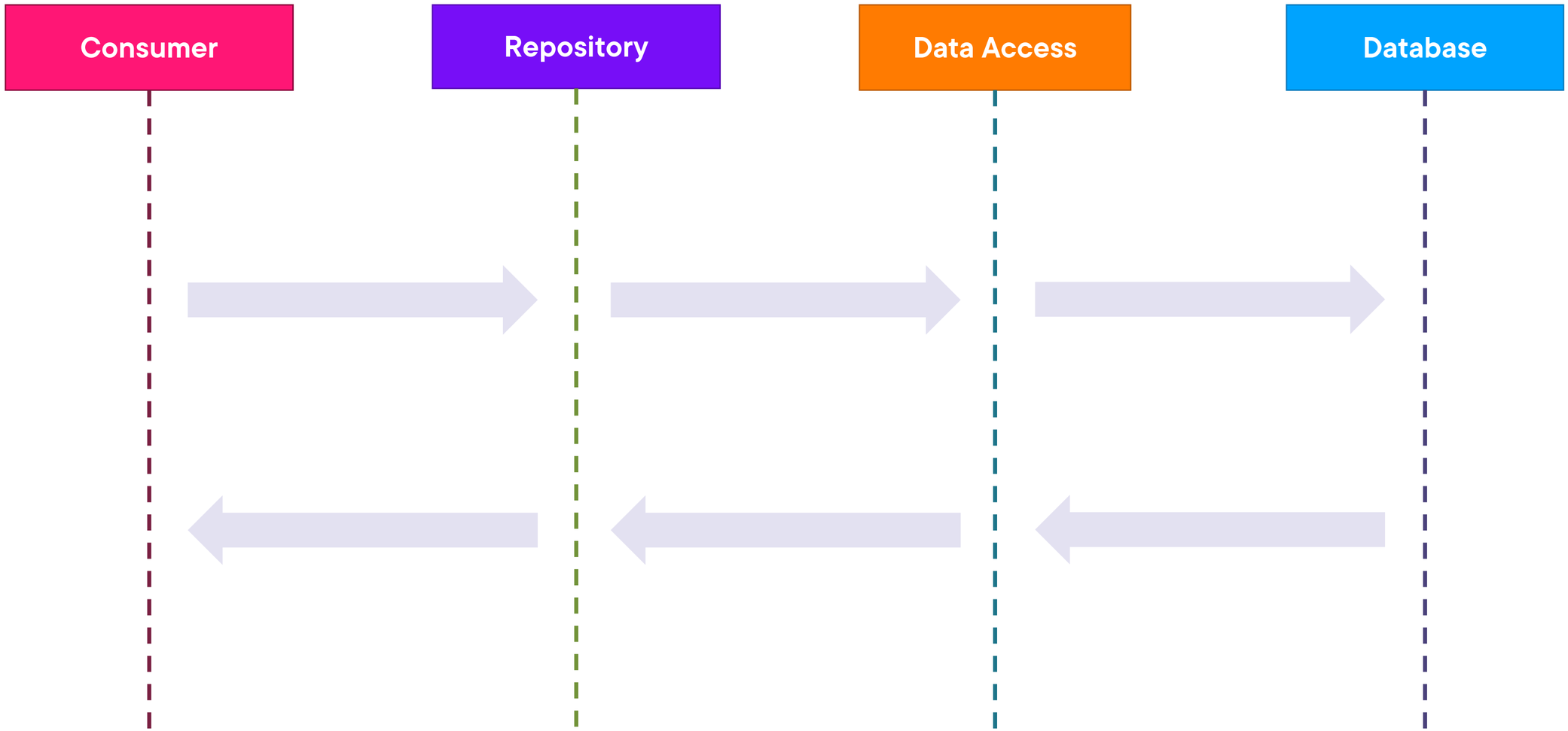
Introducing Repositories and a Data Access Layer

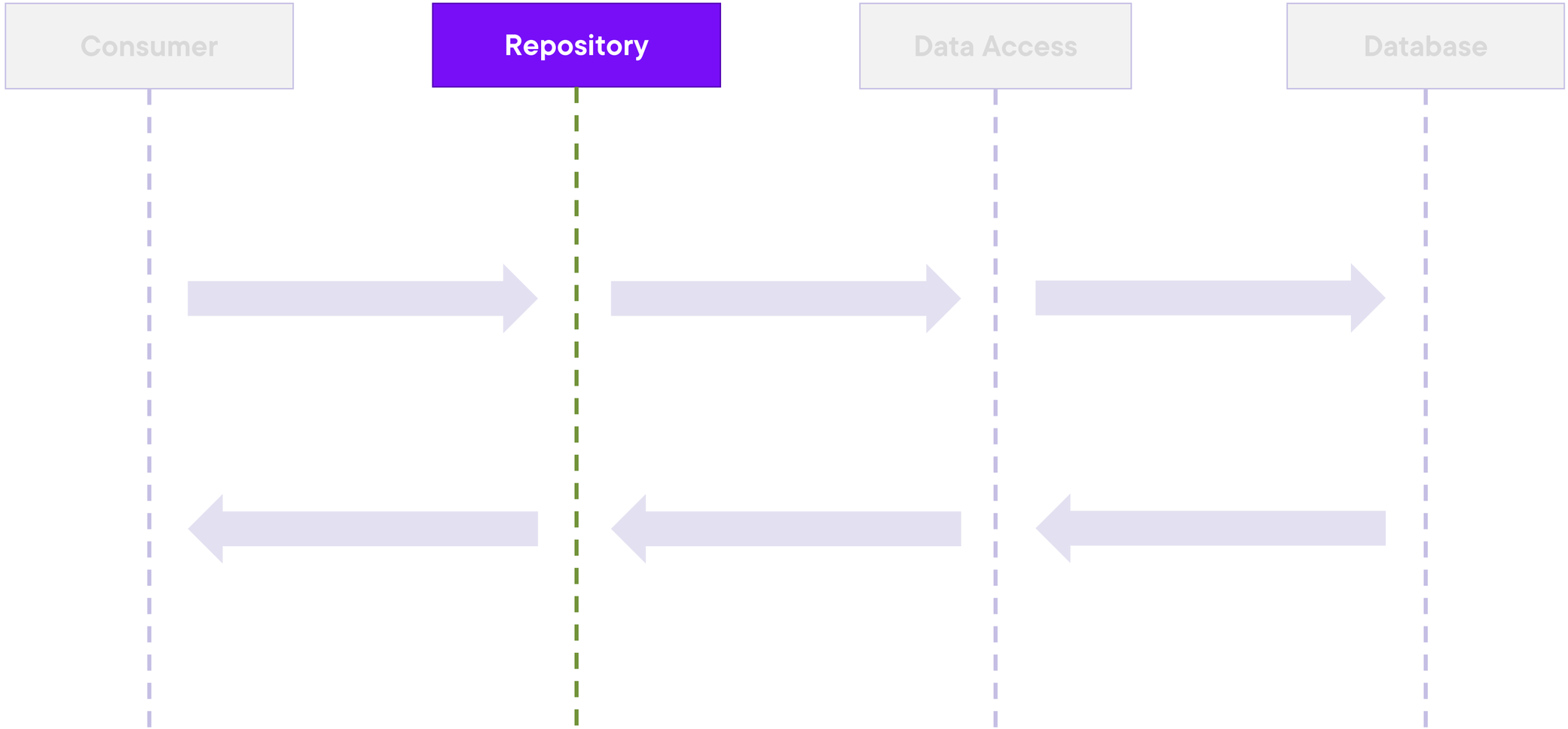
Filip Ekberg
Principal Consultant & CEO

@fekberg | fekberg.com

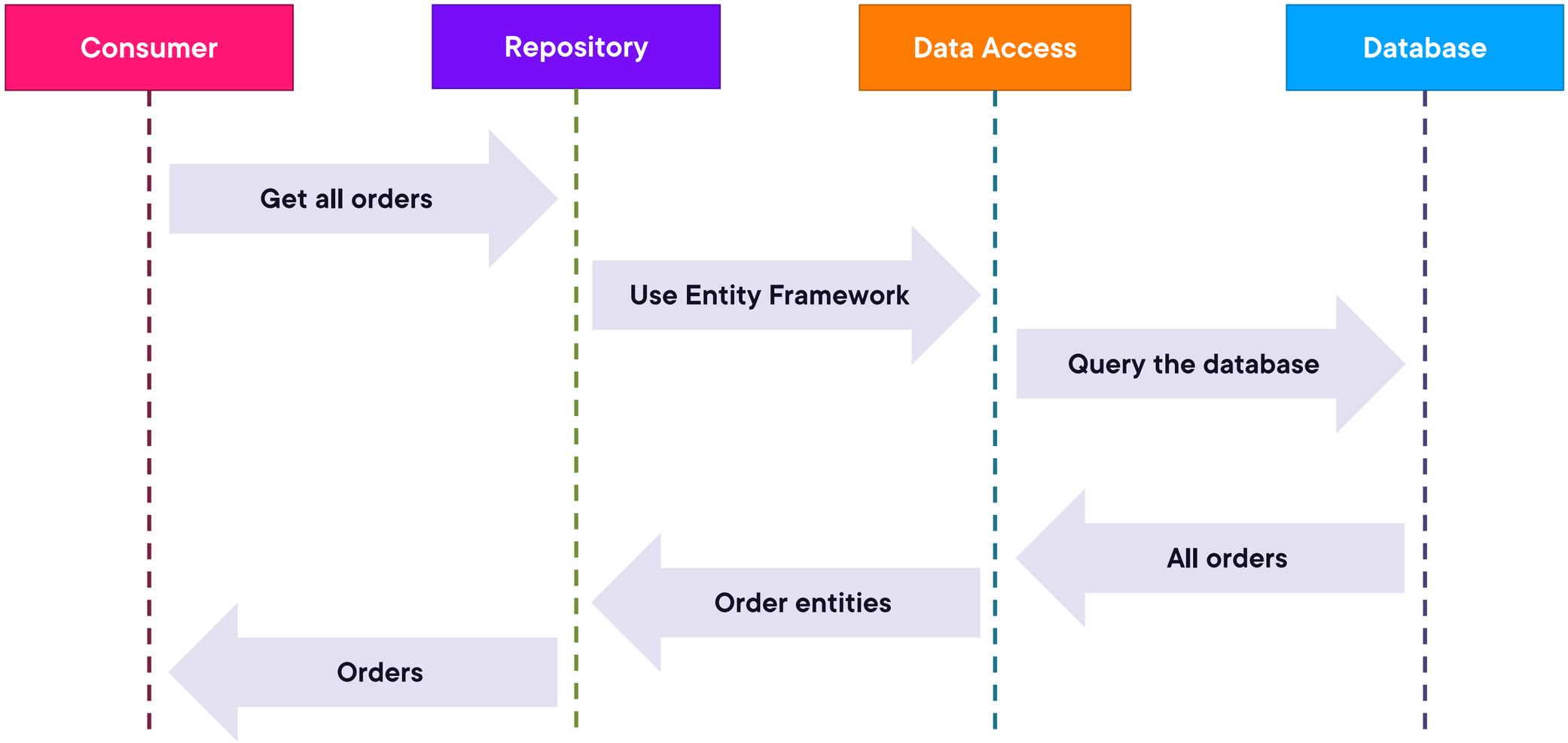
The Repository Pattern



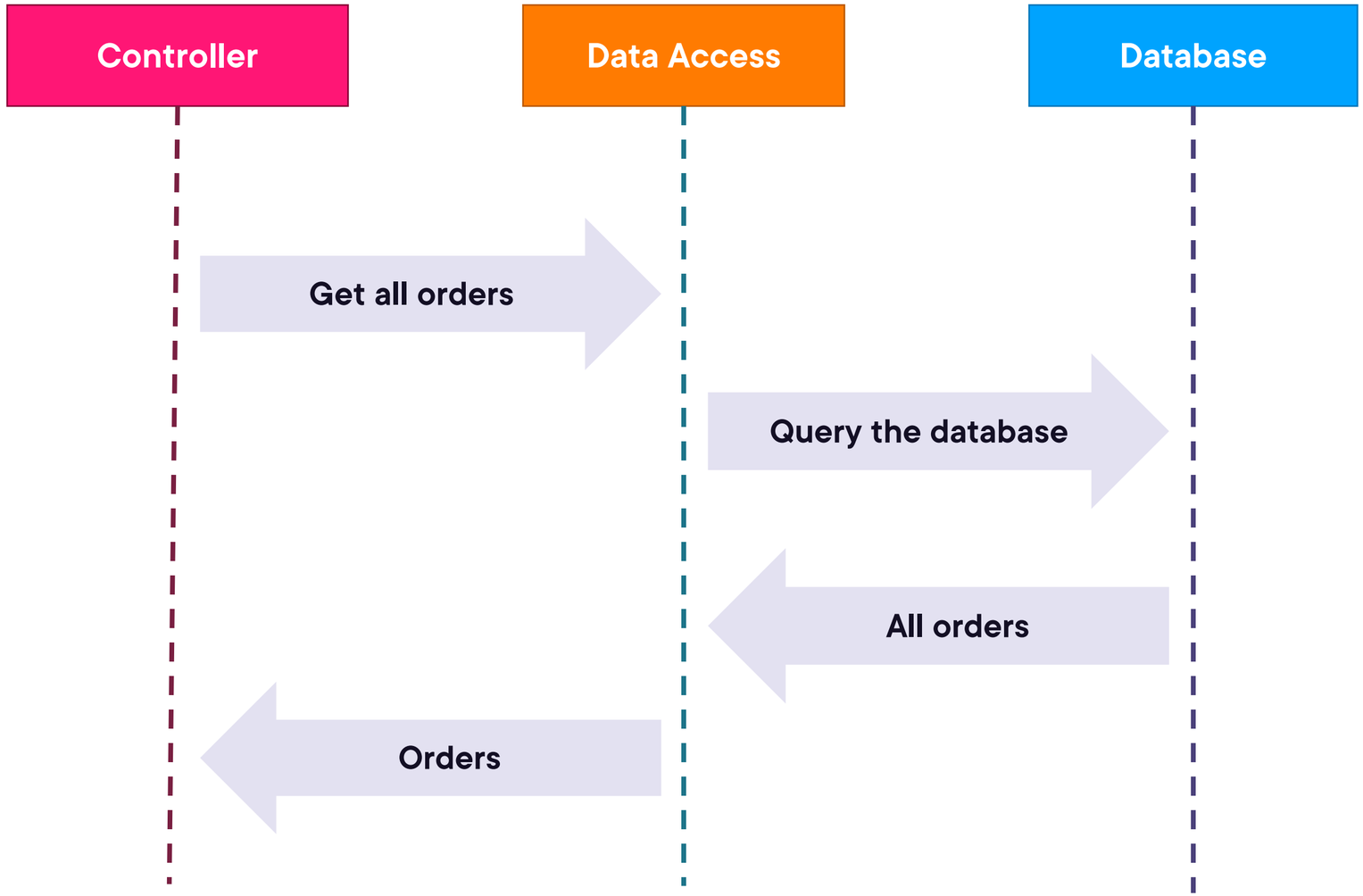
The Repository Pattern



The Repository Pattern



Without a Repository Pattern



The **data access** patterns can be **applied** in **any type** of **application**

Why This Design Is Problematic

The controller is tightly coupled with the data access layer

It is difficult to write a test for the controller without side effects

Hard to extend entities with domain specific behaviour

Avoid tightly coupling the data access in the application layer

An **abstraction** that **encapsulates data access**, making your code **testable**, **reusable** as well as **maintainable**

Benefits of the Repository Pattern

- The consumer (controller) is now separated (decoupled) from the data access
- Easy to write a test without side-effects
- Modify and extend entities before they are passed on to the consumer
- A sharable abstraction resulting in less duplication of code
- Improved maintainability

The **Infrastructure project** could be **referenced** from **many** different **applications**

IRepository<T>

```
public interface IRepository<T>
{



}
```

IRepository<T>

```
public interface IRepository<T>
{
    T Add(T entity);




}
```

IRepository<T>

```
public interface IRepository<T>
{
    T Add(T entity);
    T Update(T entity);



}
```

IRepository<T>

```
public interface IRepository<T>
{
    T Add(T entity);
    T Update(T entity);
    T Get(Guid id);



}
```

IRepository<T>

```
public interface IRepository<T>
{
    T Add(T entity);
    T Update(T entity);
    T Get(Guid id);
    IEnumerable<T> All();


}
```

IRepository<T>

```
public interface IRepository<T>
{
    T Add(T entity);
    T Update(T entity);
    T Get(Guid id);
    IEnumerable<T> All();
    IEnumerable<T> Find(Expression<Func<T, bool>> predicate);

}
```

IRepository<T>

```
public interface IRepository<T>
{
    T Add(T entity);
    T Update(T entity);
    T Get(Guid id);
    IEnumerable<T> All();
    IEnumerable<T> Find(Expression<Func<T, bool>> predicate);
    void SaveChanges();
}
```

Using the IRepository<T>

```
IRepository<Order> orderRepository = ...

Order yourOrder = orderRepository
                .Find(order => order.Customer.Name == "Filip");
```

A **consumer** of this **respository won't** have to **care** about **where** the **data** is **coming from**!

Whoever calls `All()` won't know its side effects!

Using the GenericRepository<T>

```
public class OrderRepository : GenericRepository<Order>
{
    public OrderRepository(WarehouseContext context)
        : base(context)
    {
    }
}

IRepository<Order> repository = new
OrderRepository(context);

var orders = repository.All();
```

GenericRepository<T>

```
public abstract class GenericRepository<T>
    : IRepository<T> where T : class
{
    protected WarehouseContext context;

    ...

    public virtual T Add(T entity)
    {
        var addedEntity = context.Add(entity).Entity;

        return addedEntity;
    }

    ...
}
```

The **consumer** is now **decoupled** from the data access layer

Passing a Fake Implementation

```
class FakeOrderRepository : IRepository<Order>
{
    public Order Get(Guid id) => return new Order { };

    ...
}

var controller = new OrderController(
    new FakeOrderRepository(), ...
);
```

Dependency Injection

“A form of inversion of control, dependency injection aims to separate the concerns of constructing objects and using them, leading to loosely coupled programs”

Example:
The program automatically creates an instance of the `OrderRepository` when it sees the `IRepository<Order>` in the constructor.

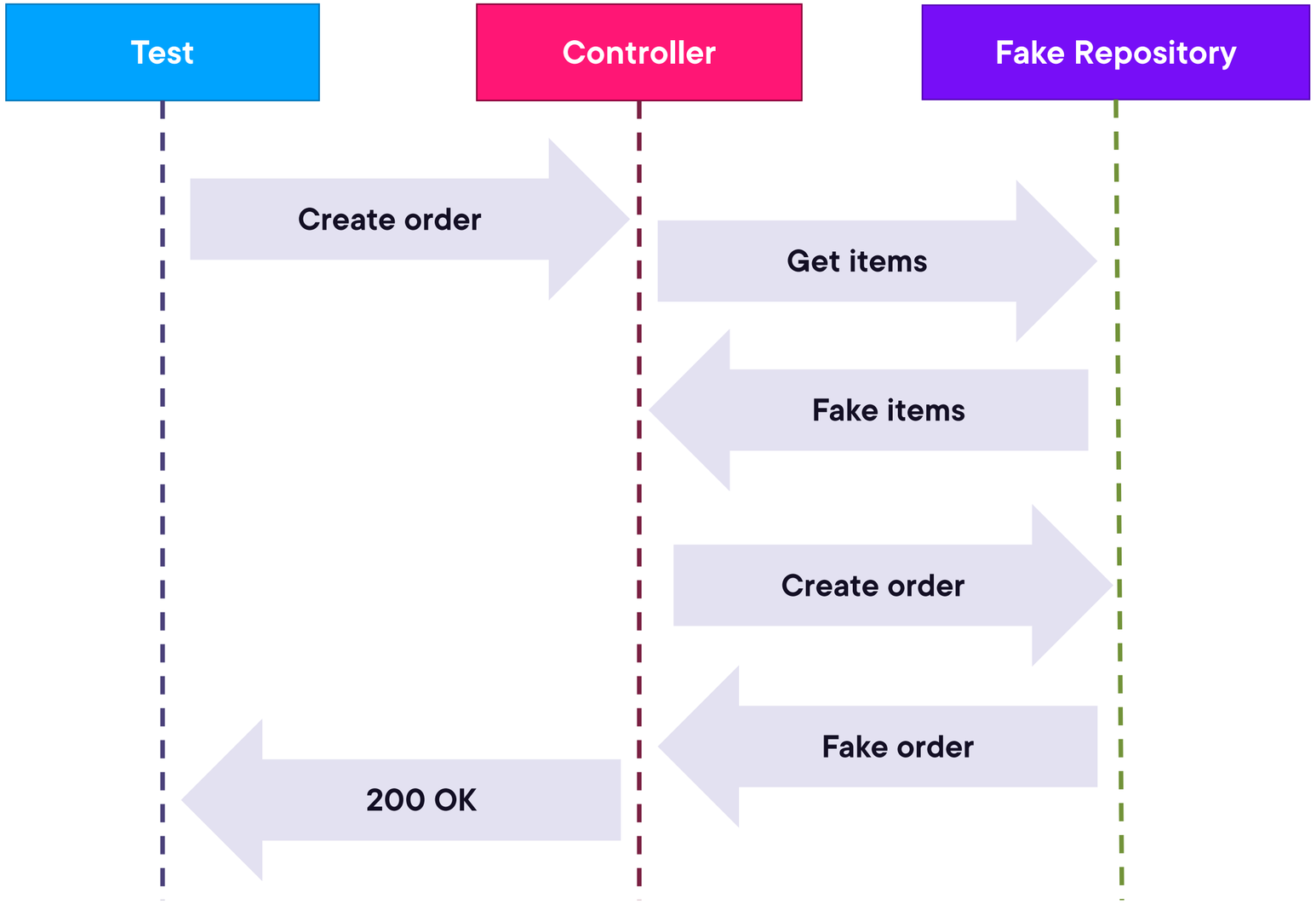
en.wikipedia.org/wiki/Dependency_injection

Data access is decoupled!

OrderController doesn't know about the **WarehouseContext**!

The **Infrastructure** project can be **reused** throughout the solution!

Testing with the Repository Pattern



Changing the Implementation

Fake

Create a class that implements the methods on the interface with test specific logic

Mocking Framework

Use a library to automatically represent any interface. Setup custom behavior and assertions.

Example of Using Moq

```
var shippingProviderRepository =
    new Mock<IRepository<ShippingProvider>>();
```

Example of Using Moq

```
var shippingProviderRepository =
    new Mock<IRepository<ShippingProvider>>();

// When All() is called on the repository
// return a list with a new provider
shippingProviderRepository.Setup(
    repository => repository.All()
).Returns(new[] { new ShippingProvider() });
```

Example of Using Moq

```
var shippingProviderRepository =
    new Mock<IRepository<ShippingProvider>>();

// When All() is called on the repository
// return a list with a new provider
shippingProviderRepository.Setup(
    repository => repository.All()
).Returns(new[] { new ShippingProvider() });

// Verify that All was called at most once
shippingProviderRepository.Verify(
    repository => repository.All(),
    Times.AtMostOnce()
);
```

Up Next:

Unit of Work Pattern in C#

Unit of Work Characteristics

Commit changes in one transaction to reduce number of interactions

A class with references to multiple repositories used in a single unit

Entity Framework Core applies the **repository** and **unit of work** patterns!

Unit of Work

```
class UnitOfWork
{
    private WarehouseContext context;

    public UnitOfWork(WarehouseContext context)
    {
        this.context = context;

        CustomerRepository          = new CustomerRepository(context);
        OrderRepository             = new OrderRepository(context);
        ItemRepository              = new ItemRepository(context);
        ShippingProviderRepository  = new ShippingProviderRepository(context);
    }

    IRepository<Customer>           CustomerRepository { get; }
    IRepository<Order>              OrderRepository { get; }
    IRepository<Item>               ItemRepository { get; }
    IRepository<ShippingProvider>   ShippingProviderRepository { get; }

    void SaveChanges() => context.SaveChanges();
}
```

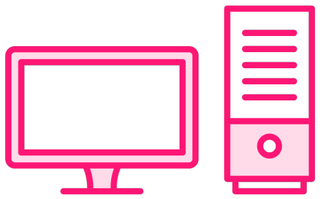
Unit of Work

```
class UnitOfWork
{
    private WarehouseContext context;

    public UnitOfWork(WarehouseContext context)
    {
        this.context = context;

        CustomerRepository          = new CustomerRepository(context);
        OrderRepository             = new OrderRepository(context);
        ItemRepository              = new ItemRepository(context);
        ShippingProviderRepository  = new ShippingProviderRepository(context);
    }

    IRepository<Customer>           CustomerRepository { get; }
    IRepository<Order>              OrderRepository { get; }
    IRepository<Item>               ItemRepository { get; }
    IRepository<ShippingProvider>   ShippingProviderRepository { get; }

    void SaveChanges() => context.SaveChanges();
}
```

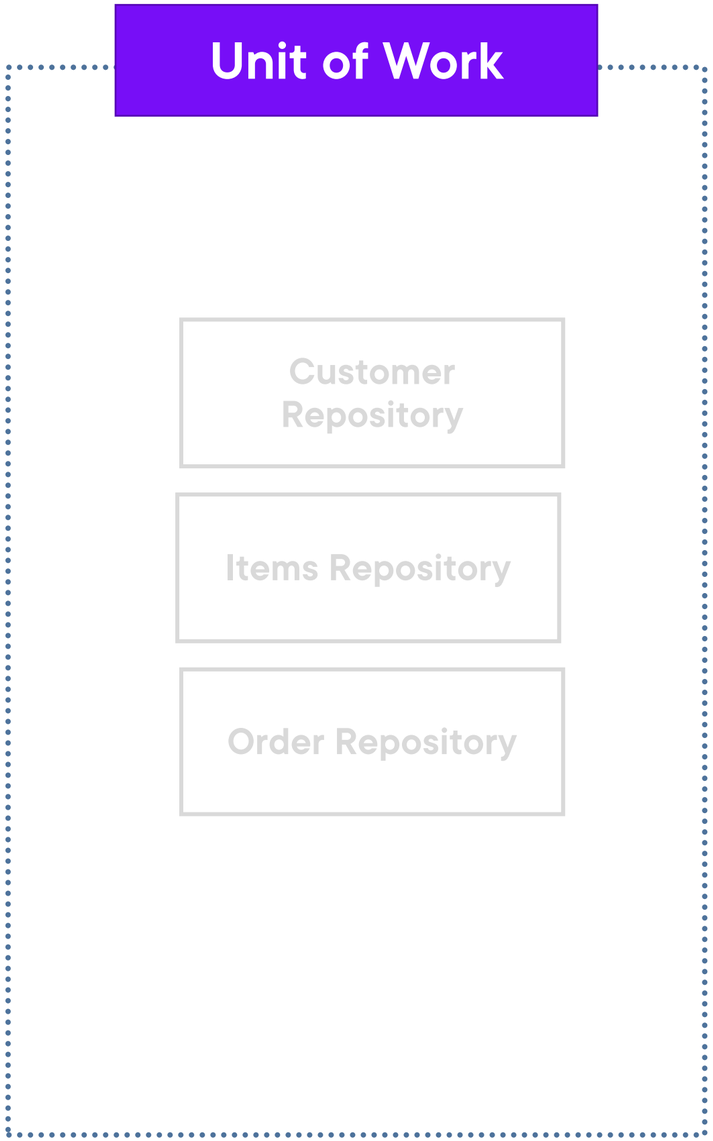
Unit of Work

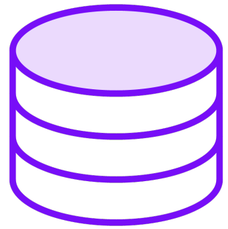
```
class UnitOfWork
{
    private WarehouseContext context;

    public UnitOfWork(WarehouseContext context)
    {
        this.context = context;

        CustomerRepository          = new CustomerRepository(context);
        OrderRepository             = new OrderRepository(context);
        ItemRepository              = new ItemRepository(context);
        ShippingProviderRepository  = new ShippingProviderRepository(context);
    }

    IRepository<Customer>           CustomerRepository { get; }
    IRepository<Order>              OrderRepository { get; }
    IRepository<Item>               ItemRepository { get; }
    IRepository<ShippingProvider>   ShippingProviderRepository { get; }

    void SaveChanges() => context.SaveChanges();
}
```

Applying the Unit of Work Pattern



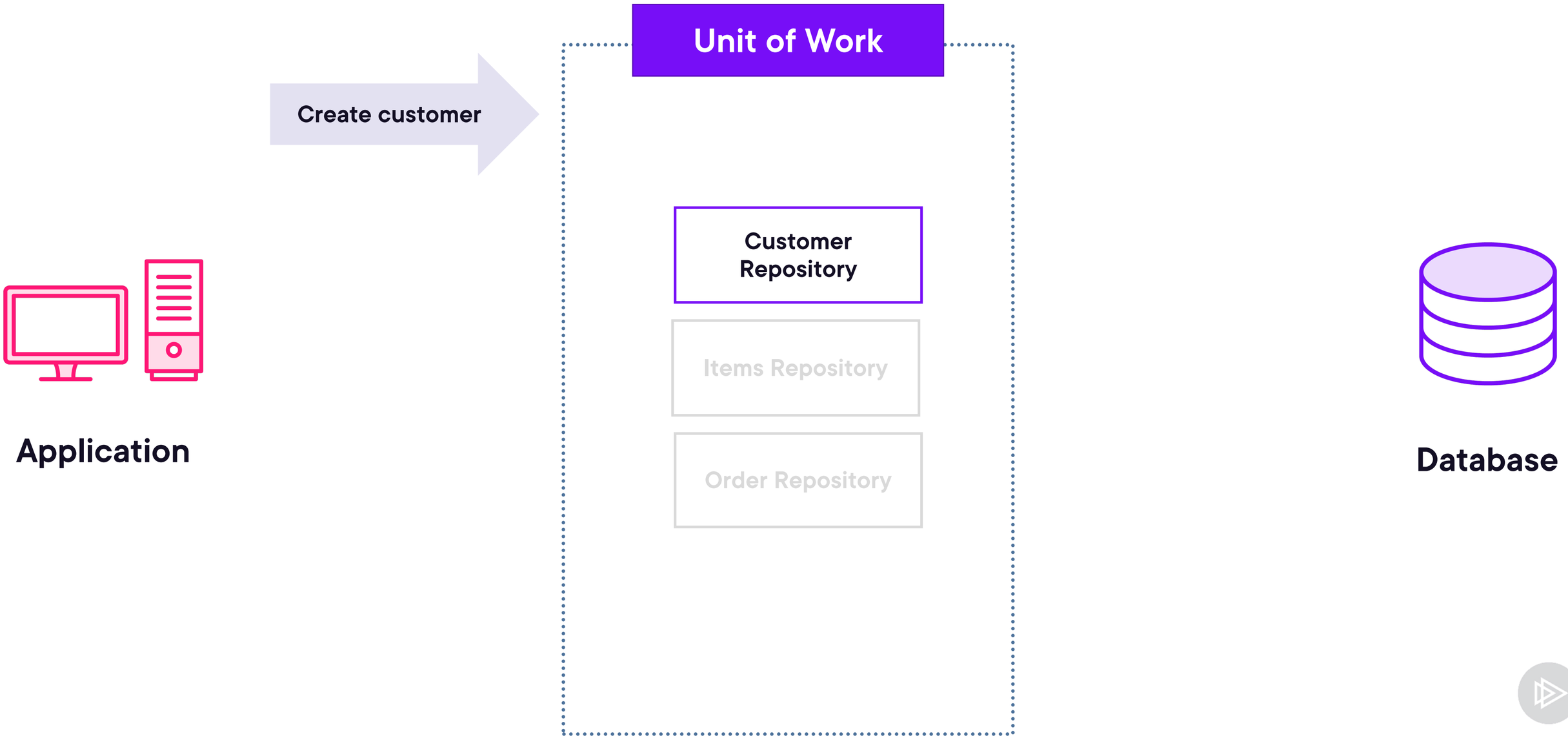
Application



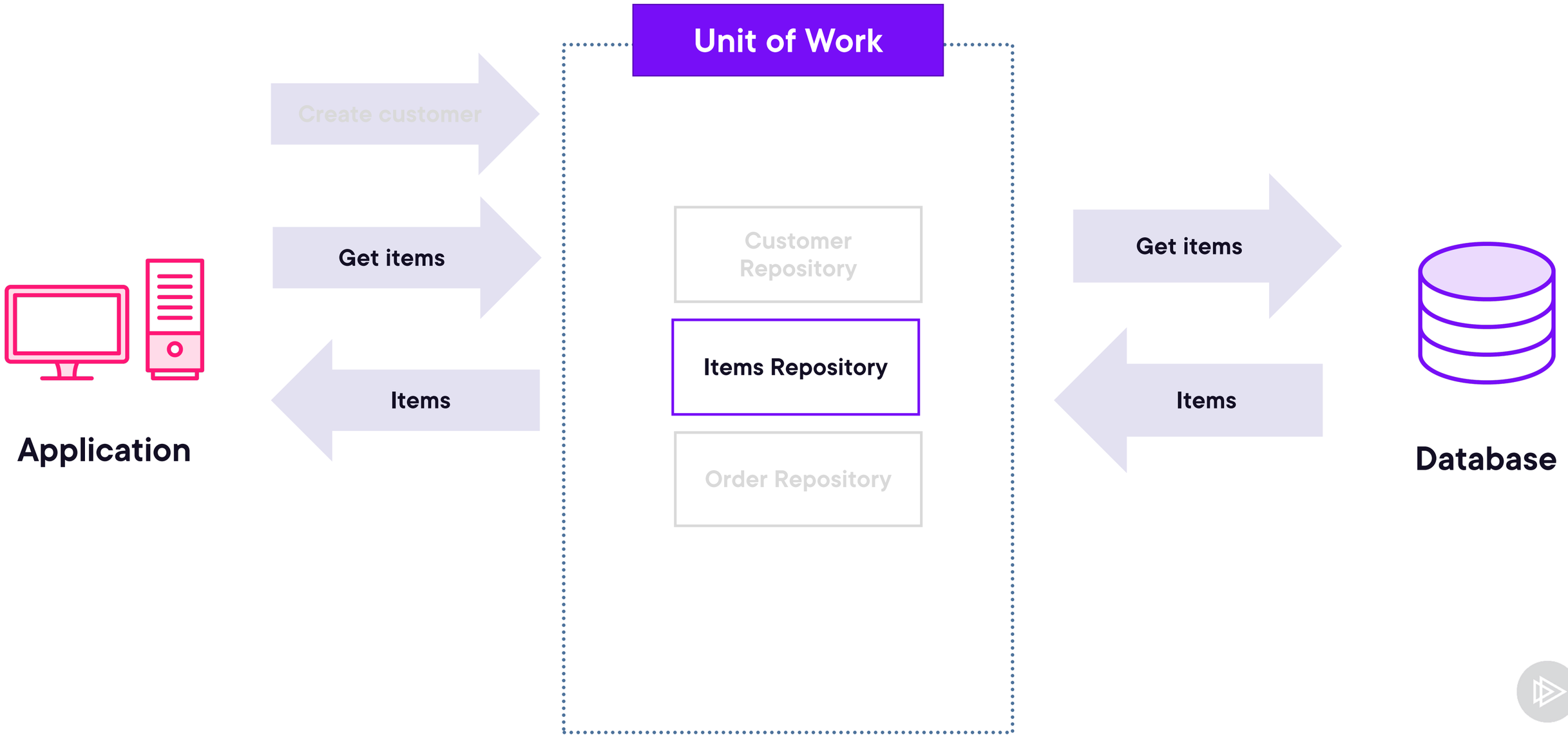


Database

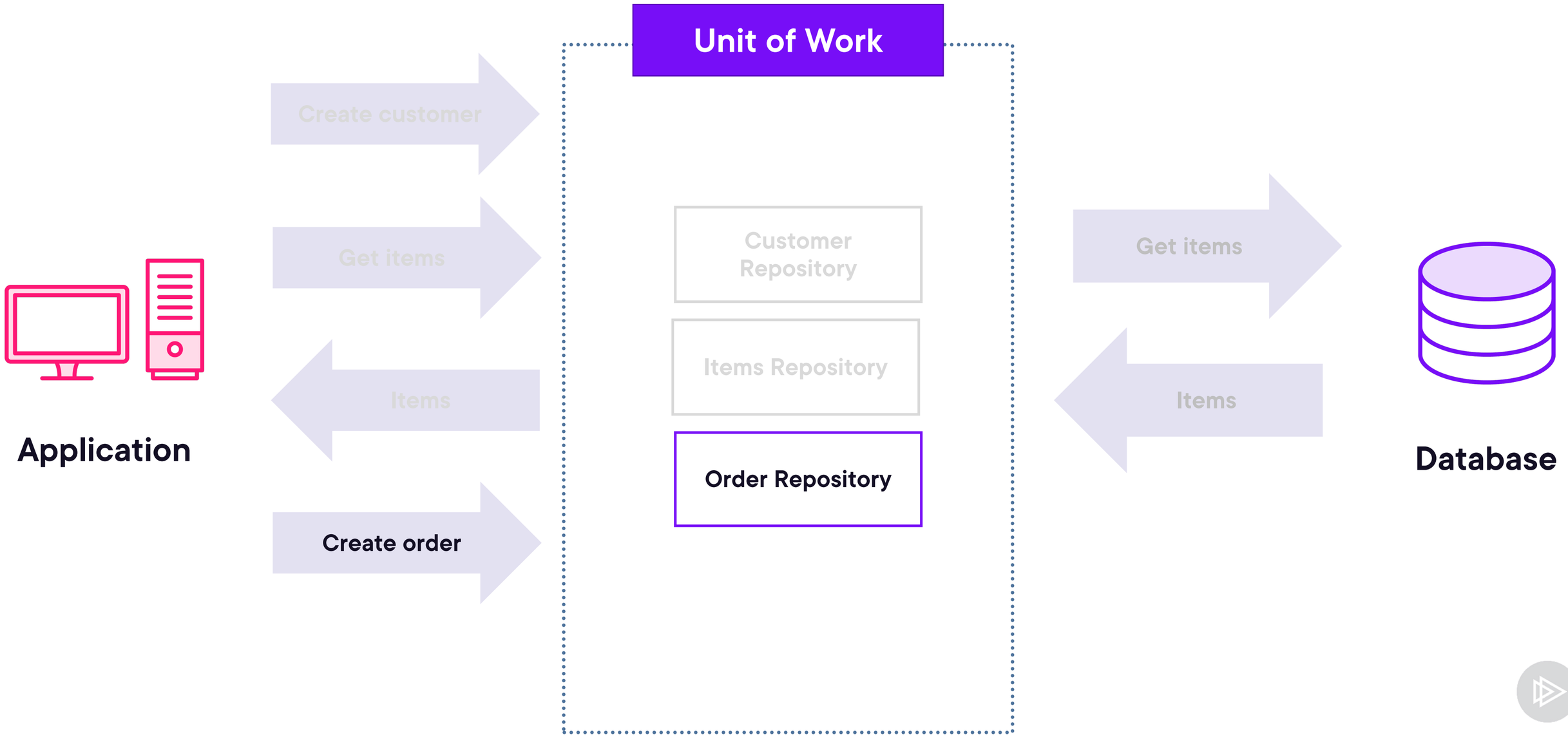
Applying the Unit of Work Pattern



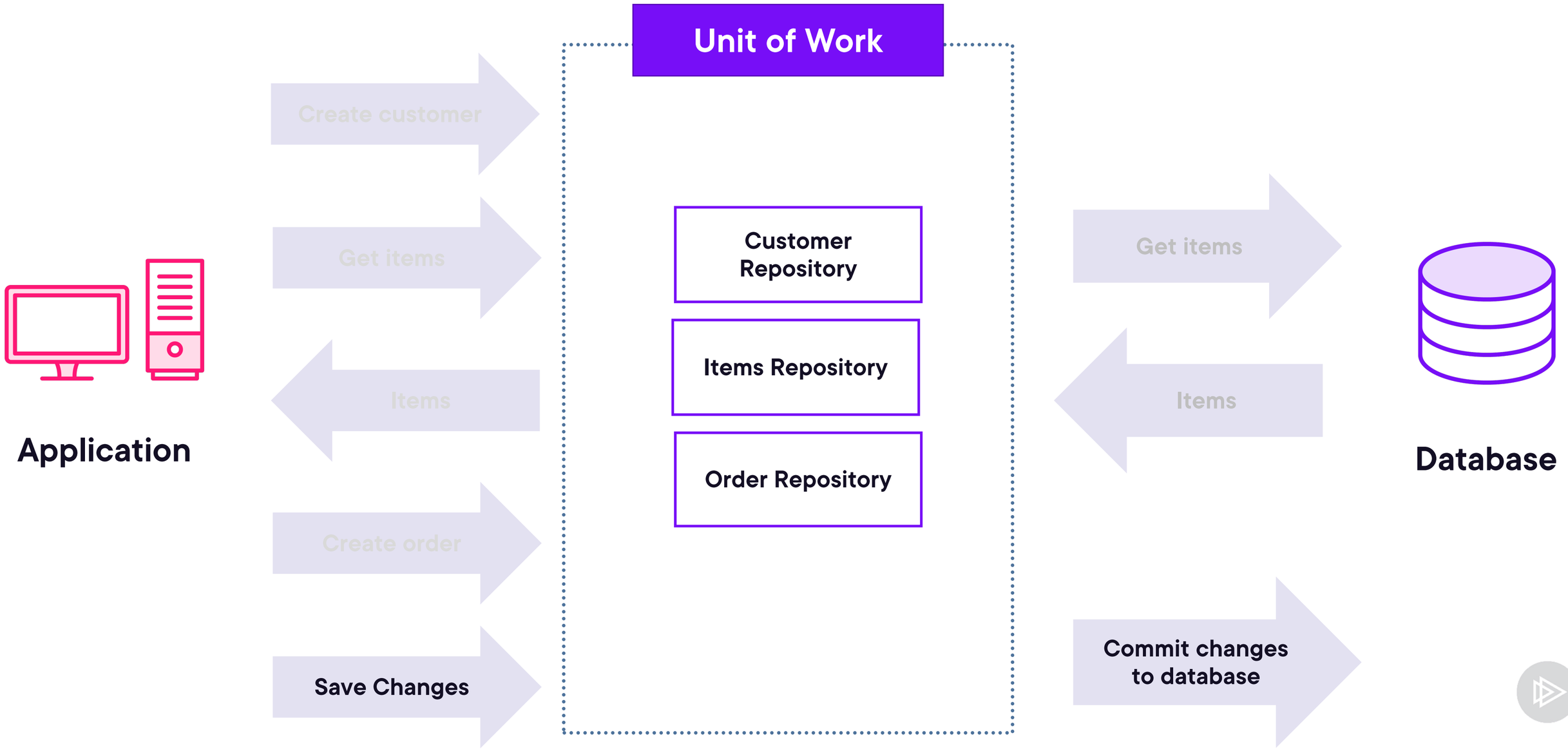
Applying the Unit of Work Pattern



Applying the Unit of Work Pattern



Applying the Unit of Work Pattern



Be less chatty with the database

Don’t introduce a pattern just for the sake of using a pattern!

Create or Update

```
var customer = customerRepository
            .Find(customer => customer.Name == model.Customer.Name)
            .FirstOrDefault();
```

Create or Update

```
var customer = customerRepository
            .Find(customer => customer.Name == model.Customer.Name)
            .FirstOrDefault();

if (customer is null)
{
    customer = new Customer { ... };
}
```

Create or Update

```
var customer = customerRepository
            .Find(customer => customer.Name == model.Customer.Name)
            .FirstOrDefault();

if (customer is null)
{
    customer = new Customer { ... };
}
else
{
    customer.Address = model.Customer.Address;
    ...

    customerRepository.Update(customer);
    customerRepository.SaveChanges();
}
```

Sharing the **same context** means **change tracking** works as expected!

Unit of Work in Entity Framework Core

```
public class WarehouseContext
    : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Warehouse> Warehouses { get; set; }
    public DbSet<Item> Items { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<ShippingProvider> ShippingProviders { get; set; }
}
```

Unit of Work in Entity Framework Core

```
public class WarehouseContext
    : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Warehouse> Warehouses { get; set; }
    public DbSet<Item> Items { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<ShippingProvider> ShippingProviders { get; set; }
}

var context = new WarehouseContext();
context.Customers.Add(new());
context.Warehouses.Add(new());
Context.SaveChanges();
```

Unit of Work in Entity Framework Core

```
public class WarehouseContext
    : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Warehouse> Warehouses { get; set; }
    public DbSet<Item> Items { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<ShippingProvider> ShippingProviders { get; set; }
}

var context = new WarehouseContext();
context.Customers.Add(new());
context.Warehouses.Add(new());
Context.SaveChanges();
```

Repositories

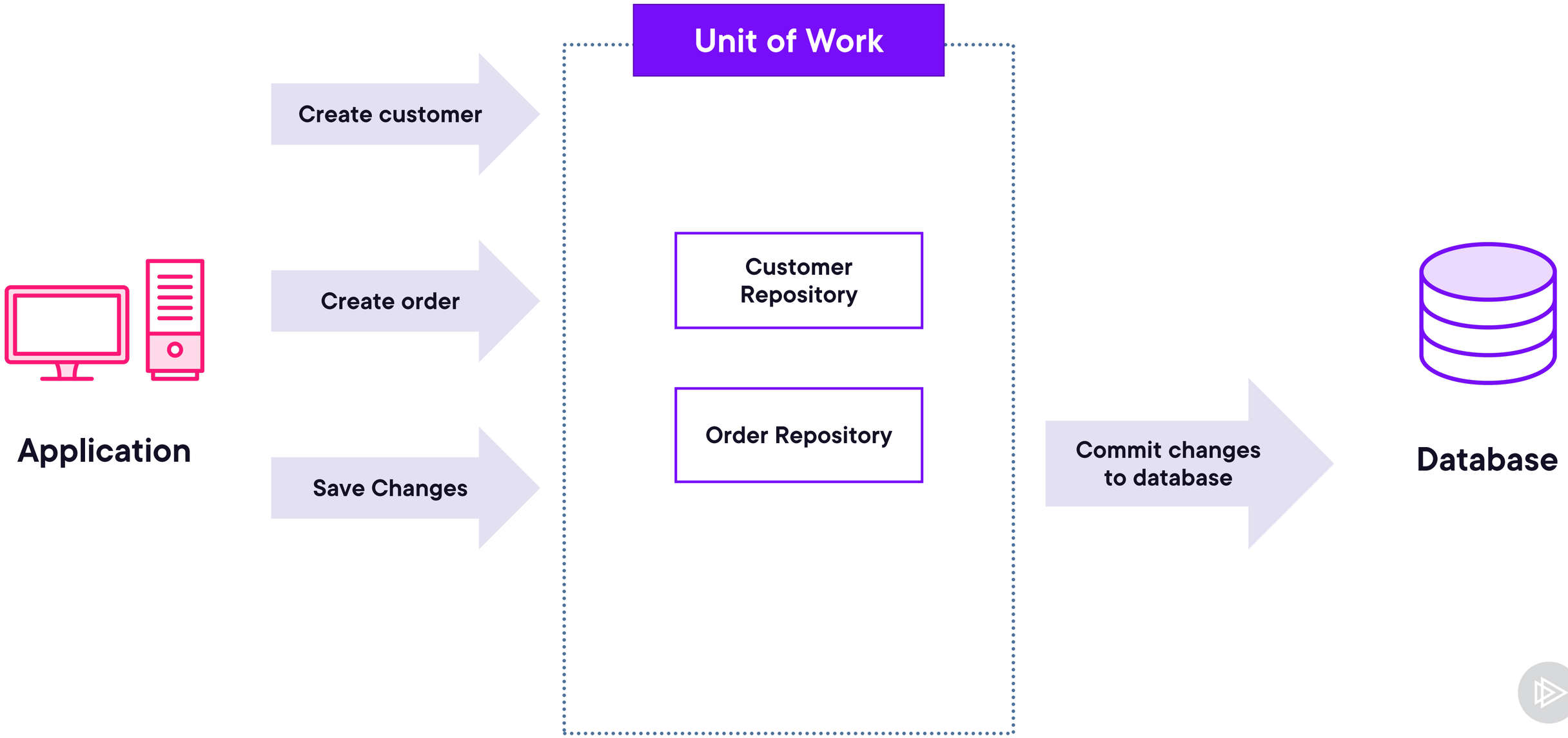
Commit all the work in a single transaction

Learn About async and await

Asynchronous Programming in C#

Filip Ekberg

Unit of Work Pattern



In **high performance** environments **limiting** the **communication** can be crucial

Don’t create too big units!

It can be difficult to know which repositories are used together when writing a test.

Don’t **eagerly load** data you won’t use!

Different Flavors of Lazy Loading

Lazy Initialization

Virtual proxies

Value holders

Ghost objects

Lazy Initialization

```
private byte[] logo;
public byte[] Logo
{
    get
    {


        return logo;
    }
    set { logo = value; }
}
```

Lazy Initialization

```
private byte[] logo;
public byte[] Logo
{
    get
    {
        if (logo == null)
        {
            logo = LogoService.GetFor(Name);
        }
        return logo;
    }
    set { logo = value; }
}
```

Lazy Initialization

```
private byte[] logo;
public byte[] Logo
{
    get
    {
        if (logo == null)
        {
            logo = LogoService.GetFor(Name);
        }
        return logo;
    }
    set { logo = value; }
}
```

Only **load** the data once, the **first time** it's accessed

Lazy Initialization

When the value is null we try to load the data

This requires the entity to know about accessing the services or databases

You have to **update the entity** after accessing the **lazy property** if you want the value cached!

Lazy Loading in Entity Framework Core

```
public class Logo
{
    public Guid CustomerId { get; set; }
    public virtual Customer Customer { get; set; }

    public byte[] ImageData { get; set; }
}

public class Customer
{
    ...
    public virtual Logo Logo { get; set; }
}
```

The **entity** is now **coupled** with **logic** to **load additional data**

Using a Value Holder and Value Loader

```
public class Customer {
    ...
    [NotMapped]
    public Lazy<byte[]> LogoValueHolder { get; set; }
}
```

Using a Value Holder and Value Loader

```
public class Customer {
    ...
    [NotMapped]
    public Lazy<byte[]> LogoValueHolder { get; set; }
}

public class CustomerRepository : GenericRepository<Customer> {
    public override Customer Get(Guid id)
    {




    }
}
```

Using a Value Holder and Value Loader

```
public class Customer {
    ...
    [NotMapped]
    public Lazy<byte[]> LogoValueHolder { get; set; }
}

public class CustomerRepository : GenericRepository<Customer> {
    public override Customer Get(Guid id)
    {
        var customer = base.Get(id);




        return customer;
    }
}
```

Using a Value Holder and Value Loader

```
public class Customer {
    ...
    [NotMapped]
    public Lazy<byte[]> LogoValueHolder { get; set; }
}

public class CustomerRepository : GenericRepository<Customer> {
    public override Customer Get(Guid id)
    {
        var customer = base.Get(id);

        customer.LogoValueHolder = new(() =>
            LogoService.GetFor(customer.Name)
        );

        return customer;
    }
}
```

Holds the **value** and **loads** it the **first time** its accessed **through** a **value loader**

Using a Value Holder and Value Loader

```
public class Customer {
    ...
    [NotMapped]
    public Lazy<byte[]> LogoValueHolder { get; set; }
}

public class CustomerRepository : GenericRepository<Customer> {
    public override Customer Get(Guid id)
    {
        var customer = base.Get(id);

        customer.LogoValueHolder = new(() =>
            LogoService.GetFor(customer.Name)
        );

        return customer;
    }
}
```

Using a Value Holder and Value Loader

```
public class Customer {
    ...
    [NotMapped]
    public Lazy<byte[]> LogoValueHolder { get; set; }
}

public class CustomerRepository : GenericRepository<Customer> {
    public override Customer Get(Guid id)
    {
        var customer = base.Get(id);

        customer.LogoValueHolder = new(() =>
            LogoService.GetFor(customer.Name)
        );

        return customer;
    }
}
```

Using a Value Holder and Value Loader

```
public class Customer {
    ...
    [NotMapped]
    public Lazy<byte[]> LogoValueHolder { get; set; }
}

public class CustomerRepository : GenericRepository<Customer> {
    public override Customer Get(Guid id)
    {
        var customer = base.Get(id);

        customer.LogoValueHolder = new(() =>
            LogoService.GetFor(customer.Name)
        );

        return customer;
    }
}
```

Entity no longer coupled with the logic to load the logo!

The value holder is marked as being ignored by Entity Framework Core

Lazy<T>

“Provides support for lazy initialization.”

Implements the value holder and value loader pattern.

Example:

```
Lazy<byte[]> lazyLogo = new(() =>
    LogoService.GetFor(customer.Name)
);
```

learn.microsoft.com/en-us/dotnet/api/system.lazy-1

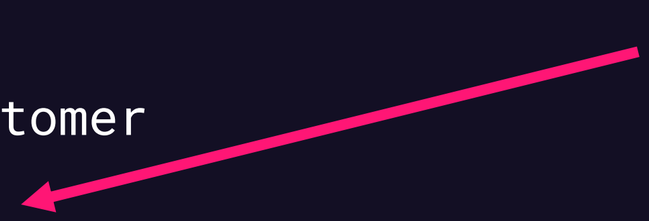
Virtual Proxies

```
public class CustomerProxy : Customer
{
    public override byte[] Logo
    {
        get {
            if (base.Logo == null) {
                base.Logo = LogoService.GetFor(Name);
            }
            return base.Logo;
        }
    }
}

public class Customer
{
    public virtual byte[] Logo { get; set; }
}
```

Virtual Proxies

```
public class CustomerProxy : Customer
{
    public override byte[] Logo
    {
        get {
            if (base.Logo == null) {
                base.Logo = LogoService.GetFor(Name);
            }
            return base.Logo;
        }
    }
}
```

Mark the property as **virtual**



```
public class Customer
{
    public virtual byte[] Logo { get; set; }
}
```

Virtual Proxies

```
public class CustomerProxy : Customer
{
    public override byte[] Logo
    {
        get {
            if (base.Logo == null) {
                base.Logo = LogoService.GetFor(Name);
            }
            return base.Logo;
        }
    }
}

public class Customer
{
    public virtual byte[] Logo { get; set; }
}
```

Using a Virtual Proxy

```
var customer = base.Get(id);

var proxy = new CustomerProxy
{
    Name = customer.Name,
    PostalCode = customer.PostalCode,
    Address = customer.ShippingAddress,
    Country = customer.Country,
    PhoneNumber = customer.PhoneNumber
};

return proxy;
```

Ghost Objects

```
public class CustomerGhost : Customer
{
    public override string Name {
        get {
            Load();
            return base.Name;
        }
        set {
            Load();
            base.Name = value;
        }
    }
}
```

Extending the GenericRepository<T>

```
public class OrderRepository : GenericRepository<Order>
{
    public OrderRepository(WarehouseContext context)
        : base(context) { }

    public override IEnumerable<Order>
        Find(Expression<Func<Order, bool>> predicate)
    {
        return context.Orders
        .Include(order => order.LineItems)
        .ThenInclude(lineItem => lineItem.Item)
        .Where(predicate)
        .ToList();
    }
}
```

Virtual Proxy and Ghost Object

Virtual Proxy

Sub-class overrides the property to define how the data is loaded

Ghost Object

Represent the entity in a partial state. Load all data when any property is accessed.