



LESSON

# Querying Data with the MongoDB Aggregation Framework

Google slide deck available [here](#)

This work is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)  
(CC BY-NC-SA 3.0)

## Overview



### Learning Objectives

At the end of this lesson, learners will be able to:

- Use the stages \$match, \$project, \$unwind, \$facet, and \$merge to enhance query functionalities with MQL.
- Identify how to use the aggregation framework to conduct complex queries in MongoDB.
- Describe the purpose of the aggregation framework.

### Suggested Uses

- A whole lesson spread out across multiple lecture periods
- Handouts / asynchronous learning
- Supplemental reading material - read on your own / not part of formal teaching
- Complement to University courses [Introduction to MongoDB](#), [MongoDB for SQL Professionals](#), and [MongoDB Aggregation](#)

This lesson is a part of the courses [MongoDB Aggregation Framework](#) and [Introduction to Modern Databases with MongoDB](#).

### At a Glance



Length:  
120 minutes



Level:  
Intermediate



Prerequisites: [The MongoDB Aggregation Framework](#) and [The MongoDB Query Language \(MQL\)](#)

This work is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#)  
(CC BY-NC-SA 3.0)

Share your feedback: We hope these curriculum materials will be a valuable resource for you and your learners. Let us know how the materials work for you, what we can improve on, and how MongoDB for Academia can support you via our brief [feedback form](#).

MongoDB for Academia: MongoDB for Academia offers resources for educators and students to support teaching and learning MongoDB. Check out our [educator resources](#) and join the Educator Community. Students can receive \$50 in Atlas credits and free certification through the [GitHub Student Developer Pack](#).

Last Update: March 2025

# This lesson includes exercises



Follow along using these tools

## Create a Database

- [MongoDB Atlas](#) (cloud)
- [MongoDB Community](#) (local install)

## Connect to Your Database

- [MongoDB Shell](#) (open source)

For more instructions on how to use MongoDB Atlas with your students, see [Atlas for Educators](#)

These exercises were designed to work in the MongoDB Shell. If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).



# \$match

We previously covered MQL or the MongoDB Query Language in our lessons. It's provides a means of interacting with data in a single collection.

The Aggregation Framework extends what can be done with data beyond the capabilities of MQL. It provides a framework to perform complex data processing on the documents through a series of stages.

In this lesson, we'll explore more about the Aggregation Framework and what it can provide you in terms of functionality.

Let's look at the \$match stage as this is typically used when querying data and it's a good place to start.



## Aggregation Framework \$match

```
db.<collection>.aggregate({ $match: { <field>:<value> } })
```

Match uses filter document

More complex queries are possible with aggregation expressions

```
db.collection.aggregate({ $match: { $expr: { <expr>: { <field1>: <value1> } } } })
```

Use as early as possible in the pipeline, ideally as the first stage. This will improve the performance of the pipeline

If \$match is the first stage in the pipeline, then indexes can be used

We've already introduced the \$match stage in our last aggregation lesson but let's quickly recap the stage here.

It is similar to the MQL find() syntax using a field and a value for comparison matching.



## Aggregation Framework \$match

```
db.<collection>.aggregate({ $match: { <field>:<value> } })
```

Match uses filter document

More complex queries are possible with aggregation expressions

```
db.collection.aggregate({ $match: { $expr: { <expr>: { <field1>: <value1> } } } }).
```

Use as early as possible in the pipeline, ideally as the first stage. This will improve the performance of the pipeline.

If first stage in the pipeline, can use indexes

It uses a query filter document to define the expressions that limit the query to the subset of results you want to return.

The query filter document is a similar to standard JSON and consists of field-value/key-value expressions.

If you use an empty query filter document {} then like find() it will return all of the documents in the collection.



## Aggregation Framework \$match

```
db.<collection>.aggregate({ $match: { <field>:<value> } })
```

Match uses filter document

More complex queries are possible with aggregation expressions

```
db.collection.aggregate({ $match: { $expr: { <expr>: { <field1>: <value1> } } } })
```

Use as early as possible in the pipeline, ideally as the first stage. This will improve the performance of the pipeline

If first stage in the pipeline, can use indexes

The \$match stage does not take 'raw' aggregation expressions, instead you need to use the \$expr operator for these. This allows the use of all the aggregation expressions in the filtering criteria for the \$match stage.



## Aggregation Framework \$match

```
db.<collection>.aggregate({ $match: { <field>:<value> } })
```

Match uses filter document

More complex queries are possible with aggregation expressions

```
db.collection.aggregate({ $match: { $expr: { <expr>: { <field1>: <value1> } } } })
```

Use as early as possible in the pipeline, ideally as the first stage. This will improve the performance of the pipeline

If first stage in the pipeline, can use indexes

It is recommended by MongoDB that you use the \$match stage as early as possible in an Aggregation Framework pipeline, ideally as the first stage.

\$match will limit the total number of documents being processed by the aggregation pipeline so the earlier it is used, the more processing it minimizes in the pipeline.





## Aggregation Framework \$match

```
db.<collection>.aggregate({ $match: { <field>:<value> } })
```

Match uses filter document

More complex queries are possible with aggregation expressions

```
db.collection.aggregate({ $match: { $expr: { <expr>: { <field1>: <value1> } } } })
```

Use as early as possible in the pipeline, ideally as the first stage. This will improve the performance of the pipeline

If first stage in the pipeline, can use indexes

**\$match will be able to use indexes if it is in the first stage**

# Quiz





# Quiz

Which of the following are true for the \$match aggregation stage in MongoDB? More than one answer choice can be correct.

- ☐ A. \$match uses a filter document to specify the match criteria
- ☐ B. \$match can use 'raw' aggregation expressions
- ☐ C. \$match should be placed early in the pipeline
- ☐ D. Indexes can be used by \$match if it is the first stage of the pipeline

# Quiz



Which of the following are true for the \$match aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$match uses a filter document to specify the match criteria
- ☐ B. \$match can use 'raw' aggregation expressions
- ☒ C. \$match should be placed early in the pipeline
- ☒ D. Indexes can be used by \$match if it is the first stage of the pipeline

CORRECT: \$match uses a filter document to specify the match criteria -

INCORRECT: \$match can use 'raw' aggregation expressions -

CORRECT: \$match should be placed early in the pipeline -

CORRECT: Indexes can be used by \$match if it is the first stage of the pipeline -

# Quiz



Which of the following are true for the \$match aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$match uses a filter document to specify the match criteria
- ☐ B. \$match can use 'raw' aggregation expressions
- ☒ C. \$match should be placed early in the pipeline
- ☒ D. Indexes can be used by \$match if it is the first stage of the pipeline

*This is correct.  
\$match uses a document to specify the criteria used for matching.*

CORRECT: \$match uses a filter document to specify the criteria used for matching.

# Quiz



Which of the following are true for the \$match aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$match uses a filter document to specify the match criteria
- ☐ B. \$match can use 'raw' aggregation expressions
- ☒ C. \$match should be placed early in the pipeline
- ☒ D. Indexes can be used by \$match if it is the first stage of the pipeline

*This incorrect. The \$match stage cannot use 'raw' aggregation expression, it can however use the \$expr operator which can itself use 'raw' aggregation expressions.*

INCORRECT: The \$match stage cannot use 'raw' aggregation expression, it can however use the \$expr operator which can itself use 'raw' aggregation expressions.

# Quiz



Which of the following are true for the \$match aggregation stage in MongoDB? More than 1 answer choice can be correct.

- ☒ A. \$match uses a filter document to specify the match criteria
- ☐ B. \$match can use 'raw' aggregation expressions
- ☒ C. \$match should be placed early in the pipeline
- ☒ D. Indexes can be used by \$match if it is the first stage of the pipeline

*This is correct. Placing the \$match stage early in the pipeline will help reduce the processing required for the aggregation and is recommended practice.*

**CORRECT:** This is correct. Placing the \$match stage early in the pipeline will help reduce the processing required for the aggregation and is recommended practice.

# Quiz



Which of the following are true for the \$match aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$match uses a filter document to specify the match criteria
- ☐ B. \$match can use 'raw' aggregation expressions
- ☒ C. \$match should be placed early in the pipeline
- ☒ D. Indexes can be used by \$match if it is the first stage of the pipeline

*This is correct. It is possible to use indexes with the \$match stage but only if it is the first stage of the pipeline.*

**CORRECT:** This is correct. It is possible to use indexes with the \$match stage but only if it is the first stage of the pipeline.





# \$match Stage: Exercise

# Monthly Budget: Example Document

Let's focus on a monthly budget example and look at one document:

```
{ "_id" : 1, "category" : "food",  
  "budget": 400, "spent": 450 }
```



Let's take a few minutes to explore Aggregation and the \$match stage, we'll walk through the syntax, create the data and then query the database to get the results. Follow along using the [MongoDB Shell](#). If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#). In this exercise, we'll look at example using monthly budgets and expenses.



# \$match Stage: Exercise

Let's focus on the Aggregation Framework syntax:

```
db.monthlybudget.aggregate([
  {
    $match: {
      $expr: { $gt:
        [ "$spent",
          "$budget" ] }
    }
  }
])
```

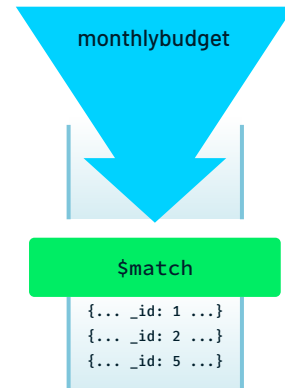
\$expr for aggregation expressions

Let's look at the syntax we'll need in the Aggregation Framework with \$match to find such documents.

We can see the use of \$expr in the \$match stage. This allows the use of aggregation expressions as we noted \$match can't use these expression in the 'raw' and must use them with the \$expr operator.

In order to determine which categories had an overspend, we can compare documents and look for documents where the "spent" field is greater than the "budget" field. This will answer our question, to find where there were monthly budget overspends.

```
[
  {
    $match: {
      $expr: {
        $gt: [
          "$spent",
          "$budget" ]
        }
      }
    }
  ]
```



We can see this aggregation stage broken down into the stage \$match, the \$expr operator to allow the use of aggregation expressions and the use of the comparison greater than \$gt expression to compare two fields in a given document to check for budget overspends.

# \$match Stage: Exercise



Let's insert some real data on a budget!

```
>>> db.monthlybudget.insertMany([{"_id" : 1, "category" : "food", "budget": 400,
"spent": 450 }, {"_id" : 2, "category" : "drinks", "budget": 100, "spent": 150
}, {"_id" : 3, "category" : "clothes", "budget": 100, "spent": 50 }, {"_id" :
4, "category" : "misc", "budget": 500, "spent": 300 }, {"_id" : 5, "category" :
"travel", "budget": 200, "spent": 650 }])

...
{ "acknowledged" : true, "insertedIds" : [ 1, 2, 3, 4, 5 ] }
```

You should cut and paste the following command directly from the slide or from these notes into the prompt (indicated by >>>). Once they have been inserted you will see the following output on the screen. The result will be the same as we are explicitly setting the ObjectIds for these documents.

```
db.monthlybudget.insertMany([{"_id" : 1, "category" :
"food", "budget": 400, "spent": 450 }, {"_id" : 2,
"category" : "drinks", "budget": 100, "spent": 150 }, {"
_id" : 3, "category" : "clothes", "budget": 100, "spent":
50 }, {"_id" : 4, "category" : "misc", "budget": 500,
"spent": 300 }, {"_id" : 5, "category" : "travel",
"budget": 200, "spent": 650 }])
```

See: <https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/>

# \$match Stage: Exercise



## Results

```
db.monthlybudget.aggregate({ $match: { $expr: {  
  $gt: [ "$spent" , "$budget" ] } } } )  
  
{ "_id" : 1, "category" : "food", "budget" : 400, "spent" : 450 }  
{ "_id" : 2, "category" : "drinks", "budget" : 100, "spent" : 150 }  
{ "_id" : 5, "category" : "travel", "budget" : 200, "spent" : 650 }
```

Let's just recap the aggregation pipeline again, we are using \$match with \$expr to allow us to use the aggregation expression \$gt (greater than). We're checking for documents where the value in the "spent" field is greater than the "budget" field indicating an overspend.

Unlike the MongoDB Query Language we can add more processing at this point by simply adding more stages to the current pipeline which only have a single stage with \$match.

We have the first result being an overspend of 50 for the category "food"  
Then we have another overspend of 50 for the category "drinks".  
Finally, we have an overspend of 450 for the category "travel".



## \$match Stage: Exercise

Find where 500+ was spent and which overspent.

Using the same window, change **<a>** to the field representing what was spent.

Change **<b>** to the value necessary to find all people whose age is greater than 500.

```
>>> db.monthlybudget.aggregate({ $match: { $and: [ { $expr: {  
  $gt: [ "$spent" , "$budget" ] } } ], { <a>: { <b>: 500 } } ] }  
  })  
  
{ "_id" : 5, "category" : "travel", "budget" : 200, "spent" : 650 }
```

It's your turn to find documents whose spend is greater than (\$gte or \$ge) 500, you will need to change **<a>** to the operator for greater than or greater than equal and you will need to change **<b>** to the field which represents what was spent for the category.

The result in the code block is what you should see if you are successful.



\$project



# Aggregation Framework \$project



```
db.<collection>.aggregate({ $project: { <specification(s)> } })
```

Project specification document

Aggregation expressions can reset an existing field or add a new field

```
db.collection.aggregate({ $project: { $field: <expr> } })
```

Include or exclude specific fields from the document

```
db.collection.aggregate({ $project: { $field: <1 or true> } })
```

We've already introduced the \$project stage in our last aggregation lesson but let's quickly recap the stage here.

This stage allows for the reshaping of a document so fields can be included, excluded, reset or new fields can be added.

# Aggregation Framework \$project



```
db.<collection>.aggregate({ $project: { <specification(s)> } })
```

## Project specification document

Aggregation expressions can reset an existing field or add a new field

```
db.collection.aggregate({ $project: { $field: <expr> } })
```

Include or exclude specific fields from the document

```
db.collection.aggregate({ $project: { $field: <1 or true> } })
```

The \$project specification document can specify the inclusion or the exclusion of fields, the suppression of the `_id` field, the addition of new fields, and the resetting of values for existing fields.

# Aggregation Framework \$project



```
db.<collection>.aggregate({ $project: { <specification(s)> } })
```

Project specification document

Aggregation expressions can reset an existing field or add a new field

```
db.collection.aggregate({ $project: { $field: <expr> } })
```

Include or exclude specific fields from the document

```
db.collection.aggregate({ $project: { $field: <1 or true> } })
```

The \$project can make use of aggregation expression, specifically these can be used to reset the value of an existing field in the document or to add a new field to the document.

# Aggregation Framework \$project



```
db.<collection>.aggregate({ $project: { <specification(s)> } })
```

Project specification document

Aggregation expressions can reset an existing field or add a new field

```
db.collection.aggregate({ $project: { $field: <expr> } })
```

Include or exclude specific fields from the document

```
db.collection.aggregate({ $project: { $field: <1 or true> } })
```

It is possible to include or to exclude specific fields from the document as it passes through the \$project stage. To include a field you can specify 1 or true, whilst to exclude a field you can specify 0 or false.



# Budget Item: Example Document

Let's focus on the budget item example and look at one document:

```
{ "_id" : 1, "category" : "travel", "expense":  
  "airplane", "spent": 250, "budget": 200,  
  details: { "airline": "United", "flight_type":  
    "return", "seat_class": "economy"} }
```

Fields of  
interest



Let's take a few minutes to explore the \$project stage with a hands-on session, we'll walk through the syntax, create the data and then query the database to get the results.

Follow along using the [MongoDB Shell](#). If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).

This builds on what we did for our previous session with the \$match stage in this lesson.

Let's firstly look at a sample document for an single budget item example to see what data is contained that will help us create a query to find where we spend more money than budgeted for the item.

The two fields that are immediately useful to support this query, are budget and spent. It is clear looking at the document for the "travel" category, that it had an overspent. We'll reuse our earlier \$match condition to find all of the documents in the collection that had an overspent.

We'll then add a new \$project condition to pull out a sub-set of the information we want to keep.



# \$project Stage: Exercise

Let's focus on the Aggregation Framework syntax:

```
db.budgetitem.aggregate([ { $match: { $and: [ { $expr: { $gt: [
"$spent" , "$budget" ] } } ], { spent: { $gte: 250 } } }, {
"details": { $exists: true } } ] } }, { $project: { _id: 0,
"expense": 1, "airline": "$details.airline", "flight_class":
"$details.seat_class", "flight_type": "$details.flight_type",
overspend: { $subtract: [ "$spent" , "$budget" ] } } } ] )
```

This example builds on our earlier example to include the \$match stage with some additional logic and adds a new \$project stage. We'll look at each stage in more detail to better understand what the pipeline is doing.



# \$project Stage: Exercise

Let's focus on the Aggregation Framework syntax:

```
db.budgetitem.aggregate([  
  $match: { $and: [ { $expr: {  
    $gt: [ "$spent" , "$budget" ] } },  
    { spent: { $gte: 250 } }, {  
    "details": { $exists: true } } ] }  
  }, { $project: { ... } } ])
```

\$match  
stage  
expanded  
from prior  
example

Building on our earlier example we again are interested in the documents where there is an overspent. This \$match uses a \$and to ensure all of the conditions within it are true. Specifically, there are three conditions. The first condition is that we spent more than we budgeted which is captured by the \$gt operator. The next condition within the \$and tracks only documents where the spend field has 250 or more (let's focus on the big ticket expenses). The third and final condition within the \$and uses the \$exists operator to only return documents which have a details sub-document.

Restating this \$match, we want documents where there is an overspend, where the spend is greater than 250, and where there is a details sub-document.

The documents which pass all of these criteria are then sent to the \$project stage that we'll look at next.

We can see the use of \$expr in the \$match stage. This allows the use of aggregation expressions as we noted \$match can't use these expression in the 'raw' and must use them with the \$expr operator.

In order to determine which categories had an overspend, we can compare documents and look for documents where the "spent" field is greater than the "budget" field. This will answer our question, to find where there were monthly budget overspends.



# \$project Stage: Exercise

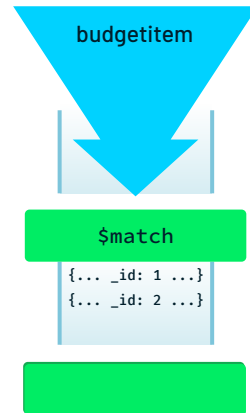
Let's focus on the Aggregation Framework syntax:

```
db.budgetitem.aggregate([ { $match: { ... } }, { $project: { _id: 0, "expense": 1, "airline": "$details.airline", "flight_class": "$details.seat_class", "flight_type": "$details.flight_type", overspend: { $subtract: [ "$spent" , "$budget" ] } } } ] )
```

Focusing now on the \$project stage, we can highlight a few aspects. We can see that the \_id field is explicitly excluded and the expense field is explicitly included. We can also see how to promote fields from an embedded field to the top level of the output document. The last item to note is how we can create a new field, in this case the “overspend” field. This is created by subtracting the budget from the spent amount which will give the specific figure for the overspent.



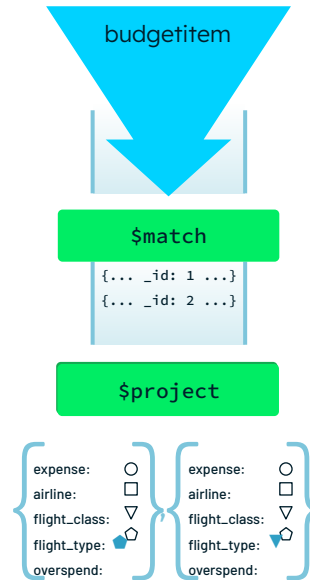
```
[{ $match: { $and: [
  { $expr: { $gt: [ "$spent" ,
"$budget" ] } },
  { spent: { $gte: 250 } },
  { "details": { $exists: true
} } ] } },
{ $project: {
  _id: 0, "expense": 1, "airline":
"$details.airline",
  "flight_class":
"$details.seat_class",
  "flight_type":
"$details.flight_type",
  overspend: { $subtract: [
"$spent" , "$budget" ] } } } ]
```



Let's look at how this would work when we combine them. Firstly focusing on the \$match stage, we can see that we will only return two documents that meet the criteria specified in the match.

It's worth noting at this point, these documents have not been modified in any way as we can see by their \_id field still being present.

```
[{ $match: { $and: [
  { $expr: { $gt: [ "$spent" ,
"$budget" ] } },
  { spent: { $gte: 250 } },
  { "details": { $exists: true
} } ] } },
{ $project: {
  _id: 0, "expense": 1, "airline":
"$details.airline",
  "flight_class":
"$details.seat_class",
  "flight_type":
"$details.flight_type",
  overspend: { $subtract: [
"$spent" , "$budget" ] } } } ]
```



Let's now look at the \$project stage, we can see the `_id` field has been excluded. We have included the `expense` field and promoted the fields `airline`, `seat_class`, and `flight_type` from the `details` sub-document to the top-level of the document. Finally, we have added a new field, `overspend`, which contains the exact amount of the overspend for this expense item.

# \$project Stage: Exercise



Let's insert some real data on budget items!

```
>>> db.budgetitem.insertMany([{"_id" : 1, "category" : "travel", "expense":  
"airplane", "spent": 250, "budget": 200, details: { "airline": "United",  
"flight_type": "return", "seat_class": "economy"}}, {"_id" : 2, "category" :  
"travel", "expense": "airplane", "spent": 450, "budget": 200, details: { "airline":  
"United", "flight_type": "return", "seat_class": "first"}}, {"_id" : 3, "category" :  
"travel", "expense": "train", "spent": 50, "budget": 75}, {"_id" : 4, "category" :  
"travel", "expense": "bus", "spent": 25, "budget": 25}])  
  
...  
{ "acknowledged" : true, "insertedIds" : [ 1, 2, 3, 4] }
```

You should cut and paste the following command directly from the slide or from these notes into the prompt (indicated by >>>). Once they have been inserted you will see the following output on the screen. The result will be the same as we are explicitly setting the ObjectIds for these documents.

```
db.budgetitem.insertMany([{"_id" : 1, "category" : "travel",  
"expense": "airplane", "spent": 250, "budget": 200, details: {  
"airline": "United", "flight_type": "return", "seat_class":  
"economy"}}, {"_id" : 2, "category" : "travel", "expense":  
"airplane", "spent": 450, "budget": 200, details: { "airline":  
"United", "flight_type": "return", "seat_class": "first"}}, {"  
"_id" : 3, "category" : "travel", "expense": "train", "spent":  
50, "budget": 75}, {"_id" : 4, "category" : "travel",  
"expense": "bus", "spent": 25, "budget": 25}])
```

See: <https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/>



# \$project Stage: Exercise

## Example

```
db.budgetitem.aggregate([ { $match: { $and: [ { $expr: { $gt: [ "$spent" ,
"$budget" ] } } ], { spent: { $gte: 250 } }, { "details": { $exists: true } } ] }
}, { $project: { _id: 0, "expense": 1, "airline": "$details.airline",
"flight_class": "$details.seat_class", "flight_type": "$details.flight_type",
overspend: { $subtract: [ "$spent" , "$budget" ] } } } ] )

{ "expense" : "airplane", "airline" : "United", "flight_class" : "economy",
"overspend" : 50 }
{ "expense" : "airplane", "airline" : "United", "flight_class" : "first",
"overspend" : 250 }
```

Let's just recap the aggregation pipeline again, we are using \$match with \$expr to allow us to use the aggregation expression \$gt (greater than). We're checking for documents where the value in the "spent" field is greater than the "budget" field indicating an overspend. We're also filtering out documents where the spent on the item is less than 250 and finally we are only looking for documents with a details sub-document. The \$and operator means all three of these conditions must be true for a document to be passed to the next stage of the aggregation.

In the second stage of the pipeline we use \$project to filter out fields, to promote several fields from the sub-document to the top of the document hierarchy, we also include the expense field as well as adding a new field which includes the calculation of the specific amount overspent for the budget item.



## \$project Stage: Exercise

Next we will find the docs with no sub-docs and the overspend. Using the same window, change **<a>** to the boolean to false to select only documents without the “details” embedded document. Change **<b>** to the operator necessary to calculate the difference between the spent and the budget fields.

```
>>> db.budgetitem.aggregate([ { $match: { "details": { $exists: <a> } } },  
{ $project: { _id: 0, "expense": 1, overspend: { <b>: [ "$spent" ,  
"$budget" ] } } } ] )  
  
{ "expense" : "train", "overspend" : -25 }  
{ "expense" : "bus", "overspend" : 0 }
```

It's your turn to find documents which do not have an embedded “details” sub-document and where the expense and the overspend between the ‘spent’ and the ‘budget’ fields is included in the output document. This means you will need to change **<a>** to the boolean so it selects only documents without the “details” field. You will also need to change **<b>** to the operator necessary to calculate the overspend. This is done through subtraction.

The result in the code block is what you should see if you are successful.

```
db.budgetitem.aggregate([ { $match: { "details": { $exists:  
false } } }, { $project: { _id: 0, "expense": 1, overspend: {  
$subtract: [ "$spent" , "$budget" ] } } } ] )
```

# Quiz





# Quiz

Which of the following are true for the \$project aggregation stage in MongoDB? More than one answer choice can be correct.

- ☐ A. \$project uses a document to specify criteria
- ☐ B. \$project can include and exclude fields to be outputted
- ☐ C. \$project can use array indexes
- ☐ D. \$project cannot promote fields from embedded arrays or sub-documents

# Quiz



Which of the following are true for the \$project aggregation stage in MongoDB? More than one answer choice can be correct.

- ✓ A. \$project uses a document to specify criteria
- ✓ B. \$project can include and exclude fields to be outputted
- ✗ C. \$project can use array indexes
- ✗ D. \$project cannot promote fields from embedded arrays or sub-documents

CORRECT: \$project uses a document to specify criteria - This is correct. \$project uses a document to hold the specification criteria used for the stage.

CORRECT: \$project can include and exclude fields to be outputted - This is correct. The \$project stages can exclude and include fields in the outputted document.

INCORRECT: \$project can use array indexes - This is incorrect. The \$project works on the documents passed to it, this could be an entire collection or a subset but it is unable to map these back to array indexes..

INCORRECT: \$project cannot promote fields from embedded arrays or sub-documents - This is incorrect. The \$project stage can promote fields to the top level of the document by adding a new field by using field paths to set the embedded field as the value for the new field.





# Quiz

Which of the following are true for the \$project aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$project uses a document to specify criteria
- ☒ B. \$project can include and exclude fields to be outputted
- ☐ C. \$project can use array indexes
- ☐ D. \$project cannot promote fields from embedded arrays or sub-documents

*This is correct. \$project uses a document to hold the specification criteria used for the stage.*

CORRECT: \$project uses a document to specify criteria - This is correct. \$project uses a document to hold the specification criteria used for the stage.

# Quiz



Which of the following are true for the \$project aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$project uses a document to specify criteria
- ☒ B. \$project can include and exclude fields to be outputted
- ☐ C. \$project can use array indexes
- ☐ D. \$project cannot promote fields from embedded arrays or sub-documents

*This is correct. \$project uses a document to hold the specification criteria used for the stage.*

CORRECT: \$project can include and exclude fields to be outputted - This is correct. The \$project stages can exclude and include fields in the outputted document.

# Quiz



Which of the following are true for the \$project aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$project uses a document to specify criteria
- ☒ B. \$project can include and exclude fields to be outputted
- ☐ C. \$project can use array indexes
- ☐ D. \$project cannot promote fields from embedded arrays or sub-documents

*This is incorrect. The \$project works on the documents passed to it, this could be an entire collection or a subset but it is unable to map these back to array indexes.*

INCORRECT: \$project can use indexes - This is incorrect. The \$project works on the documents passed to it, this could be an entire collection or a subset but it is unable to map these back to array indexes.

# Quiz



Which of the following are true for the \$project aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$project uses a document to specify criteria
- ☒ B. \$project can include and exclude fields to be outputted
- ☐ C. \$project can use array indexes
- ☐ D. \$project cannot promote fields from embedded arrays or sub-documents

*This is incorrect. The \$project stage can promote fields to the top level of the document by adding a new field by using field paths to set the embedded field as the value for the new field.*

INCORRECT: \$project cannot promote fields from embedded arrays or sub-documents - This is incorrect. The \$project stage can promote fields to the top level of the document by adding a new field by using field paths to set the embedded field as the value for the new field.



# \$unwind

Let's look at the \$unwind stage as this can be useful when querying data and particularly for restructuring it.

# Aggregation Framework \$unwind



Deconstruct an array field outputting a document for each element

Specify a field path to indicate the array to be deconstructed or specify a document operator

Unwind nested arrays

Control output document if there are empty or null arrays

Let's firstly introduce the purpose of the \$unwind stage. It deconstructs an array field to output a document for each element in that array. This means that each output document represents one of the array field values and each value will have its own document.

# Aggregation Framework \$unwind



Deconstruct an array field outputting a document for each element

Specify a field path to indicate the array to be deconstructed or specify a document operator

Unwind nested arrays

Control output document if there are empty or null arrays

The \$unwind stage can take either a field path or a document operator when specifying the array to unwind.

# Aggregation Framework \$unwind



Deconstruct an array field outputting a document for each element

Specify a field path to indicate the array to be deconstructed or specify a document operator

Unwind nested arrays

Control output document if there are empty or null arrays

\$unwind can be used to unwind embedded arrays. We'll look at an example of this shortly.



# Aggregation Framework \$unwind



Deconstruct an array field outputting a document for each element

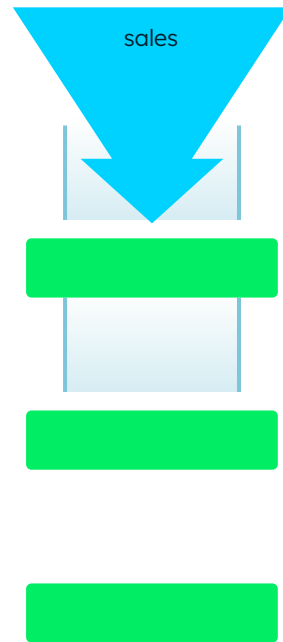
Specify a field path to indicate the array to be deconstructed or specify a document operator

Unwind nested arrays

Control output document if there are empty or null arrays

The control of which documents are output can be controlled if there are empty or null arrays in input documents. This can be useful to error out or to continue depending on the context for the aggregation pipeline.

# \$unwind example using sales data



Let's take a few minutes to explore the \$unwind stage with a hands-on session, we'll walk through the syntax, create the data and then query the database to get the results.

Follow along using the [MongoDB Shell](#). If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).

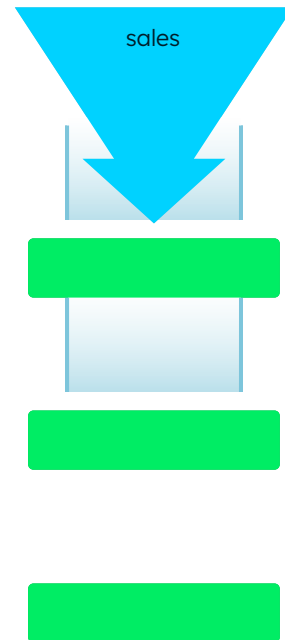
This builds on what we did for our previous session with the \$match stage in this lesson.

Let's imagine our aggregation pipeline as literally stages in a pipe and for this example we can think of it as three interconnected but separate pipes.

```
[
  { $unwind: "$items" },

  { $unwind: "$items.tags" },

  { $group: {
    _id: "$items.tags",
    totalSalesAmount: { $sum: {
      $multiply:
        [ "$items.price",
          "$items.quantity" ] } } }
  }
]
```

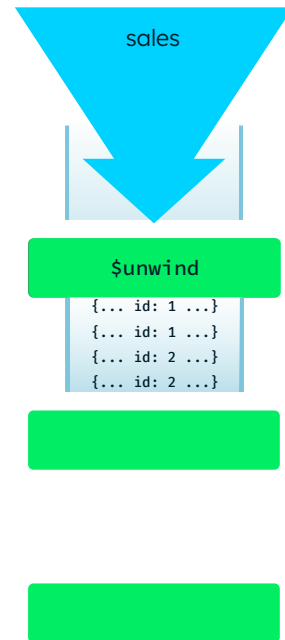


Let's imagine our aggregation pipeline as literally stages in a pipe and for this example we can think of it as three interconnected but separate pipes.

```
[
  { $unwind: "$items" },

  { $unwind: "$items.tags" },

  { $group: {
    _id: "$items.tags",
    totalSalesAmount: { $sum: {
      $multiply:
        [ "$items.price",
          "$items.quantity" ] } } }
  }
]
```

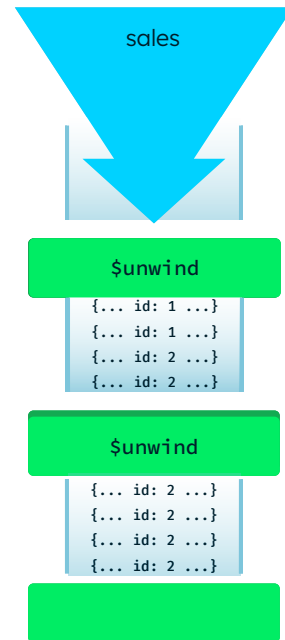


The first unwind will break out each item from the array of items. This will create four documents.

```
[
  { $unwind: "$items" },

  { $unwind: "$items.tags" },

  { $group: {
    _id: "$items.tags",
    totalSalesAmount: { $sum: {
      $multiply:
        [ "$items.price",
          "$items.quantity" ] } } }
  }
]
```

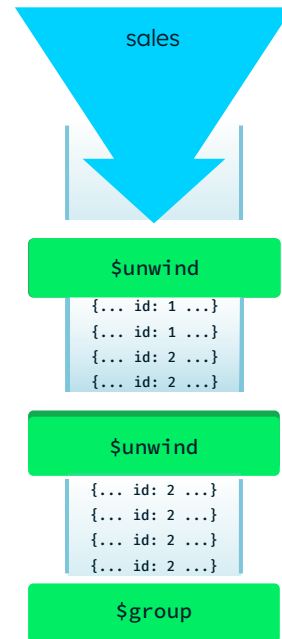


The second \$unwind will break out each of the tags into their own individual document, in this example it will create 10 documents from the input 4 documents.

```
[
  { $unwind: "$items" },

  { $unwind: "$items.tags" },

  { $group: {
    _id: "$items.tags",
    totalSalesAmount: { $sum: {
      $multiply:
        [ "$items.price",
          "$items.quantity" ] } } }
  }
]
```



In the final stage we'll group the document to have two fields, an `_id` that represents a tag and a `totalSalesAmount` which calculates how many sales there were for that specific item. We calculate this as we had the number sold and the price per item but the total sales figure wasn't stored so we calculate it and store it now.



# \$unwind Stage: Exercise

Let's focus on the sales example and look at one document:

```
{ _id: "1", "items" : [ { "name" : "pens", "tags"
: [ "writing", "office", "school", "stationary" ],
"price" : NumberDecimal("12.00"), "quantity" :
NumberInt("5") }, { "name" : "envelopes", "tags" [
"stationary", "office" ], "price" :
NumberDecimal("1.95"), "quantity" : NumberInt("8")
} ]
}
```

**Nested  
arrays**

Let's firstly look at a sample document for a single sales example to see what data is contained that will help us create a query to find where we total sales amount per each tag. In this document, we can see that "items" is an array holding documents, for each of the "items" documents.

We can further see they each possess a further array 'tags' which holds the tags related to that item. This is a great example schema to show how \$unwind can be really useful in processing and restructuring documents.



# \$unwind Stage: Exercise

Let's focus on the Aggregation Framework syntax:

```
db.sales.aggregate([
  { $unwind: "$items" },
  { $unwind: "$items.tags" },
  { $group: { _id: "$items.tags", totalSalesAmount: { $sum: {
    $multiply: [ "$items.price", "$items.quantity" ] } } } } ])
```

There are three stages in this aggregation example, two of the \$unwind stages are used to extract an array elements from within another array (an array within an array). The \$group stage is used to group the documents by tag and then using the \$sum and \$multiple operators to calculate the total sales for each item with that tag.



# \$unwind Stage: Exercise



Let's insert some real data on sales!

```
db.sales.insertMany([
  { _id: "1", "items" : [ { "name" : "pens", "tags" : [ "writing", "office",
"school", "stationary" ], "price" : NumberDecimal("12.00"), "quantity" :
NumberInt("5") }, { "name" : "envelopes", "tags" : [ "stationary", "office" ],
"price" : NumberDecimal("1.95"), "quantity" : NumberInt("8") } ] },
  { _id: "2", "items" : [ { "name" : "laptop", "tags" : [ "office", "electronics"
], "price" : NumberDecimal("800.00"), "quantity" : NumberInt("1") }, { "name" :
"notepad", "tags" : [ "stationary", "school" ], "price" : NumberDecimal("14.95"),
"quantity" : NumberInt("3") } ] } ] )
...
{ "acknowledged" : true, "insertedIds" : [ 1, 2 ] }
```

We'll add two sample documents into a new 'sales' collection but have the same structure with the field items holding an array, each item in that array being a document and within those documents there being a second array associated to the 'tags' field.



# \$unwind Stage: Exercise

## Example

```
db.sales.aggregate([ { $unwind: "$items" }, { $unwind: "$items.tags" }, {  
$group: { _id: "$items.tags", totalSalesAmount: { $sum: { $multiply: [  
"$items.price", "$items.quantity" ] } } } } ] )
```

```
{ "_id" : "writing", "totalSalesAmount" : NumberDecimal("60.00") }  
{ "_id" : "office", "totalSalesAmount" : NumberDecimal("875.60") }  
{ "_id" : "school", "totalSalesAmount" : NumberDecimal("104.85") }  
{ "_id" : "stationary", "totalSalesAmount" : NumberDecimal("120.45") }  
{ "_id" : "electronics", "totalSalesAmount" : NumberDecimal("800.00") }
```

Let's recap on the aggregation pipeline, it uses three stages. The first two stages \$unwind all of the items in firstly in the \$item array and then in the tags nested array. The third and final stage groups the items according to their tag and calculates the total sales amount for that tag.

We can see five documents being returned. another array (an array within an array). The \$group stage is used group the documents by tag and then using the \$sum and \$multiple operators to calculate the total sales for each item with that tag.



# \$unwind Stage: Exercise

Find the docs with the 'office' tag

Using the same window, change `<a>` to the operator for equal to and change `<b>` to the field in the document which holds the tags information.

```
>>> db.sales.aggregate([ { $unwind: "$items" }, { $unwind: "$items.tags" }, {  
$match: { $expr: { <a>: ["<b>", "office"] } } } ])
```

```
{ "_id" : "1", "items" : { "name" : "pens", "tags" : "office", "price" :  
NumberDecimal("12.00"), "quantity" : 5 } }  
{ "_id" : "1", "items" : { "name" : "envelopes", "tags" : "office", "price" :  
NumberDecimal("1.95"), "quantity" : 8 } }  
{ "_id" : "2", "items" : { "name" : "laptop", "tags" : "office", "price" :  
NumberDecimal("800.00"), "quantity" : 1 } }
```

It's your turn to find documents where the tags match the category "office". In this pipeline you will need to add the equal operator and also the field for the tags of the items, a hint is that this is an embedded field in the output document of the second unwind stage.

The result in the code block is what you should see if you are successful.

```
db.sales.aggregate([ { $unwind: "$items" }, { $unwind:  
"$items.tags" }, { $match: { $expr: { $eq: ["$items.tags",  
"office"] } } } ])
```

# Quiz



# Quiz



Which of the following are true for the \$unwind aggregation stage in MongoDB? More than one answer choice can be correct.

- ☐ A. \$unwind uses field paths to identify the array to be unwound
- ☐ B. \$unwind can be used multiple times for embedded arrays
- ☐ C. \$unwind can use indexes
- ☐ D. \$unwind cannot be used when the arrays contain empty or null entries

# Quiz



Which of the following are true for the \$unwind aggregation stage in MongoDB? More than one answer choice can be correct.

- ✓ A. \$unwind uses field paths to identify the array to be unwound
- ✓ B. \$unwind can be used multiple times for embedded arrays
- ✗ C. \$unwind can use indexes
- ✗ D. \$unwind cannot be used when the arrays contain empty or null entries

CORRECT: \$unwind uses field paths to identify the array to be unwound- This is correct. Field paths are used directly or with document operators to specify the array to be unwound by the \$unwind stage.

CORRECT: \$unwind can be used multiple times for embedded arrays - This is correct. You can use a number of stages of \$unwind sequentially to unwind deeply nested arrays.

INCORRECT: \$unwind can use indexes - This is incorrect. It is not possible to use \$unwind with indexes. The focus of this stage is data manipulation or data restructuring rather than data querying.

The \$unwind stage is focused on the stream of documents and pull aparting the array elements into individual documents.

INCORRECT: \$unwind cannot be used when the arrays contain empty or null entries - This is incorrect. The preserveNullAndEmptyArrays flag can be used to process arrays with null or empty entries.

# Quiz



Which of the following are true for the \$unwind aggregation stage in MongoDB? More than one answer choice can be correct.

- ✓ A. \$unwind uses field paths to identify the array to be unwound
- ✓ B. \$unwind can be used multiple times for embedded arrays
- ✗ C. \$unwind can use indexes
- ✗ D. \$unwind cannot be used when the arrays contain empty or null entries

*This is correct. \$project uses a document to hold the specification criteria used for the stage. Field paths are used directly or with document operators to specify the array to be unwound by the \$unwind stage.*

CORRECT: \$unwind uses field paths to identify the array to be unwound- This is correct. Field paths are used directly or with document operators to specify the array to be unwound by the \$unwind stage.

# Quiz



Which of the following are true for the \$unwind aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$unwind uses field paths to identify the array to be unwound
- ☒ B. \$unwind can be used multiple times for embedded arrays
- ☐ C. \$unwind can use indexes
- ☐ D. \$unwind cannot be used when the arrays contain empty or null entries

*This is correct. You can use a number of stages of \$unwind sequentially to unwind deeply nested arrays.*

CORRECT: \$unwind can be used multiple times for embedded arrays - This is correct. You can use a number of stages of \$unwind sequentially to unwind deeply nested arrays.



# Quiz



Which of the following are true for the \$unwind aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$unwind uses field paths to identify the array to be unwound
- ☒ B. \$unwind can be used multiple times for embedded arrays
- ☐ C. \$unwind can use indexes
- ☐ D. \$unwind cannot be used when the arrays contain empty or null entries

*This incorrect. It is not possible to use \$unwind with indexes. The focus of this stage is data manipulation or data restructuring rather than data querying.*

INCORRECT: \$unwind can use indexes - This is incorrect. It is not possible to use \$unwind with indexes. The focus of this stage is data manipulation or data restructuring rather than data querying.

The \$unwind stage is focused on the stream of documents and pull aparting the array elements into individual documents.

# Quiz



Which of the following are true for the \$unwind aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$unwind uses field paths to identify the array to be unwound
- ☒ B. \$unwind can be used multiple times for embedded arrays
- ☐ C. \$unwind can use indexes
- ☐ D. \$unwind cannot be used when the arrays contain empty or null entries

*This incorrect. The preserveNullAndEmptyArrays flag can be used to process arrays with null or empty entries.*

INCORRECT: \$unwind cannot be used when the arrays contain empty or null entries - This is incorrect. The preserveNullAndEmptyArrays flag can be used to process arrays with null or empty entries.



# \$facet

Let's look at the \$facet stage as this can be useful when querying data and particularly for running multiple pipelines with a single aggregation.

## Aggregation Framework \$facet



Run multiple aggregation pipelines in one stage

All the pipelines act on the same set of input documents

Each sub-pipeline returns a field in the output document

The results of each sub-pipeline are saved as an array of documents

\$facet allows for multiple aggregation pipelines to be run within a single stage on the same set of input documents. Each of the sub-pipelines has its own field in the output document where the results from it are saved as an array of documents.

## Aggregation Framework \$facet



Run multiple aggregations pipelines in one stage

All the pipelines act on the same set of input documents

Each sub-pipeline returns a field in the output document

The results of each sub-pipeline are saved as an array of documents

Each of the pipelines defined within the \$facet stage operate on the same set of input documents. This allows for the documents to be classified across several dimensions within this one stage.

## Aggregation Framework \$facet



Run multiple aggregations pipelines in one stage

All the pipelines act on the same set of input documents

Each sub-pipeline returns a field in the output document

The results of each sub-pipeline are saved as an array of documents

The output of each sub-pipeline is returned as a single field within the output document. These fields can be used in later stages after the \$facet stage.

## Aggregation Framework \$facet



Run multiple aggregations pipelines in one stage

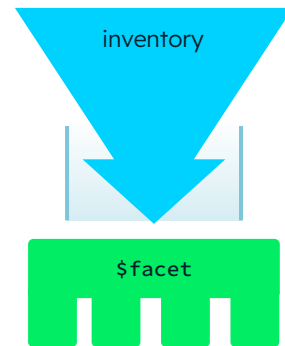
All the pipelines act on the same set of input documents

Each sub-pipeline returns a field in the output document

The results of each sub-pipeline are saved as an array of documents

The results of each sub-pipeline are saved as an array of documents.

# \$facet example using sales data



Let's take a few minutes to explore the \$facet stage with a hands-on session, we'll walk through the syntax, create the data and then query the database to get the results.

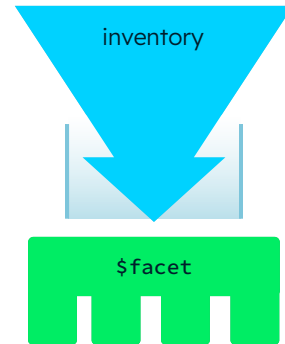
Follow along using the [MongoDB Shell](#). If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).

Looking at the \$facet stage, we input the documents from the inventory collection and then perform a number of sub-pipelines on those documents. Each of the sub-stages within \$facet will return a single field with the results of that stage stored in it as an array.



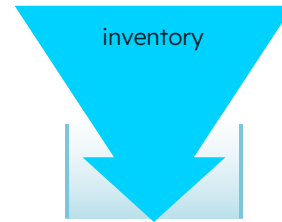


```
[
  { $facet: { "categorizedByManufacturer": [{
    $sortByCount: "$manufacturer" } ]},
    "categorizedByTags": [{ $unwind: "$tags" },{
    $sortByCount: "$tags" } ]},
    "categorizedByPrice": [ { $match: { price: {
    $exists: 1 } } }, { $bucket: { groupBy:
    "$price", boundaries: [ 0, 15, 20, 25, 30 ],
    default: "Other", output: { "count": { $sum:
    1 }, "titles":{ $push: "$title" }}}}],
    "categorizedByQuantity(Auto)": [{
    $bucketAuto: {groupBy: "$quantity", buckets:
    3 } } ]
  } }
]
```



Looking at the `$facet` stage, we input the documents from the `inventory` collection and then perform a number of sub-pipelines on those documents. Each of the sub-stages within `$facet` will return a single field with the results of that stage stored in it as an array.

```
[
  { $facet: {
    "categorizedByManufacturer": [{
      $sortByCount: "$manufacturer" }],
    "categorizedByTags": [{ $unwind: "$tags"
    }, { $sortByCount: "$tags" }],
    "categorizedByPrice": [ { $match: {
      price: { $exists: 1 } } }, { $bucket: {
      groupBy: "$price", boundaries: [ 0, 15,
      20, 25, 30 ], default: "Other", output: {
      "count": { $sum: 1 }, "titles": { $push:
      "$title" }}}}],
    "categorizedByQuantity(Auto)": [{
      $bucketAuto: {groupBy: "$quantity",
      buckets: 3 }}]
  } }
]
```



```
{...
  categorizedByManufacturer: [],
  categorizedByTags: [],
  categorizedByPrice: [],
  categorizedByQuantity(Auto): [],
  ...}
```

Looking a little deeper into the \$facet sub-stages, we can see the four stages which can be used to explore different aspects of the data. The pipeline in this example is looking at tags, price, manufacturer details and how many are in stock. Each of the sub-stages within \$facet will return a single field with the results of that stage stored in it as an array.



# \$facet Stage: Exercise

Let's focus on the inventory example and look at one document:

```
{
  "_id" : 1, "title" : "Jinhao Fountain Pen",
  "manufacturer": "Jinhao",
  "quantity": NumberInt("5"),
  "tags": [ "writing", "office", "school",
            "stationary" ],
  "price" : NumberDecimal("15.00")
}
```

Let's firstly look at a sample document for a single inventory example to see what data is contained that will help us create the aggregation for the document.

We can see that it might be interesting to look at the inventory items by who made them (manufacturer), by how many of them we have in stock (quantity), and by their price.

We can further see they each possess a further array 'tags' which holds the tags related to that item. We can again use \$unwind to help explore these tags and further combine it to determine how many items we have in each 'tag' category.



# \$facet Stage: Exercise

Let's focus on the Aggregation Framework syntax:

```
db.inventory.aggregate([ { $facet: { "categorizedByManufacturer": [ {  
  $sortByCount: "$manufacturer" } ], "categorizedByTags": [ { $unwind:  
  "$tags" }, { $sortByCount: "$tags" } ], "categorizedByPrice": [ {  
  $match: { price: { $exists: 1 } } }, { $bucket: { groupBy: "$price",  
  boundaries: [ 0, 15, 20, 25, 30 ], default: "Other", output: {  
  "count": { $sum: 1 }, "titles": { $push: "$title" } } } ] },  
  "categorizedByQuantity(Auto)": [ { $bucketAuto: { groupBy: "$quantity",  
  buckets: 3 } } ] } } ] )
```

Drilling into the aggregation framework \$facet stage here, we can see it is essentially calling four different sub-pipelines within the \$facet stage.

# \$facet Stage: Exercise



Let's insert some real data on inventory

```
db.inventory.insertMany([ {"_id" : 1, "title" : "Jinhao Fountain Pen",
"manufacturer": "Jinhao", "quantity": NumberInt("5"), "tags": [ "writing",
"office", "school", "stationary" ], "price" : NumberDecimal("15.00") }, {"_id"
: 2, "title" : "Pilot MR Zig Zag Fountain Pen", "manufacturer": "Pilot",
"quantity": NumberInt("10"), "tags": [ "writing", "office", "school",
"stationary" ], "price" : NumberDecimal("20.00")}, {"_id" : 3, "title" :
"Pilot MR2 Tiger Fountain Pen", "manufacturer": "Pilot", "quantity":
NumberInt("5"), "tags": [ "writing", "office", "school", "stationary" ],
"price" : NumberDecimal("25.00")} ])
...
{ "acknowledged" : true, "insertedIds" : [ 1, 2, 3 ] }
```

We'll add three sample documents into a new inventory collection and we'll use these to show how \$facet works.

Copy and paste into the Shell:

```
db.inventory.insertMany([ {"_id" : 1, "title" : "Jinhao
Fountain Pen", "manufacturer": "Jinhao", "quantity":
NumberInt("5"), "tags": [ "writing", "office", "school",
"stationary" ], "price" : NumberDecimal("15.00") }, {"_id" : 2,
"title" : "Pilot MR Zig Zag Fountain Pen", "manufacturer":
"Pilot", "quantity": NumberInt("10"), "tags": [ "writing",
"office", "school", "stationary" ], "price" :
NumberDecimal("20.00")}, {"_id" : 3, "title" : "Pilot MR2 Tiger
Fountain Pen", "manufacturer": "Pilot", "quantity":
NumberInt("5"), "tags": [ "writing", "office", "school",
"stationary" ], "price" : NumberDecimal("25.00")} ])
```



# \$facet Stage: Exercise

Results:

```
{ "categorizedByManufacturer" : [ { "_id" : "Pilot", "count" : 2 }, {
  "_id" : "Jinhao", "count" : 1 } ], "categorizedByTags" : [ { "_id" :
  "office", "count" : 3 }, { "_id" : "stationary", "count" : 3 }, {
  "_id" : "writing", "count" : 3 }, { "_id" : "school", "count" : 3 } ],
  "categorizedByPrice" : [ { "_id" : 15, "count" : 1, "titles" : [
  "Jinhao Fountain Pen" ] }, { "_id" : 20, "count" : 1, "titles" : [
  "Pilot MR Zig Zag Fountain Pen" ] }, { "_id" : 25, "count" : 1,
  "titles" : [ "Pilot MR2 Tiger Fountain Pen" ] } ],
  "categorizedByQuantity(Auto)" : [ { "_id" : { "min" : 5, "max" : 10 },
  "count" : 2 }, { "_id" : { "min" : 10, "max" : 10 }, "count" : 1 } ] }
```

Here is the output of the \$facet we can see each of the four fields from the sub-pipelines creating arrays with the results of that specific sub-pipeline.

The small number of input documents (3) gives a taste of what the results of \$facet might be for a much larger inventory collection of tens or hundreds of documents.

# Quiz



# Quiz



Which of the following are true for the \$facet aggregation stage in MongoDB? More than one answer choice can be correct.

- ☐ A. \$facet allows for multiple sub-pipelines to be run in the stage
- ☐ B. Each sub-pipeline adds a field with an array holding its output
- ☐ C. \$facet can use indexes
- ☐ D. An output field from \$facet can be used as an input field within the same \$facet stage



# Quiz



Which of the following are true for the \$facet aggregation stage in MongoDB? More than one answer choice can be correct.

- ✓ A. \$facet allows for multiple sub-pipelines to be run in the stage
- ✓ B. Each sub-pipeline adds a field with an array holding its output
- ✗ C. \$facet can use indexes
- ✗ D. An output field from \$facet can be used as an input field within the same \$facet stage

CORRECT: \$facet allows for multiple sub-pipelines to be run in the stage - This is correct. The \$facet stage allows for one or more sub-pipelines to be run within the stage.

CORRECT: Each sub-pipeline adds a field with an array holding its output- This is correct. Each of the sub-pipeline stages in a \$facet stage will provide one field in the output document with its output stored as an array.

INCORRECT: \$facet can use indexes. This is incorrect. Additionally, even if a \$match stage is used as the first stage in a pipeline this aggregation will not be able to use indexes. A \$facet stage will force the aggregation pipeline it is present in to use a collection scan (COLSCAN).

INCORRECT: An output field from \$facet can be used as an input field within the same \$facet stage. This is incorrect. It is possible to use the output field from a \$facet sub-pipeline as an input to a second later \$facet stage in the overall aggregation pipeline. It is not possible to use an output field as an input field within the same \$facet stage.

# Quiz



Which of the following are true for the \$facet aggregation stage in MongoDB? More than 1 answer choice can be correct.

- ☒ A. \$facet allows for multiple sub-pipelines to be run in the stage
- ☒ B. Each sub-pipeline adds a field with an array holding its output
- ☐ C. \$facet can use indexes
- ☐ D. An output field from \$facet can be used as an input field within the same \$facet stage

*This is correct. The \$facet stage allows for one or more sub-pipelines to be run within the stage.*

CORRECT: \$facet allows for multiple sub-pipelines to be run in the stage - This is correct. The \$facet stage allows for one or more sub-pipelines to be run within the stage.

# Quiz



Which of the following are true for the \$facet aggregation stage in MongoDB? More than one answer choice can be correct.

- ✓ A. \$facet allows for multiple sub-pipelines to be run in the stage
- ✓ B. Each sub-pipeline adds a field with an array holding its output
- ✗ C. \$facet can use indexes
- ✗ D. An output field from \$facet can be used as an input field within the same \$facet stage

*This is correct. Each of the sub-pipeline stages in a \$facet stage will provide one field in the output document with its output stored as an array.*

CORRECT: Each sub-pipeline adds a field with an array holding its output- This is correct. Each of the sub-pipeline stages in a \$facet stage will provide one field in the output document with its output stored as an array.

# Quiz



Which of the following are true for the \$facet aggregation stage in MongoDB? More than 1 answer choice can be correct.

- ✓ A. \$facet allows for multiple sub-pipelines to be run in the stage
- ✓ B. Each sub-pipeline adds a field with an array holding its output
- ✗ C. \$facet can use indexes
- ✗ D. An output field from \$facet can be used as an input field within the same \$facet stage

*This is incorrect. Additionally, even when a \$match stage is used as the first stage it will not be able to use indexes. Using a \$facet stage only allows a collection scan (COLSCAN).*

INCORRECT: \$facet can use indexes. This is incorrect. Additionally, even when a \$match stage is used as the first stage it will not be able to use indexes. Using a \$facet stage only allows a collection scan (COLSCAN).

# Quiz



Which of the following are true for the \$facet aggregation stage in MongoDB? More than 1 answer choice can be correct.

- ☒ A. \$facet allows for multiple sub-pipelines to be run in the stage
- ☒ B. Each sub-pipeline adds a field with an array holding its output
- ☐ C. \$facet can use indexes
- ☐ D. An output field from \$facet can be used as an input field within the same \$facet stage

*This is incorrect. It is possible to use the output field from a \$facet sub-pipeline as an input to a second later \$facet stage in the overall aggregation pipeline.*

INCORRECT: An output field from \$facet can be used as an input field within the same \$facet stage. This is incorrect. It is possible to use the output field from a \$facet sub-pipeline as an input to a second later \$facet stage in the overall aggregation pipeline.



\$merge

## Aggregation Framework \$merge



Writes the results of a pipeline to a collection, must be the last stage

Can output to the same collection as being inputted, can be sharded

Creates a new collection if it does not already exist

Fine grained control of what happens to the documents in terms of updating, merging, deleting if there are existing documents

Let's firstly introduce the purpose of the \$merge stage. It is similar to the \$out stage as it also must be the final stage in a pipeline.

## Aggregation Framework \$merge



Writes the results of a pipeline to a collection, must be the last stage

Can output to the same collection as being inputted, can be sharded

Creates a new collection if it does not already exist

Fine grained control of what happens to the documents in terms of updating, merging, deleting if there are existing documents

This stage allows for the results of an aggregation pipeline to be written to a collection. It can output to a sharded collection. It can also output to the same collection which is used as the input to the aggregation pipeline.



## Aggregation Framework \$merge



Writes the results of a pipeline to a collection, must be the last stage

Can output to the same collection as being inputted, can be sharded

Creates a new collection if it does not already exist

Fine grained control of what happens to the documents in terms of updating, merging, deleting if there are existing documents

If the collection doesn't already exist, it will create it.

## Aggregation Framework \$merge



Writes the results of a pipeline to a collection, must be the last stage

Can output to the same collection as being inputted, can be sharded

Creates a new collection if it does not already exist

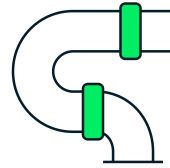
Fine grained control of what happens to the documents in terms of updating, merging, deleting if there are existing documents

It can be thought as of similar to the \$out stage but with more fine grained controls on how that output is performed.



**whenMatched** controls how the \$merge will occur where fields exist in both the result document and input document when matched on the \_id field

**whenNotMatched** controls how the \$merge will occur where the fields are merged to the result document where there is no match for the \_id field in the input document



## Fine Grained Control with \$merge

\$merge is much more flexible when compared to the \$out stage and offers more fine grained controls. The whenMatched option is a good example. It sets the behaviour of \$merge when fields exist in both the result document and input document when matched on the \_id field

The whenNotMatched option sets the behaviour of \$merge when the fields do not exist in both the result document and input document when matched on the \_id field.



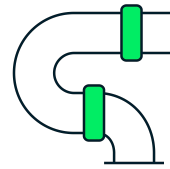
**replace:** Replaces the complete document with the version created by the \$merge stage

**keepExisting:** Keeps the existing document where the \_id matches

**merge:** Similar to \$mergeObjects , adds new fields and replaces old fields with the version created in the \$merge stage

**fail:** Stops and fails the aggregation operation, any earlier changes kept

**pipeline:** Runs a further aggregation pipeline on the documents



## \$merge whenMatched

The \$merge stage whenMatched option has five distinct options that can be used.

Firstly “replace” which replaces the complete document with the version created by the \$merge stage.

Next is “keepExisting,” which keeps the existing document where the \_id matches

Thirdly is “merge” and this is similar to \$mergeObjects. It adds new fields and replaces old fields with the version created in the \$merge stage.

Fourthly, is “fail” which stops and fails the aggregation operation, any earlier changes to documents in the output collection will be kept.

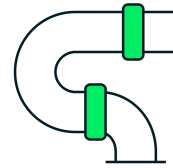
Finally, there is “pipeline” which runs a further aggregation pipeline on the documents.



**insert:** Inserts the document into the output collection

**discard:** Discards the document and continues the aggregation stage

**fail:** Stops and fails the aggregation operation, any earlier changes kept



## \$merge whenNotMatched

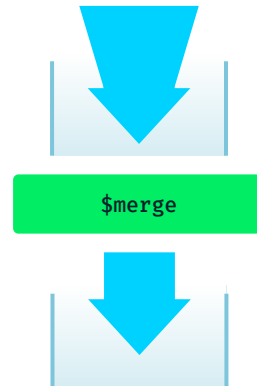
The \$merge stage whenNotMatched option has three distinct options that can be used.

Firstly “insert,” this inserts the document into the output collection.

Secondly, there is “discard” and this option discards the document and continues the aggregation stage.

Finally, there is “fail” which stops and fails the aggregation operation, any earlier changes kept.

# \$merge example using sales data

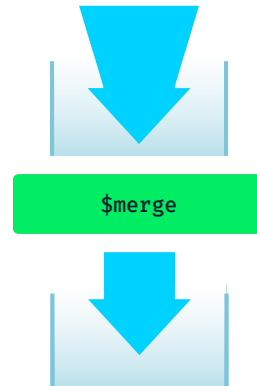


Let's take a few minutes to explore the \$merge stage with a hands-on session, we'll walk through the syntax, create the data and then query the database to get the results.

Follow along using the [MongoDB Shell](#). If you prefer to use a GUI to interact with your data, please use [MongoDB Compass](#) or the [MongoDB Atlas UI](#).

## \$merge

Writes the documents from the pipeline to a collection which can be sharded. It can replace existing documents or update documents unlike \$out.



Let's just recap on the \$merge stage, it writes the documents from the pipeline to a collection which can be sharded. It can replace existing documents or update documents unlike \$out.



# \$merge Stage: Exercise

First, let's focus on the salary example and look at one document:

```
{  
  "_id" : 1, employee: "Ant",  
  dept: "A", salary: 100000,  
  fiscal_year: 2018  
}
```

Let's firstly look at a sample document for a single salary example to see what data is contained that will help us create the aggregation for the document.

We can see that the employee's name, department, salary, and year that the salary refers to are included in the salary document. This means that each employee will have one document for their salary per fiscal year.





## \$merge Stage: Exercise

Now, let's focus on the Aggregation Framework syntax:

```
db.salaries.aggregate( [  
  { $group: { _id: { fiscal_year: "$fiscal_year", dept:  
"$dept" }, salaries: { $sum: "$salary" } } },  
  { $merge : { into: { db: "reporting", coll: "budgets"  
}, on: "_id", whenMatched: "replace", whenNotMatched:  
"insert" } }  
] )
```

Let's look at an example aggregation using the salary documents, firstly grouping them to get the information of how much salaries were spent per department per fiscal year using \$group. We then feed the output of that stage into \$merge, specifically into the “budgets” collection in the “reporting” database. The \$merge stage will insert documents when the \_id (a composite of fiscal year and department) is not present already, if the \_id is present it will replace the existing document with the corresponding document in the pipeline.

The resulting collection gives a high level overview per year per department on each department's total salary expenditure.

# \$merge Stage: Exercise



Let's insert some real data on salaries!

```
db.salaries.insertMany([ { "_id" : 1, employee: "Ant", dept: "A", salary: 100000,
fiscal_year: 2017 }, { "_id" : 2, employee: "Bee", dept: "A", salary: 120000,
fiscal_year: 2017 }, { "_id" : 3, employee: "Cat", dept: "Z", salary: 115000,
fiscal_year: 2017 }, { "_id" : 4, employee: "Ant", dept: "A", salary: 115000,
fiscal_year: 2018 }, { "_id" : 5, employee: "Bee", dept: "Z", salary: 145000,
fiscal_year: 2018 }, { "_id" : 6, employee: "Cat", dept: "Z", salary: 135000,
fiscal_year: 2018 }, { "_id" : 7, employee: "Gecko", dept: "A", salary: 100000,
fiscal_year: 2018 }, { "_id" : 8, employee: "Ant", dept: "A", salary: 125000,
fiscal_year: 2019 }, { "_id" : 9, employee: "Bee", dept: "Z", salary: 160000,
fiscal_year: 2019 }, { "_id" : 10, employee: "Cat", dept: "Z", salary: 150000,
fiscal_year: 2019 } ] )

...

{ "acknowledged" : true, "insertedIds" : [ 1,2,3,4,5,6,7,8,9,10 ] }
```

For this example, we'll add 10 documents across a number of employees and departments as well as years to give a more realistic example for the data.

Here's the code to use to insert this data:

```
db.salaries.insertMany([ { "_id" : 1, employee: "Ant", dept:
"A", salary: 100000, fiscal_year: 2017 }, { "_id" : 2,
employee: "Bee", dept: "A", salary: 120000, fiscal_year: 2017
}, { "_id" : 3, employee: "Cat", dept: "Z", salary: 115000,
fiscal_year: 2017 }, { "_id" : 4, employee: "Ant", dept: "A",
salary: 115000, fiscal_year: 2018 }, { "_id" : 5, employee:
"Bee", dept: "Z", salary: 145000, fiscal_year: 2018 }, { "_id"
: 6, employee: "Cat", dept: "Z", salary: 135000, fiscal_year:
2018 }, { "_id" : 7, employee: "Gecko", dept: "A", salary:
100000, fiscal_year: 2018 }, { "_id" : 8, employee: "Ant",
dept: "A", salary: 125000, fiscal_year: 2019 }, { "_id" : 9,
employee: "Bee", dept: "Z", salary: 160000, fiscal_year: 2019
}, { "_id" : 10, employee: "Cat", dept: "Z", salary: 150000,
fiscal_year: 2019 } ] )
```



# \$merge Stage: Exercise

Results:

Let's look at the results of using our pipeline with \$group and \$merge as a new collection, budgets:

```
{ "_id" : { "fiscal_year" : 2017, "dept" : "A" }, "salaries" : 220000 }
{ "_id" : { "fiscal_year" : 2019, "dept" : "A" }, "salaries" : 125000 }
{ "_id" : { "fiscal_year" : 2019, "dept" : "Z" }, "salaries" : 310000 }
{ "_id" : { "fiscal_year" : 2018, "dept" : "A" }, "salaries" : 215000 }
{ "_id" : { "fiscal_year" : 2018, "dept" : "Z" }, "salaries" : 280000 }
{ "_id" : { "fiscal_year" : 2017, "dept" : "Z" }, "salaries" : 115000 }
```

Here is the resulting output from the aggregation pipeline (\$group and \$merge) which created a new collection reporting in the database, budgets.

We can see an interesting pattern where the department 'A' over the three years had their total salaries decreased and the opposite happened with department 'Z' where over the same period the salaries increased.

This aggregation pipeline is a good example of how the MongoDB Aggregation Framework can be used to create a collection which can be used in reports or potentially visualised with another, for example MongoDB Charts.



## \$merge Stage: Exercise

Find where 50+ was spent and was below budget. Using the same window, change **<a>** and **<b>** to the options for matching and for not matching fields. Change **<c>** to the option where the aggregation will stop on not finding a match between `_id` fields.

```
>>> db.salaries.aggregate( [ { $group: { _id: { fiscal_year:
"$fiscal_year", dept: "$dept" }, max_salary: { $max: "$salary" } } }
, { $merge : { into: { db: "reporting", coll: "budgets" }, on:
"_id",  <a>: "replace", <b>: <c> } } ] )

{ "_id" : { "fiscal_year" : 2017, "dept" : "A" }, "max_salary" : 120000
}
```

In this example, we'll look to find the maximum salary for the department but we'll reuse the same output db and collection. In this exercise, you should change **<a>** and **<b>** to the options for matching and for not matching fields respectively. The value for **<c>** should be set the value that stops an aggregation where there is no match found between the `_id` fields. In this case, as we've already run the previous aggregation this aggregation will find matches.

The result in the code block is what you should see if you are successful.

```
db.salaries.aggregate( [
  { $group: { _id: { fiscal_year: "$fiscal_year", dept:
"$dept" }, max_salary: { $max: "$salary" } } },
  { $merge : { into: { db: "reporting", coll: "budgets" }, on:
"_id",  whenMatched: "replace", whenNotMatched: "fail" } }
] )
```

# Quiz



# Quiz



Which of the following are true for the \$merge aggregation stage in MongoDB? More than one answer choice can be correct.

- ☐ A. \$merge allows for the results to be written to a collection in the same database
- ☐ B. \$merge will only overwrite the existing documents in a collection
- ☐ C. \$merge cannot create a new collection
- ☐ D. \$merge must be the last stage in an aggregation pipeline

# Quiz



Which of the following are true for the \$merge aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$merge allows for the results to be written to a collection in the same database
- ☒ B. \$merge will only overwrite the existing documents in a collection
- ☒ C. \$merge cannot create a new collection
- ☐ D. \$merge must be the last stage in an aggregation pipeline

INCORRECT: \$merge allows for the results to be written to a collection in the same database - This is incorrect. \$merge can write to a collection in a different database.

INCORRECT: \$merge will only overwrite the existing documents in a collection - This is incorrect. It can ignore existing documents in a collection only adding new documents for instance.

INCORRECT: \$merge cannot create a new collection - This is incorrect. \$merge can create a new collection or use an existing collection.

CORRECT: \$merge must be the last stage in an aggregation pipeline - This is correct. \$merge must be the last stage in an aggregation pipeline.



# Quiz

Which of the following are true for the \$merge aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$merge allows for the results to be written to a collection in the same database *This is incorrect. \$merge can write to a collection in a different database.*
- ☒ B. \$merge will only overwrite the existing documents in a collection
- ☒ C. \$merge cannot create a new collection
- ☒ D. \$merge must be the last stage in an aggregation pipeline

INCORRECT: \$merge allows for the results to be written to a collection in the same database - This is incorrect. \$merge can write to a collection in a different database.



# Quiz



Which of the following are true for the \$merge aggregation stage in MongoDB? More than one answer choice can be correct.

- ☒ A. \$merge allows for the results to be written to a collection in the same database
- ☒ B. \$merge will only overwrite the existing documents in a collection
- ☒ C. \$merge cannot create a new collection
- ☒ D. \$merge must be the last stage in an aggregation pipeline

*This is incorrect. It can ignore existing documents in a collection only adding new documents for instance.*

INCORRECT: \$merge will only overwrite the existing documents in a collection - This is incorrect. It can ignore existing documents in a collection only adding new documents for instance.

# Quiz



Which of the following are true for the \$merge aggregation stage in MongoDB? More than 1 answer choice can be correct.

- ☐ A. \$merge allows for the results to be written to a collection in the same database
- ☐ B. \$merge will only overwrite the existing documents in a collection
- ☐ C. \$merge cannot create a new collection
- ☒ D. \$merge must be the last stage in an aggregation pipeline

*This is incorrect.  
\$merge can create a new collection or use an existing collection.*

INCORRECT: \$merge cannot create a new collection - This is incorrect. \$merge can create a new collection or use an existing collection.

# Quiz



Which of the following are true for the \$merge aggregation stage in MongoDB? More than one answer choice can be correct.

- ☐ A. \$merge allows for the results to be written to a collection in the same database
- ☐ B. \$merge will overwrite the existing documents in a collection
- ☐ C. \$merge cannot create a new collection
- ☒ D. \$merge must be the last stage in an aggregation pipeline

*This is correct.  
\$merge must be the  
last stage in an  
aggregation  
pipeline.*

CORRECT: \$merge must be the last stage in an aggregation pipeline - This is correct.  
\$merge must be the last stage in an aggregation pipeline.

## Continue Learning!



[MongoDB University](#) has free self-paced courses and labs ranging from beginner to advanced levels.

## GitHub Student Developer Pack



Sign up for the [MongoDB Student Pack](#) to receive \$50 in Atlas credits and free certification!

This concludes the material for this lesson. However, there are many more ways to learn about MongoDB and non-relational databases, and they are all free! Check out [MongoDB's University](#) page to find free courses that go into more depth about everything MongoDB and non-relational. For students and educators alike, MongoDB for Academia is here to offer support in many forms. Check out our [educator resources](#) and join the Educator Community. Students can receive \$50 in Atlas credits and free certification through the [GitHub Student Developer Pack](#).