Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10)

The Contoso University sample web application demonstrates how to create ASP.NET MVC applications using the Entity Framework. The sample application is a website for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments.

This tutorial series explains the steps taken to build the Contoso University sample application. You can download the completed application or create it by following the steps in the tutorial. The tutorial shows examples in C#. The downloadable sample contains code in both C# and Visual Basic. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET Entity Framework forum or the Entity Framework and LINQ to Entities forum.

This tutorial series assumes you know how to work with ASP.NET MVC in Visual Studio. If you don't, a good place to start is a basic ASP.NET MVC Tutorial. If you prefer to work with the ASP.NET Web Forms model, see the Getting Started with the Entity Framework and Continuing with the Entity Framework tutorials.

Before you start, make sure you have the following software installed on your computer:

- Visual Studio 2010 SP1 or Visual Web Developer Express 2010 SP1 (If you use one of these links, the following items will be installed automatically.)
- ASP.NET MVC 3 Tools Update
- Microsoft SQL Server Compact 4.0
- Microsoft Visual Studio 2010 SP1 Tools for SQL Server Compact 4.0

Download the complete application | Download complete tutorial in PDF form

In the previous tutorial you used inheritance to reduce redundant code in the `Student` and `Instructor` entity classes. In this tutorial you'll see some ways to use the repository and unit of work patterns for CRUD operations. As in the previous tutorial, in this one you'll change the way your code works with pages you already created rather than creating new pages.

The Repository and Unit of Work Patterns

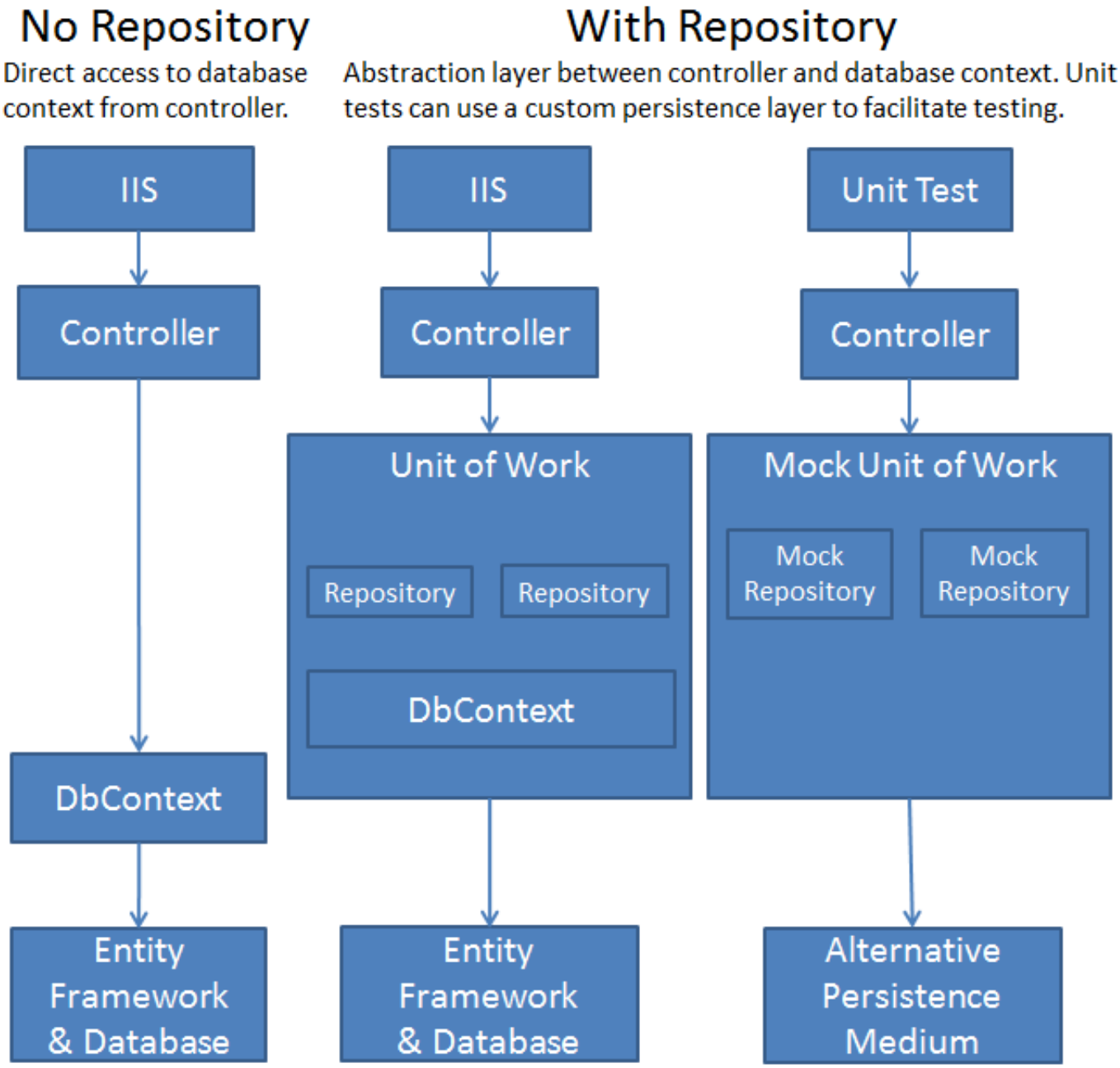
The repository and unit of work patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD).

In this tutorial you'll implement a repository class for each entity type. For the `Student` entity type you'll create a repository interface and a repository class. When you instantiate the repository in your controller, you'll use the interface so that the controller will accept a reference to any object that implements the repository interface. When the controller runs under a web server, it receives a repository

implements the repository interface. When the controller runs under a web server, it receives a repository that works with the Entity Framework. When the controller runs under a unit test class, it receives a repository that works with data stored in a way that you can easily manipulate for testing, such as an in-memory collection.

Later in the tutorial you'll use multiple repositories and a unit of work class for the `Course` and `Department` entity types in the `Course` controller. The unit of work class coordinates the work of multiple repositories by creating a single database context class shared by all of them. If you wanted to be able to perform automated unit testing, you'd create and use interfaces for these classes in the same way you did for the `Student` repository. However, to keep the tutorial simple, you'll create and use these classes without interfaces.

The following illustration shows one way to conceptualize the relationships between the controller and context classes compared to not using the repository or unit of work pattern at all.



You can't create unit tests in this tutorial series. For an introduction to TDD with an MVC application that

You won't create unit tests in this tutorial series. For an introduction to TDD with an MVC application that uses the repository pattern, see Walkthrough: Using TDD with ASP.NET MVC on the MSDN Library web site. For more information about the repository pattern, see Using Repository and Unit of Work patterns with Entity Framework 4.0 on the Entity Framework team blog and the Agile Entity Framework 4 Repository series of posts on Julie Lerman's blog.

Note There are many ways to implement the repository and unit of work patterns. You can use repository classes with or without a unit of work class. You can implement a single repository for all entity types, or one for each type. If you implement one for each type, you can use separate classes, a generic base class and derived classes, or an abstract base class and derived classes. You can include business logic in your repository or restrict it to data access logic. You can also build an abstraction layer into your database context class by using `IDbSet` interfaces there instead of `DbSet` types for your entity sets. The approach to implementing an abstraction layer shown in this tutorial is one option for you to consider, not a recommendation for all scenarios and environments.

Creating the Student Repository Class

In the *DAL* folder, create a class file named *IStudentRepository.cs* and replace the existing code with the following code:

```
using System;using System.Collections.Generic;using System.Linq;u
sing System.Web;using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{public interface IStudentRepository : IDisposable{IEnumerable<St
udent> GetStudents();Student GetStudentByID(int studentId);void I
nsertStudent(Student student);void DeleteStudent(int studentID);v
oid UpdateStudent(Student student);void Save();}}
```

This code declares a typical set of CRUD methods, including two read methods — one that returns all `Student` entities, and one that finds a single `Student` entity by ID.

In the *DAL* folder, create a class file named *StudentRepository.cs* file. Replace the existing code with the following code, which implements the `IStudentRepository` interface:

```
using System;using System.Collections.Generic;using System.Linq;u
sing System.Data;using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{public class StudentRepository : IStudentRepository, IDisposable
{private SchoolContext context;

        public StudentRepository(SchoolContext context){this.cont
ext = context;}
```

```
        public IEnumerable<Student> GetStudents(){return context.
Students.ToList();}

        public Student GetStudentByID(int id){return context.Stud
ents.Find(id);}

        public void InsertStudent(Student student){
            context.Students.Add(student);}

        public void DeleteStudent(int studentID){Student student
= context.Students.Find(studentID);
            context.Students.Remove(student);}

        public void UpdateStudent(Student student){
            context.Entry(student).State = EntityState.Modified;}

        public void Save(){
            context.SaveChanges();}

        private bool disposed = false;

        protected virtual void Dispose(bool disposing){if (!this.
disposed){if (disposing){
                    context.Dispose();}}this.disposed = true;}

        public void Dispose(){Dispose(true);
            GC.SuppressFinalize(this);}}}
```

The database context is defined in a class variable, and the constructor expects the calling object to pass in an instance of the context:

```
private SchoolContext context;

public StudentRepository(SchoolContext context){this.context = co
ntext;}
```

You could instantiate a new context in the repository, but then if you used multiple repositories in one controller, each would end up with a separate context. Later you'll use multiple repositories in the `Course` controller, and you'll see how a unit of work class can ensure that all repositories use the same context.

The repository implements `IDisposable` and disposes the database context as you saw earlier in

the controller, and its CRUD methods make calls to the database context in the same way that you saw earlier.

Changing the Student Controller to Use the Repository

In *StudentController.cs*, replace the code currently in the class with the following code:

```
using System;using System.Collections.Generic;using System.Data;u
sing System.Data.Entity;using System.Linq;using System.Web;using
System.Web.Mvc;using ContosoUniversity.Models;using ContosoUniver
sity.DAL;using PagedList;

namespace ContosoUniversity.Controllers{public class StudentContr
oller : Controller{private IStudentRepository studentRepository;



        public StudentController(){this.studentRepository = new S
tudentRepository(new SchoolContext());}

        public StudentController(IStudentRepository studentReposi
tory){this.studentRepository = studentRepository;}



        //// GET: /Student/

        public ViewResult Index(string sortOrder, string currentF
ilter, string searchString, int? page){ViewBag.CurrentSort = sort
Order;ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "N
ame desc" : "";ViewBag.DateSortParm = sortOrder == "Date" ? "Date
 desc" : "Date";

            if (Request.HttpMethod == "GET"){
                searchString = currentFilter;}else{
                page = 1;}ViewBag.CurrentFilter = searchString;

            var students = from s in studentRepository.GetStudent
s()select s;if (!String.IsNullOrEmpty(searchString)){
                students = students.Where(s => s.LastName.ToUpper
().Contains(searchString.ToUpper())|| s.FirstMidName.ToUpper().Co
ntains(searchString.ToUpper()));}switch (sortOrder){case "Name de
sc":
                    students = students.OrderByDescending(s => s.
LastName);break;case "Date":
```

```
                    students = students.OrderBy(s => s.EnrollmentDate);break;case "Date desc":
                    students = students.OrderByDescending(s => s.EnrollmentDate);break;default:
                    students = students.OrderBy(s => s.LastName);break;}

            int pageSize = 3;int pageIndex = (page ?? 1) - 1;return View(students.ToPagedList(pageIndex, pageSize));}


        //// GET: /Student/Details/5

        public ViewResult Details(int id){Student student = studentRepository.GetStudentByID(id);return View(student);}

        //// GET: /Student/Create

        public ActionResult Create(){return View();}

        //// POST: /Student/Create

        [HttpPost]public ActionResult Create(Student student){try{if (ModelState.IsValid){
                    studentRepository.InsertStudent(student);
                    studentRepository.Save();return RedirectToAction("Index");}}catch (DataException){//Log the error (add a variable name after DataException)ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists see your system administrator.");}return View(student);}

        //// GET: /Student/Edit/5

        public ActionResult Edit(int id){Student student = studentRepository.GetStudentByID(id);return View(student);}

        //// POST: /Student/Edit/5

        [HttpPost]public ActionResult Edit(Student student){try{if (ModelState.IsValid){
                    studentRepository.UpdateStudent(student);
                    studentRepository.Save();return RedirectToAction("Index");}}catch (DataException){//Log the error (add a varia
```

```
ble name after DataException)ModelState.AddModelError("", "Unable
 to save changes. Try again, and if the problem persists see your
 system administrator.");}return View(student);}

        //// GET: /Student/Delete/5

        public ActionResult Delete(int id, bool? saveChangesError
){if (saveChangesError.GetValueOrDefault()){ViewBag.ErrorMessage
= "Unable to save changes. Try again, and if the problem persists
 see your system administrator.";}Student student = studentReposi
tory.GetStudentByID(id);return View(student);}


        //// POST: /Student/Delete/5

        [HttpPost, ActionName("Delete")]public ActionResult Delet
eConfirmed(int id){try{Student student = studentRepository.GetStu
dentByID(id);
                studentRepository.DeleteStudent(id);
                studentRepository.Save();}catch (DataException){/
/Log the error (add a variable name after DataException)return Re
directToAction("Delete",new System.Web.Routing.RouteValueDictiona
ry {{ "id", id },{ "saveChangesError", true } });}return Redirect
ToAction("Index");}

        protected override void Dispose(bool disposing){
            studentRepository.Dispose();base.Dispose(disposing);}
}}
```

The controller now declares a class variable for an object that implements the `IStudentRepository` interface instead of the context class:

```
        private IStudentRepository studentRepository;
```

The default constructor creates a new context instance, and an optional constructor allows the caller to pass in a context instance.

```
public StudentController(){this.studentRepository = new StudentRe
pository(new SchoolContext());}

public StudentController(IStudentRepository studentRepository){th
is.studentRepository = studentRepository;}
```

(If you were using *dependency injection*, or DI, you wouldn't need the default constructor because the DI software would ensure that the correct repository object would always be provided.)

In the CRUD methods, the repository is now called instead of the context:

```
var students = from s in studentRepository.GetStudents()select s;
```

```
Student student = studentRepository.GetStudentByID(id);
```

```
studentRepository.InsertStudent(student);
studentRepository.Save();
```

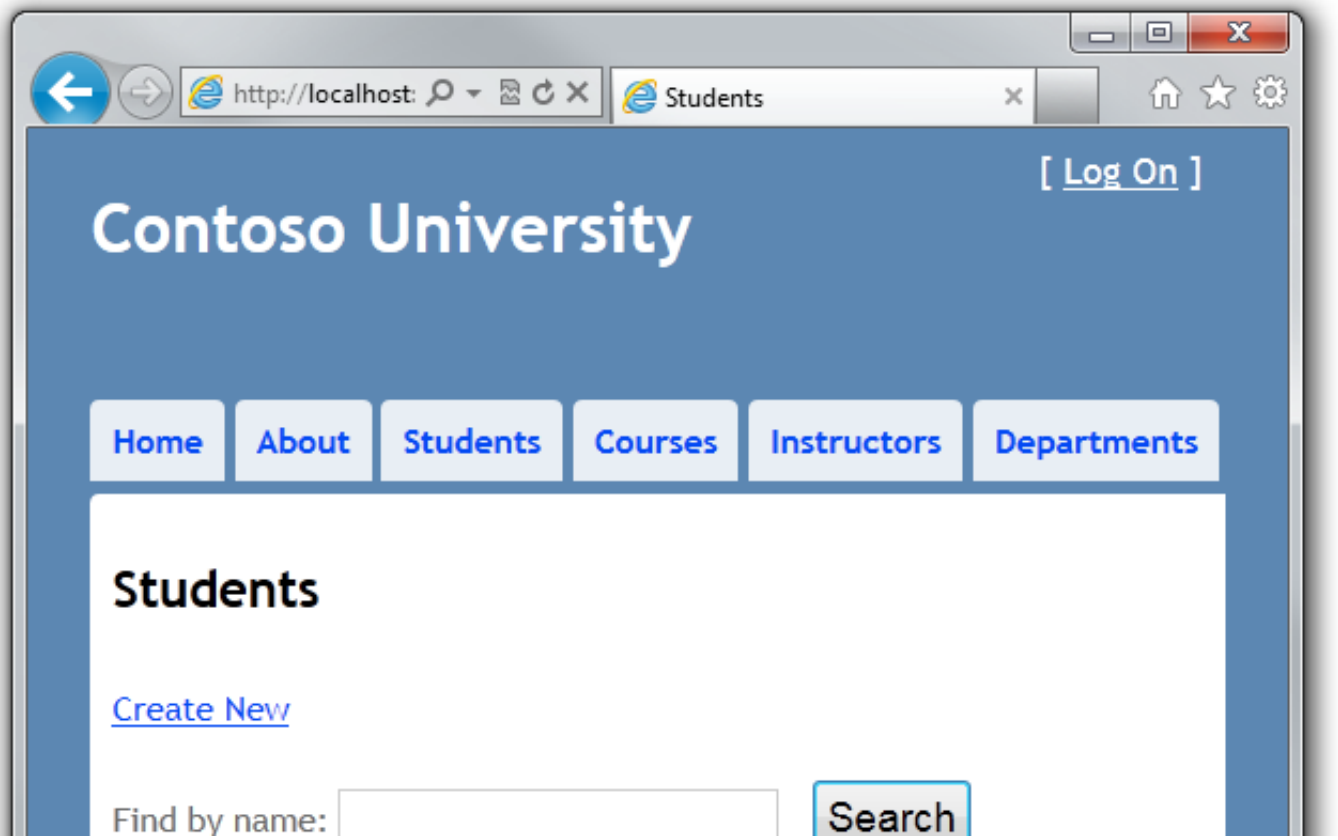
```
studentRepository.UpdateStudent(student);
studentRepository.Save();
```

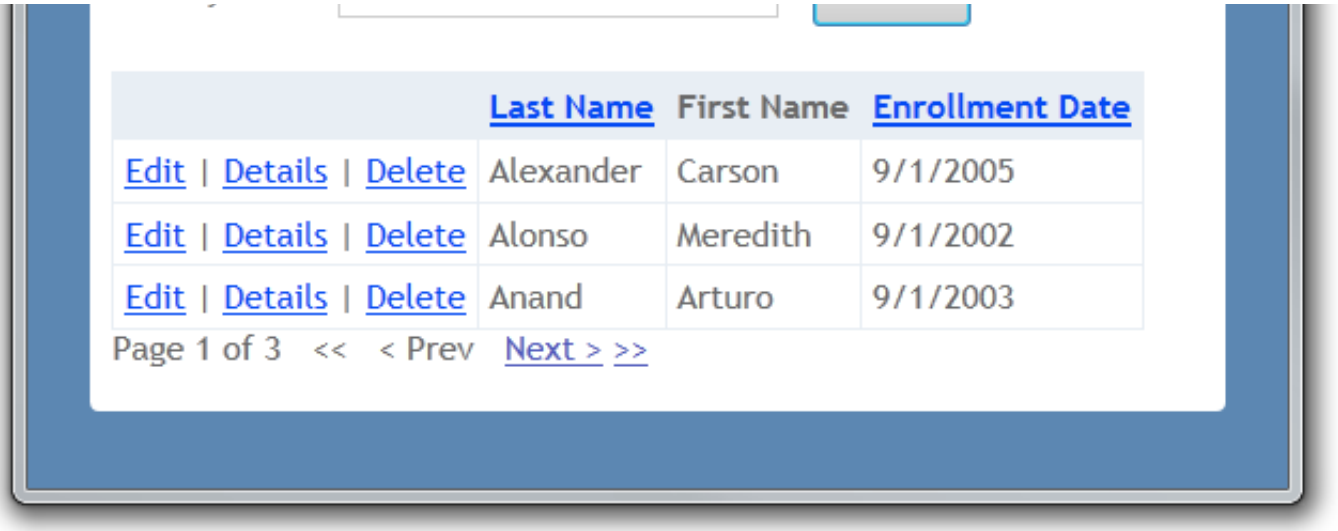
```
studentRepository.DeleteStudent(id);
studentRepository.Save();
```

And the `Dispose` method now disposes the repository instead of the context:

```
studentRepository.Dispose();
```

Run the site and click the **Students** tab.





The page looks and works the same as it did before you changed the code to use the repository, and the other Student pages also work the same. However, there's an important difference in the way the `Index` method of the controller does filtering and ordering. The original version of this method contained the following code:

```
var students = from s in context.Studentsselect s;if (!String.IsN
ullOrEmpty(searchString)){
    students = students.Where(s => s.LastName.ToUpper().Contains(
searchString.ToUpper())|| s.FirstMidName.ToUpper().Contains(searc
hString.ToUpper()));}
```

In the original version of the code, `students` is typed as an `IQueryable` object. The query isn't sent to the database until it's converted into a collection using a method such as `ToList`, which means that this `Where` method becomes a `WHERE` clause in the SQL query and is processed by the database. That in turn means that only the selected entities are returned by the database. However, as a result of changing `context.Students` to `studentRepository.GetStudents()`, the `students` variable after this statement is an `IEnumerable` collection that includes all students in the database. The end result of applying the `Where` method is the same, but now the work is done in memory on the web server and not by the database. For large volumes of data, this is likely to be inefficient. The following section shows how to implement repository methods that enable you to specify that this work should be done by the database.

You've now created an abstraction layer between the controller and the Entity Framework database context. If you were going to perform automated unit testing with this application, you could create an alternative repository class in a unit test project that implements `IStudentRepository`. Instead of calling the context to read and write data, this mock repository class could manipulate in-memory collections in order to test controller functions.

Implementing a Generic Repository and a Unit of Work Class

Creating a repository class for each entity type could result in a lot of redundant code, and it could result in

partial updates. For example, suppose you have to update two different entity types as part of the same transaction. If each uses a separate database context instance, one might succeed and the other might fail. One way to minimize redundant code is to use a generic repository, and one way to ensure that all repositories use the same database context (and thus coordinate all updates) is to use a unit of work class.

In this section of the tutorial, you'll create a `GenericRepository` class and a `UnitOfWork` class, and use them in the `Course` controller to access both the `Department` and the `Course` entity sets. As explained earlier, to keep this part of the tutorial simple, you aren't creating interfaces for these classes. But if you were going to use them to facilitate TDD, you'd typically implement them with interfaces the same way you did the `Student` repository.

Creating a Generic Repository

In the *DAL* folder, create *GenericRepository.cs* and replace the existing code with the following code:

```
using System;using System.Collections.Generic;using System.Linq;u
sing System.Data;using System.Data.Entity;using ContosoUniversity
.Models;using System.Linq.Expressions;

namespace ContosoUniversity.DAL
{public class GenericRepository<TEntity> where TEntity : class{in
ternal SchoolContext context;internal DbSet<TEntity> dbSet;

        public GenericRepository(SchoolContext context){this.cont
ext = context;this.dbSet = context.Set<TEntity>();}

        public virtual IEnumerable<TEntity> Get(Expression<Func<T
Entity, bool>> filter = null,Func<IQueryable<TEntity>, IOrderedQu
eryable<TEntity>> orderBy = null,string includeProperties = ""){I
Queryable<TEntity> query = dbSet;

            if (filter != null){
                query = query.Where(filter);}

            foreach (var includeProperty in includeProperties.Spl
it(new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries)){
                query = query.Include(includeProperty);}

            if (orderBy != null){return orderBy(query).ToList();}
else{return query.ToList();}}

        public virtual TEntity GetByID(object id){return dbSet.Fi
nd(id);}
```

```
        public virtual void Insert(TEntity entity){
            dbSet.Add(entity);}

        public virtual void Delete(object id){TEntity entityToDel
ete = dbSet.Find(id);Delete(entityToDelete);}

        public virtual void Delete(TEntity entityToDelete){if (co
ntext.Entry(entityToDelete).State == EntityState.Detached){
                dbSet.Attach(entityToDelete);}
            dbSet.Remove(entityToDelete);}

        public virtual void Update(TEntity entityToUpdate){
            dbSet.Attach(entityToUpdate);
            context.Entry(entityToUpdate).State = EntityState.Mod
ified;}}}
```

Class variables are declared for the database context and for the entity set that the repository is instantiated for:

```
internal SchoolContext context;internal DbSet dbSet;
```

The constructor accepts a database context instance and initializes the entity set variable:

```
public GenericRepository(SchoolContext context){this.context = co
ntext;this.dbSet = context.Set();}
```

The `Get` method uses lambda expressions to allow the calling code to specify a filter condition and a column to order the results by, and a string parameter lets the caller provide a comma-delimited list of navigation properties for eager loading:

```
public virtual IEnumerable<TEntity> Get(Expression<Func<TEntity,
bool>> filter = null,Func<IQueryable<TEntity>, IOrderedQueryable<
TEntity>> orderBy = null,string includeProperties = "")
```

The code `Expression<Func<TEntity, bool>> filter` means the caller will provide a lambda expression based on the `TEntity` type, and this expression will return a Boolean value. For example, if the repository is instantiated for the `Student` entity type, the code in the calling method might specify `student => student.LastName == "Smith"` for the `filter` parameter.

The code `Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy` also means the caller will provide a lambda expression. But in this case, the input to the

expression is an `IQueryable` object for the `TEntity` type. The expression will return an ordered version of that `IQueryable` object. For example, if the repository is instantiated for the `Student` entity type, the code in the calling method might specify `q => q.OrderBy(s => s.LastName)` for the `orderBy` parameter.

The code in the `Get` method creates an `IQueryable` object and then applies the filter expression if there is one:

```
IQueryable<TEntity> query = dbSet;

if (filter != null){
    query = query.Where(filter);}
```

Next it applies the eager-loading expressions after parsing the comma-delimited list:

```
foreach (var includeProperty in includeProperties.Split(new char[
] { ',' }, StringSplitOptions.RemoveEmptyEntries)){
    query = query.Include(includeProperty);}
```

Finally, it applies the `orderBy` expression if there is one and returns the results; otherwise it returns the results from the unordered query:

```
if (orderBy != null){return orderBy(query).ToList();}else{return
query.ToList();}
```

When you call the `Get` method, you could do filtering and sorting on the `IEnumerable` collection returned by the method instead of providing parameters for these functions. But the sorting and filtering work would then be done in memory on the web server. By using these parameters, you ensure that the work is done by the database rather than the web server. An alternative is to create derived classes for specific entity types and add specialized `Get` methods, such as `GetStudentsInNameOrder` or `GetStudentsByName`. However, in a complex application, this can result in a large number of such derived classes and specialized methods, which could be more work to maintain.

The code in the `GetByID`, `Insert`, and `Update` methods is similar to what you saw in the non-generic repository. (You aren't providing an eager loading parameter in the `GetByID` signature, because you can't do eager loading with the `Find` method.)

Two overloads are provided for the `Delete` method:

```
public virtual void Delete(object id){TEntity entityToDelete = db
Set.Find(id);
    dbSet.Remove(entityToDelete);}
```

```
public virtual void Delete(TEntity entityToDelete){
    context.Entry(entityToDelete).State = EntityState.Deleted;}
```

One of these lets you pass in just the ID of the entity to be deleted, and one takes an entity instance. As you saw in the Handling Concurrency tutorial, for concurrency handling you need a `Delete` method that takes an entity instance that includes the original value of a tracking property.

This generic repository will handle typical CRUD requirements. When a particular entity type has special requirements, such as more complex filtering or ordering, you can create a derived class that has additional methods for that type.

Creating the Unit of Work Class

The unit of work class serves one purpose: to make sure that when you use multiple repositories, they share a single database context. That way, when a unit of work is complete you can call the `SaveChanges` method on that instance of the context and be assured that all related changes will be coordinated. All that the class needs is a `Save` method and a property for each repository. Each repository property returns a repository instance that has been instantiated using the same database context instance as the other repository instances.

In the *DAL* folder, create a class file named *UnitOfWork.cs* and replace the existing code with the following code:

```
using System;using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{public class UnitOfWork : IDisposable{private SchoolContext cont
ext = new SchoolContext();private GenericRepository<Department> d
epartmentRepository;private GenericRepository<Course> courseRepos
itory;

        public GenericRepository<Department> DepartmentRepository
{get{

                if (this.departmentRepository == null){this.depar
tmentRepository = new GenericRepository<Department>(context);}ret
urn departmentRepository;}}

        public GenericRepository<Course> CourseRepository{get{

                if (this.courseRepository == null){this.courseRep
ository = new GenericRepository<Course>(context);}return courseRe
pository;}}
```

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

        private bool disposed = false;

        protected virtual void Dispose(bool disposing){if (!this.
disposed){if (disposing){
                    context.Dispose();}}this.disposed = true;}

        public void Dispose(){Dispose(true);
            GC.SuppressFinalize(this);}}}
```

The code creates class variables for the database context and each repository. For the `context` variable, a new context is instantiated:

```
private SchoolContext context = new SchoolContext();private Gener
icRepository<Department> departmentRepository;private GenericRepo
sitory<Course> courseRepository;
```

Each repository property checks whether the repository already exists. If not, it instantiates the repository, passing in the context instance. As a result, all repositories share the same context instance.

```
public GenericRepository<Department> DepartmentRepository{get{

        if (this.departmentRepository == null){this.departmentRep
ository = new GenericRepository<Department>(context);}return depa
rtmentRepository;}}
```

The `Save` method calls `SaveChanges` on the database context.

Like any class that instantiates a database context in a class variable, the `UnitOfWork` class implements `IDisposable` and disposes the context.

Changing the Course Controller to use the UnitOfWork Class and Repositories

Replace the code you currently have in *CourseController.cs* with the following code:

```
using System;using System.Collections.Generic;using System.Data;u
sing System.Data.Entity;using System.Linq;using System.Web;using
System.Web.Mvc;using ContosoUniversity.Models;using ContosoUniver
sity.DAL;
```

```
namespace ContosoUniversity.Controllers{public class CourseContro
ller : Controller{private UnitOfWork unitOfWork = new UnitOfWork(
);

        //// GET: /Course/

        public ViewResult Index(){var courses = unitOfWork.Course
Repository.Get(includeProperties: "Department");return View(cours
es.ToList());}

        //// GET: /Course/Details/5

        public ViewResult Details(int id){Course course = unitOfW
ork.CourseRepository.GetByID(id);return View(course);}

        //// GET: /Course/Create

        public ActionResult Create(){PopulateDepartmentsDropDownL
ist();return View();}

        [HttpPost]public ActionResult Create(Course course){try{i
f (ModelState.IsValid){
                    unitOfWork.CourseRepository.Insert(course);
                    unitOfWork.Save();return RedirectToAction("In
dex");}}catch (DataException){//Log the error (add a variable nam
e after DataException)ModelState.AddModelError("", "Unable to sav
e changes. Try again, and if the problem persists, see your syste
m administrator.");}PopulateDepartmentsDropDownList(course.Depart
mentID);return View(course);}

        public ActionResult Edit(int id){Course course = unitOfWo
rk.CourseRepository.GetByID(id);PopulateDepartmentsDropDownList(c
ourse.DepartmentID);return View(course);}

        [HttpPost]public ActionResult Edit(Course course){try{if
(ModelState.IsValid){
                    unitOfWork.CourseRepository.Update(course);
                    unitOfWork.Save();return RedirectToAction("In
dex");}}catch (DataException){//Log the error (add a variable nam
e after DataException)ModelState.AddModelError("", "Unable to sav
e changes. Try again, and if the problem persists, see your syste
m administrator.");}PopulateDepartmentsDropDownList(course.Depart
mentID);return View(course);}
```

```
        private void PopulateDepartmentsDropDownList(object selec
tedDepartment = null){var departmentsQuery = unitOfWork.Departmen
tRepository.Get(
                orderBy: q => q.OrderBy(d => d.Name));ViewBag.Dep
artmentID = new SelectList(departmentsQuery, "DepartmentID", "Nam
e", selectedDepartment);}

        //// GET: /Course/Delete/5

        public ActionResult Delete(int id){Course course = unitOf
Work.CourseRepository.GetByID(id);return View(course);}

        //// POST: /Course/Delete/5

        [HttpPost, ActionName("Delete")]public ActionResult Delet
eConfirmed(int id){Course course = unitOfWork.CourseRepository.Ge
tByID(id);
            unitOfWork.CourseRepository.Delete(id);
            unitOfWork.Save();return RedirectToAction("Index");}

        protected override void Dispose(bool disposing){
            unitOfWork.Dispose();base.Dispose(disposing);}}}
```

This code adds a class variable for the `UnitOfWork` class. (If you were using interfaces here, you wouldn't initialize the variable here; instead, you'd implement a pattern of two constructors just as you did for the `Student` repository.)

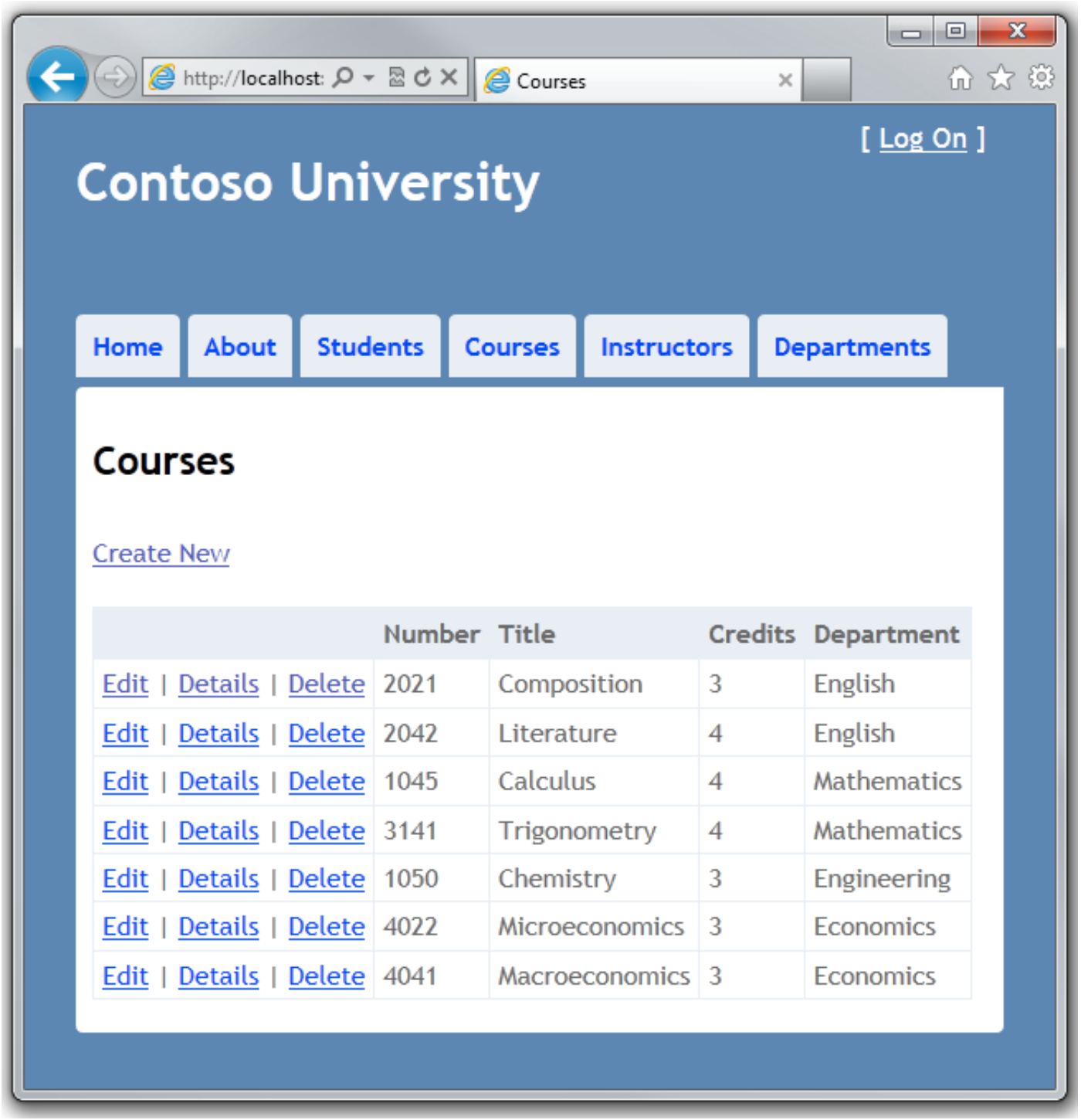
```
private UnitOfWork unitOfWork = new UnitOfWork();
```

In the rest of the class, all references to the database context are replaced by references to the appropriate repository, using `UnitOfWork` properties to access the repository. The `Dispose` method disposes the `UnitOfWork` instance.

```
var courses = unitOfWork.CourseRepository.Get(includeProperties:
"Department");// ...Course course = unitOfWork.CourseRepository.G
etByID(id);// ...
unitOfWork.CourseRepository.Insert(course);
unitOfWork.Save();// ...Course course = unitOfWork.CourseReposito
ry.GetByID(id);// ...
unitOfWork.CourseRepository.Update(course);
unitOfWork.Save();// ...var departmentsQuery = unitOfWork.Departm
entRepository.Get(
```

```
    orderBy: q => q.OrderBy(d => d.Name));// ...Course course = u
nitOfWork.CourseRepository.GetByID(id);// ...
unitOfWork.CourseRepository.Delete(id);
unitOfWork.Save();// ...
unitOfWork.Dispose();
```

Run the site and click the **Courses** tab.



The page looks and works the same as it did before your changes, and the other Course pages also work the same.

You have now implemented both the repository and unit of work patterns. You have used lambda

You have now implemented both the repository and unit of work patterns. You have used lambda expressions as method parameters in the generic repository. For more information about how to use these expressions with an `IQueryable` object, see IQueryable(T) Interface (System.Linq) in the MSDN Library. In the next tutorial you'll learn how to handle some advanced scenarios.

Links to other Entity Framework resources can be found at the end of the last tutorial in this series.

www.asp.net

http://www.asp.net/entity-framework/tutorials/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

http://goo.gl/fP7OC