MongoDB

- Wylistowanie wszystkich baz z kontenera
- Backup i restore bazy danych - mongodump + mongorestore
- Podstawowa administracja bazą danych - mongosh
- Aggregation pipeline
- Operatory wyrażeń (logika i formatowanie wewnątrz etapów)
- Upgrade silnika bazy danych

Wylistowanie wszystkich baz z kontenera

> Nazwa kontenera może być także ID kontenera

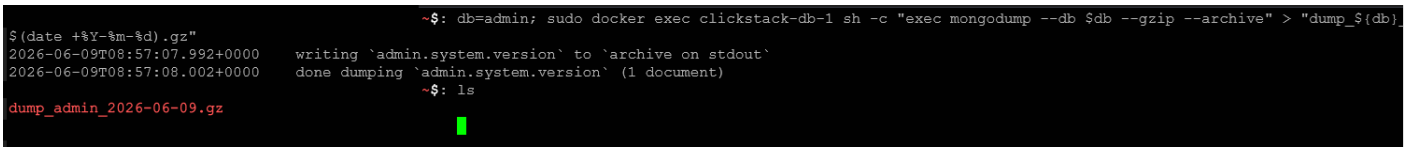
```
sudo docker exec -it <NAZWA KONTENERA> mongosh --
eval "db.getMongo().getDBNames().forEach(function(db) { print(db) });"
```

Backup i restore bazy danych - mongodump + mongorestore

> **Uwaga:** to narzędzia wiersza poleceń hosta, **nie** komendy wykonywane w powłoce mongosh. Przy strumieniowaniu binarnego archiwum przez stdout **nie** należy dodawać `-t` do `docker exec` — alokacja TTY uszkadza strumień gzip.

> Nazwa kontenera może być także ID kontenera

```
db=NAZWABAZY; sudo docker exec NAZWAKONTENERA sh -c "exec mongodump --db $db --gzip --archive" > "backup_${db}_$(date +%Y-%m-%d).gz"
```



Baremetal — narzędzia zainstalowane na hoście:

```
db=NazwaBazy; mongodump --db "$db" --gzip --archive > "backup_${db}_$(date +%Y-%m-%d).gz"
```

Docker — narzędzia w kontenerze, redirect po stronie hosta:

```
db=NAZWABAZY; docker exec NAZWAKONTENERA sh -c "exec mongodump --db $db --gzip --archive" > "backup_${db}_$(date +%Y-%m-%d).gz"
```

Odtworzenie danych z takiego archiwum (wariant dockerowy; baremetal analogicznie bez `docker exec`):

```
docker exec -i NAZWAKONTENERA sh -c "exec mongorestore --gzip --archive" < "NazwaBackupu.gz"
```

> **Screen:** log z udanego `mongodump`.

Podstawowa administracja bazą danych - mongosh

Wprowadzenie:

mongosh to interaktywna powłoka MongoDB oparta na JavaScripcie, służąca do administrowania serwerem i pracy z danymi. Zastąpiła starą powłokę `mongo`. Poniżej zebrano podstawowe komendy administracyjne — każda osobno, z opisem działania. Instrukcja zakłada pracę z hosta; jeśli serwer działa w kontenerze Docker, zobacz sekcję 1, która wyjaśnia jak „przetłumaczyć" każdą komendę uruchomieniową.

1. Dwa konteksty: baremetal vs kontener Docker

Sposób *wejścia* do powłoki zależy od tego, gdzie zainstalowany jest mongosh. Reszta komend (te zaczynające się od `db.` oraz `show ...`) wykonywana jest już wewnątrz powłoki i wygląda tak samo w obu przypadkach.

Baremetal — mongosh zainstalowany bezpośrednio na hoście:

```
mongosh
```

Docker — mongosh znajduje się w kontenerze (np. o nazwie `mongo`), więc komendę poprzedzamy wywołaniem `docker exec`:

```
docker exec -it mongo mongosh
```

Zasada przekładu: dowolną komendę `mongosh ...` z tej instrukcji można uruchomić w kontenerze, poprzedzając ją `docker exec -it <nazwa_kontenera>` (tryb interaktywny) lub samym `docker exec <nazwa_kontenera>` w przypadku jednolinijkowców `--eval`.

2. Komenda `mongosh` – uruchomienie powłoki

```
mongosh
```

Uruchamia interaktywną powłokę i łączy się z domyślnym serwerem (`localhost:27017`). Po połączeniu pojawia się znak zachęty, w który wpisuje się kolejne komendy.

Z uwierzytelnieniem (powłoka zapyta o hasło):

```
mongosh -u admin -p --authenticationDatabase admin
```

Opcja `-u` podaje użytkownika, `-p` bez wartości wymusza interaktywny monit o hasło (bezpieczniej niż wpisywanie go w linii poleceń), a `--authenticationDatabase` wskazuje bazę, w której zdefiniowano konto.

Pojedyncza komenda bez wchodzenia do powłoki (przydatne w skryptach):

```
mongosh --quiet --eval 'db.version()'
```

Opcja `--eval` wykonuje przekazany kod i kończy działanie; `--quiet` wycisza baner powitalny. W kontenerze: `docker exec mongo mongosh --quiet --eval 'db.version()'`.

> **Screen:** ekran po udanym połączeniu z powłoką.

3. Komenda `show dbs` – lista baz danych

```
show dbs
```

Wyświetla listę wszystkich baz danych na serwerze wraz z ich rozmiarem na dysku. Bazy puste (bez zapisanych danych) mogą się nie pojawić na liście.

Przykład:

```
test> show dbs
admin   180.00 KiB
config   60.00 KiB
local    72.00 KiB
```

4. Komenda `use` – wybór bazy danych

```
use admin
```

Przełącza kontekst na wskazaną bazę. Od tego momentu zmienna `db` odnosi się do tej bazy. Jeśli baza nie istnieje, zostanie utworzona dopiero przy pierwszym zapisie danych.

5. Komenda `db` – bieżąca baza

```
db
```

Wyświetla nazwę bazy danych, na której aktualnie pracujemy. Pomaga zorientować się w kontekście przed wykonaniem operacji modyfikujących dane.

6. Komenda `show collections` – lista kolekcji

```
show collections
```

Wyświetla wszystkie kolekcje (odpowiednik tabel) w bieżącej bazie. Wymaga wcześniejszego wybrania bazy poleceniem `use`.

7. Komenda `db.getUsers()` – listowanie użytkowników

```
use admin
db.getUsers()
```

Wyświetla użytkowników zdefiniowanych w bieżącej bazie wraz z ich rolami. Konta administracyjne zwykle żyją w bazie `admin`, dlatego najpierw przełączamy się do niej poleceniem `use admin`.

Skrót shellowy (to samo działanie):

```
show users
```

Pełniejsze informacje o wszystkich kontach:

```
db.runCommand({ usersInfo: 1 })
```

Zwraca szczegółowy wykaz użytkowników bieżącej bazy w formie surowego dokumentu.

> **Screen:** wynik `db.getUsers()` na bazie admin.

8. Komenda `db.createUser()` – tworzenie użytkownika

```
db.createUser({
  user: "backup",
  pwd: passwordPrompt(),
  roles: [ { role: "backup", db: "admin" } ]
})
```

Tworzy nowe konto w bieżącej bazie. Funkcja `passwordPrompt()` powoduje, że powłoka zapyta o hasło interaktywnie, zamiast zapisywać je w historii poleceń. Pole `roles` to lista par rola–baza określająca uprawnienia konta.

9. Komenda `db.dropUser()` – usuwanie użytkownika

```
db.dropUser("backup")
```

Usuwa wskazane konto z bieżącej bazy. Operacja jest nieodwracalna, dlatego należy upewnić się, że pracujemy na właściwej bazie (sprawdź poleceniem `db`).

10. Komendy `db.grantRolesToUser()` / `db.revokeRolesFromUser()` – zarządzanie rolami

```
db.grantRolesToUser("backup", [ { role: "readWrite", db: "appdb" } ])
```

Nadaje istniejącemu użytkownikowi dodatkową rolę na wskazanej bazie — tutaj prawo odczytu i zapisu w bazie `appdb`.

```
db.revokeRolesFromUser("backup", [ { role: "readWrite", db: "appdb" } ])
```

Odbiera wcześniej nadaną rolę. Składnia jest identyczna jak przy nadawaniu.

11. Komenda `db.changeUserPassword()` – zmiana hasła

```
db.changeUserPassword("backup", passwordPrompt())
```

Zmienia hasło istniejącego użytkownika. Ponownie warto użyć `passwordPrompt()`, aby nie ujawniać hasła w linii poleceń ani w historii.

12. Komenda `db.getRoles()` – przegląd ról

```
db.getRoles()
```

Wyświetla role zdefiniowane w bieżącej bazie. Oprócz ról własnych MongoDB udostępnia role wbudowane, których ta komenda standardowo nie pokazuje.

> **Typowe role wbudowane:** `read`, `readWrite`, `dbAdmin`, `userAdmin`, `dbOwner` (na konkretnej bazie) oraz globalne `readWriteAnyDatabase`, `userAdminAnyDatabase`, `backup`, `restore`, `root`.

13. Komenda `db.runCommand({ connectionStatus: 1 })` – kim jestem

```
db.runCommand({ connectionStatus: 1 })
```

Zwraca informacje o bieżącym połączeniu: zalogowanego użytkownika oraz przyznane mu role. Przydatne do szybkiej weryfikacji, czy uwierzytelnienie powiodło się i z jakimi uprawnieniami pracujemy.

14. Komenda `db.stats()` – statystyki bazy

```
db.stats()
```

Wyświetla statystyki bieżącej bazy: liczbę kolekcji, liczbę dokumentów oraz zajętość miejsca na dysku. Pomocne przy ocenie rozmiaru bazy przed wykonaniem kopii zapasowej.

> **Screen:** wynik `db.stats()`.

15. Komenda `db.serverStatus()` – status serwera

```
db.serverStatus()
```

Zwraca obszerny dokument z metrykami całego serwera: połączenia, zużycie pamięci, operacje, stan replikacji. Wynik jest bardzo duży, dlatego zwykle odczytuje się z niego pojedyncze pola, np. `db.serverStatus().connections`.

16. Komenda `db.<kolekcja>.find()` – wyszukiwanie dokumentów

```
db.uzytkownicy.find()
```

Wyświetla dokumenty z kolekcji. Bez argumentów zwraca wszystkie; w mongosh wyniki są domyślnie czytelnie sformatowane.

Z filtrem i operatorami porównań:

```
db.uzytkownicy.find({ wiek: { $gt: 25 } })          // wiek > 25
db.uzytkownicy.find({ wiek: { $gte: 18, $lt: 65 } }) // zakres
db.uzytkownicy.find({ imie: { $in: ["Anna", "Jan"] } })
```

Operatory `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, `$in` pozwalają budować warunki bardziej złożone niż proste dopasowanie wartości.

> **Screen:** przykładowy wynik `find()`.

17. Komendy `db.<kolekcja>.insertOne()` / `insertMany()`

– wstawianie

```
db.uzytkownicy.insertOne({ imie: "Anna", wiek: 30 })
```

Wstawia pojedynczy dokument do kolekcji. Jeśli kolekcja nie istnieje, zostanie utworzona automatycznie.

```
db.uzytkownicy.insertMany([
  { imie: "Jan", wiek: 41 },
  { imie: "Ola", wiek: 22 }
])
```

Wstawia wiele dokumentów jednocześnie, przekazanych jako tablica.

18. Komendy

`db.<kolekcja>.updateOne()` / `updateMany()` – aktualizacja

```
db.uzytkownicy.updateOne(
  { imie: "Anna" },
  { $set: { wiek: 31 } }
)
```

Aktualizuje pierwszy dokument pasujący do filtra. Operator `$set` ustawia wartość pola; inne przydatne operatory to `$inc` (zwiększ), `$unset` (usuń pole), `$push` i `$pull` (operacje na tablicach).

```
db.uzytkownicy.updateMany(
  { wiek: { $lt: 18 } },
  { $set: { maloletni: true } }
)
```

Aktualizuje wszystkie dokumenty pasujące do filtra. Dodanie opcji `{ upsert: true }` jako trzeciego argumentu sprawia, że dokument zostanie utworzony, jeśli żaden nie pasuje.

19. Komendy

`db.<kolekcja>.deleteOne()` / `deleteMany()` – usuwanie

```
db.uzytkownicy.deleteOne({ imie: "Jan" })
```

Usuwa pierwszy dokument pasujący do filtra.

```
db.uzytkownicy.deleteMany({ wiek: { $lt: 18 } })
```

Usuwa wszystkie dokumenty pasujące do filtra. Pusty filtr `{}` usunąłby całą zawartość kolekcji, dlatego należy zachować ostrożność.

20. Komenda

`db.<kolekcja>.createIndex()` – indeksy

```
db.uzytkownicy.createIndex({ imie: 1 })
```

Tworzy indeks na wskazanym polu (`1` = rosnąco, `-1` = malejąco), co przyspiesza wyszukiwanie po tym polu.

```
db.uzytkownicy.createIndex({ email: 1 }, { unique: true })
```

Tworzy indeks z wymuszeniem unikalności wartości w danym polu.

```
db.uzytkownicy.getIndexes()
db.uzytkownicy.dropIndex("imie_1")
```

Pierwsza komenda wypisuje istniejące indeksy kolekcji, druga usuwa indeks po jego nazwie.

Podsumowanie praktyczne

```
# Wejście do powłoki (baremetal)
mongosh -u admin -p --authenticationDatabase admin

# ...lub w kontenerze
docker exec -it mongo mongosh -u admin -p --authenticationDatabase admin
```

```
// Wewnątrz powłoki: rozeznanie i zarządzanie userami
show dbs
use admin
db.getUsers()
db.createUser({ user: "backup", pwd: passwordPrompt(), roles: [ { role: "backup", db: "admin" } ] })
db.runCommand({ connectionStatus: 1 })
```

Aggregation pipeline

Wprowadzenie:

> Aggregation pipeline to mechanizm przetwarzania danych w MongoDB, działający na zasadzie potoku (jak `|` w powłoce Linux). Dokumenty przechodzą przez kolejne etapy (*stages*), z których każdy przekształca dane i przekazuje wynik do następnego. Pozwala to grupować, filtrować, łączyć i przeliczać dane po stronie serwera, bez wyciągania ich do aplikacji.

Wszystkie poniższe komendy wykonuje się **wewnątrz powłoki mongosh**, więc wyglądają tak samo niezależnie od tego, czy serwer działa baremetal, czy w kontenerze (sposób wejścia do powłoki opisano w instrukcji „Podstawowe komendy mongosh", sekcja 1).

1. Struktura potoku – `aggregate()`

```
db.zamowienia.aggregate([
  { $match: { status: "oplacone" } },
  { $group: { _id: "$klient", suma: { $sum: "$kwota" } } },
  { $sort: { suma: -1 } }
])
```

Metoda `aggregate()` przyjmuje tablicę etapów. Kolejność ma znaczenie — dokumenty płyną od pierwszego etapu do ostatniego. Powyższy przykład czyta się jak zdanie: „wybierz opłacone zamówienia, zsumuj kwoty per klient, posortuj malejąco po sumie". Odwołania do pól dokumentu zapisuje się z prefiksem `$`, np. `"$kwota"`.

2. Etap `$match` – filtrowanie

```
db.zamowienia.aggregate([
  { $match: { kategoria: "elektronika", kwota: { $gt: 100 } } }
])
```

Przepuszcza dalej tylko dokumenty spełniające warunek. Składnia jest identyczna jak w zwykłym `find()` i obsługuje te same operatory (`$gt`, `$in` itd.). Najlepiej umieszczać `$match` na początku

potoku — ogranicza liczbę dokumentów przetwarzanych przez dalsze, cięższe etapy.

3. Etap `$project` – wybór i przekształcanie pól

```
db.zamowienia.aggregate([
  { $project: { _id: 0, klient: 1, kwota: 1 } }
])
```

Decyduje, które pola znajdą się w wyniku (`1` = pokaż, `0` = ukryj). Pozwala też tworzyć pola wyliczane:

```
db.zamowienia.aggregate([
  { $project: { klient: 1, kwotaBrutto: { $multiply: ["$kwota", 1.23] } } }
])
```

Tutaj powstaje nowe pole `kwotaBrutto` jako iloczyn istniejącej kwoty i stawki — przykład wyrażenia obliczeniowego wewnątrz potoku.

4. Etap `$group` – grupowanie i agregacja

```
db.zamowienia.aggregate([
  { $group: {
      _id: "$klient",
      liczba: { $sum: 1 },
      sumaKwot: { $sum: "$kwota" },
      srednia: { $avg: "$kwota" },
      maks: { $max: "$kwota" }
  } }
])
```

Najważniejszy etap. Pole `_id` określa klucz grupowania (tutaj: per klient). Pozostałe pola to *akumulatory* liczone w obrębie grupy: `$sum: 1` zlicza dokumenty, `$sum: "$kwota"` sumuje kwoty, a `$avg`, `$min`, `$max` liczą odpowiednio średnią i skrajne wartości. Ustawienie `_id: null` grupuje

wszystkie dokumenty w jedną całość (agregacja globalna).

Przykład wyniku:

```
appdb> ...
[
  { _id: 'Anna', liczba: 3, sumaKwot: 540, srednia: 180, maks: 300 },
  { _id: 'Jan',  liczba: 1, sumaKwot: 120, srednia: 120, maks: 120 }
]
```

> **Akumulatory do zapamiętania:** `$sum`, `$avg`, `$min`, `$max`, `$first`, `$last`, `$push` (zbiera wartości do tablicy) oraz `$addToSet` (jak `$push`, ale bez duplikatów).

> **Screen:** wynik grupowania per klient.

5. Etap `$sort` – sortowanie

```
db.zamowienia.aggregate([
  { $group: { _id: "$klient", sumaKwot: { $sum: "$kwota" } } },
  { $sort: { sumaKwot: -1 } }
])
```

Porządkuje dokumenty: `1` = rosnąco, `-1` = malejąco. Można sortować również po polach wyliczonych we wcześniejszych etapach (jak tutaj po `sumaKwot`).

6. Etapy `$limit` i `$skip` – ograniczanie wyników

```
db.zamowienia.aggregate([
  { $sort: { kwota: -1 } },
  { $skip: 10 },
  { $limit: 5 }
])
```

`$limit` zwraca co najwyżej podaną liczbę dokumentów, a `$skip` pomija pierwsze N. W połączeniu z `$sort` służą do stronicowania wyników. `$limit` warto stawiać możliwie wcześnie, by odciążyć dalsze etapy.

7. Etap `$count` – zliczanie dokumentów

```
db.zamowienia.aggregate([
  { $match: { status: "oplacone" } },
  { $count: "liczba_oplaconych" }
])
```

Zwraca pojedynczy dokument z liczbą dokumentów, które dotarły do tego etapu. Argument to nazwa pola, w którym znajdzie się wynik.

8. Etap `$unwind` – rozwijanie tablic

```
db.zamowienia.aggregate([
  { $unwind: "$produkty" }
])
```

Rozbija dokument zawierający tablicę na wiele dokumentów — po jednym na każdy element tablicy. Jeśli zamówienie miało trzy produkty w polu `produkty`, powstaną trzy dokumenty, każdy z pojedynczym produktem. Niezbędne, gdy chcemy grupować lub liczyć po elementach tablicy.

9. Etap `$lookup` – łączenie kolekcji (JOIN)

```
db.zamowienia.aggregate([
  { $lookup: {
      from: "klienci",
      localField: "klient_id",
      foreignField: "_id",
```

```
    as: "dane_klienta"
  } }
])
```

Dołącza dane z innej kolekcji — odpowiednik lewego złączenia (LEFT JOIN). Dla każdego zamówienia szuka w kolekcji `klienci` dokumentów, w których `_id` równa się polu `klient_id` z zamówienia. Dopasowania trafiają do nowego pola `dane_klienta` jako **tablica** — często łączy się `$lookup` z `$unwind`, by ją spłaszczyć.

10. Etapy `$addFields` / `$set` – dodawanie pól wyliczanych

```
db.zamowienia.aggregate([
  { $addFields: { kwotaBrutto: { $multiply: ["$kwota", 1.23] } } }
])
```

Dodaje nowe pole do dokumentu, zachowując wszystkie dotychczasowe (w przeciwieństwie do `$project`, gdzie pola trzeba wymienić jawnie). `$set` to alias `$addFields` — działa identycznie i bywa czytelniejszy.

11. Etapy `$out` / `$merge` – zapis wyników

```
db.zamowienia.aggregate([
  { $group: { _id: "$klient", suma: { $sum: "$kwota" } } },
  { $out: "raport_klientow" }
])
```

Zapisuje wynik potoku do osobnej kolekcji zamiast wyświetlać go na ekranie — przydatne do budowania raportów cyklicznych.

> **Uwaga:** `$out` **nadpisuje** całą kolekcję docelową. Jeśli chcesz aktualizować/dołączać dokumenty zamiast podmieniać całość, użyj `$merge`, który wykonuje upsert na istniejącej kolekcji.

12. Przykład złożony – pełny potok

```
db.zamowienia.aggregate([
  { $match: { data: { $gte: ISODate("2026-01-01") } } },
  { $group: { _id: "$klient", suma: { $sum: "$kwota" }, liczba: { $sum: 1 } } },
  { $match: { suma: { $gt: 1000 } } },
  { $sort: { suma: -1 } },
  { $limit: 5 }
])
```

Potok czyta się jak zapytanie biznesowe: „spośród zamówień od początku 2026 roku zsumuj kwoty i policz zamówienia per klient, zostaw tylko klientów z sumą powyżej 1000, posortuj malejąco i pokaż pięciu największych". Zwróć uwagę na drugi `$match` — filtruje już po polu `suma` wyliczonym w `$group`, czego zwykły `find()` nie potrafi.

> **Screen:** wynik pełnego potoku (top 5 klientów).

13. Dobre praktyki i wydajność

Kilka zasad, które warto stosować przy budowaniu potoków:

- **Filtruj wcześnie** – `$match` i `$limit` na początku ograniczają liczbę dokumentów przekazywanych do cięższych etapów (`$group`, `$lookup`).
- **Indeksy** – potok korzysta z indeksów tylko dla `$match` i `$sort` umieszczonych na *początku*, zanim dane zostaną przekształcone.
- **Analiza planu** – sprawdź, jak MongoDB wykona potok:

```
db.zamowienia.aggregate([ /* etapy */ ], { explain: true })
```

- **Duże operacje** – przy obszernych `$group` / `$sort` przekraczających limit pamięci dołącz opcję pozwalającą korzystać z dysku:

```
db.zamowienia.aggregate([ /* etapy */ ], { allowDiskUse: true })
```

Podsumowanie praktyczne

```
// Szkielet potoku: filtruj → grupuj → filtruj po agregacie → sortuj → ogranicz
db.zamowienia.aggregate([
  { $match: { status: "oplacone" } },
  { $group: { _id: "$klient", suma: { $sum: "$kwota" } } },
  { $match: { suma: { $gt: 1000 } } },
  { $sort: { suma: -1 } },
  { $limit: 10 }
])
```

Operatory wyrażeń (logika i formatowanie wewnątrz etapów)

Wprowadzenie:

> Operatory wyrażeń to „funkcje" używane **wewnątrz etapów potoku** (najczęściej `$project`, `$addFields` i `$group`) do obliczania wartości — warunków, działań arytmetycznych, formatowania tekstu i dat. Różnią się od operatorów zapytań (tych z `find()` i `$match`), które jedynie filtrują dokumenty. Operatory wyrażeń *tworzą* nowe wartości.

Wszystkie poniższe komendy wykonuje się w powłoce mongosh, w ramach potoku agregacji (zob. strona „Aggregation pipelines").

> **Kluczowa różnica składni:** operatory porównań w wyrażeniach przyjmują argumenty w **tablicy**, np. `{ $gt: ["$kwota", 100] }`. To nie to samo co forma zapytania `{ kwota: { $gt: 100 } }` używana w `$match` / `find()`. Mylenie tych dwóch form to najczęstszy błąd początkujących.

1. Operator `$cond` – warunek if/then/else

```
db.zamowienia.aggregate([
  { $addFields: {
      kategoriaKwoty: {
        $cond: { if: { $gte: ["$kwota", 500] }, then: "duze", else: "male" }
      }
  } }
])
```

Zwraca jedną z dwóch wartości w zależności od warunku — odpowiednik `if/else`. Powyżej każdemu zamówieniu dopisywane jest pole `kategoriaKwoty` równe `"duze"`, gdy kwota wynosi co najmniej 500, w przeciwnym razie `"male"`.

Skrócona składnia tablicowa (kolejność: warunek, then, else):

```
{ $cond: [ { $gte: ["$kwota", 500] }, "duze", "male" ] }
```

> **Screen:** wynik z dopisanym polem `kategoriaKwoty`.

2. Operator `$switch` – wielokrotny wybór

```
db.zamowienia.aggregate([
  { $addFields: {
      prog: {
        $switch: {
          branches: [
            { case: { $gte: ["$kwota", 1000] }, then: "premium" },
            { case: { $gte: ["$kwota", 500] },  then: "standard" }
          ],
          default: "podstawowy"
        }
      }
  } }
])
```

Sprawdza kolejno warunki z listy `branches` i zwraca wynik pierwszego pasującego (`then`). Jeśli żaden nie pasuje, zwraca `default`. To czytelniejsza alternatywa dla wielu zagnieżdżonych `$cond`. Warunki są sprawdzane od góry, więc kolejność ma znaczenie.

3. Operator `$ifNull` – wartość domyślna dla braków

```
db.zamowienia.aggregate([
  { $addFields: { rabat: { $ifNull: ["$rabat", 0] } } }
])
```

Zwraca pierwszą wartość, jeśli nie jest ona `null` ani nie brakuje jej w dokumencie; w przeciwnym razie zwraca wartość zastępczą. Tutaj brakujące pole `rabat` zostaje uzupełnione zerem, co zabezpiecza dalsze obliczenia przed błędami na wartościach `null`.

> **Screen:** dokument z uzupełnionym polem `rabat`.

4. Operatory porównań w wyrażeniach

```
{ $eq:  ["$status", "oplacone"] }   // równe
{ $ne:  ["$status", "anulowane"] }  // różne
{ $gt:  ["$kwota", 100] }           // wiékse niż (>)
{ $gte: ["$kwota", 100] }           // wiékse lub równe
{ $lt:  ["$kwota", 100] }           // mniejsze niż
{ $cmp: ["$a", "$b"] }              // -1, 0 lub 1
```

Zwracają wartość logiczną (`true` / `false`), więc używa się ich zwykle jako warunku w `$cond` lub `$switch`. Pamiętaj o składni tablicowej — dwa porównywane wyrażenia podaje się jako elementy tablicy.

5. Operatory logiczne – `$and`, `$or`, `$not`

```
{ $and: [ { $gte: ["$kwota", 100] }, { $eq: ["$status", "oplacone"] } ] }
{ $or:  [ { $eq: ["$vip", true] }, { $gt: ["$kwota", 1000] } ] }
{ $not: [ { $eq: ["$status", "anulowane"] } ] }
```

Łączą kilka warunków logicznych. Przyjmują tablicę wyrażeń i zwracają wartość logiczną — przydatne do budowania złożonych warunków w `$cond`.

6. Operatory arytmetyczne

```
{ $add:      ["$kwota", "$rabat"] }   // dodawanie
{ $subtract: ["$kwota", "$rabat"] }   // odejmowanie
{ $multiply: ["$kwota", 1.23] }       // mnożenie (np. brutto)
{ $divide:   ["$suma", "$liczba"] }   // dzielenie
{ $mod:      ["$liczba", 2] }         // reszta z dzielenia
{ $round:    ["$kwota", 2] }          // zaokrąglenie do 2 miejsc
```

Wykonują działania na liczbach. Argumentami mogą być zarówno pola dokumentu (z prefiksem `$`), jak i wartości stałe. Dostępne są też `$abs`, `$ceil` i `$floor`.

7. Operatory łańcuchowe (tekstowe)

```
{ $concat:  ["$imie", " ", "$nazwisko"] }  // sklejanie tekstu
{ $toUpper: "$status" }                    // wielkie litery
{ $toLower: "$email" }                     // małe litery
{ $substr:  ["$kod", 0, 3] }               // wycinek (od, długość)
{ $split:   ["$email", "@"] }              // podział na tablicé
{ $trim:    { input: "$nazwa" } }          // usuniécie spacji z brzegów
```

Służą do manipulacji tekstem. `$concat` łączy fragmenty (uwaga: jeśli któryś jest `null`, wynik też będzie `null` — warto opakować w `$ifNull`). `$split` zwraca tablicę, więc często łączy się go z operatorami tablicowymi.

8. Operator `$dateToString` – formatowanie dat

```
db.zamowienia.aggregate([
  { $addFields: {
      dataTekst: {
        $dateToString: { format: "%Y-%m-%d", date: "$data", timezone: "Europe/Warsaw" }
```

```
    }
  } }
])
```

Zamienia pole typu data na sformatowany tekst według wzorca `format`. Opcjonalny `timezone` przelicza datę na wskazaną strefę przed sformatowaniem (bez niego daty są w UTC).

Najczęstsze symbole formatu:

```
%Y  rok (4 cyfry)      %H  godzina (00-23)
%m  miesiąc (01-12)    %M  minuty (00-59)
%d  dzień (01-31)      %S  sekundy (00-59)
```

> **Screen:** dokument z polem `dataTekst` w formacie RRRR-MM-DD.

9. Operatory na datach – składowe i obliczenia

```
{ $year:      "$data" }   // sam rok
{ $month:     "$data" }   // sam miesiąc
{ $dayOfMonth: "$data" }   // sam dzień
```

Wyciągają pojedynczy składnik daty jako liczbę. Często używane w `$group` do grupowania np. po roku lub miesiącu.

Różnica i przesunięcie dat:

```
{ $dateDiff: { startDate: "$data", endDate: "$$NOW", unit: "day" } }
{ $dateAdd:  { startDate: "$data", unit: "day", amount: 30 } }
```

`$dateDiff` liczy odstęp między dwiema datami w zadanej jednostce (zmienna `$$NOW` to bieżący czas serwera), a `$dateAdd` dodaje określony okres do daty.

10. Operatory tablicowe

```
{ $size:       "$produkty" }              // liczba elementów
{ $arrayElemAt: ["$produkty", 0] }        // element o indeksie
{ $first:      "$produkty" }              // pierwszy element
{ $last:       "$produkty" }              // ostatni element
```

Operują na polach będących tablicami. `$size` bywa przydatne np. po `$lookup`, by policzyć liczbę dopasowań.

Przekształcanie i filtrowanie tablic:

```
// zostaw tylko produkty droższe niż 100
{ $filter: { input: "$produkty", as: "p", cond: { $gt: ["$$p.cena", 100] } } }


// wyciągnij samą nazwé z każdego produktu
{ $map: { input: "$produkty", as: "p", in: "$$p.nazwa" } }
```

`$filter` zwraca podzbiór tablicy spełniający warunek, a `$map` przekształca każdy element. W obu zmienna pętli (tu `$$p`) ma podwójny prefiks `$$`, bo odnosi się do zmiennej lokalnej, a nie do pola dokumentu.

11. Konwersje typów

```
{ $toInt:    "$kwotaTekst" }   // tekst -> liczba całkowita
{ $toDouble: "$kwotaTekst" }   // tekst -> liczba zmiennoprzecinkowa
{ $toString: "$kwota" }        // liczba -> tekst
{ $toDate:   "$dataTekst" }    // tekst -> data
{ $type:     "$kwota" }        // zwraca nazwé typu pola
```

Zamieniają wartości między typami — niezbędne przy porządkowaniu danych zaimportowanych jako tekst. Bardziej elastyczny jest `$convert`, który pozwala wskazać typ docelowy i wartość zastępczą na wypadek błędu konwersji:

```
{ $convert: { input: "$kwotaTekst", to: "double", onError: 0, onNull: 0 } }
```

12. Przykład złożony – operatory w akcji

```
db.zamowienia.aggregate([
  { $addFields: {
      brutto: { $round: [ { $multiply: ["$kwota", 1.23] }, 2 ] },
      miesiac: { $dateToString: { format: "%Y-%m", date: "$data" } },
      etykieta: {
        $cond: [ { $gte: ["$kwota", 500] }, "VIP", "zwykly" ]
      },
      rabat: { $ifNull: ["$rabat", 0] }
  } }
])
```

Jeden etap `$addFields` liczy kwotę brutto (zaokrągloną), wyciąga miesiąc zamówienia jako tekst, nadaje etykietę na podstawie progu kwoty i uzupełnia brakujący rabat zerem. Pokazuje, jak operatory wyrażeń składa się ze sobą, by w jednym przebiegu przygotować dane do raportu.

> **Screen:** wynik z czterema wyliczonymi polami.

Podsumowanie praktyczne

```
// Najczęściej używane operatory wyrażeń
{ $cond:   [ { $gte: ["$kwota", 500] }, "duze", "male" ] }   // warunek
{ $ifNull: ["$rabat", 0] }                                   // wartość domyślna
{ $dateToString: { format: "%Y-%m-%d", date: "$data" } }     // formatowanie daty
{ $concat: ["$imie", " ", "$nazwisko"] }                     // sklejanie tekstu
{ $round:  [ { $multiply: ["$kwota", 1.23] }, 2 ] }          // arytmetyka
```

Upgrade silnika bazy danych

Wprowadzenie:

> Aktualizacja silnika MongoDB między wersjami głównymi (np. 5.0 → 6.0) to kontrolowany proces w kilku krokach: najpierw upewniamy się, że poziom kompatybilności (FCV — Feature Compatibility Version) odpowiada wersji obecnej, potem podmieniamy silnik, a dopiero po sprawdzeniu, że nowa wersja działa, podnosimy FCV. Taka kolejność pozwala w razie problemów wrócić do poprzedniej wersji. Instrukcja opisuje **jeden skok** między sąsiednimi wersjami głównymi dla wdrożenia opartego na Docker Compose. Wariant baremetal różni się wyłącznie krokiem podmiany silnika — zaznaczono to w notkach. W przykładach `NAZWAKONTENERA` i `SERVICENAME` zastąp wartościami z własnego środowiska.

1. Zasada pojedynczego skoku

MongoDB pozwala aktualizować silnik tylko o **jedną wersję główną naraz**. Nie da się przeskoczyć np. z 5.0 od razu do 7.0 — trzeba przejść przez każdą wersję pośrednią:

```
5.0 → 6.0 → 7.0 → 8.0
```

Każdy skok to powtórzenie całej procedury opisanej poniżej. Dodatkowo FCV przed aktualizacją musi odpowiadać wersji obecnej — dlatego krok 1 to jego ustawienie/weryfikacja.

> **Najpierw kopia zapasowa.** Przed jakąkolwiek aktualizacją wykonaj `mongodump` (zob. instrukcja „Podstawowe komendy mongosh", sekcja 21). Aktualizacja zmienia format danych na dysku i bez backupu rollback bywa niemożliwy.

2. Krok 1 – ustawienie FCV na wersję obecną

Wejdź do powłoki mongosh w działającym kontenerze:

```
docker exec -it NAZWAKONTENERA mongosh
```

Sprawdź bieżący poziom kompatybilności:

```
db.adminCommand({ getParameter: 1, featureCompatibilityVersion: 1 })
```

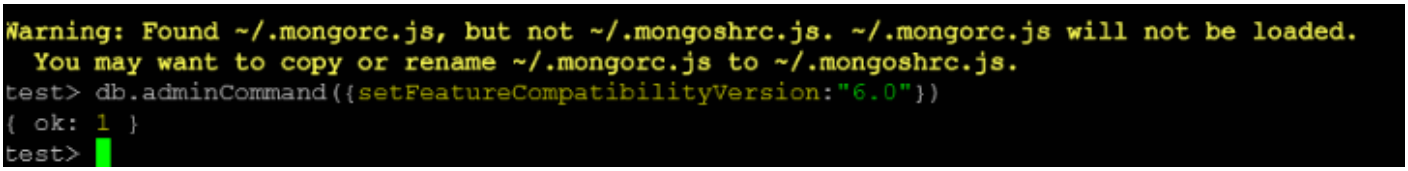
Następnie jawnie ustaw FCV na wersję, na której obecnie pracujesz (tu 5.0). To gwarantuje, że dane są w pełni zgodne z bieżącą wersją, zanim podmienisz silnik:

```
db.adminCommand({ setFeatureCompatibilityVersion: "5.0" })
```

Oczekiwany wynik:

```
{ "ok" : 1 }
```

> **Baremetal:** kroki FCV są identyczne — wchodzisz do powłoki po prostu poleceniem `mongosh` (bez `docker exec`).



3. Krok 2 – zmiana obrazu w `docker-compose.yaml`

W pliku `docker-compose.yaml` podnieś tag obrazu usługi bazy o jedną wersję główną:

```
SERVICENAME:
  image: mongo:6.0
  container_name: NAZWAKONTENERA
```

4. Krok 3 – restart kontenerów

```
docker compose down
docker compose pull
docker compose up -d
```

`down` zatrzymuje i usuwa kontenery (dane pozostają na woluminie), `pull` pobiera nowy obraz w zadeklarowanej wersji, a `up -d` uruchamia kontenery w tle na nowym silniku. Wolumin z danymi nie jest kasowany, więc baza startuje z dotychczasową zawartością.

> **Baremetal:** odpowiednikiem kroków 2–3 jest aktualizacja pakietu z repozytorium MongoDB. W tym samym miejscu procedury (po ustawieniu FCV na wersję obecną i wykonaniu kopii, a przed podniesieniem FCV) przełączasz repozytorium MongoDB na nową wersję główną, zatrzymujesz usługę bazy, instalujesz nowy pakiet i uruchamiasz usługę ponownie. Dane i konfiguracja na dysku pozostają nietknięte — podmieniasz wyłącznie binaria.

> **Screen:** log z `docker compose down` / `pull` / `up -d`.

5. Krok 4 – weryfikacja uruchomienia

Sprawdź, że kontenery wstały poprawnie:

```
docker compose ps
```

Następnie potwierdź, że baza działa już na nowej wersji silnika:

```
docker exec -it NAZWAKONTENERA mongosh --quiet --eval 'db.version()'
```

```
~$: sudo docker exec -it clickstack-db-1 mongosh --quiet --eval 'db.version()'
8.2.9
```

Na tym etapie warto pozwolić bazie popracować chwilę („burn-in"), aby upewnić się, że nowa wersja działa stabilnie — dopóki nie podniesiesz FCV, możesz jeszcze wrócić do poprzedniego silnika.

>

> **Baremetal:** stan sprawdzasz przez menedżer usług systemu zamiast `docker compose ps`; wersję — poleceniem `mongosh --quiet --eval 'db.version()'`.

> **Screen:** wynik `docker compose ps` ze statusem *Up*.

6. Krok 5 – podniesienie FCV na nową wersję

Gdy nowa wersja działa poprawnie, wejdź ponownie do powłoki i podnieś FCV do wersji docelowej — to odblokowuje funkcje nowej wersji:

```
db.adminCommand({ setFeatureCompatibilityVersion: "6.0" })
```

Po tej operacji aktualizacja pojedynczego skoku jest zakończona. Jeśli planujesz kolejny skok, wróć do kroku 1 z nowymi wartościami wersji.

> **Od MongoDB 7.0 wymagane jest pole `confirm: true`** — bez niego komenda zwróci błąd. Dla skoków do 7.0 i wyższych użyj formy:

```
db.adminCommand({ setFeatureCompatibilityVersion: "7.0", confirm: true })
```

> Przy skokach do 5.0 i 6.0 parametr `confirm` nie jest potrzebny (i na starszych binariach może zostać odrzucony jako nieznane pole).

> **Rollback:** aby cofnąć się do poprzedniej wersji silnika, najpierw obniż FCV do wersji wcześniejszej (`setFeatureCompatibilityVersion` ze starszą wartością), a dopiero potem podmień obraz / pakiet z powrotem. Starsza wersja nie wystartuje, jeśli FCV wskazuje wersję nowszą.

Podsumowanie praktyczne

Pełny cykl jednego skoku (na przykładzie 5.0 → 6.0):

```
# 1. FCV na wersję obecną (wewnątrz mongosh)
docker exec -it NAZWAKONTENERA mongosh --eval 'db.adminCommand({ setFeatureCompatibilityVersion: "5.0"
```

```
})'

# 2. docker-compose.yaml:  image: mongo:6.0
#    (baremetal: przełącz repozytorium MongoDB na 6.0 i zaktualizuj pakiet)

# 3. restart na nowy silnik
docker compose down
docker compose pull
docker compose up -d

# 4. weryfikacja
docker compose ps
docker exec -it NAZWAKONTENERA mongosh --quiet --eval 'db.version()'

# 5. FCV na wersję nową (confirm:true od 7.0 wzwyż)
docker exec -it NAZWAKONTENERA mongosh --eval 'db.adminCommand({ setFeatureCompatibilityVersion: "6.0"
})'
```