|  |
| --- |
|  |
| Layered LINQ |
| Abstractions For Data Access |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

Layered LINQ

Objectives

After completing this lab, you should understand how to do the following:

* Apply the repository and unit of work design patterns
* Build a unit test with Moq objects

Overview

We are going to take an existing ASP.NET MVC application and add some abstractions around the current data access that is in place.

Part 1 – Abstractions

1. Open the EmployeeTimeCards solution in the before folder of this lab.
2. Build and run the solution to ensure everything is working. You should be able to create and edit employees.
3. In the .Core project, add an interface definition for IEmployeeRepository. The repository should have method definitions to:
   1. Find an employee by ID.
   2. Add an employee
   3. Update an employee
   4. Find all employees.

public interface IEmployeeRepository

{

Employee Find(int id);

IEnumerable<Employee> FindAll();

void Add(Employee newEmployee);

void Update(Employee employee);

}

1. In the .Core project, add an interface definition for IUnitOfWork. The unit of work is responsible for:
   1. Committing all changes
   2. Handing out a reference to an employee repository.

public interface IUnitOfWork

{

IEmployeeRepository Employees { get;}

void Commit();

}

## Part II – Implementations

1. In the .Web project, add a class to the Infrastructure folder named SqlEmployeeRepository.
2. Make the new repository implement the IEmployeeRepository interface. You’ll need to use the existing TimeCardContext (take it as a constructor parameter – don’t create one).

public class SqlEmployeeRepository : IEmployeeRepository

{

private readonly TimeCardContext \_db;

public SqlEmployeeRepository(TimeCardContext db)

{

\_db = db;

}

public Employee Find(int id)

{

return \_db.Employees.Find(id);

}

public IEnumerable<Employee> FindAll()

{

return \_db.Employees.ToList();

}

public void Add(Employee newEmployee)

{

\_db.Employees.Add(newEmployee);

}

public void Update(Employee employee)

{

\_db.Entry(employee).State = EntityState.Modified;

}

}

1. Add another class to the .Web project named SqlUnitOfWork and implement the IUnitOfWork interface.

public class SqlUnitOfWork : IUnitOfWork

{

private TimeCardContext \_db;

public SqlUnitOfWork()

{

\_db = new TimeCardContext();

}

public IEmployeeRepository Employees

{

get

{

return new SqlEmployeeRepository(\_db);

}

}

public void Commit()

{

db.SaveChanges();

}

}

## Part III – Putting Implementations to Use

1. Open the HomeController class in the Controllers folder of the .Web project.
2. Add private fields to the controller type IEmployeeRepository and IUnitOfWork.
3. Add a default constructor to HomeController that initializes the repository and unit of work to a SqlUnitOfWork and SqlRepository.

private IUnitOfWork \_db;

private IEmployeeRepository \_employeeRepository;

public HomeController()

{

\_db = new SqlUnitOfWork();

\_employeeRepository = \_db.Employees;

}

1. Rewrite the implementations of every action to use the new abstractions instead of a TimeCardContext.

public ActionResult Index()

{

var model = \_employeeRepository.FindAll();

return View(model);

}

public ActionResult Details(int id)

{

var model = \_employeeRepository.Find(id);

return View(model);

}

[HttpGet]

public ActionResult Edit(int id)

{

var model = \_employeeRepository.Find(id);

return View(model);

}

[HttpPost]

public ActionResult Edit(Employee updatedEmployee)

{

if(ModelState.IsValid)

{

\_employeeRepository.Update(updatedEmployee);

\_db.Commit();

return RedirectToAction("Index");

}

return View(updatedEmployee);

}

[HttpPost]

public ActionResult Create(Employee newEmployee)

{

if(ModelState.IsValid)

{

\_employeeRepository.Add(newEmployee);

\_db.Commit();

return RedirectToAction("Index");

}

return View(newEmployee);

}

1. Run the application to verify your changes.

## Part IV – Test After Development

1. Add a new constructor to HomeController allowing you to pass in IUnitOfWork. Use the parameter to initialize the private fields.

public HomeController(IUnitOfWork unitOfWork)

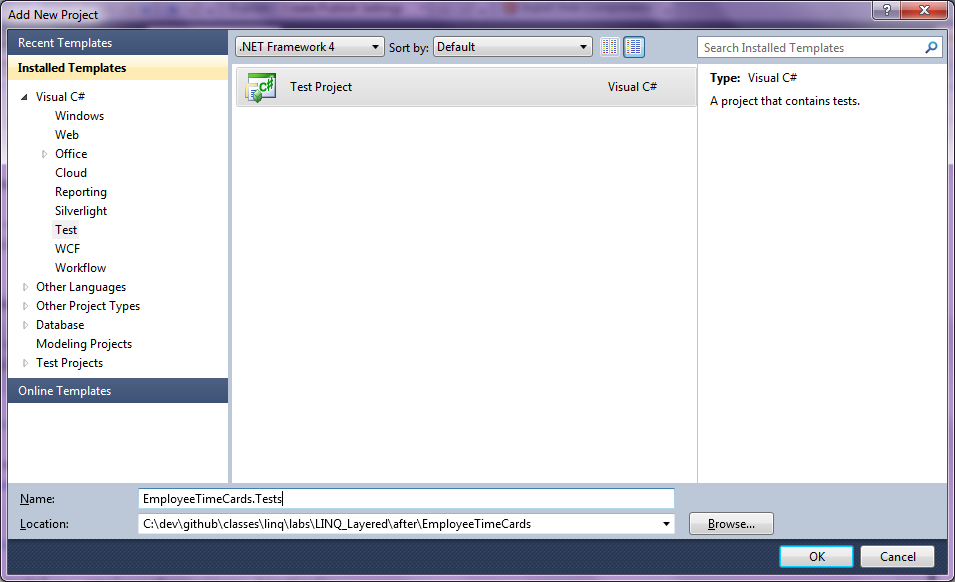
{

\_db = unitOfWork;

\_employeeRepository = \_db.Employees;

}

1. Add a new Test project to the solution. Name the project EmployeeTimeCards.Tests.



1. Delete UnitTest1.cs from the test project.
2. Add project references from the .Tests project to .Web and .Core.
3. Add a new class to the .Tests project named FakeEmployeeRepository.
4. Implement IEmployeeRepository from FakeEmployeeRepository using List<Employee> as a backing source.

public class FakeEmployeeRepository : IEmployeeRepository

{

private List<Employee> \_employees = new List<Employee>();

public Employee Find(int id)

{

return \_employees.First(e => e.Id == id);

}

public IEnumerable<Employee> FindAll()

{

return \_employees;

}

public void Add(Employee newEmployee)

{

\_employees.Add(newEmployee);

}

public void Update(Employee employee)

{

}

public void Delete(int id)

{

\_employees.Remove(Find(id));

}

}

1. Add a FakeUnitOfWork class to the test project and implement IUnitOfWork. Provider an auto property for the Employees property with both a getter and setter. Also implement a public property named Commited that will tell you if the Commit method has been invoked.

public class FakeUnitOfWork : IUnitOfWork

{

public IEmployeeRepository Employees { get; set; }

public void Commit()

{

Committed = true;

}

public bool Committed { get; set; }

}

1. Write a HomeController\_Edit test class to the .Tests project.
2. Implement a [TestInitialize] method in the unit test to instantiate a FakeUnitOfWork with a FakeEmployeeRepository and save the unit of work to a private field. Add three employees to the repository.

[TestInitialize]

public void Initialize()

{

\_unitOfWork = new FakeUnitOfWork();

\_unitOfWork.Employees = new FakeEmployeeRepository();

\_unitOfWork.Employees.Add(new Employee() { Id = 1 });

\_unitOfWork.Employees.Add(new Employee() { Id = 2 });

\_unitOfWork.Employees.Add(new Employee() { Id = 3 });

}

1. Implement a unit test to ensure the HomeController returns the correct model for the Edit method that responds to an HTTP GET request.

[TestMethod]

public void Returns\_Correct\_Model\_On\_Edit\_Get()

{

var controller = new HomeController(\_unitOfWork);

var result = controller.Edit(2);

var model = ((ViewResult) result).Model as Employee;

Assert.AreEqual(2, model.Id);

}

1. Implement another test method in the test class to ensure the controller commits its work when updating an employee.

[TestMethod]

public void Commits\_When\_Edit\_Succesfull()

{

var controller = new HomeController(\_unitOfWork);

var employeeToEdit = \_unitOfWork.Employees.Find(1);

controller.Edit(employeeToEdit);

Assert.IsTrue(\_unitOfWork.Committed);

}

1. Add a third unit test to ensure the controller does not commit changes when ModelState is marked as invalid.

[TestMethod]

public void Does\_Not\_Commit\_When\_ModelState\_Invalid()

{

var controller = new HomeController(\_unitOfWork);

var employeeToEdit = \_unitOfWork.Employees.Find(1);

controller.ModelState.AddModelError("", "");

controller.Edit(employeeToEdit);

Assert.IsFalse(\_unitOfWork.Committed);

}

## Part V – IoC Containers

1. Open the HomeController.cs file.
2. Remove the default constructor from the HomeController.
3. Run the application – you should receive an error “No parameterless constructor defined for this object”.
4. Open the Package Manager Console window (View => Other Windows => Package Manager Console).
5. Make sure the .Web project is listed as the **Default Project** in the console window.
6. Type “install-package StructureMap.MVC3” in the console.
7. Open IoC.cs from the DependencyResolution folder.
8. Replace the commented out line starting with x.For<IExample> with code to .Use SqlUnitOfWork .For IUnitOfWork.

ObjectFactory.Initialize(x =>

{

x.Scan(scan =>

{

scan.TheCallingAssembly();

scan.WithDefaultConventions();

});

**x.For<IUnitOfWork>().Use<SqlUnitOfWork>();**

});

1. Run the application again – you should be able to list and edit employees.

# Finished!

Congratulations – you’ve had a peek at how to build a more layered application using LINQ. Contrast this with an application you might have built in the past using ADO.NET.