

# **Tecnologie Cloud e Mobile**

## **Lez. 10** **MongoDB e Node.js**

**Giuseppe Psaila**

*Università di Bergamo*

*giuseppe.psaila@unibg.it*

# MongoDB

# I NoSQL Databases

- Il modello relazionale dei dati funziona benissimo
- I DBMS relazionali basati su SQL funzionano benissimo
- Eppure, l'avvento dei Big Data e dei formati semi-strutturati, come XML e JSON, li hanno resi inadatti al nuovo mondo dei Big Data
- La causa è la loro rigidità

# I NoSQL Databases

- NoSQL:
  - Not only SQL (lettura ufficiale)  
indica che non esiste solo SQL, c'è anche altro
  - No SQL (lettura ufficioso)
- I cosiddetti JSON Databases, o JSON Stores, sono una particolare categoria di NoSQL DB
- MongoDB è il più famoso e usato

# MongoDB

- Due versioni:
- DBMS da scaricare e installare  
`https://www.mongodb.com/download-center/community`
- Servizio Cloud, denominato «Atlas»  
`https://www.mongodb.com/cloud/atlas2`

# Modello dei Dati

- Database: un insieme di «collezioni»
- Collezione:  
nome univoco nel database
- Istanza della collezione:  
un insieme di documenti JSON  
senza alcun vincolo di struttura
- Una collezione è un insieme di documenti JSON eterogenei

# Oggetti e ID

- Una volta memorizzato, ogni oggetto JSON ha un identificatore univoco
- Campo «\_id»
- Questo valore può essere già presente quando l'oggetto viene caricato
- Se non è presente, viene aggiunto e valorizzato automaticamente dal sistema

# Strumenti

- MongoDB è un DBMS
- Raggiungibile su una porta specifica di un server specifico
- Nella versione linux, esiste l'interfaccia per il prompt denominata «mongo»
- Nella versione Windows, non esiste
- Esistono dei tool grafici per gestire i DB su MongoDB

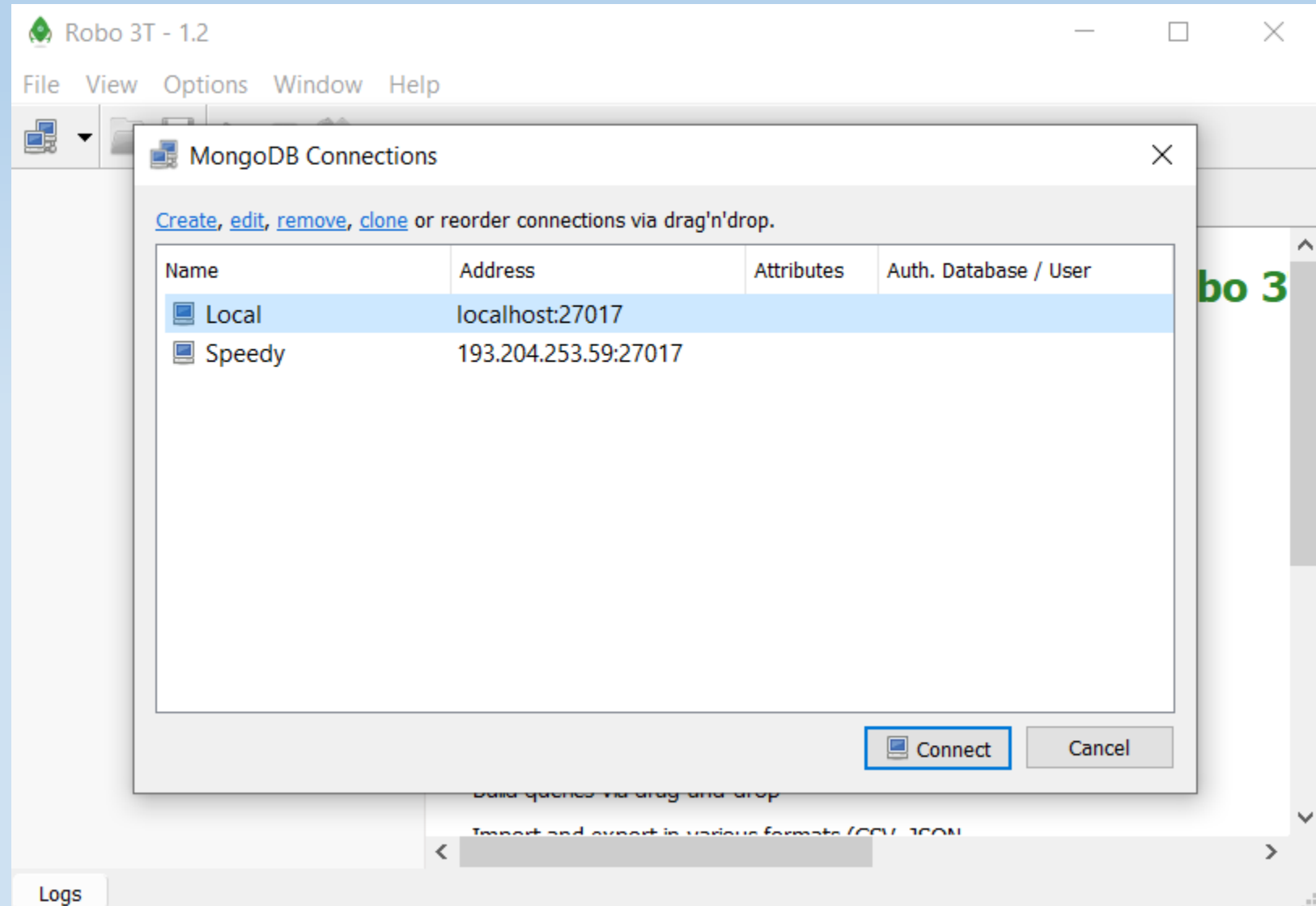


# Robo 3T

- Uno strumento per gestire i database di MongoDB è «Robo 3T» (in passato, si chiamava «RoboMongo»)
- Dowload  
**`https://robomongo.org/`**
- Purtroppo bisogna scaricare insieme anche «Studio 3T», più ricco ma anche più complesso

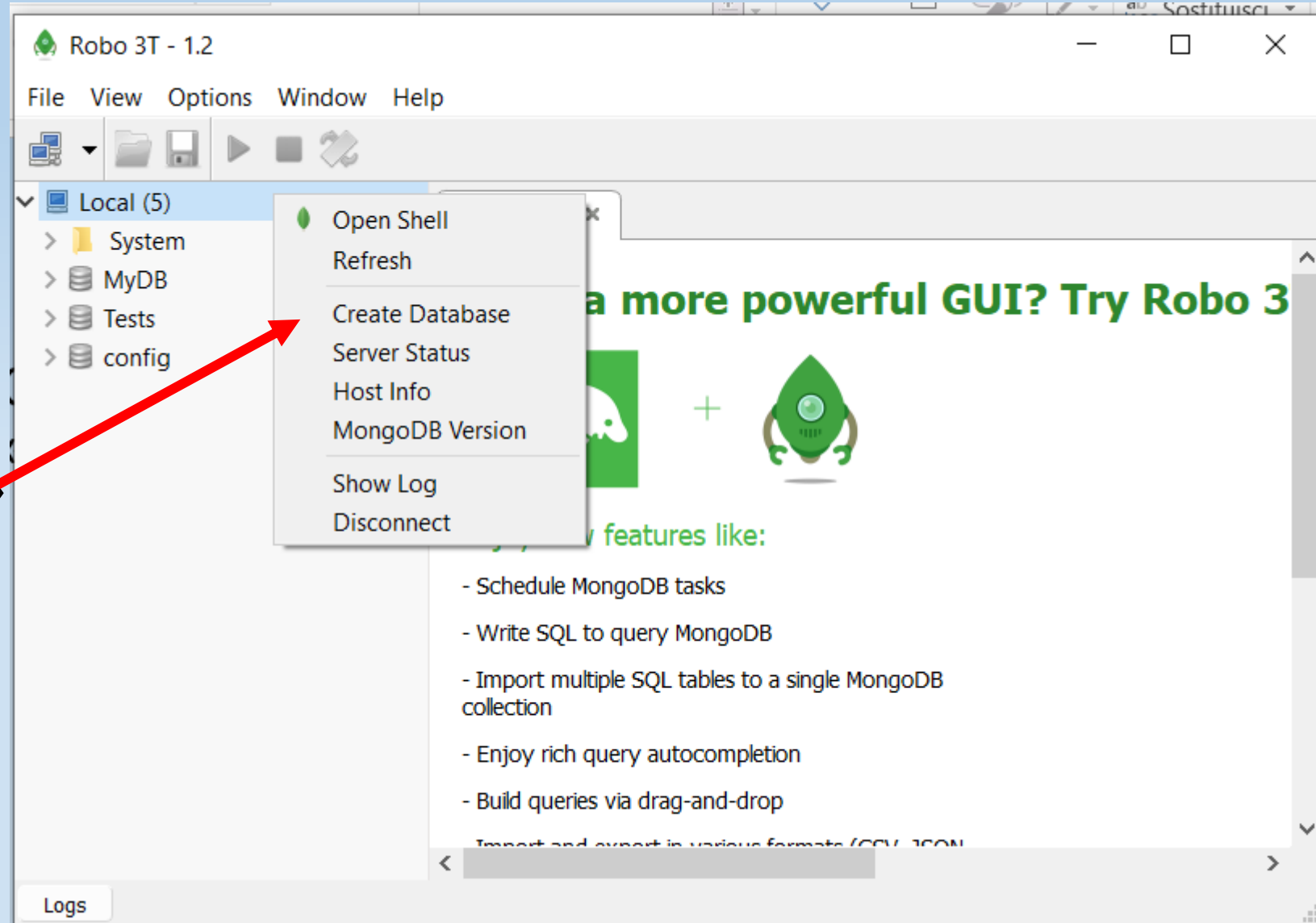
# Robo 3T

- All'avvio: scelta del server al quale connettersi
- Il server «local» dovrebbe essere pre-configurato; se non lo è, il suo indirizzo è «localhost»



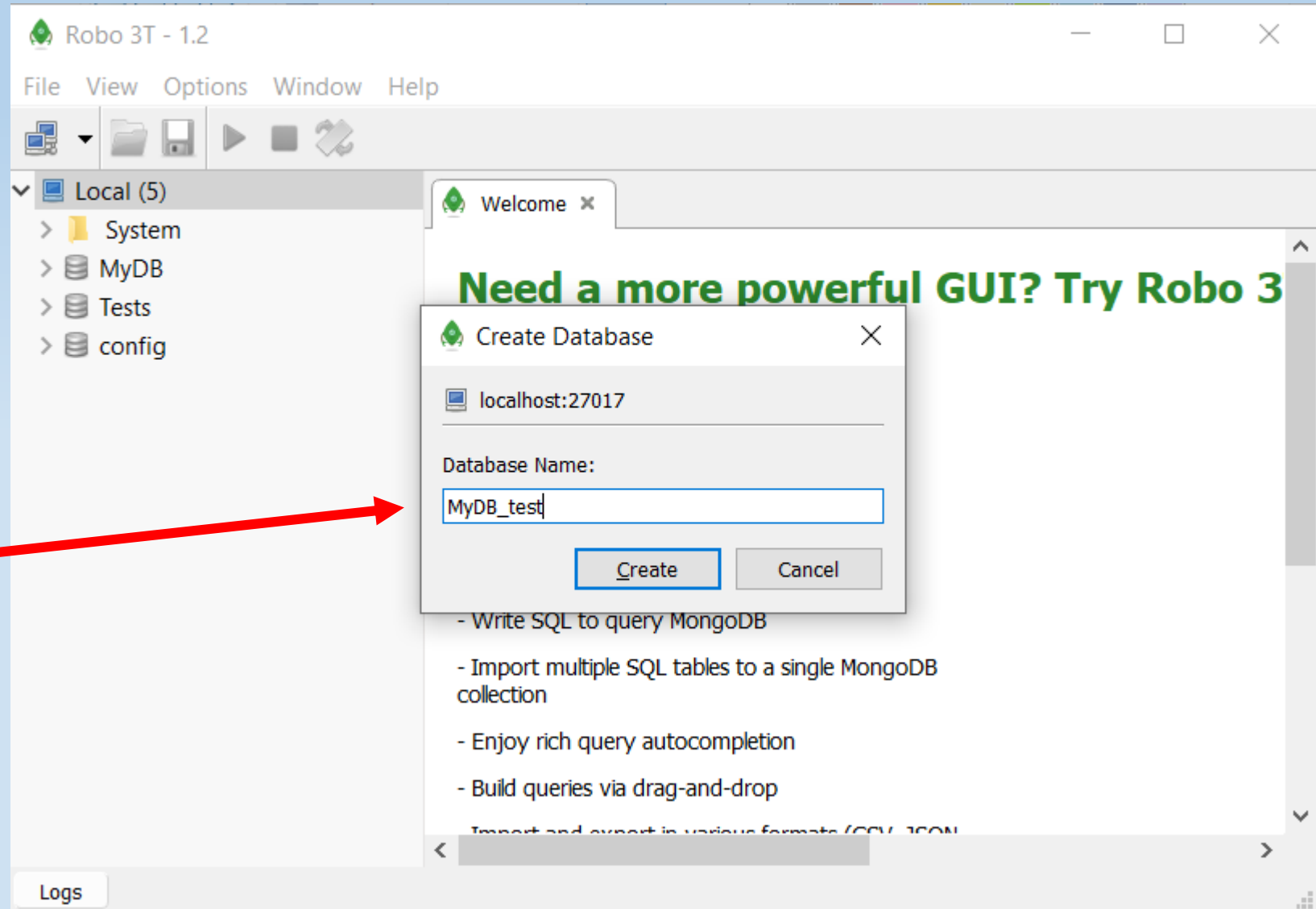
# Robo 3T

- Tasto di destra sul server, scegliamo «Create Database»



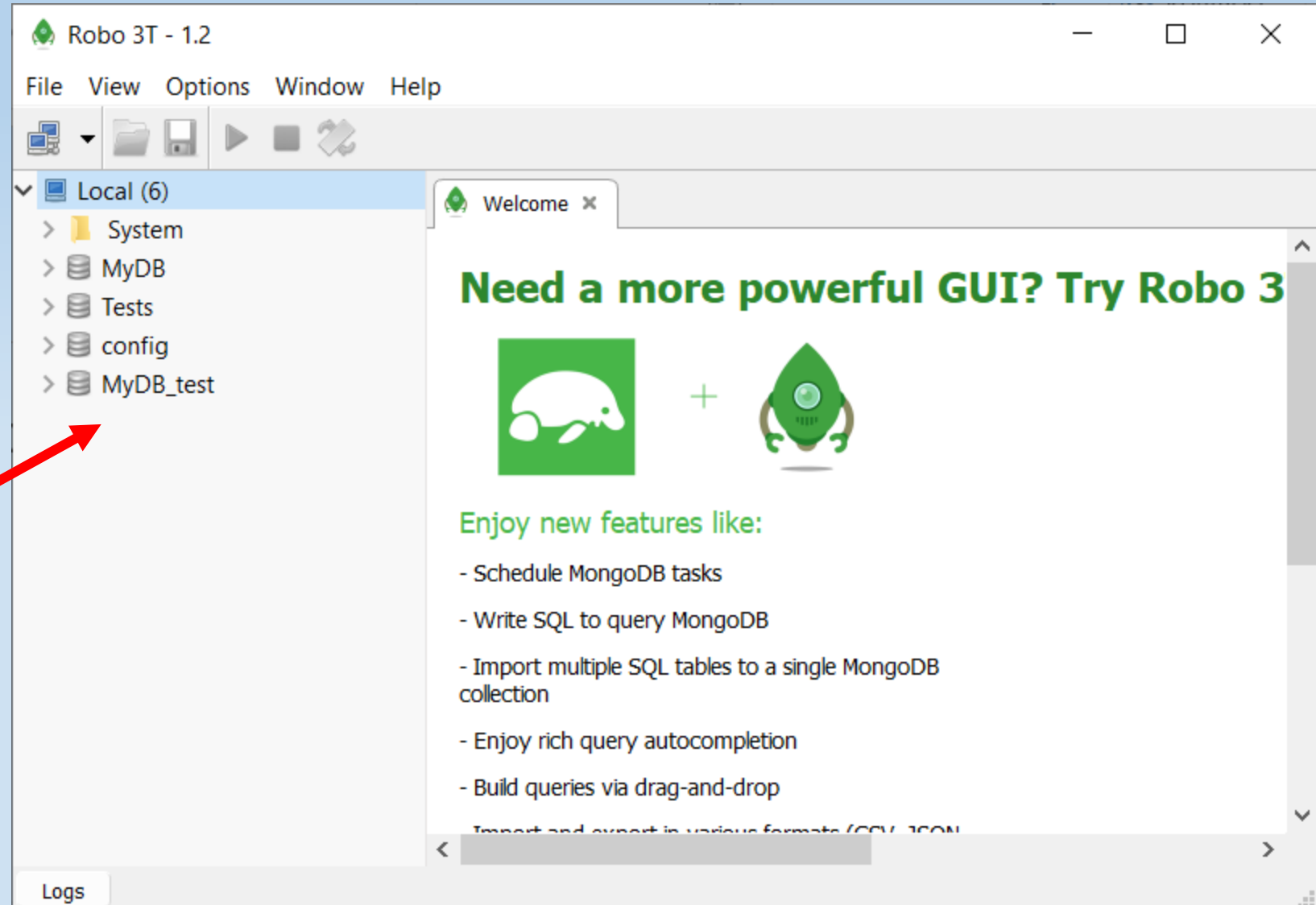
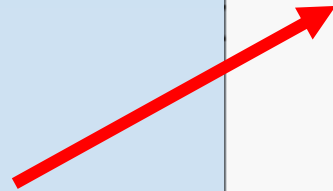
# Robo 3T

- Creiamo il db «MyDB\_test» (che useremo nel resto della lezione)



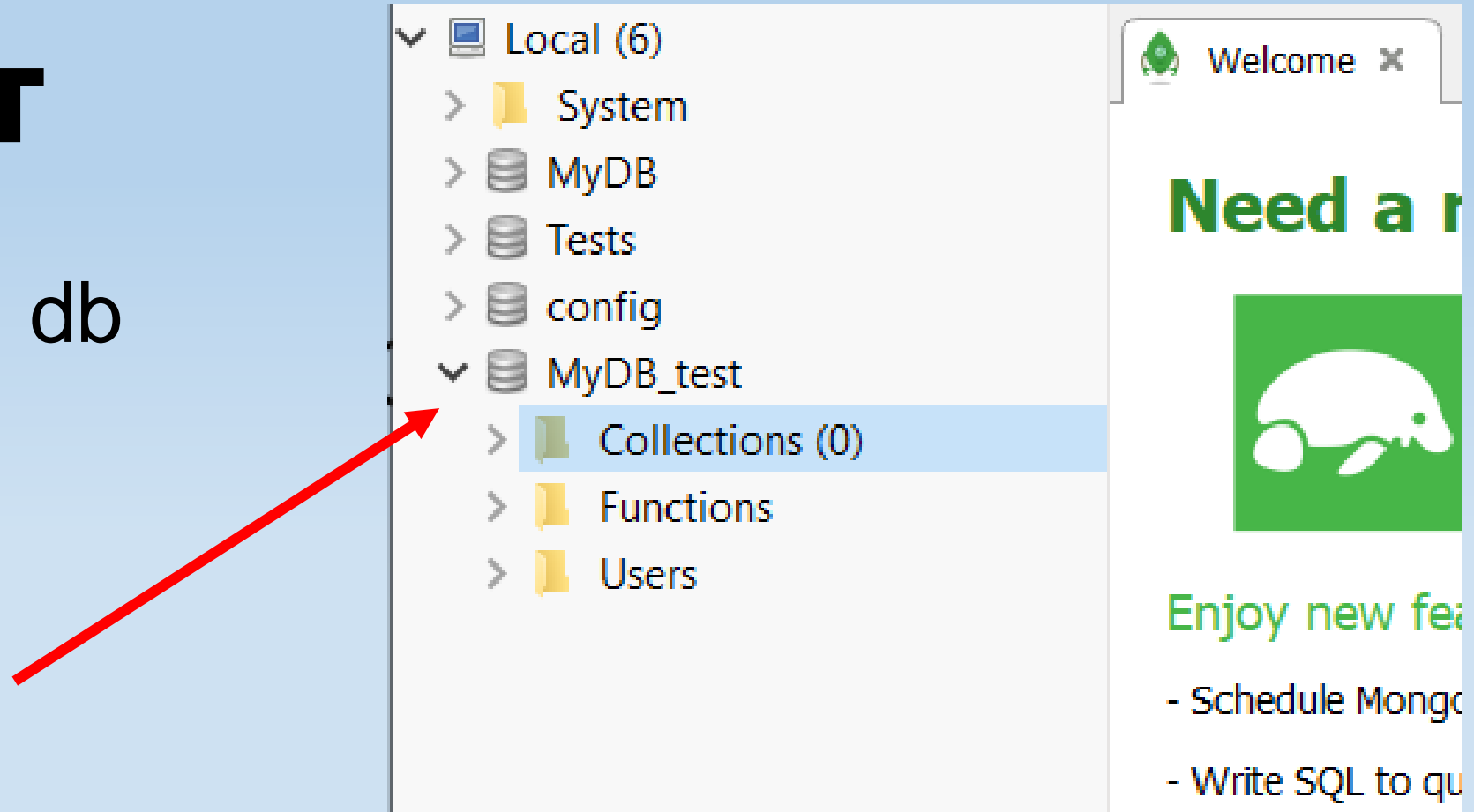
# Robo 3T

- Il db è stato creato



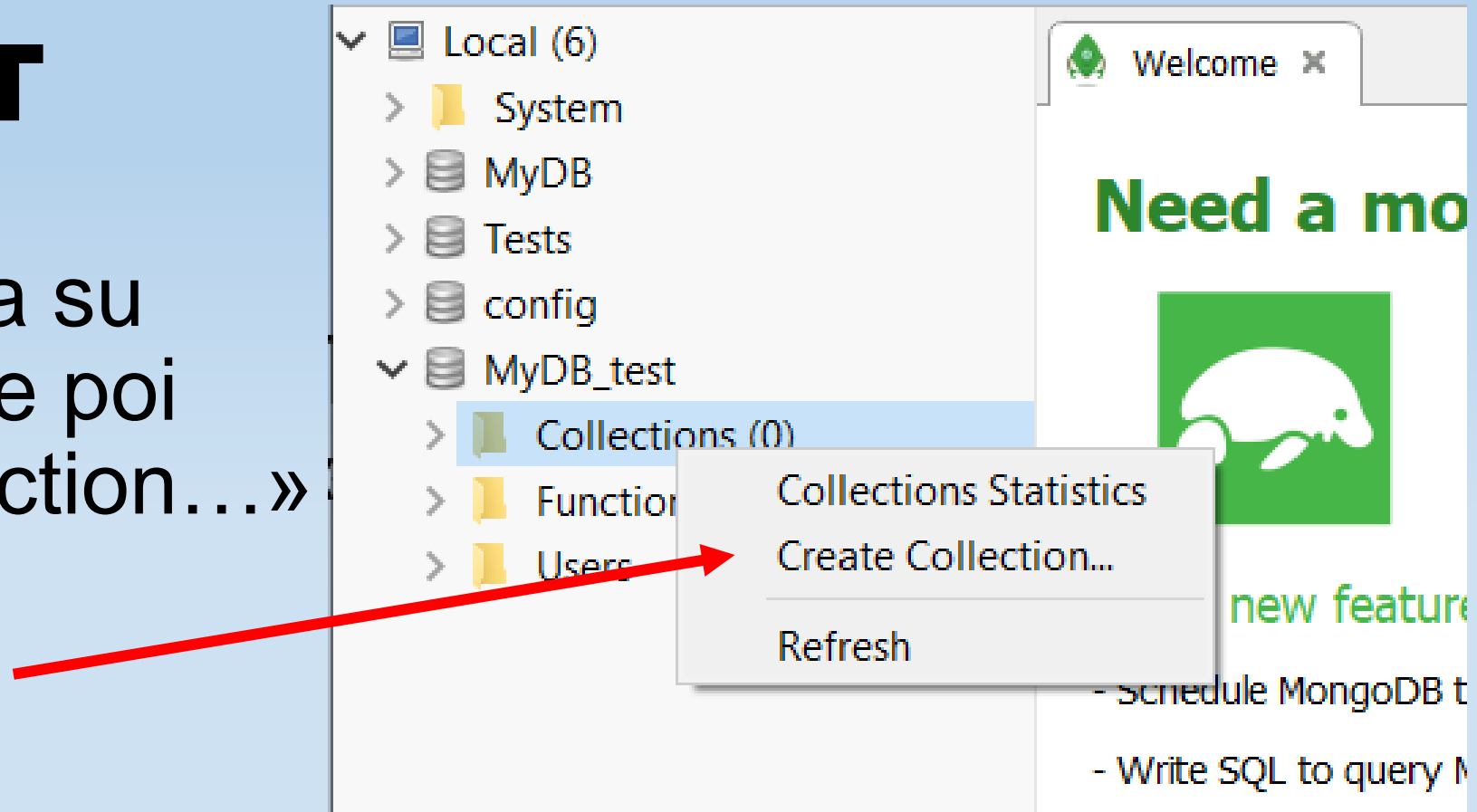
# Robo 3T

- Contenuto del db



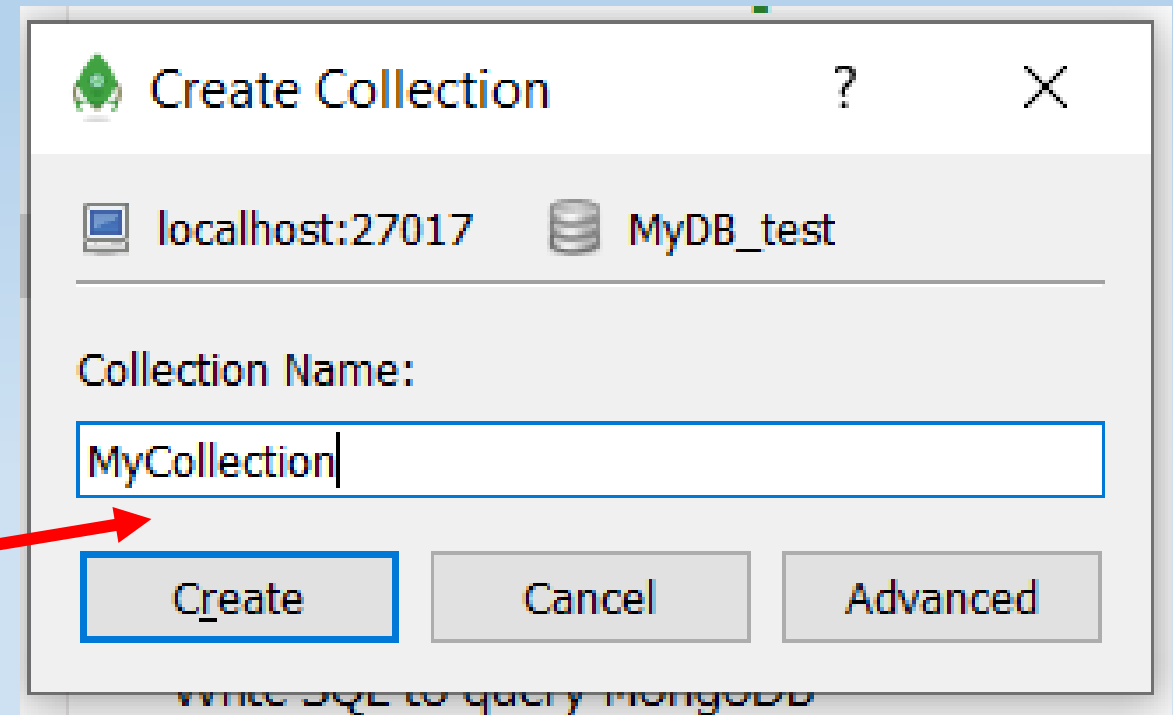
# Robo 3T

- Tasto di destra su «collections» e poi «Create Collection...»



# Robo 3T

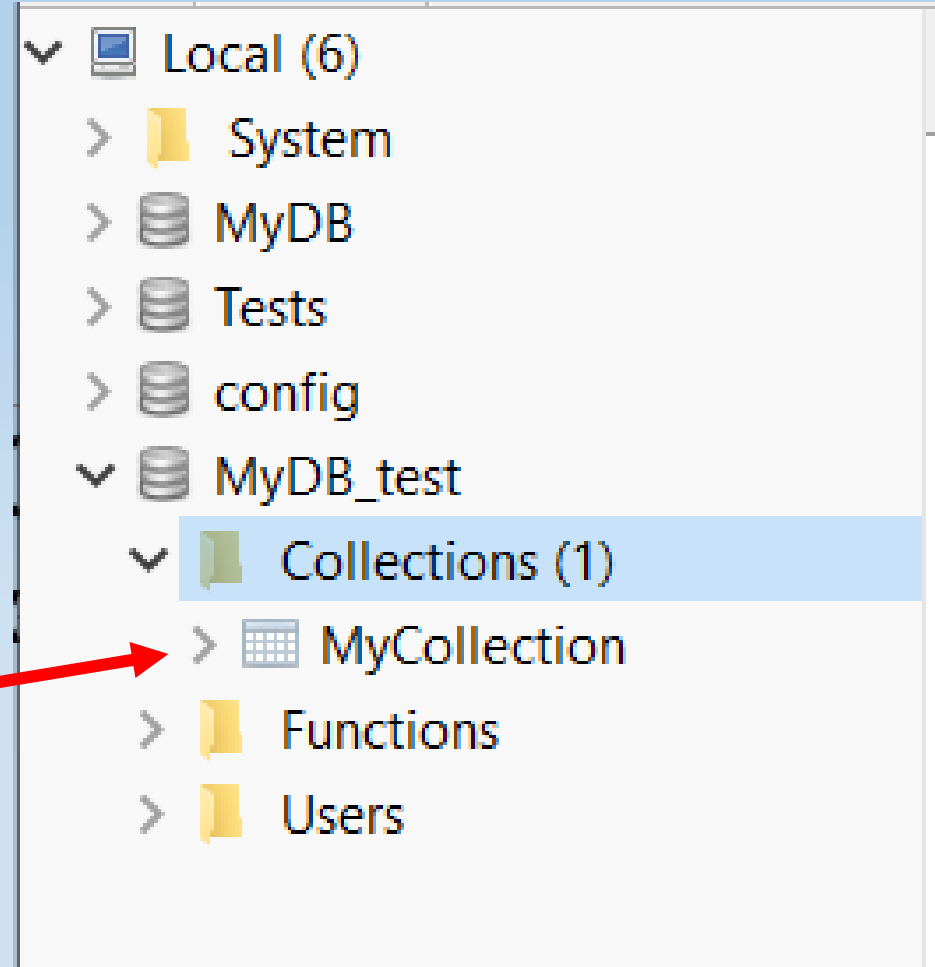
- Creiamo la collezione «MyCollection»





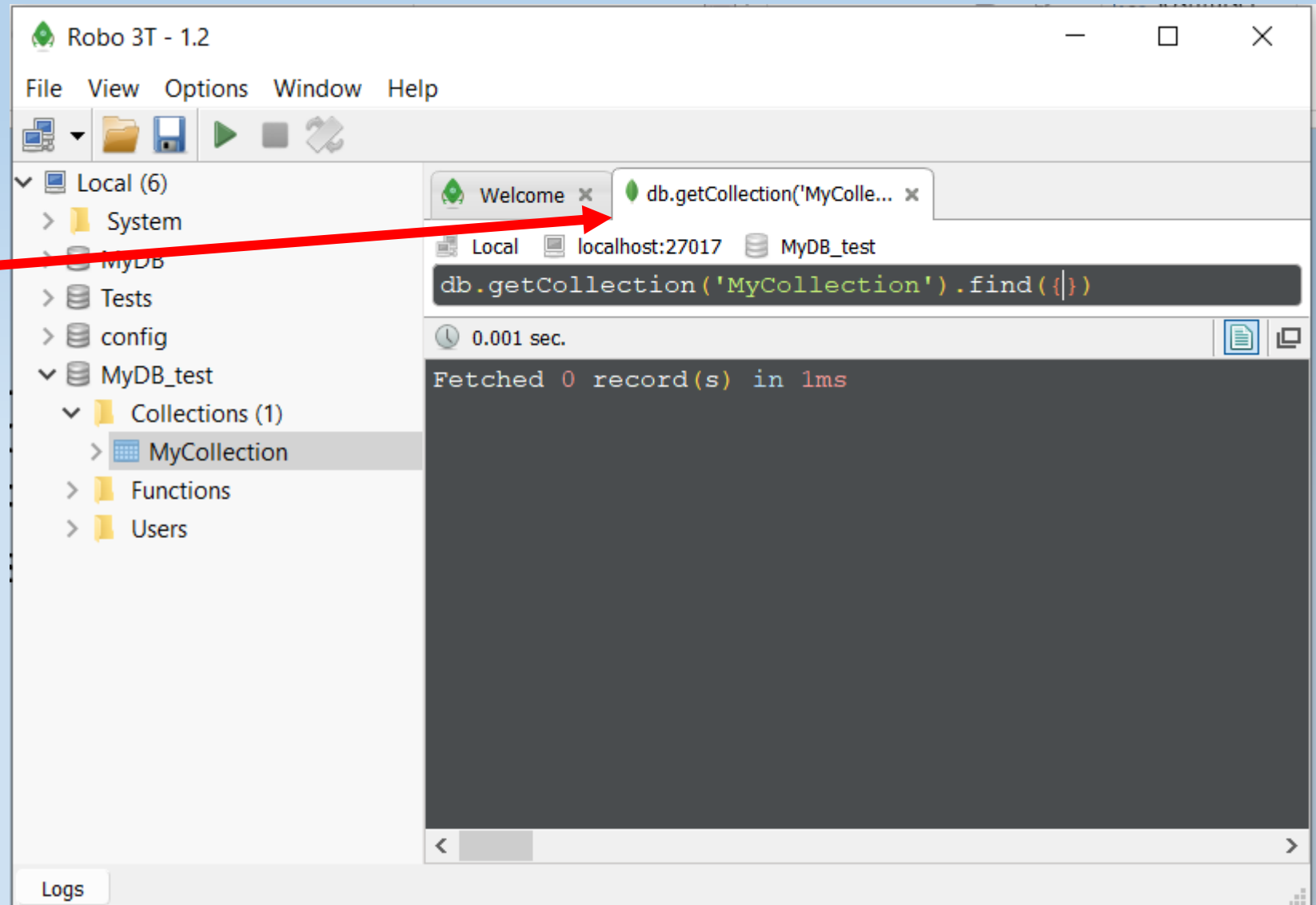
# Robo 3T

- La Collezione è stata creata
- Facendo doppio click sulla collezione ...



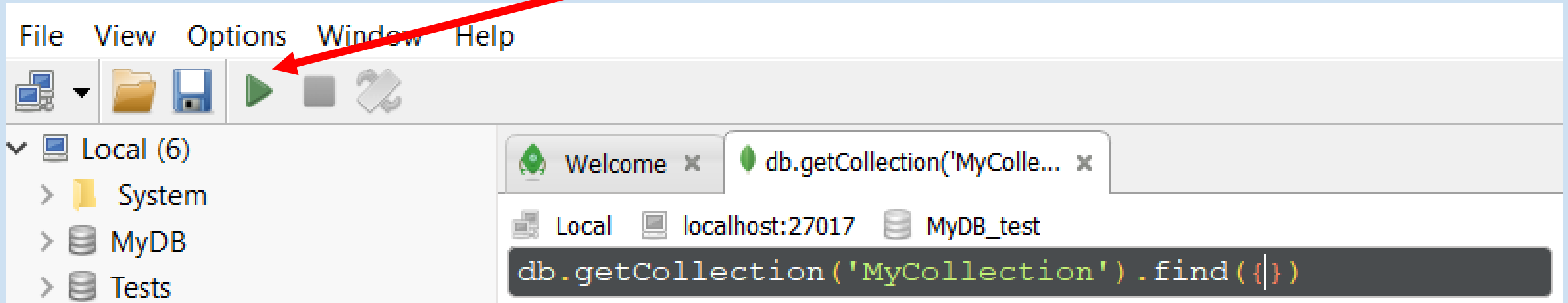
# Robo 3T

- ... si apre una scheda con
  - La query eseguita (per ottenere il contenuto della collezione)
  - La lista dei risultati (adesso vuota)



# Robo 3T

- L'area di testo che riporta la query è editabile: possiamo scrivere una nuova query
- E la possiamo eseguire con il pulsante verde



# Linguaggio di Query - MQL

- Il linguaggio di query di MongoDB è molto diverso da SQL
- Adotta un approccio a oggetti, con la dot notation
- Oggetto di base:  
db
- Corrisponde all'intero database, che ovviamente contiene le collezioni

# Linguaggio di Query: Collezioni

- Approccio alla JSON:  
ogni collezione è un campo dell'oggetto db, con il nome differente da quello delle altre collezioni  
`db.MyCollection`
- In alternativa:  
`db.getCollection('MyCollection')`
- L'oggetto collezione fornisce alcuni metodi per interrogare la collezione o modificarla (operazioni DML)

# Linguaggio di Query: JavaScript?

- Sì, è proprio così
- db è un oggetto JavaScript
- Infatti, possiamo definire funzioni JavaScript da usare nelle query
- Oppure scrivere proprio delle funzioni Javascript che svolgono query complesse (noi non lo vediamo, vi lancio la sfida, provate a cercare come si fa)

# DML: Inserimento

- **db.collezione.insertOne(oggetto)**  
Inserisce l'oggetto nella collezione.  
Restituisce unn oggetto che riporta l'id del documento creato
- **db.collezione.insertMany(array di oggetti)**  
Inserisce un array di oggetti nella collezione.  
Restituisce un oggetto con l'elenco degli ID creati

# DML: Inserimento

- `db.MyCollection.insertOne({ name: "Pippo"})`
- Viene mostrato l'oggetto restituito:  
acknowledged: true      inserimento avvenuto  
insertedId                      id assegnato al documento

```
db.MyCollection.insertOne({ name: "Pippo"})
```

0.003 sec.

Key	Value	Type
▼ (1)	{ 2 fields }	Object
<input checked="" type="checkbox"/> acknowledged	true	Boolean
<input type="checkbox"/> insertedId	ObjectId("5ea94e26b4f0...	ObjectId



# DML: Inserimento

```
db.inventory.insertMany([
  {item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" },
    status: "A" },
  {item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" },
    status: "A" },
  {item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" },
    status: "D" },
  {item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" },
    status: "D" },
  {item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" },
    status: "A" }
]);
```

# DML: Inserimento

- Nell'oggetto restituito, al posto del campo «insertedId» ora abbiamo il campo «insertedIds» (array)

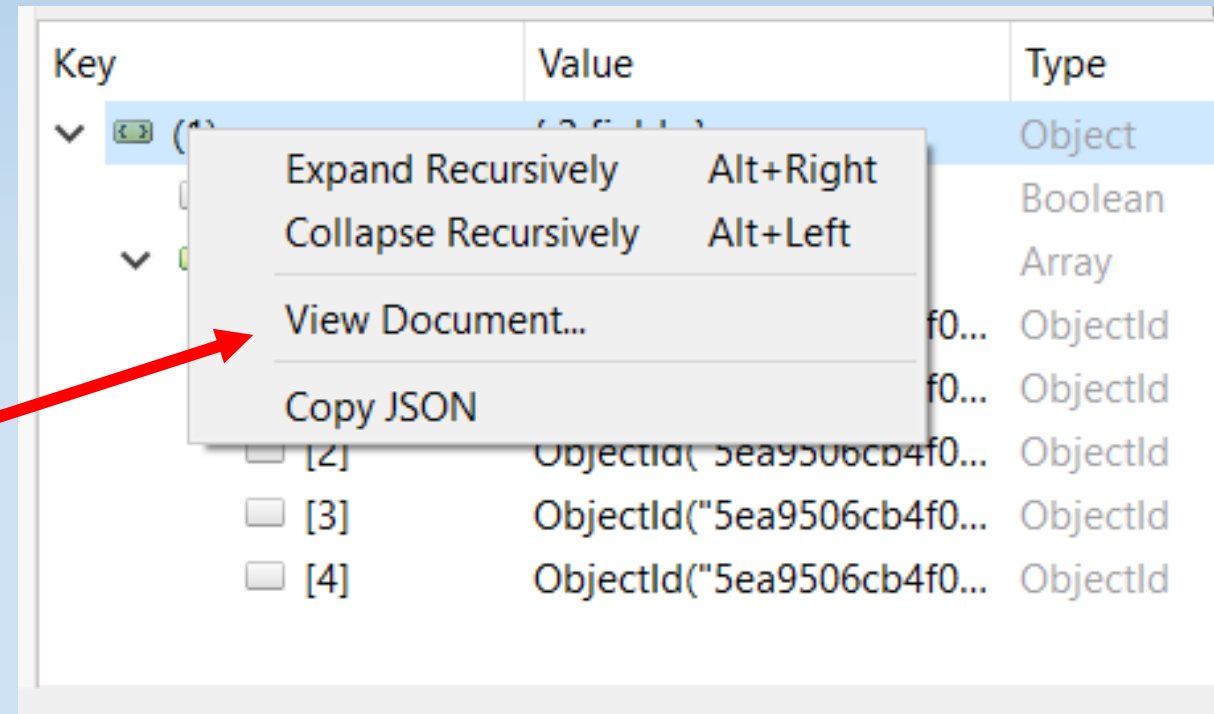
```
Local localhost:27017 MyDB_test
db.inventory.insertMany([
  { item: "journal", qty: 25, size: { h: 14, w: 21, u: "landscape" },
  { item: "notebook", qty: 50, size: { h: 8.5, w: 11, u: "portrait" },
  { item: "paper", qty: 100, size: { h: 8.5, w: 11, u: "landscape" },
  { item: "planner", qty: 75, size: { h: 22.85, w: 30, u: "landscape" },
  { item: "postcard", qty: 45, size: { h: 10, w: 15.5, u: "landscape" }
]);
```

0.036 sec.

Key	Value	Type
✓ (1)	{ 2 fields }	Object
acknowledged	true	Boolean
✓ insertedIds	[ 5 elements ]	Array
[0]	ObjectId("5ea9506cb4f0...)	ObjectId
[1]	ObjectId("5ea9506cb4f0...)	ObjectId
[2]	ObjectId("5ea9506cb4f0...)	ObjectId
[3]	ObjectId("5ea9506cb4f0...)	ObjectId
[4]	ObjectId("5ea9506cb4f0...)	ObjectId

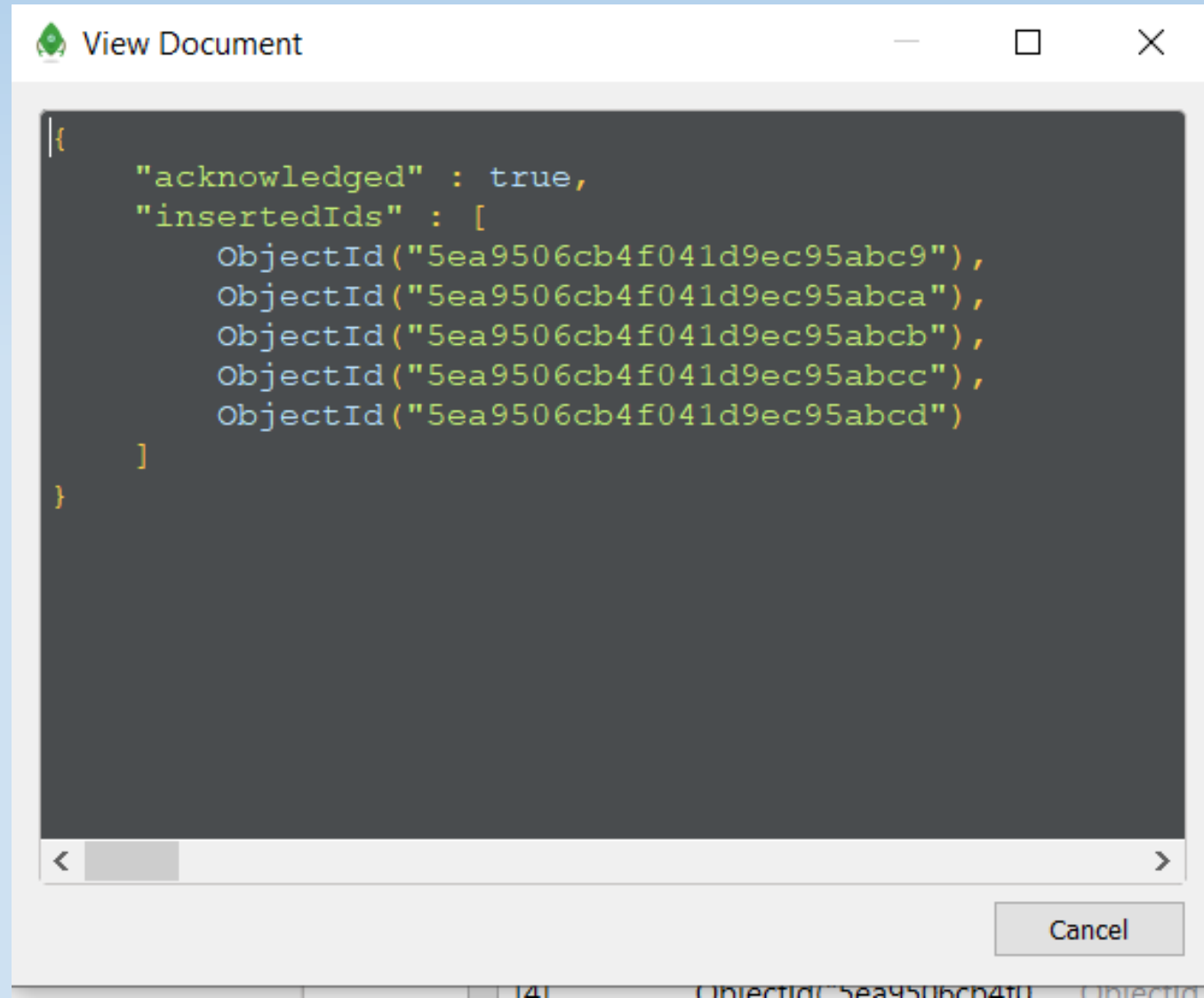
# Vedere il Documento

- Per vedere il documento cliccare con il tasto di destra sulla radice e scegliere «View Document...»



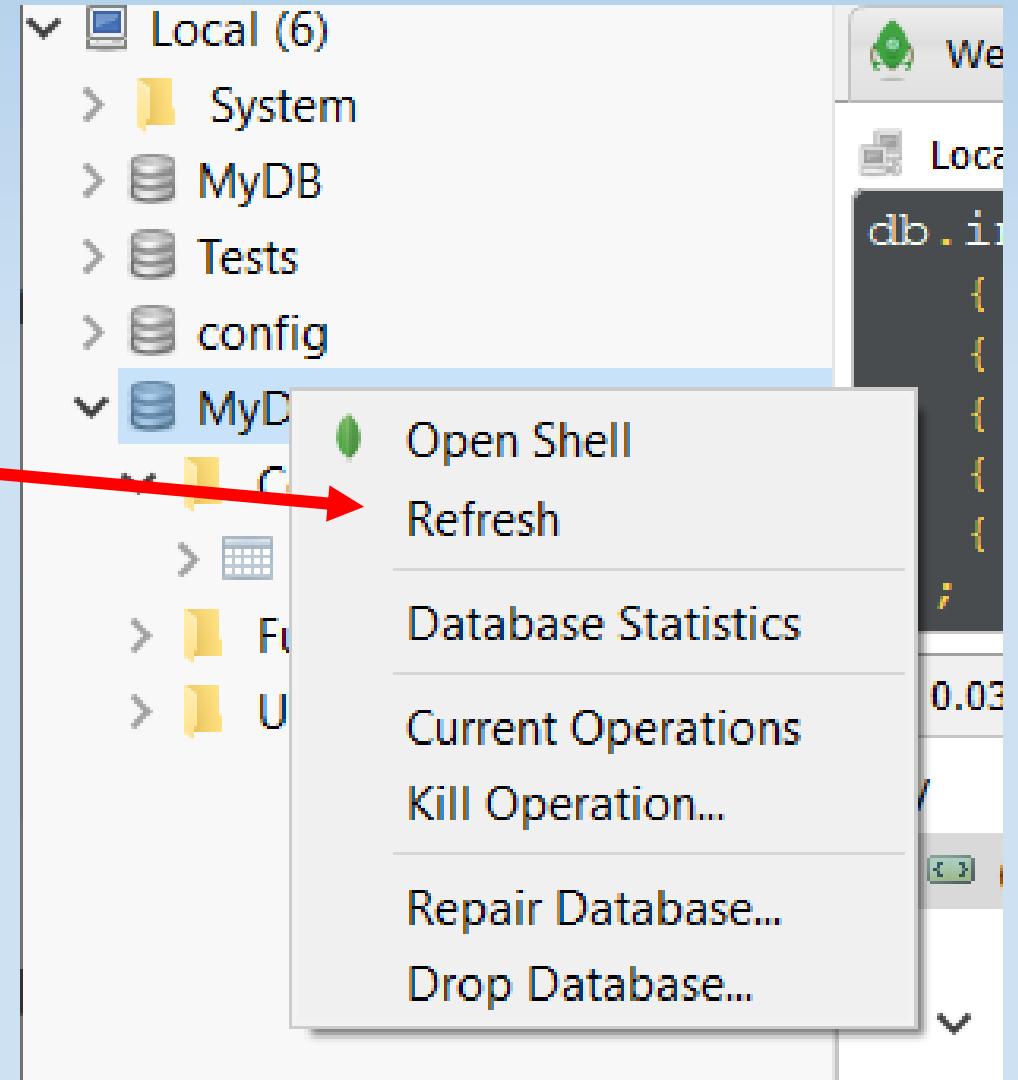
# Vedere il Documento

- Si apre una finestra secondaria che mostra il documento JSON



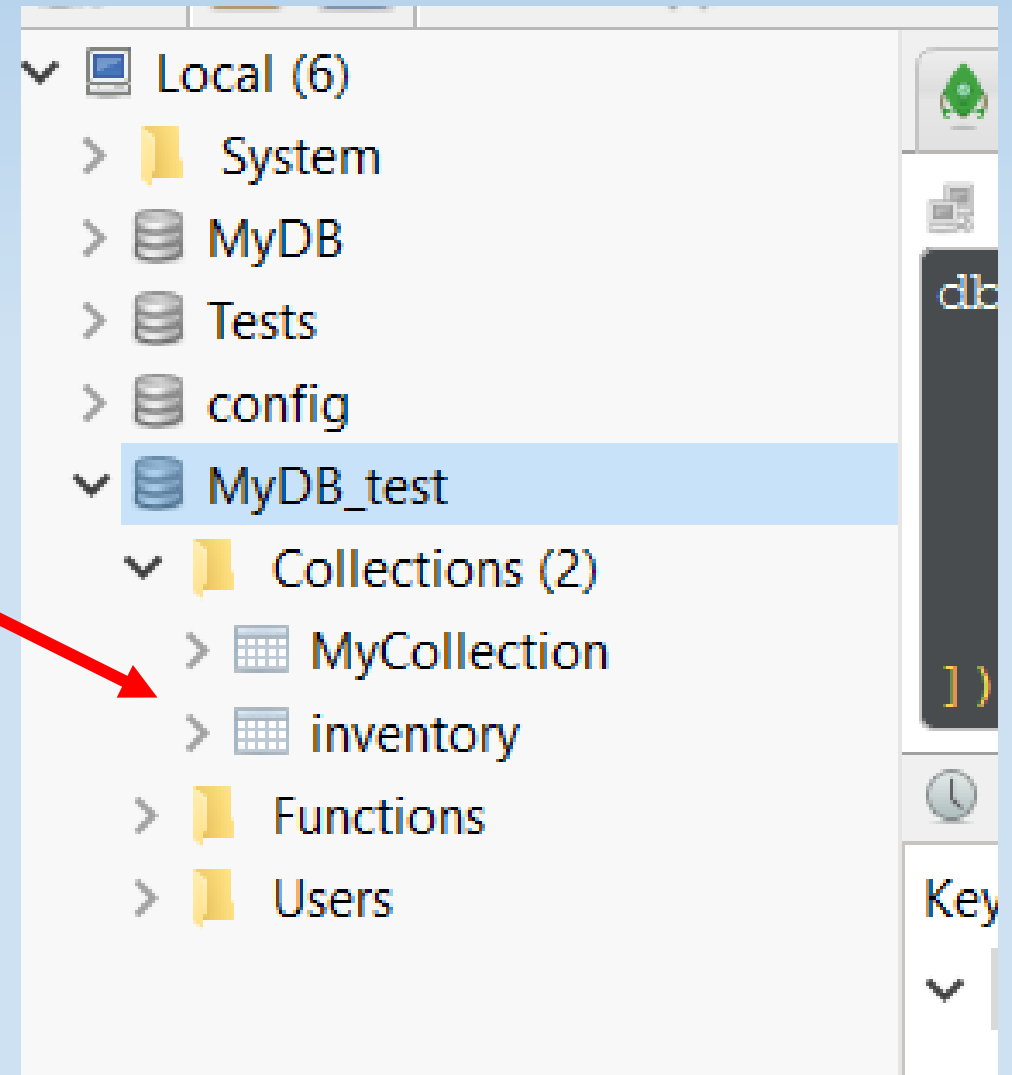
# La Collezione «inventory»?

- Non c'era
- Ma non ha dato errore
- Fate «refresh» sul db



# La Collezione «inventory»?

- La collezione è stata creata



# La Collezione «inventory»?

- Con un doppio click sulla collezione, si possono vedere i documenti

```
db.getCollection('inventory').find({})
```

Key	Value	Type
▼ (1) ObjectId("5ea...)	{ 5 fields }	Object
_id	ObjectId("5ea9506cb4f0...)	ObjectId
item	journal	String
qty	25.0	Double
> size	{ 3 fields }	Object
status	A	String
> (2) ObjectId("5ea...)	{ 5 fields }	Object
> (3) ObjectId("5ea...)	{ 5 fields }	Object
> (4) ObjectId("5ea...)	{ 5 fields }	Object
> (5) ObjectId("5ea...)	{ 5 fields }	Object

# Linguaggio di Query: find

- L'oggetto collezione fornisce alcuni metodi per interrogare la collezione. Il più usato è «find»
- **db.collezione.find( argomento )**
- L'argomento è un oggetto che specifica le condizioni di selezione
- Vediamo nel seguito come usarla



# Linguaggio di Query: find

- Se non si specifica nessun oggetto
- o si specifica l'oggetto vuoto
- Tutto il contenuto della collezione è recuperato, senza nessuna alterazione
- Esempio:  
`db.inventory.find()`  
`db.inventory.find({})`

# Linguaggio di Query: find

- Specificando un campo con un valore
- Equivale ad una condizione di uguaglianza, su un campo che deve essere presente
- La struttura non viene alterata
- Esempio:  
`db.inventory.find( { status: "D" } )`

# Linguaggio di Query: find

- Se il valore è un oggetto annidato, si possono specificare operatori di vario tipo
- Sintassi:  
**{ <field1>: { <operator1>: <value1> }, ... }**
- dove l'operatore è un campo il cui nome inizia con «\$»

# Linguaggio di Query: Operatori

- { <field>: { \$eq: <value> } }

Il campo deve avere il valore specificato

- {<field>: {\$ne: <value>} }

Il campo deve avere un valore diverso da quello indicato

# Linguaggio di Query: Operatori

- **{<field>: {\$gt: <value>} }**

Il campo deve avere un valore maggiore di quello indicato

- **{<field>: {\$gte: <value>} }**

Il campo deve avere un valore maggiore o uguale al valore indicato

# Linguaggio di Query: Operatori

- **{<field>: {\$lt: <value>} }**

Il campo deve avere un valore minore di quello indicato

- **{<field>: {\$lte: <value>} }**

Il campo deve avere un valore minore o uguale al valore indicato

# Linguaggio di Query: Operatori

- **{ field: { \$in: [<value1>, <value2>, ... <valueN> ] } }**  
Il campo deve avere un valore contenuto nell'array di valori specificato
- Esempio:  
`db.inventory.find( { qty: { $in: [ 5, 15 ] } } )`
- **\$nin**: vero se il valore non è nell'array specificato

# Linguaggio di Query: AND

- L'AND è implicito: due o più operatori o due o più campi nello stesso oggetto
- Esempio:  
`db.inventory.find( { qty: { $gte: 50, $lte: 150 } } )`  
il campo «qty» deve essere tra 50 e 150 (inclusi)
- `db.inventory.find( { qty: { $gte: 50, $lte: 150 }, status: "A" } )`
- Alla precedente condizione, aggiungiamo in AND che il campo «status» deve valere «A»



# Linguaggio di Query: AND

- C'è comunque un operatore **\$and**
- **\$and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ]**
- Il valore è un array di oggetti che riportano condizioni/confronti
- Esempio:  
`db.inventory.find( { $and: [ { price: { $ne: 1.99 } },  
 { price: { $exists: true } } ] } )`

# Altri Operatori Logici

- OR: **\$or**

```
db.inventory.find( { $or: [ { qty: { $lt: 20 } },  
                             { price: 10 } ] } )
```

- NOR (OR negato): **\$nor**

```
db.inventory.find( { $nor: [ { qty: { $lt: 20 } },  
                             { price: 10 } ] } )
```

- NOT (un solo argomento): **\$not**

```
db.inventory.find( { price: { $not: { $gt: 1.99 } } } )
```

che restituisce anche i documenti che NON hanno il campo «price»

# Altri Operatori Logici

- Questa scrittura è corretta?  
`db.inventory.find({$not: { price: { $gt: 1.99 } } } )`
- No, viene segnalato che «\$not» non è un «top-level operator»
- Il suo uso è limitato come abbiamo visto prima

# Altri Operatori sui Documenti

- **\$exists**
- **<field>: { \$exists: <boolean> }**  
se <boolean> è true, è vero se il documento contiene il campo indicato  
se <boolean> è false, è vero se il documento non contiene il campo indicato
- Esempio:  
`db.inventory.find( { qty: { $exists: true, $nin: [ 5, 15 ] } } )`

# Altri Operatori sui Documenti

- **\$type**
- **field: { \$type: <typeName> }**  
è vero se il campo ha il tipo specificato
- Esempio:  
`db.inventory.find( { qty: { $type: "string"} } )`

# Lista dei Tipi

- "double"
- "string"
- "object"
- "array"
- "binData"
- "objectId"
- "bool"
- "date"
- "null"
- "javascript"
- "int"
- "number"
- "timestamp"
- "long"

# Documenti negli Array

- [illegible]

# Altri Operatori sui Documenti

Per completezza di informazione, esistono altri operatori di confronto, che non riportiamo

- Ricerca in campi di testo
- Confronti di tipo geo-spaziale

Per gli interessati, al link seguente si trova l'elenco di tutti gli operatori

<https://docs.mongodb.com/manual/reference/operator/query/>



# Documenti Annidati

- Come gestire i documenti anidati?
- Costruendo la struttura annidata  
`db.inventory.find( { size: {h : 8.5, w : 11.0, uom : "in" } } )`
- **Problema:** il match deve essere esatto, cioè i campi devono essere esattamente in quell'ordine e devono esserci tutti

# Documenti Annidati

- Oppure, si usano le dot notation per i nomi dei campi

```
db.inventory.find( { "size.h": 8.5, "size.uom" : "in" } )
```

così, si può fare un matching parziale, con espressioni complesse

# Lavorare sul Risultato di find

- Alcune funzioni possono essere applicate a valle di find
- **limit(n)**  
limita a n il numero di oggetti nel risultato
- **skip(n)**  
salta i primi n elementi nel risultato
- **count()**  
conta gli elementi nel risultato

# Lavorare sul Risultato di find

- **sort( {<field>: <option>, ...} )**  
ordina gli oggetti del risultato in base alle chiavi di ordinamento specificate, il valore <option> può valere 1 (ordine ascendente) o -1 (ordine discendente)
- Esempio  
`db.MyCollection.find().sort({name: 1}).skip(1)`

# Lavorare sul Risultato di Cursori

- find non restituisce oggetti di tipo «collection»
- Invece, restituisce dei «cursor»
- Di conseguenza, i metodi che si possono usare dopo find non sono gli stessi che si possono usare sulle collezioni base (vedi dopo)

# Cambiare gli Oggetti

- **find( query, projection)**  
il parametro opzionale «projection» specifica come ristrutturare i documenti nel risultato
- **<field>: <option>**  
se <option> vale 1, il campo viene tenuto  
se <option> vale 0, il campo viene eliminato
- Attenzione: o si include, o si esclude  
`db.getCollection('inventory').find({}, {qty: 0, size: 0})`  
`db.getCollection('inventory').find({}, {qty: 1, size: 1})`  
`db.getCollection('inventory').find({}, {qty: 0, size: 1})`

# Collection Methods

- Le collezioni (sia prima che dopo find) hanno metodi specifici
- Le operazioni DML e find sono collection methods
- Sono molti, ne riportiamo alcuni:
- **count()**  
conta i documenti nella collezione
- **insertOne, insertMany**  
inseriscono gli oggetti nella collezione

# Collection Methods

- **deleteMany( condizione )**  
cancella i documenti che soddisfano la condizione (come in find)
- **updateOne, updateMany**  
aggiornano il contenuto dei documenti selezionati  
L'aggiornamento può essere piuttosto complicato e non riusciamo a trattarlo, o meglio, le espressioni possono essere complicate  
Nella prossima slide riportiamo un esempio esplicativo



# Collection Methods

- `db.inventory.updateOne({status: "D"},{$set: { qty: 10}})`  
`db.inventory.updateMany({qty: {$lte: 15}},`
  - `{ $set: { refurnish: true}})`
- La prima query cambia la quantità del primo documento che trova con il valore «D» per il campo «status»
- La seconda aggiunge il campo «refurnish» con valore pari a «true» per tutti i documenti che hanno valori del campo «qty» minori o uguali a 15

# **Esempio: Node.js e MongoDB**

# I Driver per il DBMS

- Per far connettere Node.js ad un DBMS, occorre installare il driver
- Il driver per MongoDB si può installare con il comando **`npm install mongodb --save`**

# Come Avviare la Connessione

- Importare il modulo «mongodb»

```
const mongo = require('mongodb')
```

- Impostare alcune costanti di servizio

```
const db_name="MyDB_test";
```

```
const collection_name = "mycollection";
```

```
const db_url = "mongodb://localhost:27017";
```

# Come Avviare la Connessione

- Creare il client che deve gestire la connessione

```
var client_config =  
  { useUnifiedTopology: true,  
    useNewUrlParser: true };
```

```
const client = new  
mongo.MongoClient(db_url, client_config)
```

# Come Avviare la Connessione

- Quando serve accedere al db, si effettua la connessione

```
client.connect(async function(err) {  
  if(err)  
  {  
    console.log("Error connecting to  
MongoDB");  
    throw err;  
  }  
  ...  
})
```
- La funzione di callback viene chiamata dopo il tentativo di connessione

# **async?**

- **async** specifica che la funzione è asincrona
- Quando viene chiamata, non viene eseguita immediatamente
- Viene inserita nelle code degli eventi ed eseguita in modo asincrono dall'event loop

# async?

- Questo è necessario perché contattare il DBMS richiede tempo e viene fatto con la gestione a eventi
- Perciò, i metodi per accedere al DBMS vanno chiamati con l'opzione «await» prima della chiamata
- Questa opzione mette in attesa la funzione chiamante, generando un nuovo evento
- Solo le funzioni «async» possono fare chiamate «await»



# Accedere al DB e alla Collezione

- Accedere al db con l'istruzione seguente  
`db = await client.db(db_name);`
- Ottenere la collezione  
`var collection =  
 await db.collection(collection_name);`
- A questo punto, si possono usare i collection method di MongoDB per fare le query e gli update

# Chiudere la Connessione

- Finito il lavoro, chiudere la connessione con `client.close()` ;
- Si evita, così, di tenere troppe connessioni aperte

# Esempio

- Nel file `serverDb.js`, trovate il codice di un server che gestisce due richieste:
- `/insert?f1&v1&f2=v2&...`  
Inserisce un nuovo documento nella collezione con i campi e i valori riportati come `searchParams`
- `/list`  
produce un documento JSON che contiene i primi 10 documenti nella collezione

# Parte Preliminare

```
const http = require('http')
const url = require('url')
const mongo = require('mongodb')

const hostname = '127.0.0.1'
const port = '8080'

const MIME_json = "application/json";
const MIME_text = "text/plain";
var MIME = MIME_text;
var output_text = "";

const db_name="MyDB_test";
const collection_name = "mycollection";
const db_url = "mongodb://localhost:27017";
```

# HTTP Server

```
const server = http.createServer(function (req, res) { ... });

server.listen(port, hostname, function () {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

# Callback di HTTP Server

```
function (req, res) {  
    console.log("Request Received\n");  
    var myurl = new url.URL("http://" + req.headers.host  
+ req.url);  
    var params = myurl.searchParams;
```

...

# Connessione al DBMS

...

```
var client_config =  
  { useUnifiedTopology: true,  
    useNewUrlParser: true };  
const client = new mongo.MongoClient(db_url,  
  client_config)  
  
// Connect to the db  
client.connect(async function(err) { ... })  
}
```

# Callback di `client.connect`

- La creazione del client va fatta per ogni richiesta http
- Perché la connessione ripetuta per lo stesso client è deprecata



# Callback di client.connect

```
async function(err) {  
  if(err)  
  {  
    console.log("Error connecting to  
MongoDB");  
    throw err;  
  }  
  db = await client.db(db_name);  
}
```

# Callback di `client.connect`

- Nella prima parte, si verifica se si è verificato un errore di connessione
- Poi, si accede al db desiderato, ottenendo l'oggetto «db»

# Richiesta /insert – Parte 1

```
if( myurl.pathname == "/insert" )  
{  
    var o = new Object();  
    for (const [name, value] of params)  
    {  
        o[name] = value;  
    }  
}
```

# Richiesta /insert – Parte 2

```
var collection =  
    await db.collection(collection_name);  
var r = await collection.insertOne( o );  
if(r.result.ok == 1 )  
{ output_text =  
    JSON.stringify( {inserted: 1 } );  
    MIME = MIME_json;  
}  
}
```

# Richiesta /insert – Parte 1

- La variabile «myurl» gestisce l'URL della richiesta HTTP
- Il campo «pathname» contiene la parte di URL che segue il nome di dominio, senza i parametri, cioè dal primo «/» a «?»
- L'istruzione if verifica se la richiesta è per «/insert»
- Se lo è, estrae tutti i parametri e crea l'oggetto «o» da inserire nel db

# **Richiesta /insert – Parte 2**

- Si accede alla collezione
- Poi si effettua l'inserimento
- L'oggetto restituito serve per generare un nuovo oggetto di output, che viene serializzato nella variabile «output\_text»
- Infine, si imposta il MIME type corretto e il lavoro terminerà con l'invio della risposta (aòòa fine della callback)

# Richiesta /list – Parte 1

```
if( myurl.pathname == "/list" )  
{  
  var collection =  
    await db.collection(collection_name);  
  const docs =  
    await collection.find().limit(10).toArray();  
  const total =  
    await collection.find().count();
```

# Richiesta /list – Parte 2

```
var o = new Object();
```

```
o.Total = total;
```

```
o.Selected = 10;
```

```
o.docs = docs;
```

```
output_text = JSON.stringify( o );
```

```
MIME = MIME_json;
```

```
}
```



# **Richiesta /list – Parte 1**

- Si accede alla collezione
- Si effettua la query, trasformando il risultato in un array che viene assegnato alla costante «docs»
- Si fa un'altra query, per contare i documenti totali nella collezione; il risultato viene assegnato alla costante «total»

# **Richiesta /insert – Parte 2**

- Si predispone l'oggetto da serializzare nella risposta, con tre campi:
  - Total, il totale dei documenti nella collezione
  - selected, fissato a 10
  - docs, l'array con i documenti
- Si serializza l'oggetto e si predispone il MIME type
- La risposta verrà inviata fuori dall'if

# Terminare la callback

```
client.close();
```

```
res.statusCode = 200
```

```
res.setHeader('Content-Type', MIME)
```

```
res.end(output_text);
```

```
}
```

# Terminare la callback

- La callback termina con la chiusura della connessione
- Seguita dall'invio della risposta
- Si noti che se la richiesta non è gestita, si invia la risposta vuota (vedi inizializzazione di «output\_text»)
- La risposta va inviata nella callback, perché è asincrona, quindi viene eseguita dopo la callback di HTTP server

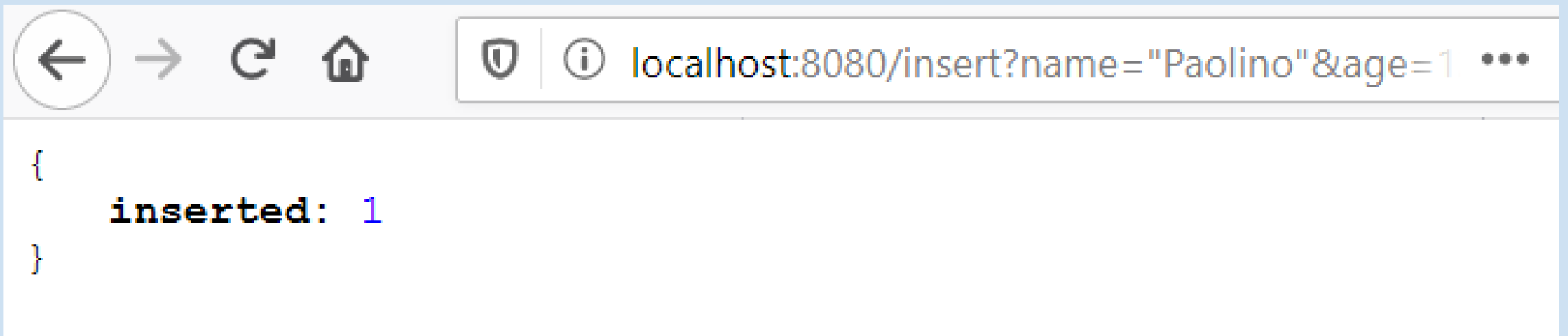
# Esecuzione

- Per prima cosa, mandiamo in esecuzione il server

```
C:\Users\Utente\Documents\Lavoro\corsi\Cloud_Mobile\Node.js>node serverDb.js  
Server running at http://127.0.0.1:8080/
```

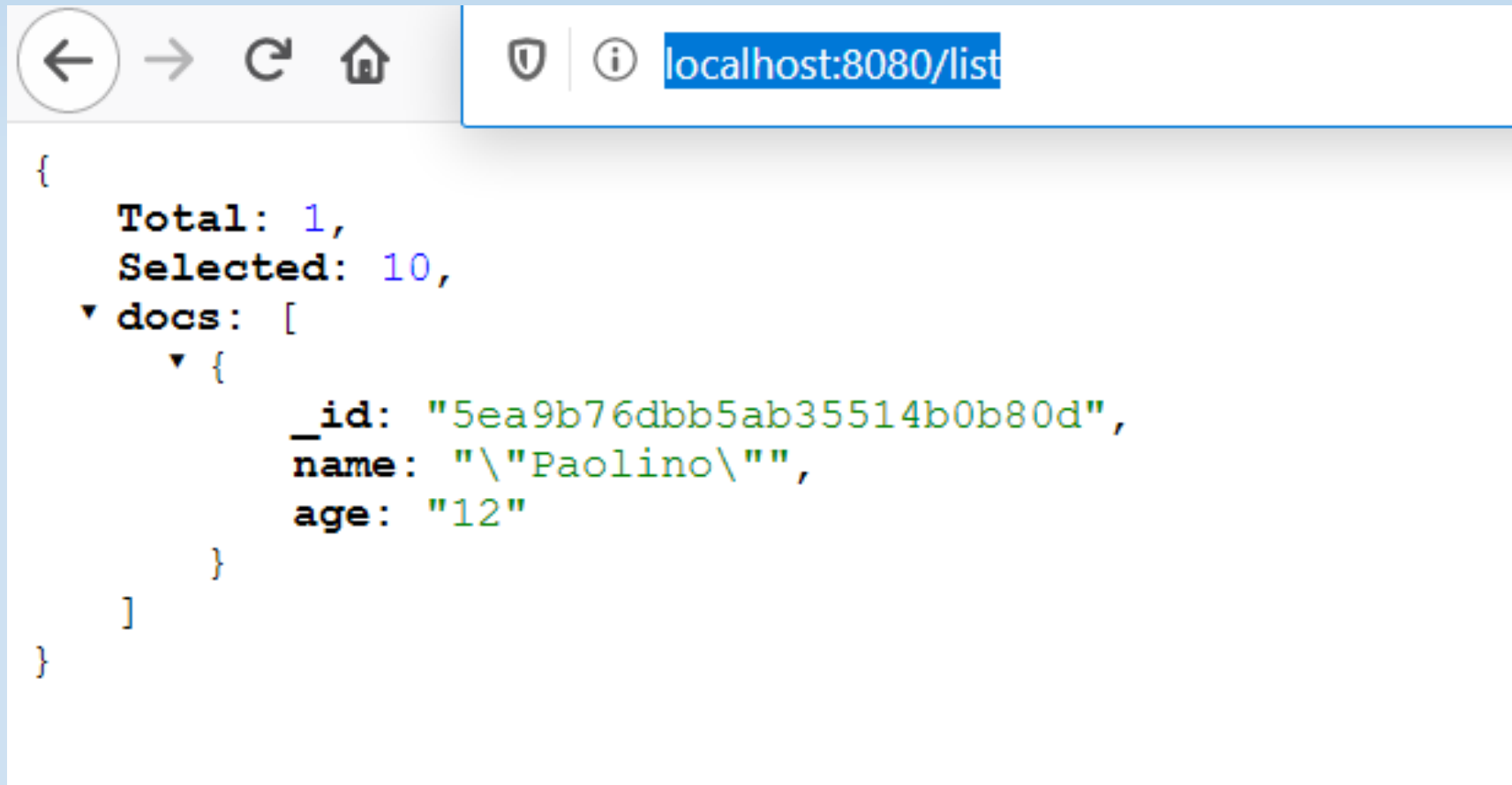
# Inseriamo un Documento

- <http://localhost:8080/insert?name=%22Paolino%22&age=12>



# Lista dei Dcoumenti

- <http://localhost:8080/list>



The screenshot shows a web browser window with the address bar displaying `localhost:8080/list`. The main content area displays a JSON response, which is partially expanded to show the `docs` array. The JSON structure is as follows:

```
{
  Total: 1,
  Selected: 10,
  docs: [
    {
      _id: "5ea9b76dbb5ab35514b0b80d",
      name: "\"Paolino\"",
      age: "12"
    }
  ]
}
```

# Ma MyCollection non era vuota

- Infatti, ma nel programma il nome è «mycollection»



The screenshot shows the MongoDB interface. On the left, the database structure is visible, with 'MyDB\_test' expanded to show 'Collections (3)', including 'MyCollection', 'inventory', and 'mycollection'. The 'mycollection' collection is selected. The main window displays the command `db.getCollection('mycollection').find({})` in the command bar. Below the command bar, the results are shown in a table format. The table has three columns: 'Key', 'Value', and 'Type'. The results show a single document with three fields: '\_id' (ObjectId), 'name' (String), and 'age' (String).

Key	Value	Type
(1) ObjectId("5ea...")	{ 3 fields }	Object
_id	ObjectId("5ea9b76dbb5...")	ObjectId
name	"Paolino"	String
age	12	String