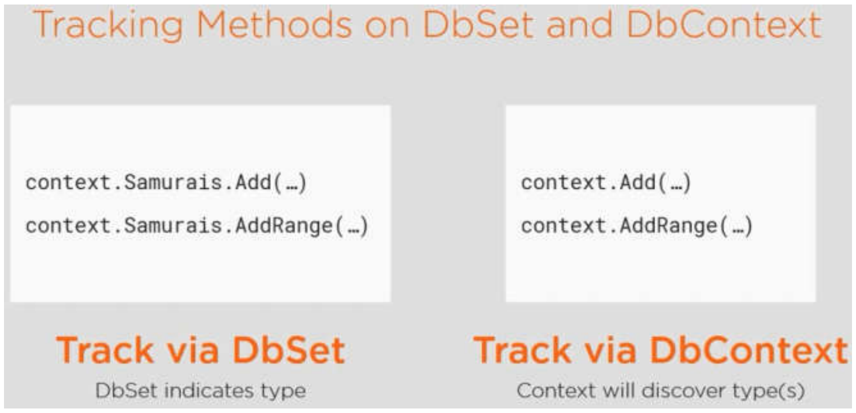
Modifying data via the DbContext

DbContext SaveChanges always starts with transaction and will rollback if any transaction failed if not it will commit this is default behavior



Tracking Methods on DbSet and DbContext

```
context.Samurais.Add(…)
context.Samurais.AddRange(…)
```

Track via DbSet

DbSet indicates type

```
context.Add(…)
context.AddRange(…)
```

Track via DbContext

Context will discover type(s)

The approach that you adopt to modifying entities depends on whether the context is currently tracking the entity being modified or not.

In the following example, the entity is obtained by the context, so the context begins tracking it immediately. When you alter property values on a tracked entity, the context changes the `EntityState` for the entity to `Modified` and the ChangeTracker records the old property values and the new property values. When `SaveChanges` is called, an `UPDATE` statement is generated and executed by the database.

```
var author = context.Authors.First(a => a.AuthorId == 1);
author.FirstName = "Bill";
context.SaveChanges();
```

Since the ChangeTracker tracks which properties have been modified, the context will issue a SQL statement that updates only those properties that were changed:

```
exec sp_executesql N'SET NOCOUNT ON;
UPDATE [Authors] SET [FirstName] = @p0
WHERE [AuthorId] = @p1;
SELECT @@ROWCOUNT;
',N'@p1 int,@p0 nvarchar(4000)',@p1=1,@p0=N'Bill'
```

Disconnected Scenario link

In a disconnected scenario such as an ASP.NET application, changes to an existing entity's property values can take place in a controller or service method, well away from the context. In these cases, the context needs to be informed that the entity is in a modified state. This can be achieved in several ways: setting the `EntityState` for the entity explicitly; using the `DbContext.Update` method (which is new in EF Core); using the `DbContext.Attach` method and then "walking the object graph" to set the state of individual properties within the graph explicitly.

Setting EntityState link

You can set the `EntityState` of an entity via the `EntityEntry.State` property, which is made available by the `DbContext.Entry` method.

```
public void Save(Author author)
{
    context.Entry(author).State = EntityState.Modified;
    context.SaveChanges();
}
```

This approach will result in just the author entity being assigned the `Modified` state. Any related objects will not be tracked. Since the ChangeTracker is unaware of which properties were modified, the context will issue an SQL statement updating *all* property values (apart from the primary key value).

DbContext Update link

The `DbContext` class provides `Update` and `UpdateRange` methods for working with individual or multiple entities.

```
public void Save(Author author)
{
    context.Update(author);
    context.SaveChanges();
}
```

As with setting the entity's `State`, this method results in the entity being tracked by the context as `Modified`. Once again, the context doesn't have any way of identifying which property values have been changed, and will generate SQL to update all properties. Where this method differs from explicitly setting the `State` property, is in the fact that the context will begin tracking any related entities (such as a collection of books in this example) in the `Modified` state, resulting in UPDATE statements being generated for each of them. If the related entity doesn't have a key value assigned, it will be marked as Added, and an INSERT statement will be generated.

Attach link

When you use the Attach method on an entity, it's state will be set to Unchanged, which will result in no database commands being generated at all. All other reachable entities with key values defined will also be set to Unchanged. Those without key values will be marked as Added. However, now that the entity is being tracked by the context, you can inform the context which properties were modified so that the correct SQL to update just those values is generated:

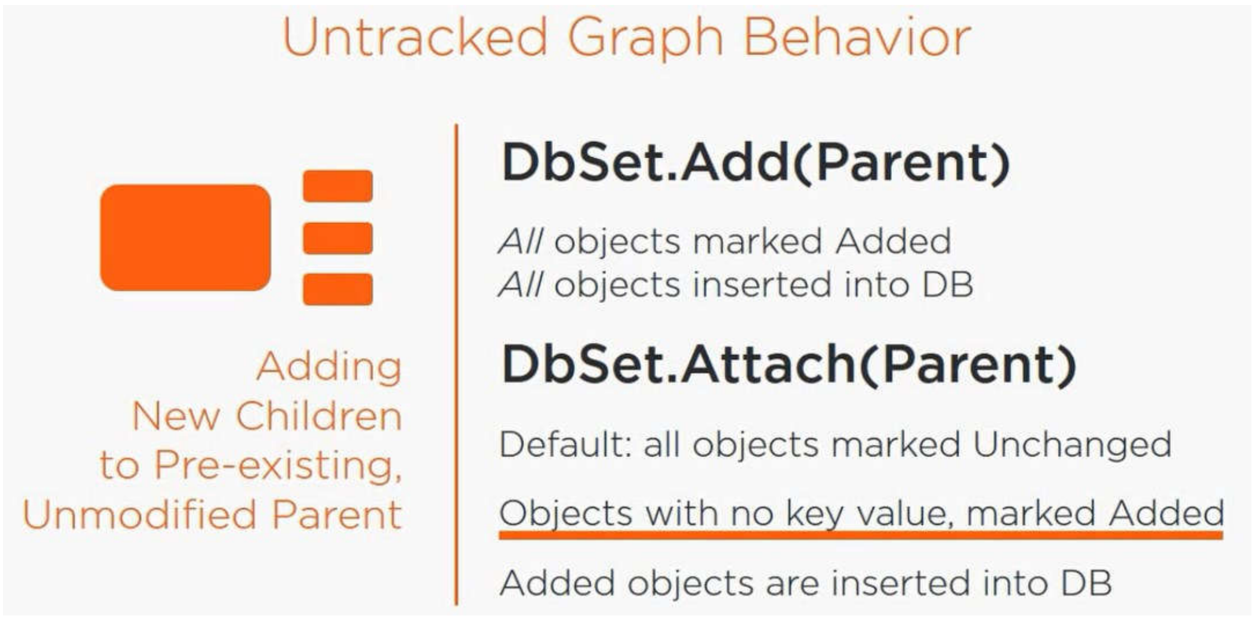
```
var context = new TestContext();
var author = new Author {
    AuthorId = 1,
    FirstName = "William",
    LastName = "Shakespeare"
};
author.Books.Add(new Book {BookId = 1, Title = "Othello" });

context.Attach(author);
context.Entry(author).Property("FirstName").IsModified = true;
context.SaveChanges();
```

The code above will result in the author entity being marked as Modified, and SQL being generated to update just the FirstName property:

```
exec sp_executesql N'SET NOCOUNT ON;
UPDATE [Authors] SET [FirstName] = @p0
WHERE [AuthorId] = @p1;
SELECT @@ROWCOUNT;
',N'@p1 int,@p0 nvarchar(4000)',@p1=1,@p0=N'William'
```

Untracked



Untracked Graph Behavior

Adding
New Children
to Pre-existing,
Unmodified Parent

DbSet.Add(Parent)

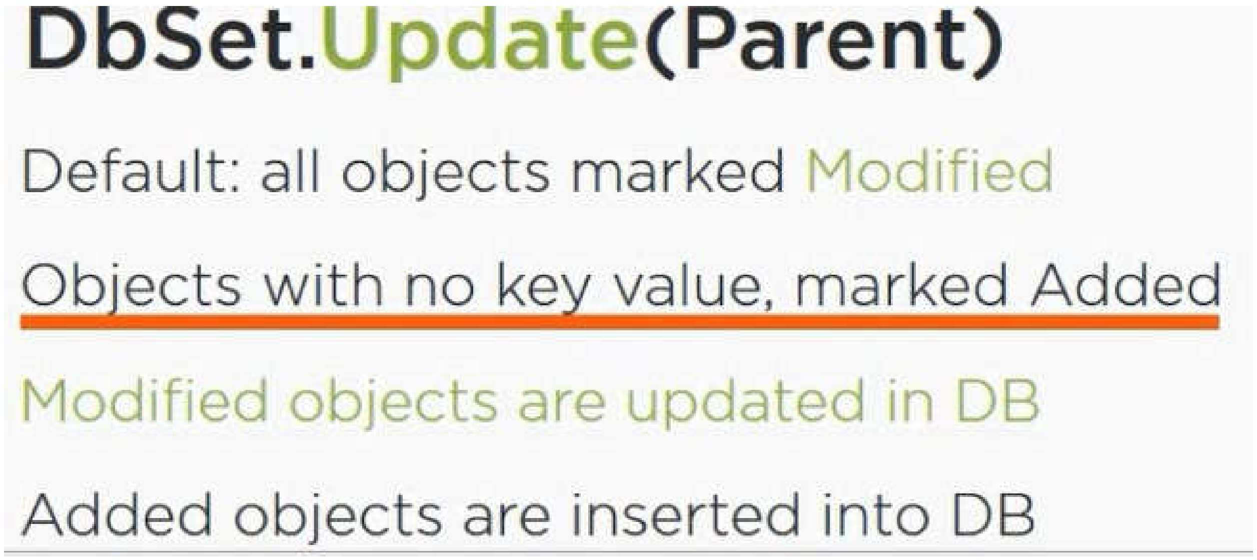
All objects marked Added
All objects inserted into DB

DbSet.Attach(Parent)

Default: all objects marked Unchanged

Objects with no key value, marked Added

Added objects are inserted into DB



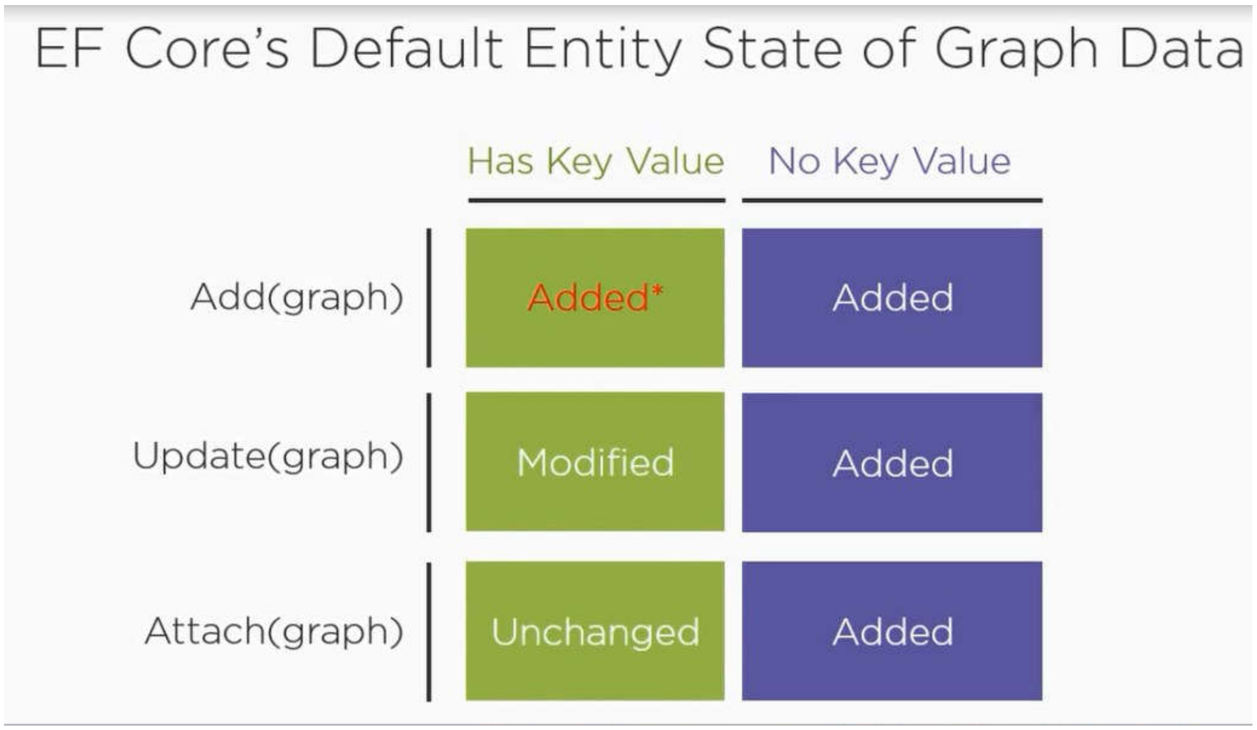
DbSet.Update(Parent)

Default: all objects marked Modified

Objects with no key value, marked Added

Modified objects are updated in DB

Added objects are inserted into DB

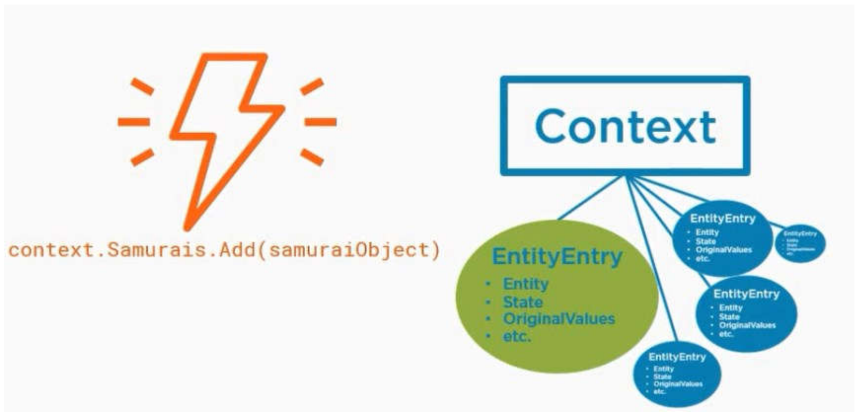


EF Core's Default Entity State of Graph Data

	Has Key Value	No Key Value
Add(graph)	Added*	Added
Update(graph)	Modified	Added
Attach(graph)	Unchanged	Added

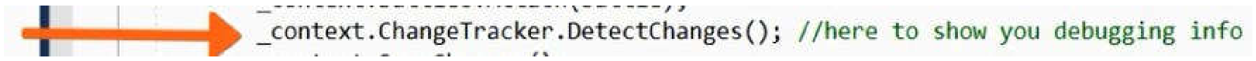
Tracked

Update samo smenetoto property ke se update



Tracking is expensive

```
   ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
//will not track
            ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.TrackAll;
//default behavior
```



```
_context.ChangeTracker.DetectChanges(); //here to show you debugging info
```

call it just before savechanges

and wirite `appDbContext.ChangeTracker.Entries()`

```
to add item to watch in Watch1
```