Little Guide for
XOD Database Users
C# Edition

XOD DB: SIMPLICITY IS POWERFUL

xod

XML-Based Object Oriented
Relational Database Engine

A Mohammed Hassan I. Sallam Project
mhisallam@outlook.com / mail@mhsallam.com

>ACEs.developers();
Software Solutions

About the Author

My name is Mohammed Hassan I. Sallam, I’m a software engineer and I built XOD.

After graduation, I worked with software development companies, after that, I moved to work with NGOs, where I used more of my expertise in GIS and ICT than actual software development, but there was no time I remember where I stopped designing, programming and building software, either as side projects or just for fun. I guess, I’m a little bit addicted to this, and that’s not a bad thing.

I used to be disconnected from the rest of the community, you know... no Facebook, no Twitter and always busy -and still- with my work and projects. But that has changed, now, I feel more productive than ever before, and I want to reach to people and I want to engage with the rest of the world, specially my software-developers community.

XOD is the first, and I plan to publish more of my works, and I hope you would like them, like I do.

I would love it if you follow me on Twitter: @mhisallam
email me: mhisallam@outlook.com or mail@mhsallam.com
find me on LinkedIn:linkedin.com/in/mhsallam
check out my works on GitHub: github.com/mhsallam
and visit my website: www.mhsallam.com

Preface

At first, I developed XOD’s very first core as a favor for a friend back in 2010, who –at that time- wanted to develop a Windows Phone 7 application. He didn’t want to use standard file serialization to store data, and there was no Microsoft SQL Server CE in Windows Phone 7, like it used to be in Windows Mobile OS. Then I said “I can build an object oriented database engine that use XML files as backend in a couple of weeks”. It took me ten days to finish it, and I named it XOD. I couldn’t spend more time to develop it -except for fixing a bug here and there- even if I wanted to, because I was busy working on a big vehicles tracking system software project at the time.

After a while, I decided to use XOD as the backend of one of my web application projects, and another major upgrade made its way to XOD. When I actually used XOD for production I actually found it very easy to use. Strange huh! Coming from the one who created it, but I’m sure you will find it easy as well.

Now, I have decided to open source XOD for whoever is interested in using it or contributing to it. And I really want to see it grows bigger and nicer.

Oh, by the way, I would like to think that XOD’s logo resembles the name XOD even if that is not totally accurate –I might need to tweak it a little more later… you know, when I get the time ☺–, but I designed it to resemble a DNA bar, and that is cool.

Before we move on with the rest of this guide, I would like to say; thanks Wameed Abbas, you are the one I first build this project for, and thank you Wail AlSalahi, for your non-stop encouragement.

--*Mohammed Sallam*

Contents

A project of Mohammed Hassan I. Sallam
mhisallam@outlook.com

Introduction

XOD database -pronounced as “ZOD”- is an object oriented relational database for .NET and JAVA developers. XOD uses embedded XML files for storage, that means the data will be readable even without the application the database created for, and it is also better for integration and simple for data migration. Even If you didn’t like your database being readable like text files, you can always secure XOD files with encryption.

Based on the development approach, developers might start designing the application model classes, design the actual database storage and then build a data mapping layer between the two. They might go back and forth between these two design layers whenever new update comes up. But what if you can skip the second, and third layers, and just work on the model classes layer only, not warring about the database design and mapping objects to database records, because XOD will take it from there; that would be great, right! Even if your application is big that one XOD database might not be sufficient for, you can use unlimited number XOD databases, even for supportive tasks like configuration data storage. You can also use XOD at the development stages only, until you feel satisfied about your application model classes and business logic, then push your verified models design and build the actual database.

This little documentation gives you brief guide lines for how to utilize XOD for .NET developers.

1. XOD Features

XOD is an object oriented relational database engine, it means you don't have to think about using any kind of ROM (object-relational mapping) techniques in your code, you just send the objects into the database and read them as such using very simple set of functions. The term relational is a little different than the one of RDB, the reason I used the term *relational* is because XOD is aware of the hierarchy design of objects. It handles all connected object references and bring them back on read operations. For instance, when we read a XOD-persisted object which has been instantiated from `class Product`; it will bring all data of the target `Product` along with its connected `ProductCategory` reference-type property, plus whatever `ProductCategory`'s sub Categories the directly connected `ProductCategory` might have. You may say, that is not an efficient way of loading data, but hey, you can always use lazy-loading option which brings you the `Product` object without reference-type properties, and then map them later when you want.

To following is a summary of XOD features:

1. Support all CRUD Operations: Create, Read, Update and Delete
2. Supported data types:
   a. Primitive data types
   b. Value type properties like *struct* and *enum*
   c. String and DateTime objects are treated as value types as well
   d. Reference type objects
   e. Complex types (explained in **[ForeignKey] Attribute** and **Complex Types** sections)
   f. Anonymous reference types … awesome! (explained in **Anonymous (Dynamic) Types** section)
   g. One dimensional arrays of any of the above types
   h. One dimensional generic collection of any of the above types (e.g. List<int>, List<Book>)
2. Queries
3. Self-join
4. Triggers
5. 1-1, 1-Shared, 1-M, and M-M Relationships
6. Cascade Update/Delete
7. Password/encryption security options
8. Primary Keys, Composite Keys
9. Unique-Value and Required validation rules
10. Autonumber/Autogenerate values
11. Encrypted properties
12. Special-character string properties (useful for HTML contents)
13. Excludable properties
14. Eager/Lazy data loading options

2. Opening XOD Database

First we need to add a reference to *AcesDevelopers.Xod.dll* file in the application, then we start by connecting to XOD database. To open a connection to XOD database file, use the following code:

```
XodContext db = new XodContext(@"<database-path>\data.xod");
```

Code #1: Open XOD database

The above code opens a connection to XOD database file or creates new one in the provided path if it was not exist. This new instance of `XodContext class` will be used for all CRUD (create/read/update/delete) operations. There are more options that can be added to this line, like disabling the default create-when-not-exist option or passing the database password if the database is secured. The code below is an expanded version of previous one.

```
XodContext db = new XodContext(
             @"<database-path>\data.xod",
             "<password>",
             new DatabaseOptions()
             {
                 InitialCreate = false,
                 LazyLoad = true
             });
```

Code #2: Open XOD database

Constructor parameters of `XodContext class` are:

- `path`: The path of (.xod) database file
- `password`: XOD database can be encrypted and secured with a password
- `options`: Instance of DatabaseObject object, it includes the following options:
  - `InitialCreate:` Create the database file in it not exist
  - `LazyLoad`: Make lazy-loading as the default procedure when reading objects from the database. Basically lazy-loading means if the object we are retrieving from the database has some reference-type properties XOD will ignore them and brings you the single-value properties, like string, date-time, numbers or enumerations.

3. XoxContext Public Properties

The other public properties of `XodContext` object are:

- `Path`: Gets the current database file path
- `LazyLoad`: Gets or sets the default data loading approach just like *lazyLoad* construction parameter
- `IsNew`: Indicates whether if this database has been created for the first time or not.

4. CRUD Operations

It’s time for you to know how to create, read, update and delete these objects.

4.1. Saving Objects into XOD

Let’s say you have `Employee` `class` among the models of your application, to save a new instance of it into the database; use `Insert()` function of `XodContext` that you have created before:

```
Employee emp = new Employee()
{
    Name = "Employee 1",
    BirthDate = new DateTime(1986, 4, 19),
    BasicSalary = 4000,
    Contacts = new List<Contact>
    {
        new Contact() { Type = ContactType.Email, Value = “employee1@gmail.com” },
    },
    Credentials = new CredentialsDetails()
    {
        UserName = "emp1",
        Password = new System.Text.UTF8Encoding().GetBytes("12345678")
    },
};

db.Insert(emp);
```

Code #3: Saving object into XOD database

Because we are going to use the same `class` more often from this point on, I will put the whole `Employee` `class` and its related classes declaration code below as a reference.

```
public class Employee
{
    [Property(AutoNumber = true)]
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
    public double BasicSalary { get; set; }
    public List<Contact> Contacts { get; set; }
    public CredentialsDetails Credentials { get; set; }

    [ForeignKey("SupervisorId")]
    public Employee Supervisor { get; set; }
    public int SupervisorId { get; set; }
}

public class Contact
{
    [Property(AutoNumber = true)]
    public int Id { get; set; }
    public string Tag { get; set; }
    public ContactType Type { get; set; }
    public string Value { get; set; }
}

public enum ContactType
{
    Phone, Fax, Email
}

public class CredentialsDetails
{
    public string UserName { get; set; }
    public byte[] Password { get; set; }
}
```

Code #4: `Employee` and related classes declaration

Note that `class Employee` has a reference-type property of the same type as itslef. See, `Supervisor` property is of type `Employee`; and this is how XOD does self-join. But there is more to this subject, and that is going to be discussed later in **XOD's Attributes**.

4.2. Reading Objects

After saving some objects into XOD database, you can simply retrieve them back using `Select<T>()` function, and with `Select<T>()` we can do queries as well:

```
Employee emp = db.Select<Employee>().FirstOrDefault(s => s.Id == 13);
if (null != emp)
{
   Console.WriteLine("Employee Details:-");
   Console.WriteLine("Id: {0}", emp.Id);
   Console.WriteLine("Name: {0}", emp.Name);
   Console.WriteLine("Date Of Birth: {0}", emp.BirthDate);
   if (emp.Contacts.Any())
   {
      Console.WriteLine();
      Console.WriteLine("Contacts...");
      foreach (var cnt in emp.Contacts)
         Console.WriteLine("{0}: {1}", cnt.Type, cnt.Value);
   }
}
```

Code #5: Reading objects from database

The default behavior of reading an object will also load all related reference-type properties, in our case (`Contacts`, `Credentials` and `Supervisor`) properties plus their own related reference-type objects down to the bottom of relationship hierarchy. But to layz-load the data, pass `false` to `db.Select<Employee>(false)`, and this will only load the properties (`Id`, `Name`, `BirthDate` and `BasicSalary`).

4.3. Updating an Object

I bet you already got the sense of it… right?! Yes, to update an object use `db.Update<T>(old-object, new-object)` function or just use `db.Update<T>(new-object)` instead, only if the object has primary-key. In this example, first, we will retrieve an object from the database, make some changes to it and then update it:

```
Employee emp = db.Select<Employee>().FirstOrDefault(s => s.Id == 13);
if (emp != null)
{
   emp.BirthDate = new DateTime(1983, 3, 15);
   if (emp.Contacts.Any())
      emp.Contacts[0].Value = "updated address..";
   else
      Console.WriteLine("Employee has no address to update..");

   if (emp.Credentials != null)
      emp.Credentials.Password = new
         System.Text.UTF8Encoding().GetBytes("new-password");
   else
      Console.WriteLine("Can't find login item..");

   db.Update(emp);
}
```

Code #6: Updating objects

From the example above we didn't only modify direct properties of employee object but we also modified some properties in reference value properties like the Password property of `Credentials` property, in this scenario XOD will update the employee object and all its direct and indirect related reference objects, except for some case that we are going to explain later in this document.

Wait a second! How did XOD know that `Id` property is the primary-key, and used it to find and update the `Employee` object? Well, XOD treats properties with the name `Id` as primary-key. But what if I don't want to use `Id` as the primary-key and use `Code` property instead; no problem, you can fix that by explicitly decorate `Code` property with `[Primary]` attribute. More will be discussed in this regard in **XOD's Attributes** section.

`Update()` function has parameter other than the object to be updated. In addition to the object, you can pass an instance of type `UpdateFilter` to set some options about what exactly you want to update, but that is optional. (`..., UpdateFilter filter`) parameter has two properties `Behavior` and `Properties`, in `Properties` property you put an array of the property names you want to exclude in the update process or the ones you want to exclusively include. The property `Behavior` is an enumeration property of type `UpdateFilterBehavior` with two choices; `UpdateFilterBehavior.Target` or `UpdateFilterBehavior.Skip`, and it lest you specify whether the names in `Properties` property are the ones you want to target in the update process or skip. Using filter with `Update()` function is only for efficiency.

4.4. Deleting an Object

Like `Update()` function, in `Delete()` function we simply pass the object we want to delete as parameter:

```
Employee emp = db.Select<Employee>().FirstOrDefault(s => s.Id == 21);
if (emp != null)
   db.Delete(emp);
```

Code #7: Deleting objects

After executing the above code, that **`Employee`** object will be deleted from the database along with all `Contact` reference objects in `Contacts` list property and `Credentials` reference object, all of them will be deleted from the their corresponding files of XOD database. Good… but what about `Supervisor` reference property, obviously supervisors should not be deleted, that's why we added `[ForeignKey]` attribute to `Supervisor` property. Now XOD understands that this Supervisor property is not an ordinary property, but in fact, it is an independent individual that happens to be of the same **`Employee`** `class`, and will not be affected by the delete operation. We will leave it here right now to keep it simple, and we will discuss these options in great details in **XOD's Attributes** section.

4.5. Insert or Update

`InsertOrUpdate()` is an additional function of **`XodContext`**. As its name sounds like, it's simply two operations in one; it starts by checking if the object you are passing as parameter does exist, if it is, then it will update the object, otherwise it will add it to the database as a new one.

How does XOD check the existence of an object? The answer is; by the value of its primary-key-property/composite-key-properties. In other words, this function works only with object with primary-key or composite-key.

Primary-key property is a single property used to uniqley identify the object, while composite-key properties are a set of properties that are used to identify the object.

5. XOD's Attributes

XOD's CRUD operations could behave differently based on the way you designed your model classes. XOD provides a set of attributes that can be used to add more meaning to your model classes.

5.1. [Property] Attribute

When you want to set autonumber feature or cascading options for a property, use `[Property]` attribute.

```
   public class Employee
   {
      [Property(AutoNumber = true, OverrideAutoNumber = true)]
      public int Id { get; set; }
...
```

Code #8: `[Property]` attribute

In the above code, `Id` property of **`Employee`** `class` has been set as auto-number, it means when we create a new instance of **`Employee`** `class` and save it into XOD database there is no need to set a value for `Id` property because XOD will generate it for you automatically.

In the code above we used `[Property]` attribute with two options, (`AutoNumber = true,` …) and (… `OverrideAutoNumber = true`); we already know what does `AutoNumber` do, but setting the second option (`OverrideAutoNumber`) to `true` means you are telling XOD that it is ok to manually pass a value to `Id` property at some cases, and if that happens, then don't generate an auto-number value and use the one provided instead, unless this manually provided value is conflicting with already existed object. `OverrideAutoNumber` option is set to `false` by default.

Note that auto numbering feature works only for numeric and Guid data types.

`Cascade` is another option in [Property] attribute, currently `Cascade` option can only be used for cascade delete functionality (obviously should be added to update operations as will, hopefully soon) and it works for applicable reference-type properties, and I say applicable because reference properties are already set to cascade delete by default like `Contacts` and `Credentials` properties because they are complex-objects (complex-objects will be discussed later in **Complex Types** section).

Another option of `[Property]` attribute is (`Position`) option which can be set to either *ValuePosition.Body* or *ValuePosition.Attribute* and it controls where the value of the property should be stored in the XML format. Personally, I don't think this feature makes significant impact and I might get rid of it in the upcoming releases.

From now on, if I want to refer to a feature that doesn't seem to be very helpful and I plan to deprecate it; I will use the tag **[deprecation-planned]** next to it, with explanation of the replacement feature or behavior.

5.2. [PrimaryKey] Attribute

We already mentioned primary-key and composite-key properties, and how they are used to uniquely identify objects in the database. We also mentioned that properties with the name `Id` are treated as primary-key property implicitly. `[PrimaryKey]` attribute is applicable for value type properties including String and excluding DateTime. And as we mentioned; we can also use `[PrimaryKey]` attribute on multiple properties and that makes it a composite-key.

The following is a list of all data types which are applicable for primary/composite-key properties:

a. Numbers of all types (Integer are recommended) **[deprecation-planned]**: Only Integer, Long Integer and Byte will be supported in upcoming releases, Double and Float will be excluded.
b. Guid: Unlike auto-numbered numeric properties, Guid doesn't provide incremented values, but provides auto-generated values when we apply `AutoNumber` option to it. Also, unlike numeric properties. Also, if we didn't explicitly defined a primary or composite key, and we had a property called `Id` that was not decorated with primary-key nor auto-number options; this `Id` property will automatically considered as primary-key and auto-number, weather it was numeric or Guid.
c. String: String is not compatible with auto-numbering though.

5.3. [ForeignKey] Attribute

Complex property is reference-type (`class` instance) property we declare with no attributes define its relationship to its referee `class` like `[ForeignKey]`, `[ParentKey]` or `[Children]`. It is simply an encapsulated set of properties that are related to each other in a single `class` or `struct` unit. In the other hand, when we add `[ForeignKey]` attribute to it, it becomes something different in terms of behavior. It adds some more features to the property and no longer called complex-type.

As we said complex-type properties are just extensions to the main object and can't live without it. When we delete the object these extensions are no longer needed and will be deleted as well. But there are other scenarios where we want to link other objects of different types to each other while remain totally independent from each other. `Contact` and `Credentials` of `Employee class` are good examples of complex-type properties. But `Supervisor` of `Employee class` is a good example of independent reference-type properties. We decorated it with `[ForeignKey]` attribute, and that makes it independent reference, and that's why when we delete an `Employee` object, the actual `Supervisor` object remains intact in the database.

You see, `[ForeignKey]` attribute introduces a new type of relationship between classes. From now on, we will call it One-To-Shared relationship.

While [ForeignKey] creates One-To-Shared relationship, complex-type creates One-To-One.

```
    public class Employee
    {
...
         [ForeignKey("SupervisorId")]
         public Employee Supervisor { get; set; }
         public int SupervisorId { get; set; }
    }
```

Code #9: [ForeignKey] attribute

In Code#9, we decorated Supervisor property with [ForeingKey("SupervisorId")] attribute and then added another property SupervisorId of type int to hold the actual id number (the primary-key value) of the supervisor, if there is any. And because SupervisorId property name has been assigned to [ForeignKey("SupervisorId")] attribute; both SupervisorId and Supervisor properties will be connected to each other from now on. For instance, if I have created an object from Employee class and only passed the id number to the SupervidorId property, XOD will take care of mapping the actual reference object to Supervisor property automatically and vice versa.

Either we do:

```
Employee emp2 = new Employee()
{
    Name = "Employee 2",
    SupervisorId = 1
};
db.Insert(emp2);
```

Code #10: Passing supervisor instance by its Id

Or:

```
Employee emp2 = new Employee()
{
    Name = "Employee 2",
    Supervisor = emp1
};
db.Insert(emp2);
```

Code #11: Passing supervisor instance by its actual reference

Usually, when we use [ForeignKey] attribute we pass two parameters, one for the locally connected property (in our case SupervisorId) and the other for the remote one in the other end class (in our case Id), just like traditional RDB relationship, but we can ignore the remote property it was the primary-key or its name was Id.

5.4. [Children] & [ParentKey] Attribute

We know One-To-Many or Master-Details relationship in database design; it is used when we have a master object that has a collection of sub-objects that are somehow related to each other through their master object, e.g. customers/customer-orders and order/order-details. XOD supports this kind of relationships by using `[Children]` and `[ParentKey]` attributes.

To create such a relationship between two classes, first we go to the master `class` (e.g. **`Order`** `class`), create a collection property of the child `class` type (e.g. `List<`**`OrderDetails`**`>`), and then decorate this collection property with `[Children]` attribute. After that, we go to the child `class` and add two properties for master object referencing. The following code explains that.

```
   public class Order
   {
       public int Id { get; set; }
...
       [Children]
       public List<OrderDetails> Details { get; set; }
...
```

Code #12: Decorating children property in the master `class`

```
    public class OrderDetails
    {
 ...
        [ParentKey("MasterId")]
        public Order Master { get; set; }
        public int MasterId { get; set; }
 ...
```

Code #13: Decorating parent property in the child `class`

Between Code #12 and Code #13 we created One-To-Many relationship, and so we inherited the following benefits:

1. Automatic cascade delete, it means when the parent object gets deleted all referenced children will be deleted as well, and we don't need to use `[Property(Cascade=CascadeOptions.Delete)]` on `Details` property of **`Order`** `class` to accomplish this.
2. When we try to read a certain **`OrderDetails`** object directly from the database by using `Select<`**`OrderDetails`**`>()` function we will have access to the parent data through `Master` or `MasterId` properties in **`OrderDetails`** `class`.
3. Children objects will be reserved for the parent object. This works like a validation rule, and it means any child of certain **`Order`** can't be shared with another **`Order`** object. Try to do that and you will get `ReservedChildException` exception. More information about XOD's Exceptions is available in **Exceptions** section.

The following is a complete example of how to define one-to-many or master-details relationship in XOD:

```
public class Order
{
    public int Id { get; set; }
    public DateTime Date { get; set; }

    [Children]
    public List<OrderDetails> Details { get; set; }

    [ParentKey("CustomerId")]
    public Customer Customer { get; set; }
    public int CustomerId { get; set; }
}

public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string Phone { get; set; }

    [Children]
    public List<Order> Orders { get; set; }
}

public class OrderDetails
{
    public long Id { get; set; }

    [ForeignKey("ItemId")]
    public ProductItem Item { get; set; }
    public Guid ItemId { get; set; }
    public int Quantity { get; set; }

    [ParentKey("MasterId")]
    public Order Master { get; set; }
    public int MasterId { get; set; }
}

public class ProductItem
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public double Price { get; set; }
}
```

Code #14: One-To-Many Relationships

```
ProductItem item1 = new ProductItem()
{
    Name = "Product 1",
    Price = 15.5
};
db.Insert(item);

Customer customer1 = new Customer()
{
    Name = "Sallam",
    Orders = new List<Order>
    {
        new Order()
        {
            Date = DateTime.Now,
            Details = new List<OrderDetails>
            {
                new OrderDetails() { Quantity = 2, Item = item1 }
            }
        }
    }
};

db.Insert(customer1);
```

Code #15: saving master-details objects into XOD

In Code#14 we defined two-levels master-details relationship between **`Customer`**, **`Order`** and **`OrderDetails`** classes, and one shared-reference relationship (using `[ForeignKey]` attribute) between **`OrderDetails`** and **`ProductItem`** classes. But In Code#15 we initialized a set of related objects and saved them into XOD database.

We can learn more than one thing from Code#15, for instance, we created a new customer with a new complete order (two levels of master-details relationship) in one-go and saved all related objects with just one `db.Insert()` call. I also intentionally created a product item and saved it separately then mapped it to the customer order we created, just to show you that you can do that.

A more realistic example would be by creating and saving both product-item and customer objects separately, and when comes the time where we have an order, we create a new object from **`Order`** `class` and assign both OrderDetails collection and Customer properties, then save this newly created **`Order`** object into the database. For clarification, check the following code:

```
    ProductItem item1 = new ProductItem()
    {
        Name = "Product 1",
        Price = 15.5
    };
    db.Insert(item1);

    Customer customer1 = new Customer() { Name = "Sallam" };
    db.Insert(customer1);

...

    Order order = new Order()
    {
        Date = DateTime.Now,
        Details = new List<OrderDetails>
        {
            new OrderDetails() { Quantity = 2, Item = item1 }
        },
        CustomerId = customer1.Id
    };
    db.Insert(order);
```

Code #16: Mapping existed reference objects when saving new objects

In the code above, we mapped the customer to this new order by assigning actual customer id to `CustomerId` property. And this is enough for XOD to bring the actual reference object to `Customer` property. But the opposite is also true, we also can assign the whole object customer to `Customer` and then XOD will fill `CustomerId` property for you.

5.5. [Required] Attribute

`[Required]` attribute is a validation role and can be applied to any property of any data type. Filling those required properties with values other than (null) or default value will be mandatory and failing to fulfill this will raise `RequiredPropertyException` exception.

5.6. [NotMapped] Attribute

Properties that are marked as `[NotMapped]` will simply be ignored and will not be persisted in the database.

5.7. [Markup] Attribute

Because XOD stores the data as XML, assigning XML-like contents such as (HTML) in string properties could cause a problem. But we can prevent that from happening if we simply decorated these string properties with `[Markup]` attribute. When we do that, the content of these string properties will not be parsed by the XML parser avoiding potential special- characters parsing problems.

5.8. [Crypto] Attribute

Using `[Crypto]` attribute with string property encrypts its value in XOD database file. So even if the database was not secured and entirly encrypted, using `[Crypto]` attribute will grant encryption in those parts of your choice.

You can specify one of two available encryption methods, `CryptoMethod.SHA1` or `CryptoMethod.MD5`, when using `[Crypto]` attribute without parameters, XOD selects `CryptoMethod.MD5` by default.

6. Queries

Search for single or a set of objects is quite simple with XOD. XOD for .NET actually uses LINQ query expressions (lambda functions) to handle this tasks, and we saw that in some of our examples.

To make a query we start with `db.Select<T>()` function, and because `db.Select<T>()` returns **`IEnumerable<T>`**, we can use functions like `FirstOrDefault()`, `Where()`, `Select()` or a combination of them, to search for the desired objects.

```
    var emp = db.Select<Employee>().FirstOrDefault(s =>
        s.Name.StartsWith("Adel")
        && s.Name.EndsWith("Hassan"));
...
    var emps = db.Select<Employee>().Where(s => s.BasicSalary > 3000);
...
    var names = db.Select<Employee>()
        .Where(s => s.BasicSalary > 3000)
        .Select(s => s.Name);
```

Code #17: Query Examples

The examples in Code #19 shows different ways of writing queries. The first statement returns the first one object which its name starts with “Aden” and ends with “Hassan”. The second one returns a collection of employees, whom their basic salary exceeds $3000. The last one is similar to the second one except it will not return a collection of Employee objects; instead it will return a collection of string representing employee names only.

7. Complex Type Properties

We already talked about complex-types, and how they simply are an extension of the main `class` data structure (check the beginning of **[ForeignKey] Attribute** section for details). There are some similarities between `[ForeignKey]` and complex-type properties, but they are totally different and used for different reasons. To add some more clarification, I put the following comparison table:

	`[ForeignKey]` Properties	**Complex-Type Properties**
Value	Class Instance	Class or Struct Instance
Relationship Type	One-To-Shared	One-To-One
Requirement	PrimaryKey is required in the instance `class`	PrimaryKey is not required in the instance `class`
Cascade Delete/Update	Not by Default: e.g. delete the referee object will not affect the `[ForeignKey]` actual object	Yes (Implicitly): e.g. delete the referee object will delete complex-type actual object
When to Use	With independent entities: e.g. Student is an independent entity that can be related to Enrolment, Class and BorrowedBook objects.	With dependent data set: e.g. Contact detail of an Employee is totally dependent on the existence of that Employee.

Table#1: Comparision between [ForeignKey] Property and Complex-type property

8. Anonymous (Dynamic) Types

Anonymous, dynamic or generic are all synonyms for the same thing, which is "unknown object type at design time" (time of writing `class` code). This is a great feature of XOD, you can simply define a property of type `object`, and at runtime, assign whatever object you want to it. Awesome, right! But there is one thing you have to do first, use `RegisterType<T>()` function to register all possible types XOD may work with for these anonymous properties. It is recommended to write the code of registering anonymous type right after initializing database connection.

Usually when we insert or update an object, XOD investigates all properties of that object and build a small database with information of all object types it finds, but when it come across objects merely defined as `object`, XOD will fail in expecting the exact type this property may hold in the future, because it is simply unknown. That's why we need to pre-register these types that are potentially will be assigned to this anonymous property. The code below explains it all.

```
    //Model Classes
    public class Bag
    {
        public int Id { get; set; }
        public string Description { get; set; }
        //Content property designed to have
        //either Papers, Food or Toys instance
        public object Content { get; set; }
    }

    public class Papers { }
    public class Food { }
    public class Toys { }
...
    //Connecting to database
    XodContext db = new XodContext(path);
    //Registering anonymous types
    db.RegisterType<Papers>();
    db.RegisterType<Food>();
    db.RegisterType<Toys>();
...
    //Insert a new instance of Bag object with Food as content
    db.Insert(new Bag()
    {
        Description = "Today, I have food in my bag.",
        Content = new Food()
    });
...
    //Insert a new instance of Bag object with Toys as content
    db.Insert(new Bag()
    {
        Description = "Today, I have toys in my bag.",
        Content = new Toys()
    });
```

Code #20: Designing and registering anonymous types

9. Triggers

Triggers are your way of interfering with usual `Insert()`, `Update()`, `Delete()`, `Drop()` and `DropAll()` operations. XOD provides two events `BeforeAction` and `AfterAction`, for example, if you want create `Log` item whenever your application creates a new `Employee`, you can add an handler for `AfterAction` even, and inside that handler, catch `Insert` actions only, and test them whether the inserted object's type equals to `Employee`, and then use these information to create a new `Log` item.

```
    //Creating trigger handler
    EventHandler<TriggerEventArgs> after = (s, e) =>
    {
        if (e.Action == DatabaseActions.Insert && e.Type == typeof(Employee))
        {
            //code for creating new log item
        }
    };
    db.AfterAction += after;
...
    db.Insert(emp1);
...
    //Deactivating AfterAction triggers
    db.AfterAction -= after;
```

Code #18: Insert action trigger

In the code above, we created an event handler of type `EventHandler<TriggerEventArgs>` and then assign it to `AfterAction` trigger, now, whenever a new employee gets inserted into the database this trigger will be executed and it will continue to be executed until we deactivate the trigger somewhere in the application code using (`db.AfterAction -= after`).

The parameter `e` that comes with trigger event handler holds some useful data. You can use it to access to the actual (inserted, updated, etc.) object through `e.Item` property, its data type through `e.Type` property and trigger action type through `e.Action` property. There is also `e.Cancel` option, which is applicable with `BeforeAction` triggers only, and allows you cancel the whole operation, for instance, if it did not meet certain conditions.

10. Dropping Types

Besides CRUD functions, there are `Drop()` and `DropAll()` functions, and as the name refer to, `Drop()` function drops all objects of specific type, it's like deleting a table in RDB. All references and tracks of objects of the same type will be removed from other object of different type in the database. In the other hand, `DropAll()` terminates the whole database objects.

It's important to know that calling `Drop()` or `DropAll()` function will not take effect unless you use `BeforeAction` trigger to confirm the operation, because when it comes to drop operations `e.Cancel` of `BeforeAction` trigger is set to `true` by default and cancels the operation automatically. That's why you need to activate BeforeAction trigger, and confirm the drop by setting (`e.Cancel = false`) explicitly. Check the following code for clarification:

```
//Activate trigger
EventHandler<TriggerEventArgs> before = (s, e) =>
{
   if (e.Action == DatabaseActions.Drop && e.Type == typeof(Contact))
      e.Cancel = false;
};
db.BeforeAction += before;
db.Drop<Contact>();
//Deactivate trigger
db.BeforeAction -= before;
```

Code #19: Confirming drop operations for **`Contact`** objects

11. Security

Protecting databases is important, and XOD offers simple -as the nature of the database- yet strong security method, which is the Password. When you secure the XOD with a password, all database backend XML files get encrypted, and no longer readable but by XOD engine and the password you protected the database with.

To secure XOD database, use `Secure("<password>")` function and pass the new password, and if you want to change the password call `ChangePassword("<current-password>", "<new-password>")`. But if you want to remove the database protection, call `Loose("<password>")` function and pass the password.

12. Exceptions

XOD's CRUD operations could raise some exceptions based on the validity of your inputs. The following list shows these exceptions and the reason they might be raised for:

- `ArgumentNullException`
  When you pass null values to `Insert()`, `Update()`, `InsertOrUpdate()` or `Delete()` functions.

- `MissingPrimaryKeyValueException`
  If you passed an object designed to have primary-key/composite-key to either of `Insert()`, `Update()` or `InserOrUpdate()` functions, and you forgot to set values to these primary-key/composite-key properties, XOD will alert you with `MissingPrimaryKeyValueException` exception.

- `RequiredPropertyException`
  Calling `Insert()` or `Update()` function could raise this exception if the object you inserting or updating has properties decorated with `[required]` attribute, but left with null or the default values.

- `PrimaryKeyDataTypeException`
  Primary-key and composite-key properties should be value type properties including String. An attempt to use reference type properties as primary-key will raise `PrimaryKeyDataTypeException` exception.

- `AutoNumberDataTypeException`
  Properties with `AutoNumber` option of `[Property]` attribute should be numeric or Guid. An attempt to use any other datatype for this purpose will raise `AutoNumberDataTypeException` exception.

- `ReservedPrimaryKeyException`
  When you don't use `AutoNumber` option of `[Property]` attribute with primary-key (or if you use it with `OverrideAutoNumber` option) there will be a chance to set a value in the primary-key that is already reserved for another persisted object.

- `ReservedUniqueKeyException`
  This exception is similer to `ReservedPrimaryKeyException`, this one is testing the values against `[Required]` properties.

- `MissingParentKeyException`
  `[Children]` attribute works only when we add `[ParentKey]` decorated property in child `class`, missing `[ParentKey]` decoration will raise `MissingParentKeyException` exception.

- `PropertyKeyNameException`
  Declaring [ForeignKey] or [ParentKey] attribute needs some property names references. *PropertyKeyNameException* exception will be raised if you provided wrong property names in one of those attributes.

- `ReservedChildException`
  An attempt to set a child object that has been reserved for a specific parent object to another parent object will raise `ReservedChildException` exception. The only way to get around this is by assigning the new parent object to the child object's parent property, not the opposite.

- `ReservedKeyWordException`
  Currently user can't use the (refType, collType, hostProp or dataType) keywords as primary-key or composite-key property names because they are part of XOD's dedicated XML attributes. Also, avoid using them with any other properties where `Position` option of `[Property]` attribute is set to `ValuePosition.Attribute`.

- `AnonymousTypeException`
  In **Anonymous (Dynamic) Types** section, we mentioned how you need to register these anonymous types after establishing a connection to XOD database. If it happens that XOD came across an anonymous object that has not been registered, a `AnonymousTypeException` exception. Will be raised.

- `SecurityException`
  When the database is protected, and you tried to open it with the wrong password, XOD will raise SecurityException exception.

- `DatabaseFileException`
  It simply means, you passed a wrong database path.

Now, that you have learned about XOD and how to use it, I strongly encarage you to check it out and actually use it. Don’t forget to send me your feedback.

My contact information have been listed in **About the Author** area, at the very beginning of this guide.

Thank You,