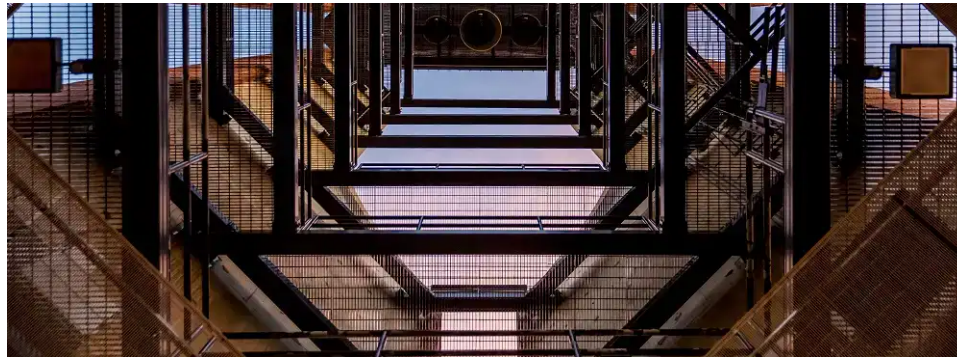




Kyle Galbraith

A blog about cloud computing, software development, and my entrepreneur adventures



Getting Familiar With The Awesome Repository Pattern

 06 March, 2018 – Kyle Galbraith

The repository pattern is another abstraction, like most things in Computer Science. It is a pattern that is applicable in many different languages. In fact a lot of developers use the repository pattern and don't even realize it.

In this post I am going to transform a piece of code. We start with a piece of code that is loading a single record from a database. Once the record is fetched it is returned to the caller. Let's take a look at some code.

Needs Improvement

The record we are loading out of our database is `PersonModel`.

```
public class PersonModel
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

The service that is loading a person out of the database is `ICompanyLogic`. It consists of the following method definition.

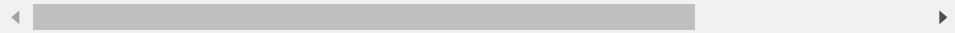
```
public interface ICompanyLogic
{
    PersonModel GetPersonByName(string name);
}
```

The implementation of the `ICompanyLogic` is handled by `CompanyLogic`.

```
public class CompanyLogic: ICompanyLogic
{
```

```
private IPersonDataContext _personDataContext;
public PersonService(IPersonDataContext personDa
{
    _personDataContext= personDataContext;
}

public PersonModel GetPersonByName(string name)
{
    using(var ctx = _personDataContext.NewContex
    {
        var person = ctx.People.First(p => p.Nam
        return person;
    }
}
}
```



So far, this isn't so bad. We have a business service
CompanyLogic that can retrieve a single person from the database.

But then we have a new requirement that says we also need a way
to load a company from another database. So we need to add a
new method and extend CompanyLogic.

CompanyModel represents the model stored in the company
database.

```
public class CompanyModel
{
    public string Name { get; set; }
    public int Size { get; set; }
```

```
public bool Public { get; set; }  
}
```

We extend `CompanyLogic` to have a method that returns a company by name.

```
public class CompanyLogic: ICompanyLogic  
{  
    private IPersonDataContext _personDataContext;  
    private ICompanyDataContext _companyDataContext;  
    public PersonService(IPersonDataContext personDa  
        ICompanyDataContext company  
    {  
        _personDataContext= personDataContext;  
        _companyDataContext = companyDataContext;  
    }  
  
    public PersonModel GetPersonByName(string name)  
    {  
        using(var ctx = _personDataContext.NewContex  
        {  
            var person = ctx.People.First(p => p.Nam  
            return person;  
        }  
    }  
  
    public CompanyModel GetCompanyByName(string comp  
    {  
        using(var ctx = _companyDataContext.NewConte
```

```
{  
    var person = ctx.Company.First(c => c.Name == name);  
    return person;  
}  
}
```

Now we are starting to see the problems with this initial solution. Here is a short list of things that are not ideal.

- `CompanyLogic`, knows how to access two different databases.
- We have duplicated code with our `using` statements.
- Our logic knows how people and companies are stored.
- `GetPersonByName` and `GetCompanyByName` cannot be reused without bringing in all of `CompanyLogic`.

In addition to all of these things, how do we test `CompanyLogic` in its current state? We have to mock the data context for people and companies to have literal database records. This is possible to do. But our hard work should be going into testing our logic, not mocking database objects.

Implementing Repository Pattern

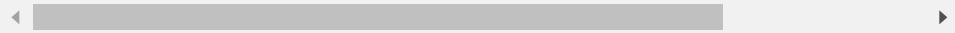
The repository pattern adds an abstraction layer over the top of data access. A little bit of abstraction goes a long way. With the repository pattern we can add a thin layer of abstraction for accessing the people and company databases. Then `CompanyLogic` or any other logic can leverage those abstractions.

Let's begin by creating our `IPersonRepository` interface and its accompanying implementation.

```
public interface IPersonRepository
{
    PersonModel GetPersonByName(string name);
}

public class PersonRepository: IPersonRepository
{
    private IPersonDataContext _personDataContext;
    public PersonRepository(IPersonDataContext perso
    {
        _personDataContext= personDataContext;
    }

    public PersonModel GetPersonByName(string name)
    {
        using(var ctx = _personDataContext.NewContext
        {
            return ctx.People.First(p => p.Name.Equa
        }
    }
}
```

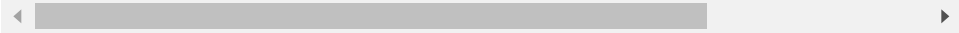


Then we can do something very similar for companies. We can create the `ICompanyRepository` interface and its implementation.

```
public interface ICompanyRepository
{
    PersonModel GetCompanyByName(string name);
}

public class CompanyRepository: ICompanyRepository
{
    private ICompanyDataContext _companyDataContext;
    public CompanyRepository(ICompanyDataContext comp
    {
        _companyDataContext= personDataContext;
    }

    public CompanyModel GetCompanyByName(string name
    {
        using(var ctx = _companyDataContext.NewConte
        {
            return ctx.Company.First(p => p.Name.Equ
        }
    }
}
```



We now have two separate repositories. PersonRepository knows how to load a given person by name from the person database. CompanyRepository can load companies by name from the company database. Now let's refactor CompanyLogic to leverage these repositories instead of the data contexts.

```
public class CompanyLogic: ICompanyLogic
{
    private IPersonRepository _personRepo;
    private ICompanyRepository _companyRepo;
    public PersonService(IPersonRepository personRep
                        ICompanyRepository companyR
    {
        _personRepo= personRepo;
        _companyRepo= companyRepo;
    }

    public PersonModel GetPersonByName(string name)
    {
        return _personRepo.GetPersonByName(name);
    }

    public CompanyModel GetCompanyByName(string comp
    {
        return _companyRepo.GetCompanyByName(company
    }
}
```

Look at that, our logic layer no longer knows anything about databases. We have abstracted away how a person and a company are loaded. So what benefits have we gained?

- The repository interfaces are reusable. They could be used in other logic layers without changing a thing.
- Testing is a lot simpler. We mock the interface response so we can focus on testing our logic.

- Database access code for people and companies is centrally managed in one place.
- Optimizations can be made at a repository level. The interface is defined and agreed upon. The developer working on the repository can then store data how she sees fit.

Repository pattern provides us with a nice abstraction for our data. This is applicable to a variety of languages. The moral of the story is that data access should be a single responsibility interface. This interface can then be injected into business layers to add any additional logic.

Hungry To Learn Amazon Web Services?

There is a lot of people that are hungry to learn Amazon Web Services. Inspired by this fact I have created a course focused on learning Amazon Web Services by using it. Focusing on the problem of hosting, securing, and delivering static websites. You learn services like S3, API Gateway, CloudFront, Lambda, and WAF by building a solution to the problem.

There is a sea of information out there around AWS. It is easy to get lost and not make any progress in learning. By working through this problem we can cut through the information and speed up your learning. My goal with this book and video course is to share what I have learned with you.

Sound interesting? Check out the landing page to learn more and pick a package that works for you, [here](#).

If you'd like to get updates on my future articles and projects, please subscribe to my newsletter.

enter your email...

Subscribe

Did you enjoy this post? Share the ♥ with others.



© 2018 Kyle Galbraith. All Rights Reserved.

~~Software Manager~~

[How To Get Started With Test Driven Development Today →](#)