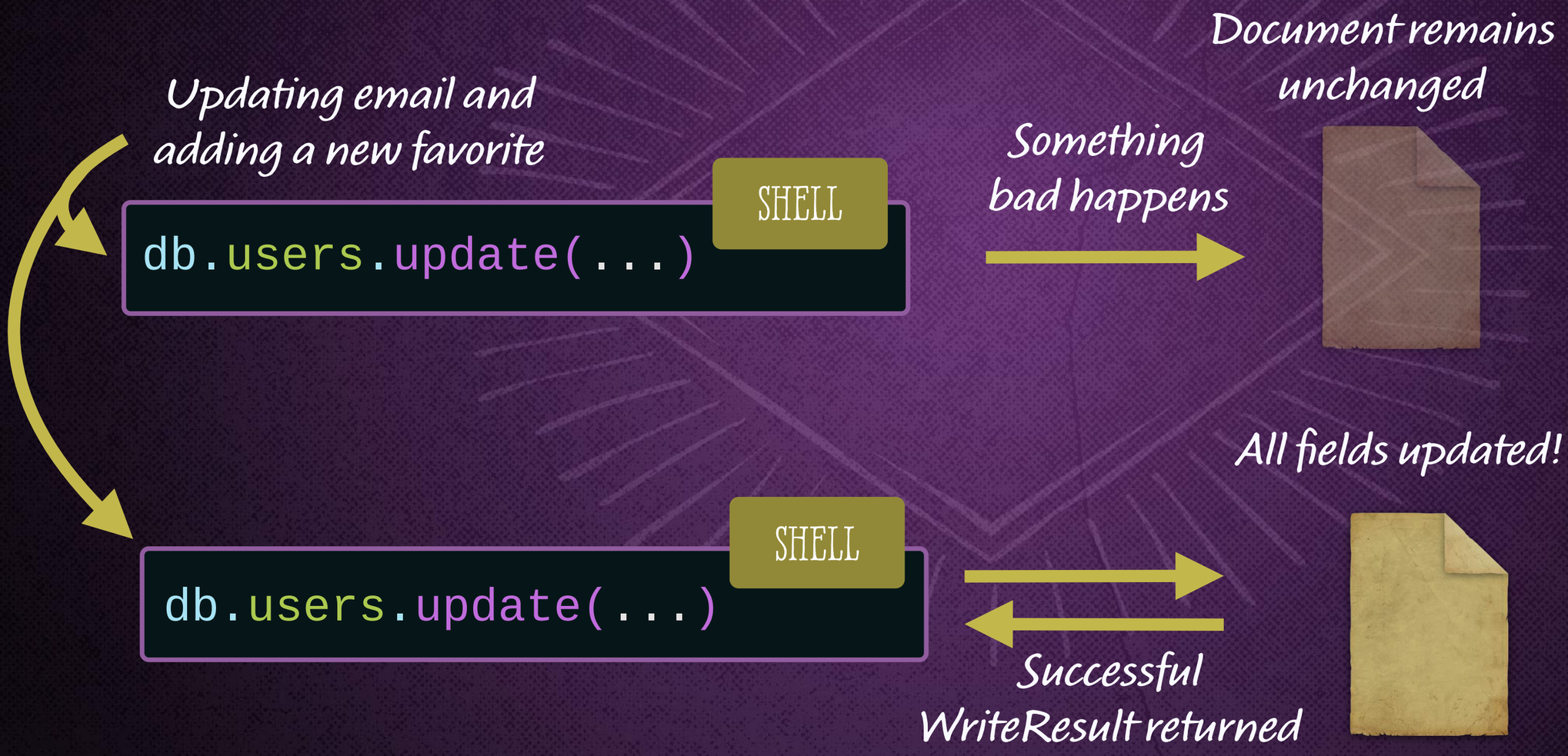
SHELL

```
db.users.insert({
  "username": "Azrius",
  "email": "azrius@example.com",
  "favorites": ["Newt...", "Sleeping", "Love"]
})
```

Guarantee that the document write completely happens or doesn't at all

Why Does Atomicity Matter?

If we update a user's email and add a favorite potion, but an error occurs in the favorites portion of the update, then none of the update will occur.



Referenced Documents Exist Independently

Since we've referenced vendor documents, they now exist entirely on their own outside of potion documents.

One place to edit vendor information

```
db.vendors.update({"_id": ObjectId(...)},{...})
```

Multi-document writes not supported

New Potion

SHELL

```
db.potions.insert({...})
```

New Potion's Vendor

SHELL

```
db.vendors.insert({...})
```

No guarantee that both write operations will occur

Multi-document Write Operations

Currently, MongoDB doesn't support multi-document writes. We aren't able to guarantee write operations on an atomic level.

Adding new potion

```
db.potions.insert(...)
```

SHELL

Successful write

New potion created!

Adding the new vendor

```
db.vendors.insert(...)
```

SHELL

Error occurs

No vendor document created

Dangers of Not Having Transactions

MongoDB doesn't recognize document relationships, so we now have a potion with a nonexistent vendor.

New potion created!

No vendor document created

Vendor never got created!

Potion document

```
{
  "_id": ObjectId(...),
  "name": "Time Travel",
  "vendor_id": "Magical Inc.",
  ...
}
```

The references still exist!

Morphing Models

Level 4 – Section 2

Data Modeling Decisions

Choosing Which Route to Take

When deciding how to model a one-to-many relationship, we must carefully consider how our data will be used in order to decide between embedding and referencing.

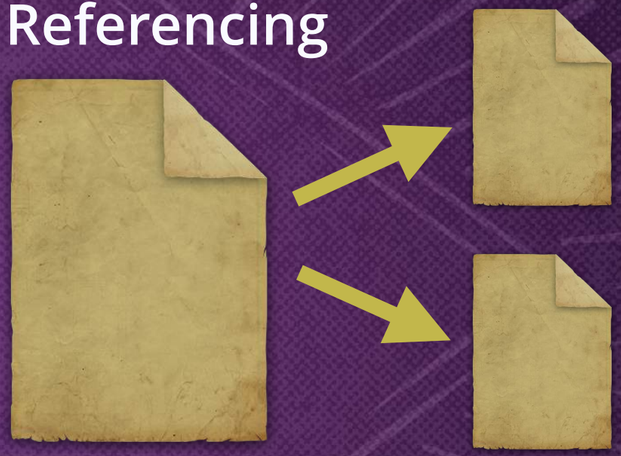
Embedding

Single query

Documents accessed through parent

Atomic writes

Referencing



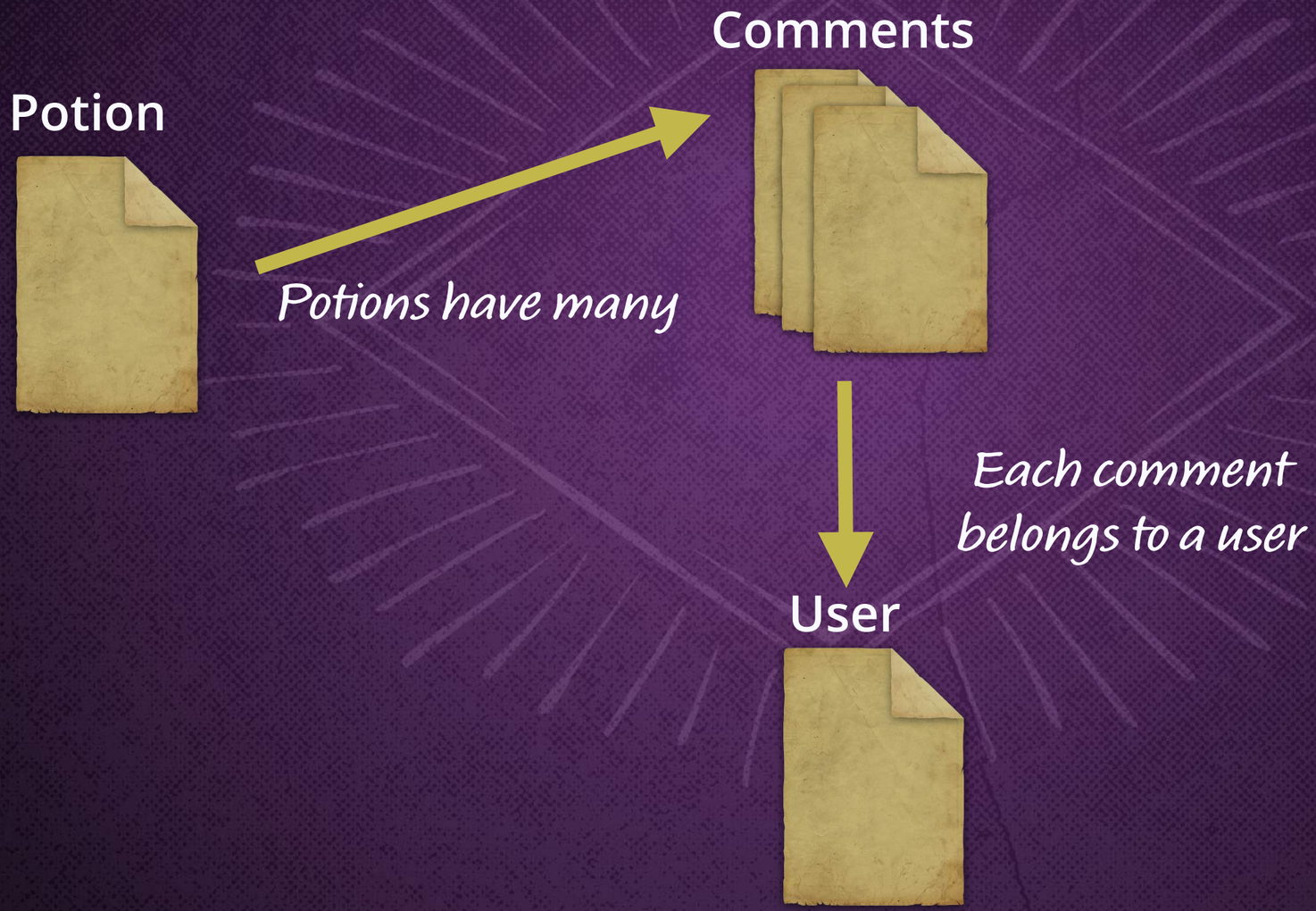
Requires 2 queries

Documents exist independently

Doesn't support multi-document writes

Adding Comments to Potions

We'd like to allow users to comment on the potions and need to determine the best route.



First Question: How Will the Data Be Used?

Data that is frequently used together will benefit from being embedded while data that's rarely used can afford the cost of referencing.

How often is the data used together?	Always	Sometimes	Rarely
Embed	✓	✓	✓
Reference		✓	✓

Embedding will work most of the time

Either option can work effectively

Data Usage: Potion Comments

Embedding

Referencing

We would be forced to perform multiple queries

Whenever we display potions, we'll always want to display comments

Username displayed for each comment

THE MAGICAL MARVELS OF MONGODB

Second Question: What's the Size of the Data?

The size of the data in a relationship has a significant impact on data modeling. It's crucial to think ahead!

Expected Size	Less than 100	More than a few hundred	Thousands
Embed	✓	✓	
Reference		✓	✓

Might start to see a decline in read performance when embedded

Data Size: Potion Reviews

Embedding

Referencing

When the data size is over 100, we can consider referencing

Most potions won't get more than 50 comments, and each comment has a single author

THE MAGICAL MARVELS OF MONGODB

Third Question: Will the Data Change Often?

Sometimes embedding data can lead to data duplication. Depending on whether or not the duplicate data changes a lot can factor into our decision making.

Frequency of Change	Never/Rarely	Occasionally	Constantly
Embed	✓	✓	
Reference		✓	✓

Prevents inconsistencies from duplication

Data duplication is okay if we don't expect change

Depends on whether or not we want the overhead of managing duplication

Data Change: Potion Comments

Embedding

Referencing

Requires updating all of a user's comments if their information changes

Embedding

Referencing

Embedding Comments in Potions

We can confidently embed comments within potions. We know we'll have less than 100 comments per potion, they're used together often, and the data doesn't change often.

```
{
   "name": "Invisibility",
   ...
   "comments":[
     {
      "title": "The best potion!",
      "body": "Lorem ipsum abra cadrbra"
     },
     ...
   ]
}
```

Referencing Users in Comments

We only need the username from each user, and embedding user documents could lead to duplication issues. Let's reference a user by using his or her unique username.

```
{
    "name": "Invisibility",
    ...
    "comments":[
      {
       "title": "The best potion!",
       "body": "Lorem ipsum abra cadrbra",
       "user_id": "Azrius"
      },
      ...
    ]
}
```

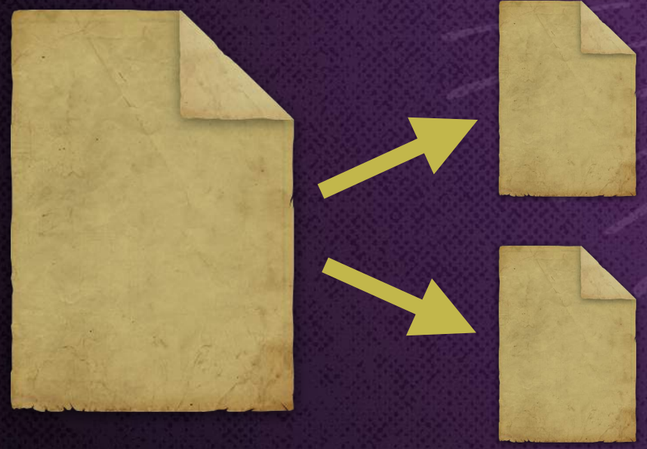
Usernames can't be changed and are unique

Data Modeling Guidelines and Takeaways

Generally, embedding is the best starting point

Focus on how your data will be used

If you find you need complex references, consider a relational database



Reference data when you need to access document independently

Consider referencing when you have large data sizes

THE MAGICAL MARVELS OF MONGODB

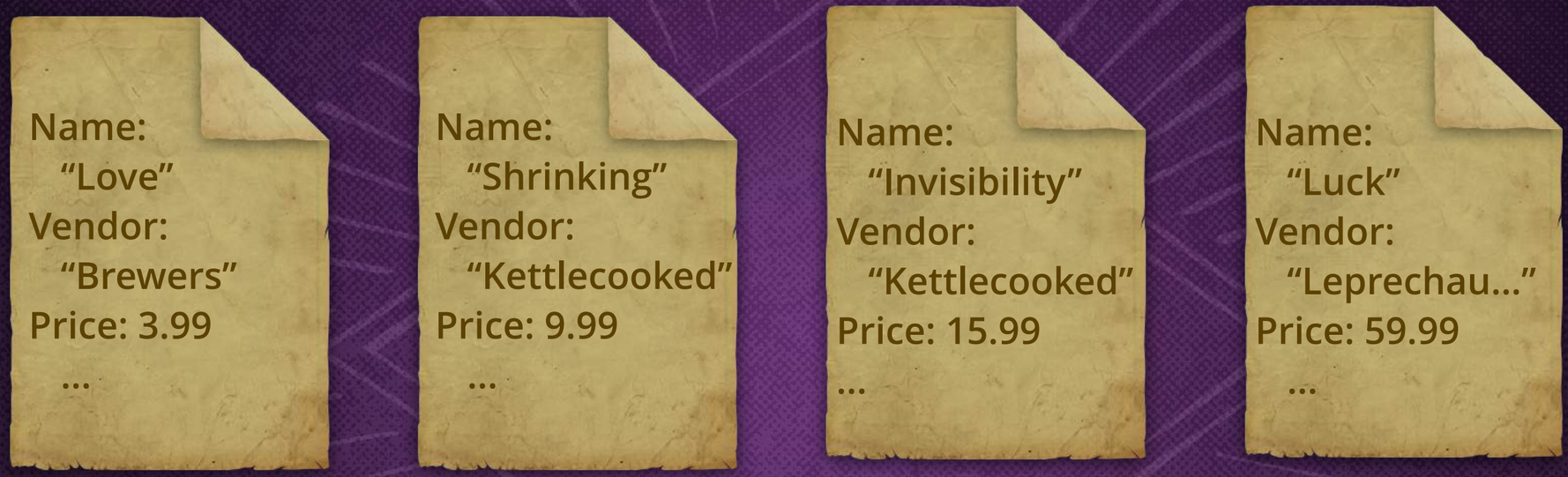
Aggregation Apparitions

Level 5 – Section 1

Common Aggregations

Finding the Number of Potions Per Vendor

Time for an audit! We need to know how many potions we have per vendor.



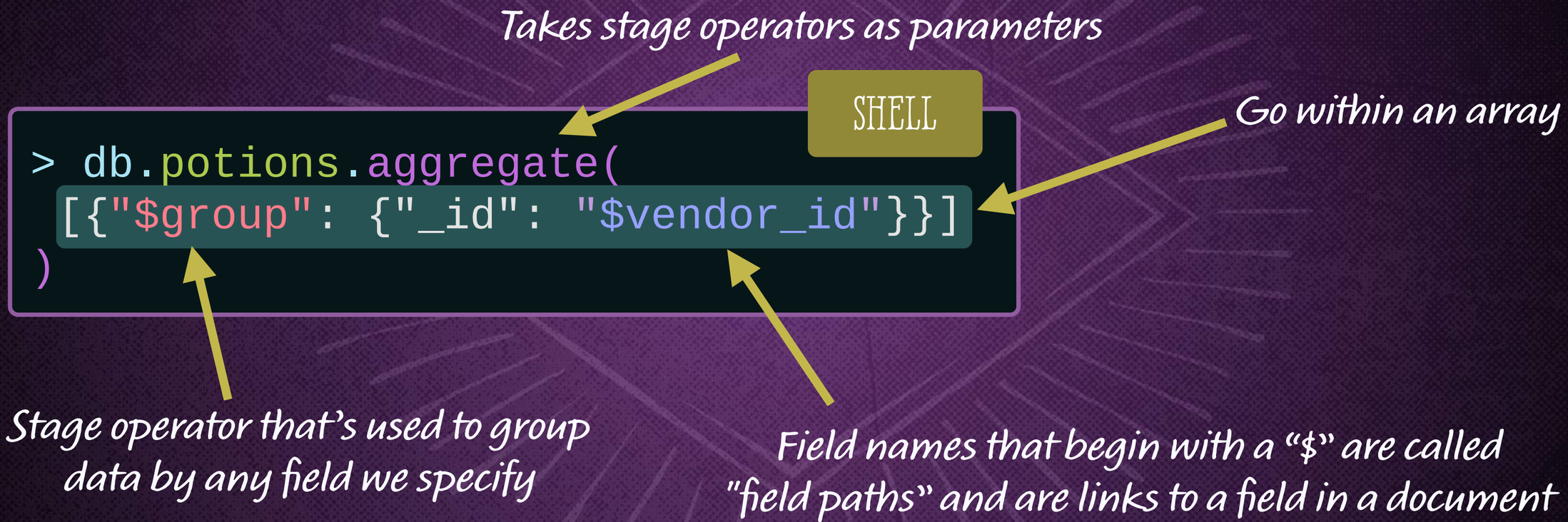

```
> db.potions.find({}, {"name": true, "vendor": true})
```

We could manually pull all the data and count everything, but it's better to have MongoDB handle that for us!

THE MAGICAL MARVELS OF MONGODB

Introducing the Aggregation Framework

The aggregation framework allows for advanced computations.



"Aggregate" is a fancy word for combining data.

Using the Aggregation Framework to Group Data

SHELL

```
> db.potions.aggregate(
  [{"$group": {"_id": "$vendor_id"}}]
)
```

This is known as the "group key" and is required

Returns result object containing the unique vendors in the inventory collection

Potions Collection

$group

Results

```
{"_id": "Kettlecooked"},
{"_id": "Brewers"},
{"_id": "Leprechaun Inc"}
```

Using Accumulators

Anything specified after the group key is considered an accumulator. Accumulators take a single expression and compute the expression for grouped documents.

SHELL

```
> db.potions.aggregate([
 {"$group": {"_id": "$vendor_id", "total": {"$sum": 1}}}
])
```

Group key

Will add 1 for each matching document

Accumulator

Inventory

$group

Results

Total number of documents per vendor!

```
{"_id": "Kettlecooked", "total": 2},
{"_id": "Brewers", "total": 1,},
{"_id": "Leprechaun Inc”, "total": 1}
```

Field Paths Vs. Operators

When values begin with a "$", they represent field paths that point to the value

SHELL

```
> db.potions.aggregate([
 {"$group": {"_id": "$vendor_id", "total": {"$sum": 1}}}
])
```

When fields begin with a "$", they are operators that perform a task

Summing the Grade Per Vendor

SHELL

```
> db.potions.aggregate([
   { "$group": {
                    "_id": "$vendor_id",
                    "total": {"$sum": 1},
                    "grade_total": {"$sum": "$grade"}
                }
   }
])
```

Field path

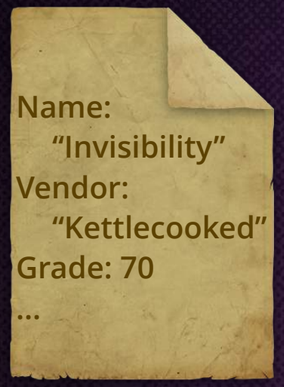
Sums the grade values for potions in their group

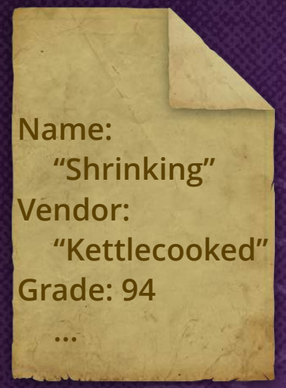
Second accumulator

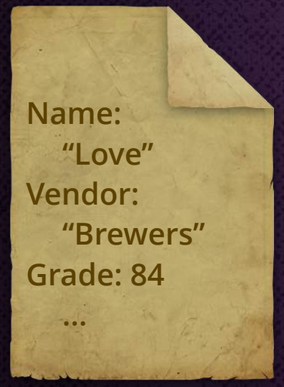
Results

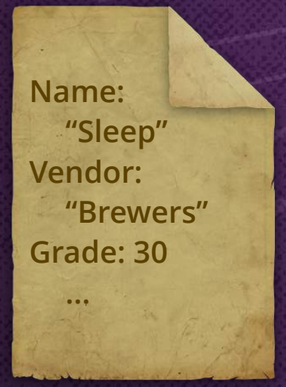
```
{"_id": "Kettlecooked", "total": 2, "grade_total": 400},
{"_id": "Brewers","total: 1, "grade_total": 340},
{"_id": "Leprechaun Inc”, "total": 1, "grade_total": 92}
```

Averaging Potion Grade Per Vendor









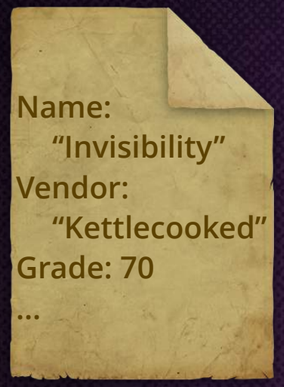
SHELL

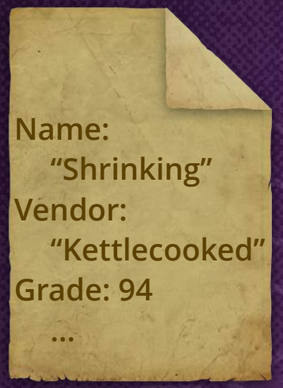
```
> db.potions.aggregate([
 { "$group": {
     "_id": "$vendor_id",
     "avg_grade": {"$avg": "$grade"}
   }
 }
])
```

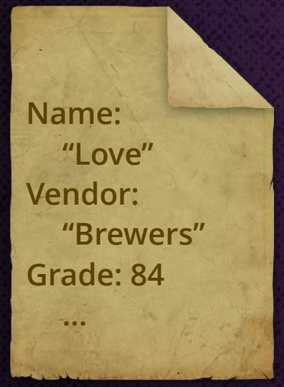
Results

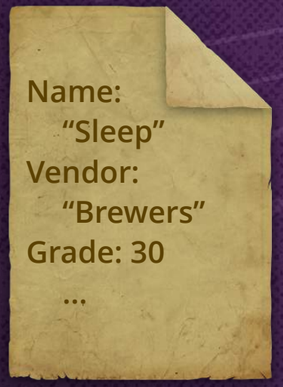
```
{"_id": "Kettlecooked", "avg_grade": 82 },
{"_id": "Brewers","avg_grade": 57 }
```

Returning the Max Grade Per Vendor









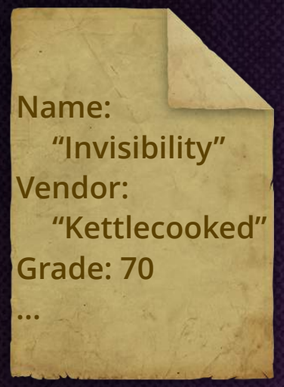
SHELL

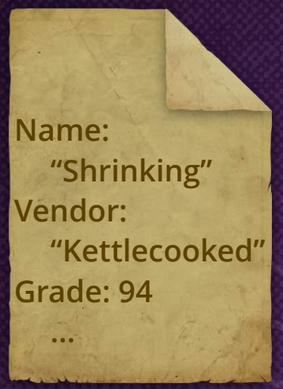
```
> db.potions.aggregate([
 { "$group": {
     "_id": "$vendor_id",
     "max_grade": {"$max": "$grade"}
   }
 }
])
```

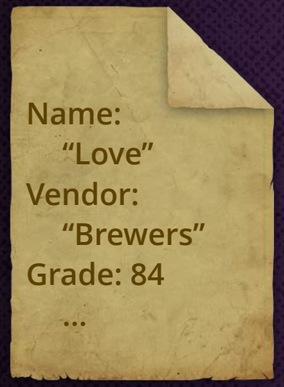
Results

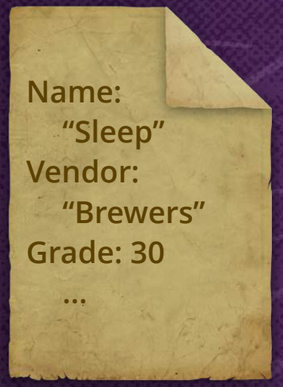
```
{"_id": "Kettlecooked", "max_grade": 94},
{"_id": "Brewers","max_grade": 84}
```

Using $max and $min Together









SHELL

```
> db.potions.aggregate([
  { "$group": {
      "_id": "$vendor_id",
      "max_grade": {"$max": "$grade"},
      "min_grade": {"$min": "$grade"}
    }
  }
])
```

We can use the same field in multiple accumulators

Results

```
{"_id": "Kettlecooked", "max_grade": 94, "min_grade": 70 },
{"_id": "Brewers", "max_grade": 84, "min_grade": 30 }
```

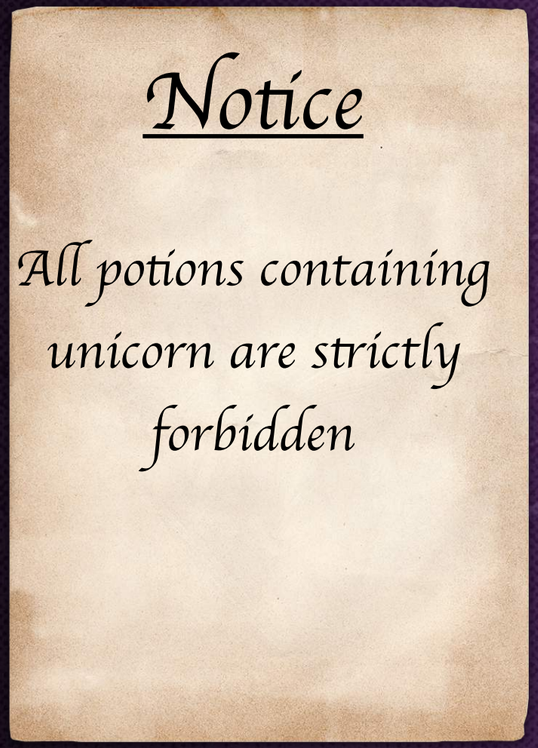
Aggregation Apparitions

Level 5 – Section 2

The Aggregation Pipeline

Pulling Conditional Vendor Information

It turns out that potions made with unicorn aren't permitted, so we need to count the number of potions per vendor that contain it.



Steps to find potions with unicorns:

1) Query potions

2) Group by vendor

3) Sum the number of potions per vendor

THE MAGICAL MARVELS OF MONGODB

Introducing the Aggregation Pipeline

The aggregate method acts like a pipeline, where we can pass data through many stages in order to change it along the way.

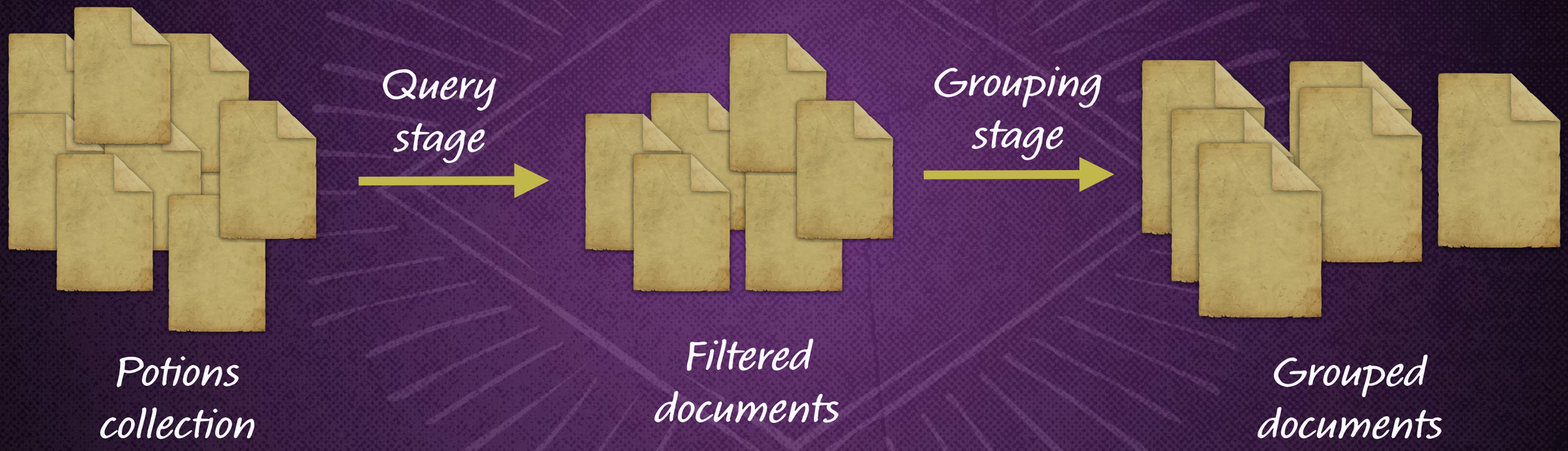
```
db.potions.aggregate([stage, stage, stage])
```

We've already seen what a stage looks like in the last section

```
db.potions.aggregate([
 {"$group": {"_id": "$vendor_id", "total": {"$sum": 1}}}
])
```

How the Pipeline Works

Each stage modifies the working data set and then passes the altered documents to the next stage until we get our desired result.



Using the $match Stage Operator

$match is just like a normal query and will only pass documents to the next stage if they meet the specified condition(s).

SHELL

```
db.potions.aggregate([
 {"$match": {"ingredients": "unicorn"}}
])
```

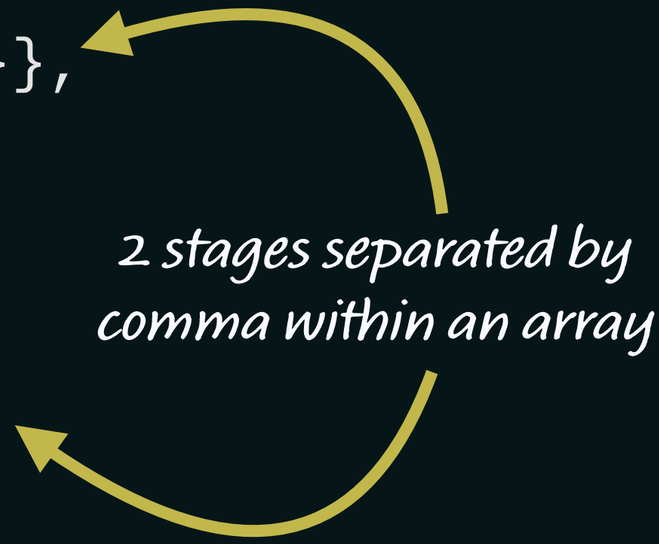
We can use the same query we would use with find()

Use match early to reduce the number of documents for better performance.

Grouping Potions With Data

SHELL

```
db.potions.aggregate([
 {"$match": {"ingredients": "unicorn"}},
 {"$group":
   {
     "_id": "$vendor_id",
     "potion_count": {"$sum": 1}
   }
 }
])
```



Result

```
{"_id": "Poof", "potion_count": 20},
{"_id": "Kettlecooked","potion_count": 1}
```

Top 3 Vendors With Potions Under $15

1) Query for potions with a price less than 15

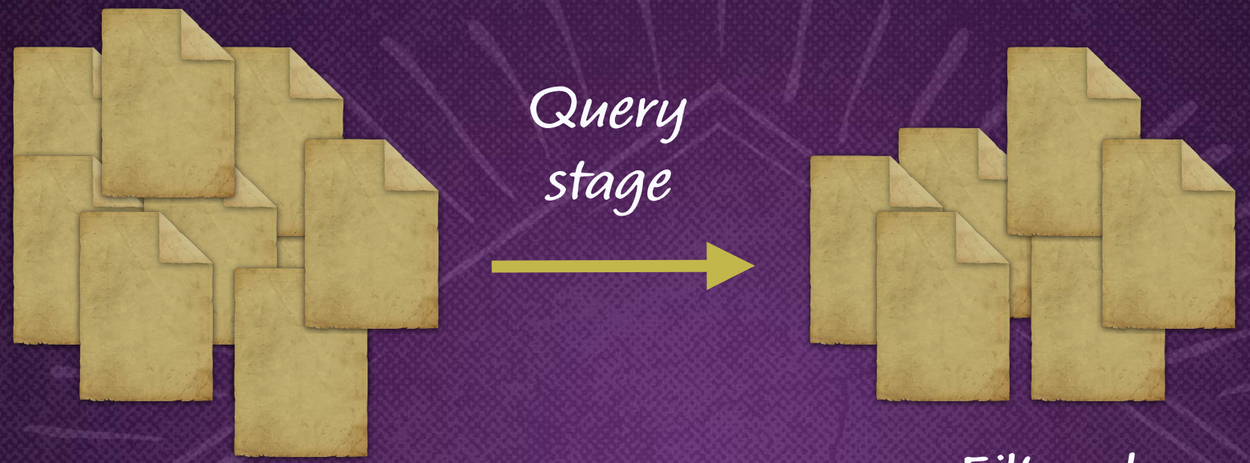
2) Group potions by vendor and average their grades

3) Sort the results by grade average

4) Limit results to only 3 vendors

THE MAGICAL MARVELS OF MONGODB

Matching Potions Under $15



SHELL

```
db.potions.aggregate([
 {"$match": {"price": {"$lt": 15}}}
])
```

It's good practice to limit the number of results early on

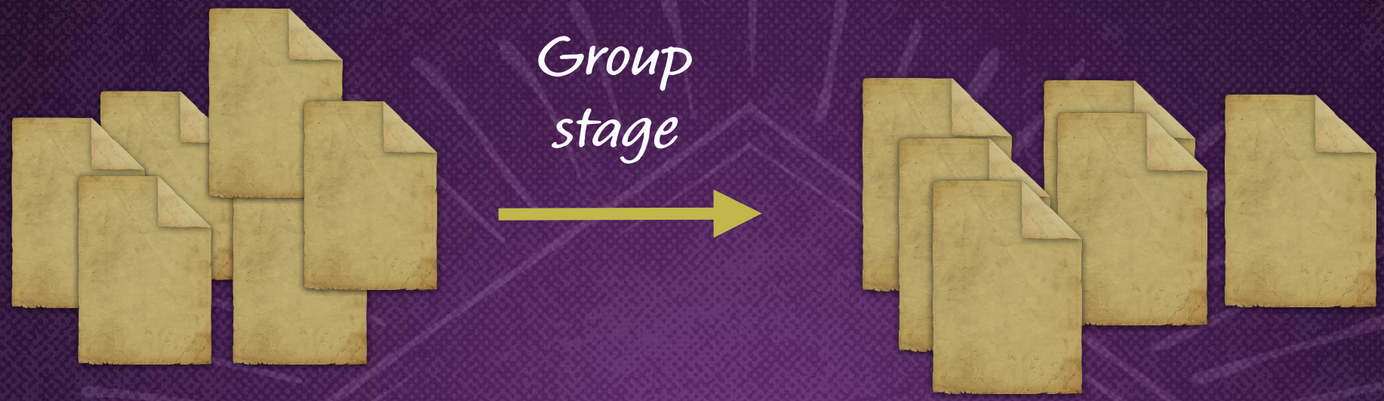
Matching Potions Under $15

Filtered documents

SHELL

```
db.potions.aggregate([
 {"$match": {"price": {"$lt": 15}}}
])
```

Grouping Potions by Vendor



SHELL

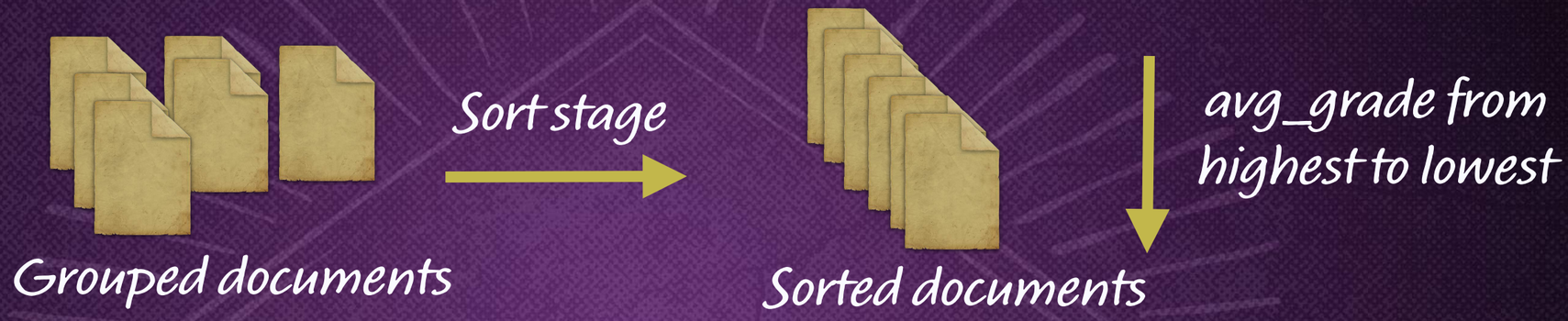
```
db.potions.aggregate([
 {"$match": {"price": {"$lt": 15}}},
 {"$group": {"_id": "$vendor_id","avg_grade": {"$avg": "$grade"}}}
])
```

Vendor name as group key

Average of potion grades

Sorting Vendors by Average Grade

We can sort potions by using the ***$sort*** stage operator.



SHELL

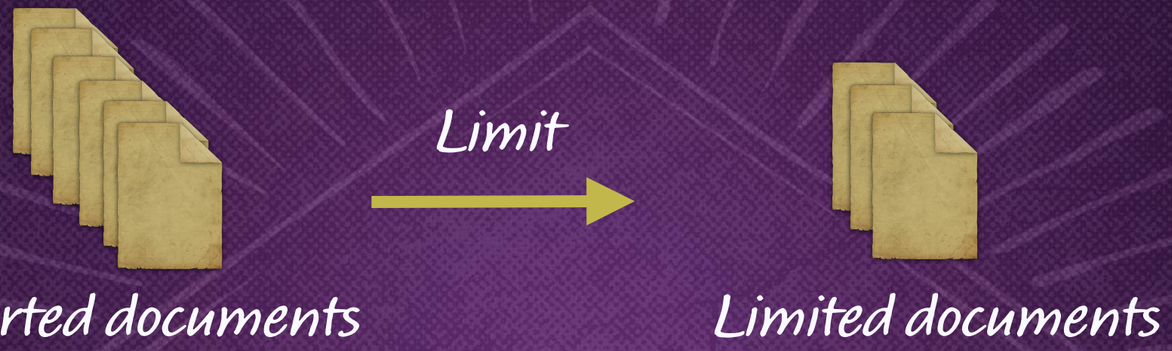
```
db.potions.aggregate([
 {"$match": {"price": {"$lt": 15}}},
 {"$group": {"_id": "$vendor_id","avg_grade": {"$avg": "$grade"}}},
 {"$sort": {"avg_grade": -1}}
])
```

We can sort the field we created during the group stage

Sort by the field in descending order

Limiting the Number of Documents

We can use the ***$limit*** stage operator to limit the number of documents that get passed on.



SHELL

```
db.potions.aggregate([
  {"$match": {"price": {"$lt": 15}}},
  {"$group": {"_id": "$vendor_id","avg_grade": {"$avg": "$grade"}}},
  {"$sort": {"avg_grade": -1}},
  {"$limit": 3}
])
```

Specify the number of documents to limit

Optimizing the Pipeline

It’s best practice to only send the needed data from stage to stage. Let’s see what data we really need.

SHELL

```
db.potions.aggregate([
 {"$match": {"price": {"$lt": 15}}},
 {"$group": {"_id": "$vendor_id","avg_grade": {"$avg": "$grade"}}},
 {"$sort": {"avg_grade": -1}},
 {"$limit": 3}
])
```

Only need vendor and grade for each potion after the match stage

Projections While Aggregating

We can limit the fields we send over by using ***$project***, which functions the same way as projections when we're querying with ***find()***.

SHELL

```
db.potions.aggregate([
 {"$match": {"price": {"$lt": 15}}},
 {"$project":{"_id": false, "vendor_id": true, "grade": true}},
 {"$group": {"_id": "$vendor_id","avg_grade": {"$avg": "$grade"}}},
 {"$sort": {"avg_grade": -1}},
 {"$limit": 3}
])
```

We want to use $project as soon as possible

Vendor and grade for each potion after the match stage

It's common to see $match and $project used together early on and throughout the pipeline.

Aggregation Results

SHELL

```
db.potions.aggregate([
 {"$match": {"price": {"$lt": 15}}},
 {"$project":{"_id": false, "vendor_id": true, "grade": true}},
 {"$group": {"_id": "$vendor_id","avg_grade": {"$avg": "$grade"}}},
 {"$sort": {"avg_grade": -1}},
 {"$limit": 3}
])
```

```
 {"_id": "Kettlecooked", "avg_grade": 99 },
 {"_id": "Leprechaun Inc", "avg_grade": 95 },
 {"_id": "Brewers", "avg_grade": 90 }
```

Group key

Average potion grade