LiteDB

Fork me on GitHub

# Object Mapping

The LiteDB mapper converts POCO classes documents. When you get a `ILiteCollection<T>` instance from `LiteDatabase.GetCollection<T>` , `T` will be your document type. If `T` is not a `BsonDocument` , LiteDB internally maps your class to `BsonDocument` . To do this, LiteDB uses the `BsonMapper` class:

```
// Simple strongly-typed document
public class Customer
{
    public ObjectId CustomerId { get; set; }
    public string Name { get; set; }
    public DateTime CreateDate { get; set; }
    public List<Phone> Phones { get; set; }
    public bool IsActive { get; set; }
}

var typedCustomerCollection = db.GetCollection<Customer>("customer");

var schemelessCollection = db.GetCollection("customer"); // <T> is BsonDocument
```

## Mapper conventions

`BsonMapper.ToDocument()` auto converts each property of a class to a document field following these conventions:

- Properties can be read-only or read/write
- The class should have an `Id` property, `<ClassName>Id` property, a property with `[BsonId]` attribute or mapped by the fluent API.
- A property can be decorated with `[BsonIgnore]` in order not to be mapped to a document field
- A property can be decorated with `[BsonField("fieldName")]` to customize the name of the document field
- No circular references are allowed
- By default, max depth of 20 inner classes (this can be changed in the `BsonMapper`)
- You can use `BsonMapper` global instance (`BsonMapper.Global`) or a custom instance and pass to `LiteDatabase` in its constructor. Keep this instance in a single place to avoid re-creating the mappings each time you use a database.

In addition to basic BSON types, `BsonMapper` maps others .NET types to BSON data type:

| .NET type | BSON type |
|---|---|
| `Int16` , `UInt16` , `Byte` , `SByte` | Int32 |
| `UInt32` , `UInt64` | Int64 |
| `Single` | Double |
| `Char` , `Enum` | String |
| `IList<T>` | Array |
| `T[]` | Array |
| `IDictionary<K,T>` | Document |
| Any other .NET type | Document |

- `Nullable<T>` are accepted. If value is `null` the BSON type is Null, otherwise the mapper will use `T?` .

- For `IDictionary<K, T>`, `K` key must be `String` or a simple type (convertible using `Convert.ToString(..)`).

## Constructors

Starting with version 5 of LiteDB you can use `BsonCtorAttribute` to indicate which constructor the mapper must use. Fields no longer need to have a public setter and can be initialized by the constructor.

```
public class Customer
{
    public ObjectId CustomerId { get; }
    public string Name { get; }
    public DateTime CreationDate { get; }
    public bool IsActive { get; }

    public Customer(string name, bool isActive)
    {
        CustomerId = ObjectId.NewObjectId();
        Name = name;
        CreationDate = DateTime.Now;
        IsActive = true;
    }

    [BsonCtor]
    public Customer(ObjectId _id, string name, DateTime creationDate, bool isAct
    {
        CustomerId = _id;
        Name = name;
        CreationDate = creationDate;
        IsActive = isActive;
    }
}

var typedCustomerCollection = db.GetCollection<Customer>("customer");
```

When `GetCollection<T>` is called, it tries to create instances of `T` by searching for a constructor in the following order:

- First, it searches for a constructor with `BsonCtorAttribute`
- Then, it searches for a parameterless constructor (and assumes all serialized fields are public and all serialized properties have public setters)
- Finally, it searches for a constructor whose parameters names match with the names of the fields in the document

Please note that all the parameters in the constructor annotated with `BsonCtorAttribute` must be of a simple type, `BsonDocument` or `BsonArray` .

## Register a custom type

You can register your own map function, using the `RegisterType<T>` instance method. To register, you need to provide both serialize and deserialize functions.

```
BsonMapper.Global.RegisterType<Uri>
(
    serialize: (uri) => uri.AbsoluteUri,
    deserialize: (bson) => new Uri(bson.AsString)
);
```

- `serialize` function receives an instance of `T` and returns an instance of `BsonValue`
- `deserialize` function receives an instance of `BsonValue` and returns an instance of `T`
- `RegisterType` supports complex objects via `BsonDocument` or `BsonArray`

## Mapping options

`BsonMapper` class settings:

| Name | Default | Description |
|---|---|---|
| `SerializeNullValues` | false | Serialize field if value is `null` |
| `TrimWhitespace` | true | Trim strings properties before mapping to document |
| `EmptyStringToNull` | true | Empty strings convert to `null` |
| `ResolvePropertyName` | (s) => s | A function to map property name to document field name |
| `EnumAsInteger` | false | Map enum to `string` (default) or to `int` |
| `IncludeFields` | false | If mapper should include all class fields |
| `IncludeNonPublic` | false | If mapper should include all private/protected fields/properties |

| Name | Default | Description |
|---|---|---|
| `ResolveCollectionName` | typeof(T).Name | When collection name are omitted, use this collection name resolver function |

Please note that Linq expressions in typed collections will only work over Enum fields if `EnumAsInteger = true`.

`BsonMapper` offers 2 predefined functions to resolve property names: `UseCamelCase()` and `UseLowerCaseDelimiter('_')`.

```
BsonMapper.Global.UseLowerCaseDelimiter('_');

public class Customer
{
    public int CustomerId { get; set; }

    public string FirstName { get; set; }

    [BsonField("customerLastName")]
    public string LastName { get; set; }
}

var doc = BsonMapper.Global.ToDocument(new Customer { FirstName = "John", LastNa

var id = doc["_id"].AsInt;
var john = doc["first_name"].AsString;
var doe = doc["customerLastName"].AsString;
```

## AutoId

There are 4 built-in auto-id functions implemented:

- `ObjectId`: `ObjectId.NewObjectId()`
- `Guid`: `Guid.NewGuid()` method
- `Int32/Int64`: New collection sequence

AutoId is only used when there is no `_id` field in the document upon insertion. In strongly-typed documents, `BsonMapper` removes the `_id` field for empty values (like `0` for `Int` or `Guid.Empty` for `Guid`). Please note that AutoId requires the id field to have a public setter.

## Fluent Mapping

LiteDB offers a complete fluent API to create custom mappings without using attributes, keeping you domain classes without external references.

Fluent API uses `EntityBuilder` to add custom mappings to your classes.

```
var mapper = BsonMapper.Global;

mapper.Entity<MyEntity>()
    .Id(x => x.MyCustomKey) // set your document ID
    .Ignore(x => x.DoNotSerializeThis) // ignore this property (do not store)
    .Field(x => x.CustomerName, "cust_name"); // rename document field
```