

## Wykorzystanie bazy dokumentowej MongoDB

Krzysztof Wyszynski

### 1. Zbiór danych

Podczas realizacji projektu wykorzystany zostanie zbiór danych dostępny pod adresem: <https://www.kaggle.com/karangadiya/fifa19>. Składa się on ze szczegółowych danych dotyczących zawodników w grze FIFA 19 – posiada ponad 18000 rekordów oraz 89 kolumn. Ponieważ dane nie są powiązane relacjami, stanowią dobrą próbkę dla bazy dokumentowej, jaką jest MongoDB.

Dane wstępnie pozyskane z pliku *data.csv* zostaną zmodyfikowane w taki sposób, aby w bazie danych wykorzystać zarówno dokumenty zagnieżdżone, jak i referencje do powiązania danych. Pierwotna forma danych nie jest spójna – każdy zawodnik ma przypisane informacje o klubie, a zatem jeśli zawodnik byłby dokumentem zawierającym pola dotyczące klubu, zmiana informacji o klubie w jednym dokumencie nie wpłynęłaby na pozostałe dokumenty – występuje redundancja. Wyeliminować ten problem można na dwa sposoby – stworzyć dokument klubu piłkarskiego agregujący zawodników (1 kolekcja), bądź stworzyć dwie kolekcje przechowujące oddzielnie informacje o klubach i zawodnikach, a każdy zawodnik odwoływałby się do danego klubu poprzez referencję.

### 2. Instalacja i konfiguracja MongoDB

Za pomocą instrukcji zawartych pod adresem: <https://docs.mongodb.com/manual/installation/> zainstalowano MongoDB. Warto podkreślić, że MongoDB oferuje również darmowe usługi chmurowe w ograniczonym zakresie w platformie MongoDB Atlas. W wersji lokalnej po zakończonej instalacji do zarządzania bazą danych wykorzystać można interfejs graficzny *MongoDB Compass*. Aby połączyć się z bazą danych za pomocą aplikacji, wystarczy ustawić nowe, domyślne połączenie lokalne:

# New Connection

☆ FAVORITE

[Paste connection string](#)

Hostname

More Options

Hostname

localhost

Port

27017

SRV Record

☐

Authentication

None

Connect

Następnie można już przystąpić do utworzenia nowej bazy danych.

### 3. Tworzenie bazy danych i kolekcji

Za pomocą narzędzia *MongoDB Compass* stworzona została baza danych o nazwie *fifa19* oraz kolekcja *clubs\_nested*, która przechowywać będzie informacje o klubach i zawodnikach w formie zagnieżdżonej. Baza danych nie może zostać utworzona bez zadeklarowania wstępnej kolekcji.

**Create Database**

**Database Name**

fifa19

**Collection Name**

clubs\_nested

☐ **Capped Collection**  
Fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. ⓘ

☐ **Use Custom Collation**  
Collation allows users to specify language-specific rules for string comparison, such as rules for lettercase and accent marks. ⓘ

☐ **Time-Series**  
Time-series collections efficiently store sequences of measurements over a period of time.


Cancel Create Database

W rezultacie utworzona została nowa baza danych z domyślną kolekcją. Za pomocą narzędzia *MongoDB Compass* można w prosty sposób zaimportować dane z pliku CSV bądź JSON. W przypadku wybranego przeze mnie zbioru danych utworzenie odpowiednich kolekcji według wstępnych założeń nie będzie trywialne.

### 3.1. Model referencyjny

Pierwszym krokiem będzie wydzielenie z pliku *data.csv* informacji o klubach i zawodnikach oraz utworzenie odpowiednich kolekcji. Stworzenie kolekcji *clubs* jest bardzo proste – wystarczy wykorzystać wcześniej wspomniany import pliku CSV:

### Select File

 clubs.csv

### Select Input File Type

JSON

CSV

### Options

Select delimiter SEMICOLON ▾

☒ Ignore empty strings

☐ Stop on errors

### Specify Fields and Types

	<input checked="" type="checkbox"/> name <span>String ▾</span>	<input checked="" type="checkbox"/> logo <span>String ▾</span>
1	FC Barcelona	<a href="https://cdn.sofifa.org/teams/2/light/241.png">https://cdn.sofifa.org/teams/2/light/241.png</a>
2	Juventus	<a href="https://cdn.sofifa.org/teams/2/light/45.png">https://cdn.sofifa.org/teams/2/light/45.png</a>
3	Paris Saint-Germain	<a href="https://cdn.sofifa.org/teams/2/light/73.png">https://cdn.sofifa.org/teams/2/light/73.png</a>
4	Manchester United	<a href="https://cdn.sofifa.org/teams/2/light/11.png">https://cdn.sofifa.org/teams/2/light/11.png</a>
5	Manchester City	<a href="https://cdn.sofifa.org/teams/2/light/10.png">https://cdn.sofifa.org/teams/2/light/10.png</a>
6	Chelsea	<a href="https://cdn.sofifa.org/teams/2/light/5.png">https://cdn.sofifa.org/teams/2/light/5.png</a>
7	Real Madrid	<a href="https://cdn.sofifa.org/teams/2/light/243.png">https://cdn.sofifa.org/teams/2/light/243.png</a>
8	Atlético Madrid	<a href="https://cdn.sofifa.org/teams/2/light/240.png">https://cdn.sofifa.org/teams/2/light/240.png</a>
9	FC Bayern München	<a href="https://cdn.sofifa.org/teams/2/light/21.png">https://cdn.sofifa.org/teams/2/light/21.png</a>
10	Tottenham Hotspur	<a href="https://cdn.sofifa.org/teams/2/light/18.png">https://cdn.sofifa.org/teams/2/light/18.png</a>

Poniżej przedstawiono przykładowe dokumenty powstałe wskutek importu danych:

```
_id: ObjectId("61cc5eb1189f94e456a9b8f7")
name: "FC Barcelona"
logo: "https://cdn.sofifa.org/teams/2/light/241.png"
```

---

```
_id: ObjectId("61cc5eb1189f94e456a9b8f8")
name: "Juventus"
logo: "https://cdn.sofifa.org/teams/2/light/45.png"
```

---

```
_id: ObjectId("61cc5eb1189f94e456a9b8f9")
name: "Paris Saint-Germain"
logo: "https://cdn.sofifa.org/teams/2/light/73.png"
```

---

W następnym kroku należy stworzyć kolekcję *players* i każdemu zawodnikowi przypisać referencję do klubu. W tym celu najwygodniej będzie skorzystać z biblioteki *mongoose* zawierającej sterownik do bazy MongoDB oraz pozwalającej na obsługę bazy danych za pomocą prostszej, bardziej przyjaznej programiście składni. Aby przydzielić zawodników do konkretnych klubów, konieczne będzie stworzenie jeszcze jednej kolekcji : *players\_nested*, która zawierać będzie wszystkie dane pozyskane bezpośrednio z pliku *data.csv*. Następnie, dla każdego dokumentu kolekcji *players\_nested* odnajdywany będzie identyfikator klubu z kolekcji *clubs*, poprzez wyszukiwanie po nazwie (zakładamy, że nazwy klubów są unikatowe). Poniżej przedstawiono fragment przykładowego dokumentu kolekcji *players\_nested*:

```
{ _id: ObjectId("61cb58dd6a45cb79ca235a86"),
  Name: 'L. Messi',
  Age: 31,
  Photo: 'https://cdn.sofifa.org/players/4/19/158023.png',
  Nationality: 'Argentina',
  Flag: 'https://cdn.sofifa.org/flags/52.png',
  Overall: '94',
  Potential: '94',
  Club: 'FC Barcelona',
  'Club Logo': 'https://cdn.sofifa.org/teams/2/light/241.png',
  Value: '€110.5M',
  Wage: '€565K',
  Special: '2202',
```

```
'Preferred Foot': 'Left',
'International Reputation': '5',
'Weak Foot': '4',
'Skill Moves': '4',
'Work Rate': 'Medium/ Medium',
'Body Type': 'Messi',
'Real Face': 'Yes',
Position: 'RF',
'Jersey Number': '10',
Joined: '01.07.2004',
'Contract Valid Until': '2021',
Height: '5\'7',
Weight: '159lbs',
positions:
{ LS: '88+2',
  ST: '88+2',
  RS: '88+2',
  LW: '92+2',
  LF: '93+2',
  CF: '93+2',
  RF: '93+2',
  RW: '92+2',
  LAM: '93+2',
  CAM: '93+2',
  RAM: '93+2',
  LM: '91+2',
  LCM: '84+2',
  CM: '84+2',
  RCM: '84+2',
  RM: '91+2',
  LWB: '64+2',
  LDM: '61+2',
  CDM: '61+2',
  RDM: '61+2',
  RWB: '64+2',
  LB: '59+2',
  LCB: '47+2',
  CB: '47+2',
```

```
RCB: '47+2',
RB: '59+2' },
skills:
{ Crossing: '84',
  Finishing: '95',
  HeadingAccuracy: '70',
  ShortPassing: '90',
  Volleys: '86',
  Dribbling: '97',
  Curve: '93',
  FKAccuracy: '94',
  LongPassing: '87',
  BallControl: '96',
  Acceleration: '91',
  SprintSpeed: '86',
  Agility: '91',
  Reactions: '95',
  Balance: '95',
  ShotPower: '85',
  Jumping: '68',
  Stamina: '72',
  Strength: '59',
  LongShots: '94',
  Aggression: '48',
  Interceptions: '22',
  Positioning: '94',
  Vision: '94',
  Penalties: '75',
  Composure: '96',
  Marking: '33',
  StandingTackle: '28',
  SlidingTackle: '26',
gk:
{ GKDivining: '6',
  GKHandling: '11',
  GKKicking: '15',
  GKPositioning: '14',
  GKReflexes: '8' } },
```

```
'Release Clause': '€226.5M' }
```

Narzędzie Compass umożliwia realizację wielu funkcjonalności bazy w prosty sposób za pomocą interfejsu graficznego. Warto jednak zapoznać się ze składnią języka zapytań bazy MongoDB, chociażby w celu obsługi bazy z poziomu aplikacji serwerowych. Dokument przedstawiony na powyższym zrzucie ekranu uzyskany został za pomocą narzędzia *mongosh*. Jest to środowisko pozwalające na komunikację z bazą danych za pomocą języka **JavaScript**. Interfejs graficzny Compass posiada to środowisko, w związku z czym można z niego wygodnie korzystać w ramach oprogramowania. Dokument przedstawiony powyżej uzyskano za pomocą prostego zapytania:

```
>_MONGOSH
```

```
> use fifa19
```

```
< 'switched to db fifa19'
```

```
> db.players_nested.findOne({Name: 'L. Messi'})
```

Początkowo nastąpiło połączenie z bazą **fifa19**. Następnie, z kolekcji *players\_nested* wybrano dokument, który zawiera w sobie pole *Name* o wartości 'L. Messi'.

W celu utworzenia kolekcji *players* połączono się z bazą danych za pomocą biblioteki *mongoose*:

```
const mongoose = require('mongoose')
const addNewPlayer = require('./services/player_service')
mongoose.connect('mongodb://localhost:27017/fifa19', () => {
  console.log('Successfully connected to mongoDB')
});
```

Następnie stworzone zostały odpowiednie schematy dokumentów wchodzących w skład kolekcji wraz z walidacją pól:



```
1  const mongoose = require('mongoose')
2  |
3  const ClubSchema = new mongoose.Schema({
4    _id: mongoose.Schema.Types.ObjectId,
5    name: {
6      type: String,
7      required : true,
8      unique: true
9    },
10   logo: {
11     type: string,
12     required : true
13   }
14
15
16 })
17
18 module.exports = mongoose.model('Club', ClubSchema, 'clubs')
```

Powyższy schemat odnosi się do wcześniej utworzonej kolekcji *clubs*. Pola *name*, *logo* są wymagane podczas dodawania nowego dokumentu, a wartość pola *name* dodatkowo musi być unikalna.

```

models > JS player.js > [?] PlayerSchema > 🔑 club > 🔑 type
1  const mongoose = require('mongoose')
2
3  const PlayerSchema = new mongoose.Schema({
4    _id: mongoose.Schema.Types.ObjectId,
5    Name: {
6      type: String,
7      required: true
8    },
9    Age: {
10     type: Number,
11     required: true,
12     min: 15,
13     max: 60
14   },
15   club: {
16     type: mongoose.Schema.Types.ObjectId,
17     ref: 'Club'
18   },
19 }, {
20   strict: false
21 })
22
23 module.exports = mongoose.model('Player', PlayerSchema)

```

Drugi schemat dotyczy kolekcji *players*. Ponieważ zawodnik posiada ponad 80 właściwości (pól), pominięto walidację wszystkich pól i umieszczono jedynie pola *Name*, *Age* w celach demonstracyjnych. Jak można zauważyć, schemat zawiera również pole *club*, które odpowiada za referencję do konkretnego dokumentu w kolekcji *clubs*. Dzięki wykorzystaniu pola „ref: 'Club'” możliwe będzie automatyczne pozyskiwanie danych o klubie podczas wyszukiwania zawodników. Ustawienie opcji {strict: false} dla schematu oznacza, że można dodawać dokumenty zawierające inne pola niż zdefiniowane w schemacie.

W przypadku kolekcji *players\_nested* pominięte zostanie stworzenie schematu – utworzony zostanie jedynie model odnoszący się do kolekcji w bazie danych, aby pozwolić bibliotece *mongoose* na wykonywanie operacji na dokumentach tej kolekcji.

```

1  const mongoose = require('mongoose')
2  module.exports = mongoose.model('NestedPlayer', {}, 'players_nested')

```

Po stworzeniu odpowiednich modeli pozostaje przeszukanie kolekcji *players\_nested* oraz utworzenie odpowiednich dokumentów w kolekcji *players*. W tym celu wykonano następujące operacje:

1. Stworzono serwis pobierający wszystkie dokumenty z kolekcji *players\_nested*:

```
services > JS nested_player_service.js > [?] <unknown>
1   const NestedPlayer = require('../models/nested_player')
2
3   const findAll = () => NestedPlayer.find().lean().exec()
4
5   module.exports = findAll
```

Funkcja `lean()` zwraca dane dokumentu w postaci czystego obiektu (POJO). W innym wypadku zostałby zwrócony obiekt odpowiadający danemu dokumentowi.

2. Stworzono serwis znajdujący dokument z kolekcji *clubs* po nazwie:

```
services > JS club_service.js > [?] <unknown>
1   const Club = require('../models/club')
2
3   const findByName = (name) => Club.findOne({name: name}).lean().exec()
4
5   module.exports = findByName
```

3. Stworzono serwis tworzący kolekcję *players*:

```

services > JS player_service.js > [⌘] createPlayersCollection
 1  const mongoose = require('mongoose')
 2  const findByName = require('./club_service')
 3  const findAll = require('./nested_player_service')
 4  const Player = require('../models/player')
 5
 6
 7
 8  const createPlayersCollection = async () => {
 9      const nestedPlayers = await findAll()
10      const players = []
11      for(const player of nestedPlayers){
12          const club = await findByName(player['Club'])
13          delete player['Club']
14          delete player['Club Logo']
15          delete player['_id']
16          const newPlayer = new Player({
17              _id: new mongoose.Types.ObjectId,
18              ...player,
19              club: club._id
20          })
21          players.push(newPlayer)
22      }
23      Player.insertMany(players)
24  }

```

W rezultacie kolekcja *players* została wypełniona dokumentami zawierającymi dane zawodników oraz referencje do odpowiednich klubów. Poniżej przedstawiono przykładowy rezultat:

```
Potential: "94"  
Value: "€110.5M"  
Wage: "€565K"  
Special: "2202"  
Preferred Foot: "Left"  
International Reput... : "5"  
Weak Foot: "4"  
Skill Moves: "4"  
Work Rate: "Medium/ Medium"  
Body Type: "Messi"  
Real Face: "Yes"  
Position: "RF"  
Jersey Number: "10"  
Joined: "01.07.2004"  
Contract Valid Until: "2021"  
Height: "5'7"  
Weight: "159lbs"  
> positions: Object  
> skills: Object  
Release Clause: "€226.5M"  
Name: "L. Messi"  
Age: 31  
club: ObjectId("61cc5eb1189f94e456a9b8f7")
```

Za pomocą biblioteki *mongoose* istnieje możliwość odczytu informacji o klubie podczas wyszukiwania zawodnika:

```
const getPlayers = () => Player.find().populate('club').exec()
```

Przykładowy rezultat:

```

Acceleration: '91',
SprintSpeed: '86',
Agility: '91',
Reactions: '95',
Balance: '95',
ShotPower: '85',
Jumping: '68',
Stamina: '72',
Strength: '59',
LongShots: '94',
Aggression: '48',
Interceptions: '22',
Positioning: '94',
Vision: '94',
Penalties: '75',
Composure: '96',
Marking: '33',
StandingTackle: '28',
SlidingTackle: '26',
gk: {
  GKDiving: '6',
  GKHandling: '11',
  GKKicking: '15',
  GKPositioning: '14',
  GKReflexes: '8'
},
'Release Clause': '€226.5M',
Name: 'L. Messi',
Age: 31,
club: {
  _id: new ObjectId("61cc5eb1189f94e456a9b8f7"),
  name: 'FC Barcelona',
  logo: 'https://cdn.sofifa.org/teams/2/light/241.png'
},

```

### 3.2. Model zagnieżdżony

W celu stworzenia kolekcji dokumentów zagnieżdżonych *clubs\_nested* stworzono kolejne dwa schematy:

```
const PlayerSchema = new mongoose.Schema({
  _id: mongoose.Schema.Types.ObjectId,
  Name: {
    type: String,
    required: true
  },
  Age: {
    type: Number,
    required: true,
    min: 15,
    max: 60
  }
}, {strict: false})
```

Schemat *PlayerSchema*, tak jak poprzednio, posiada luźną strukturę z dwoma polami podlegającymi walidacji.

```
17  const NestedClubSchema = new mongoose.Schema({
18    _id: mongoose.Schema.Types.ObjectId,
19    name: {
20      type: String,
21      required: true,
22      unique: true
23    },
24    logo: {
25      type: String,
26      required: true
27    },
28    players: [
29      PlayerSchema
30    ]
31  } )
32
33  module.exports = mongoose.model('NestedClub', NestedClubSchema)
```

Schemat *NestedClubSchema* składa się z trzech pól : *name*, *logo*, *players*, gdzie *players* to tablica przechowująca dokumenty zawierające dane zawodników. W celu wypełnienia kolekcji danymi, z utworzonych wcześniej kolekcji *clubs* oraz *players* zostaną wyselekcjonowane dane w odpowiedni sposób.

```

const createNestedClubsCollection = async() => {
  const clubs = await getAllClubs()
  const newClubs = []
  for(const club of clubs){
    const players = await getPlayersByClubId(club._id)
    const newClub = new NestedClub({
      _id: new mongoose.Types.ObjectId,
      name: club.name,
      logo: club.logo,
      players: players
    })
    newClubs.push(newClub)
  }
  NestedClub.insertMany(newClubs)
}

```

Poniżej przedstawiono przykładowy dokument z utworzonej kolekcji:

```

_id: ObjectId("61cc8ed90027a497b1240bf6")
name: "FC Barcelona"
logo: "https://cdn.sofifa.org/teams/2/light/241.png"
▼ players: Array
  > 0: Object
  > 1: Object
  > 2: Object
  > 3: Object
  > 4: Object
  > 5: Object
  > 6: Object
  > 7: Object
  > 8: Object
  > 9: Object
  > 10: Object
  > 11: Object
  > 12: Object
  > 13: Object
  > 14: Object
  > 15: Object
  > 16: Object
  > 17: Object
  > 18: Object
  --

```



```
▼ players: Array
  > 0: Object
  ▼ 1: Object
    Photo: "https://cdn.sofifa.org/players/4/19/176580.png"
    Nationality: "Uruguay"
    Flag: "https://cdn.sofifa.org/flags/60.png"
    Overall: "91"
    Potential: "91"
    Value: "€80M"
    Wage: "€455K"
    Special: "2346"
    Preferred Foot: "Right"
    International Reput... : "5"
    Weak Foot: "4"
    Skill Moves: "3"
    Work Rate: "High/ Medium"
    Body Type: "Normal"
    Real Face: "Yes"
    Position: "RS"
    Jersey Number: "9"
    Joined: "11.07.2014"
    Contract Valid Until: "2021"
    Height: "6'0"
    Weight: "190lbs"
```

## 4. Podstawowe funkcje MongoDB Compass

### 4.1.Schema

Interfejs graficzny wykorzystywany w ramach projektu posiada kilka ciekawych, bardzo intuicyjnych funkcji. Pierwszą z nich jest możliwość analizy schematu:



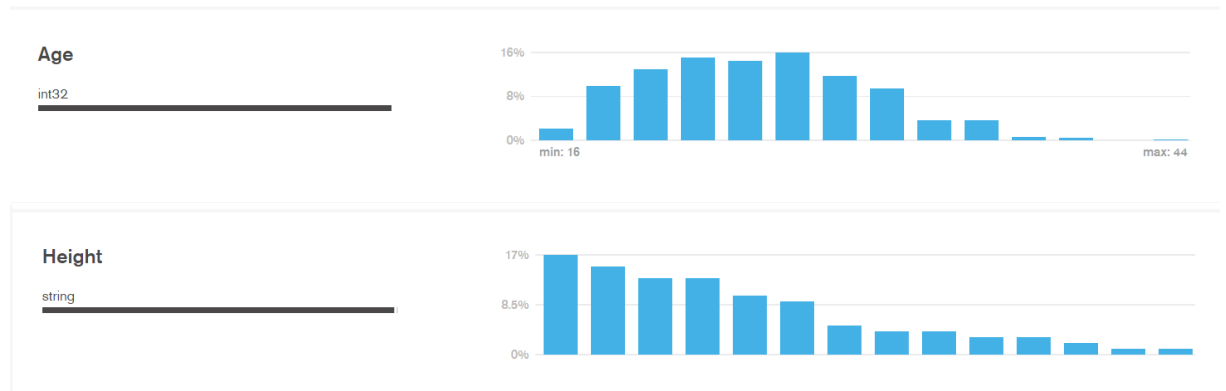
## Explore your schema

Quickly visualize your schema to understand the frequency, types and ranges of fields in your data set.

Analyze Schema

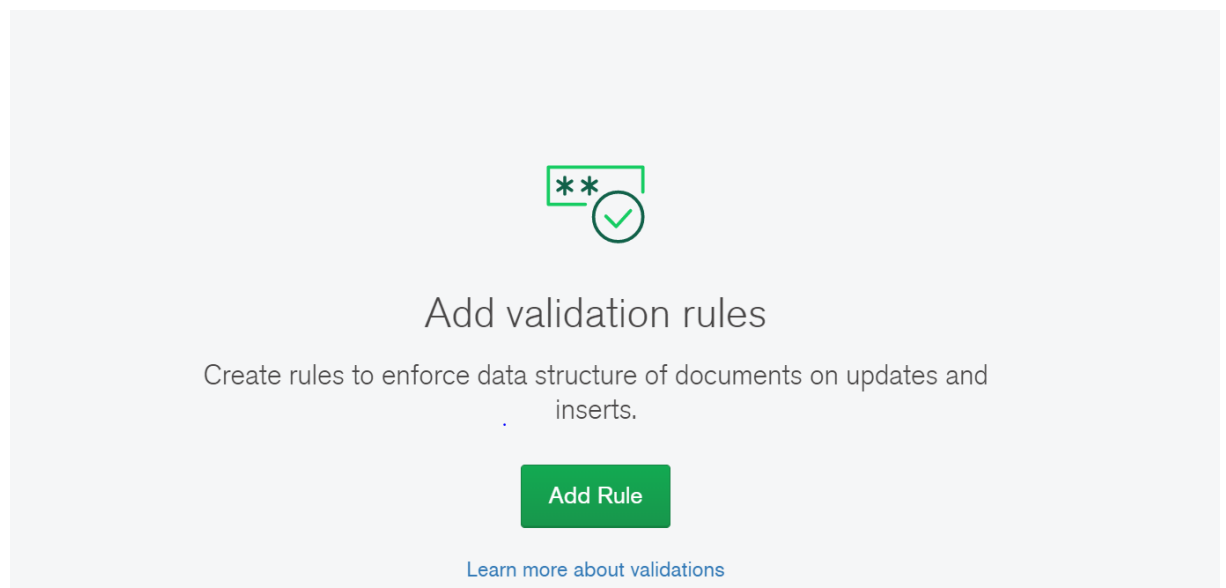
[Learn more about schema analysis in Compass](#)

Po kliknięciu przycisku *Analyze Schema* użytkownik uzyskuje informacje na temat statystycznego rozkładu pól w ramach kolekcji:



## 4.2. Validation

Za pomocą MongoDB Compass istnieje możliwość walidacji pól wprowadzanych dokumentów.



Validation Action ⓘ ERROR ▾

Validation Level ⓘ STRICT ▾

```
1 {
2   $jsonSchema: {
3     required: ['Name', 'Age'],
4     properties: {
5       Name: {
6         bsonType: "string",
7         description: "must be a string and is required"
8       }
9     }
10  }
```

Validation modified

CANCEL UPDATE

W przedstawionym powyżej oknie należało wprowadzić zasady walidacyjne, zgodnie z instrukcjami zawartymi w dokumentacji. Narzędzie Compass w wielu miejscach zawiera odnośniki do odpowiednich rozdziałów dokumentacji. Na powyższym przykładzie określono pola *Name* oraz *Age* jako wymagane, a typ pola *Name* jako *string*. W przypadku próby dodania dokumentu do kolekcji niespełniającego wymagań walidacyjnych zwrócony zostanie błąd. Dodatkowo, interfejs graficzny przedstawia przykładowy dokument spośród już będących w kolekcji spełniający wymagania oraz (jeśli taki wystąpi) niespełniający wymogów:

✓ Sample Document That Passed Validation

✗ Sample Document That Failed Validation

```
_id: ObjectId("61cb58dd6a45cb79ca235a86")
Name: "L. Messi"
Age: 31
Photo: "https://cdn.sofifa.org/players/4/19/158023.png"
Nationality: "Argentina"
Flag: "https://cdn.sofifa.org/flags/52.png"
Overall: "94"
Potential: "94"
```

No Preview Documents

Spróbujmy dodać nowy dokument za pomocą *mongosh*:

```
> db.players_nested.insertOne({Name: 15})
```

✖ ▼ **MongoServerError:** Document failed validation

Additional information:

```
{ failingDocumentId: {},
  details:
    { operatorName: '$jsonSchema',
      schemaRulesNotSatisfied:
        [ { operatorName: 'properties',
            propertiesNotSatisfied:
              [ { propertyName: 'Name',
                  details:
                    [ { operatorName: 'bsonType',
                        specifiedAs: { bsonType: 'string' },
                        reason: 'type did not match',
                        consideredValue: 15,
                        consideredType: 'int' } ] } ] },
            { operatorName: 'required',
              specifiedAs: { required: [ 'Name', 'Age' ] },
              missingProperties: [ 'Age' ] } ] } ] }
```

Próba utworzenia dokumentu okazała się niepomyślna – zabrakło pola *Age* oraz podano nieprawidłowy typ pola *Name*.

### 4.3. Documents

Zakładka *Documents* pozwala na obserwację, edycję, filtrowanie danych w ramach kolekcji oraz dodawanie i usuwanie dokumentów.



fifa19.clubs\_nested

DOCUMENTS

652

TOTAL SIZE

28.4MB

AVG. SIZE

43.5KB

INDEXES

2

TOTAL SIZE

57.3KB

AVG. SIZE

28.7KB

Documents

Aggregations

Schema

Explain Plan

Indexes

Validation

FILTER

{'players.Nationality': 'Poland'}

OPTIONS

EXPLAIN

RESET

↺

⋮

VIEW DETAILS AS

VISUAL TREE

RAW JSON

Query Performance Summary

Documents Returned: 81

Index Keys Examined: 0

Documents Examined: 652

Actual Query Execution Time (ms): 8

Sorted in Memory: no

No index available for this query.

COLLSCAN

nReturned: 81

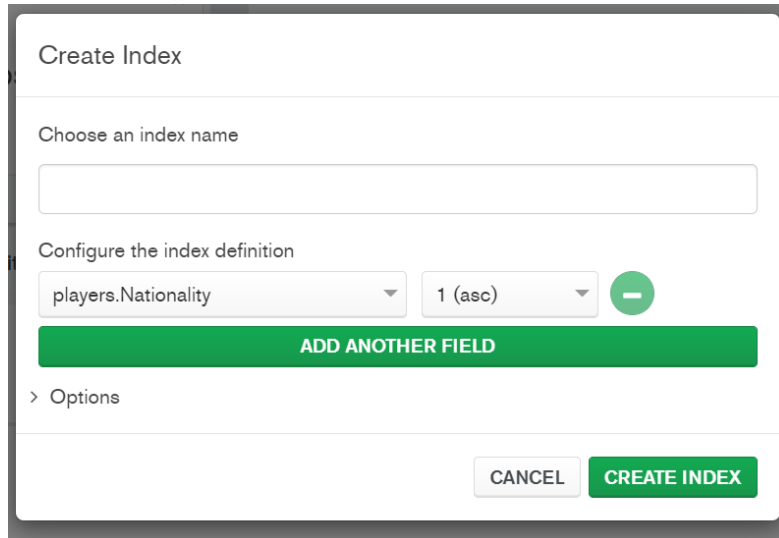
Execution Time: 0 ms

Documents Examined: 652

DETAILS

Jak można zauważyć, w tym przypadku wykonano COLLSCAN, co oznacza przeskanowanie całej kolekcji. Spośród 652 dokumentów kryteria spełniło 81 klubów. Nie wykorzystano żadnych indeksów, a czas realizacji zapytania wyniósł 8ms.

Nałożmy indeks na pole *players.Nationality* i porównajmy otrzymane rezultaty:



#### Query Performance Summary

Documents Returned: **81**


Index Keys Examined: **81**

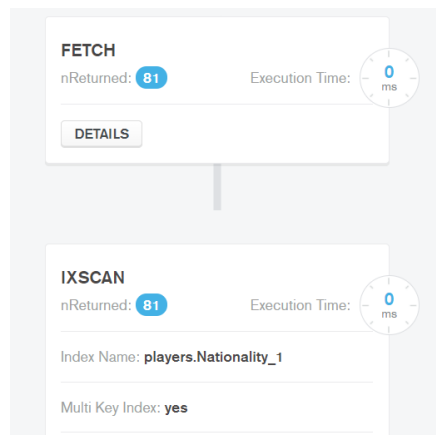
Documents Examined: **81**

Actual Query Execution Time (ms): **0**

Sorted in Memory: **no**

Query used the following index:

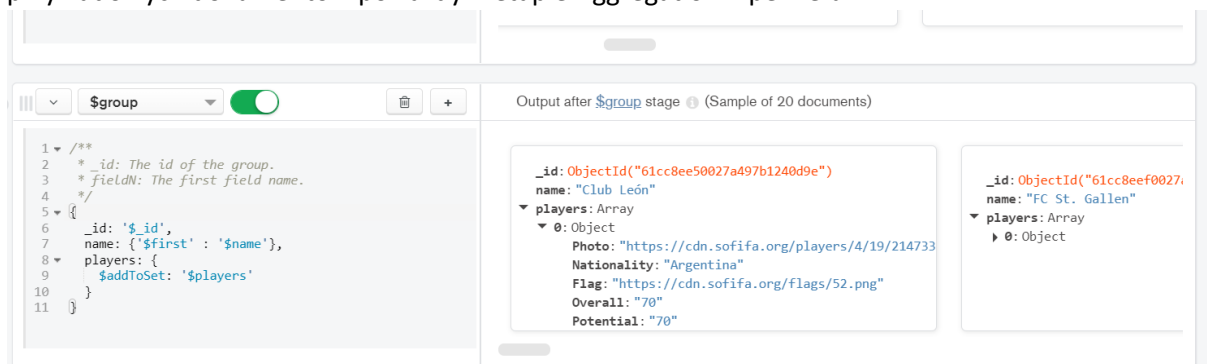
players.Nationality 



Tym razem uzyskano skan indeksu – przeszukano dokładnie tyle dokumentów, ile zwrócono. Czas wykonania zapytania okazał się bardzo krótki – wyniósł mniej niż 1 ms, co jest doskonałym wynikiem.

## 4.6. Aggregations

W tej zakładce istnieje możliwość tworzenia agregacji oraz ich zapisu. Uważam, że jest to dosyć przydatne narzędzie – główną jego zaletą jest możliwość podglądu „na żywo” przykładowych dokumentów po każdym etapie Aggregation Pipeline’u.



Powyższy zrzut ekranu prezentuje w lewym oknie treść fragmentu Pipeline’u (grupowanie), a po prawej stronie przykładowe rezultaty etapu agregacji. Więcej szczegółów dotyczących agregacji zostanie przedstawionych w późniejszych punktach.

## 5. Zagnieżdżanie czy referencje?

MongoDB nie jest bazą relacyjną – w większości przypadków zalecane jest korzystanie z dokumentów zagnieżdżonych. Podejście to sprawia, że operacje na dokumencie zagnieżdżonym są **atomowe** i wydajne. Ponieważ trudno znaleźć jednak rozwiązanie sprawdzające się w każdym przypadku, istnieje możliwość imitowania relacji poprzez referencje między dokumentami.

Referencja to nic innego jak utworzenie pola przechowującego wartość `_id` innego dokumentu w tworzonego dokumencie. Zewnętrzne sterowniki MongoDB pozwalają również na wykorzystanie *DBRefs*, czyli automatycznych referencji do dokumentów z innych kolekcji bądź baz danych. Pozyskanie wszystkich danych z dokumentu zawierającego referencje wymaga dodatkowych zapytań.

Wybór metody przechowywania danych jest kwestią zależną od ich rodzaju i operacji, które będą na nich wykonywane. Maksymalny rozmiar pojedynczego dokumentu w MongoDB to 16MB – jeśli rozmiar danych przekracza limit koniecznym będzie skorzystanie z referencji. W przypadku relacji *One to One* wykorzystanie referencji będzie niewygodne i prawdopodobnie znacznie mniej wydajne niż zagnieżdżanie.

## 6. Logika biznesowa – zapytania, transakcje, agregacje

W punkcie trzecim przedstawiono przykłady wykorzystania biblioteki *mongoose* do tworzenia nowych kolekcji w bazie danych na podstawie istniejących dokumentów. Przedstawione zostało również w jaki sposób można tworzyć i walidować schematy. W tym punkcie zaprezentowane zostaną przykłady realizacji logiki biznesowej.

### 6.1. Zapytania



### a) Model referencyjny

Jak już wcześniej wspomniano, wykorzystanie referencji wiąże się z koniecznością kierowania dodatkowych zapytań w celu pozyskania wszystkich danych. MongoDB pozwala na skorzystanie z opcji *DBRef*, lecz nie jest ona zalecana ze względu na potencjalne problemy wydajnościowe.

Spróbujmy znaleźć wszystkich zawodników należących do klubu FC Barcelona. Aby to osiągnąć, należy odnaleźć w kolekcji *clubs* dokument dotyczący nazwy klubu, a następnie przefiltrować kolekcję *players* po identyfikatorze klubu.

```
const getPlayersForClub = async (name) => {  
  const club = await findByName(name)  
  const players = await Player.find({club: club._id}).exec()  
}
```

W przypadku wykorzystania *DBRef* wykonać można następujące operacje:

```
const getPlayersForClub = async (name) => {  
  let players = await Player.find().lean().populate('club').exec()  
  players = players.filter(player => player.club.name === 'FC Barcelona')  
}
```

Nie jest to najlepszy pomysł – w tym przypadku pobrane zostaną wszystkie dokumenty. Dodatkowo dla każdego dokumentu automatycznie zostaną pozyskane informacje o klubie. Następująca później operacja filtrowania jest już stricte działaniem JavaScriptu.

### b) Model zagnieżdżony

Aby uzyskać dane piłkarzy z klubu FC Barcelona w dokumencie zagnieżdżonym wystarczy wykonać proste zapytanie:

```
db.clubs.nested.find({name: 'FC Barcelona'}, {players: 1})  
(mongosh)
```

```
NestedClub.findOne({name: 'FC Barcelona'}, 'players').exec()
```

(mongoose)

Obiekt `{players: 1}` jest projekcją (*projection*). Projekcja pozwala na zdecydowanie, których pól autor zapytania oczekuje w odpowiedzi, bądź które z nich chciałby wykluczyć. W tym przypadku, `{players: 1}` oznacza, że w odpowiedzi oczekiwana jest tylko zawartość pola *players*. W przypadku `{players: 0}` zwrócone zostaną wszystkie pola z wyjątkiem pola *players*.

## 6.2. Transakcje

Problem transakcyjności nie występuje w dokumentach zagnieżdżonych – wszelkie zmiany zawsze odbywają się w ramach pojedynczego dokumentu. W przypadku operacji na kilku dokumentach wymagających atomowości, istnieje możliwość tworzenia transakcji. Przykładową operacją wymagającą transakcji może być obustronny transfer dwóch zawodników z jednego klubu do drugiego. Znajdźmy zawodników: Lionel Messi oraz Robert Lewandowski wraz z informacją o klubach, do których należą:

```
const getPlayersByNames = (names) => Player.find({Name : {$in : names}}).lean().populate('club').exec()

(async () => {
  const players = await getPlayersByNames(['R. Lewandowski', 'L. Messi'])
  console.log(players)
})()
```

Następnie dokonamy wymiany klubów, do których należą zawodnicy – Lionel Messi zostanie przetransferowany do drużyny Bayern Munchen, a Robert Lewandowski do FC Barcelona.

```
const transferPlayerToClub = (playerId, clubId) => Player.findByIdAndUpdate(playerId, {club: clubId}, {new: true})
```

Powyższa funkcja odnajduje dokument o wskazanym `_id` w kolekcji `players`, aktualizuje identyfikator klubu, a następnie zwraca zmodyfikowany dokument.

```
const players = await getPlayersByNames(['R. Lewandowski', 'L. Messi'])
const firstPlayer = players[0]
const secondPlayer = players[1]
const transferredFirstPlayer = await transferPlayerToClub(firstPlayer._id, secondPlayer.club._id)
const transferredSecondPlayer = await transferPlayerToClub(secondPlayer._id, firstPlayer.club._id)
```

```

L. Messi
{
  _id: new ObjectId("61cc8d7f189f94e456aa02e4"),
  name: 'FC Bayern München',
  logo: 'https://cdn.sofifa.org/teams/2/light/21.png'
}
R. Lewandowski
{
  _id: new ObjectId("61cc8d7f189f94e456aa02dc"),
  name: 'FC Barcelona',
  logo: 'https://cdn.sofifa.org/teams/2/light/241.png'
}

```

Jak można zauważyć, zamiana klubów przebiegła pomyślnie. Co gdyby jednak pomiędzy pierwszym, a drugim transferem wystąpił niespodziewany błąd?

```

const players = await getPlayersByNames(['R. Lewandowski', 'L. Messi'])
const firstPlayer = players[0]
const secondPlayer = players[1]
const transferredFirstPlayer = await transferPlayerToClub(firstPlayer._id, secondPlayer.club._id)
if(1) throw new Error('Unexpected error!')
const transferredSecondPlayer = await transferPlayerToClub(secondPlayer._id, firstPlayer.club._id)

```

W tym przypadku jeden z zawodników zmieni klub, podczas gdy drugi pozostanie w poprzednim:

```

L. Messi
{
  _id: new ObjectId("61cc8d7f189f94e456aa02e4"),
  name: 'FC Bayern München',
  logo: 'https://cdn.sofifa.org/teams/2/light/21.png'
}
R. Lewandowski
{
  _id: new ObjectId("61cc8d7f189f94e456aa02e4"),
  name: 'FC Bayern München',
  logo: 'https://cdn.sofifa.org/teams/2/light/21.png'
}

```

Aby zapobiec takiej sytuacji wykorzystać można transakcje. Każda transakcja powstaje w ramach sesji. Operacja wykonywana w bazie danych musi zawierać informację o sesji, w ramach której wykonywana jest transakcja. Dzięki temu możliwe jest wycofanie

transakcji w przypadku wystąpienia błędu. Poniżej przedstawiono zmodyfikowany kod obsługujący transakcję:

```
const transferPlayerToClub = (playerId, clubId, session) => Player.findByIdAndUpdate({
  playerId,
  { club: clubId },
  { new: true, session: session })
  .populate('club').exec()
```

```
(async () => {
  const session = await mongoose.startSession()
  await session.withTransaction(async() => {
    const players = await getPlayersByNames(['R. Lewandowski', 'L. Messi'])
    const firstPlayer = players[0]
    const secondPlayer = players[1]
    const transferredFirstPlayer = await transferPlayerToClub(firstPlayer._id, secondPlayer.club._id, session)
    if(1) throw new Error('Unexpected error!')
    const transferredSecondPlayer = await transferPlayerToClub(secondPlayer._id, firstPlayer.club._id, session)
  })
  await session.endSession()
})()
```

Wystąpienie błędu powoduje wycofanie transakcji, a tym samym wycofanie aktualizacji dokumentu.

### 6.3. Agregacje

Agregacje umożliwiają wykonywanie bardziej złożonych operacji na dokumentach, niż proste zapytania. Pojęcie *Aggregation Pipeline* można rozumieć jako strumień – agregacja to operacja wieloetapowa, gdzie każdy kolejny etap modyfikuje zmiany z poprzedniego etapu. Mogą okazać się przydatne w dokumentach zagnieżdżonych. Przykładowo, spróbujmy z kolekcji *clubs\_nested* pozyskać kluby z zawodnikami konkretnej narodowości.

```

const getPlayersByNationality = () => NestedClub.aggregate(
  [
    {
      $unwind: {
        path: '$players'
      }
    },
    {
      $match: {
        'players.Nationality': 'Poland'
      }
    },
    {
      $group: {
        _id: '$_id',
        name: { '$first': '$name' },
        logo: { '$first': '$logo' },
        players: {
          '$addToSet': '$players'
        }
      }
    }
  ]
)

```

Przedstawiona wyżej agregacja rozbija początkowo zagnieżdżone tablice subdokumentów – oznacza to, że z dokumentu zawierającego przykładowo 10 zawodników powstaje 10 dokumentów. Jest to efekt komendy \$unwind. Następnie spośród wszystkich dokumentów wybierane są te, w których zawodnik jest narodowości polskiej (\$match). Ostatnim etapem jest grupowanie – określenie, w jaki sposób wyglądać będzie wyjściowy rezultat agregacji. W powyższym przykładzie każdy nowopowstały dokument będzie zawierał pola *\_id*, *name*, *logo* z klubu oraz przefiltrowaną tablicę zawodników.

```
{
  _id: new ObjectId("61cc8ef20027a497b1240f8a"),
  name: 'Lechia Gdańsk',
  logo: 'https://cdn.sofifa.org/teams/2/light/111091.png',
  players: [
    [Object], [Object], [Object],
    [Object], [Object], [Object],
    [Object], [Object], [Object],
    [Object], [Object], [Object],
    [Object], [Object], [Object],
    [Object]
  ]
},
{
  _id: new ObjectId("61cc8edc0027a497b1240c3e"),
  name: 'AS Monaco',
  logo: 'https://cdn.sofifa.org/teams/2/light/69.png',
  players: [ [Object] ]
},
{
  _id: new ObjectId("61cc8ee20027a497b1240d20"),
  name: 'FC Girondins de Bordeaux',
  logo: 'https://cdn.sofifa.org/teams/2/light/59.png',
  players: [ [Object] ]
},
}
```

## 7. Podsumowanie

1. Baza dokumentowa MongoDB była dobrym wyborem w przypadku wybranego przeze mnie zbioru danych. Wynika to z braku zawiłych powiązań relacyjnych między danymi.
2. Struktura dokumentowa pozwala na szybkie wyszukiwanie danych, również w dokumentach zagnieżdżonych.
3. W moim subiektywnym odczuciu dokumentacja bazy MongoDB jest bardziej przyjazna i zrozumiała niż dokumentacja systemu MSSQL. Uważam, że brakuje w niej jednak bardziej skomplikowanych przykładów dotyczących dokumentów zagnieżdżonych – agregacje w przypadku dokumentów zagnieżdżonych zdają się być największym wyzwaniem dla niedoświadczonych użytkowników.
4. Interfejs graficzny MongoDB Compass pozwala na wykonywanie prostych działań w prosty sposób. Jest to zarówno zaleta, jak i wada – poziom zaawansowania narzędzia jest znacznie niższy niż przykładowo w przypadku oprogramowania Microsoft SQL Server Management Studio, jednak zdecydowanie zwycięża pod względem komfortu użytkowania.
5. Uważam, że dużą zaletą bazy MongoDB jest jej dynamiczny rozwój i regularne wprowadzanie nowych rozwiązań ułatwiających korzystanie z bazy.