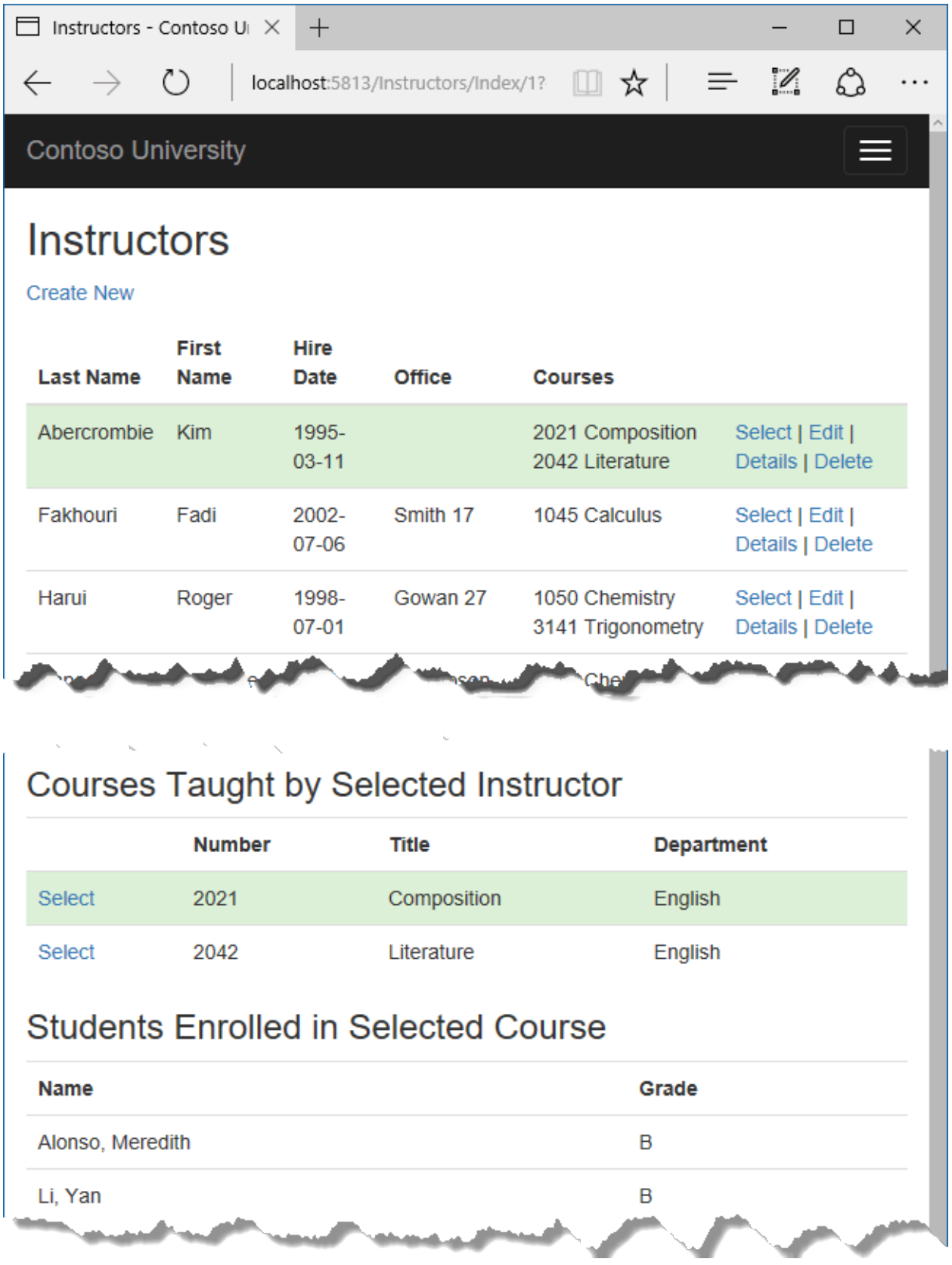


This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity. The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. You'll use eager loading for the `OfficeAssignment` entities. As explained earlier, eager loading is typically more efficient when you need the

related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.

- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. You'll use eager loading for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity set is displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship. You'll use separate queries for `Enrollment` entities and their related `Student` entities.

Create a view model for the Instructor Index view

The Instructors page shows data from three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

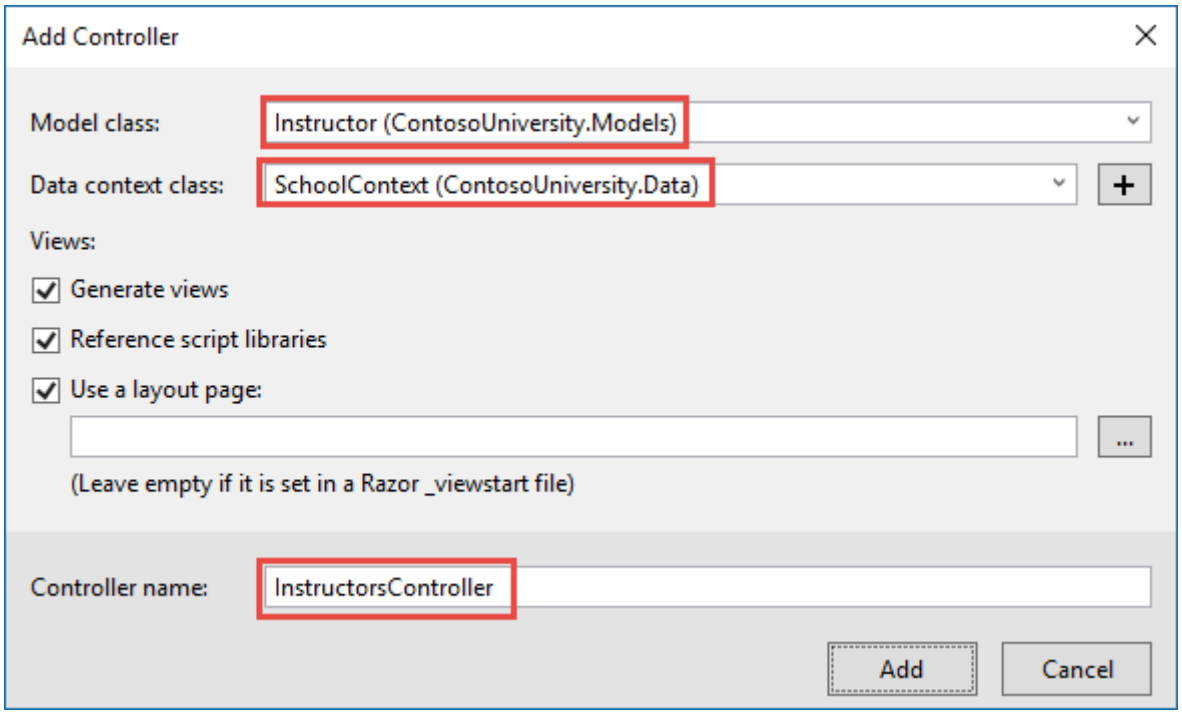
In the *SchoolViewModels* folder, create `InstructorIndexData.cs` and replace the existing code with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Create the Instructor controller and views

Create an Instructors controller with EF read/write actions as shown in the following illustration:



Open `InstructorsController.cs` and add a using statement for the ViewModels namespace:

C#

```
using ContosoUniversity.Models.SchoolViewModels;
```

Replace the Index method with the following code to do eager loading of related data and put it in the view model.

C#

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
```

```
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s =>
s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}
```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.CourseAssignments` navigation properties. Within the `CourseAssignments` property, the `Course` property is loaded, and within that, the `Enrollments` and `Department` properties are loaded, and within each `Enrollment` entity the `Student` property is loaded.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Since the view always requires the `OfficeAssignment` entity, it's more efficient to fetch that in the same query. Course entities are required when an instructor is selected in the web page, so a single query is better than multiple queries only if the page is displayed more often with a course selected than without.

The code repeats `CourseAssignments` and `Course` because you need two properties from `Course`. The first string of `ThenInclude` calls gets `CourseAssignment.Course`, `Course.Enrollments`, and `Enrollment.Student`.

You can read more about including multiple levels of related data here.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

At that point in the code, another `ThenInclude` would be for navigation properties of `Student`, which you don't need. But calling `Include` starts over with `Instructor` properties, so you have to go through the chain again, this time specifying `Course.Department` instead of `Course.Enrollments`.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The following code executes when an instructor was selected. The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

C#

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single Instructor entity being returned. The `Single` method converts the collection into a single `Instructor` entity, which gives you access to that entity's `CourseAssignments` property. The `CourseAssignments` property contains `CourseAssignment` entities, from which you want only the related `Course` entities.

You use the `Single` method on a collection when you know the collection will have only one item. The `Single` method throws an exception if the collection passed to it's empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a null reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the Where condition instead of calling the `Where` method separately:

C#

```
.Single(i => i.ID == id.Value)
```

Instead of:

C#

```
.Where(i => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the Enrollment entities from that course's `Enrollments` navigation property.

C#

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
```

```
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Tracking vs no-tracking

No-tracking queries are useful when the results are used in a read-only scenario. They're generally quicker to execute because there's no need to set up the change tracking information. If the entities retrieved from the database don't need to be updated, then a no-tracking query is likely to perform better than a tracking query.

In some cases a tracking query is more efficient than a no-tracking query. For more information, see Tracking vs. No-Tracking Queries.

Modify the Instructor Index view

In `Views/Instructors/Index.cshtml`, replace the template code with the following code. The changes are highlighted.

CSHTML

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructors)
        {
```

```
            string selectedRow = "";
            if (item.ID == (int?)ViewData["InstructorID"])
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @foreach (var course in item.CourseAssignments)
                    {
                        @course.Course.CourseID @course.Course.Title <br />
                    }
                </td>
                <td>
                    <a asp-action="Index" asp-route-id="@item.ID">Select</a>
|
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. (Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.)

CSHTML

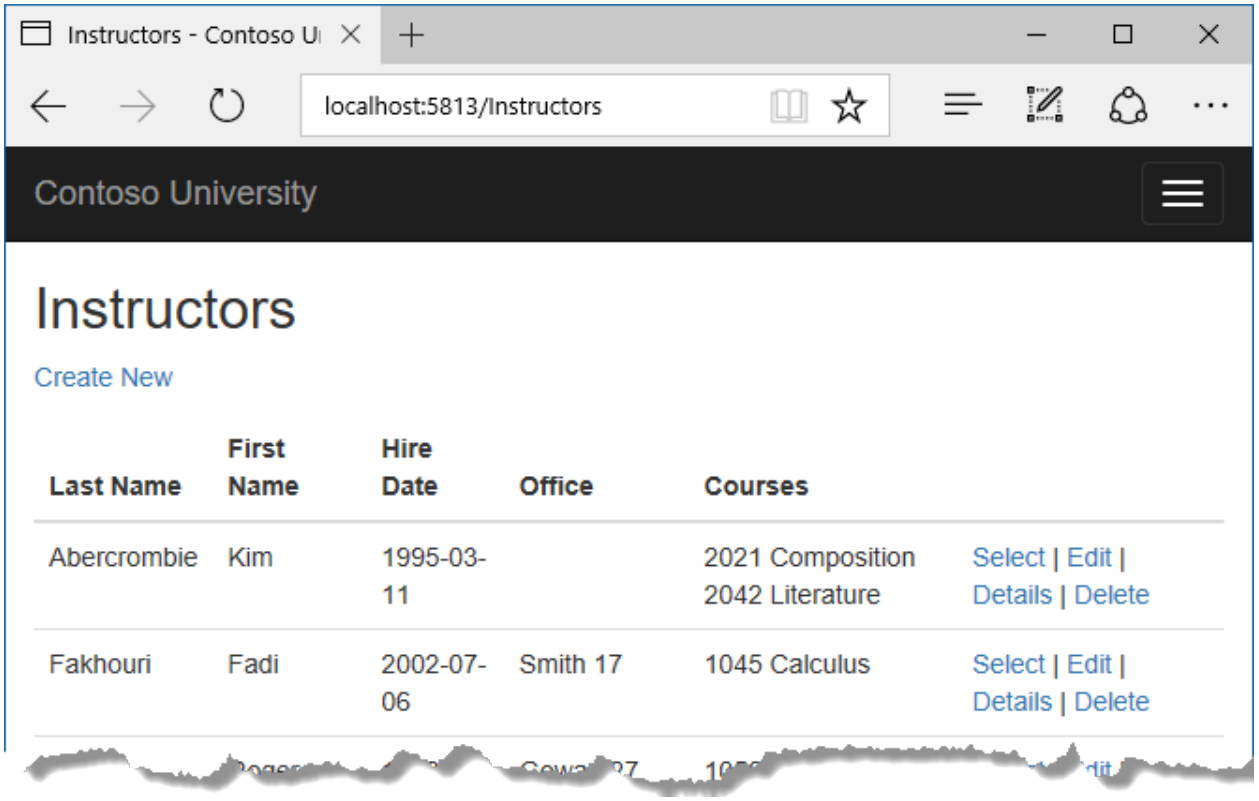
```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- Added a **Courses** column that displays courses taught by each instructor. For more information, see the Explicit line transition section of the Razor syntax article.
- Added code that conditionally adds a Bootstrap CSS class to the `tr` element of the selected instructor. This class sets a background color for the selected row.
- Added a new hyperlink labeled **Select** immediately before the other links in each row, which causes the selected instructor's ID to be sent to the `Index` method.

CSHTML

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the app and select the **Instructors** tab. The page displays the Location property of related OfficeAssignment entities and an empty table cell when there's no related OfficeAssignment entity.



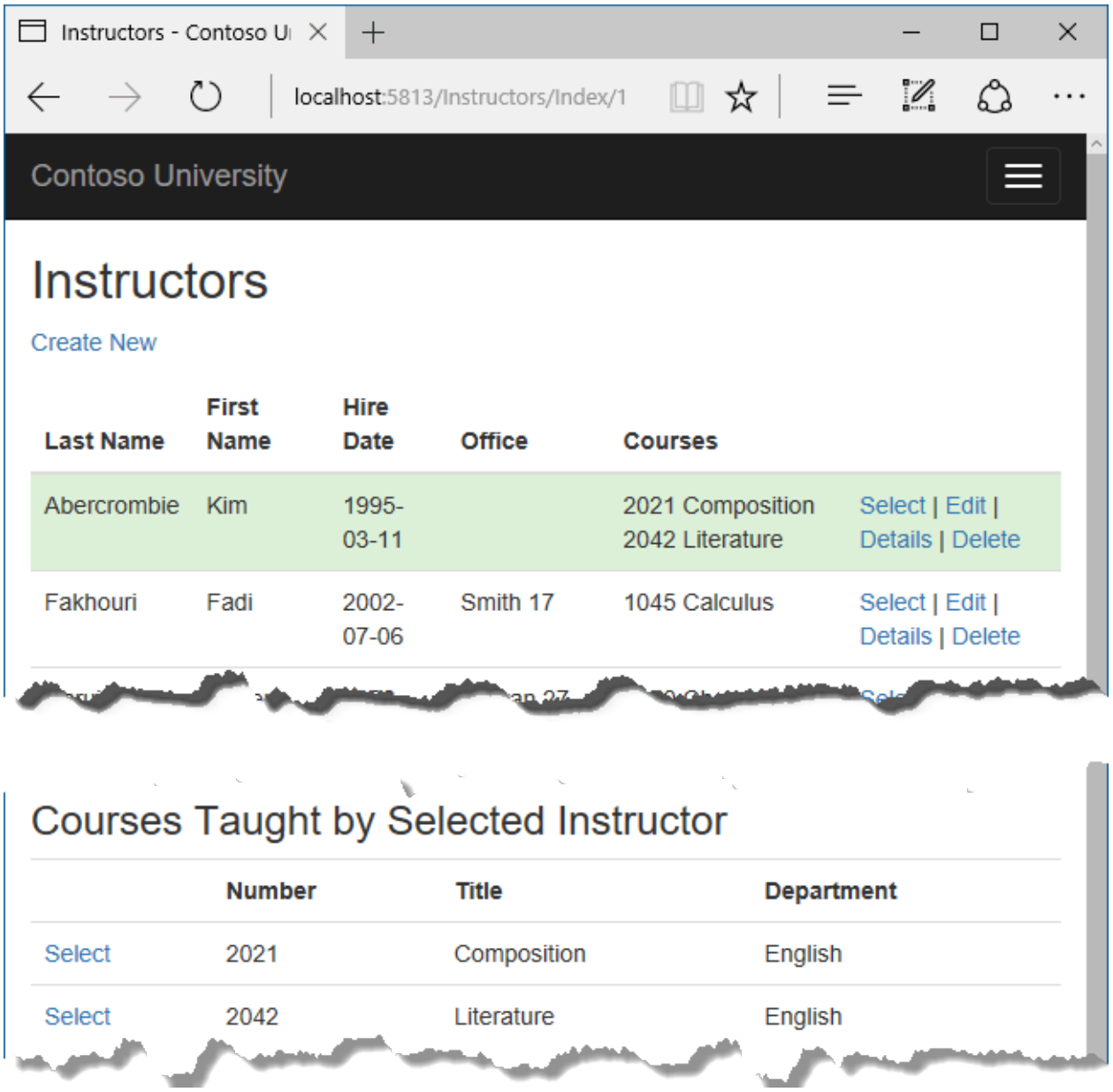
In the `Views/Instructors/Index.cshtml` file, after the closing table element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

CSHTML

```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID =
item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a **Select** hyperlink that sends the ID of the selected course to the `Index` action method.

Refresh the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.



After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

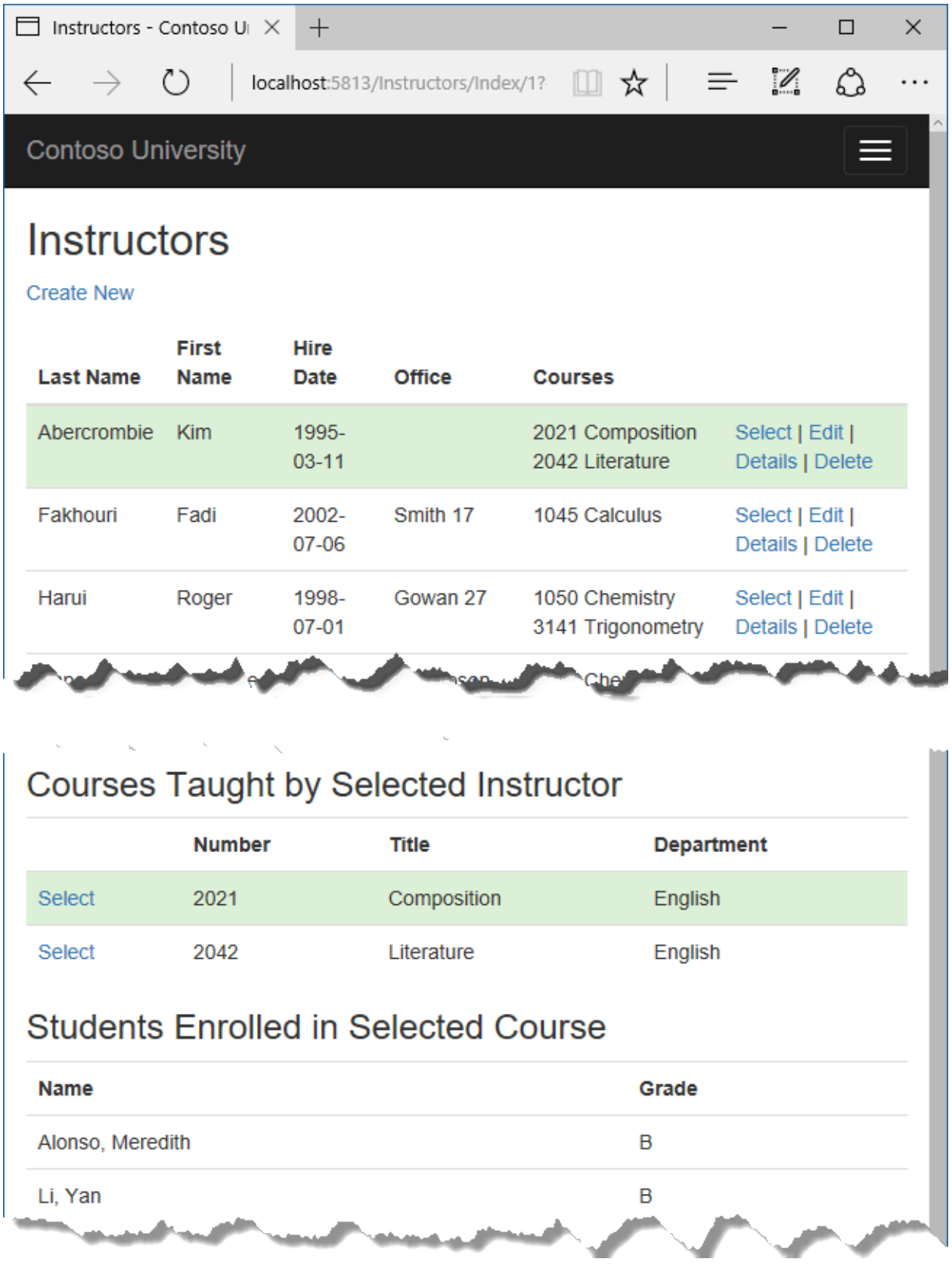
CSHTML

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
```

```
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the `Enrollments` property of the view model in order to display a list of students enrolled in the course.

Refresh the page again and select an instructor. Then select a course to see the list of enrolled students and their grades.



About explicit loading

When you retrieved the list of instructors in `InstructorsController.cs`, you specified eager loading for the `CourseAssignments` navigation property.

Suppose you expected users to only rarely want to see enrollments in a selected instructor and course. In that case, you might want to load the enrollment data only if

it's requested. To see an example of how to do explicit loading, replace the `Index` method with the following code, which removes eager loading for `Enrollments` and loads that property explicitly. The code changes are highlighted.

C#

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s =>
s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID ==
courseID).Single();
        await _context.Entry(selectedCourse).Collection(x =>
x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x =>
x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

The new code drops the `ThenInclude` method calls for enrollment data from the code that retrieves instructor entities. It also drops `AsNoTracking`. If an instructor and course are selected, the highlighted code retrieves `Enrollment` entities for the selected course, and `Student` entities for each `Enrollment`.

Run the app, go to the Instructors Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Learned how to load related data
- ✓ Created a Courses page
- ✓ Created an Instructors page
- ✓ Learned about explicit loading

Advance to the next tutorial to learn how to update related data.

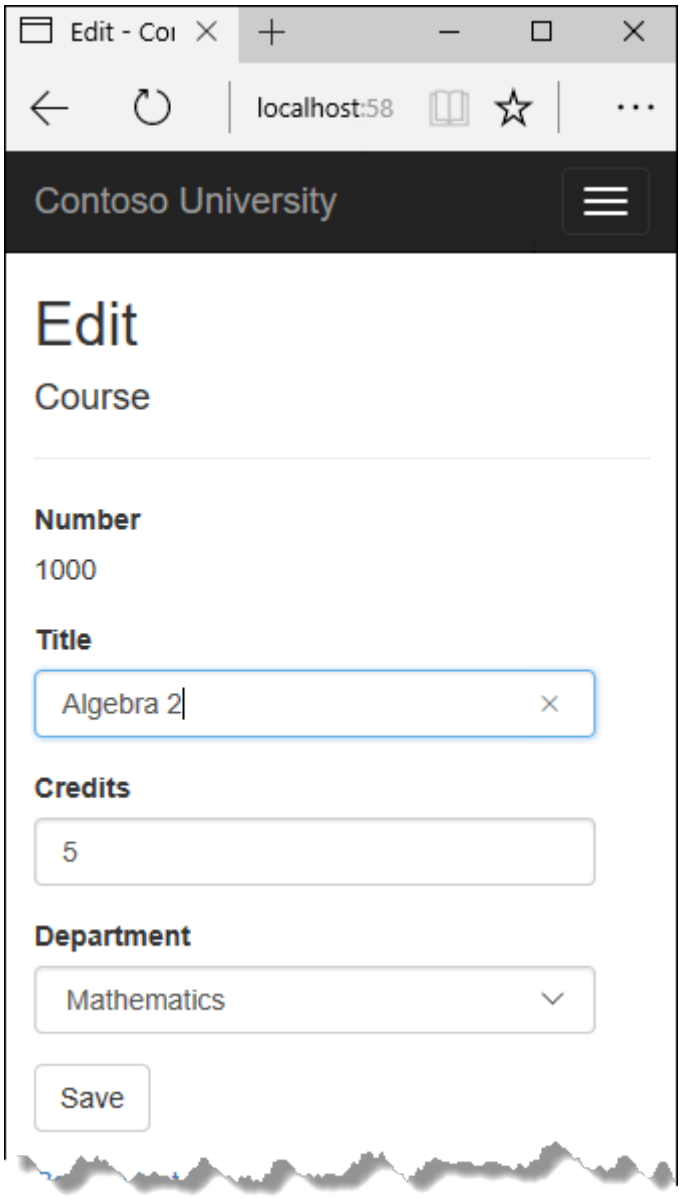
Update related data

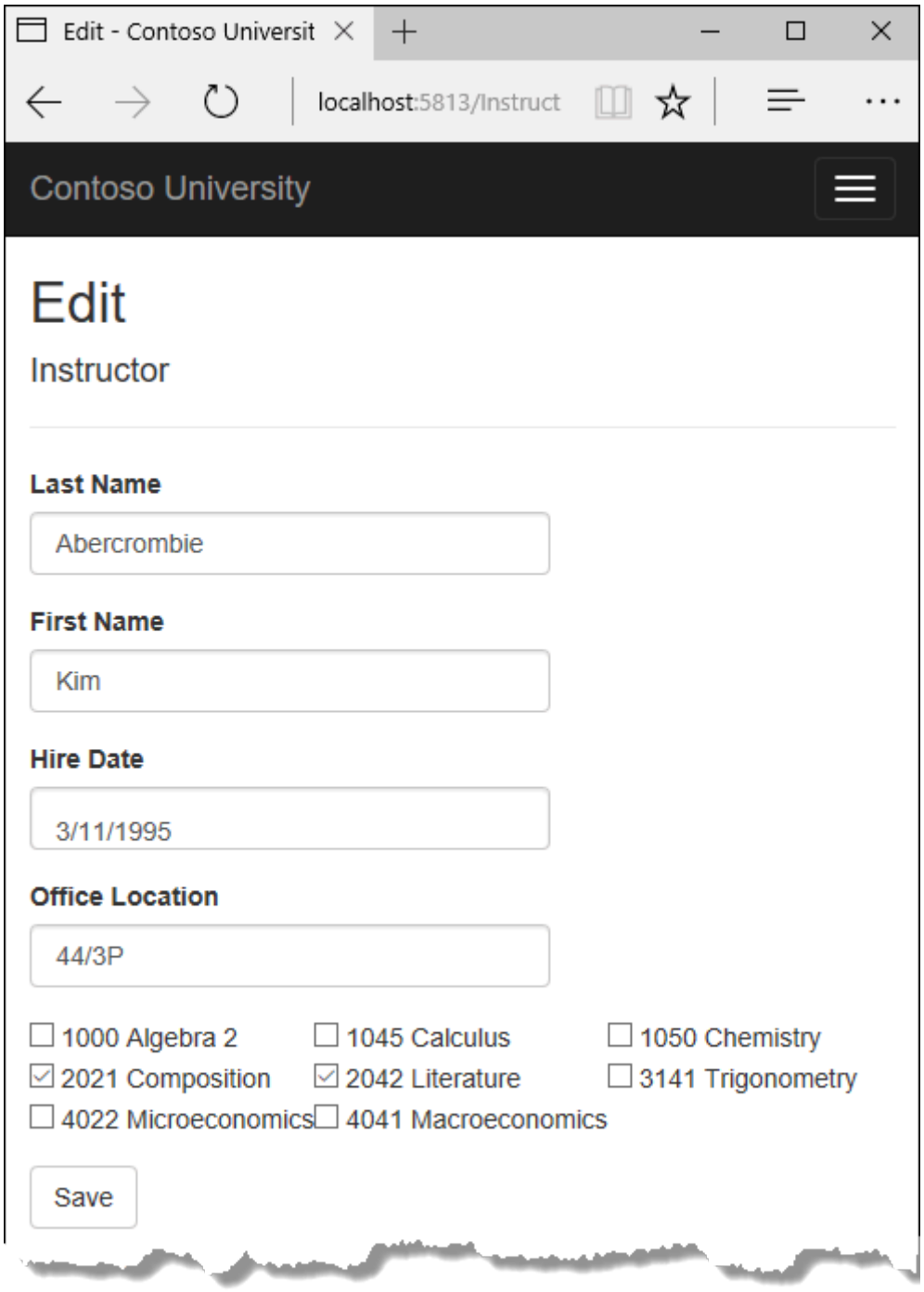
Tutorial: Update related data - ASP.NET MVC with EF Core

Article • 04/10/2024

In the previous tutorial you displayed related data; in this tutorial you'll update related data by updating foreign key fields and navigation properties.

The following illustrations show some of the pages that you'll work with.





In this tutorial, you:

- ✓ Customize Courses pages
- ✓ Add Instructors Edit page
- ✓ Add courses to Edit page
- ✓ Update Delete page
- ✓ Add office location and courses to Create page

Prerequisites

- Read related data

Customize Courses pages

When a new `Course` entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate `Department` entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In `CoursesController.cs`, delete the four Create and Edit methods and replace them with the following code:

C#

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
```

```
    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

C#

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .FirstOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}
```

After the `Edit` HttpPost method, create a new method that loads department info for the drop-down list.

C#

```
private void PopulateDepartmentsDropDownList(object selectedDepartment =
null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(),
"DepartmentID", "Name", selectedDepartment);
}
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in `ViewBag`. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name "DepartmentID" to the `<select>` tag helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named "DepartmentID".

The HttpGet `Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department isn't established yet:

C#

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

The HttpGet `Edit` method sets the selected item, based on the ID of the department that's already assigned to the course being edited:

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
```

```
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The HttpPost methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error. This ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

Add .AsNoTracking to Details and Delete methods

To optimize performance of the Course Details and Delete pages, add `AsNoTracking` calls in the `Details` and HttpGet `Delete` methods.

C#

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

C#

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
```

```
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

Modify the Course views

In `Views/Courses/Create.cshtml`, add a "Select Department" option to the **Department** drop-down list, change the caption from **DepartmentID** to **Department**, and add a validation message.

CSHTML

```
<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="DepartmentID" class="form-control" asp-
items="ViewBag.DepartmentID">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="DepartmentID" class="text-danger" />
</div>
```

In `Views/Courses/Edit.cshtml`, make the same change for the Department field that you just did in `Create.cshtml`.

Also in `Views/Courses/Edit.cshtml`, add a course number field before the **Title** field. Because the course number is the primary key, it's displayed, but it can't be changed.

CSHTML

```
<div class="form-group">
    <label asp-for="CourseID" class="control-label"></label>
    <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>
```

There's already a hidden field (`<input type="hidden">`) for the course number in the Edit view. Adding a `<label>` tag helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the **Edit** page.

In `Views/Courses/Delete.cshtml`, add a course number field at the top and change department ID to department name.

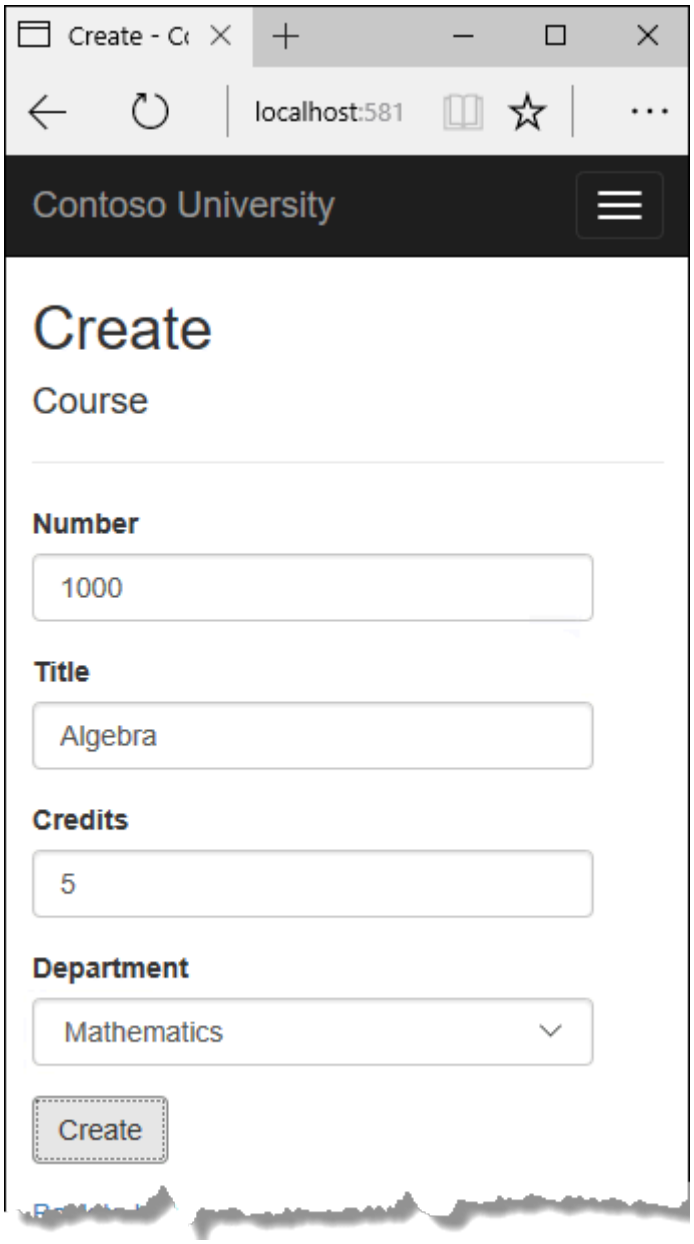
CSHTML

```
@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

In `Views/Courses/Details.cshtml`, make the same change that you just did for `Delete.cshtml`.

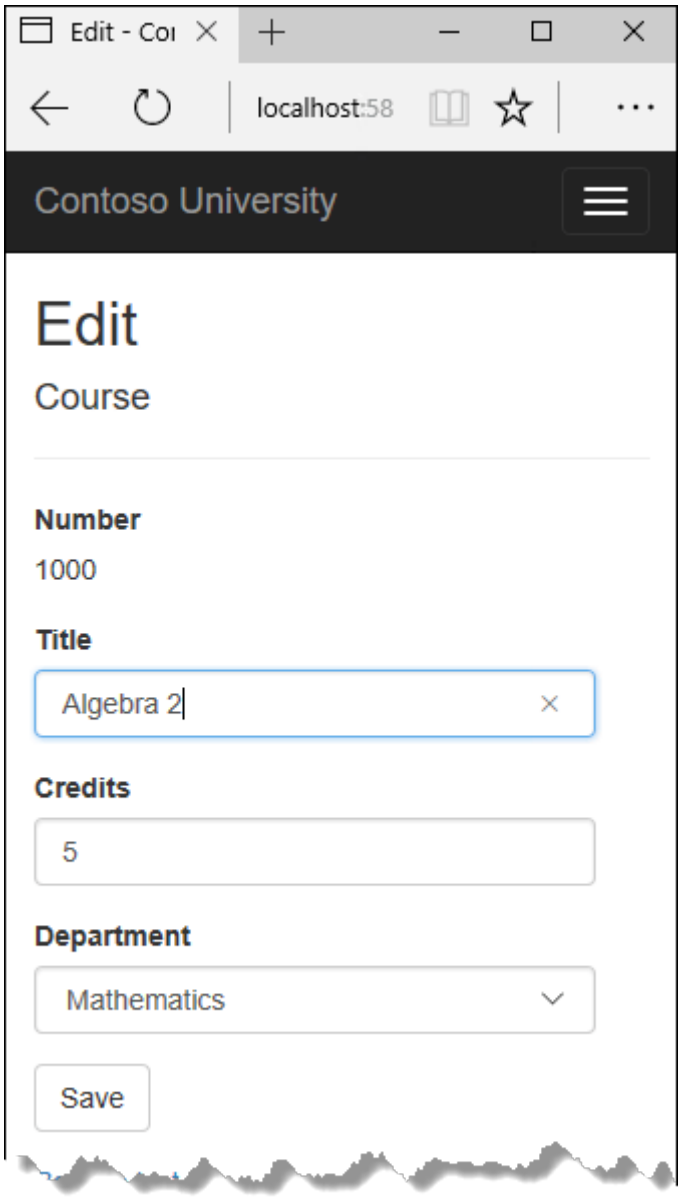
Test the Course pages

Run the app, select the **Courses** tab, click **Create New**, and enter data for a new course:



Click **Create**. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Click **Edit** on a course in the Courses Index page.



Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add Instructors Edit page

When you edit an instructor record, you want to be able to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity, which means your code has to handle the following situations:

- If the user clears the office assignment and it originally had a value, delete the `OfficeAssignment` entity.
- If the user enters an office assignment value and it originally was empty, create a new `OfficeAssignment` entity.
- If the user changes the value of an office assignment, change the value in an existing `OfficeAssignment` entity.

Update the Instructors controller

In `InstructorsController.cs`, change the code in the HttpGet `Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `AsNoTracking`:

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

Replace the HttpPost `Edit` method with the following code to handle office assignment updates:

C#

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .FirstOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
i.OfficeAssignment))
    {
```

```
        if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}
```

The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the HttpGet `Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property. This is the same as what you did in the HttpGet `Edit` method.
- Updates the retrieved `Instructor` entity with values from the model binder. The `TryUpdateModel` overload enables you to declare the properties you want to include. This prevents over-posting, as explained in the second tutorial.

  C#

  ```
  if (await TryUpdateModelAsync<Instructor>(
      instructorToUpdate,
      "",
      i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
  i.OfficeAssignment))
  ```

- If the office location is blank, sets the `Instructor.OfficeAssignment` property to null so that the related row in the `OfficeAssignment` table will be deleted.

  C#

```
if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Locatio
n))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

- Saves the changes to the database.

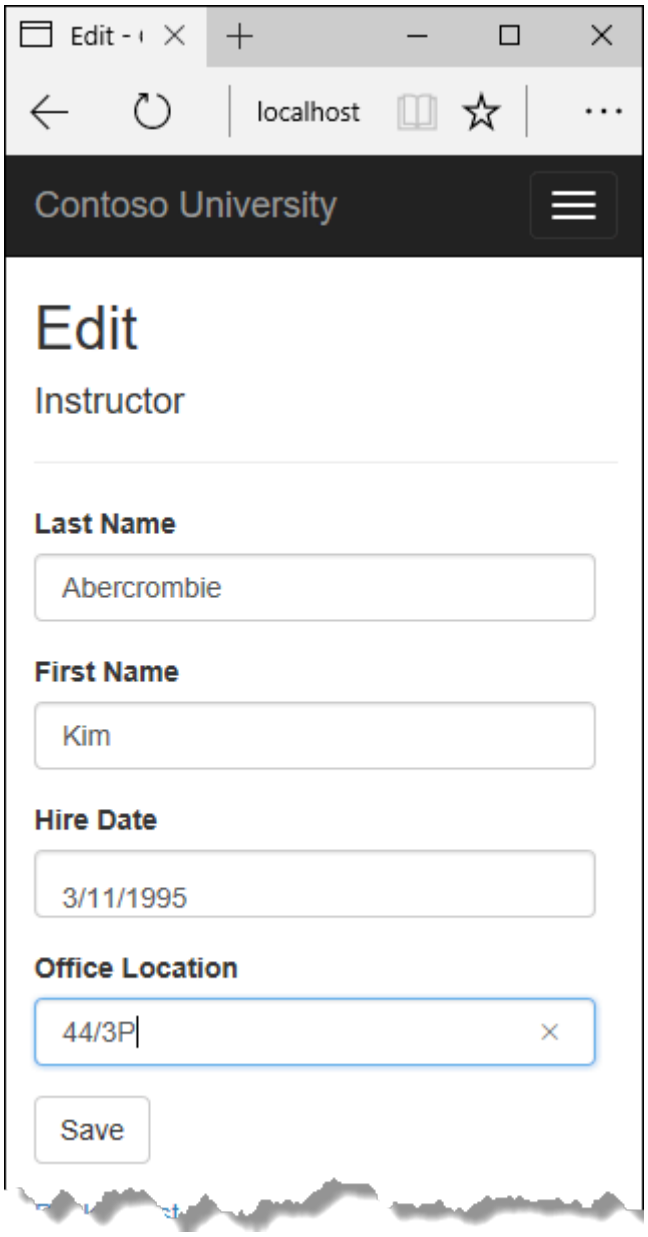
Update the Instructor Edit view

In `Views/Instructors/Edit.cshtml`, add a new field for editing the office location, at the end before the **Save** button:

CSHTML

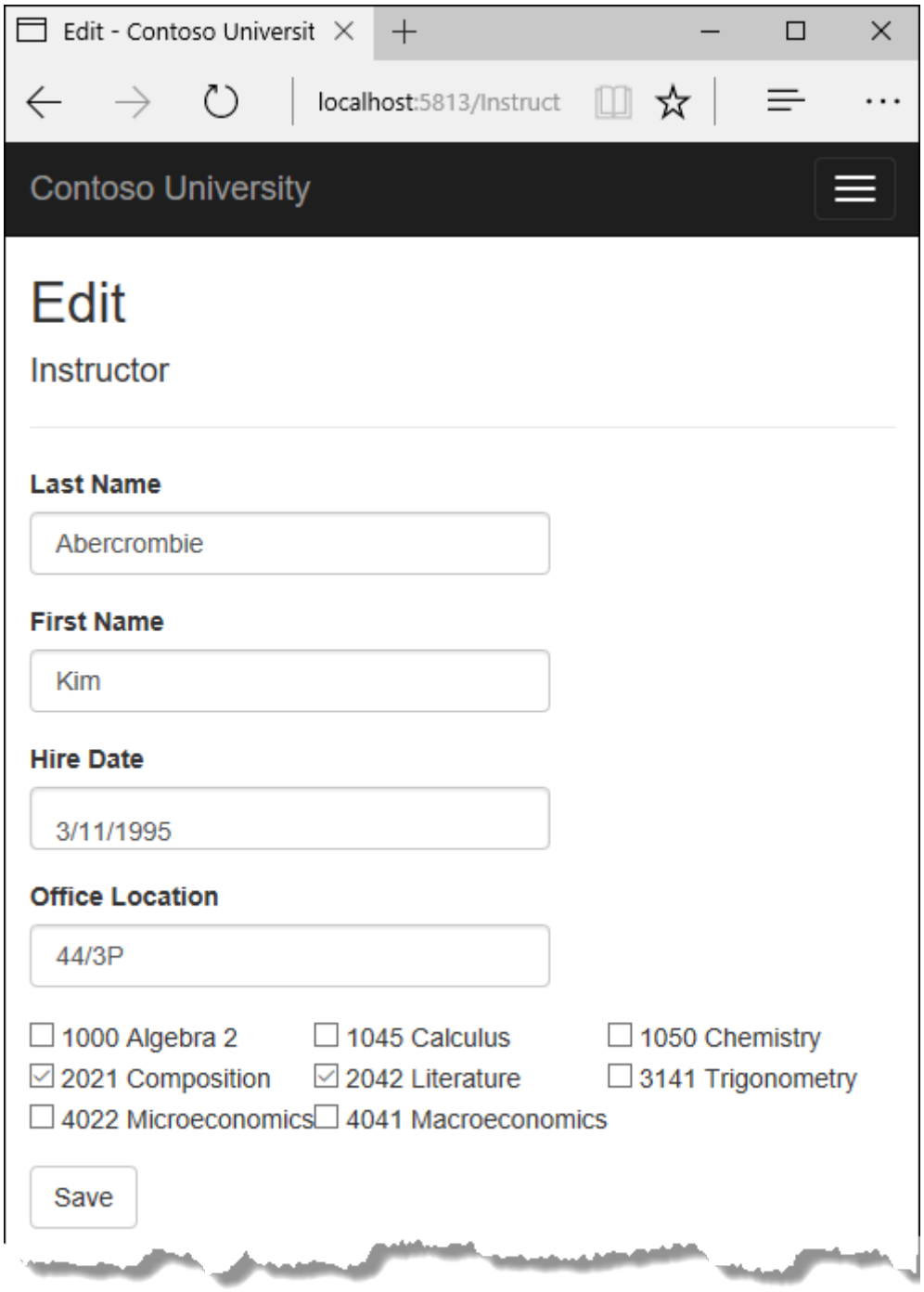
```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label">
</label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger"
/>
</div>
```

Run the app, select the **Instructors** tab, and then click **Edit** on an instructor. Change the **Office Location** and click **Save**.



Add courses to Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of checkboxes, as shown in the following screen shot:



The relationship between the `Course` and `Instructor` entities is many-to-many. To add and remove relationships, you add and remove entities to and from the `CourseAssignments` join entity set.

The UI that enables you to change which courses an instructor is assigned to is a group of checkboxes. A checkbox for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of checkboxes, you'll use a view model class.

Create `AssignedCourseData.cs` in the *SchoolViewModels* folder and replace the existing code with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In `InstructorsController.cs`, replace the HttpGet `Edit` method with the following code. The changes are highlighted.

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
```

```
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>
(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the checkbox array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all `Course` entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection. The `Assigned` property is set to true for courses the instructor is assigned to. The view will use this property to determine which checkboxes must be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the Instructor entity.

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
```

```
            .FirstOrDefaultAsync(m => m.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "",
            i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
i.OfficeAssignment))
        {
            if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(selectedCourses, instructorToUpdate);
            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateException /* ex */)
            {
                //Log the error (uncomment ex variable name and write a log.)
                ModelState.AddModelError("", "Unable to save changes. " +
                    "Try again, and if the problem persists, " +
                    "see your system administrator.");
            }
            return RedirectToAction(nameof(Index));
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(instructorToUpdate);
        return View(instructorToUpdate);
    }
```

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The method signature is now different from the HttpGet `Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore, you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the overload that requires explicit approval and `CourseAssignments` isn't in the include list.

If no checkboxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the checkbox for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection

in the navigation property.

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

If the checkbox for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
```

```
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

Update the Instructor views

In `Views/Instructors/Edit.cshtml`, add a **Courses** field with an array of checkboxes by adding the following code immediately after the `div` elements for the **Office** field and before the `div` element for the **Save** button.

> ⓘ Note
>
> When you paste the code in Visual Studio, line breaks might be changed in a way that breaks the code. If the code looks different after pasting, press Ctrl+Z one time to undo the automatic formatting. This will fix the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@:</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. With the block of new code selected, press Tab three

times to line up the new code with the existing code. This problem is fixed in Visual Studio 2019.

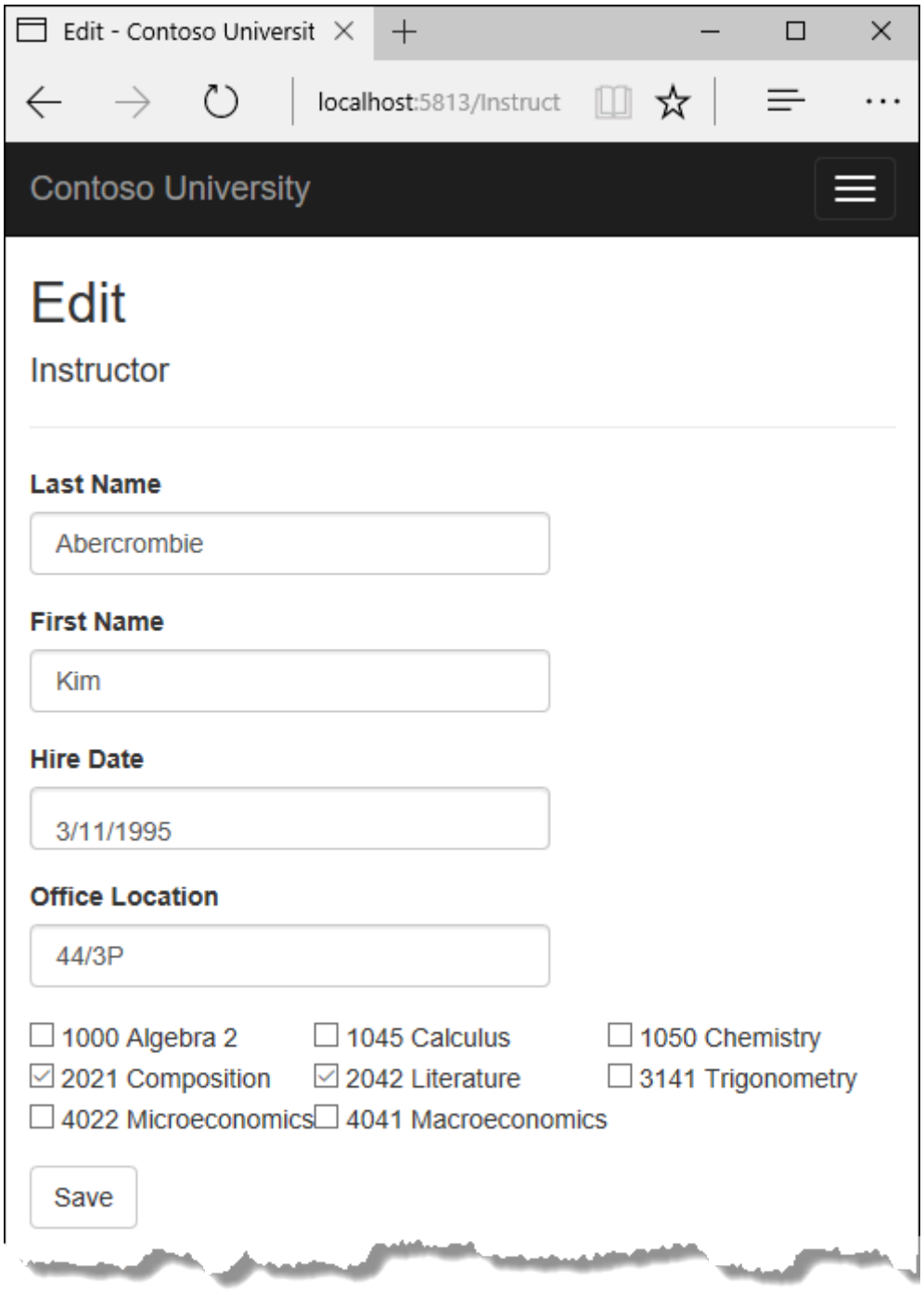
CSHTML

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

This code creates an HTML table that has three columns. In each column is a checkbox followed by a caption that consists of the course number and title. The checkboxes all have the same name ("selectedCourses"), which informs the model binder that they're to be treated as a group. The value attribute of each checkbox is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the checkboxes which are selected.

When the checkboxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the app, select the **Instructors** tab, and click **Edit** on an instructor to see the **Edit** page.



Change some course assignments and click Save. The changes you make are reflected on the Index page.

> ⓘ **Note**
>
> The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update Delete page

In `InstructorsController.cs`, delete the `DeleteConfirmed` method and insert the following code in its place.

```
C#

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

This code makes the following changes:

- Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to Create page

In `InstructorsController.cs`, delete the HttpGet and HttpPost `Create` methods, and then add the following code in their place:

```
C#

public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}
```

```
// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID =
instructor.ID, CourseID = int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The HttpGet `Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The HttpPost `Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

C#

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

As an alternative to doing this in controller code, you could do it in the `Instructor` model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

C#

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new
List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In `Views/Instructor/Create.cshtml`, add an office location text box and checkboxes for courses before the Submit button. As in the case of the Edit page, fix the formatting if Visual Studio reformats the code when you paste it.

CSHTML

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label">
</label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger"
/>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
```

```
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                            @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

Test by running the app and creating an instructor.

Handling Transactions

As explained in the CRUD tutorial, the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized Courses pages
- ✓ Added Instructors Edit page
- ✓ Added courses to Edit page
- ✓ Updated Delete page
- ✓ Added office location and courses to Create page

Advance to the next tutorial to learn how to handle concurrency conflicts.

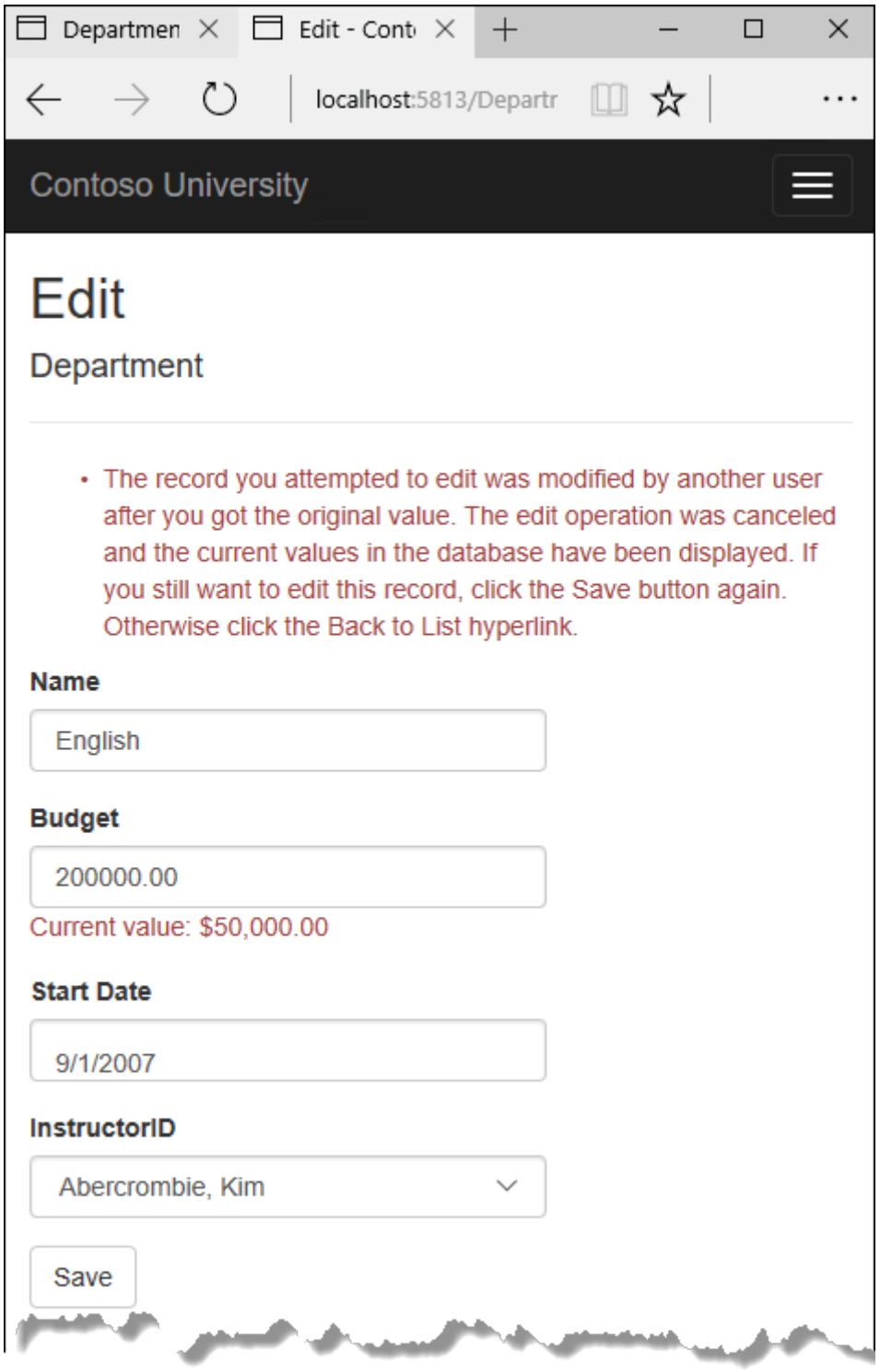
Handle concurrency conflicts

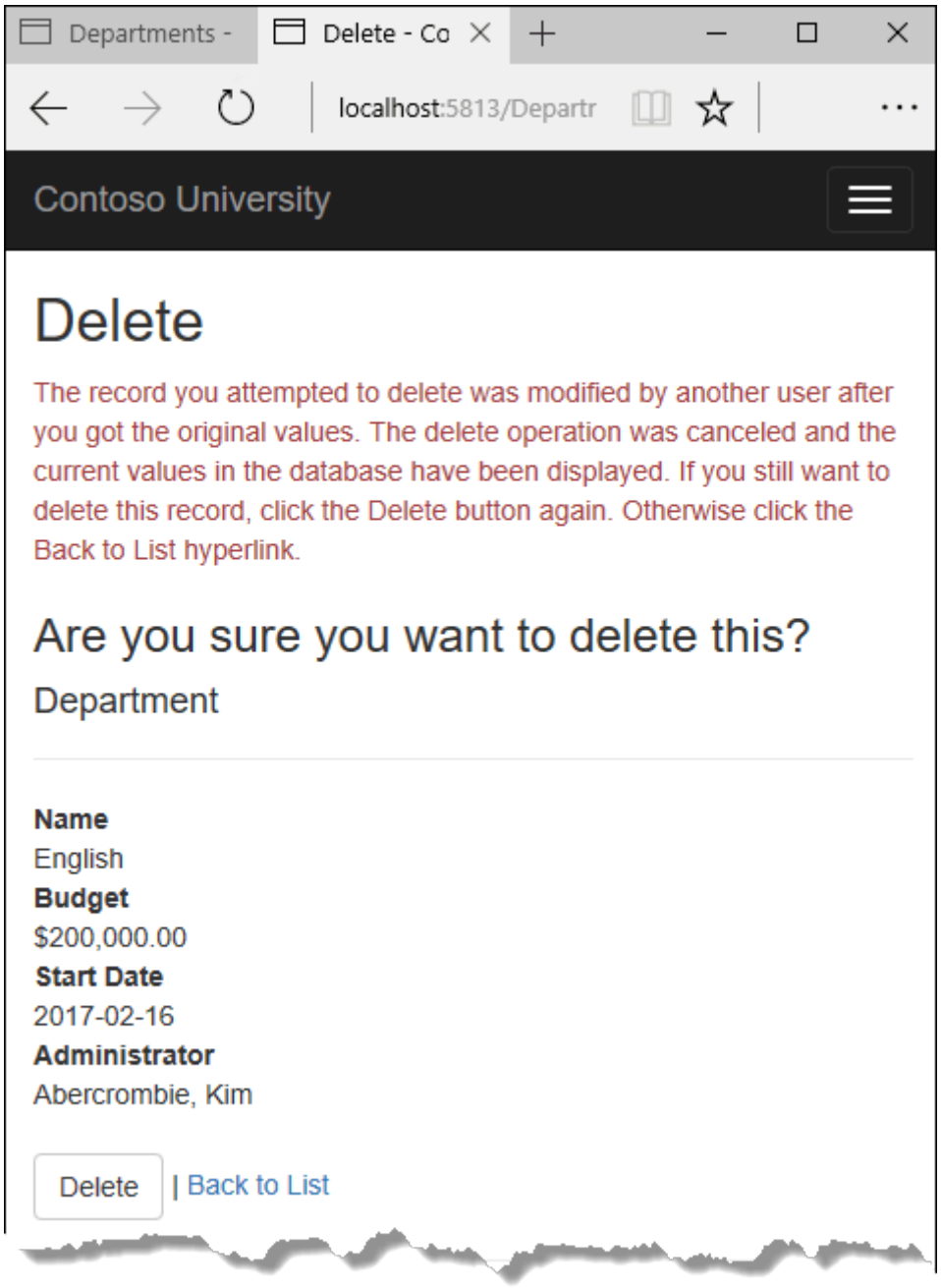
Tutorial: Handle concurrency - ASP.NET MVC with EF Core

Article • 07/09/2024

In earlier tutorials, you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the `Department` entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.


Departmen
Edit - Cont
localhost:5813/Departr
Contoso University
Edit
Department
The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink.
Name
English
Budget
200000.00
Current value: $50,000.00
Start Date
9/1/2007
InstructorID
Abercrombie, Kim
Save



In this tutorial, you:

- ✓ Learn about concurrency conflicts
- ✓ Add a tracking property
- ✓ Create Departments controller and views
- ✓ Update Index view
- ✓ Update Edit methods
- ✓ Update Edit view
- ✓ Test concurrency conflicts
- ✓ Update the Delete page
- ✓ Update Details and Create views

Prerequisites

- Update related data

Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

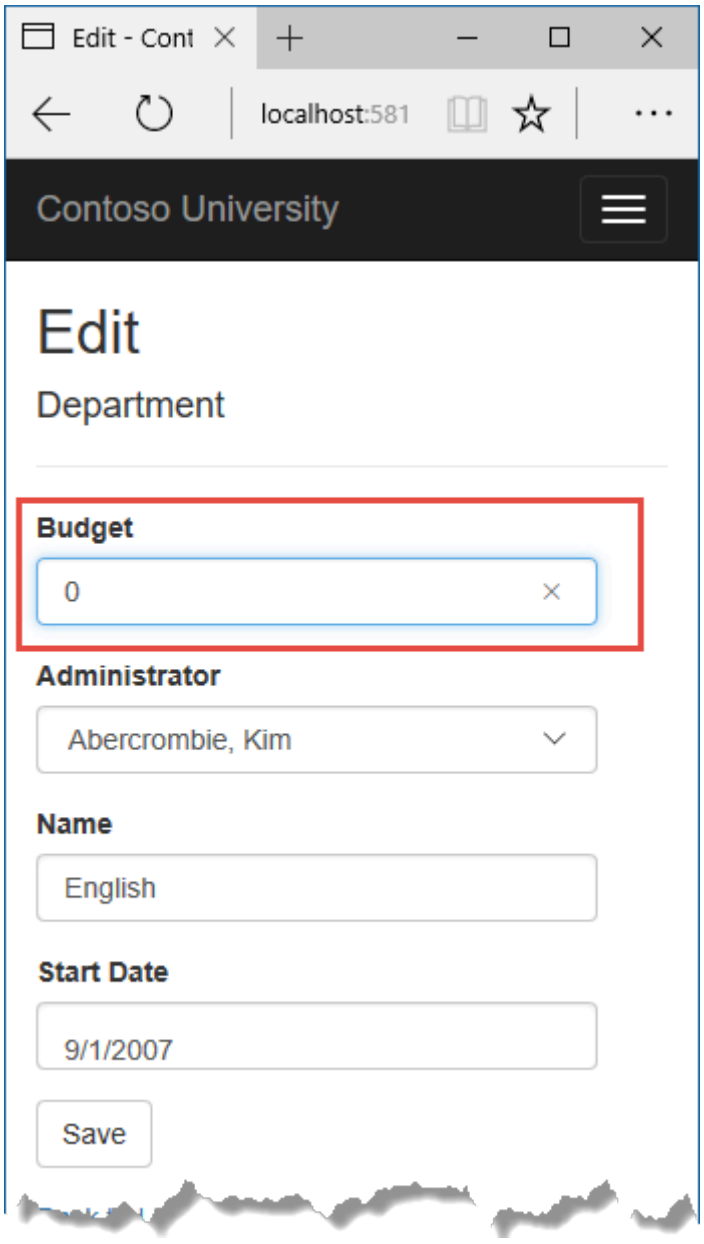
Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

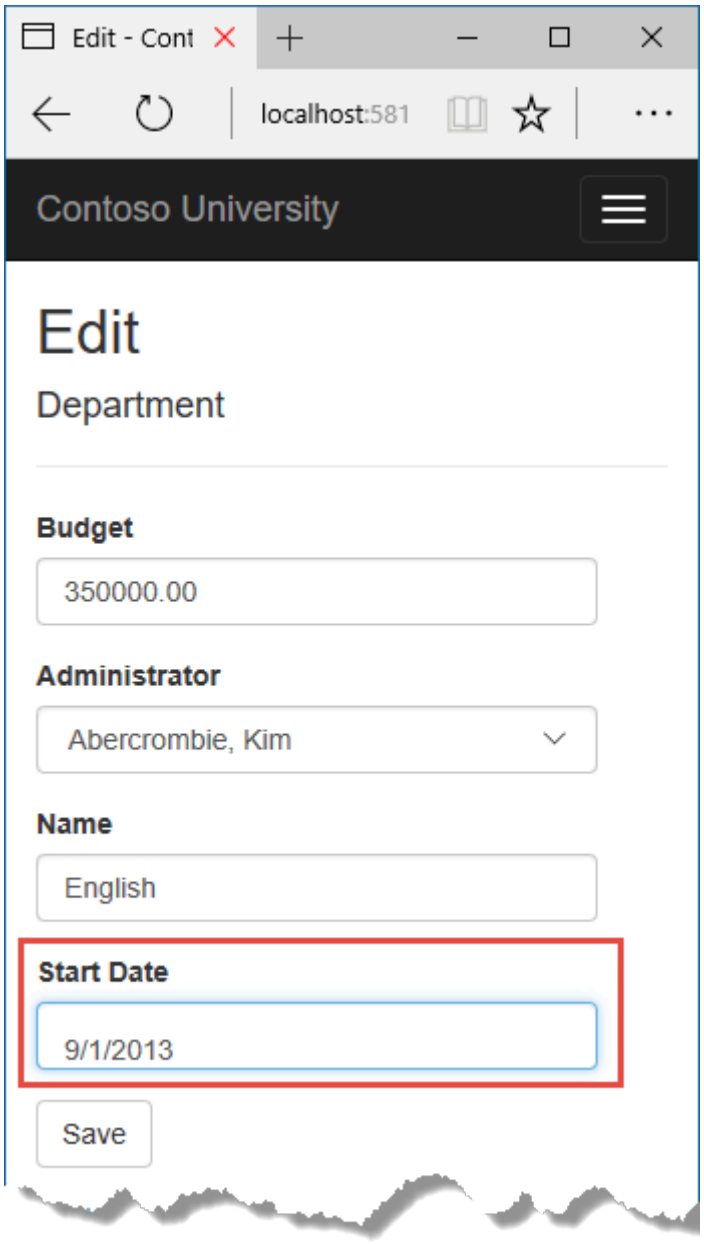
Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

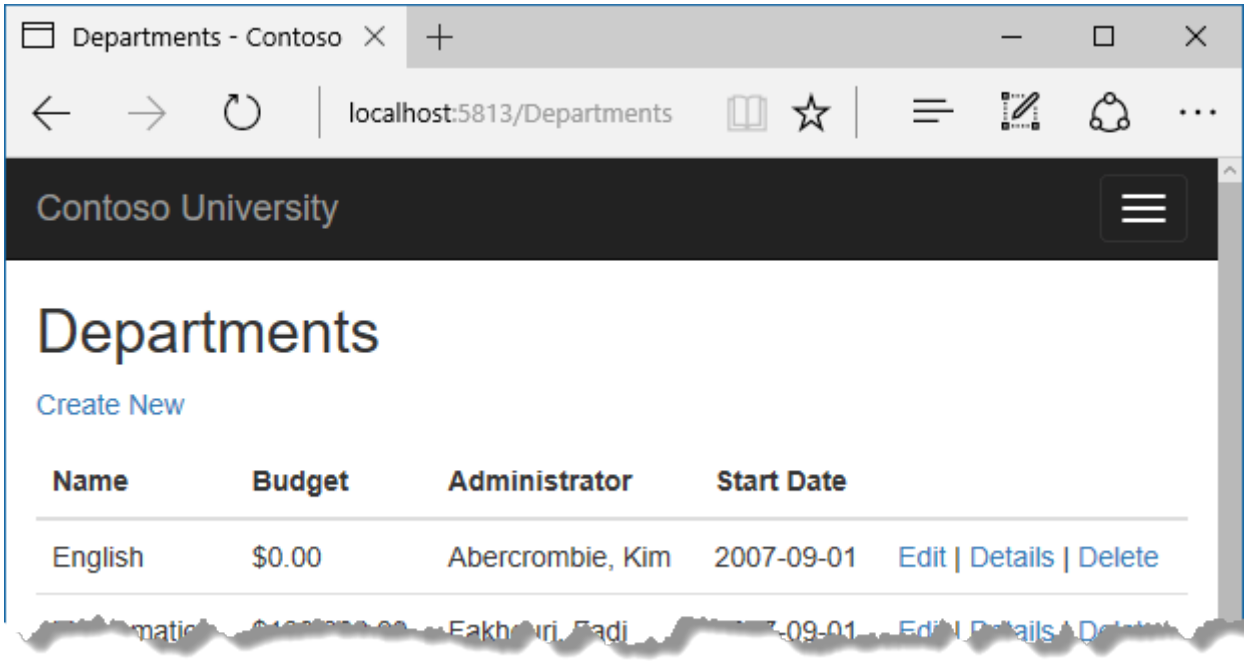
The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser returns to the Index page.



Then John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts.

Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

  In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent John's change from being updated in the database.

  Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting concurrency conflicts

You can resolve conflicts by handling `DbConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database

and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

  The data type of the tracking column is typically `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version) . If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

  As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

  If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the `ConcurrencyCheck` attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a `rowversion` tracking property to the Department entity, create a controller and views, and test to verify that everything works correctly.

Add a tracking property

In `Models/Department.cs`, add a tracking property named RowVersion:

```
```

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The `Timestamp` attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` replaced it. The .NET type for `rowversion` is a byte array.

If you prefer to use the fluent API, you can use the `IsRowVersion()` method (in `Data/SchoolContext.cs`) to specify the tracking property, as shown in the following example:

C#

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsRowVersion();
```

By adding a property you changed the database model, so you need to do another migration.

Save your changes and build the project, and then enter the following commands in the command window:

.NET CLI

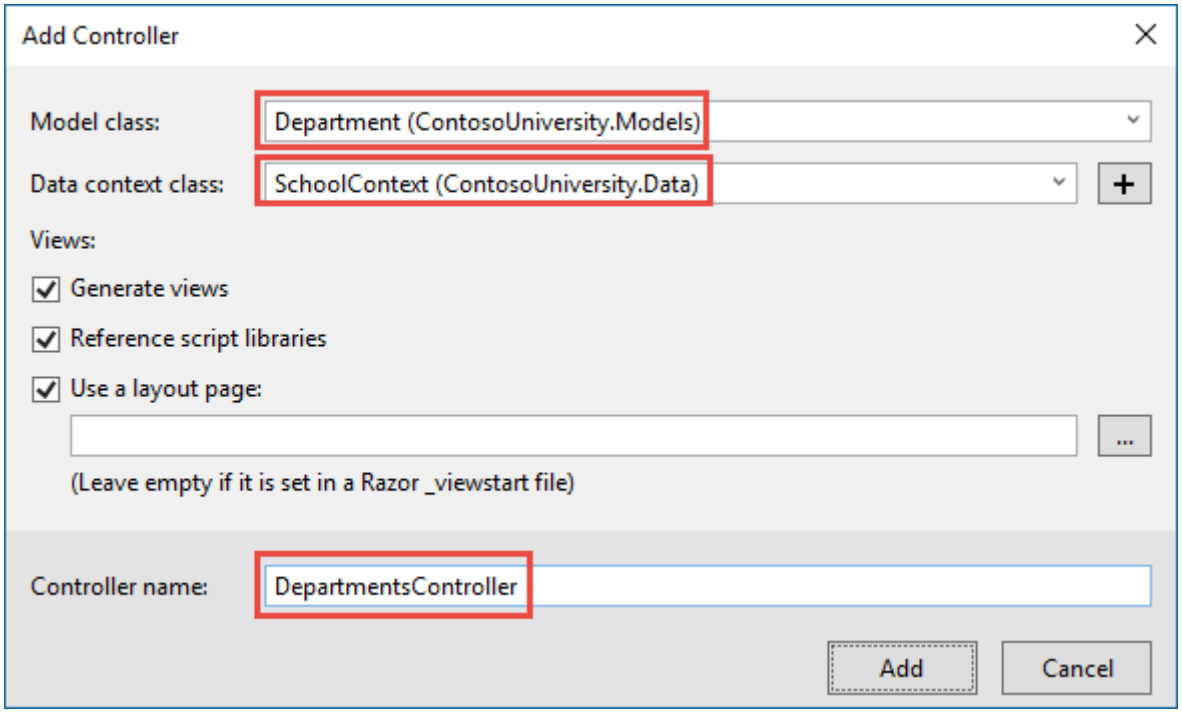
```
dotnet ef migrations add RowVersion
```

.NET CLI

```
dotnet ef database update
```

Create Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.



In the `DepartmentsController.cs` file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

C#

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID",
"FullName", department.InstructorID);
```

Update Index view

The scaffolding engine created a `RowVersion` column in the Index view, but that field shouldn't be displayed.

Replace the code in `Views/Departments/Index.cshtml` with the following code.

CSHTML

```
@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
```

```
                    @Html.DisplayFor(modelItem =>
item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-
id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

This changes the heading to "Departments", deletes the `RowVersion` column, and shows full name instead of first name for the administrator.

Update Edit methods

In both the HttpGet `Edit` method and the `Details` method, add `AsNoTracking`. In the HttpGet `Edit` method, add eager loading for the Administrator.

C#

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .FirstOrDefaultAsync(m => m.DepartmentID == id);
```

Replace the existing code for the HttpPost `Edit` method with the following code:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i =>
i.Administrator).FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
```

```
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another
user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors,
"ID", "FullName", deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue
= rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by
another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value:
{databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value:
{databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {
                    ModelState.AddModelError("StartDate", $"Current value:
{databaseValues.StartDate:d}");
                }
                if (databaseValues.InstructorID !=
clientValues.InstructorID)
```

```
                {
                    Instructor databaseInstructor = await
_context.Instructors.FirstOrDefaultAsync(i => i.ID ==
databaseValues.InstructorID);
                    ModelState.AddModelError("InstructorID", $"Current
value: {databaseInstructor?.FullName}");
                }

                ModelState.AddModelError(string.Empty, "The record you
attempted to edit "
                        + "was modified by another user after you got the
original value. The "
                        + "edit operation was canceled and the current
values in the database "
                        + "have been displayed. If you still want to edit
this record, click "
                        + "the Save button again. Otherwise click the Back
to List hyperlink.");
                departmentToUpdate.RowVersion =
(byte[])databaseValues.RowVersion;
                ModelState.Remove("RowVersion");
            }
        }
    }
    ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID",
"FullName", departmentToUpdate.InstructorID);
    return View(departmentToUpdate);
}
```

The code begins by trying to read the department to be updated. If the `FirstOrDefaultAsync` method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a `Department` entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the `Department` entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and this method receives that value in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity.

C#

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue =
rowVersion;
```

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original `RowVersion` value. If no

rows are affected by the UPDATE command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the `Entries` property on the exception object.

C#

```
var exceptionEntry = ex.Entries.Single();
```

The `Entries` collection will have just one `EntityEntry` object. You can use that object to get the new values entered by the user and the current database values.

C#

```
var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

C#

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value:
{databaseValues.Name}");
```

Finally, the code sets the `RowVersion` value of the `departmentToUpdate` to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

C#

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the view, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update Edit view

In `Views/Departments/Edit.cshtml`, make the following changes:

- Add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property.
- Add a "Select Administrator" option to the drop-down list.

CSHTML

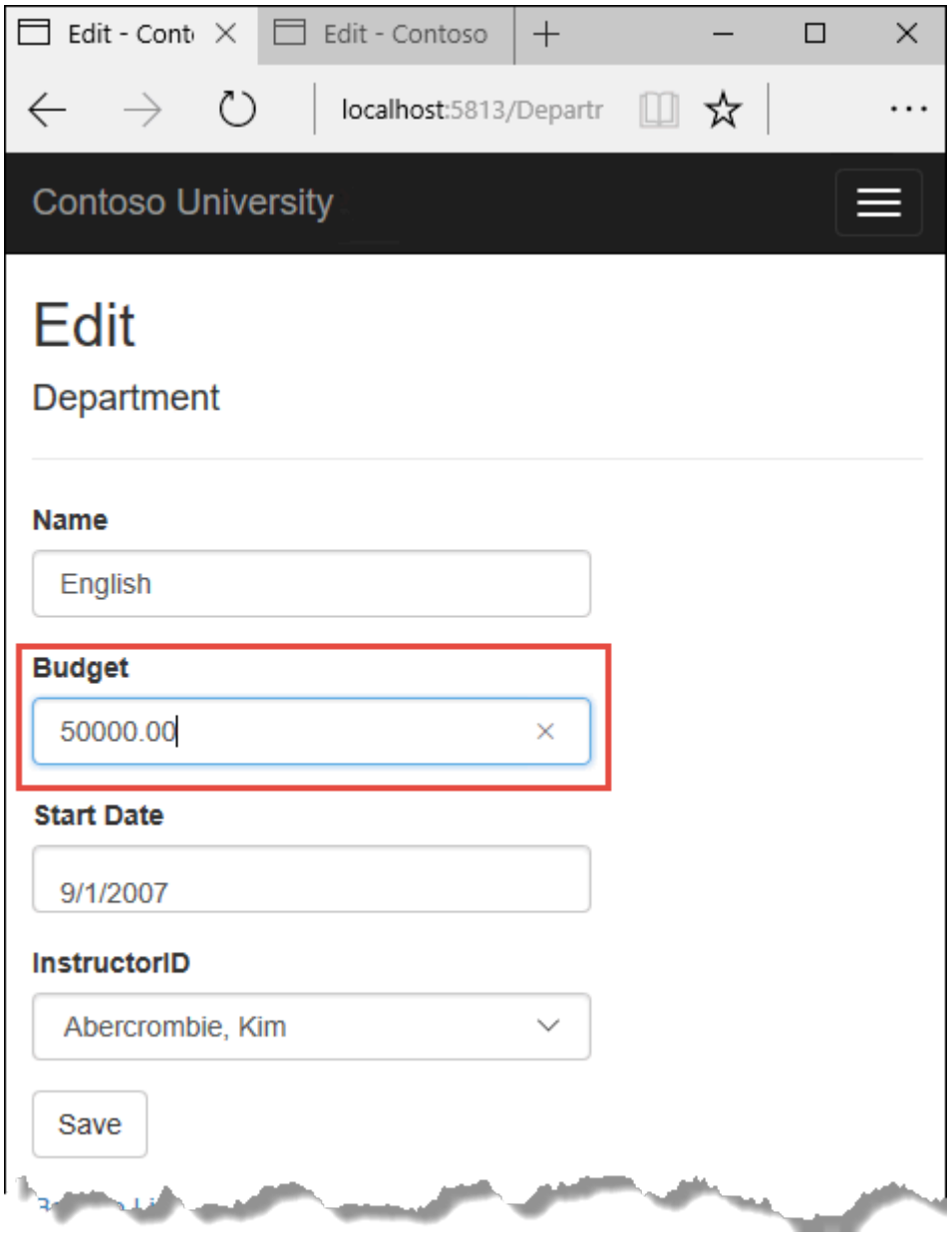
```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-
items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
                <span asp-validation-for="InstructorID" class="text-danger">
```

```
</span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test concurrency conflicts

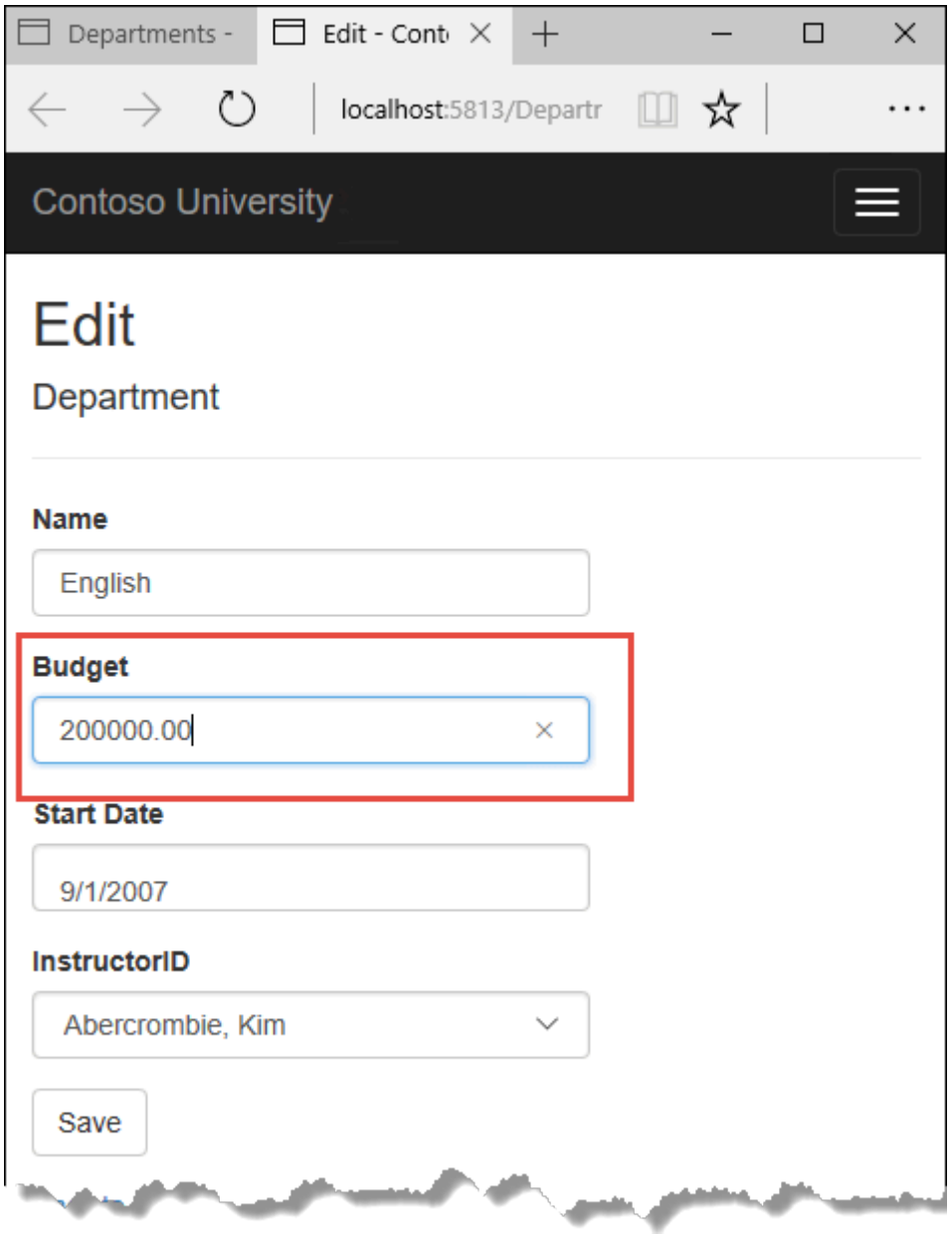
Run the app and go to the Departments Index page. Right-click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click **Save**.

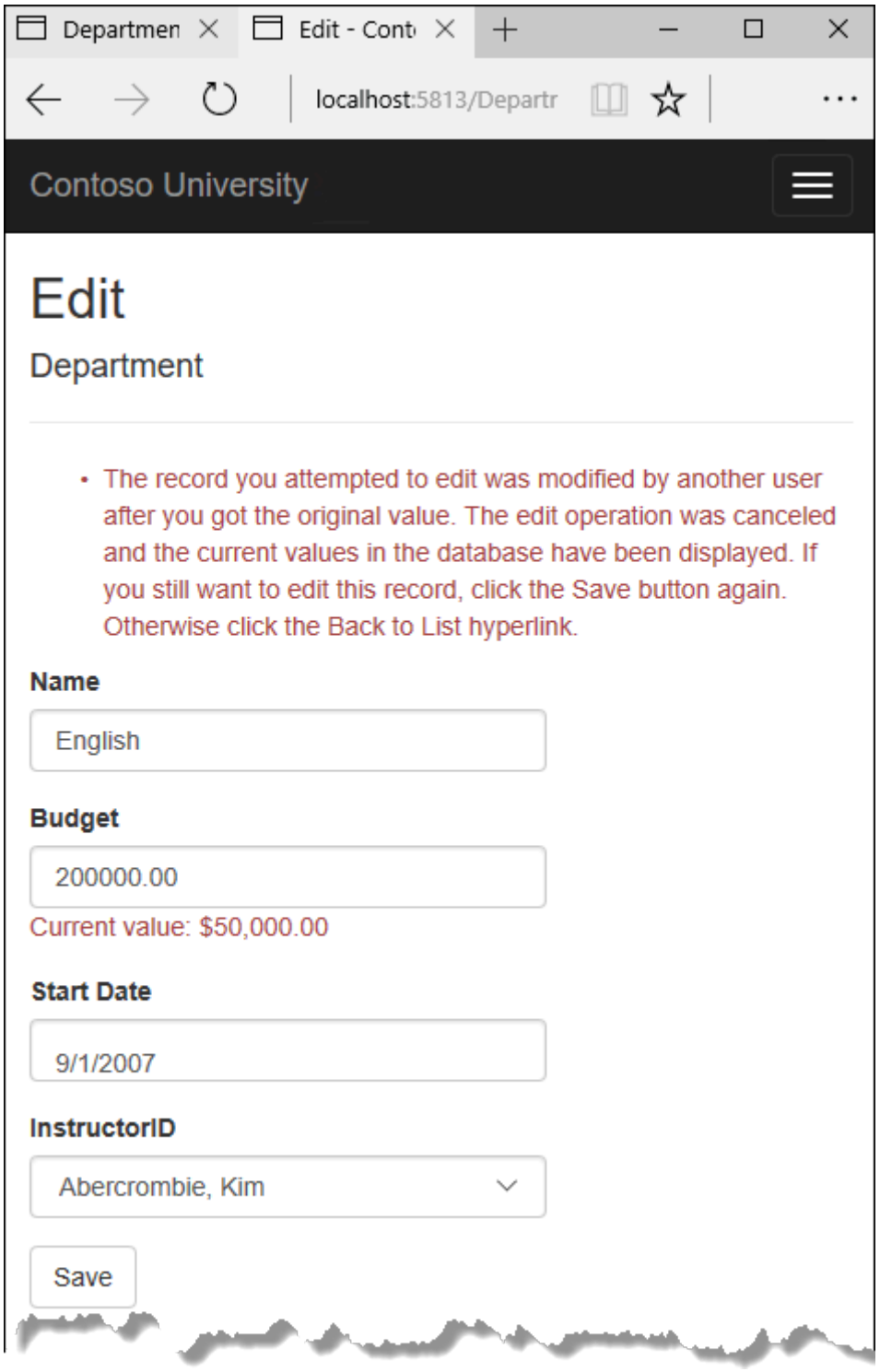


The browser shows the Index page with the changed value.

Change a field in the second browser tab.



Click **Save**. You see an error message:



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the HttpGet `Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the HttpPost `Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL

DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the HttpGet `Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

Update the Delete methods in the Departments controller

In `DepartmentsController.cs`, replace the HttpGet `Delete` method with the following code:

C#

```
public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to
delete "
            + "was modified by another user after you got the original
values. "
            + "The delete operation was canceled and the current values in
the "
            + "database have been displayed. If you still want to delete
this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }
```

```
    return View(department);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the department does exist, it was changed by another user. In that case, the code sends an error message to the view using `ViewData`.

Replace the code in the HttpPost `Delete` method (named `DeleteConfirmed`) with the following code:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID ==
department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { concurrencyError =
true, id = department.DepartmentID });
    }
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

C#

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a `Department` entity instance created by the model binder. This gives EF access to the RowVers`ion property value in addition to the record key.

C#

```
public async Task<IActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code used the name `DeleteConfirmed` to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the `AnyAsync` method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

Update the Delete view

In `Views/Departments/Delete.cshtml`, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
```

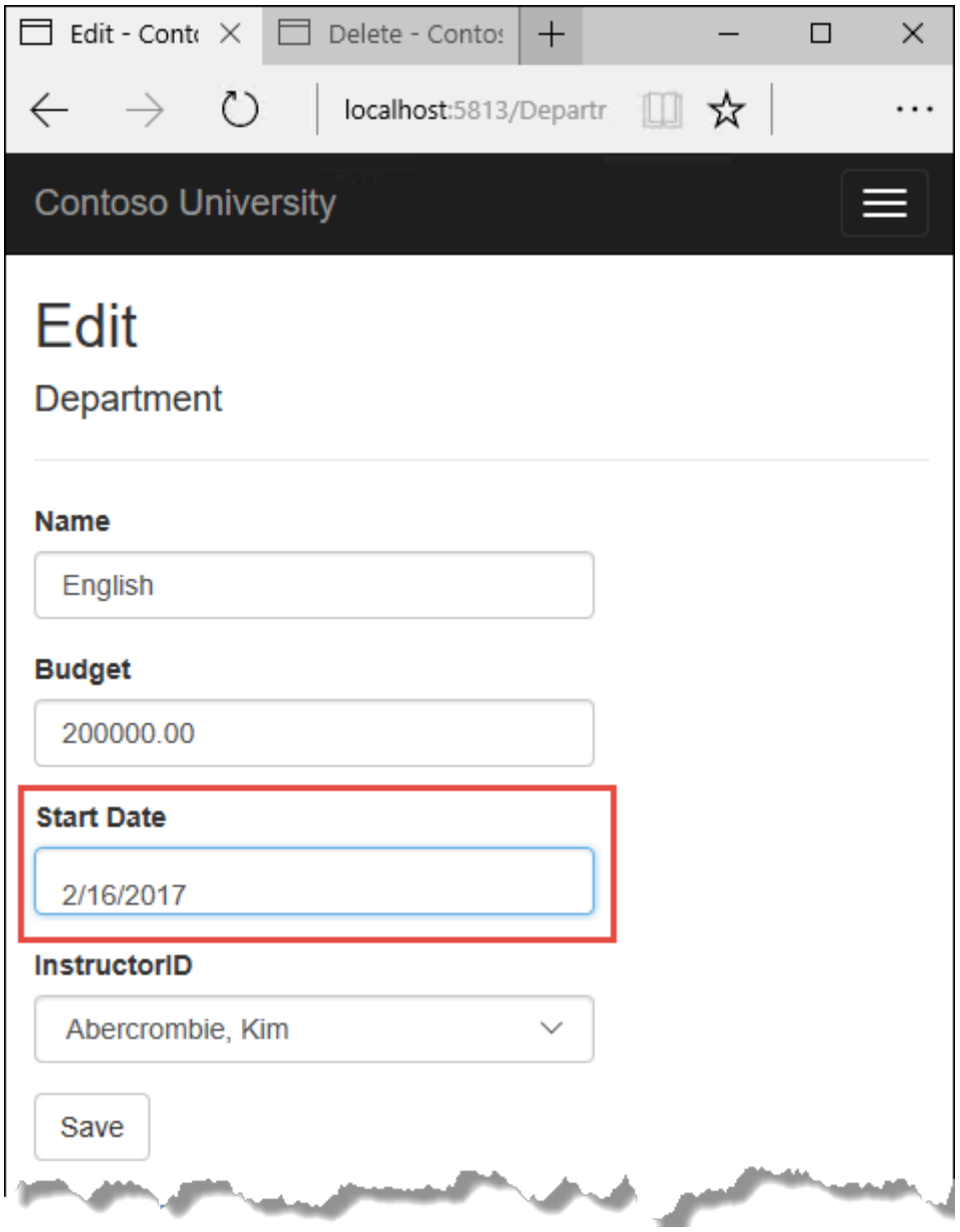
```
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

This makes the following changes:

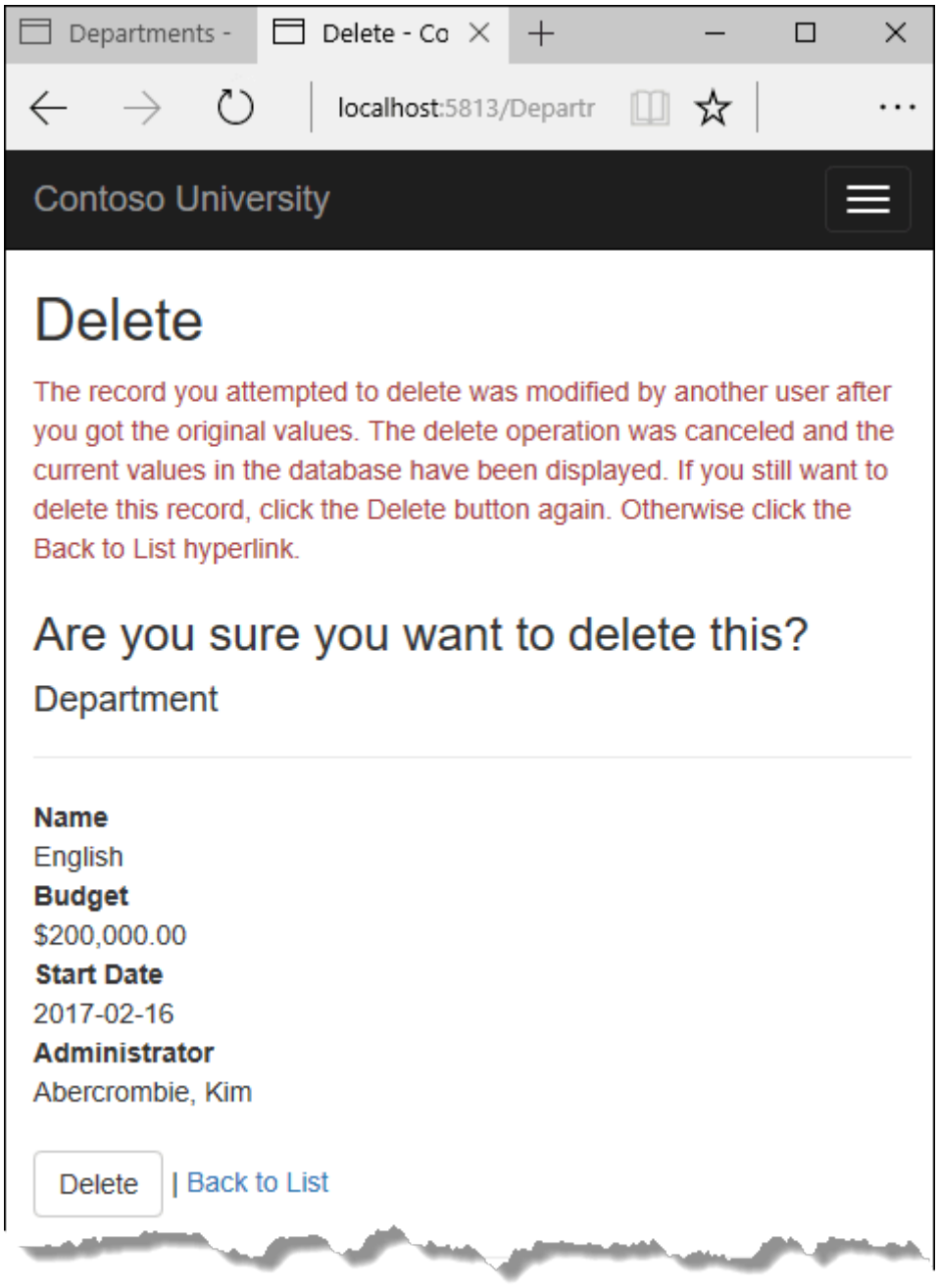
- Adds an error message between the `h2` and `h3` headings.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Removes the RowVersion field.
- Adds a hidden field for the `RowVersion` property.

Run the app and go to the Departments Index page. Right-click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**:



In the second tab, click **Delete**. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in `Views/Departments/Details.cshtml` to delete the RowVersion column and show the full name of the Administrator.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
```

```
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Replace the code in `Views/Departments/Create.cshtml` to add a Select option to the drop-down list.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Department</h4>
```

```
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-
items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default"
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Get the code

Download or view the completed application.

Additional resources

For more information about how to handle concurrency in EF Core, see Concurrency conflicts.

Next steps

In this tutorial, you:

- ✓ Learned about concurrency conflicts
- ✓ Added a tracking property
- ✓ Created Departments controller and views
- ✓ Updated Index view
- ✓ Updated Edit methods
- ✓ Updated Edit view
- ✓ Tested concurrency conflicts
- ✓ Updated the Delete page
- ✓ Updated Details and Create views

Advance to the next tutorial to learn how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

Next: Implement table-per-hierarchy inheritance

Tutorial: Implement inheritance - ASP.NET MVC with EF Core

Article • 04/10/2024

In the previous tutorial, you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

In this tutorial, you:

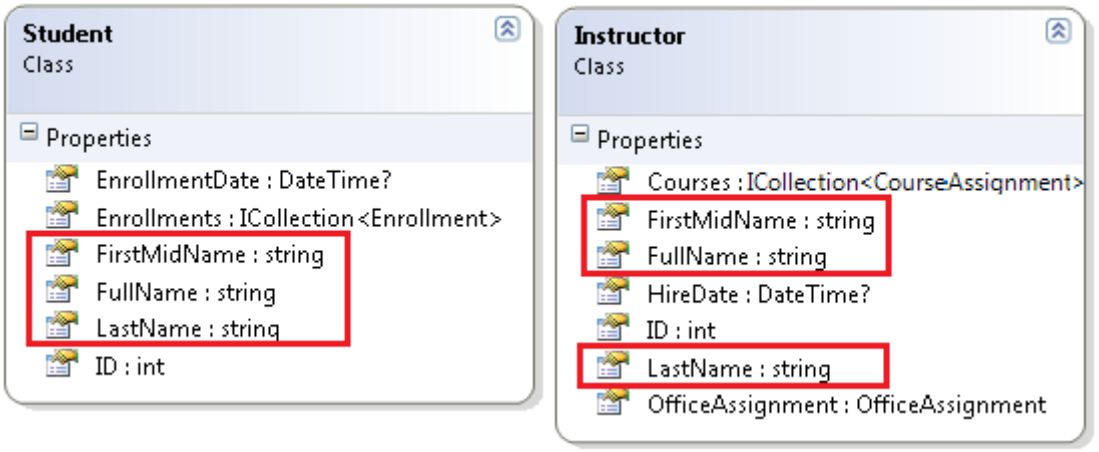
- ✓ Map inheritance to database
- ✓ Create the Person class
- ✓ Update Instructor and Student
- ✓ Add Person to the model
- ✓ Create and update migrations
- ✓ Test the implementation

Prerequisites

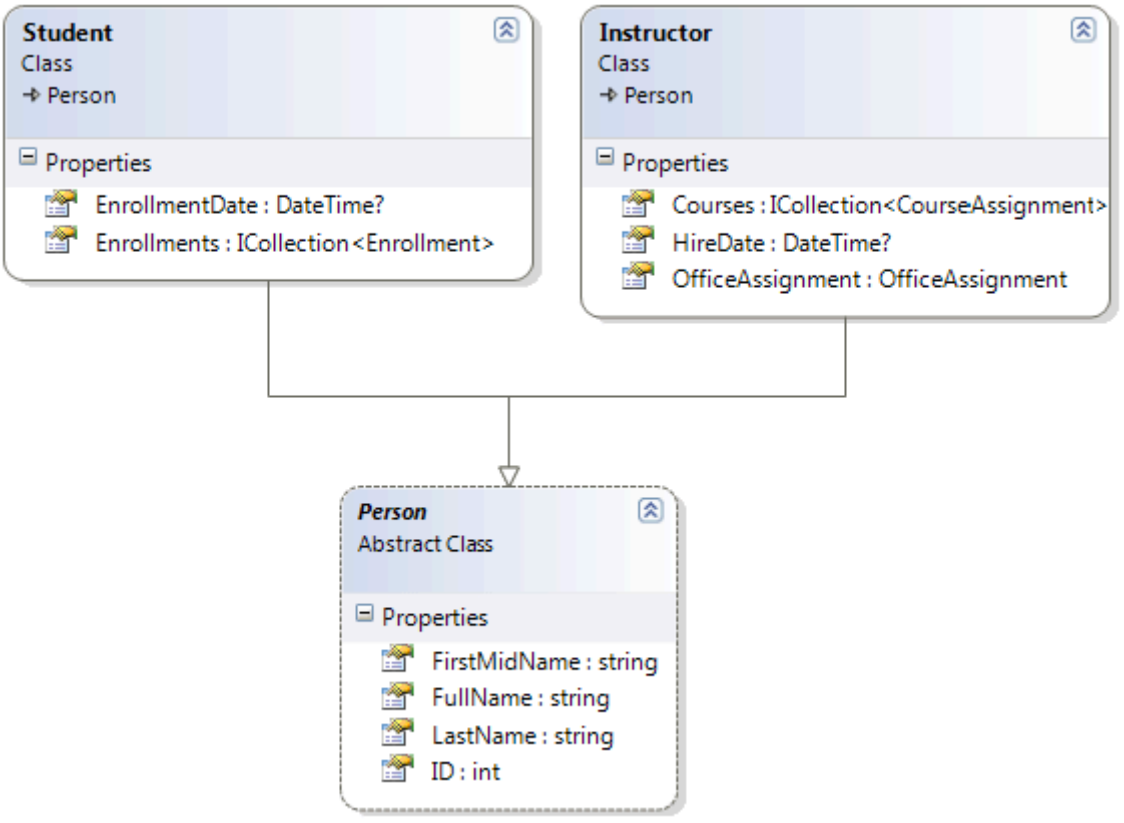
- Handle Concurrency

Map inheritance to database

The `Instructor` and `Student` classes in the School data model have several properties that are identical:

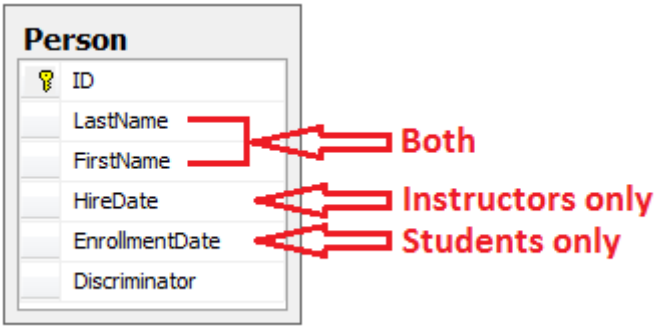


Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class that contains only those shared properties, then make the `Instructor` and `Student` classes inherit from that base class, as shown in the following illustration:



There are several ways this inheritance structure could be represented in the database. You could have a `Person` table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (HireDate), some only to students (EnrollmentDate), some to both (LastName, FirstName). Typically, you'd have a discriminator column to indicate which type each row

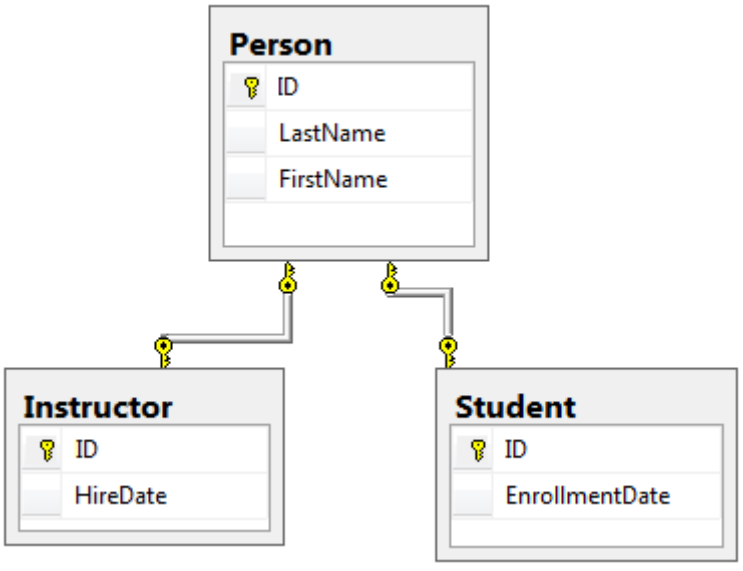
represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy (TPH)* inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the `Person` table and have separate `Instructor` and `Student` tables with the date fields.

> ⚠ **Warning**
>
> Table-Per-Type (TPT) is not supported by EF Core 3.x, however it is has been implemented in EF Core 5.0.



This pattern of making a database table for each entity class is called *table-per-type (TPT)* inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called *Table-per-Concrete Class (TPC)* inheritance. If you implemented TPC

inheritance for the `Person`, `Student`, and `Instructor` classes as shown earlier, the `Student` and `Instructor` tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration.

> **Tip**
>
> Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.

Create the Person class

In the Models folder, create Person.cs and replace the template code with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than
50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
```

```
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}
```

Update Instructor and Student

In `Instructor.cs`, derive the Instructor class from the Person class and remove the key and name fields. The code will look like the following example:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Make the same changes in `Student.cs`.

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
```

```
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }


        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Add Person to the model

Add the Person entity type to `SchoolContext.cs`. The new lines are highlighted.

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
        public DbSet<Person> People { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>
().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>
().ToTable("CourseAssignment");
```

```
            modelBuilder.Entity<Person>().ToTable("Person");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

Create and update migrations

Save your changes and build the project. Then open the command window in the project folder and enter the following command:

.NET CLI

```
dotnet ef migrations add Inheritance
```

Don't run the `database update` command yet. That command will result in lost data because it will drop the Instructor table and rename the Student table to Person. You need to provide custom code to preserve existing data.

Open `Migrations/<timestamp>_Inheritance.cs` and replace the `Up` method with the following code:

C#

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table:
"Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table:
"Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table:
"Person", nullable: false, maxLength: 128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table:
"Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person",
```

```
nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName,
HireDate, EnrollmentDate, Discriminator, OldId) SELECT LastName, FirstName,
null AS HireDate, EnrollmentDate, 'Student' AS Discriminator, ID AS OldId
FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID
FROM dbo.Person WHERE OldId = Enrollment.StudentId AND Discriminator =
'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
         name: "IX_Enrollment_StudentID",
         table: "Enrollment",
         column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}
```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
- Adds nullable EnrollmentDate for students.
- Adds Discriminator column to indicate whether a row is for a student or an instructor.
- Makes HireDate nullable since student rows won't have hire dates.
- Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they will get new primary key values.

- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `database update` command:

.NET CLI

```
dotnet ef database update
```

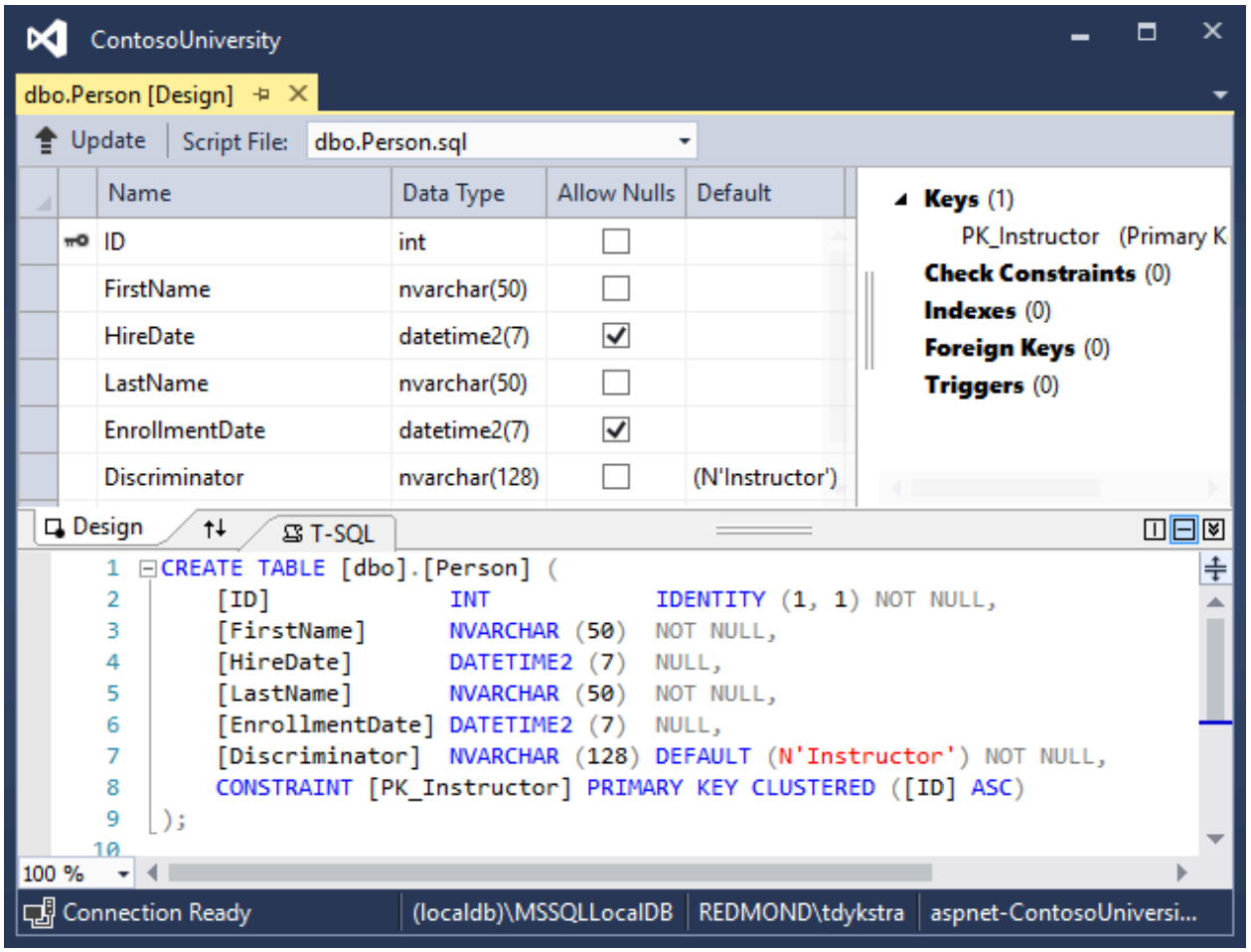
(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

> **Note**
>
> It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the `database drop` CLI command.

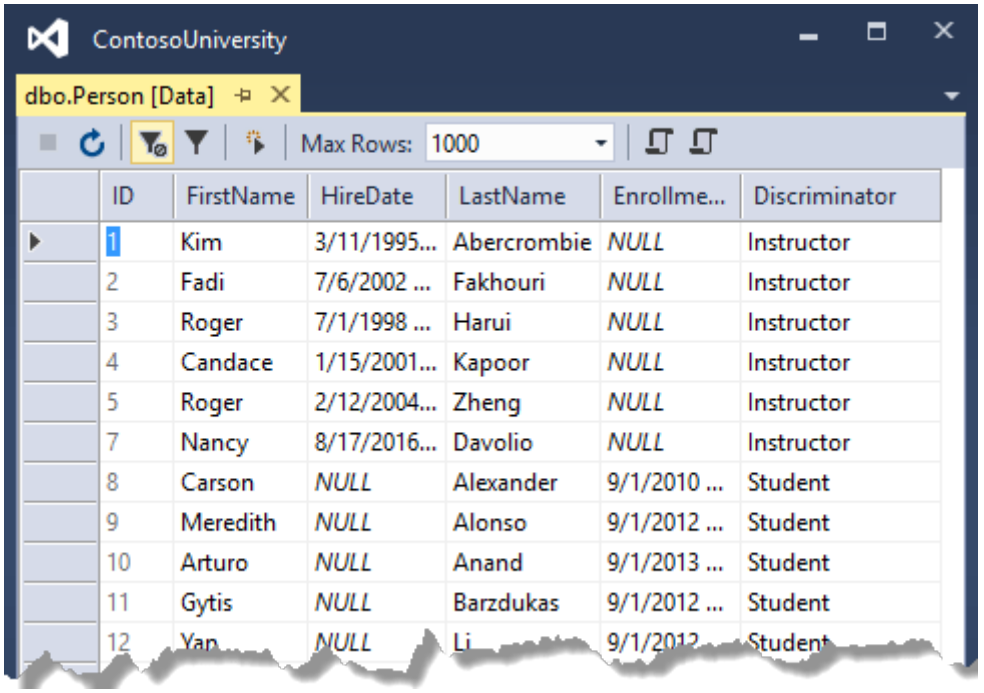
Test the implementation

Run the app and try various pages. Everything works the same as it did before.

In **SQL Server Object Explorer**, expand **Data Connections/SchoolContext** and then **Tables**, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Right-click the Person table, and then click **Show Table Data** to see the discriminator column.



Get the code

Download or view the completed application.

Additional resources

For more information about inheritance in Entity Framework Core, see Inheritance.

Next steps

In this tutorial, you:

- ✓ Mapped inheritance to database
- ✓ Created the Person class
- ✓ Updated Instructor and Student
- ✓ Added Person to the model
- ✓ Created and update migrations
- ✓ Tested the implementation

Advance to the next tutorial to learn how to handle a variety of relatively advanced Entity Framework scenarios.

Next: Advanced topics

Tutorial: Learn about advanced scenarios - ASP.NET MVC with EF Core

Article • 04/10/2024

In the previous tutorial, you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET Core web applications that use Entity Framework Core.

In this tutorial, you:

- ✓ Perform raw SQL queries
- ✓ Call a query to return entities
- ✓ Call a query to return other types
- ✓ Call an update query
- ✓ Examine SQL queries
- ✓ Create an abstraction layer
- ✓ Learn about Automatic change detection
- ✓ Learn about EF Core source code and development plans
- ✓ Learn how to use dynamic LINQ to simplify code

Prerequisites

- Implement Inheritance

Perform raw SQL queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

- Use the `DbSet.FromSql` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they're automatically tracked by the database context unless you turn tracking off.
- Use the `Database.ExecuteSqlCommand` for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Call a query to return entities

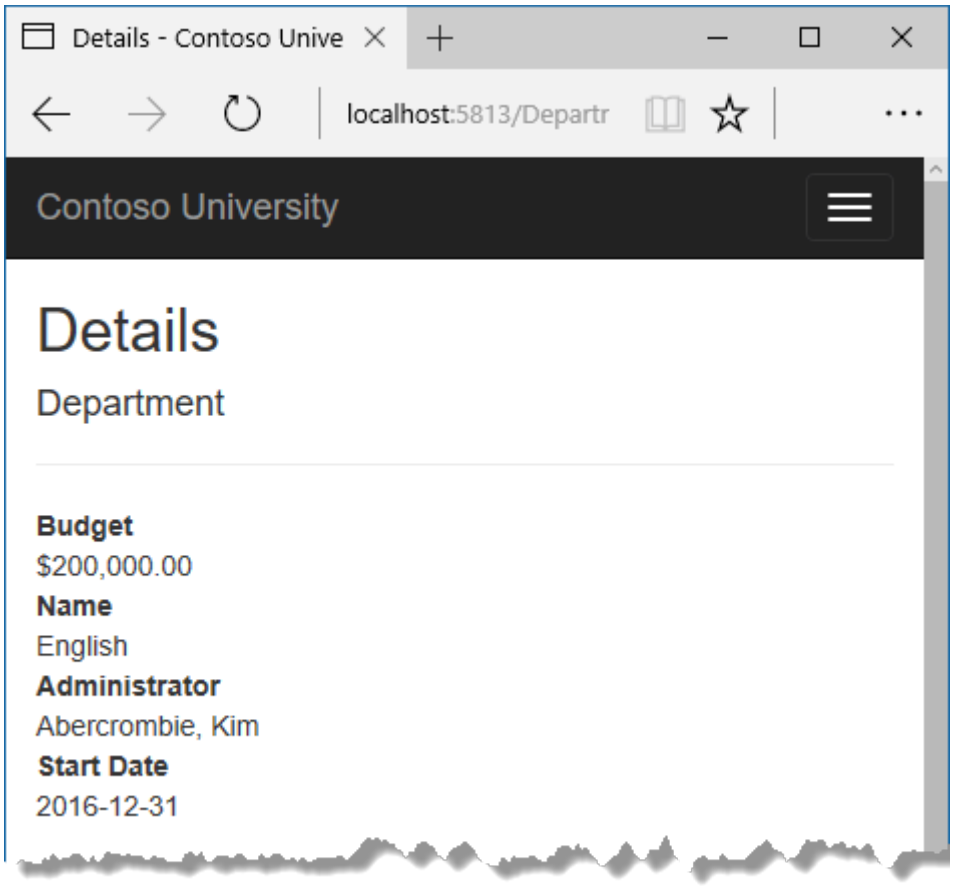
The `DbSet<TEntity>` class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the Department controller.

In `DepartmentsController.cs`, in the `Details` method, replace the code that retrieves a department with a `FromSql` method call, as shown in the following highlighted code:

C#

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}
```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Call a query to return other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (`_context.Students`) and used LINQ to project the results into a list of `EnrollmentDateGroup` view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is to write ADO.NET code and get the database connection from EF.

In `HomeController.cs`, replace the `About` method with the following code:

C#

```
public async Task<ActionResult> About()
{
```

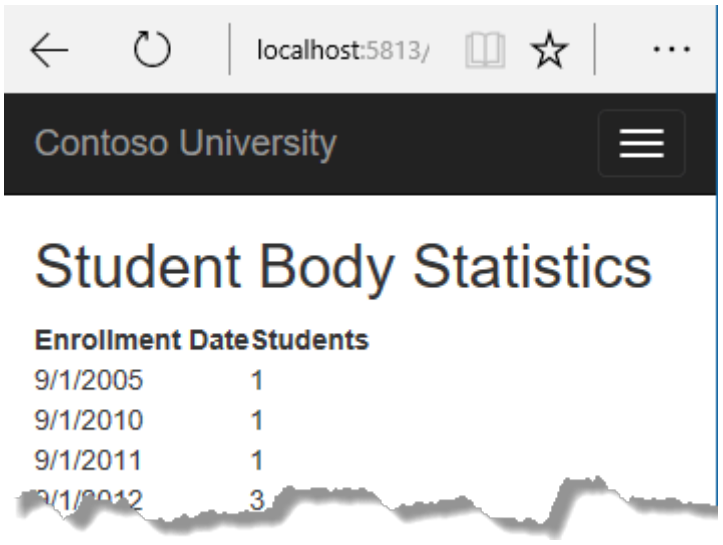
```
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
    var conn = _context.Database.GetDbConnection();
    try
    {
        await conn.OpenAsync();
        using (var command = conn.CreateCommand())
        {
            string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount
"
                + "FROM Person "
                + "WHERE Discriminator = 'Student' "
                + "GROUP BY EnrollmentDate";
            command.CommandText = query;
            DbDataReader reader = await command.ExecuteReaderAsync();

            if (reader.HasRows)
            {
                while (await reader.ReadAsync())
                {
                    var row = new EnrollmentDateGroup { EnrollmentDate =
reader.GetDateTime(0), StudentCount = reader.GetInt32(1) };
                    groups.Add(row);
                }
            }
            reader.Dispose();
        }
    }
    finally
    {
        conn.Close();
    }
    return View(groups);
}
```

Add a using statement:

C#

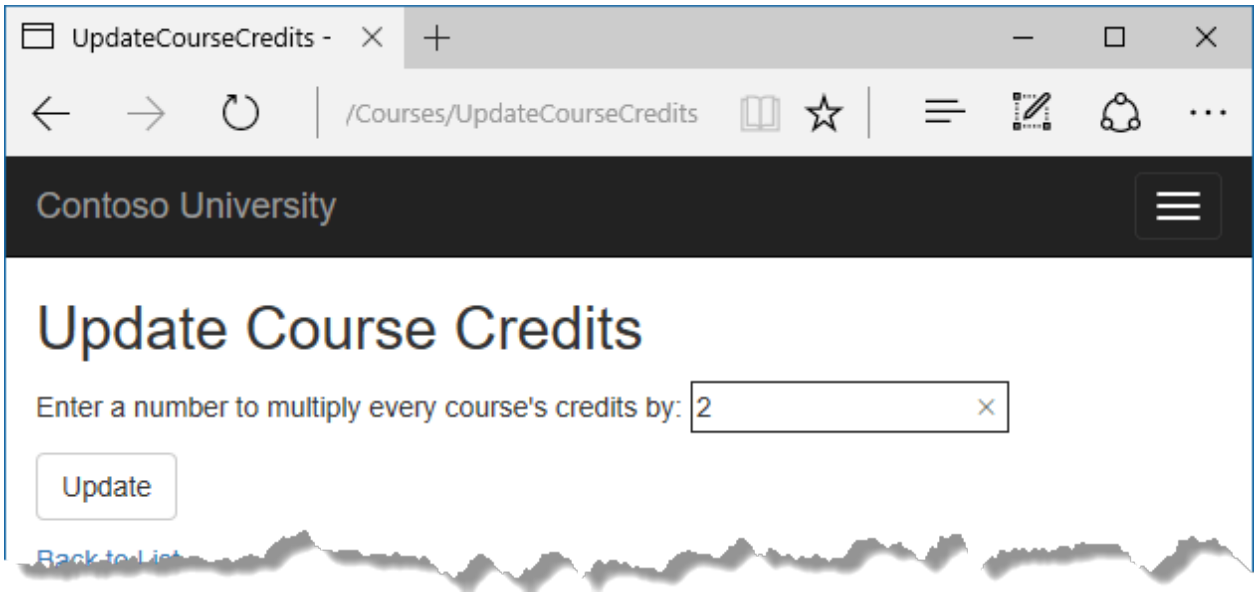
```
using System.Data.Common;
```

Run the app and go to the About page. It displays the same data it did before.



Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL UPDATE statement. The web page will look like the following illustration:



In `CoursesController.cs`, add UpdateCourseCredits methods for HttpGet and HttpPost:

C#

```
public IActionResult UpdateCourseCredits()
{
```

```
    return View();
}
```

C#

```
[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

When the controller processes an HttpGet request, nothing is returned in `ViewData["RowsAffected"]`, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the HttpPost method is called, and multiplier has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in `ViewData`. When the view gets a `RowsAffected` value, it displays the number of rows updated.

In **Solution Explorer**, right-click the *Views/Courses* folder, and then click **Add** > **New Item**.

In the **Add New Item** dialog, click **ASP.NET Core** under **Installed** in the left pane, click **Razor View**, and name the new view `UpdateCourseCredits.cshtml`.

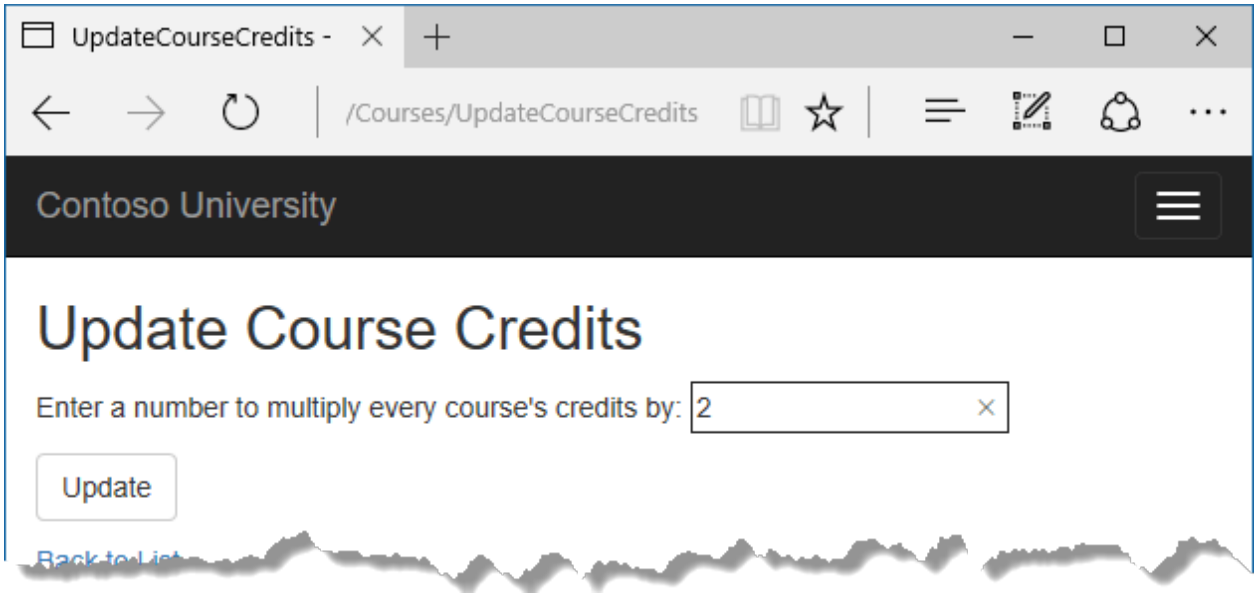
In `Views/Courses/UpdateCourseCredits.cshtml`, replace the template code with the following code:

CSHTML

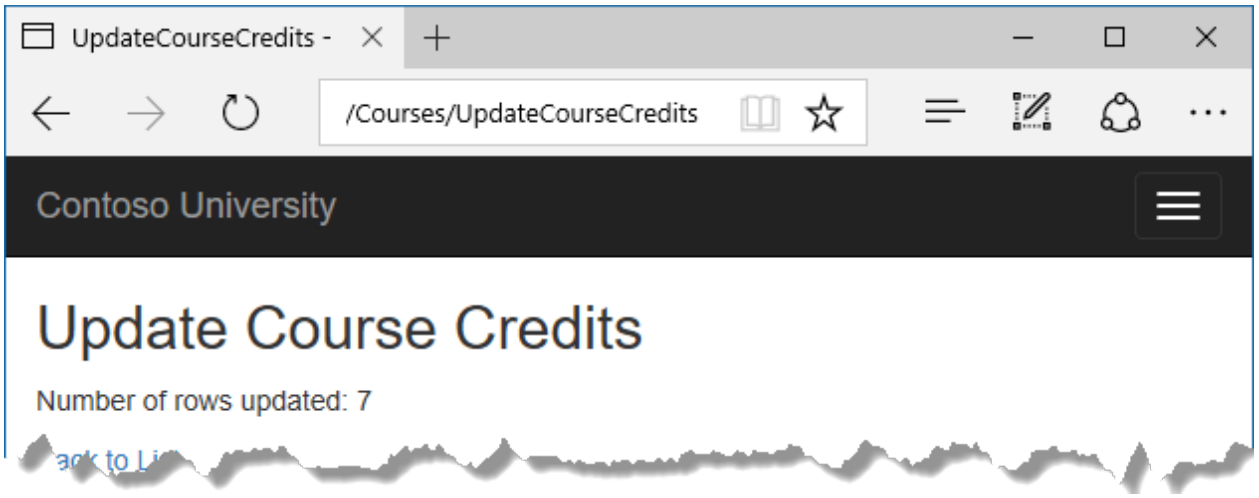
```
@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
```

```
            <p>
                Enter a number to multiply every course's credits by:
@Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default"
/>
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: `http://localhost:5813/Courses/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The `Credits` property has a `[Range(0, 5)]` attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see Raw SQL Queries.

Examine SQL queries

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open `StudentsController.cs` and in the `Details` method set a breakpoint on the `if (student == null)` statement.

Run the app in debug mode, and go to the Details page for a student.

Go to the **Output** window showing debug output, and you see the query:

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed
DbCommand (56ms) [Parameters=[@__id_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].
[LastName], [s].[EnrollmentDate]
FROM [Person] AS [s]
WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
ORDER BY [s].[ID]
```

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed
DbCommand (122ms) [Parameters=[@__id_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID],
[s.Enrollments].[Grade], [s.Enrollments].[StudentID], [e.Course].[CourseID],
[e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]
FROM [Enrollment] AS [s.Enrollments]
INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] =
[e.Course].[CourseID]
INNER JOIN (
    SELECT TOP(1) [s0].[ID]
    FROM [Person] AS [s0]
    WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @__id_0)
    ORDER BY [s0].[ID]
) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]
ORDER BY [t].[ID]
```

You'll notice something here that might surprise you: the SQL selects up to 2 rows (`TOP(2)`) from the Person table. The `SingleOrDefaultAsync` method doesn't resolve to 1 row on the server. Here's why:

- If the query would return multiple rows, the method returns null.
- To determine whether the query would return multiple rows, EF has to check if it returns at least 2.

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the **Output** window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

Create an abstraction layer

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These 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). However, writing additional code to implement these patterns isn't always the best choice for applications that use EF, for several reasons:

- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see the Entity Framework 5 version of this tutorial series.

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see Test with InMemory.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- DbContext.SaveChanges
- DbContext.Entry
- ChangeTracker.Entries

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the `ChangeTracker.AutoDetectChangesEnabled` property. For example:

```
C#

_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

EF Core source code and development plans

The Entity Framework Core source is at https://github.com/dotnet/efcore. The EF Core repository contains nightly builds, issue tracking, feature specs, design meeting notes, and the roadmap for future development. You can file or find bugs, and contribute.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the scaffold-dbcontext command. See the getting-started tutorial.

Use dynamic LINQ to simplify code

The third tutorial in this series shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

C#

```
 public async Task<IActionResult> Index(
     string sortOrder,
     string currentFilter,
     string searchString,
     int? pageNumber)
 {
     ViewData["CurrentSort"] = sortOrder;
     ViewData["NameSortParm"] =
         String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
     ViewData["DateSortParm"] =
         sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" :
"EnrollmentDate";

     if (searchString != null)
     {
         pageNumber = 1;
     }
     else
     {
         searchString = currentFilter;
     }

     ViewData["CurrentFilter"] = searchString;

     var students = from s in _context.Students
                    select s;

     if (!String.IsNullOrEmpty(searchString))
     {
         students = students.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
     }

     if (string.IsNullOrEmpty(sortOrder))
     {
         sortOrder = "LastName";
     }
```

```
      bool descending = false;
      if (sortOrder.EndsWith("_desc"))
      {
          sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
          descending = true;
      }

      if (descending)
      {
          students = students.OrderByDescending(e => EF.Property<object>(e,
sortOrder));
      }
      else
      {
          students = students.OrderBy(e => EF.Property<object>(e,
sortOrder));
      }

      int pageSize = 3;
      return View(await
PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
          pageNumber ?? 1, pageSize));
  }
```

Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials. John Parente and Paul Goldman worked on updating the tutorial for ASP.NET Core 2.2.

Troubleshoot common errors

ContosoUniversity.dll used by another process

Error message:

> Cannot open '...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click **Stop Site**.

Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the `migrations add` command, EF won't find your changes.

Solution:

Run the `migrations remove` command, save your code changes and rerun the `migrations add` command.

Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in `appsettings.json`. The next time you run `database update`, a new database will be created.

To delete a database in SSOX, right-click the database, click **Delete**, and then in the **Delete Database** dialog box select **Close existing connections** and click **OK**.

To delete a database by using the CLI, run the `database drop` CLI command:

.NET CLI

```
dotnet ef database drop
```

Error locating SQL Server instance

Error Message:

> A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify

> that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.

Get the code

Download or view the completed application.

Additional resources

For more information about EF Core, see the Entity Framework Core documentation. A book is also available: Entity Framework Core in Action.

For information on how to deploy a web app, see Host and deploy ASP.NET Core.

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see Overview of ASP.NET Core.

Next steps

In this tutorial, you:

- ✓ Performed raw SQL queries
- ✓ Called a query to return entities
- ✓ Called a query to return other types
- ✓ Called an update query
- ✓ Examined SQL queries
- ✓ Created an abstraction layer
- ✓ Learned about Automatic change detection
- ✓ Learned about EF Core source code and development plans
- ✓ Learned how to use dynamic LINQ to simplify code

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET Core MVC application. This series worked with a new database; an alternative is to reverse engineer a model from an existing database.

Tutorial: EF Core with MVC, existing database

ASP.NET Core and Entity Framework 6

Article • 04/10/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Patrick Goode

Using Entity Framework 6 with ASP.NET Core

Entity Framework Core should be used for new development. The download sample uses Entity Framework 6 (EF6), which can be used to migrate existing apps to ASP.NET Core.

Additional resources

- Entity Framework - Code-Based Configuration

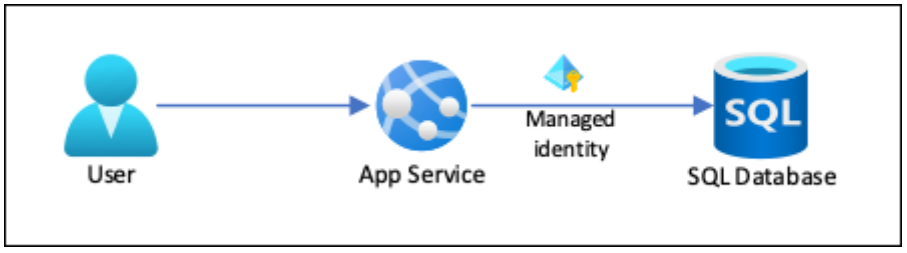
Tutorial: Connect to SQL Database from .NET App Service without secrets using a managed identity

Article • 04/17/2024

App Service provides a highly scalable, self-patching web hosting service in Azure. It also provides a managed identity for your app, which is a turn-key solution for securing access to Azure SQL Database and other Azure services. Managed identities in App Service make your app more secure by eliminating secrets from your app, such as credentials in the connection strings. In this tutorial, you add managed identity to the sample web app you built in one of the following tutorials:

- Tutorial: Build an ASP.NET app in Azure with Azure SQL Database
- Tutorial: Build an ASP.NET Core and Azure SQL Database app in Azure App Service

When you're finished, your sample app will connect to SQL Database securely without the need of username and passwords.



> **Note**
>
> The steps covered in this tutorial support the following versions:
>
> - .NET Framework 4.8 and above
> - .NET 6.0 and above
>
> For guidance for Azure Database for MySQL or Azure Database for PostgreSQL in other language frameworks (Node.js, Python, and Java), see Tutorial: Connect to Azure databases from App Service without secrets using a managed identity.

What you will learn:

- ✓ Enable managed identities
- ✓ Grant SQL Database access to the managed identity

- ✓ Configure Entity Framework to use Microsoft Entra authentication with SQL Database
- ✓ Connect to SQL Database from Visual Studio using Microsoft Entra authentication

> **Note**
>
> Microsoft Entra authentication is *different* from Integrated Windows authentication in on-premises Active Directory (AD DS). AD DS and Microsoft Entra ID use completely different authentication protocols. For more information, see Microsoft Entra Domain Services documentation.

If you don't have an Azure subscription, create an Azure free account before you begin.

Prerequisites

This article continues where you left off in either one of the following tutorials:

- Tutorial: Build an ASP.NET app in Azure with SQL Database
- Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service.

If you haven't already, follow one of the two tutorials first. Alternatively, you can adapt the steps for your own .NET app with SQL Database.

To debug your app using SQL Database as the back end, make sure that you've allowed client connection from your computer. If not, add the client IP by following the steps at Manage server-level IP firewall rules using the Azure portal.

Prepare your environment for the Azure CLI.

- Use the Bash environment in Azure Cloud Shell. For more information, see Quickstart for Bash in Azure Cloud Shell.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.

- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
- Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

1. Grant database access to Microsoft Entra user

First, enable Microsoft Entra authentication to SQL Database by assigning a Microsoft Entra user as the admin of the server. This user is different from the Microsoft account you used to sign up for your Azure subscription. It must be a user that you created, imported, synced, or invited into Microsoft Entra ID. For more information on allowed Microsoft Entra users, see Microsoft Entra features and limitations in SQL Database.

1. If your Microsoft Entra tenant doesn't have a user yet, create one by following the steps at Add or delete users using Microsoft Entra ID.
2. Find the object ID of the Microsoft Entra user using the az ad user list and replace *<user-principal-name>*. The result is saved to a variable.

   Azure CLI

   ```
   $azureaduser=(az ad user list --filter "userPrincipalName eq '<user-principal-name>'" --query '[].id' --output tsv)
   ```

   > **Tip**
   >
   > To see the list of all user principal names in Microsoft Entra ID, run `az ad user list --query '[].userPrincipalName'`.

3. Add this Microsoft Entra user as an Active Directory admin using az sql server ad-admin create command in the Cloud Shell. In the following command, replace *<server-name>* with the server name (without the `.database.windows.net` suffix).

   Azure CLI

   ```
   az sql server ad-admin create --resource-group myResourceGroup --server-name <server-name> --display-name ADMIN --object-id $azureaduser
   ```

For more information on adding an Active Directory admin, see Provision a Microsoft Entra administrator for your server

2. Set up your dev environment

Visual Studio Windows

1. Visual Studio for Windows is integrated with Microsoft Entra authentication. To enable development and debugging in Visual Studio, add your Microsoft Entra user in Visual Studio by selecting **File** > **Account Settings** from the menu, and select **Sign in** or **Add**.
2. To set the Microsoft Entra user for Azure service authentication, select **Tools** > **Options** from the menu, then select **Azure Service Authentication** > **Account Selection**. Select the Microsoft Entra user you added and select **OK**.

For more information about setting up your dev environment for Microsoft Entra authentication, see Azure Identity client library for .NET.

You're now ready to develop and debug your app with the SQL Database as the back end, using Microsoft Entra authentication.

3. Modify your project

> **Note**
>
> **Microsoft.Azure.Services.AppAuthentication** is no longer recommended to use with new Azure SDK. It is replaced with new **Azure Identity client library** available for .NET, Java, TypeScript and Python and should be used for all new development. Information about how to migrate to `Azure Identity` can be found here: AppAuthentication to Azure.Identity Migration Guidance.

The steps you follow for your project depends on whether you're using Entity Framework Core (default for ASP.NET Core) or Entity Framework (default for ASP.NET).

Entity Framework Core

1. In Visual Studio, open the Package Manager Console and add the NuGet package Microsoft.Data.SqlClient:

   PowerShell

```
Install-Package Microsoft.Data.SqlClient -Version 5.1.0
```

2. In the ASP.NET Core and SQL Database tutorial, the `MyDbConnection` connection string in *appsettings.json* isn't used at all yet. The local environment and the Azure environment both get connection strings from their respective environment variables in order to keep connection secrets out of the source file. But now with Active Directory authentication, there are no more secrets. In *appsettings.json*, replace the value of the `MyDbConnection` connection string with:

   JSON

   ```
   "Server=tcp:<server-name>.database.windows.net;Authentication=Active Directory Default; Database=<database-name>;"
   ```

   > **Note**
   >
   > The Active Directory Default authentication type can be used both on your local machine and in Azure App Service. The driver attempts to acquire a token from Microsoft Entra ID using various means. If the app is deployed, it gets a token from the app's system-assigned managed identity. It can also authenticate with a user-assigned managed identity if you include: `User Id=<client-id-of-user-assigned-managed-identity>;` in your connection string. If the app is running locally, it tries to get a token from Visual Studio, Visual Studio Code, and Azure CLI.

   That's everything you need to connect to SQL Database. When you debug in Visual Studio, your code uses the Microsoft Entra user you configured in 2. Set up your dev environment. You'll set up SQL Database later to allow connection from the managed identity of your App Service app. The `DefaultAzureCredential` class caches the token in memory and retrieves it from Microsoft Entra ID just before expiration. You don't need any custom code to refresh the token.

3. Type `Ctrl+F5` to run the app again. The same CRUD app in your browser is now connecting to the Azure SQL Database directly, using Microsoft Entra authentication. This setup lets you run database migrations from Visual Studio.

4. Use managed identity connectivity

Next, you configure your App Service app to connect to SQL Database with a system-assigned managed identity.

> ⓘ Note
>
> The instructions in this section are for a system-assigned identity, To use a user-assigned identity, see Tutorial: Connect to Azure databases from App Service without secrets using a managed identity.

Enable managed identity on app

To enable a managed identity for your Azure app, use the az webapp identity assign command in the Cloud Shell. In the following command, replace *<app-name>*.

Azure CLI

```
az webapp identity assign --resource-group myResourceGroup --name <app-name>
```

> ⓘ Note
>
> To enable managed identity for a deployment slot, add `--slot <slot-name>` and use the name of the slot in *<slot-name>*.

Here's an example of the output:


```
{
  "additionalProperties": {},
  "principalId": "aaaaaaaa-bbbb-cccc-1111-222222222222",
  "tenantId": "aaaabbbb-0000-cccc-1111-dddd2222eeee",
  "type": "SystemAssigned"
}
```


Grant permissions to managed identity

> ⓘ Note
>
> If you want, you can add the identity to an Microsoft Entra group, then grant SQL Database access to the Microsoft Entra group instead of the identity. For example,

the following commands add the managed identity from the previous step to a new group called *myAzureSQLDBAccessGroup*:

Azure CLI

```
$groupid=(az ad group create --display-name myAzureSQLDBAccessGroup --
mail-nickname myAzureSQLDBAccessGroup --query objectId --output tsv)
$msiobjectid=(az webapp identity show --resource-group myResourceGroup
--name <app-name> --query principalId --output tsv)
az ad group member add --group $groupid --member-id $msiobjectid
az ad group member list -g $groupid
```

1. In the Cloud Shell, sign in to SQL Database by using the SQLCMD command. Replace <*server-name*> with your server name, <*db-name*> with the database name your app uses, and <*aad-user-name*> and <*aad-password*> with your Microsoft Entra user's credentials.

   Bash

   ```
   sqlcmd -S <server-name>.database.windows.net -d <db-name> -U <aad-user-
   name> -P "<aad-password>" -G -l 30
   ```

2. In the SQL prompt for the database you want, run the following commands to grant the minimum permissions your app needs. For example,

   SQL

   ```
   CREATE USER [<identity-name>] FROM EXTERNAL PROVIDER;
   ALTER ROLE db_datareader ADD MEMBER [<identity-name>];
   ALTER ROLE db_datawriter ADD MEMBER [<identity-name>];
   ALTER ROLE db_ddladmin ADD MEMBER [<identity-name>];
   GO
   ```

   <*identity-name*> is the name of the managed identity in Microsoft Entra ID. If the identity is system-assigned, the name is always the same as the name of your App Service app. For a deployment slot, the name of its system-assigned identity is <*app-name*>/slots/<*slot-name*>. To grant permissions for a Microsoft Entra group, use the group's display name instead (for example, *myAzureSQLDBAccessGroup*).

3. Type `EXIT` to return to the Cloud Shell prompt.

   Note

The back-end services of managed identities also maintains a token cache that updates the token for a target resource only when it expires. If you make a mistake configuring your SQL Database permissions and try to modify the permissions *after* trying to get a token with your app, you don't actually get a new token with the updated permissions until the cached token expires.

> **Note**
>
> Microsoft Entra ID and managed identities are not supported for on-premises SQL Server.

Modify connection string

Remember that the same changes you made in *Web.config* or *appsettings.json* works with the managed identity, so the only thing to do is to remove the existing connection string in App Service, which Visual Studio created deploying your app the first time. Use the following command, but replace <*app-name*> with the name of your app.

Azure CLI

```
az webapp config connection-string delete --resource-group myResourceGroup -
-name <app-name> --setting-names MyDbConnection
```

5. Publish your changes

All that's left now is to publish your changes to Azure.

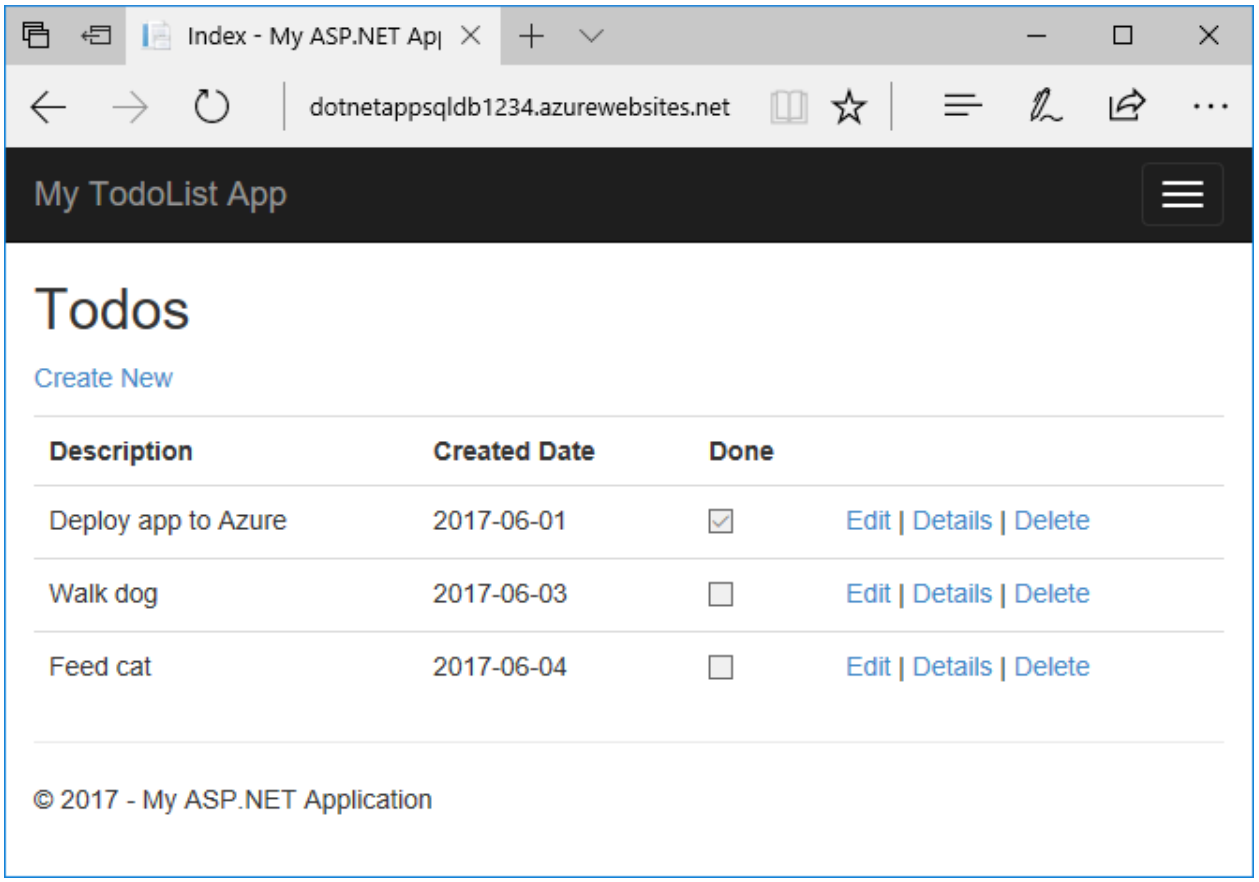
ASP.NET Core

If you came from Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service, publish your changes using Git, with the following commands:

Bash

```
git commit -am "configure managed identity"
git push azure main
```

When the new webpage shows your to-do list, your app is connecting to the database using the managed identity.



You should now be able to edit the to-do list as before.

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, delete the resource group by running the following command in the Cloud Shell:

Azure CLI

```
az group delete --name myResourceGroup
```

This command may take a minute to run.

Next steps

What you learned:

- ✓ Enable managed identities
- ✓ Grant SQL Database access to the managed identity
- ✓ Configure Entity Framework to use Microsoft Entra authentication with SQL Database
- ✓ Connect to SQL Database from Visual Studio using Microsoft Entra authentication

Secure with custom domain and certificate

Tutorial: Connect an App Service app to SQL Database on behalf of the signed-in user

Tutorial: Connect to Azure databases from App Service without secrets using a managed identity

Tutorial: Connect to Azure services that don't support managed identities (using Key Vault)

Tutorial: Isolate back-end communication with Virtual Network integration

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Adding Azure Storage by using Visual Studio Connected Services

Article • 10/31/2024

With Visual Studio, you can connect any of the following to Azure Storage by using the **Connected Services** feature:

- .NET Framework console app
- ASP.NET Model-View-Controller (MVC) (.NET Framework)
- ASP.NET Core
- .NET Core (including console app, WPF, Windows Forms, class library)
- .NET Core Worker Role
- Azure Functions
- Universal Windows Platform App
- Xamarin
- Cordova

The connected service functionality adds all the needed references and connection code to your project, and modifies your configuration files appropriately.

Prerequisites

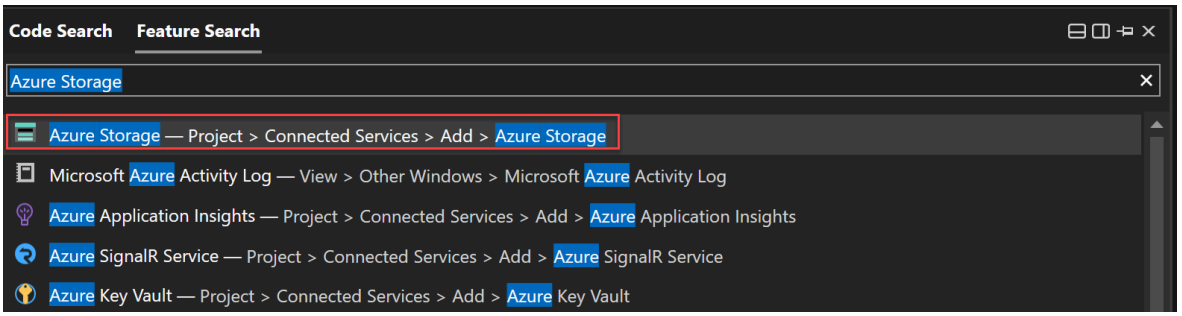
- Visual Studio (see [Visual Studio downloads] (https://visualstudio.microsoft.com/downloads/?cid=learn-onpage-download-cta)) with the **Azure development** workload installed.
- A project of one of the supported types
- An Azure account. If you don't have an Azure account, activate your Azure benefits for Visual Studio subscribers or sign up for a free trial.

Connect to Azure Storage using Connected Services

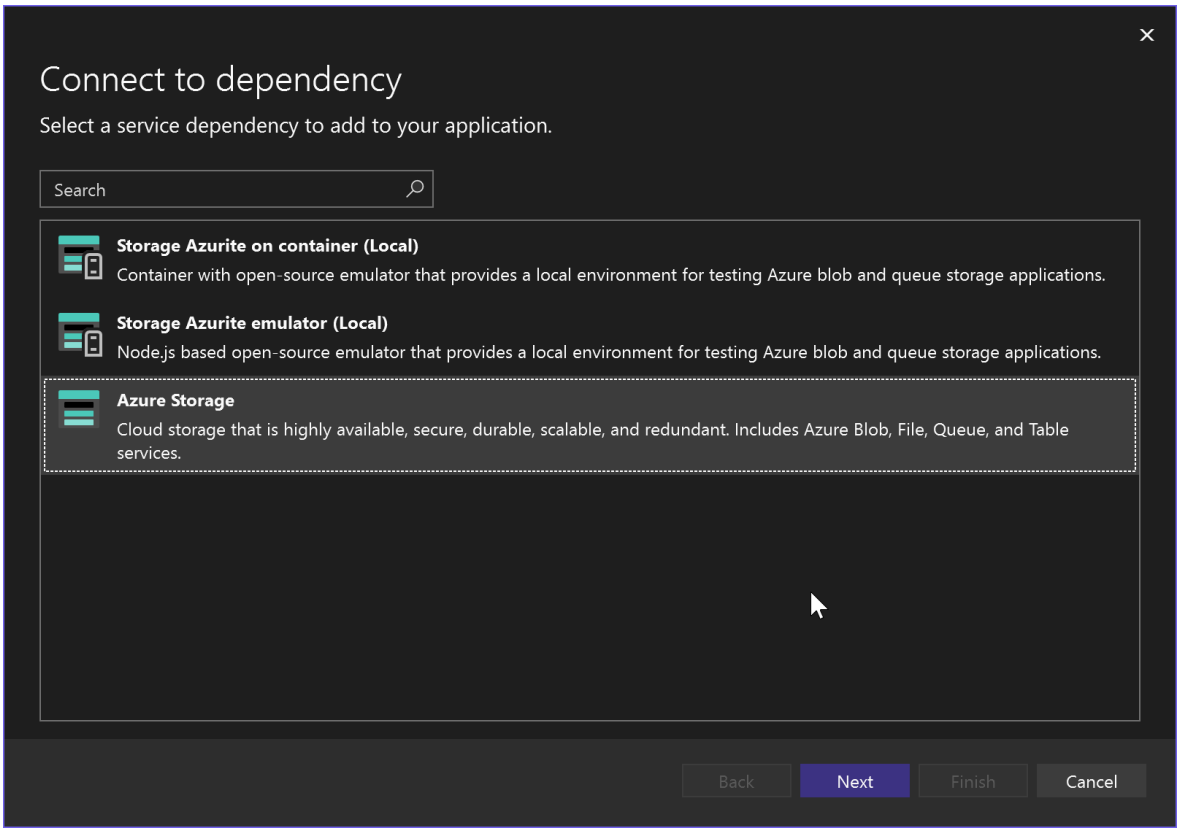
1. Open your project in Visual Studio.

2. Press **Ctrl**+**Q** (or use the **Search** button in the Visual Studio IDE to the right of the main menu bar).

3. In **Feature search**, enter `Azure Storage`, and choose **Azure Storage - Project > Connected Services > Add > Azure Storage**.



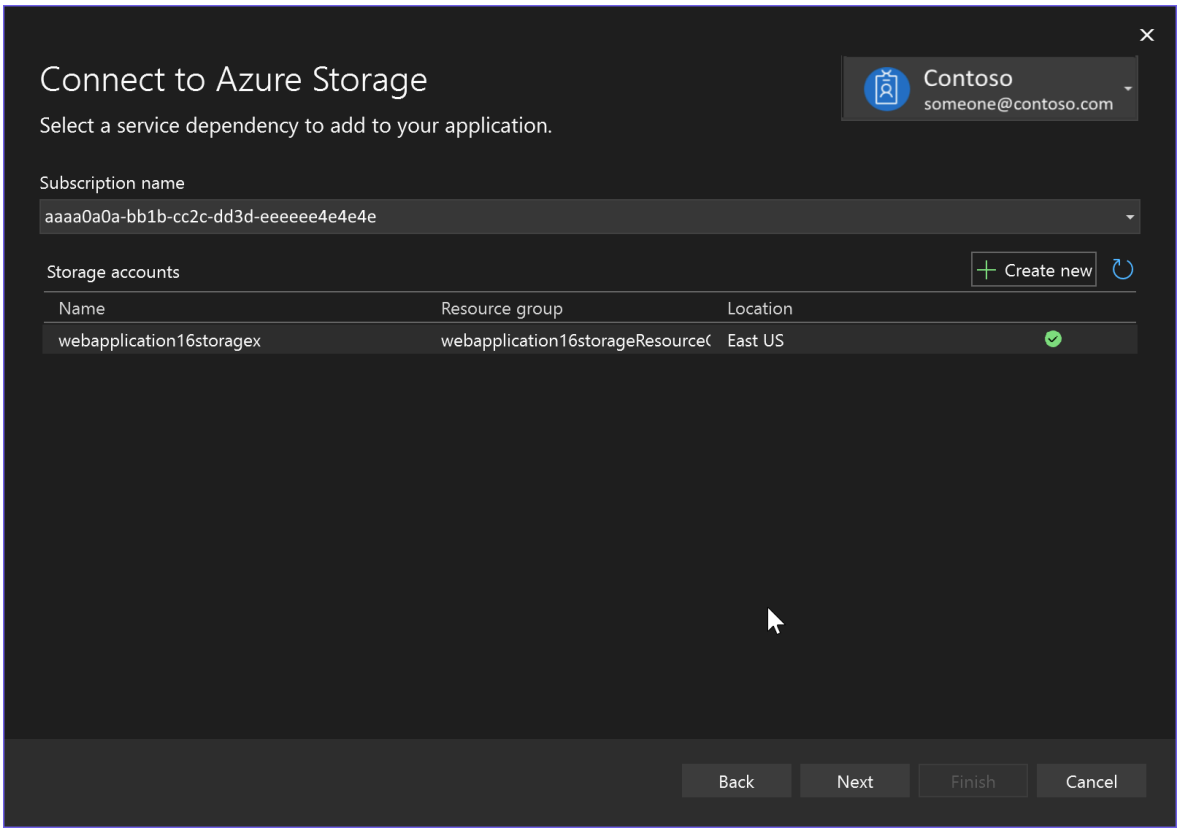
4. In the **Connect to dependency** page, select **Azure Storage**, and then select **Next**.



   If you aren't signed in already, sign in to your Azure account. If you don't have an Azure account, you can sign up for a free trial.

5. In the **Connect to Azure Storage** screen, select an existing storage account, and select **Next**.

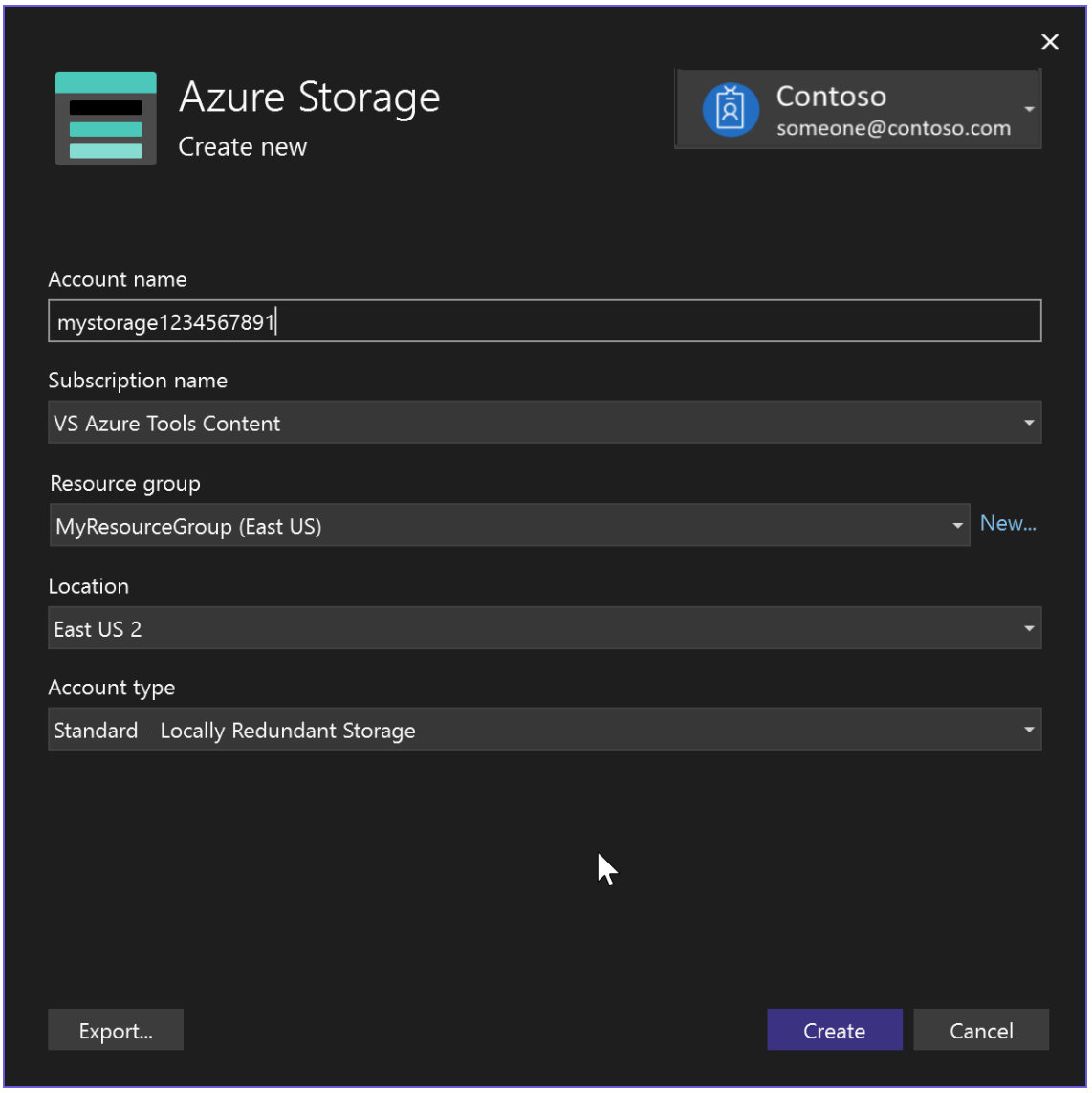
   If you need to create a storage account, go to the next step. Otherwise, skip to the following step.



6. To create a storage account:

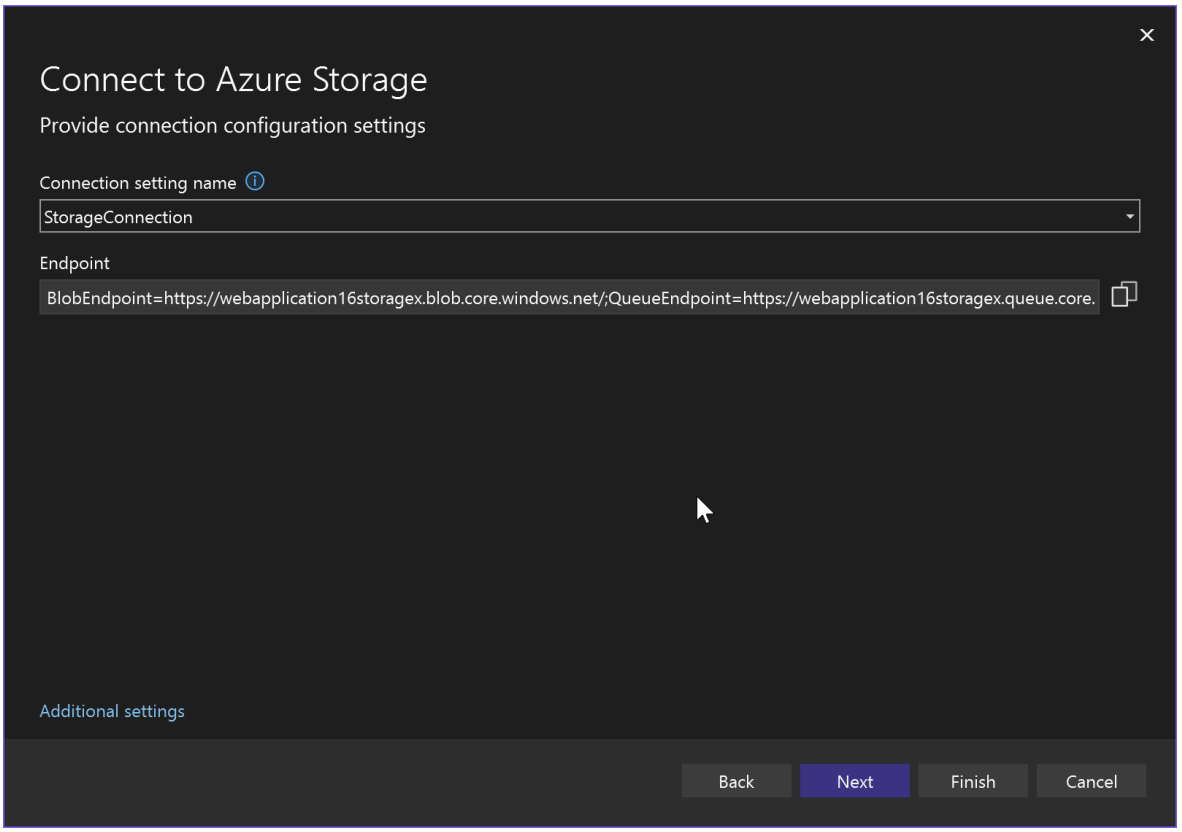
   a. Select **Create new** by the green plus sign.

   b. Fill out the **Azure Storage: Create new** dialog, and select **Create**.

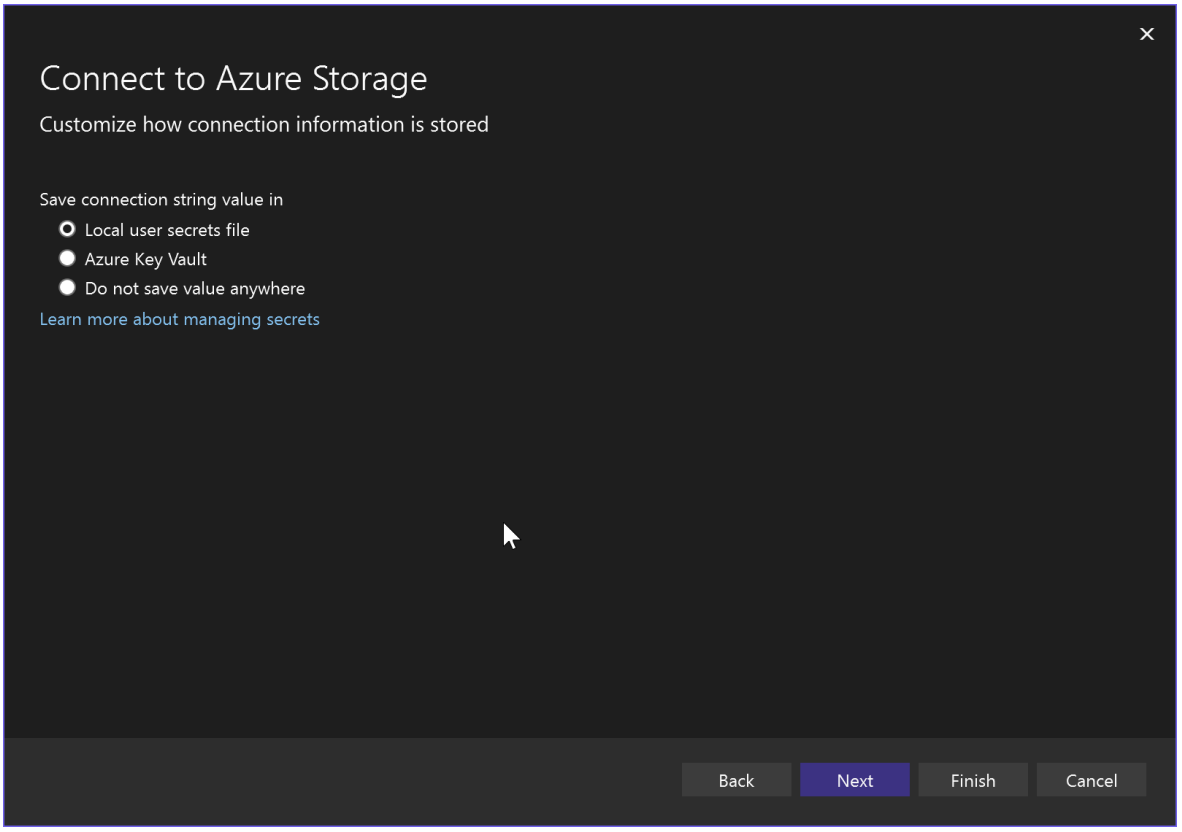


c. When the **Azure Storage** dialog is displayed, the new storage account appears in the list. Select the new storage account in the list, and select **Next**.

7. Enter a connection string setting name. The setting name references the name of the connection string setting as it appears in the *secrets.json* file, or in Azure Key Vault.



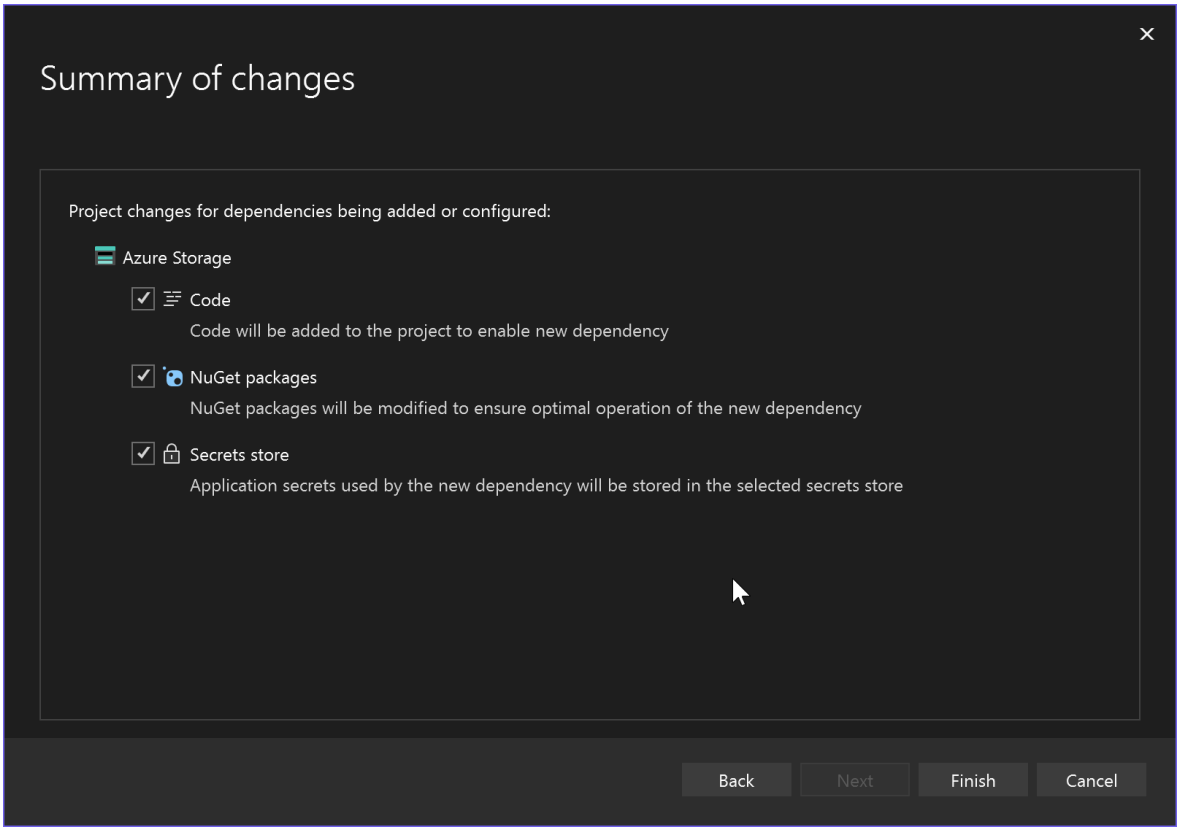
8. Choose whether you want the connection string stored in a local secrets file, in Azure Key Vault, or not stored anywhere.



> **⊗ Caution**
>
> If you're using a version of Visual Studio earlier than Visual Studio 17.12, and you choose to use a *secrets.json* file, you must take security precautions, since

the connection string in the local secrets.json file could be exposed. If you're using Visual Studio 2022 version 17.12 or later, this procedure produces more secure result, because it yields a connection setting value, instead of a connection string with authentication credentials.

9. The **Summary of changes** screen shows all the modifications that will be made to your project if you complete the process. If the changes look OK, choose **Finish**.



10. The storage connected service appears under the **Connected Services** node of your project.

Understand authentication

After you run the previous procedure, your app is set up to use authentication to access the storage account. The connection information for this authentication are stored locally, if you chose the *secrets.json* method, or in your Azure key vault.

If you used the *secrets.json* file, open the file by using the three dots next to **Secrets.json** on the **Connected Services** tab to open a menu, and choose **Manage user secrets**. With Visual Studio 2022 version 17.12 and later, this file contains settings that reference a URI to obtain the secure connection string, rather than the connection string itself.

JSON

```
{
  "StorageConnection:blobServiceUri":
"https://webapplication16storagex.blob.core.windows.net/",
  "StorageConnection:queueServiceUri":
"https://webapplication16storagex.queue.core.windows.net/",
  "StorageConnection:tableServiceUri":
"https://webapplication16storagex.table.core.windows.net/"
}
```

With these settings in Visual Studio 17.12 and later, authentication is automatic and flexible. When you run or debug locally from Visual Studio, your Azure credentials saved by Visual Studio are used to access the Azure Storage account. If you launch your app from the command-line, you first need to sign in using the Azure CLI, and those credentials are automatically detected and used. But when your app is deployed to Azure and runs in Azure, it uses managed identity, without any code changes. The authentication works in all hosting environments because the Azure Identity APIs check for all chained credentials in sequence and use them when they're found. See DefaultAzureCredential.

Next steps

Azure Storage supports blobs and queues, as well as other features.

To learn about working with blobs, you can continue with the quickstart for blob storage, but instead of starting at the beginning, you can start at Azure blobs quickstart (.NET) - Code examples.

To learn about working with queues, start at Azure Queue Storage quickstart (.NET) - Code examples.

Related content

- Azure Storage forum
- Azure Storage documentation

Feedback

Quickstart: Azure Blob Storage client library for .NET

Article • 09/13/2024

> **Note**
>
> The **Build from scratch** option walks you step by step through the process of creating a new project, installing packages, writing the code, and running a basic console app. This approach is recommended if you want to understand all the details involved in creating an app that connects to Azure Blob Storage. If you prefer to automate deployment tasks and start with a completed project, choose Start with a template.

Get started with the Azure Blob Storage client library for .NET. Azure Blob Storage is Microsoft's object storage solution for the cloud, and is optimized for storing massive amounts of unstructured data.

In this article, you follow steps to install the package and try out example code for basic tasks.

API reference documentation | Library source code | Package (NuGet) | Samples

This video shows you how to start using the Azure Blob Storage client library for .NET. https://learn-video.azurefd.net/vod/player?id=cdae65e7-1892-48fe-934a-70edfbe147be&locale=en-us&embedUrl=%2Fazure%2Fstorage%2Fblobs%2Fstorage-quickstart-blobs-dotnet

The steps in the video are also described in the following sections.

Prerequisites

- Azure subscription - create one for free
- Azure storage account - create a storage account
- Latest .NET SDK for your operating system. Be sure to get the SDK and not the runtime.

Setting up

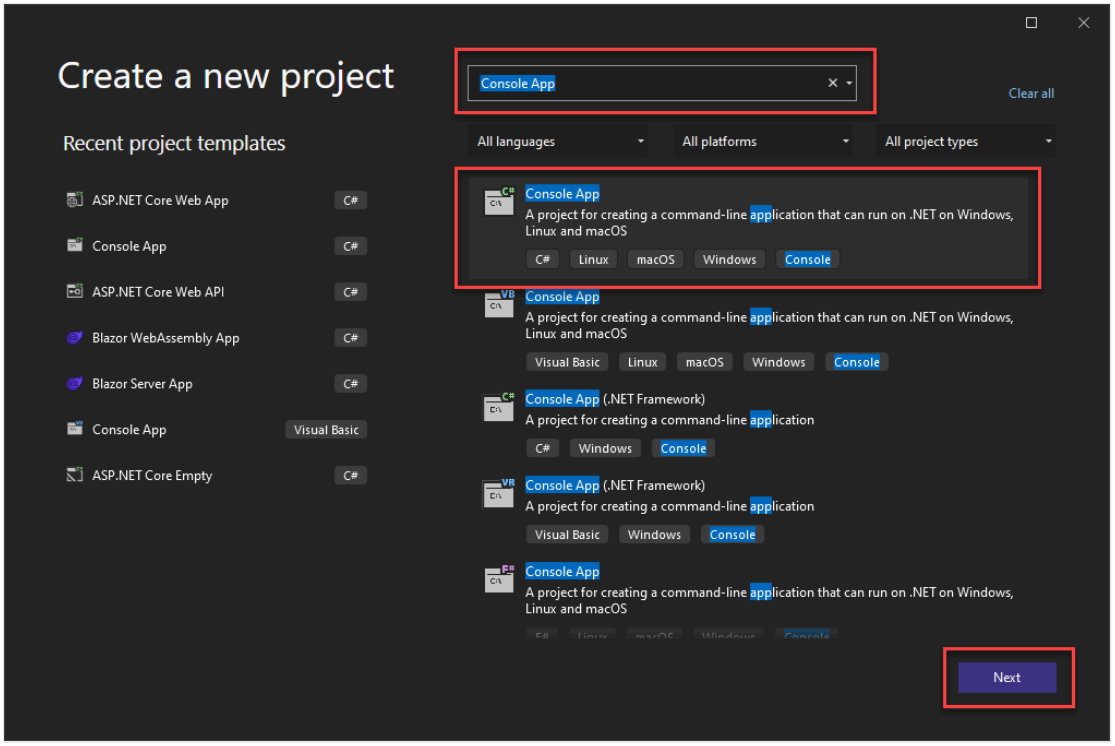
This section walks you through preparing a project to work with the Azure Blob Storage client library for .NET.

Create the project

Create a .NET console app using either the .NET CLI or Visual Studio 2022.

Visual Studio 2022

1. At the top of Visual Studio, navigate to **File** > **New** > **Project...**
2. In the dialog window, enter *console app* into the project template search box and select the first result. Choose **Next** at the bottom of the dialog.



3. For the **Project Name**, enter *BlobQuickstart*. Leave the default values for the rest of the fields and select **Next**.
4. For the **Framework**, ensure the latest installed version of .NET is selected. Then choose **Create**. The new project opens inside the Visual Studio environment.

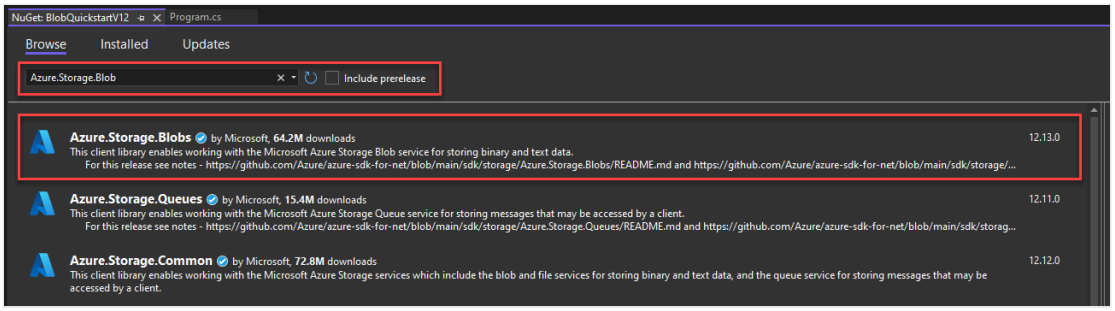
Install the package

To interact with Azure Blob Storage, install the Azure Blob Storage client library for .NET.

Visual Studio 2022

1. In **Solution Explorer**, right-click the **Dependencies** node of your project. Select **Manage NuGet Packages**.

2. In the resulting window, search for *Azure.Storage.Blobs*. Select the appropriate result, and select **Install**.



Set up the app code

Replace the starting code in the `Program.cs` file so that it matches the following example, which includes the necessary `using` statements for this exercise.

C#

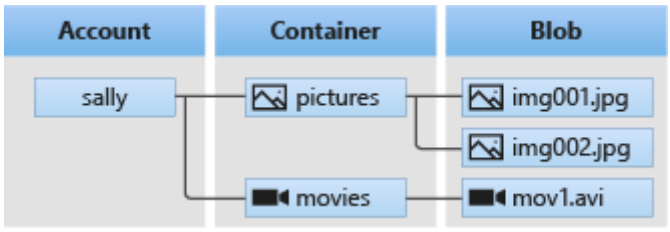
```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using System;
using System.IO;

// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Object model

Azure Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data doesn't adhere to a particular data model or definition, such as text or binary data. Blob storage offers three types of resources:

- The storage account
- A container in the storage account
- A blob in the container

The following diagram shows the relationship between these resources.



Use the following .NET classes to interact with these resources:

- BlobServiceClient: The `BlobServiceClient` class allows you to manipulate Azure Storage resources and blob containers.
- BlobContainerClient: The `BlobContainerClient` class allows you to manipulate Azure Storage containers and their blobs.
- BlobClient: The `BlobClient` class allows you to manipulate Azure Storage blobs.

Code examples

The sample code snippets in the following sections demonstrate how to perform the following tasks with the Azure Blob Storage client library for .NET:

- Authenticate to Azure and authorize access to blob data
- Create a container
- Upload a blob to a container
- List blobs in a container
- Download a blob
- Delete a container

> ⓘ **Important**
>
> Make sure you've installed the correct NuGet packages and added the necessary using statements in order for the code samples to work, as described in the **setting up** section.

Authenticate to Azure and authorize access to blob data

Application requests to Azure Blob Storage must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code, including Blob Storage.

You can also authorize requests to Azure Blob Storage by using the account access key. However, this approach should be used with caution. Developers must be diligent to

never expose the access key in an unsecure location. Anyone who has the access key is able to authorize requests against the storage account, and effectively has access to all the data. `DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

Passwordless (Recommended)

`DefaultAzureCredential` is a class provided by the Azure Identity client library for .NET, which you can learn more about on the DefaultAzureCredential overview. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` looks for credentials can be found in the Azure Identity library overview.

For example, your app can authenticate using your Visual Studio sign-in credentials with when developing locally. Your app can then use a managed identity once it has been deployed to Azure. No code changes are required for this transition.

Assign roles to your Microsoft Entra user account

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the scope overview page.

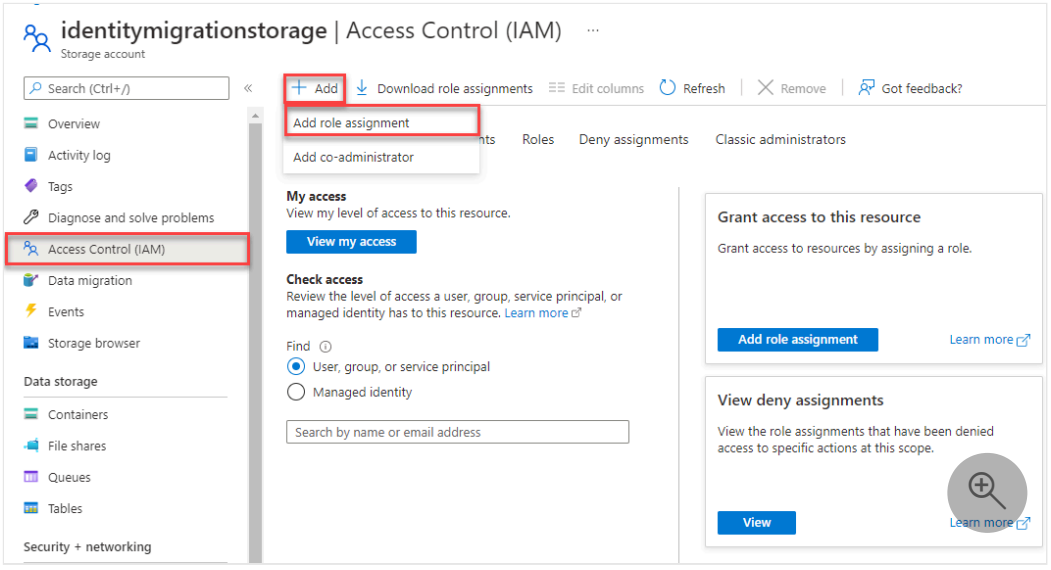
In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the Principle of Least Privilege. This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

> **Important**
>
> In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the

dialog.

8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Sign in and connect your app code to Azure using DefaultAzureCredential

You can authorize access to data in your storage account using the following steps:

1. For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

   Azure CLI

   Sign-in to Azure through the Azure CLI using the following command:

   Azure CLI

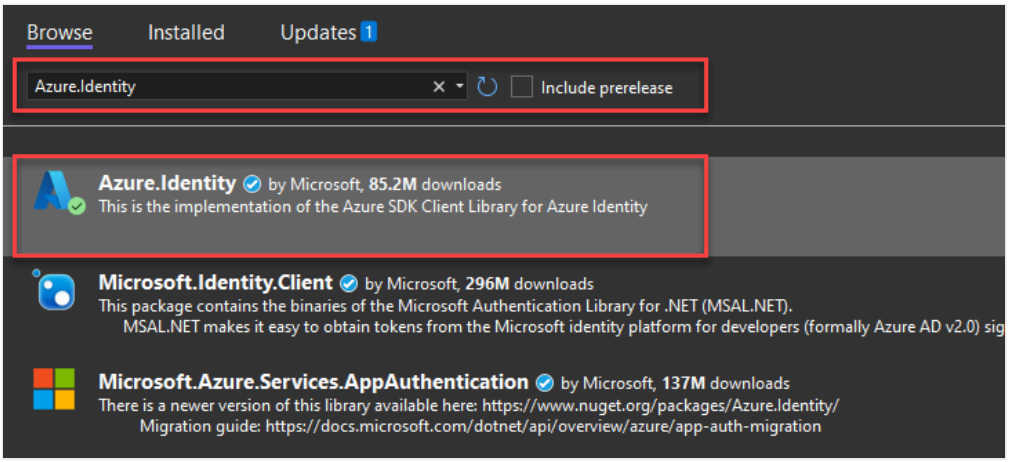
   ```
   az login
   ```

2. To use `DefaultAzureCredential`, add the **Azure.Identity** package to your application.

   Visual Studio

   a. In **Solution Explorer**, right-click the **Dependencies** node of your project. Select **Manage NuGet Packages**.

   b. In the resulting window, search for *Azure.Identity*. Select the appropriate result, and select **Install**.

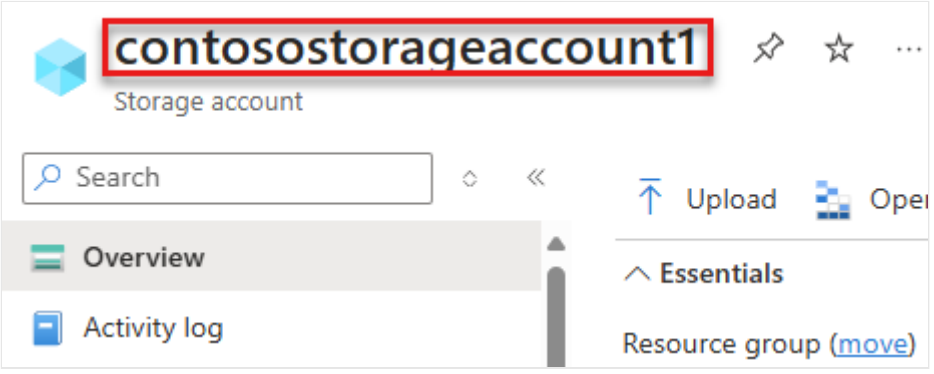


3. Update your *Program.cs* code to match the following example. When the code is run on your local workstation during development, it will use the developer credentials of the prioritized tool you're logged into to authenticate to Azure, such as the Azure CLI or Visual Studio.

C#

```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using System;
using System.IO;
using Azure.Identity;

// TODO: Replace <storage-account-name> with your actual storage
account name
var blobServiceClient = new BlobServiceClient(
        new Uri("https://<storage-account-
name>.blob.core.windows.net"),
        new DefaultAzureCredential());
```

4. Make sure to update the storage account name in the URI of your `BlobServiceClient`. The storage account name can be found on the overview page of the Azure portal.



> ⓘ Note

When deployed to Azure, this same code can be used to authorize requests to Azure Storage from an application running in Azure. However, you'll need to enable managed identity on your app in Azure. Then configure your storage account to allow that managed identity to connect. For detailed instructions on configuring this connection between Azure services, see the Auth from Azure-hosted apps tutorial.

Create a container

Create a new container in your storage account by calling the CreateBlobContainerAsync method on the `blobServiceClient` object. In this example, the code appends a GUID value to the container name to ensure that it's unique.

Add the following code to the end of the `Program.cs` file:

C#

```
// TODO: Replace <storage-account-name> with your actual storage account name
var blobServiceClient = new BlobServiceClient(
        new Uri("https://<storage-account-name>.blob.core.windows.net"),
        new DefaultAzureCredential());

//Create a unique name for the container
string containerName = "quickstartblobs" + Guid.NewGuid().ToString();

// Create the container and return a container client object
BlobContainerClient containerClient = await
blobServiceClient.CreateBlobContainerAsync(containerName);
```

To learn more about creating a container, and to explore more code samples, see Create a blob container with .NET.

ⓘ Important

Container names must be lowercase. For more information about naming containers and blobs, see Naming and Referencing Containers, Blobs, and Metadata.

Upload a blob to a container

Upload a blob to a container using UploadAsync. The example code creates a text file in the local *data* directory to upload to the container.

Add the following code to the end of the `Program.cs` file:

C#

```
// Create a local file in the ./data/ directory for uploading and
downloading
string localPath = "data";
Directory.CreateDirectory(localPath);
string fileName = "quickstart" + Guid.NewGuid().ToString() + ".txt";
string localFilePath = Path.Combine(localPath, fileName);

// Write text to the file
await File.WriteAllTextAsync(localFilePath, "Hello, World!");

// Get a reference to a blob
BlobClient blobClient = containerClient.GetBlobClient(fileName);

Console.WriteLine("Uploading to Blob storage as blob:\n\t {0}\n",
blobClient.Uri);

// Upload data from the local file, overwrite the blob if it already exists
await blobClient.UploadAsync(localFilePath, true);
```

To learn more about uploading blobs, and to explore more code samples, see Upload a blob with .NET.

List blobs in a container

List the blobs in the container by calling the GetBlobsAsync method.

Add the following code to the end of the `Program.cs` file:

C#

```
Console.WriteLine("Listing blobs...");

// List all blobs in the container
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())
{
    Console.WriteLine("\t" + blobItem.Name);
}
```

To learn more about listing blobs, and to explore more code samples, see List blobs with .NET.

Download a blob

Download the blob we created earlier by calling the DownloadToAsync method. The example code appends the string "DOWNLOADED" to the file name so that you can see both files in local file system.

Add the following code to the end of the `Program.cs` file:

C#

```
// Download the blob to a local file
// Append the string "DOWNLOADED" before the .txt extension
// so you can compare the files in the data directory
string downloadFilePath = localFilePath.Replace(".txt", "DOWNLOADED.txt");

Console.WriteLine("\nDownloading blob to\n\t{0}\n", downloadFilePath);

// Download the blob's contents and save it to a file
await blobClient.DownloadToAsync(downloadFilePath);
```

To learn more about downloading blobs, and to explore more code samples, see Download a blob with .NET.

Delete a container

The following code cleans up the resources the app created by deleting the container using DeleteAsync. The code example also deletes the local files created by the app.

The app pauses for user input by calling `Console.ReadLine` before it deletes the blob, container, and local files. This is a good chance to verify that the resources were created correctly, before they're deleted.

Add the following code to the end of the `Program.cs` file:

C#

```
// Clean up
Console.Write("Press any key to begin clean up");
Console.ReadLine();

Console.WriteLine("Deleting blob container...");
await containerClient.DeleteAsync();

Console.WriteLine("Deleting the local source and downloaded files...");
File.Delete(localFilePath);
File.Delete(downloadFilePath);
```

```
Console.WriteLine("Done");
```

To learn more about deleting a container, and to explore more code samples, see Delete and restore a blob container with .NET.

The completed code

After completing these steps, the code in your `Program.cs` file should now resemble the following:

Passwordless (Recommended)

C#

```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using Azure.Identity;

// TODO: Replace <storage-account-name> with your actual storage account
name
var blobServiceClient = new BlobServiceClient(
        new Uri("https://<storage-account-name>.blob.core.windows.net"),
        new DefaultAzureCredential());

//Create a unique name for the container
string containerName = "quickstartblobs" + Guid.NewGuid().ToString();

// Create the container and return a container client object
BlobContainerClient containerClient = await
blobServiceClient.CreateBlobContainerAsync(containerName);

// Create a local file in the ./data/ directory for uploading and
downloading
string localPath = "data";
Directory.CreateDirectory(localPath);
string fileName = "quickstart" + Guid.NewGuid().ToString() + ".txt";
string localFilePath = Path.Combine(localPath, fileName);

// Write text to the file
await File.WriteAllTextAsync(localFilePath, "Hello, World!");

// Get a reference to a blob
BlobClient blobClient = containerClient.GetBlobClient(fileName);

Console.WriteLine("Uploading to Blob storage as blob:\n\t {0}\n",
blobClient.Uri);

// Upload data from the local file
await blobClient.UploadAsync(localFilePath, true);
```

```
Console.WriteLine("Listing blobs...");

// List all blobs in the container
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())
{
    Console.WriteLine("\t" + blobItem.Name);
}

// Download the blob to a local file
// Append the string "DOWNLOADED" before the .txt extension
// so you can compare the files in the data directory
string downloadFilePath = localFilePath.Replace(".txt",
"DOWNLOADED.txt");

Console.WriteLine("\nDownloading blob to\n\t{0}\n", downloadFilePath);

// Download the blob's contents and save it to a file
await blobClient.DownloadToAsync(downloadFilePath);

// Clean up
Console.Write("Press any key to begin clean up");
Console.ReadLine();

Console.WriteLine("Deleting blob container...");
await containerClient.DeleteAsync();

Console.WriteLine("Deleting the local source and downloaded files...");
File.Delete(localFilePath);
File.Delete(downloadFilePath);

Console.WriteLine("Done");
```

Run the code

This app creates a test file in your local *data* folder and uploads it to Blob storage. The example then lists the blobs in the container and downloads the file with a new name so that you can compare the old and new files.

If you're using Visual Studio, press F5 to build and run the code and interact with the console app. If you're using the .NET CLI, navigate to your application directory, then build and run the application.

Console

```
dotnet build
```

Console

```
dotnet run
```

The output of the app is similar to the following example (GUID values omitted for readability):

Output

```
Azure Blob Storage - .NET quickstart sample

Uploading to Blob storage as blob:

https://mystorageacct.blob.core.windows.net/quickstartblobsGUID/quickstartGU
ID.txt

Listing blobs...
        quickstartGUID.txt

Downloading blob to
        ./data/quickstartGUIDDOWNLOADED.txt

Press any key to begin clean up
Deleting blob container...
Deleting the local source and downloaded files...
Done
```

Before you begin the clean-up process, check your *data* folder for the two files. You can open them and observe that they're identical.

Clean up resources

After you verify the files and finish testing, press the **Enter** key to delete the test files along with the container you created in the storage account. You can also use Azure CLI to delete resources.

Next step

Azure Storage samples and developer guides for .NET

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Feedback

Azure Queue storage documentation

Azure Queue storage is a service for storing large numbers of messages. Access messages from anywhere in the world via HTTP or HTTPS.

About Azure queues

OVERVIEW

What are Azure queues?

Get started

QUICKSTART

Create a queue and add a message with the Azure portal

Develop with queues

QUICKSTART

Azure Queue storage for .NET

Azure Queue storage for Java

Azure Queue storage for Python

Azure Queue storage for JavaScript

HOW-TO GUIDE

Get started with Azure Queue storage using .NET

How to use Queue storage from Java

How to use Queue storage from Node.js

How to use Queue Storage from C++

How to use Queue storage from Python

Manage queues

HOW-TO GUIDE

Perform Azure Queue storage operations with Azure PowerShell

Tutorial

TUTORIAL

Work with Azure storage queues (.NET)

What is Azure Table storage?

Article • 11/29/2022

> **Tip**
>
> The content in this article applies to the original Azure Table storage. However, the same concepts apply to the newer Azure Cosmos DB for Table, which offers higher performance and availability, global distribution, and automatic secondary indexes. It is also available in a consumption-based **serverless** mode. There are some **feature differences** between Table API in Azure Cosmos DB and Azure Table storage. For more information, see **Azure Cosmos DB for Table**. For ease of development, we now provide a unified **Azure Tables SDK** that can be used to target both Azure Table storage and Azure Cosmos DB for Table.

Azure Table storage is a service that stores non-relational structured data (also known as structured NoSQL data) in the cloud, providing a key/attribute store with a schemaless design. Because Table storage is schemaless, it's easy to adapt your data as the needs of your application evolve. Access to Table storage data is fast and cost-effective for many types of applications, and is typically lower in cost than traditional SQL for similar volumes of data.

You can use Table storage to store flexible datasets like user data for web applications, address books, device information, or other types of metadata your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

What is Table storage

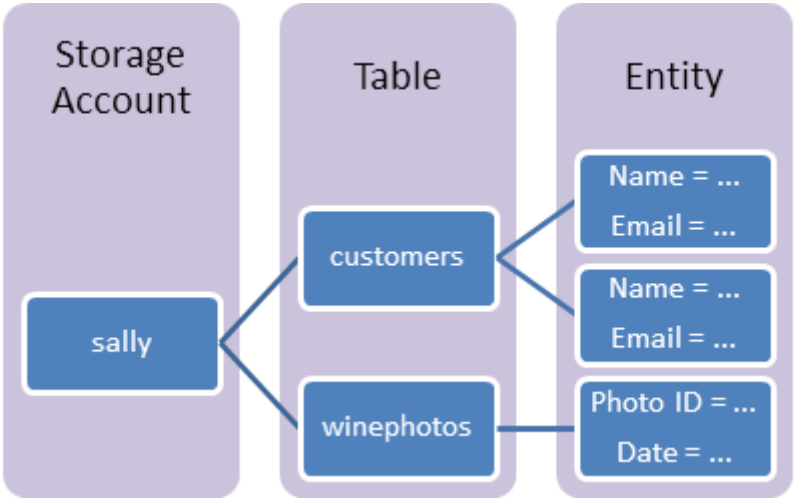
Azure Table storage stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of Table storage include:

- Storing TBs of structured data capable of serving web scale applications
- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use Table storage to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table storage concepts

Table storage contains the following components:



- **URL format:** Azure Table Storage accounts use this format: `http://<storage account>.table.core.windows.net/<table>`

  You can address Azure tables directly using this address with the OData protocol. For more information, see OData.org.

- **Accounts:** All access to Azure Storage is done through a storage account. For more information about storage accounts, see Storage account overview.

  All access to Azure Cosmos DB is done through an Azure Cosmos DB for Table account. For more information, see Create an Azure Cosmos DB for Table account.

- **Table**: A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties.

- **Entity**: An entity is a set of properties, similar to a database row. An entity in Azure Storage can be up to 1MB in size. An entity in Azure Cosmos DB can be up to 2MB in size.

- **Properties**: A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has three system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see Understanding the Table Service Data Model.

Next steps

- Microsoft Azure Storage Explorer is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, macOS, and Linux.
- Get started with Azure Table Storage in .NET
- View the Table service reference documentation for complete details about available APIs:
  - Storage Client Library for .NET reference
  - REST API reference

Feedback

Host and deploy ASP.NET Core

Article • 09/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

In general, to deploy an ASP.NET Core app to a hosting environment:

- Deploy the published app to a folder on the hosting server.
- Set up a process manager that starts the app when requests arrive and restarts the app after it crashes or the server reboots.
- For configuration of a reverse proxy, set up a reverse proxy to forward requests to the app.

For Blazor host and deploy guidance, which adds to or supersedes the guidance in this node, see Host and deploy ASP.NET Core Blazor.

Publish to a folder

The dotnet publish command compiles app code and copies the files required to run the app into a *publish* folder. When deploying from Visual Studio, the `dotnet publish` step occurs automatically before the files are copied to the deployment destination.

Run the published app locally

To run the published app locally, run `dotnet <ApplicationName>.dll` from the *publish* folder.

Publish settings files

`*.json` files are published by default. To publish other settings files, specify them in an <ItemGroup><Content Include= ... /> element in the project file. The following example publishes XML files:

XML

```xml
<ItemGroup>
  <Content Include="**\*.xml" Exclude="bin\**\*;obj\**\*"
    CopyToOutputDirectory="PreserveNewest" />
</ItemGroup>
```

Folder contents

The *publish* folder contains one or more app assembly files, dependencies, and optionally the .NET runtime.

A .NET Core app can be published as *self-contained deployment* or *framework-dependent deployment*. If the app is self-contained, the assembly files that contain the .NET runtime are included in the *publish* folder. If the app is framework-dependent, the .NET runtime files aren't included because the app has a reference to a version of .NET that's installed on the server. The default deployment model is framework-dependent. For more information, see .NET Core application deployment.

In addition to *.exe* and *.dll* files, the *publish* folder for an ASP.NET Core app typically contains configuration files, static assets, and MVC views. For more information, see ASP.NET Core directory structure.

Set up a process manager

An ASP.NET Core app is a console app that must be started when a server boots and restarted if it crashes. To automate starts and restarts, a process manager is required. The most common process managers for ASP.NET Core are:

- Linux
  - Nginx
- Windows
  - IIS
  - Windows Service

Set up a reverse proxy

If the app uses the Kestrel server, Nginx, or IIS can be used as a reverse proxy server. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.

Either configuration—with or without a reverse proxy server—is a supported hosting configuration. For more information, see When to use Kestrel with a reverse proxy.

Proxy server and load balancer scenarios

Additional configuration might be required for apps hosted behind proxy servers and load balancers. Without additional configuration, an app might not have access to the scheme (HTTP/HTTPS) and the remote IP address where a request originated. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Use Visual Studio and MSBuild to automate deployments

Deployment often requires additional tasks besides copying the output from dotnet publish to a server. For example, extra files might be required or excluded from the *publish* folder. Visual Studio uses MSBuild for web deployment, and MSBuild can be customized to do many other tasks during deployment. For more information, see Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment and the Using MSBuild and Team Foundation Build book.

By using the Publish Web feature apps can be deployed directly from Visual Studio to the Azure App Service. Azure DevOps Services supports continuous deployment to Azure App Service. For more information, see DevOps for ASP.NET Core Developers.

Publish to Azure

See Publish an ASP.NET Core app to Azure with Visual Studio for instructions on how to publish an app to Azure using Visual Studio. An additional example is provided by Create an ASP.NET Core web app in Azure.

Publish with MSDeploy on Windows

See Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment for instructions on how to publish an app with a Visual Studio publish profile, including from a Windows command prompt using the dotnet msbuild command.

Internet Information Services (IIS)

For deployments to Internet Information Services (IIS) with configuration provided by the *web.config* file, see the articles under Host ASP.NET Core on Windows with IIS.

Host in a web farm

For information on configuration for hosting ASP.NET Core apps in a web farm environment (for example, deployment of multiple instances of your app for scalability), see Host ASP.NET Core in a web farm.

Host on Docker

For more information, see Host ASP.NET Core in Docker containers.

Perform health checks

Use Health Check Middleware to perform health checks on an app and its dependencies. For more information, see Health checks in ASP.NET Core.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- ASP.NET Hosting

Deploy ASP.NET Core apps to Azure App Service

Article • 09/19/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

Azure App Service is a Microsoft cloud computing platform service for hosting web apps, including ASP.NET Core.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Useful resources

App Service Documentation is the home for Azure Apps documentation, tutorials, samples, how-to guides, and other resources. Two notable tutorials that pertain to hosting ASP.NET Core apps are:

Create an ASP.NET Core web app in Azure
Use Visual Studio to create and deploy an ASP.NET Core web app to Azure App Service on Windows.

Create an ASP.NET Core app in App Service on Linux
Use the command line to create and deploy an ASP.NET Core web app to Azure App Service on Linux.

See the ASP.NET Core on App Service Dashboard for the version of ASP.NET Core available on Azure App service.

Subscribe to the App Service Announcements repository and monitor the issues. The App Service team regularly posts announcements and scenarios arriving in App Service.

The following articles are available in ASP.NET Core documentation:

Publish an ASP.NET Core app to Azure with Visual Studio
Learn how to publish an ASP.NET Core app to Azure App Service using Visual Studio.

Create your first pipeline
Set up a CI build for an ASP.NET Core app, then create a continuous deployment release to Azure App Service.

Azure Web App sandbox
Discover Azure App Service runtime execution limitations enforced by the Azure Apps platform.

Troubleshoot and debug ASP.NET Core projects
Understand and troubleshoot warnings and errors with ASP.NET Core projects.

Application configuration

Platform

The platform architecture (x86/x64) of an App Services app is set in the app's settings in the Azure Portal for apps that are hosted on an A-series compute (Basic) or higher hosting tier. Confirm that the app's publish settings (for example, in the Visual Studio publish profile (.pubxml)) match the setting in the app's service configuration in the Azure Portal.

ASP.NET Core apps can be published framework-dependent because the runtimes for 64-bit (x64) and 32-bit (x86) apps are present on Azure App Service. The .NET Core SDK available on App Service is 32-bit, but you can deploy 64-bit apps built locally using the Kudu console or the publish process in Visual Studio. For more information, see the Publish and deploy the app section.

For more information on .NET Core framework components and distribution methods, such as information on the .NET Core runtime and the .NET Core SDK, see About .NET Core: Composition.

Packages

Include the following NuGet packages to provide automatic logging features for apps deployed to Azure App Service:

- Microsoft.AspNetCore.AzureAppServices.HostingStartup uses IHostingStartup to provide ASP.NET Core logging integration with Azure App Service. The added logging features are provided by the `Microsoft.AspNetCore.AzureAppServicesIntegration` package.
- Microsoft.AspNetCore.AzureAppServicesIntegration executes AddAzureWebAppDiagnostics to add Azure App Service diagnostics logging providers in the `Microsoft.Extensions.Logging.AzureAppServices` package.
- Microsoft.Extensions.Logging.AzureAppServices provides logger implementations to support Azure App Service diagnostics logs and log streaming features.

The preceding packages must be explicitly referenced in the app's project file.

Override app configuration using the Azure Portal

App settings in the Azure Portal permit you to set environment variables for the app. Environment variables can be consumed by the Environment Variables Configuration Provider.

When an app setting is created or modified in the Azure Portal and the **Save** button is selected, the Azure App is restarted. The environment variable is available to the app after the service restarts.

Environment variables are loaded into the app's configuration when CreateBuilder is called to build the host. For more information, see the Environment Variables Configuration Provider.

Proxy server and load balancer scenarios

The IIS Integration Middleware, which configures Forwarded Headers Middleware when hosting out-of-process, and the ASP.NET Core Module are configured to forward the scheme (HTTP/HTTPS) and the remote IP address where the request originated. Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Monitoring and logging

ASP.NET Core apps deployed to App Service automatically receive an App Service extension, **ASP.NET Core Logging Integration**. The extension enables logging integration for ASP.NET Core apps on Azure App Service.

For monitoring, logging, and troubleshooting information, see the following articles:

Monitor apps in Azure App Service
Learn how to review quotas and metrics for apps and App Service plans.

Enable diagnostics logging for apps in Azure App Service
Discover how to enable and access diagnostic logging for HTTP status codes, failed requests, and web server activity.

Handle errors in ASP.NET Core
Understand common approaches to handling errors in ASP.NET Core apps.

Troubleshoot ASP.NET Core on Azure App Service and IIS
Learn how to diagnose issues with Azure App Service deployments with ASP.NET Core apps.

Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
See the common deployment configuration errors for apps hosted by Azure App Service/IIS with troubleshooting advice.

Data Protection key ring and deployment slots

Data Protection keys are persisted to the *%HOME%\ASP.NET\DataProtection-Keys* folder. This folder is backed by network storage and is synchronized across all machines hosting the app. Keys aren't protected at rest. This folder supplies the key ring to all instances of an app in a single deployment slot. Separate deployment slots, such as Staging and Production, don't share a key ring.

When swapping between deployment slots, any system using data protection won't be able to decrypt stored data using the key ring inside the previous slot. ASP.NET Cookie Middleware uses data protection to protect its cookies. This leads to users being signed out of an app that uses the standard ASP.NET Cookie Middleware. For a slot-independent key ring solution, use an external key ring provider, such as:

- Azure Blob Storage
- Azure Key Vault
- SQL store

- Redis cache

For more information, see Key storage providers in ASP.NET Core.

Deploy an ASP.NET Core app that uses a .NET Core preview

To deploy an app that uses a preview release of .NET Core, see the following resources. These approaches are also used when the runtime is available but the SDK hasn't been installed on Azure App Service.

- Specify the .NET Core SDK Version using Azure Pipelines
- Deploy a self-contained preview app
- Use Docker with Web Apps for containers
- Install the preview site extension

See the ASP.NET Core on App Service Dashboard for the version of ASP.NET Core available on Azure App service.

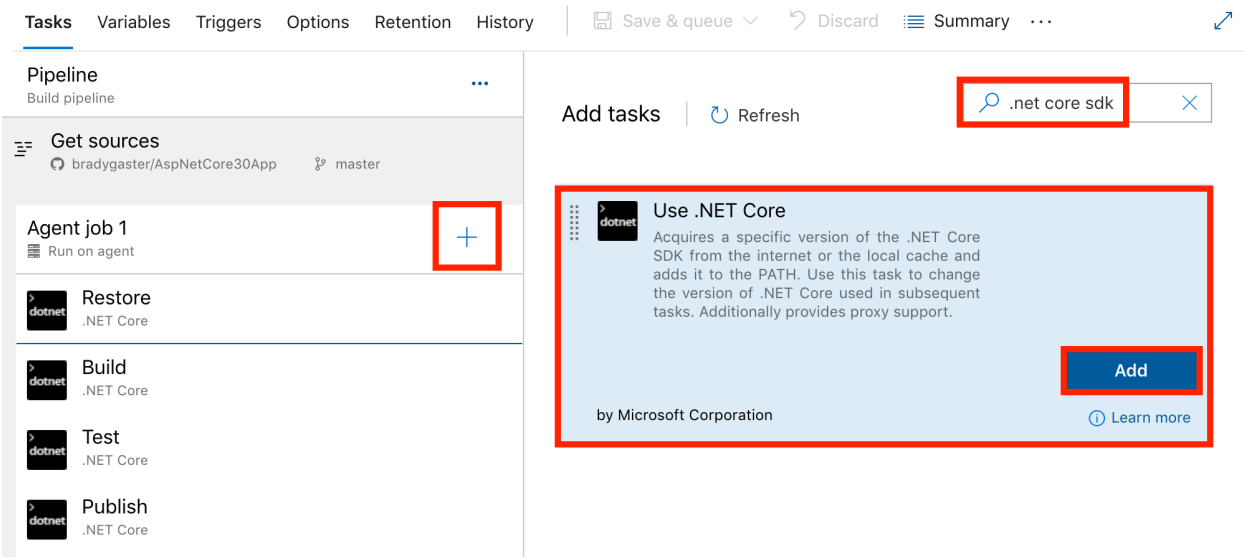
See Select the .NET Core version to use for information on selecting the version of the .NET SDK for self-contained deployments.

Specify the .NET Core SDK Version using Azure Pipelines

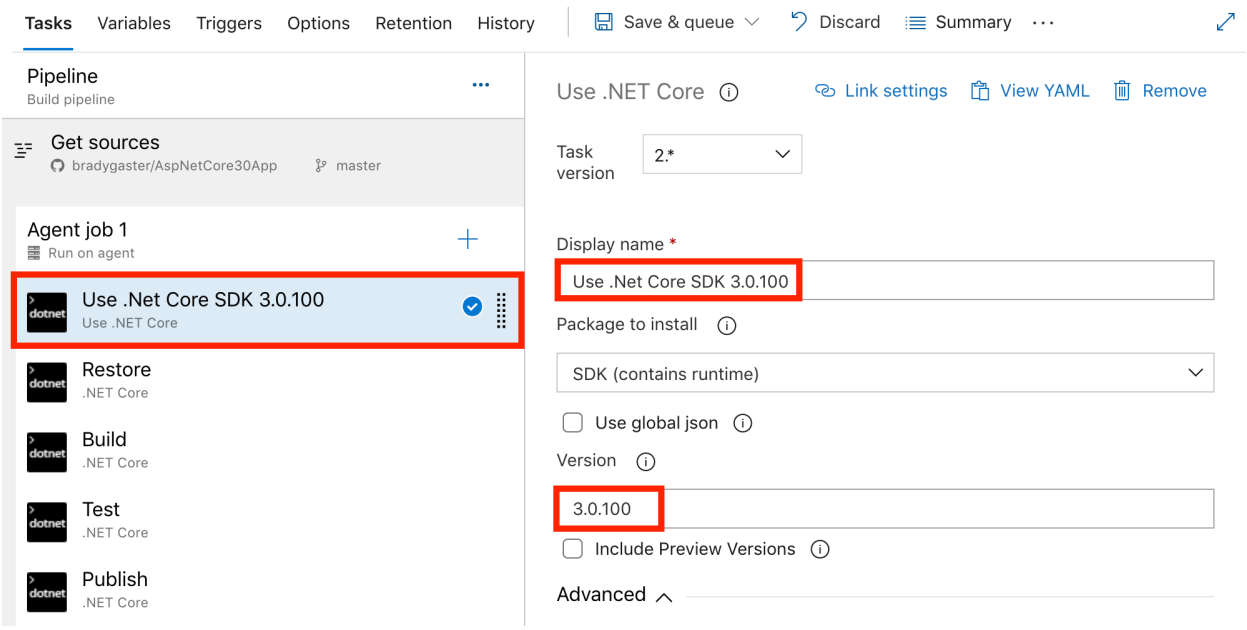
Use Azure App Service CI/CD scenarios to set up a continuous integration build with Azure DevOps. After the Azure DevOps build is created, optionally configure the build to use a specific SDK version.

Specify the .NET Core SDK version

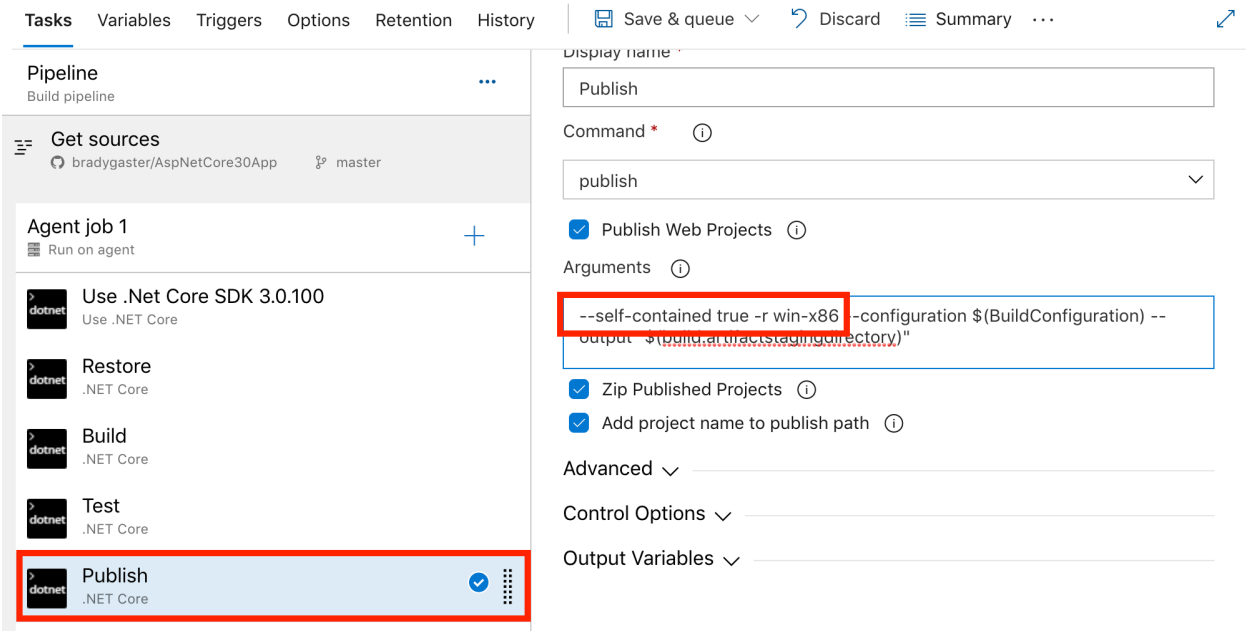
When using the App Service deployment center to create an Azure DevOps build, the default build pipeline includes steps for `Restore`, `Build`, `Test`, and `Publish`. To specify the SDK version, select the **Add (+)** button in the Agent job list to add a new step. Search for **.NET Core SDK** in the search bar.



Move the step into the first position in the build so that the steps following it use the specified version of the .NET Core SDK. Specify the version of the .NET Core SDK. In this example, the SDK is set to `3.0.100`.



To publish a self-contained deployment (SCD), configure SCD in the `Publish` step and provide the Runtime Identifier (RID).



Deploy a self-contained preview app

A self-contained deployment (SCD) that targets a preview runtime carries the preview runtime in the deployment.

When deploying a self-contained app:

- The site in Azure App Service doesn't require the preview site extension.
- The app must be published following a different approach than when publishing for a framework-dependent deployment (FDD).

Follow the guidance in the Deploy the app self-contained section.

Use Docker with Web Apps for containers

The Docker Hub at `https://hub.docker.com/_/microsoft-dotnet` contains the latest preview Docker images. The images can be used as a base image. Use the image and deploy to Web Apps for Containers normally.

Install the preview site extension

If a problem occurs using the preview site extension, open an dotnet/AspNetCore issue.

1. From the Azure Portal, navigate to the App Service.
2. Select the web app.
3. Type "ex" in the search box to filter for "Extensions" or scroll down the list of management tools.

4. Select **Extensions**.
5. Select **Add**.
6. Select the **ASP.NET Core {X.Y} ({x64|x86}) Runtime** extension from the list, where `{X.Y}` is the ASP.NET Core preview version and `{x64|x86}` specifies the platform.
7. Select **OK** to accept the legal terms.
8. Select **OK** to install the extension.

When the operation completes, the latest .NET Core preview is installed. Verify the installation:

1. Select **Advanced Tools**.
2. Select **Go** in **Advanced Tools**.
3. Select the **Debug console** > **PowerShell** menu item.
4. At the PowerShell prompt, execute the following command. Substitute the ASP.NET Core runtime version for `{X.Y}` and the platform for `{PLATFORM}` in the command:

   ```
   PowerShell

   Test-Path D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.{PLATFORM}\
   ```

   The command returns `True` when the x64 preview runtime is installed.

> **Note**
>
> The platform architecture (x86/x64) of an App Services app is set in the app's settings in the Azure Portal for apps that are hosted on an A-series compute (Basic) or higher hosting tier. Confirm that the app's publish settings (for example, in the Visual Studio publish profile (.pubxml)) match the setting in the app's service configuration in the Azure portal.
>
> If the app is run in in-process mode and the platform architecture is configured for 64-bit (x64), the ASP.NET Core Module uses the 64-bit preview runtime, if present. Install the **ASP.NET Core {X.Y} (x64) Runtime** extension using the Azure Portal.
>
> After installing the x64 preview runtime, run the following command in the Azure Kudu PowerShell command window to verify the installation. Substitute the ASP.NET Core runtime version for `{X.Y}` in the following command:
>
> ```
> PowerShell
> ```

```
Test-Path D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64\
```

The command returns `True` when the x64 preview runtime is installed.

Use the preview site extension with an ARM template

If an ARM template is used to create and deploy apps, the `Microsoft.Web/sites/siteextensions` resource type can be used to add the site extension to a web app. In the following example, the ASP.NET Core 5.0 (x64) Runtime site extension (`AspNetCoreRuntime.5.0.x64`) is added to the app:

JSON

```
{
    ...
    "parameters": {
        "site_name": {
            "defaultValue": "{SITE NAME}",
            "type": "String"
        },
        ...
    },
    ...
    "resources": [
        ...
        {
            "type": "Microsoft.Web/sites/siteextensions",
            "apiVersion": "2018-11-01",
            "name": "[concat(parameters('site_name'),
'/AspNetCoreRuntime.5.0.x64')]",
            "location": "[resourceGroup().location]",
            "dependsOn": [
                "[resourceId('Microsoft.Web/sites',
parameters('site_name'))]"
            ]
        }
    ]
}
```

For the placeholder `{SITE NAME}`, use the app's name in Azure App Service (for example, `contoso`).

Publish and deploy the app

For a 64-bit deployment:

- Use a 64-bit .NET Core SDK to build a 64-bit app.
- Set the **Platform** to **64 Bit** in the App Service's **Configuration** > **General settings**. The app must use a Basic or higher service plan to enable the choice of platform bitness.

Deploy the app framework-dependent

Apps published as framework-dependent are cross-platform and don't include the .NET runtime in the deployment. Azure App Service includes the .NET runtime.

Visual Studio

1. Right-click the project in **Solution Explorer** and select **Publish**. Alternatively, select **Build** > **Publish {Application Name}** from the Visual Studio toolbar.
2. In the **Publish** dialog, select **Azure** > **Next**.
3. Select the Azure service.
4. Select **Advanced**. The **Publish** dialog opens.
5. Select a Resource group and Hosting plan, or create new ones.
6. Select **Finish**.
7. In the **Publish** page:

    - For **Configuration**, select the pen icon **Edit Configuration**:
        - Confirm that the **Release** configuration is selected.
        - In the **Deployment Mode** drop-down list, select **Framework-Dependent**.
        - In the **Target Runtime** drop-down list, select the desired runtime. The default is `win-x86`.
    - To remove additional files upon deployment, open **File Publish Options** and select the checkbox to remove additional files at the destination.
    - Select **Save**.
    - Select **Publish**.

Deploy the app self-contained

Publishing an app as self-contained produces a platform-specific executable. The output publishing folder contains all components of the app, including the .NET libraries and target runtime. For more information, see [Publish self-contained]/dotnet/core/deploying/#publish-self-contained). Use Visual Studio or the .NET CLI for a self-contained deployment (SCD).

Visual Studio

1. Right-click the project in **Solution Explorer** and select **Publish**. Alternatively, select **Build** > **Publish {Application Name}** from the Visual Studio toolbar.
2. In the **Publish** dialog, select **Azure** > **Next**.
3. Select the Azure service.
4. Select **Advanced**. The **Publish** dialog opens.
5. Select a Resource group and Hosting plan, or create new ones.
6. Select **Finish**.
7. In the **Publish** page:

    - For **Configuration**, select the pen icon **Edit Configuration**:
        - Confirm that the **Release** configuration is selected.
        - In the **Deployment Mode** drop-down list, select **Self-Contained**.
        - In the **Target Runtime** drop-down list, select the desired runtime. The default is `win-x86`.
    - To remove additional files upon deployment, open **File Publish Options** and select the checkbox to remove additional files at the destination.
    - Select **Save**.
    - Select **Publish**.

Protocol settings (HTTPS)

Secure protocol bindings allow specifying a certificate to use when responding to requests over HTTPS. Binding requires a valid private certificate (*.pfx*) issued for the specific hostname. For more information, see Tutorial: Bind an existing custom SSL certificate to Azure App Service.

Transform web.config

If you need to transform *web.config* on publish (for example, set environment variables based on the configuration, profile, or environment), see Transform web.config.

Additional resources

- App Service overview
- Azure App Service diagnostics overview
- Host ASP.NET Core in a web farm

- Tutorial: Connect to SQL Database from .NET App Service without secrets using a managed identity

Azure App Service on Windows Server uses Internet Information Services (IIS). Kestrel and YARP on the front end provides the load balancer. The following topics pertain to the underlying IIS technology:

- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module (ANCM) for IIS
- IIS modules with ASP.NET Core
- Windows Server - IT administrator content for current and previous releases

Publish an ASP.NET Core app to Azure with Visual Studio

Article • 11/01/2023

> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see **Deploy ASP.NET Core preview release to Azure App Service**.

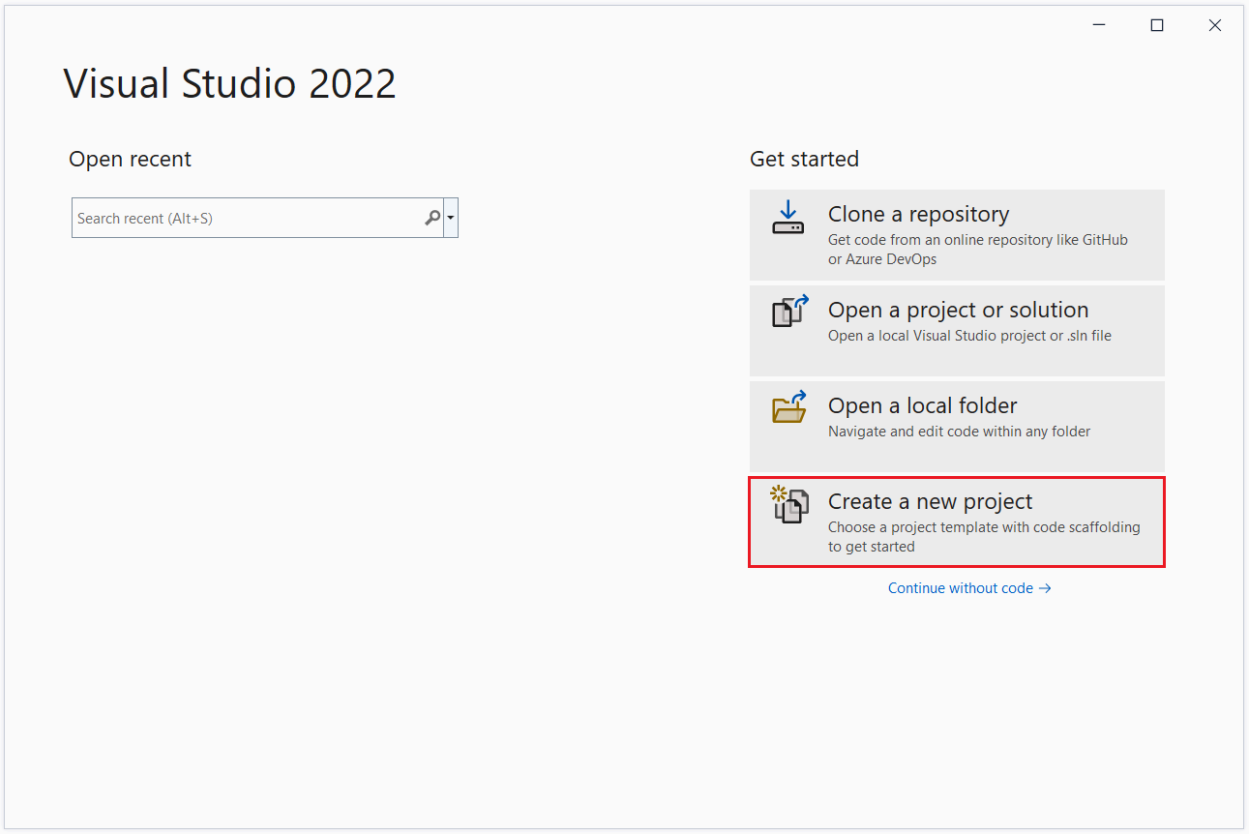
To troubleshoot an App Service deployment issue, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Set up

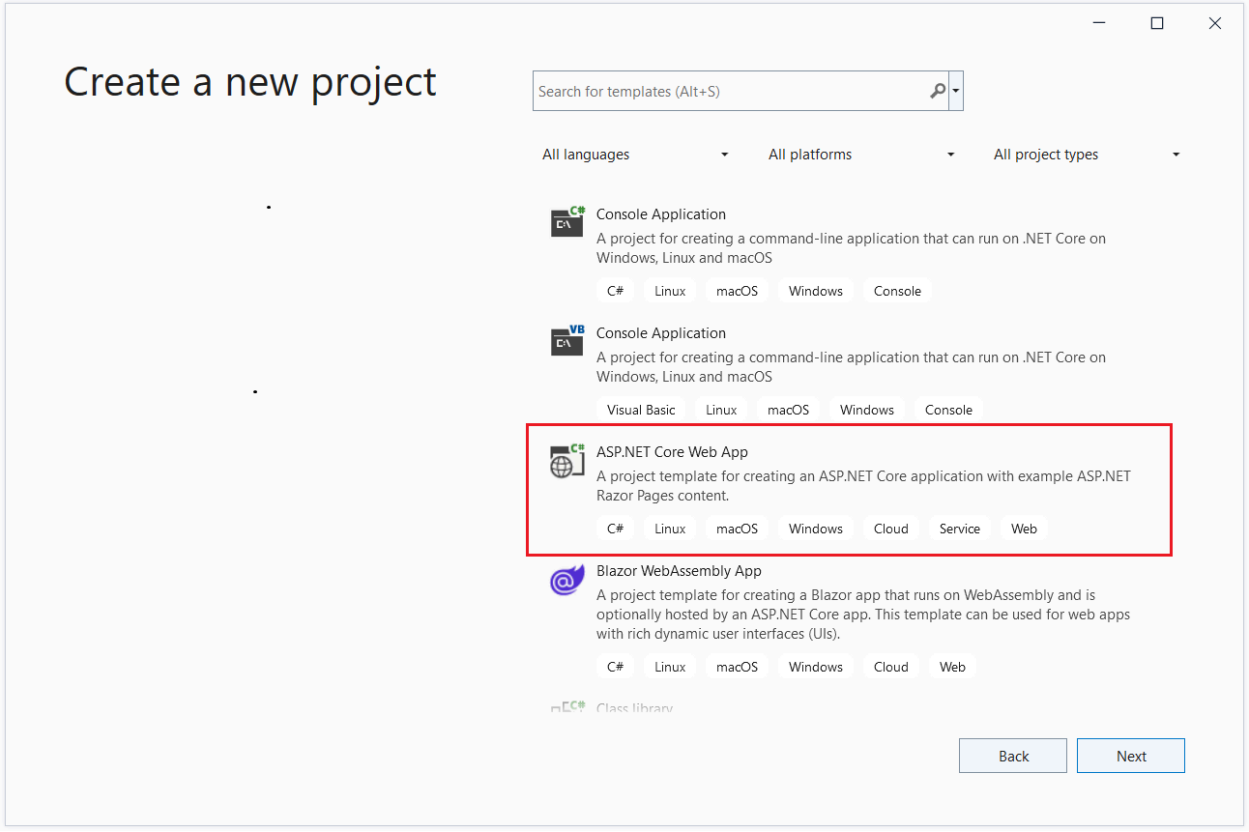
Open a free Azure account if you don't have one.

Create a web app

Start Visual Studio 2022 and select **Create a new project**.



In the **Create a new project** dialog, select **ASP.NET Core Web App**, and then select **Next**.

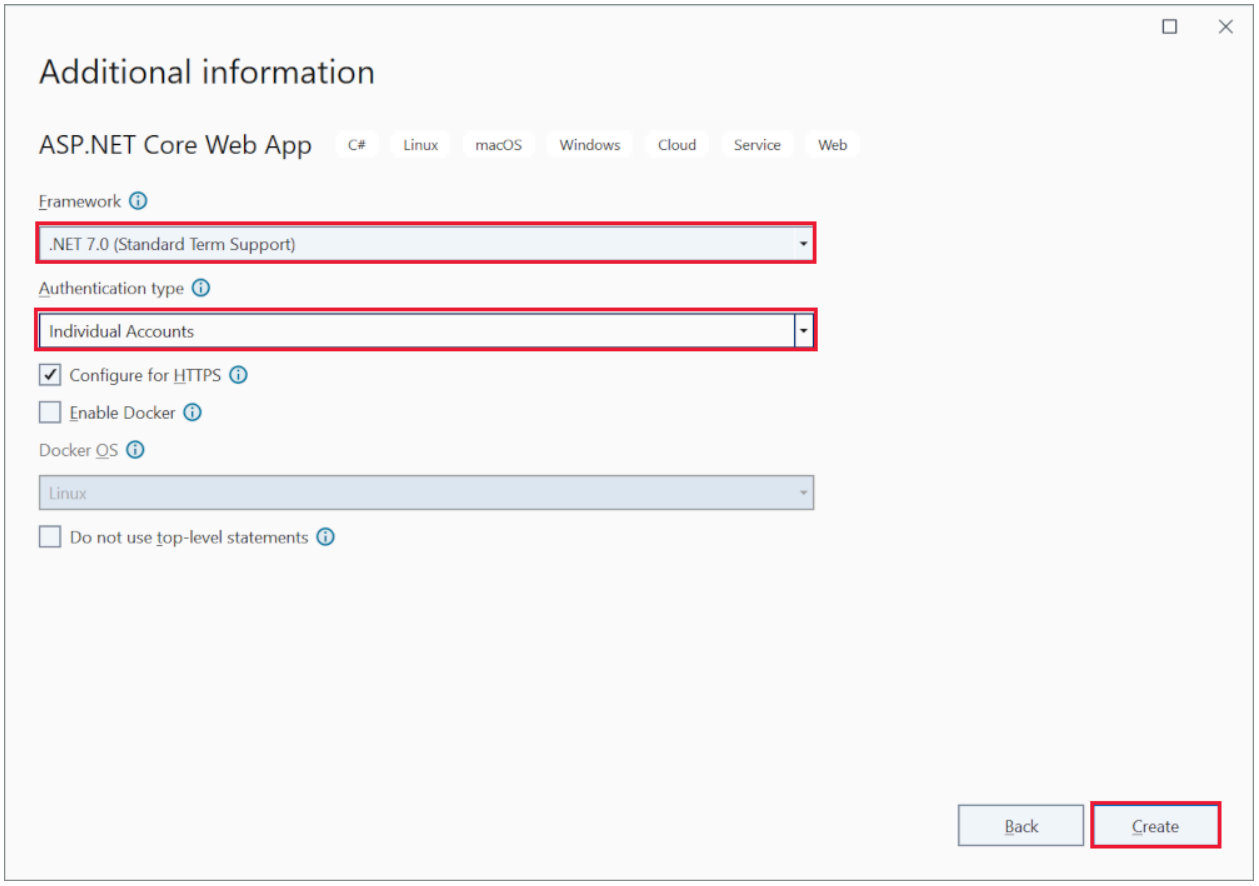


In the **Configure your new project** dialog, name your project, and then select **Next**.

In the **Additional information** dialog:

- In the **Framework** input, select **.NET 7.0 (Standard Term Support)**.

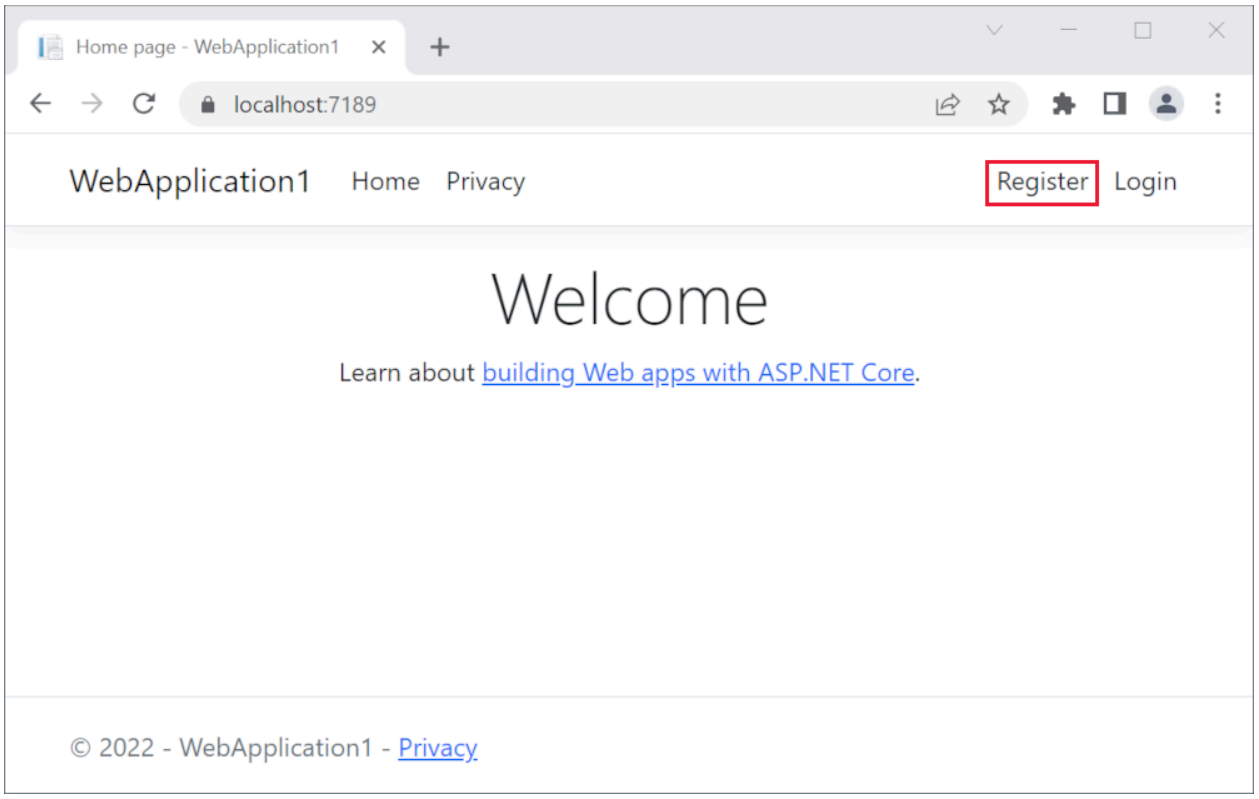
- In the **Authentication type** input, select **Individual Accounts**, and then select **Create**.



Visual Studio creates the solution.

Run the app

- Press F5 to run the project.

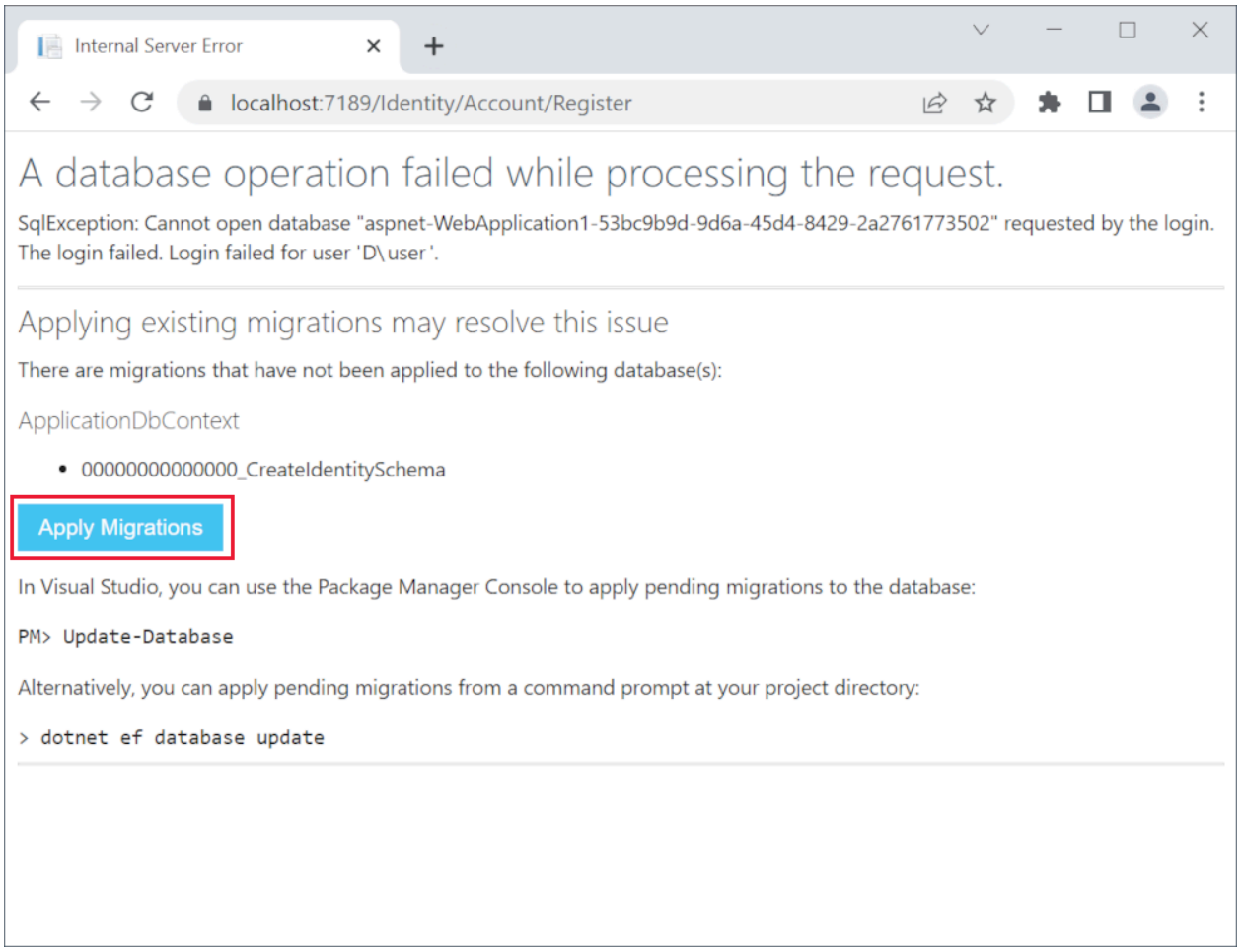


Register a user

- Select **Register** and register a new user. You can use a fictitious email address. When you submit, the page displays the following error:

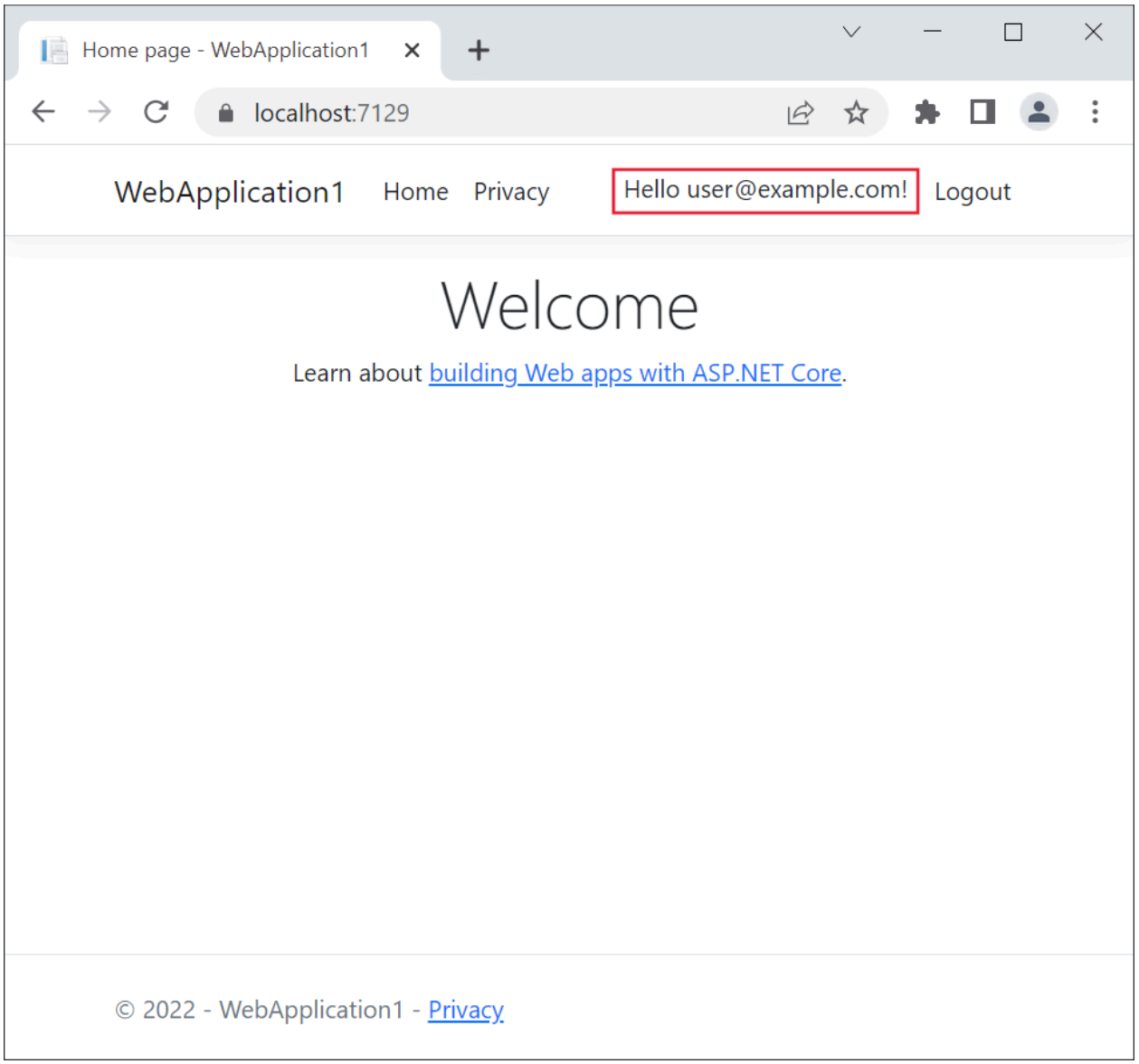
  "A database operation failed while processing the request. Applying existing migrations may resolve this issue"

- Select **Apply Migrations** and, once the page updates, refresh the page.



- A **Register confirmation** page is displayed. Select **Click here to confirm your account**.
- A **Confirm email** page is displayed.
- Log in as the new user.

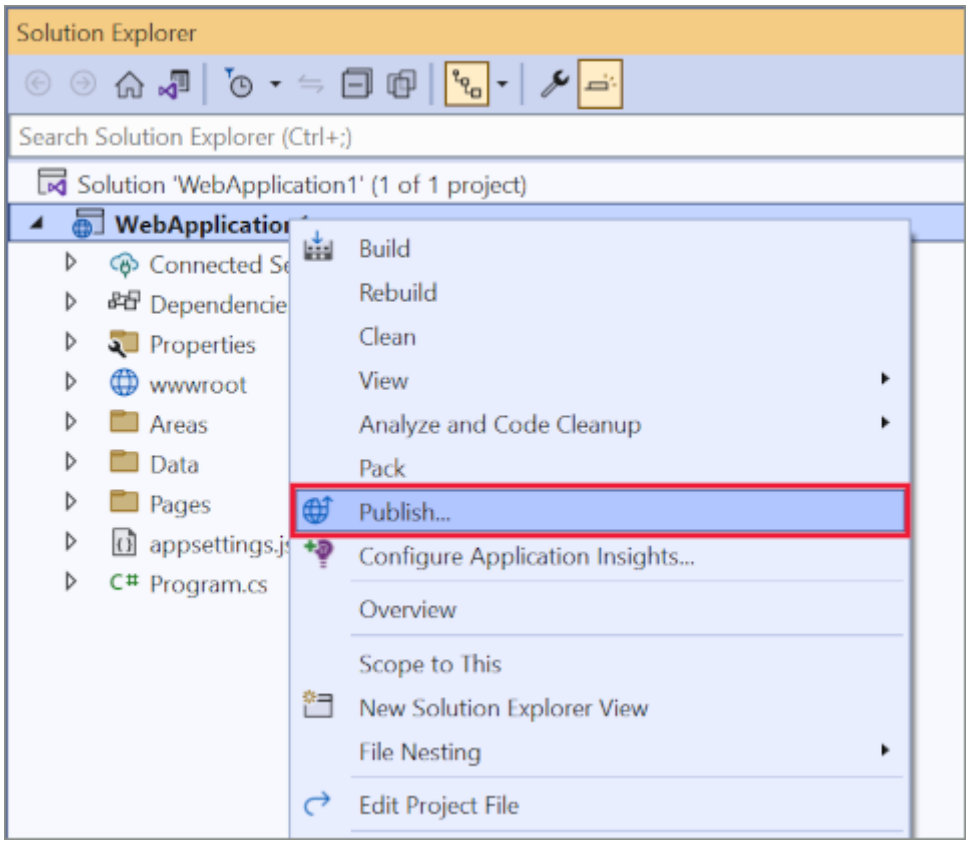
The app displays the email used to register the new user and a **Logout** link.



- Stop the application by closing the browser, or in Visual Studio select **Debug** > **Stop Debugging**.
- In Visual Studio select **Build** > **Clean Solution** to clean project items and avoid file contention.

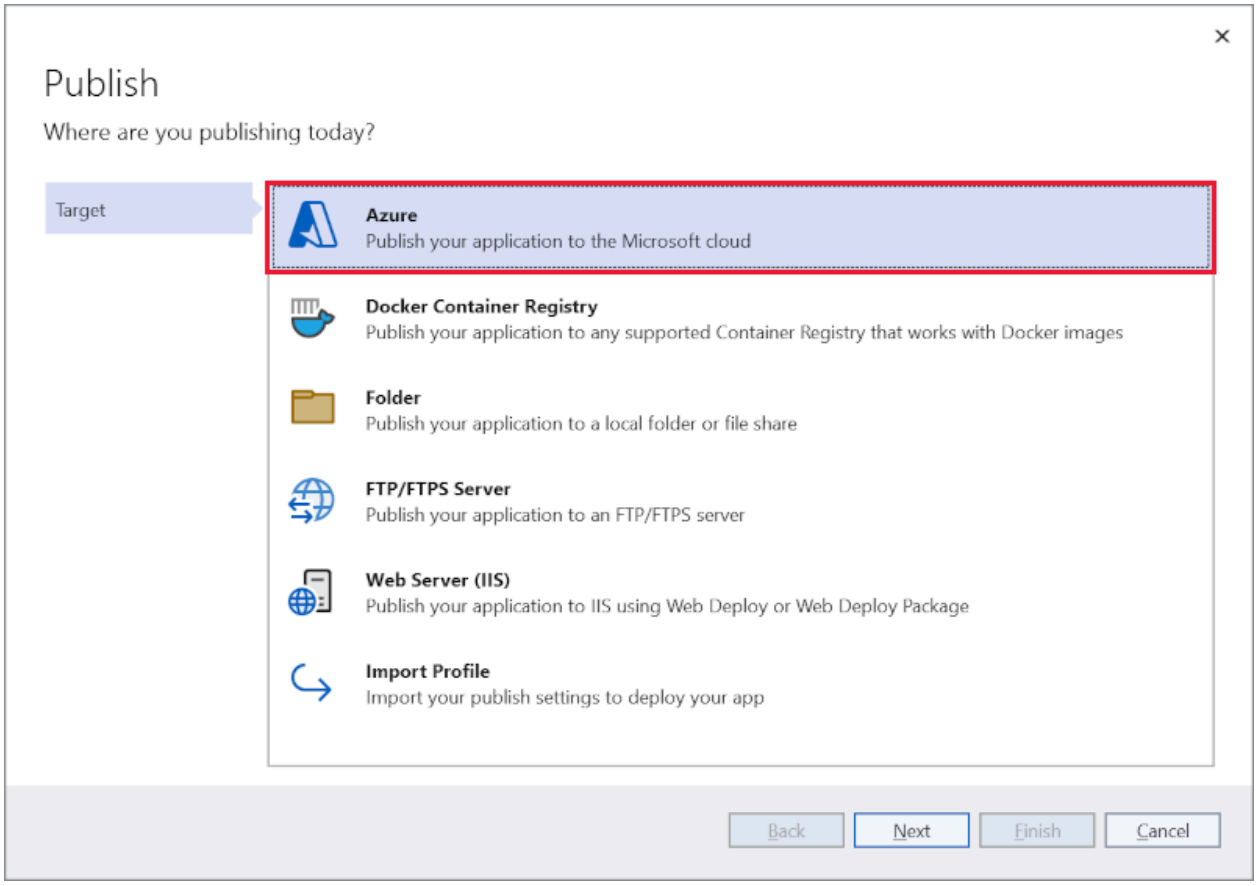
Deploy the app to Azure

Right-click on the project in Solution Explorer and select **Publish**.



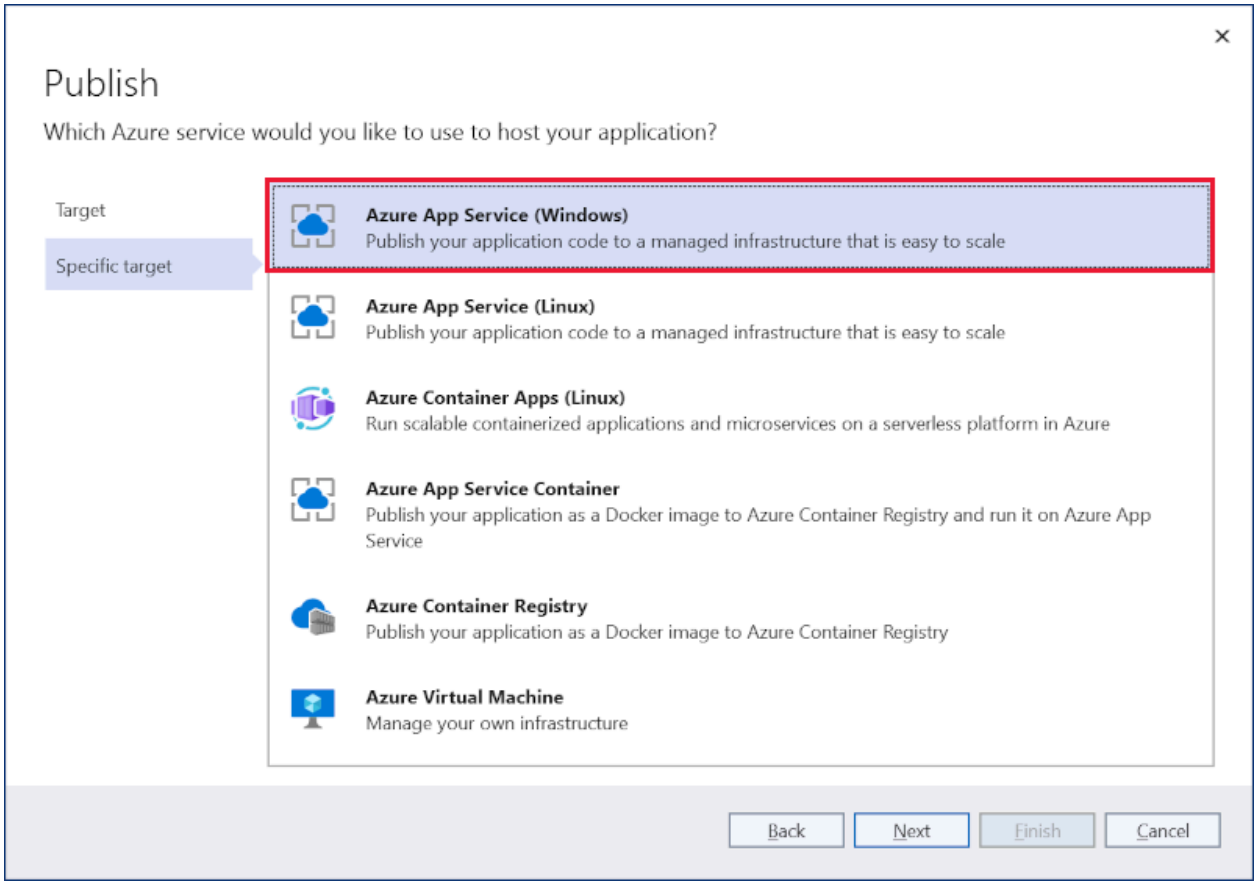
In the **Publish** dialog:

- Select **Azure**.
- Select **Next**.

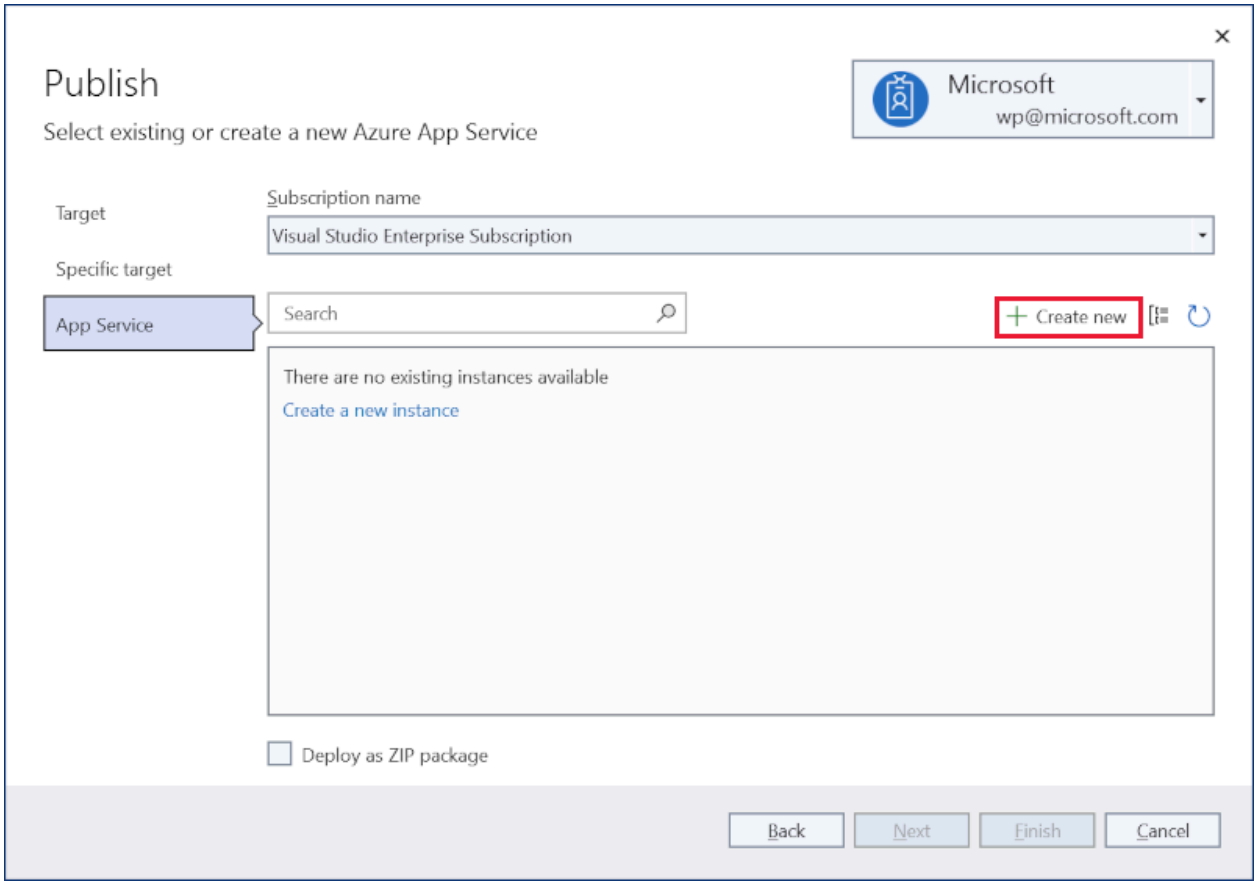


In the **Publish** dialog:

- Select **Azure App Service (Windows)**.
- Select **Next**.

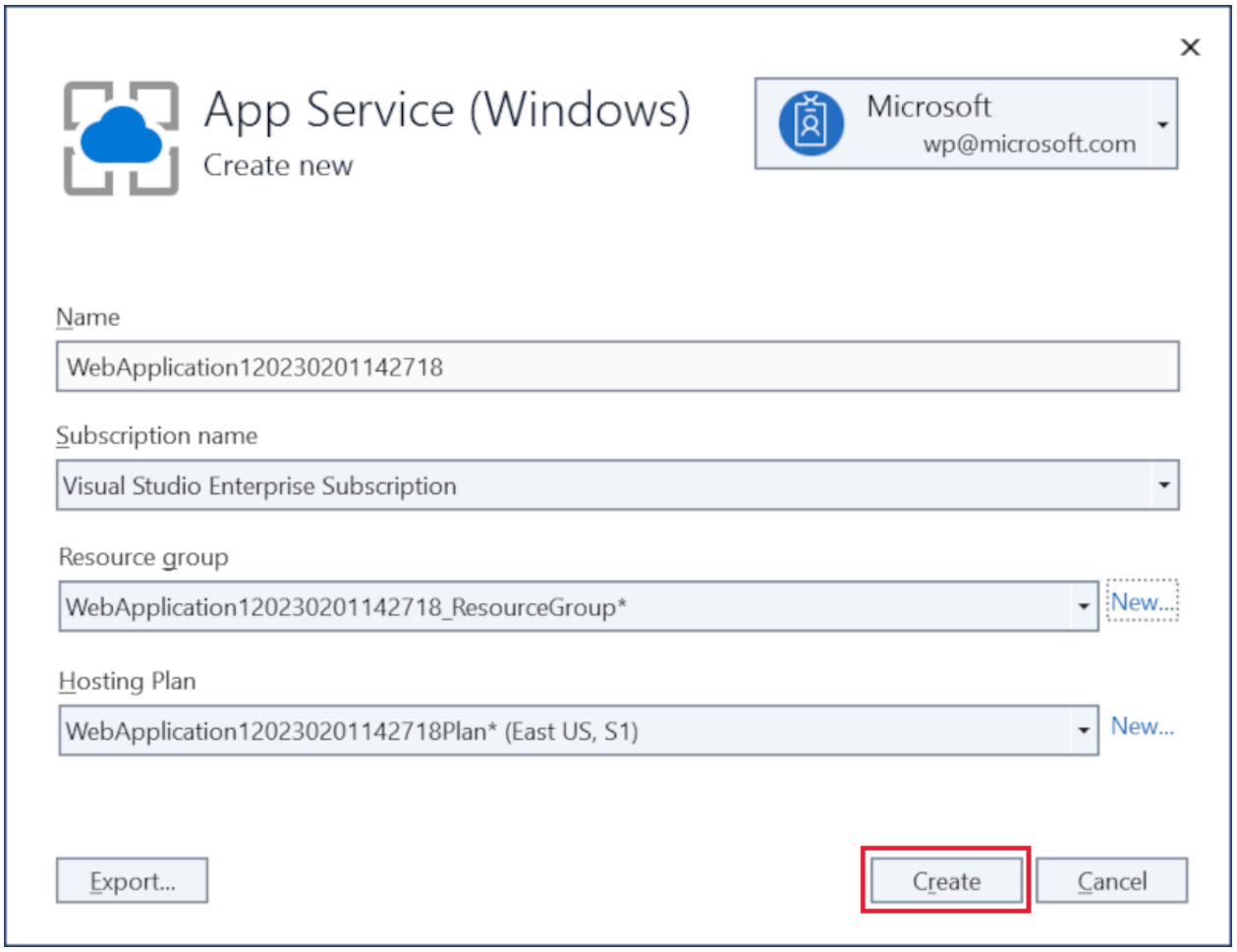


In the **Publish** dialog, in the **App Service** tab, select **Create new**.



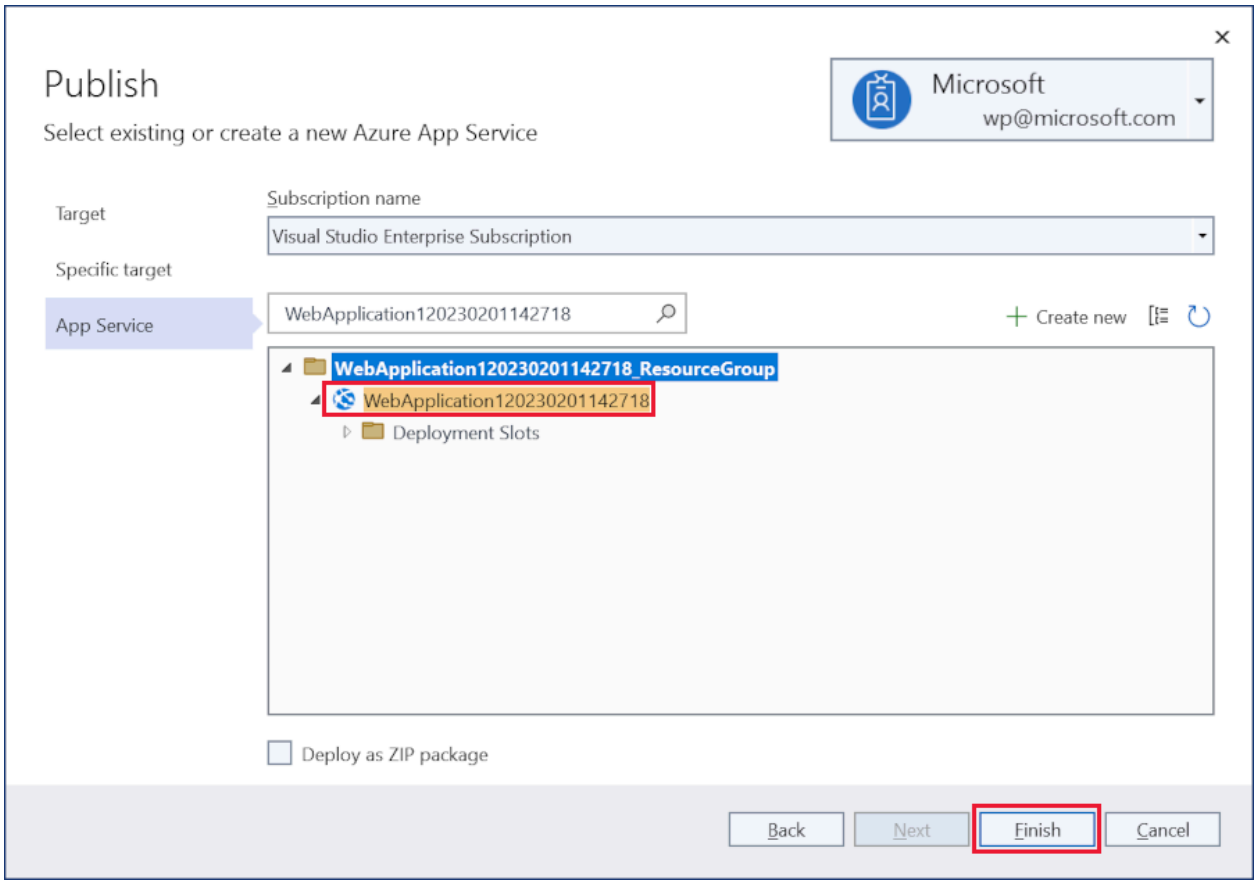
The **Create App Service** dialog appears:

- The **Name**, **Resource Group**, and **Hosting Plan** entry fields are populated. You can keep these names or change them.
- Select **Create**.



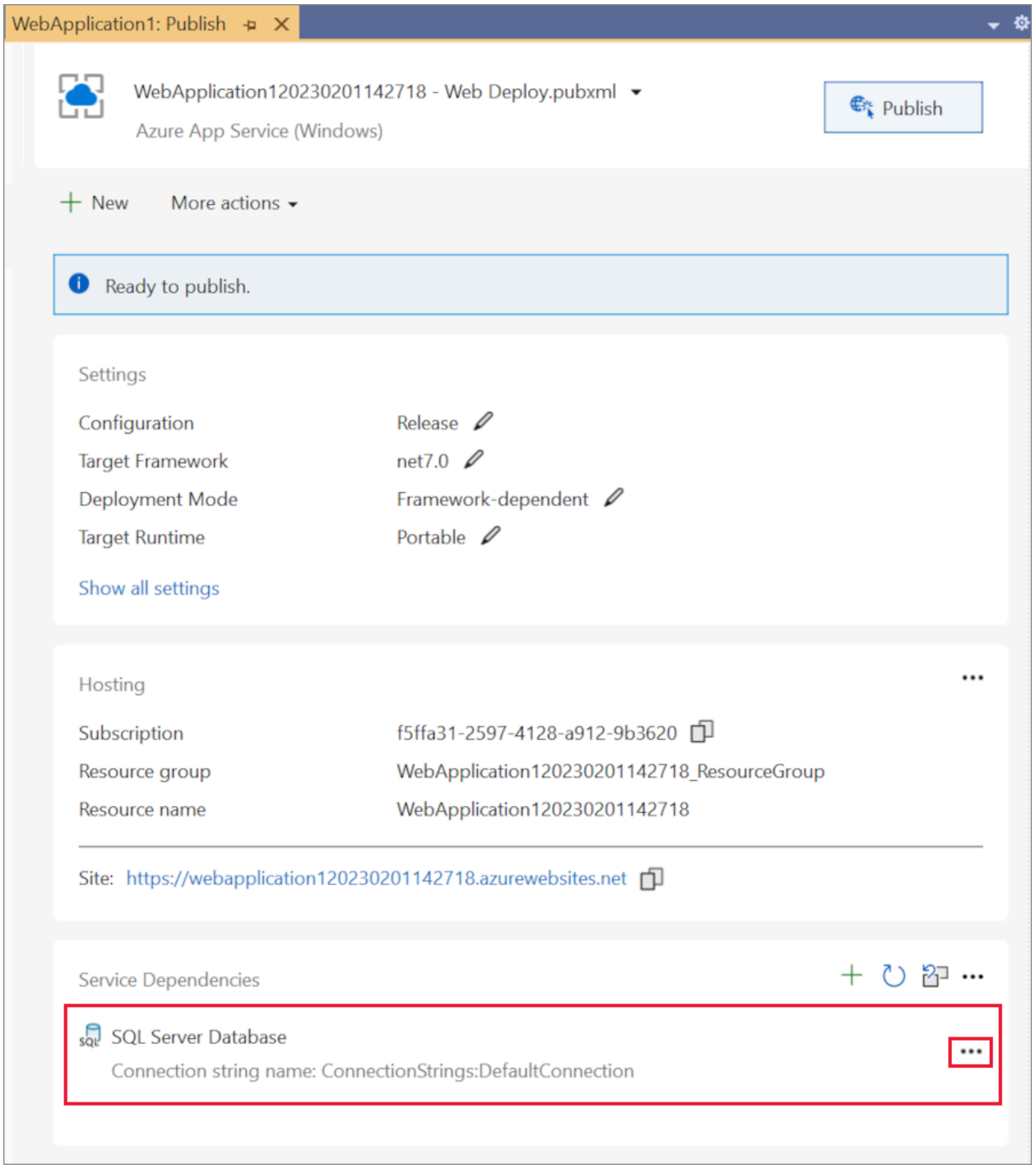
After creation is completed the dialog is automatically closed and the **Publish** dialog gets focus again:

- The new instance that was just created is automatically selected.
- Select **Finish**.



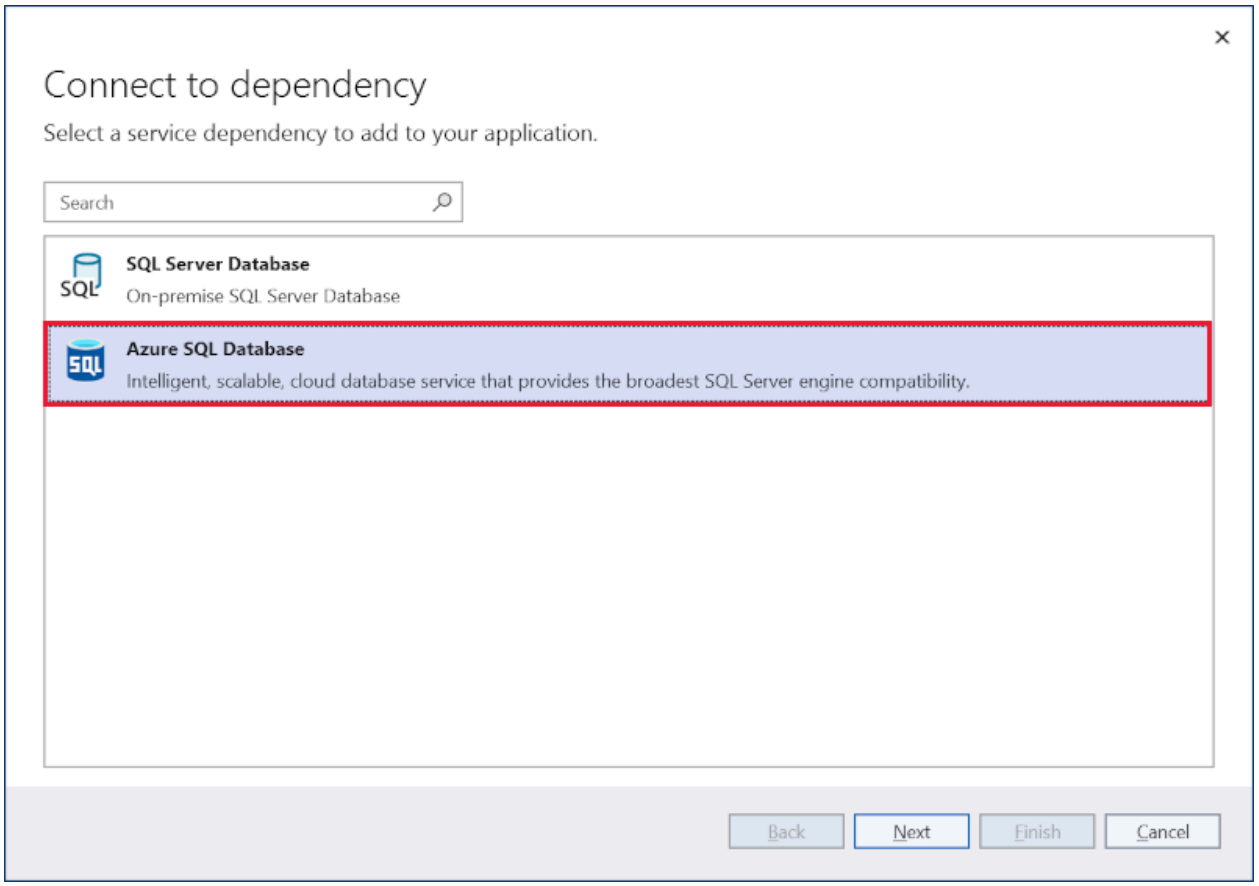
The **Publish profile creation progress** dialog confirms the publish profile was created. Select **Close**.

Next you see the **Publish Profile summary** page. Visual Studio has detected that this application requires a SQL Server database which it has listed in the Service Dependencies pane. Select the ellipsis (...) and then **Connect**.

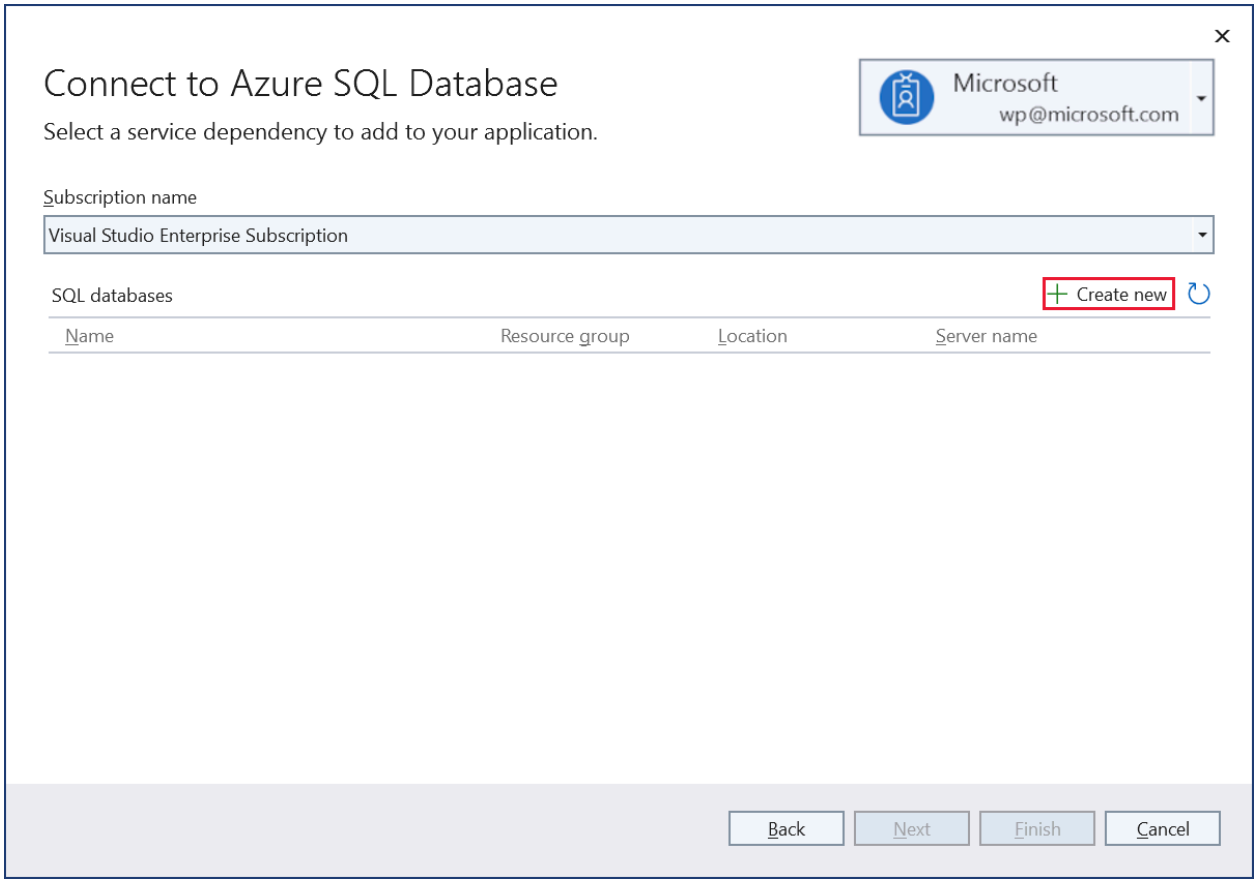


The **Connect to dependency** dialog appears:

- Select **Azure SQL Database**.
- Select **Next**.

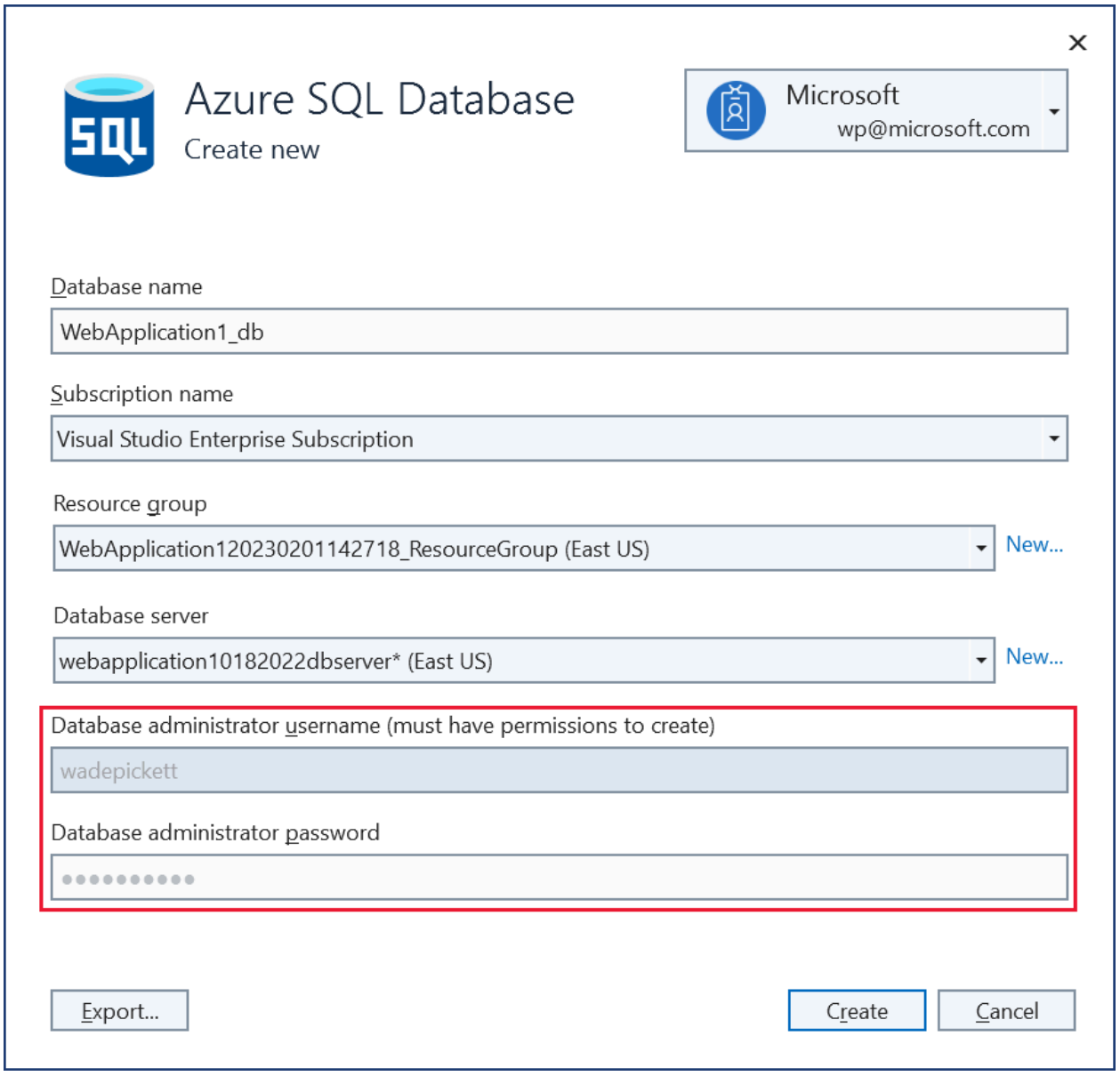


In the **Connect to Azure SQL database** dialog, select **Create new**.



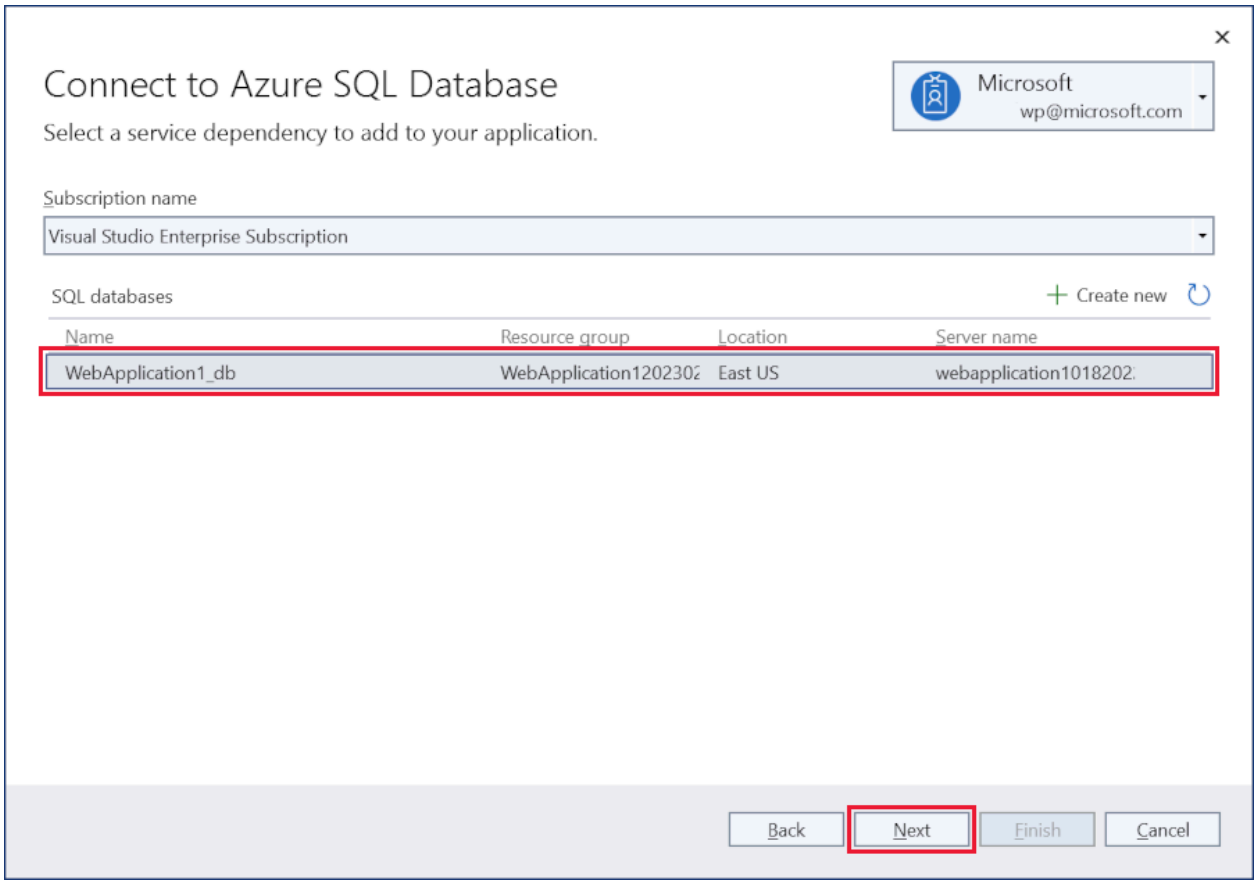
The **Create Azure SQL Database** appears:

- The **Database name**, **Resource Group**, **Database server** and **App Service Plan** entry fields are populated. You can keep these values or change them.
- Enter the **Database administrator username** and **Database administrator password** for the selected **Database server** (note the account you use must have the necessary permissions to create the new Azure SQL database)
- Select **Create**.



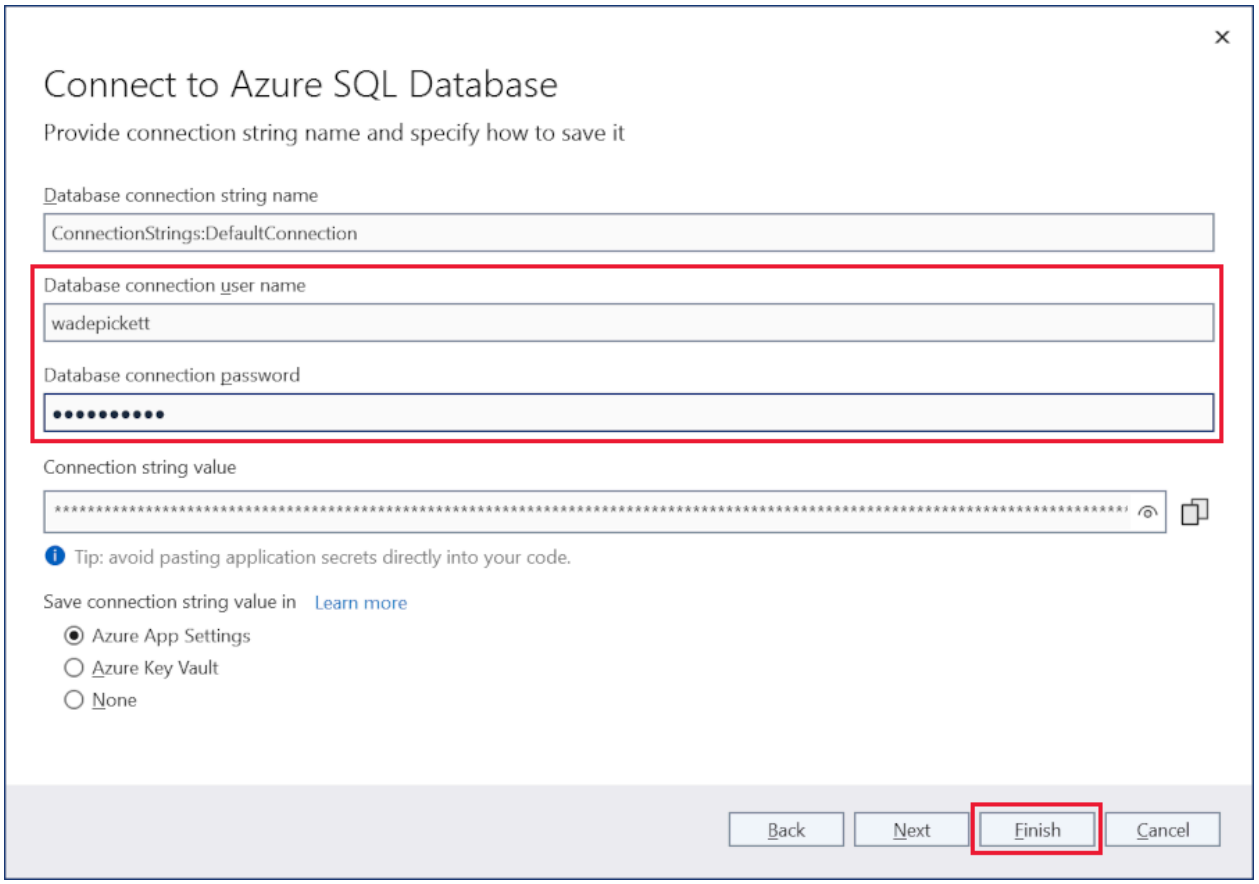
After creation is completed the dialog is automatically closed and the **Connect to Azure SQL Database** dialog gets focus again:

- The new instance that was just created is automatically selected.
- Select **Next**.



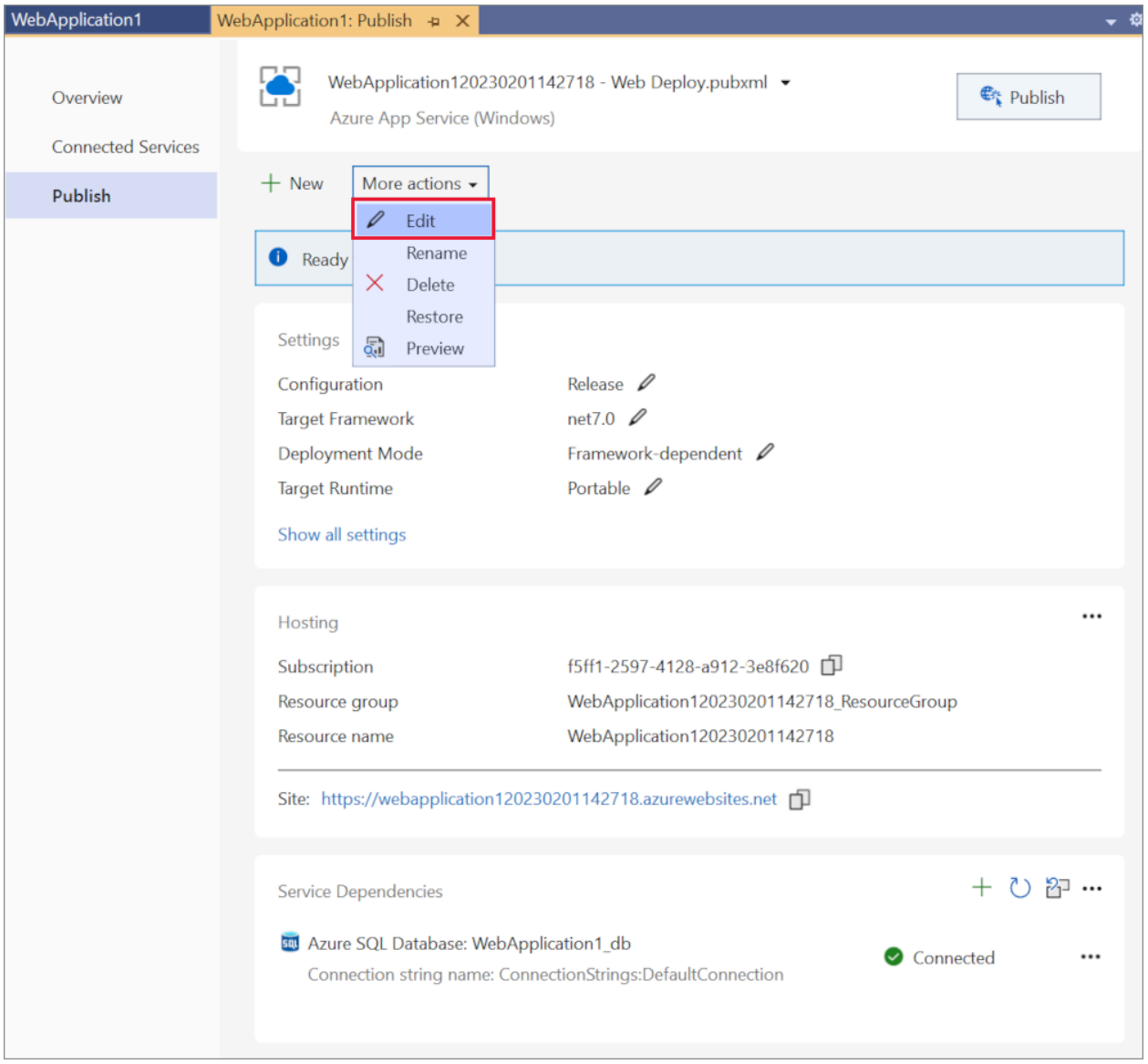
In the next step of the **Connect to Azure SQL Database** dialog:

- Enter the **Database connection user name** and **Database connection password** fields. These are the details your application will use to connect to the database at runtime. Best practice is to avoid using the same details as the admin username and password used in the previous step.
- Select **Finish**.



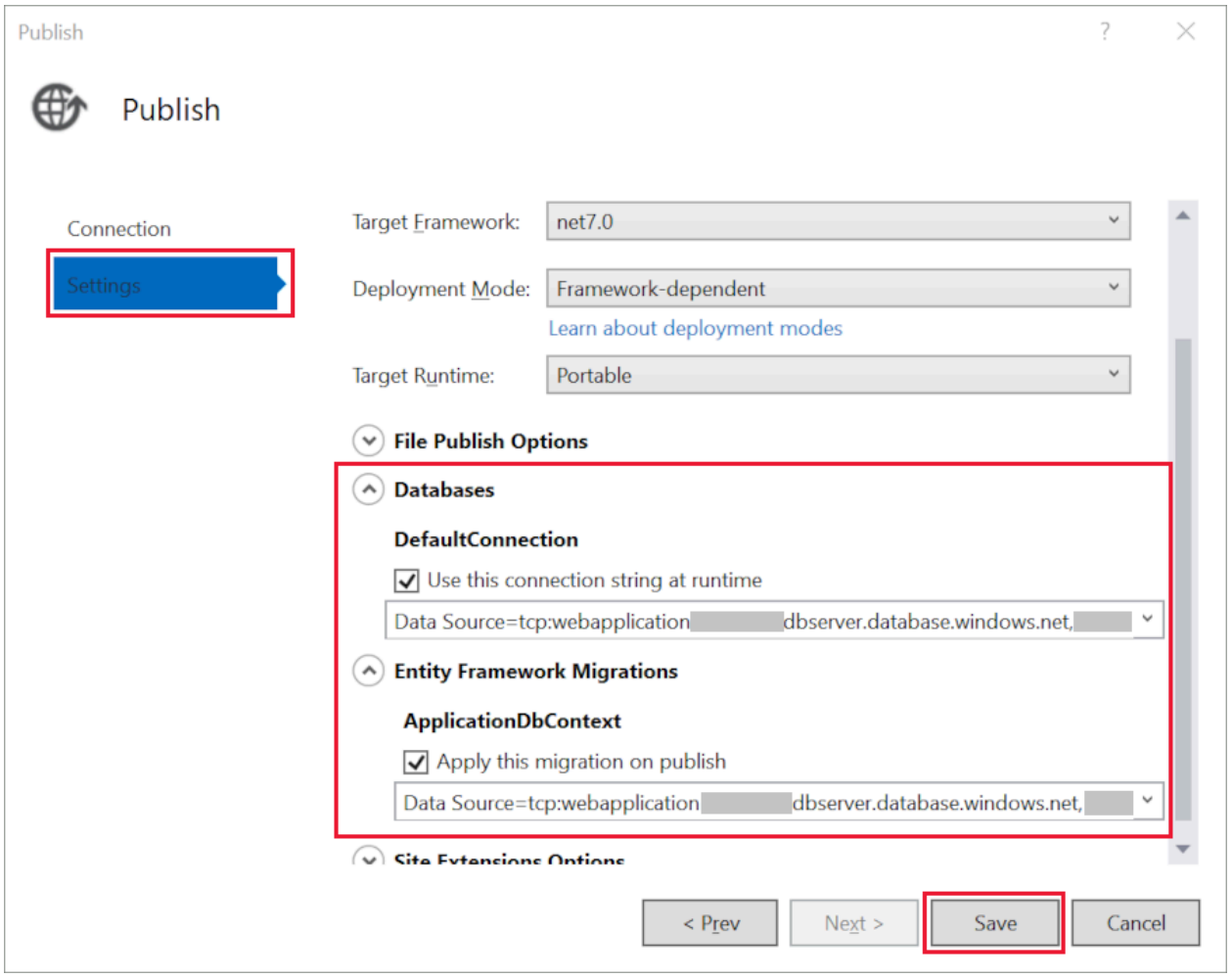
The **Dependency configuration progress** dialog confirms the Azure SQL Database is configured. Select **Close**.

In the **Publish Profile summary** page select **More actions** > **Edit**:

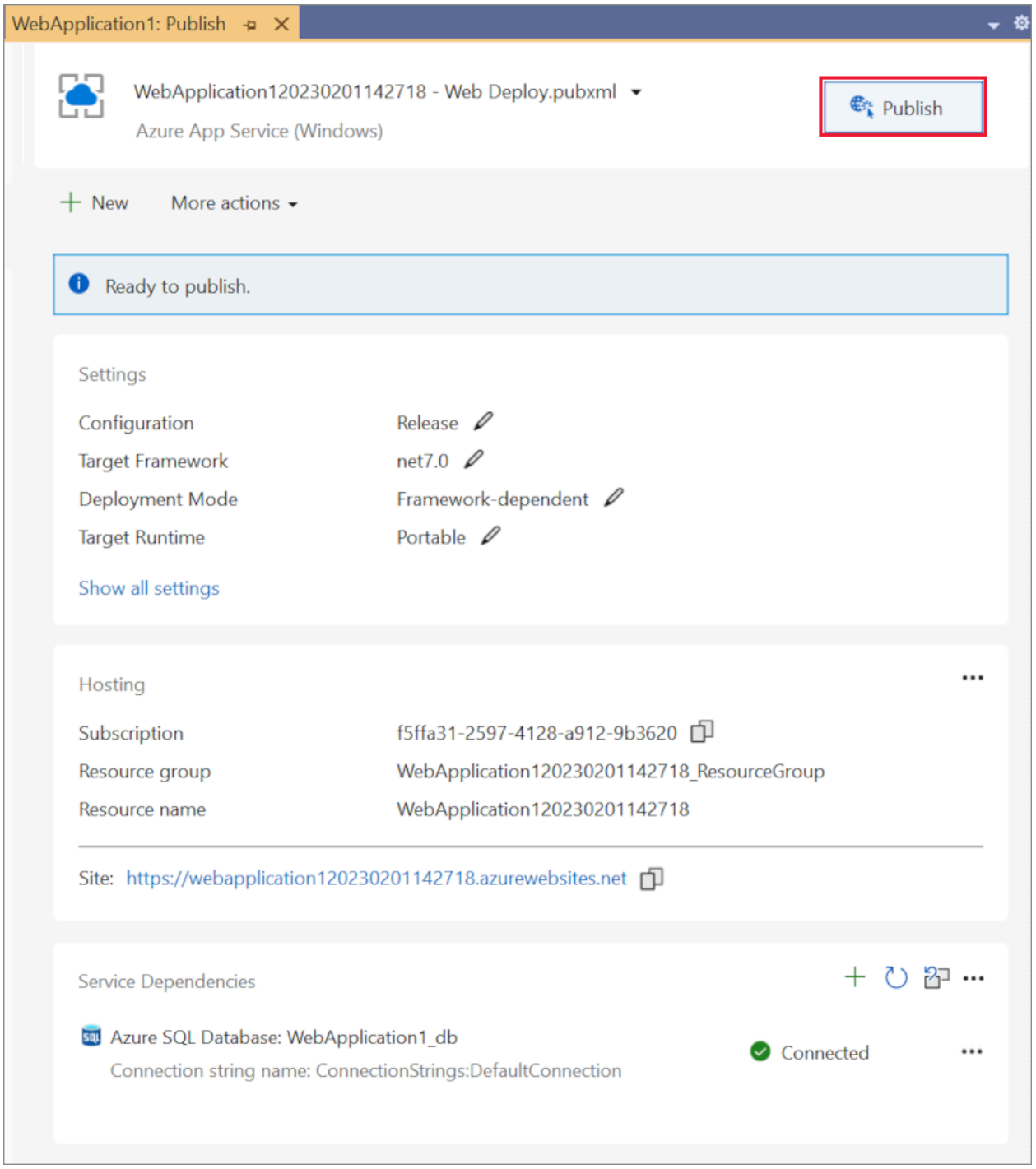


On the **Settings** tab of the **Publish** dialog:

- Expand **Databases** and check **Use this connection string at runtime**.
- Expand **Entity Framework Migrations** and select **Apply this migration on publish**.
- Select **Save**. Visual Studio returns to the **Publish** dialog.



Click **Publish**. Visual Studio publishes your app to Azure. When the deployment completes.



The app is opened in a browser. Register a new user and log in as the new user to validate the database deployment and run-time connection.

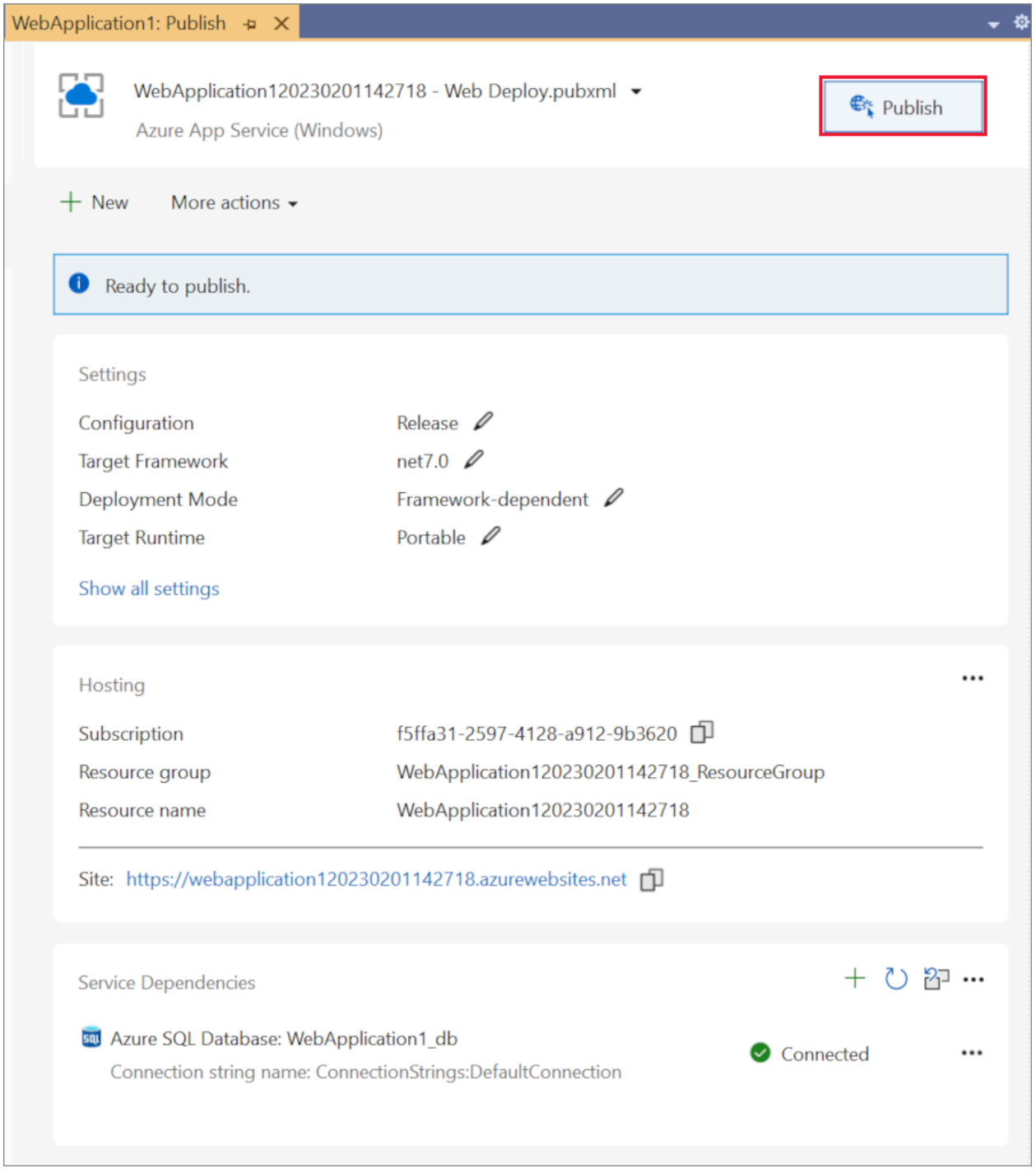
Update the app

- Edit the `Pages/Index.cshtml` Razor page and change its contents, then save the changes. For example, you can modify the paragraph to say "Hello ASP.NET Core!":

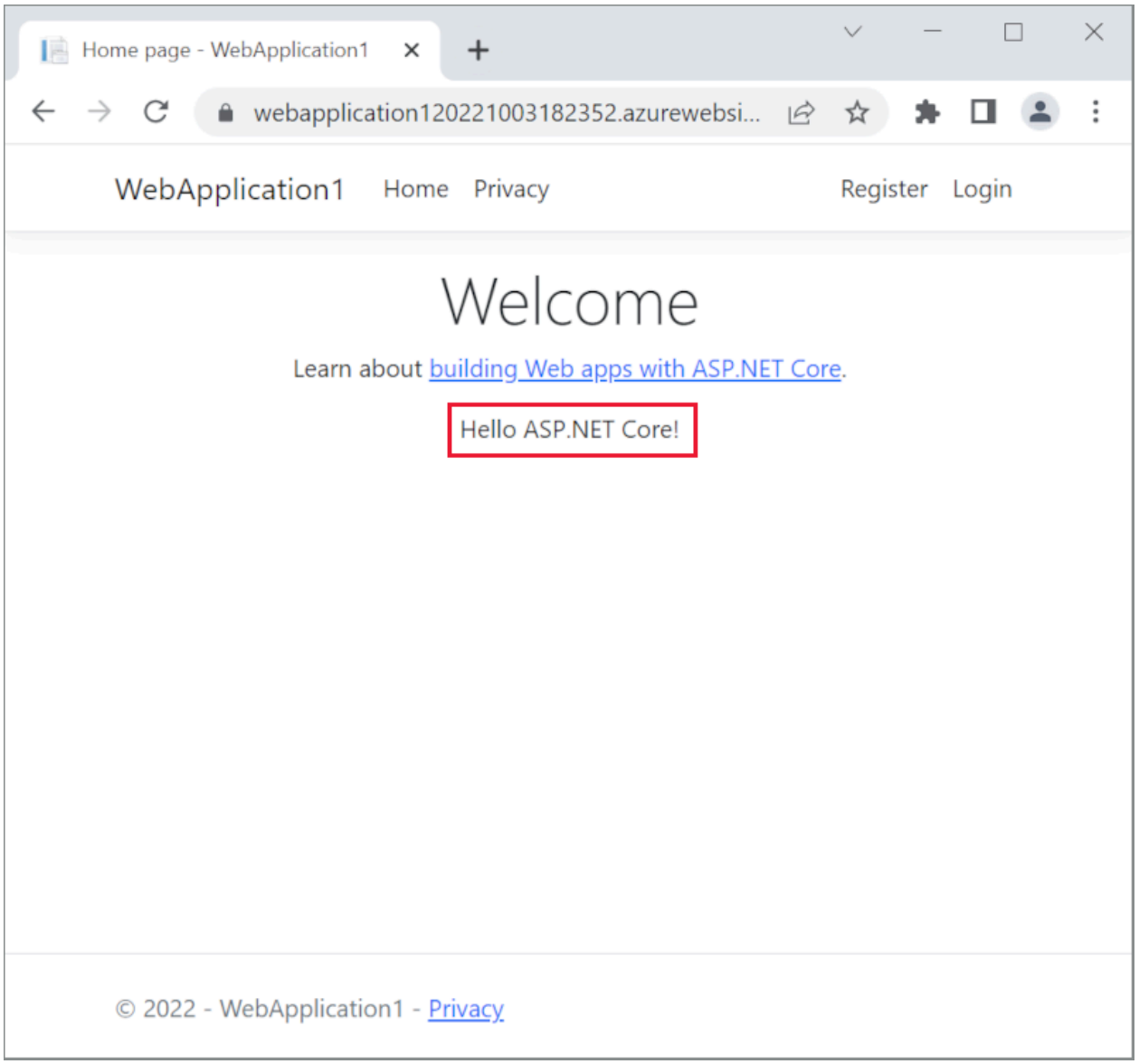
```
HTML

@page
@model IndexModel
@{
```

```
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a
href="https://docs.microsoft.com/aspnet/core">building Web apps with
ASP.NET Core</a>.</p>
    <p>Hello ASP.NET Core!</p>
</div>
```

- Select **Publish** from the **Publish Profile summary** page again.



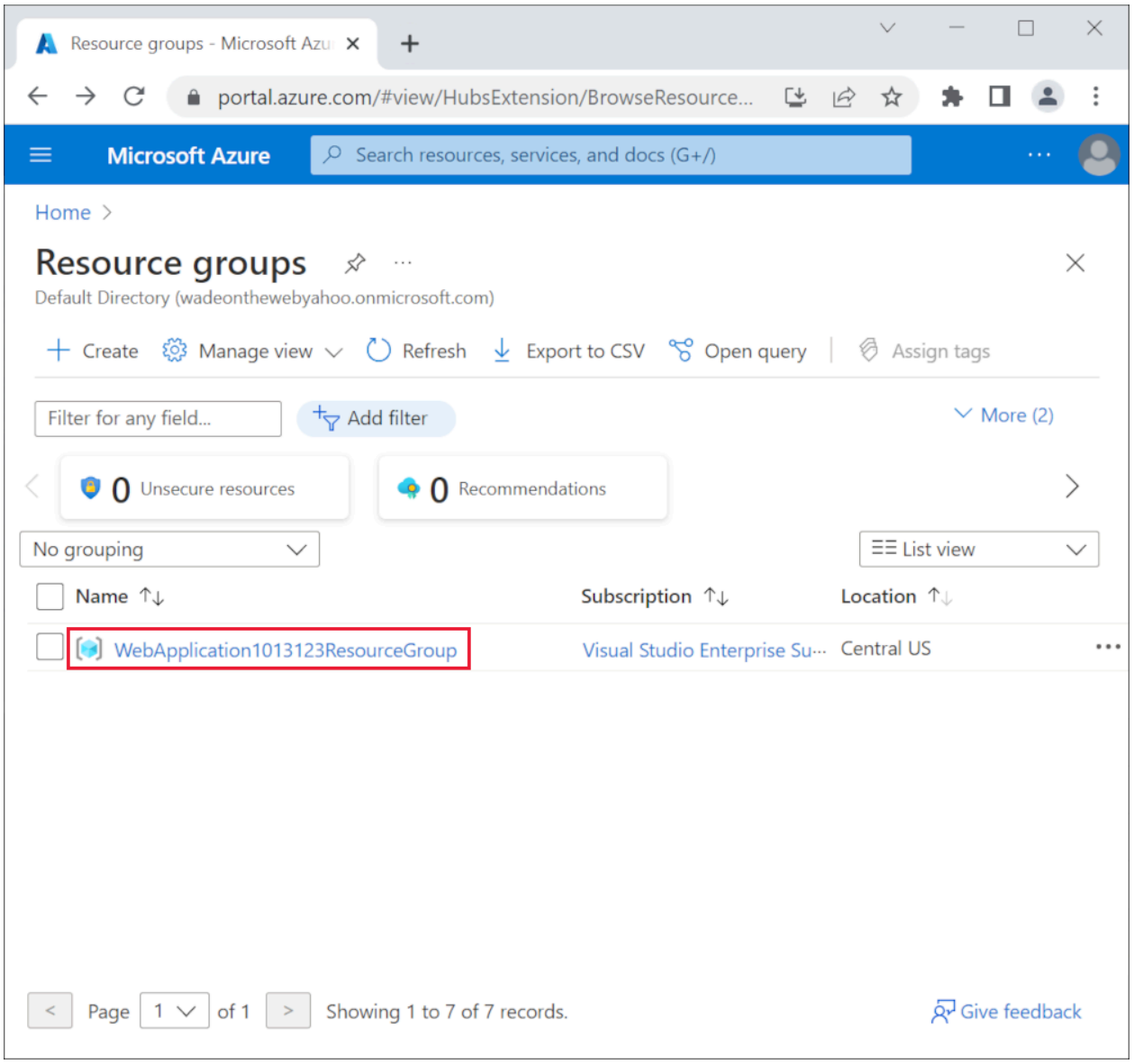
- After the app is published, refresh the page and verify that the changes you made are available on Azure.



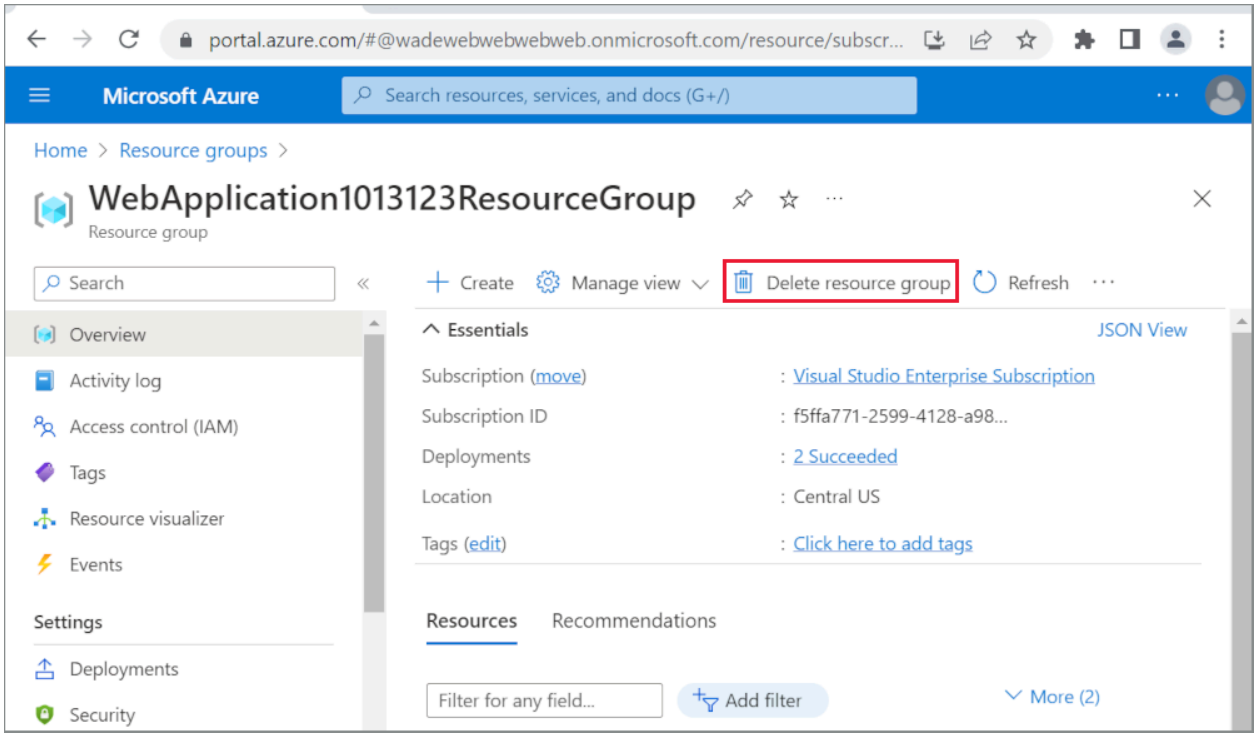
Clean up

When you have finished testing the app, go to the Azure portal and delete the app.

- Select **Resource groups**, then select the resource group you created.



- In the **Resource group** page, select **Delete resource group**.



- Enter the name of the resource group and select **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure.

Additional resources

- Azure App Service
- Azure resource groups
- Azure SQL Database
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment
- Troubleshoot ASP.NET Core on Azure App Service and IIS

Tutorial: Deploy an ASP.NET Core and Azure SQL Database app to Azure App Service

Article • 09/06/2024

In this tutorial, you learn how to deploy a data-driven ASP.NET Core app to Azure App Service and connect to an Azure SQL Database. You'll also deploy an Azure Cache for Redis to enable the caching code in your application. Azure App Service is a highly scalable, self-patching, web-hosting service that can easily deploy apps on Windows or Linux. Although this tutorial uses an ASP.NET Core 8.0 app, the process is the same for other versions of ASP.NET Core.

In this tutorial, you learn how to:

- ✓ Create a secure-by-default App Service, SQL Database, and Redis cache architecture.
- ✓ Secure connection secrets using a managed identity and Key Vault references.
- ✓ Deploy a sample ASP.NET Core app to App Service from a GitHub repository.
- ✓ Access App Service connection strings and app settings in the application code.
- ✓ Make updates and redeploy the application code.
- ✓ Generate database schema by uploading a migrations bundle.
- ✓ Stream diagnostic logs from Azure.
- ✓ Manage the app in the Azure portal.
- ✓ Provision the same architecture and deploy by using Azure Developer CLI.
- ✓ Optimize your development workflow with GitHub Codespaces and GitHub Copilot.

Prerequisites

- An Azure account with an active subscription. If you don't have an Azure account, you can create one for free.
- A GitHub account. you can also get one for free.
- Knowledge of ASP.NET Core development.
- **(Optional)** To try GitHub Copilot, a GitHub Copilot account. A 30-day free trial is available.

Skip to the end

You can quickly deploy the sample app in this tutorial and see it running in Azure. Just run the following commands in the Azure Cloud Shell, and follow the prompt:

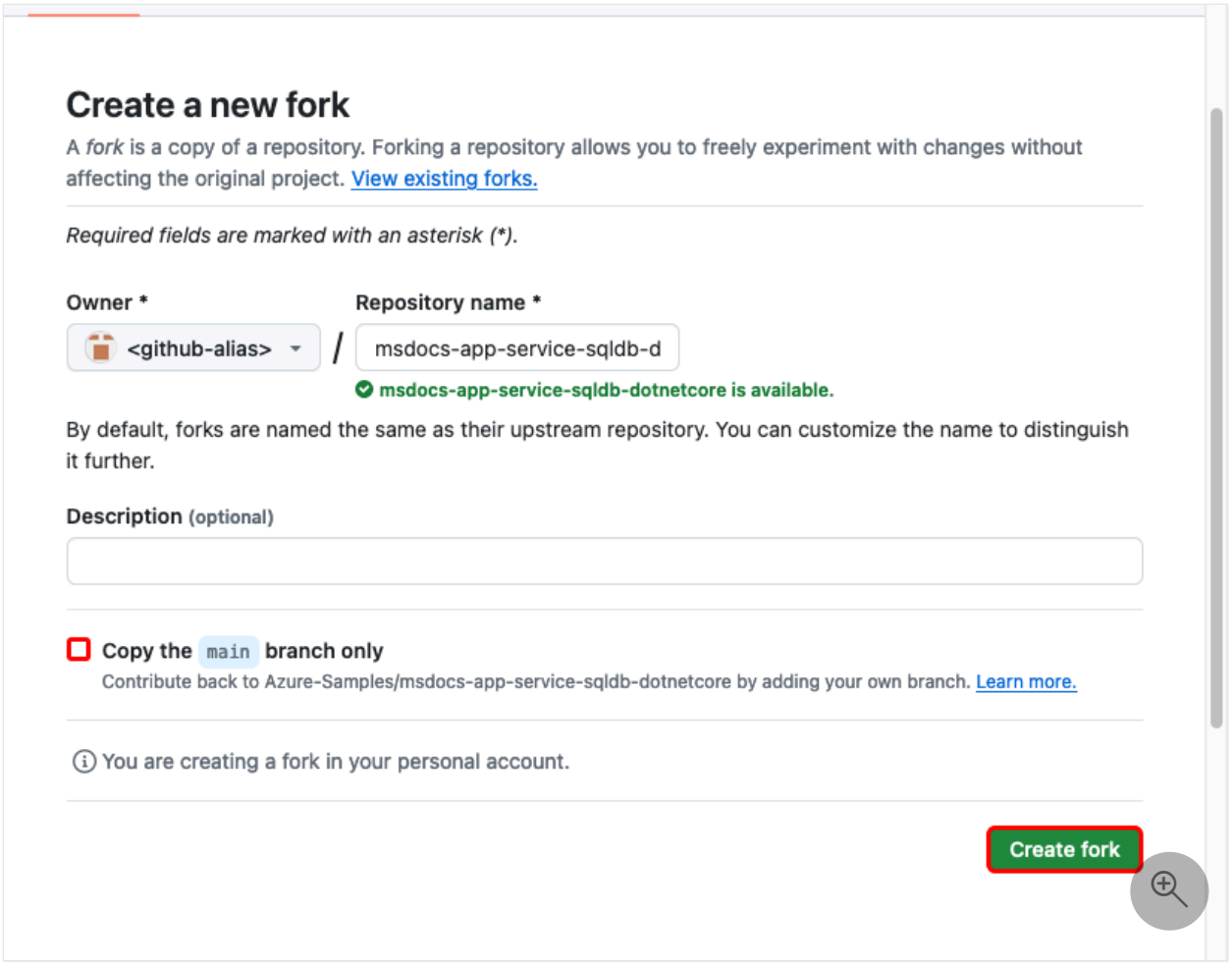
```bash
dotnet tool install --global dotnet-ef
mkdir msdocs-app-service-sqldb-dotnetcore
cd msdocs-app-service-sqldb-dotnetcore
azd init --template msdocs-app-service-sqldb-dotnetcore
azd up
```

1. Run the sample

First, you set up a sample data-driven app as a starting point. For your convenience, the sample repository, includes a dev container configuration. The dev container has everything you need to develop an application, including the database, cache, and all environment variables needed by the sample application. The dev container can run in a GitHub codespace, which means you can run the sample on any computer with a web browser.

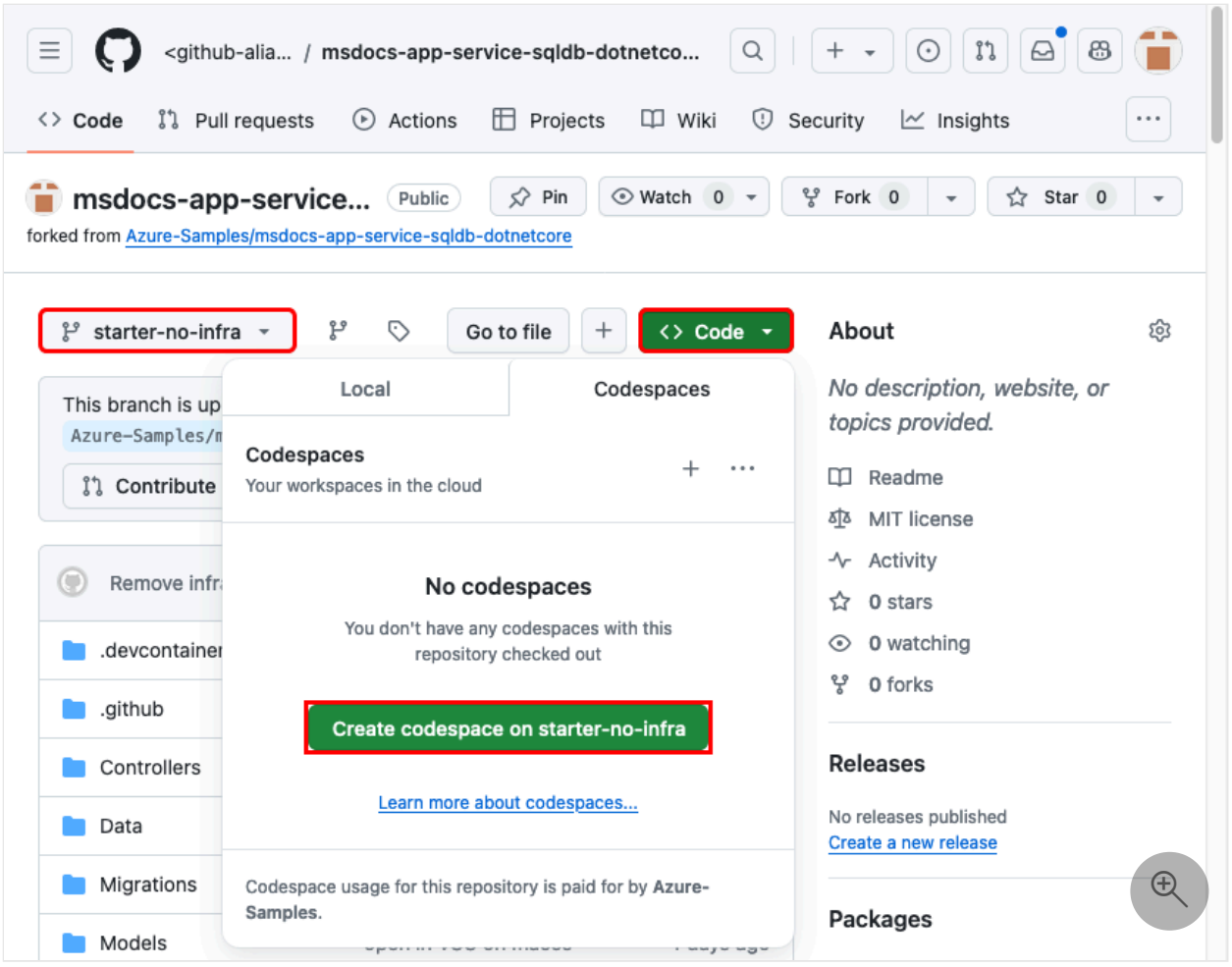
Step 1: In a new browser window:

1. Sign in to your GitHub account.
2. Navigate to https://github.com/Azure-Samples/msdocs-app-service-sqldb-dotnetcore/fork.
3. Unselect **Copy the main branch only**. You want all the branches.
4. Select **Create fork**.



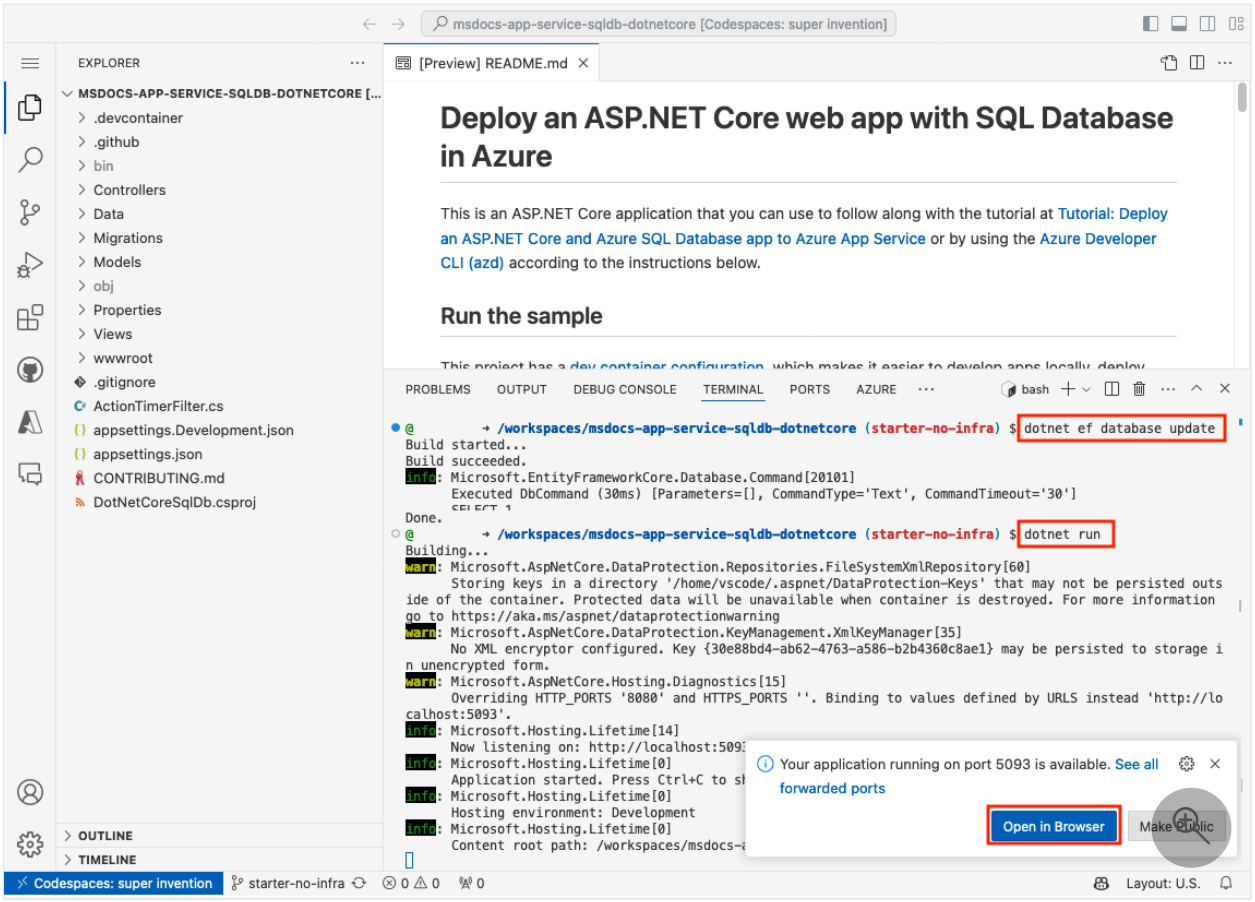
Step 2: In the GitHub fork:

1. Select **main** > **starter-no-infra** for the starter branch. This branch contains just the sample project and no Azure-related files or configuration.
2. Select **Code** > **Create codespace on starter-no-infra**. The codespace takes a few minutes to set up.



Step 3: In the codespace terminal:

1. Run database migrations with `dotnet ef database update`.
2. Run the app with `dotnet run`.
3. When you see the notification `Your application running on port 5093 is available.`, select **Open in Browser**. You should see the sample application in a new browser tab. To stop the application, type `Ctrl` + `C`.



> **Tip**
>
> You can ask GitHub Copilot about this repository. For example:
>
> - *@workspace What does this project do?*
> - *@workspace What does the .devcontainer folder do?*

Having issues? Check the Troubleshooting section.

2. Create App Service, database, and cache

In this step, you create the Azure resources. The steps used in this tutorial create a set of secure-by-default resources that include App Service, Azure SQL Database, and Azure Cache. For the creation process, you'll specify:

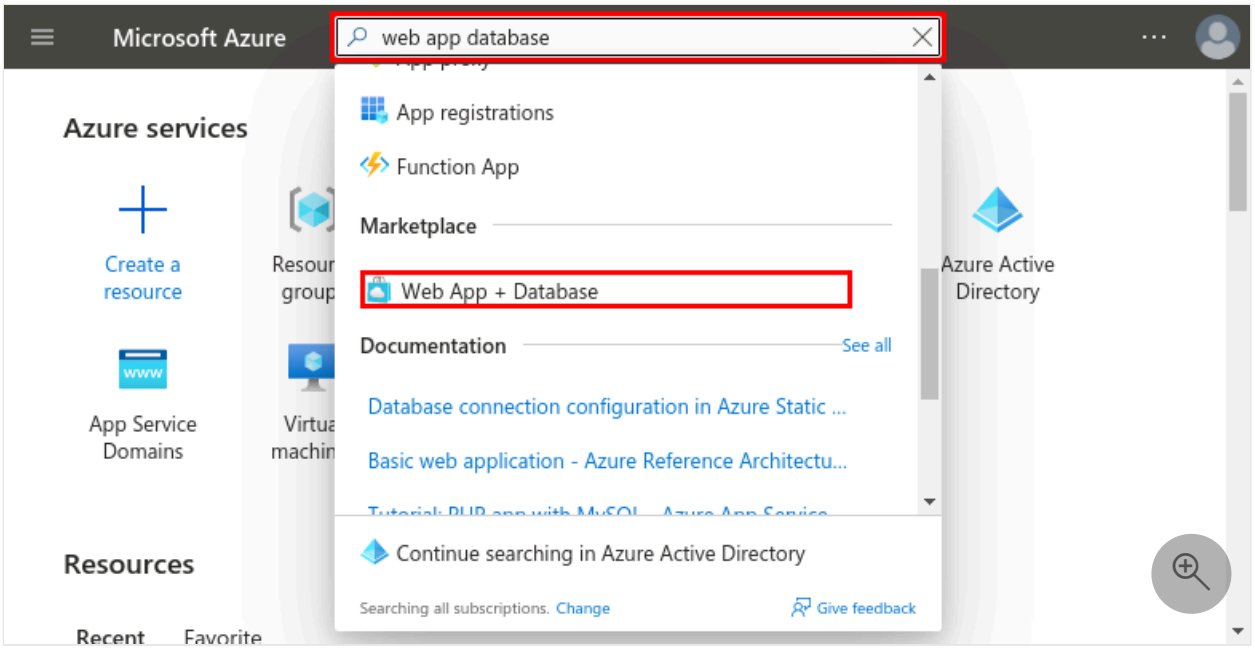
- The **Name** for the web app. It's used as part of the DNS name for your app in the form of `https://<app-name>-<hash>.<region>.azurewebsites.net`.
- The **Region** to run the app physically in the world. It's also used as part of the DNS name for your app.
- The **Runtime stack** for the app. It's where you select the .NET version to use for your app.

- The **Hosting plan** for the app. It's the pricing tier that includes the set of features and scaling capacity for your app.
- The **Resource Group** for the app. A resource group lets you group (in a logical container) all the Azure resources needed for the application.

Sign in to the Azure portal and follow these steps to create your Azure App Service resources.

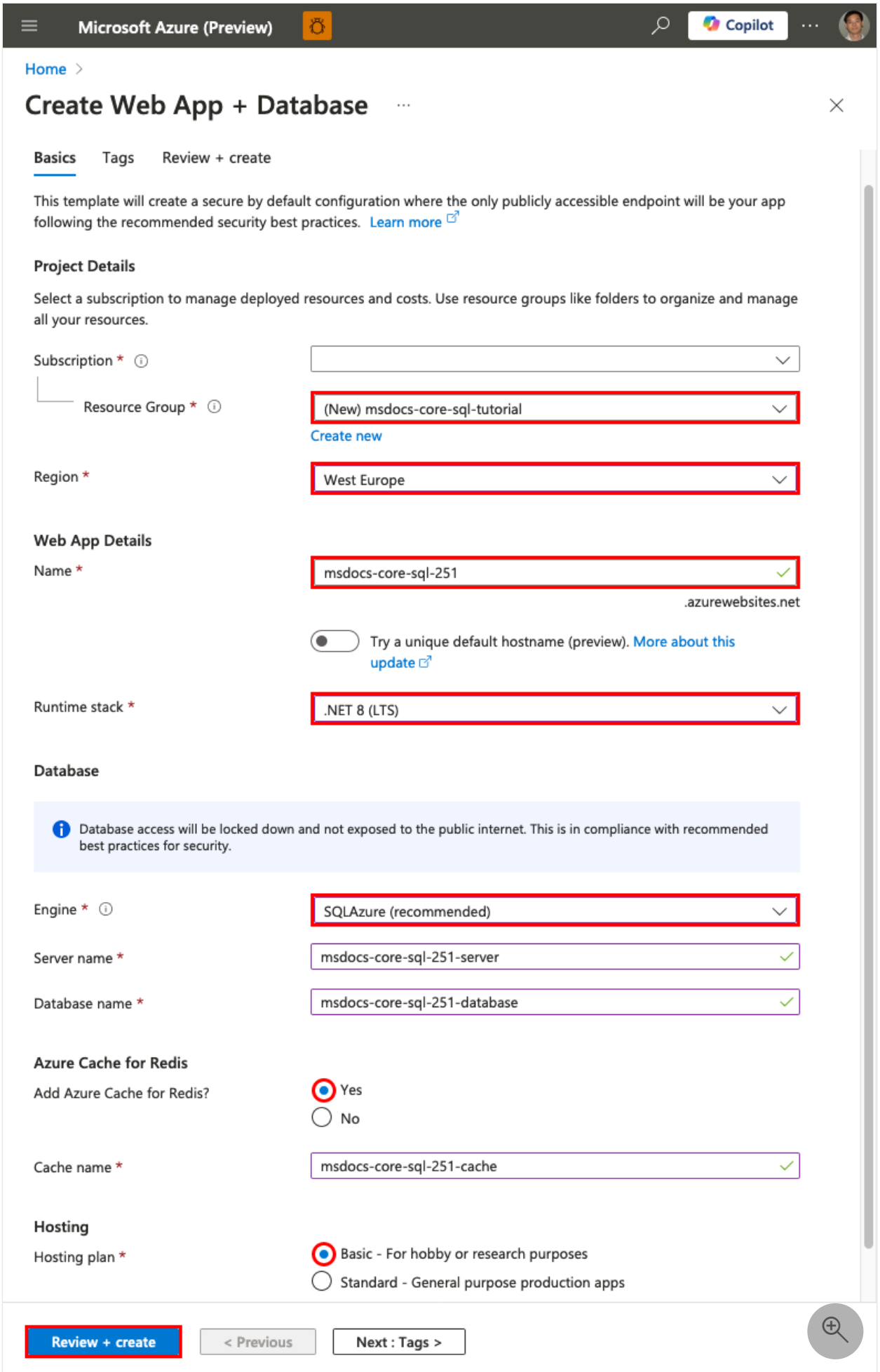
Step 1: In the Azure portal:

1. Enter "web app database" in the search bar at the top of the Azure portal.
2. Select the item labeled **Web App + Database** under the **Marketplace** heading. You can also navigate to the creation wizard directly.



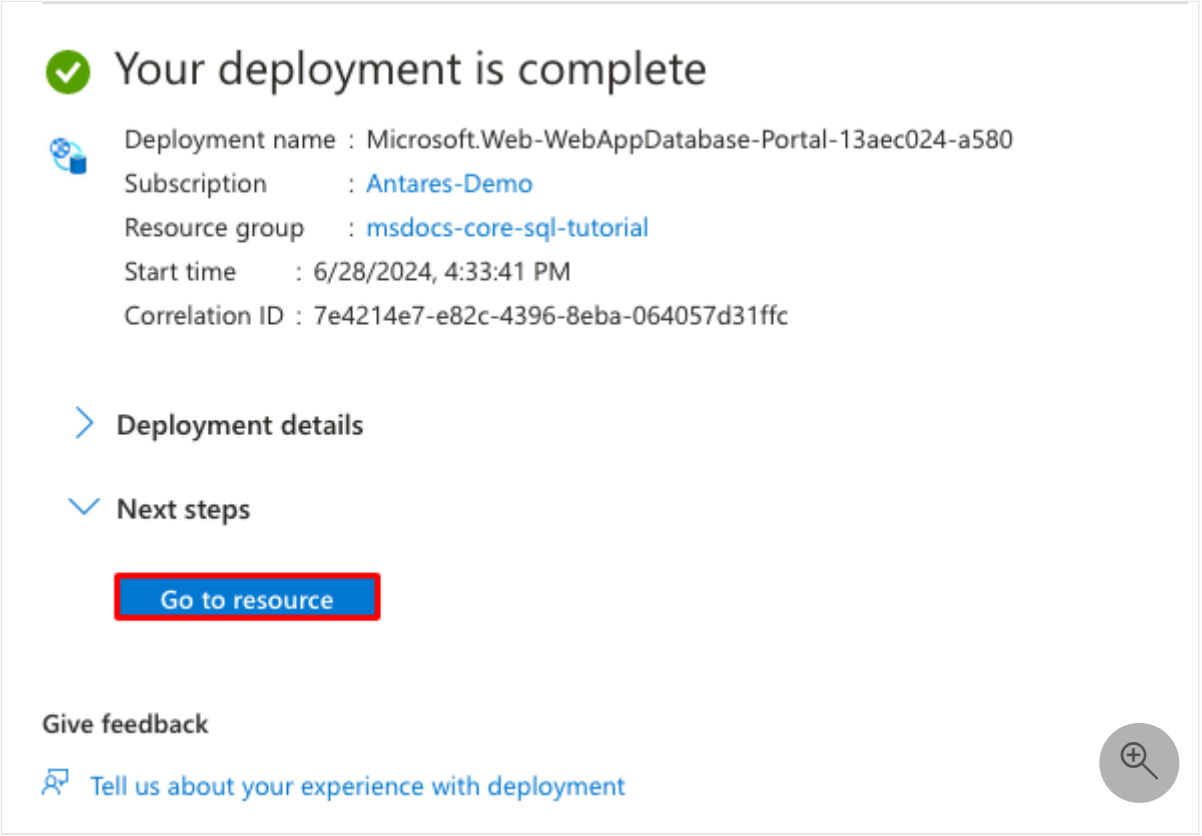
Step 2: In the **Create Web App + Database** page, fill out the form as follows.

1. *Resource Group*: Select **Create new** and use a name of **msdocs-core-sql-tutorial**.
2. *Region*: Any Azure region near you.
3. *Name*: **msdocs-core-sql-XYZ** where *XYZ* is any three random characters. This name must be unique across Azure.
4. *Runtime stack*: **.NET 8 (LTS)**.
5. *Engine*: **SQLAzure**. Azure SQL Database is a fully managed platform as a service (PaaS) database engine that's always running on the latest stable version of the SQL Server.
6. *Add Azure Cache for Redis?*: **Yes**.
7. *Hosting plan*: **Basic**. When you're ready, you can scale up to a production pricing tier.
8. Select **Review + create**.
9. After validation completes, select **Create**.


Microsoft Azure (Preview)
Copilot
Home
Create Web App + Database
Basics
Tags
Review + create
This template will create a secure by default configuration where the only publicly accessible endpoint will be your app following the recommended security best practices. Learn more
Project Details
Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.
Subscription *
Resource Group *
(New) msdocs-core-sql-tutorial
Create new
Region *
West Europe
Web App Details
Name *
msdocs-core-sql-251
.azurewebsites.net
Try a unique default hostname (preview). More about this update
Runtime stack *
.NET 8 (LTS)
Database
Database access will be locked down and not exposed to the public internet. This is in compliance with recommended best practices for security.
Engine *
SQLAzure (recommended)
Server name *
msdocs-core-sql-251-server
Database name *
msdocs-core-sql-251-database
Azure Cache for Redis
Add Azure Cache for Redis?
Yes
No
Cache name *
msdocs-core-sql-251-cache
Hosting
Hosting plan *
Basic - For hobby or research purposes
Standard - General purpose production apps
Review + create
< Previous
Next : Tags >

Step 3: The deployment takes a few minutes to complete. Once deployment completes, select the **Go to resource** button. You're taken directly to the App Service app, but the following resources are created:

- **Resource group**: The container for all the created resources.
- **App Service plan**: Defines the compute resources for App Service. A Linux plan in the *Basic* tier is created.
- **App Service**: Represents your app and runs in the App Service plan.
- **Virtual network**: Integrated with the App Service app and isolates back-end network traffic.
- **Private endpoints**: Access endpoints for the key vault, the database server, and the Redis cache in the virtual network.
- **Network interfaces**: Represents private IP addresses, one for each of the private endpoints.
- **Azure SQL Database server**: Accessible only from behind its private endpoint.
- **Azure SQL Database**: A database and a user are created for you on the server.
- **Azure Cache for Redis**: Accessible only from behind its private endpoint.
- **Key vault**: Accessible only from behind its private endpoint. Used to manage secrets for the App Service app.
- **Private DNS zones**: Enable DNS resolution of the key vault, the database server, and the Redis cache in the virtual network.



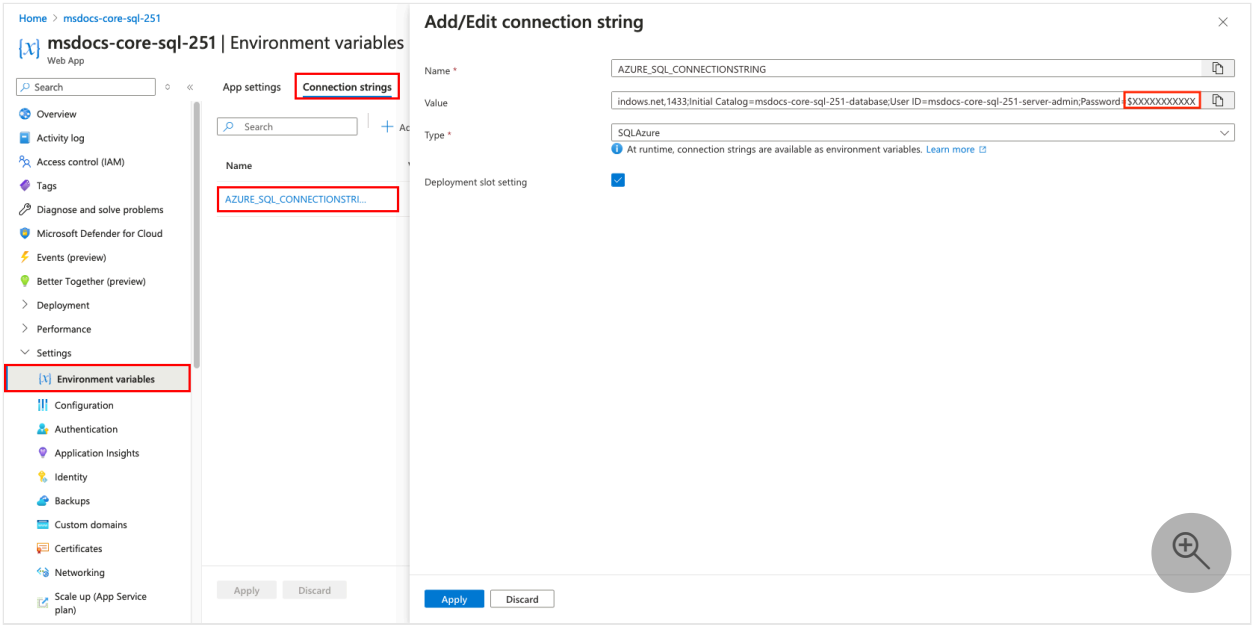
3. Secure connection secrets

The creation wizard generated the connectivity string for you already as .NET connection strings and app settings. However, the security best practice is to keep secrets out of App Service completely. You'll move your secrets to a key vault and change your app setting to Key Vault references with the help of Service Connectors.

> **Tip**
>
> To use passwordless authentication, see **How do I change the SQL Database connection to use a managed identity instead?**

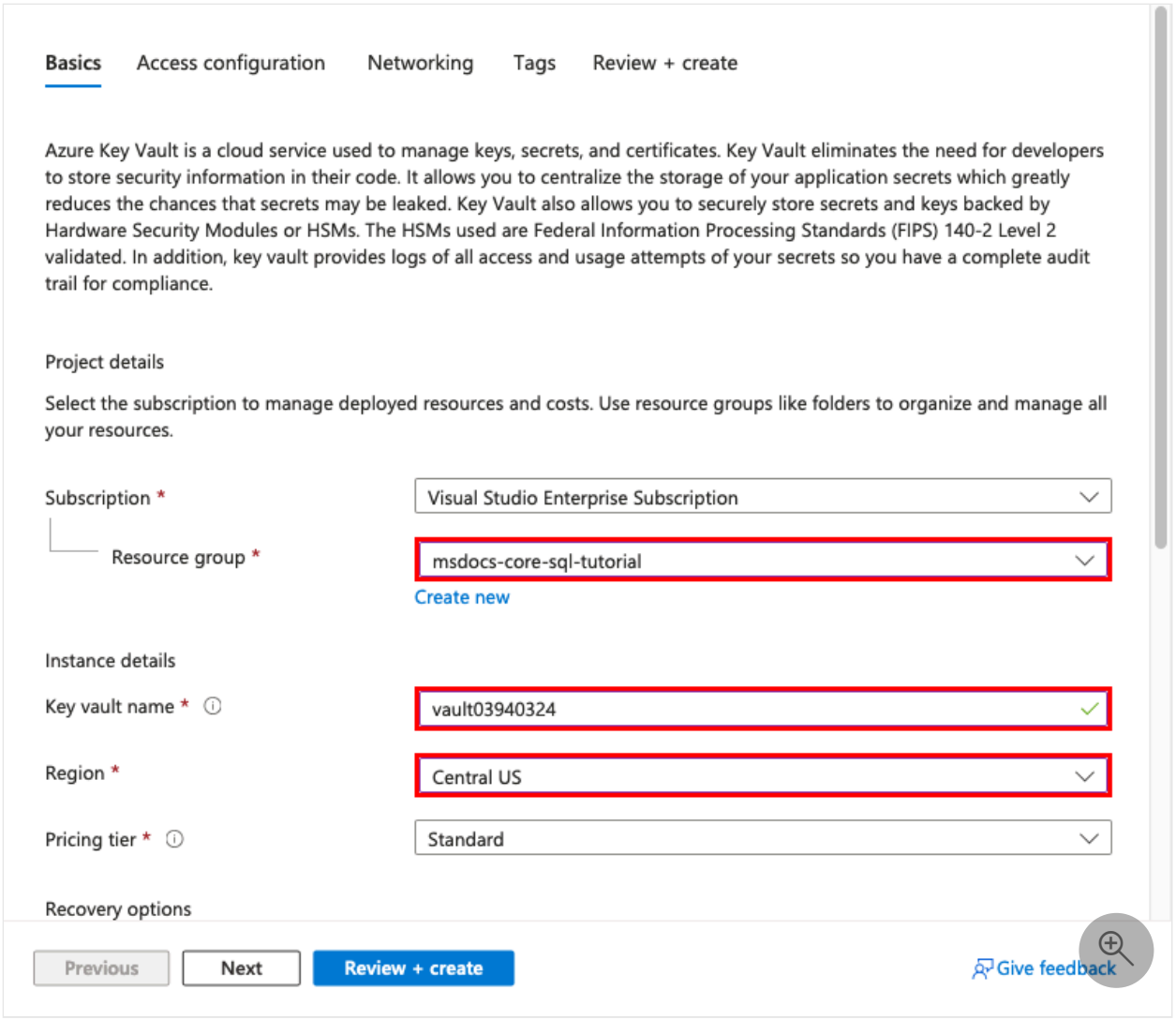
Step 1: In the App Service page:

1. In the left menu, select **Settings** > **Environment variables** > **Connection strings**.
2. Select **AZURE_SQL_CONNECTIONSTRING**.
3. In **Add/Edit connection string**, in the **Value** field, find the *Password*= part at the end of the string.
4. Copy the password string after *Password*= for use later. This connection string lets you connect to the SQL database secured behind a private endpoint. The password is saved directly in the App Service app, which isn't the best. Likewise, the Redis cache connection string in the **App settings** tab contains a secret. You'll change this.



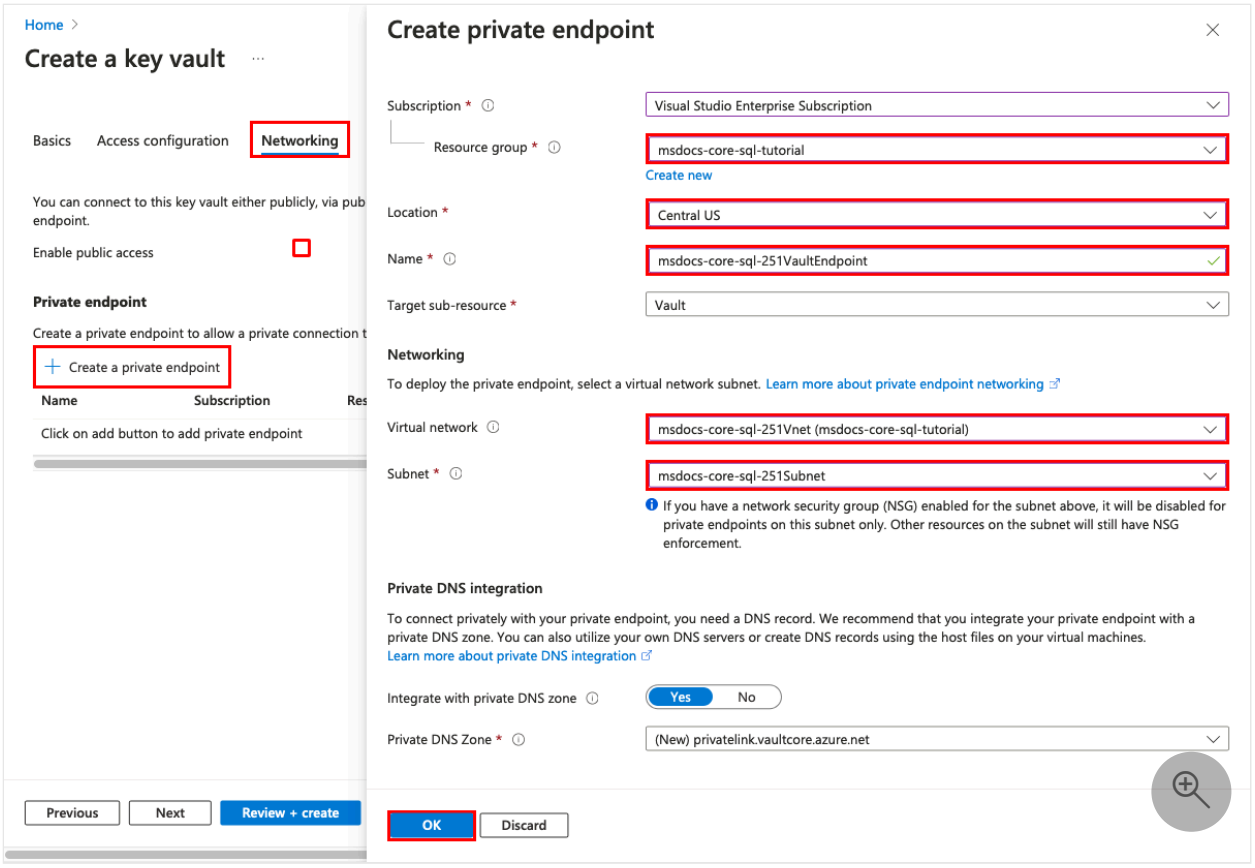
Step 2: Create a key vault for secure management of secrets.

1. In the top search bar, type "*key vault*", then select **Marketplace** > **Key Vault**.
2. In **Resource Group**, select **msdocs-core-sql-tutorial**.
3. In **Key vault name**, type a name that consists of only letters and numbers.
4. In **Region**, set it to the sample location as the resource group.



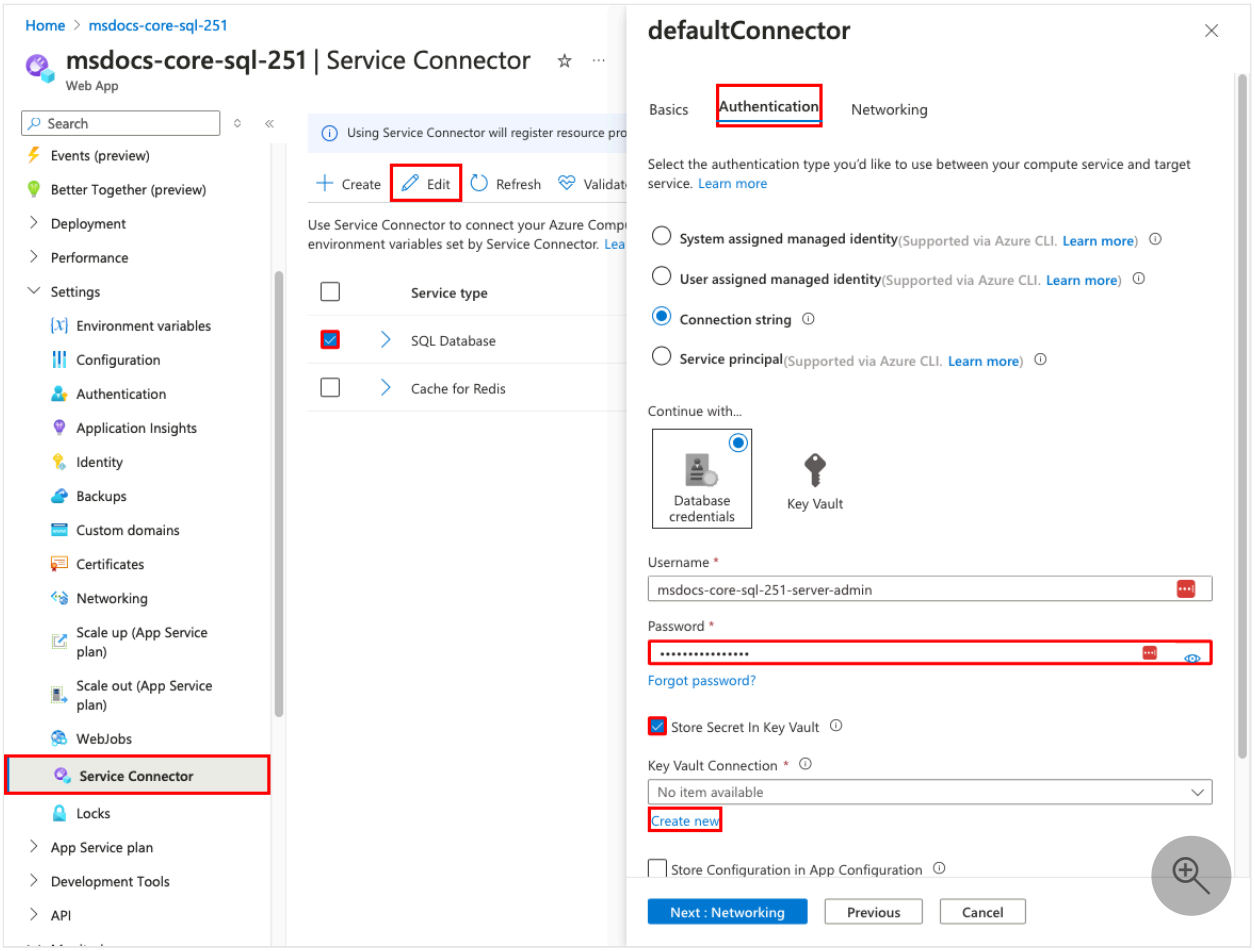
Step 3:

1. Select the **Networking** tab.
2. Unselect **Enable public access**.
3. Select **Create a private endpoint**.
4. In **Resource Group**, select **msdocs-core-sql-tutorial**.
5. In **Key vault name**, type a name that consists of only letters and numbers.
6. In **Region**, set it to the sample location as the resource group.
7. In the dialog, in **Location**, select the same location as your App Service app.
8. In **Resource Group**, select **msdocs-core-sql-tutorial**.
9. In **Name**, type **msdocs-core-sql-XYZVvaultEndpoint**.
10. In **Virtual network**, select **msdocs-core-sql-XYZVnet**.
11. In **Subnet**, **msdocs-core-sql-XYZSubnet**.
12. Select **OK**.
13. Select **Review + create**, then select **Create**. Wait for the key vault deployment to finish. You should see "Your deployment is complete."



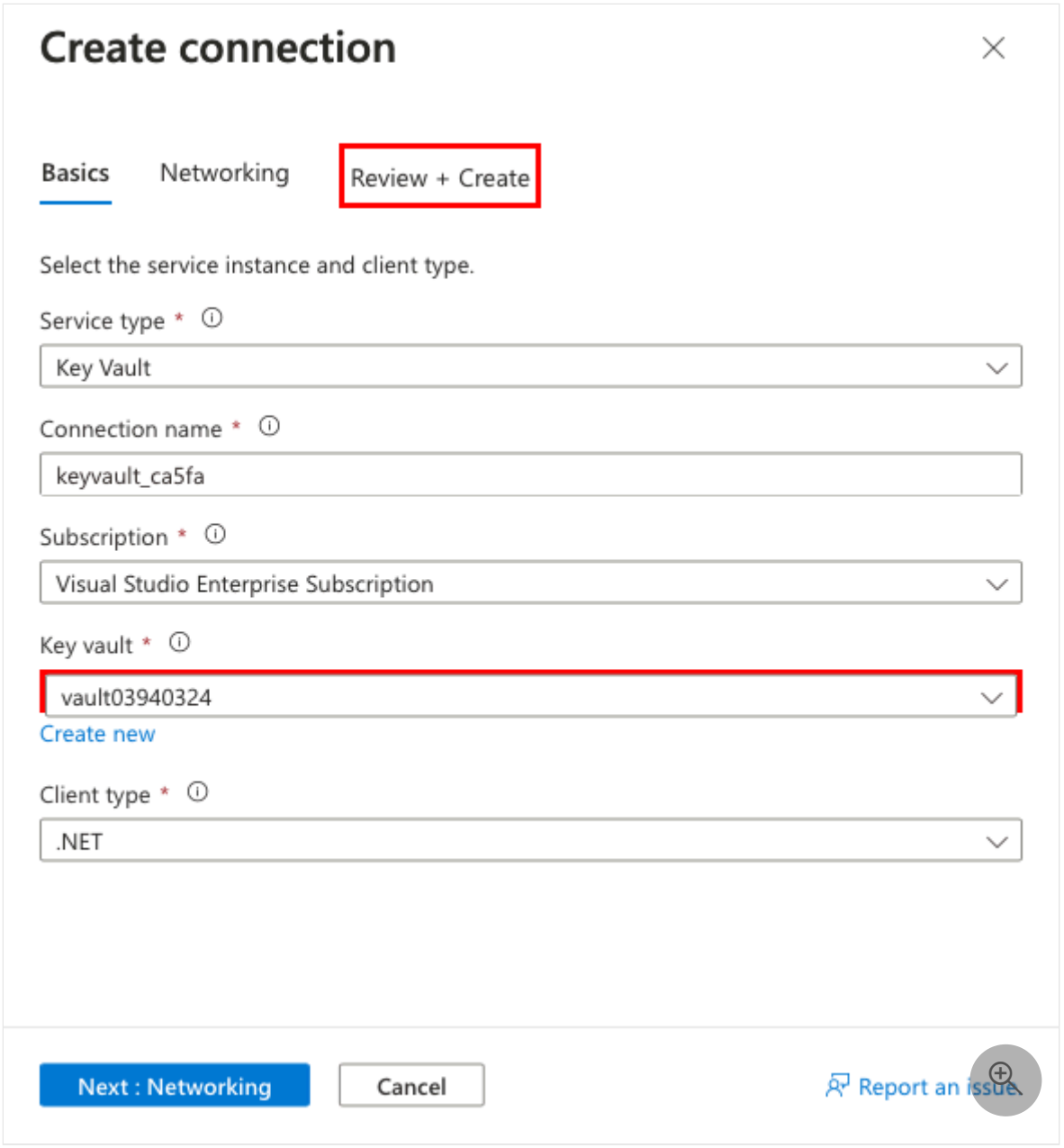
Step 4:

1. In the top search bar, type *msdocs-core-sql*, then the App Service resource called **msdocs-core-sql-XYZ**.
2. In the App Service page, in the left menu, select **Settings** > **Service Connector**. There are already two connectors, which the app creation wizard created for you.
3. Select checkbox next to the SQL Database connector, then select **Edit**.
4. Select the **Authentication** tab.
5. In **Password**, paste the password you copied earlier.
6. Select **Store Secret in Key Vault**.
7. Under **Key Vault Connection**, select **Create new**. A **Create connection** dialog is opened on top of the edit dialog.



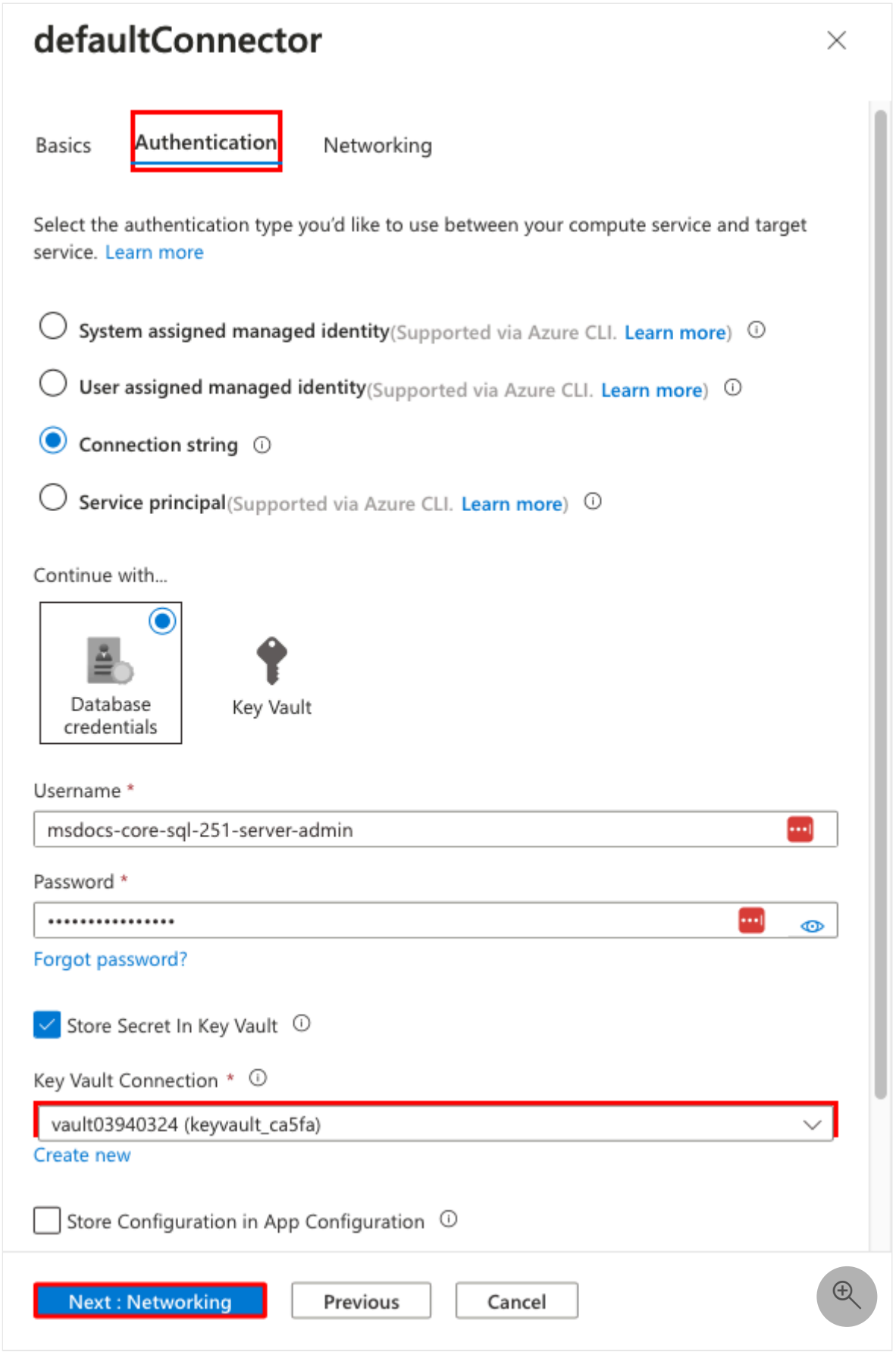
Step 5: In the **Create connection** dialog for the Key Vault connection:

1. In **Key Vault**, select the key vault you created earlier.
2. Select **Review + Create**. You should see that **System assigned managed identity** is set to **Selected**.
3. When validation completes, select **Create**.



Step 6: You're back in the edit dialog for **defaultConnector**.

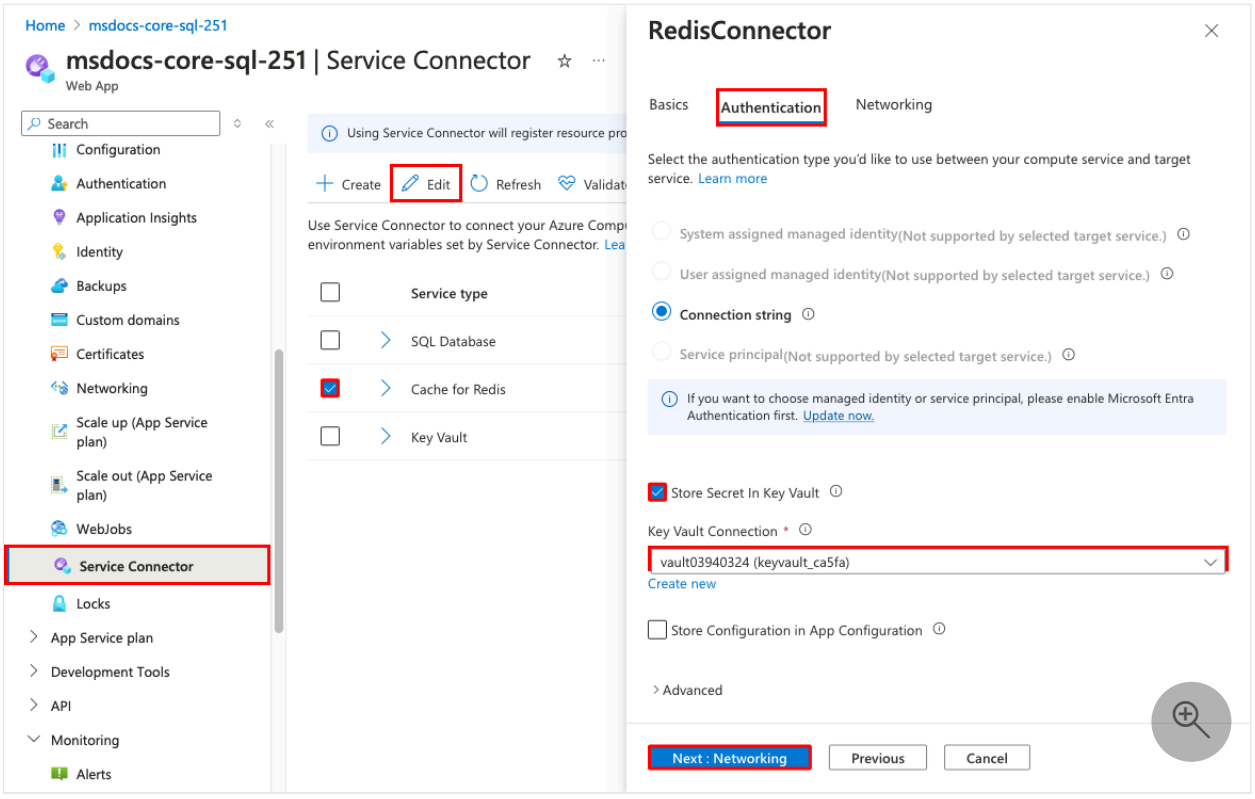
1. In the **Authentication** tab, wait for the key vault connector to be created. When it's finished, the **Key Vault Connection** dropdown automatically selects it.
2. Select **Next: Networking**.
3. Select **Configure firewall rules to enable access to target service**. The app creation wizard already secured the SQL database with a private endpoint.
4. Select **Save**. Wait until the **Update succeeded** notification appears.



Step 7: In the Service Connectors page:

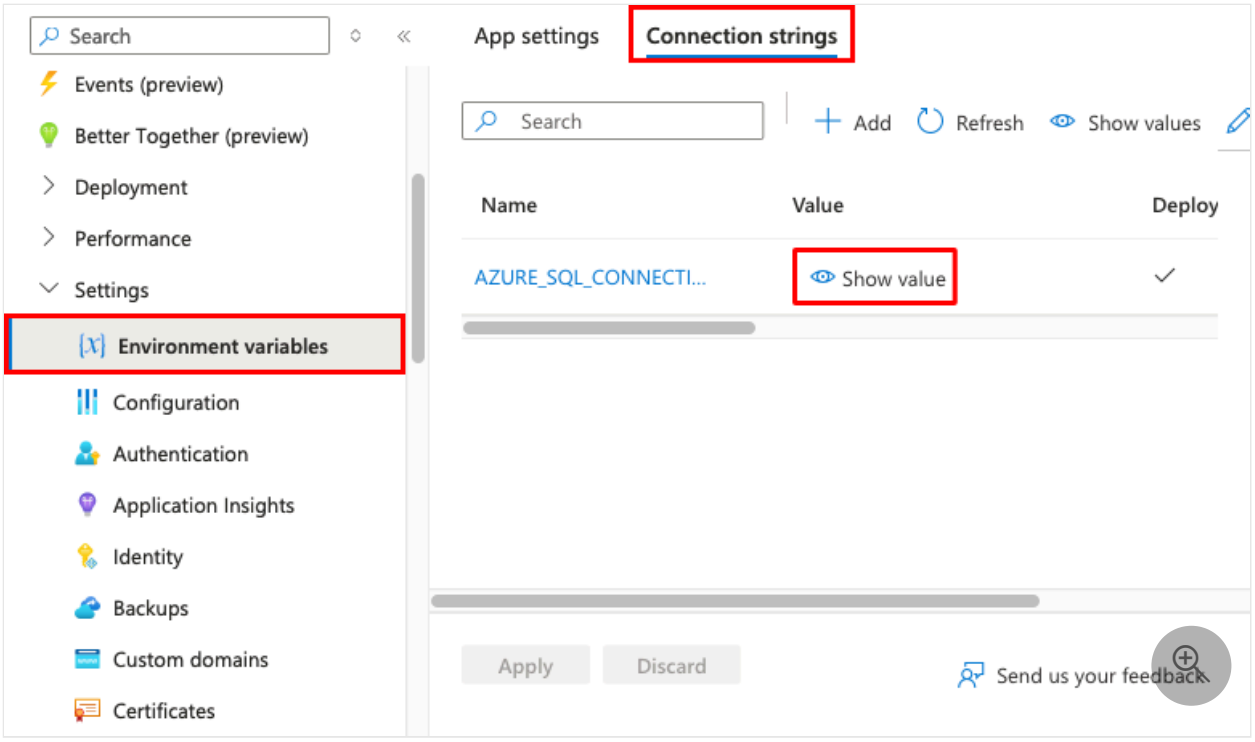
1. Select checkbox next to the Cache for Redis connector, then select **Edit**.
2. Select the **Authentication** tab.
3. Select **Store Secret in Key Vault**.

4. Under **Key Vault Connection**, select the key vault you created.
5. Select **Next: Networking**.
6. Select **Configure firewall rules to enable access to target service**. The app creation wizard already secured the SQL database with a private endpoint.
7. Select **Save**. Wait until the **Update succeeded** notification appears.



Step 8: To verify your changes:

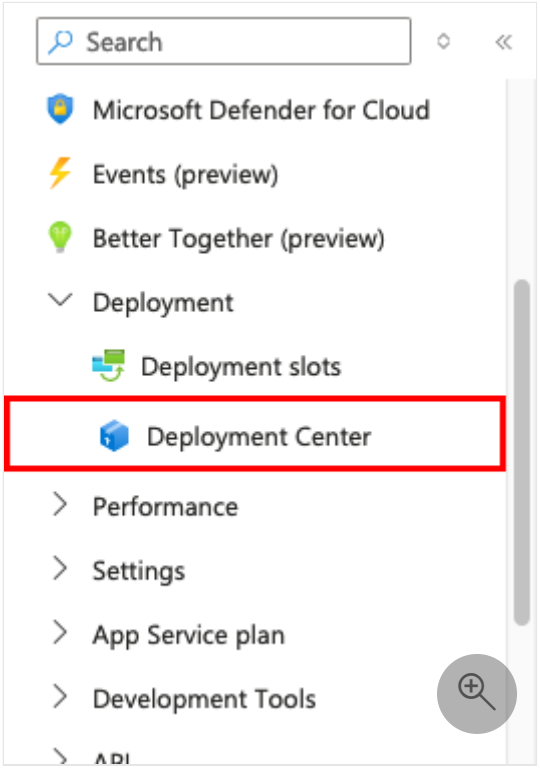
1. From the left menu, select **Environment variables** > **Connection strings** again.
2. Next to **AZURE_SQL_CONNECTIONSTRING**, select **Show value**. The value should be `@Microsoft.KeyVault(...)`, which means that it's a key vault reference because the secret is now managed in the key vault.
3. To verify the Redis connection string, select the **App setting** tab. Next to **AZURE_REDIS_CONNECTIONSTRING**, select **Show value**. The value should be `@Microsoft.KeyVault(...)` too.



4. Deploy sample code

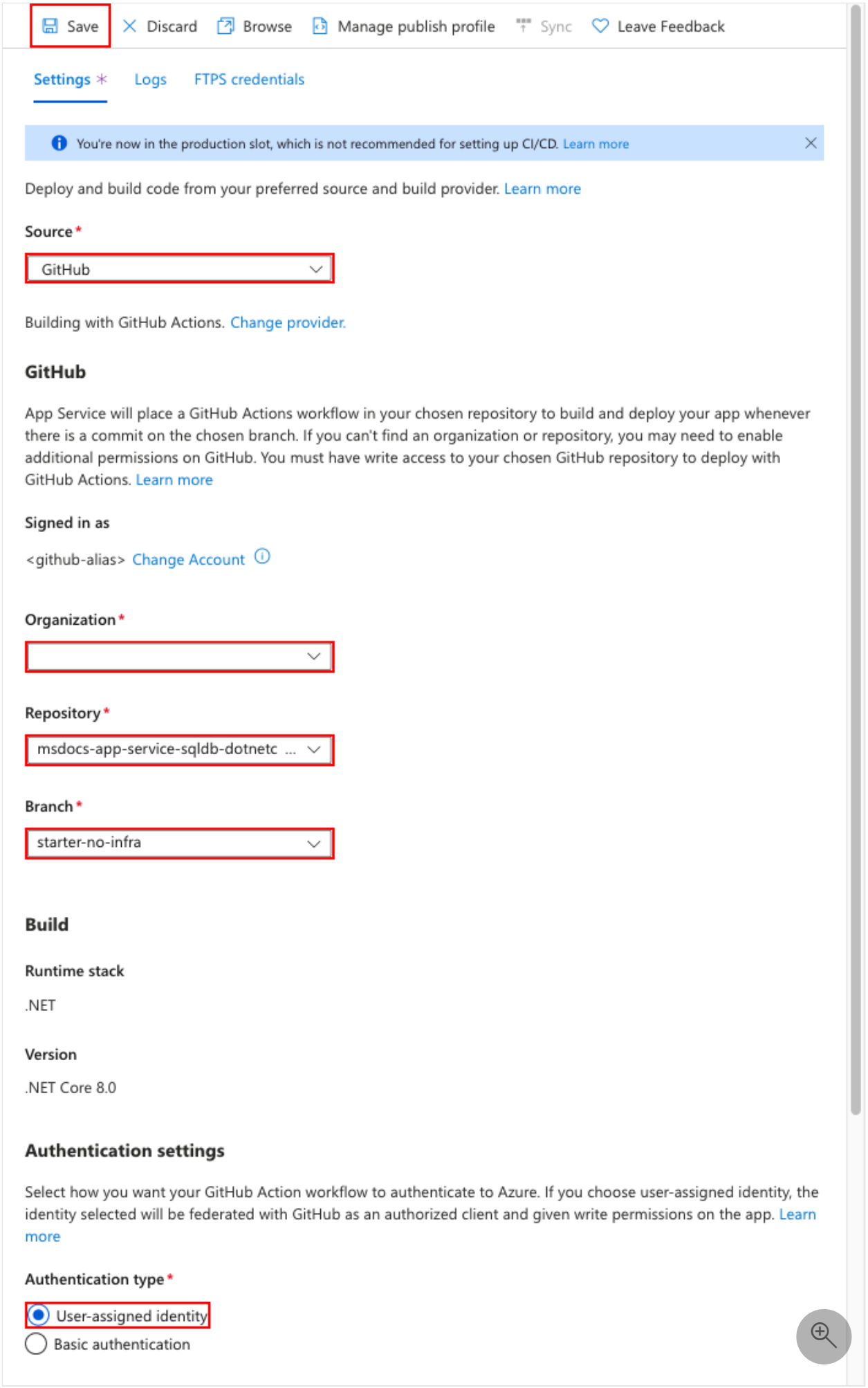
In this step, you configure GitHub deployment using GitHub Actions. It's just one of many ways to deploy to App Service, but also a great way to have continuous integration in your deployment process. By default, every `git push` to your GitHub repository kicks off the build and deploy action.

Step 1: In the left menu, select **Deployment** > **Deployment Center**.

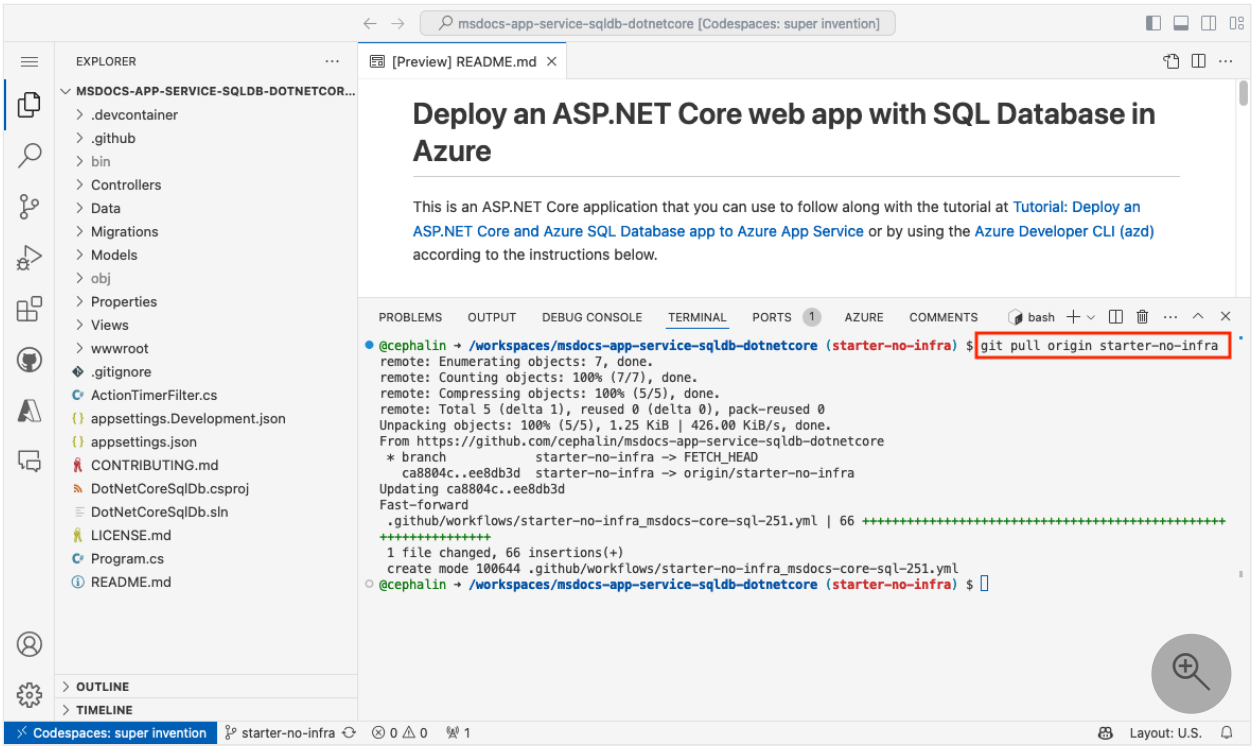


Step 2: In the Deployment Center page:

1. In **Source**, select **GitHub**. By default, **GitHub Actions** is selected as the build provider.
2. Sign in to your GitHub account and follow the prompt to authorize Azure.
3. In **Organization**, select your account.
4. In **Repository**, select **msdocs-app-service-sqldb-dotnetcore**.
5. In **Branch**, select **starter-no-infra**. This is the same branch that you worked in with your sample app, without any Azure-related files or configuration.
6. For **Authentication type**, select **User-assigned identity**.
7. In the top menu, select **Save**. App Service commits a workflow file into the chosen GitHub repository, in the `.github/workflows` directory. By default, the deployment center creates a user-assigned identity for the workflow to authenticate using Microsoft Entra (OIDC authentication). For alternative authentication options, see Deploy to App Service using GitHub Actions.

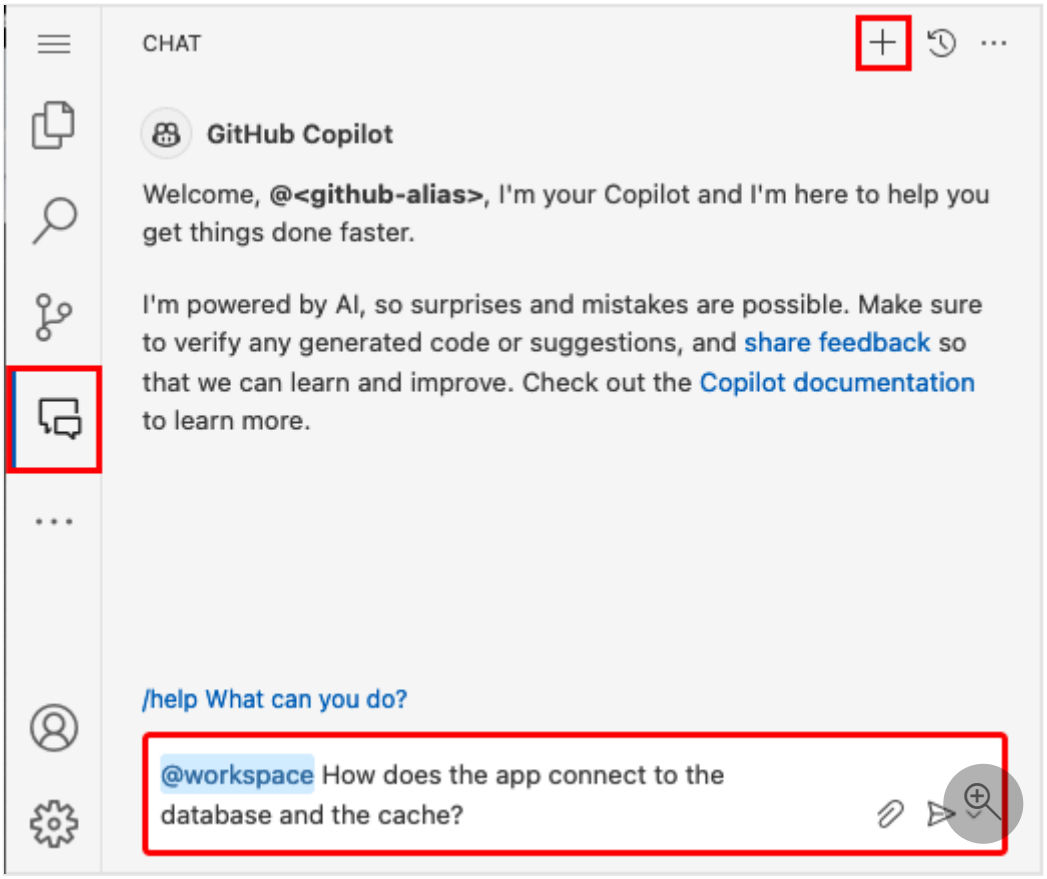

Save
Discard
Browse
Manage publish profile
Sync
Leave Feedback
Settings
Logs
FTPS credentials
You're now in the production slot, which is not recommended for setting up CI/CD. Learn more
Deploy and build code from your preferred source and build provider. Learn more
Source *
GitHub
Building with GitHub Actions. Change provider.
GitHub
App Service will place a GitHub Actions workflow in your chosen repository to build and deploy your app whenever there is a commit on the chosen branch. If you can't find an organization or repository, you may need to enable additional permissions on GitHub. You must have write access to your chosen GitHub repository to deploy with GitHub Actions. Learn more
Signed in as
<github-alias> Change Account
Organization *
Repository *
msdocs-app-service-sqldb-dotnetc ...
Branch *
starter-no-infra
Build
Runtime stack
.NET
Version
.NET Core 8.0
Authentication settings
Select how you want your GitHub Action workflow to authenticate to Azure. If you choose user-assigned identity, the identity selected will be federated with GitHub as an authorized client and given write permissions on the app. Learn more
Authentication type *
User-assigned identity
Basic authentication

Step 3: Back in the GitHub codespace of your sample fork, run `git pull origin starter-no-infra`. This pulls the newly committed workflow file into your codespace.



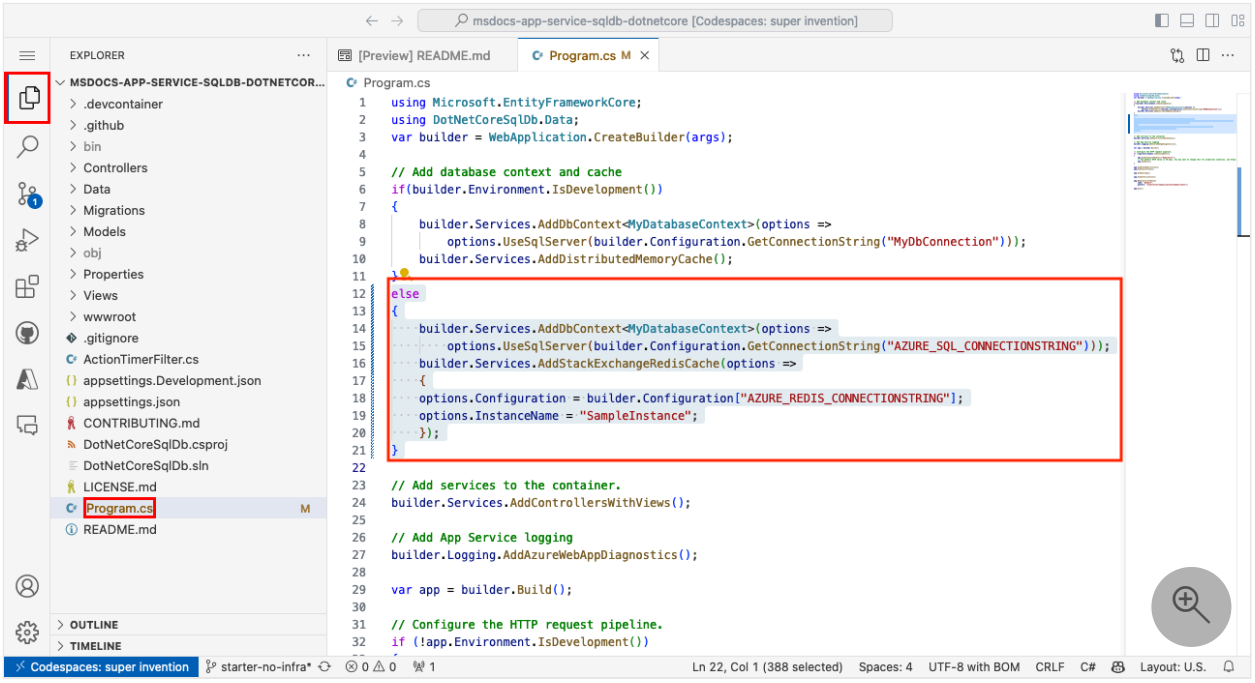
Step 4 (Option 1: with GitHub Copilot):

1. Start a new chat session by selecting the **Chat** view, then selecting **+**.
2. Ask, "*@workspace How does the app connect to the database and the cache?*" Copilot might give you some explanation about the `MyDatabaseContext` class and how it's configured in *Program.cs*.
3. Ask, "In production mode, I want the app to use the connection string called AZURE_SQL_CONNECTIONSTRING for the database and the app setting called AZURE_REDIS_CONNECTIONSTRING*." Copilot might give you a code suggestion similar to the one in the **Option 2: without GitHub Copilot** steps below and even tell you to make the change in the *Program.cs* file.
4. Open *Program.cs* in the explorer and add the code suggestion. GitHub Copilot doesn't give you the same response every time, and it's not always correct. You might need to ask more questions to fine-tune its response. For tips, see What can I do with GitHub Copilot in my codespace?.



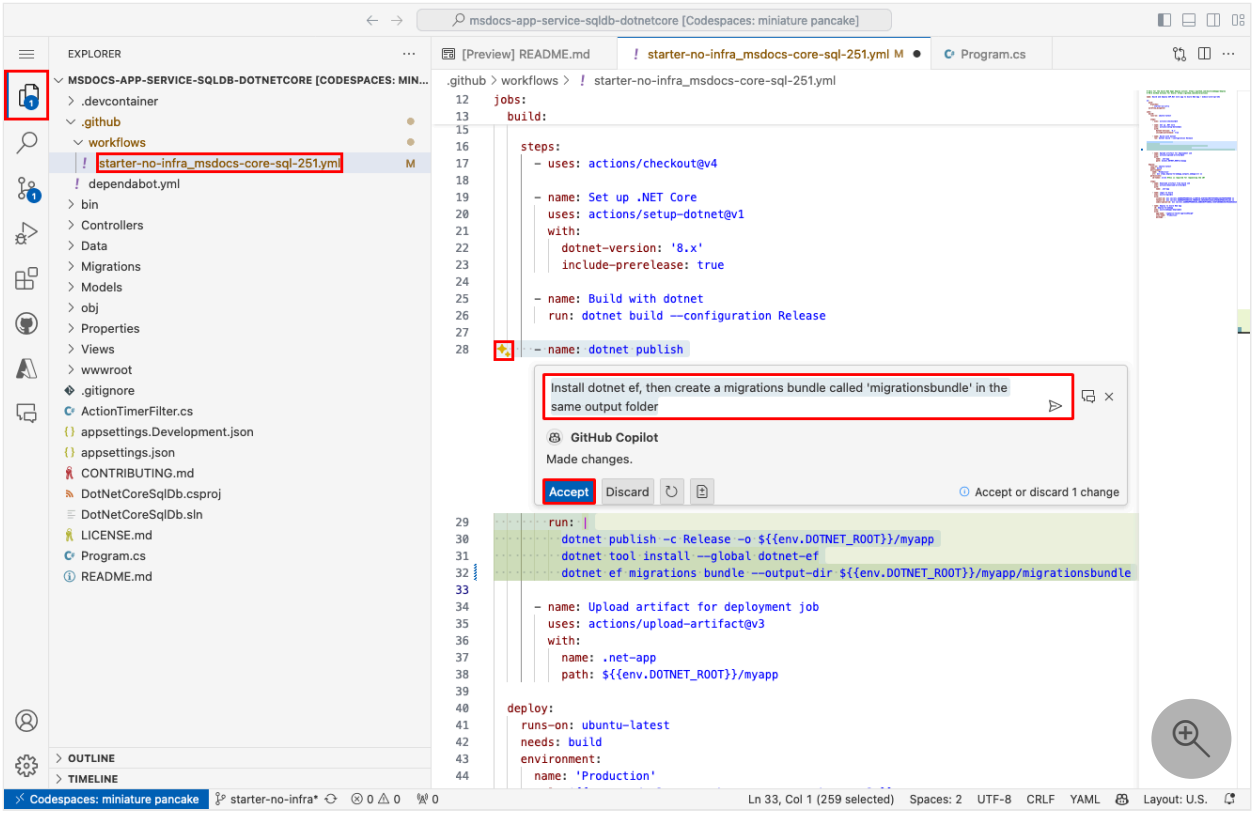
Step 4 (Option 2: without GitHub Copilot):

1. Open *Program.cs* in the explorer.
2. Find the commented code (lines 12-21) and uncomment it. This code connects to the database by using `AZURE_SQL_CONNECTIONSTRING` and connects to the Redis cache by using the app setting `AZURE_REDIS_CONNECTIONSTRING`.



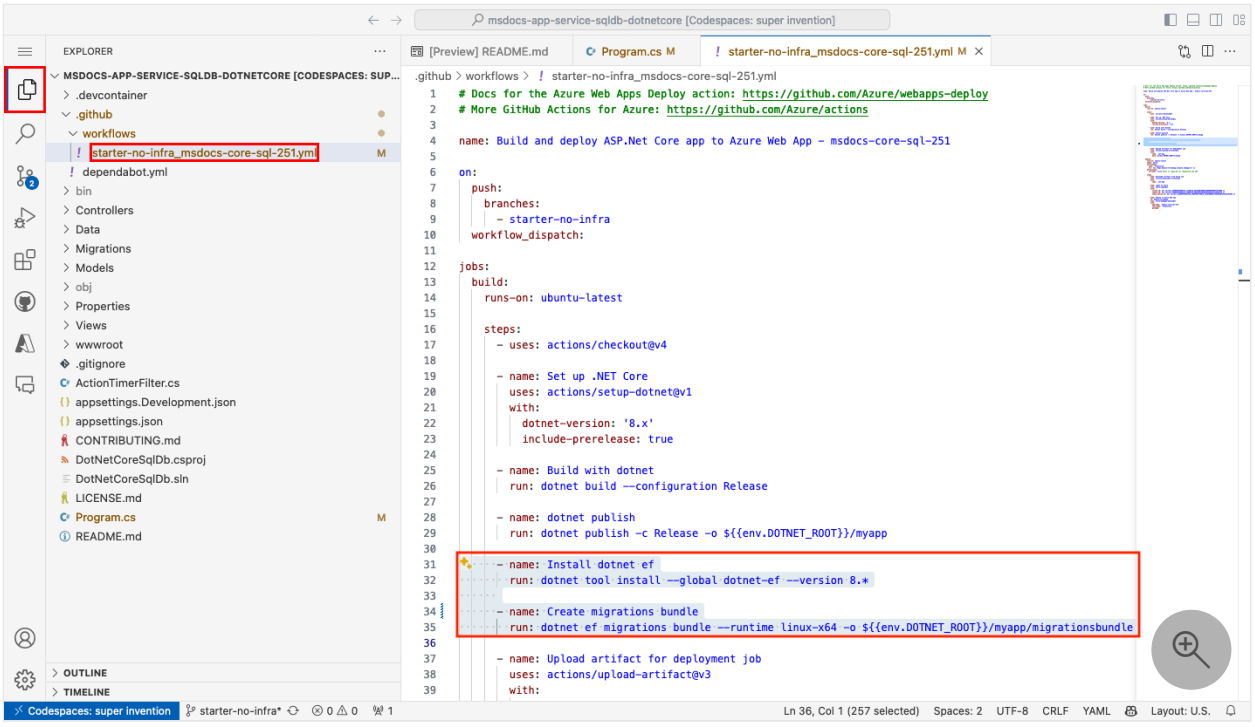
Step 5 (Option 1: with GitHub Copilot):

1. Open *.github/workflows/starter-no-infra_msdocs-core-sql-XYZ* in the explorer. This file was created by the App Service create wizard.
2. Highlight the `dotnet publish` step and select ✨.
3. Ask Copilot, "*Install dotnet ef, then create a migrations bundle in the same output folder.*"
4. If the suggestion is acceptable, select **Accept**. GitHub Copilot doesn't give you the same response every time, and it's not always correct. You might need to ask more questions to fine-tune its response. For tips, see What can I do with GitHub Copilot in my codespace?.



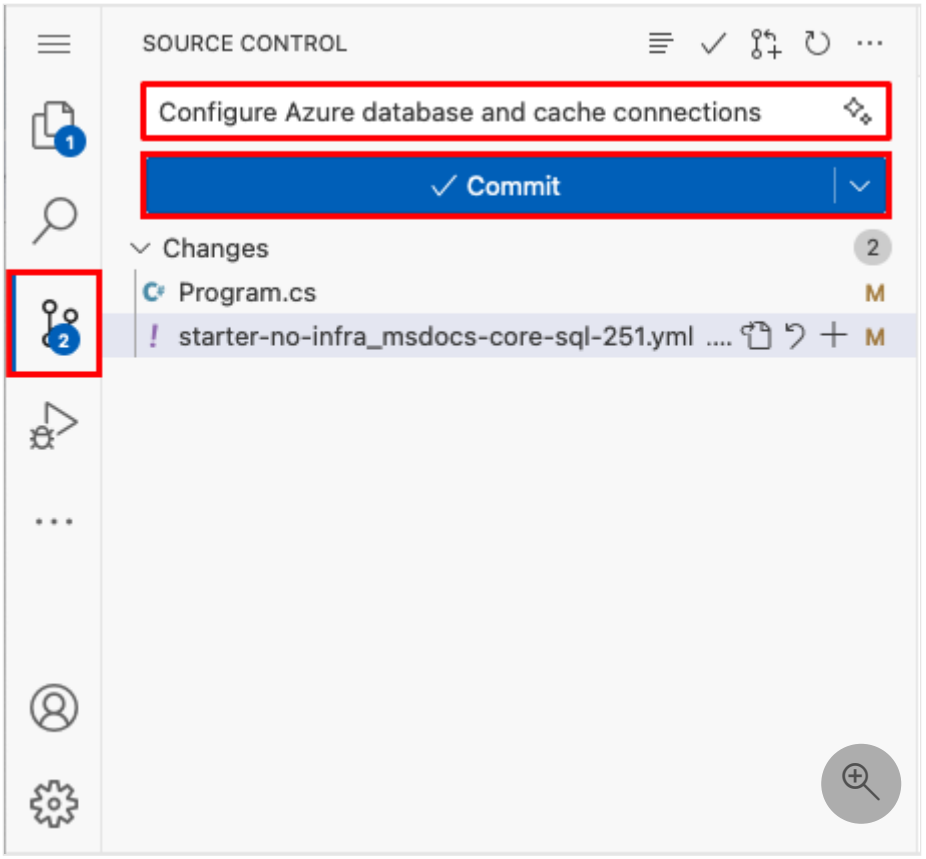
Step 5 (Option 2: without GitHub Copilot):

1. Open *.github/workflows/starter-no-infra_msdocs-core-sql-XYZ* in the explorer. This file was created by the App Service create wizard.
2. Under the `dotnet publish` step, add a step to install the Entity Framework Core tool with the command `dotnet tool install -g dotnet-ef --version 8.*`.
3. Under the new step, add another step to generate a database migration bundle in the deployment package: `dotnet ef migrations bundle --runtime linux-x64 -o ${{env.DOTNET_ROOT}}/myapp/migrationsbundle`. The migration bundle is a self-contained executable that you can run in the production environment without needing the .NET SDK. The App Service linux container only has the .NET runtime and not the .NET SDK.



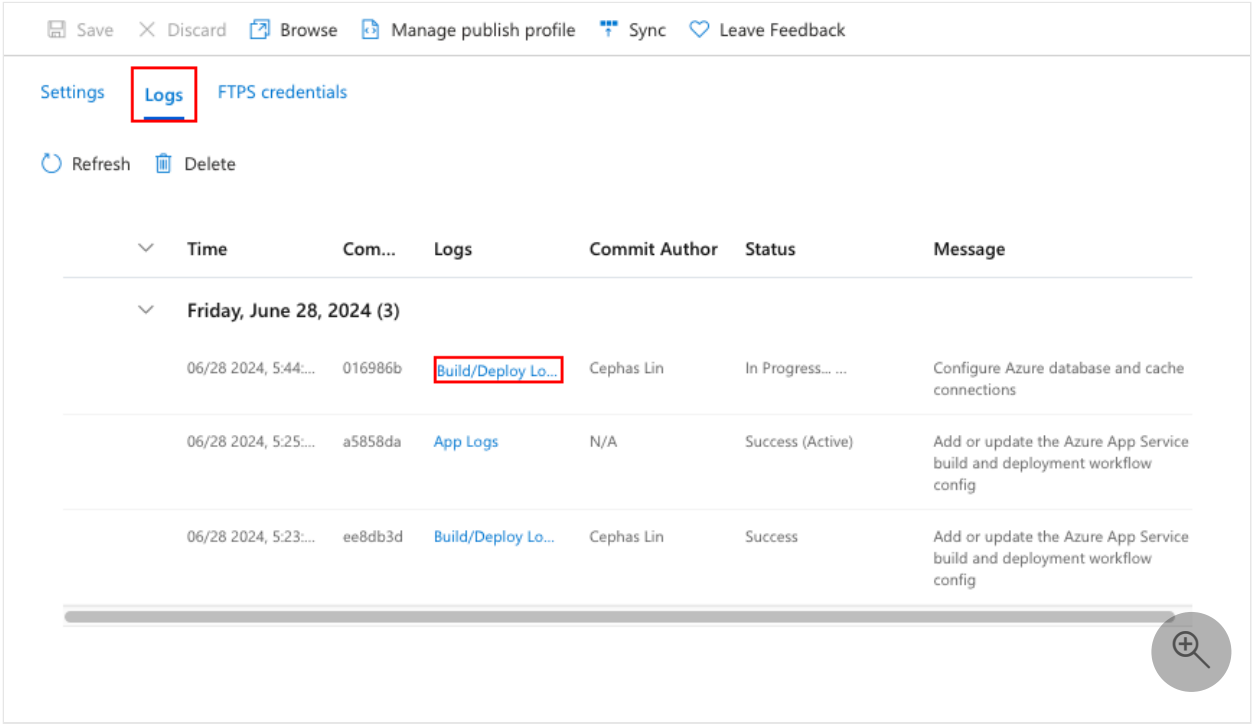
Step 6:

1. Select the **Source Control** extension.
2. In the textbox, type a commit message like `Configure Azure database and cache connections`. Or, select ✨ and let GitHub Copilot generate a commit message for you.
3. Select **Commit**, then confirm with **Yes**.
4. Select **Sync changes 1**, then confirm with **OK**.



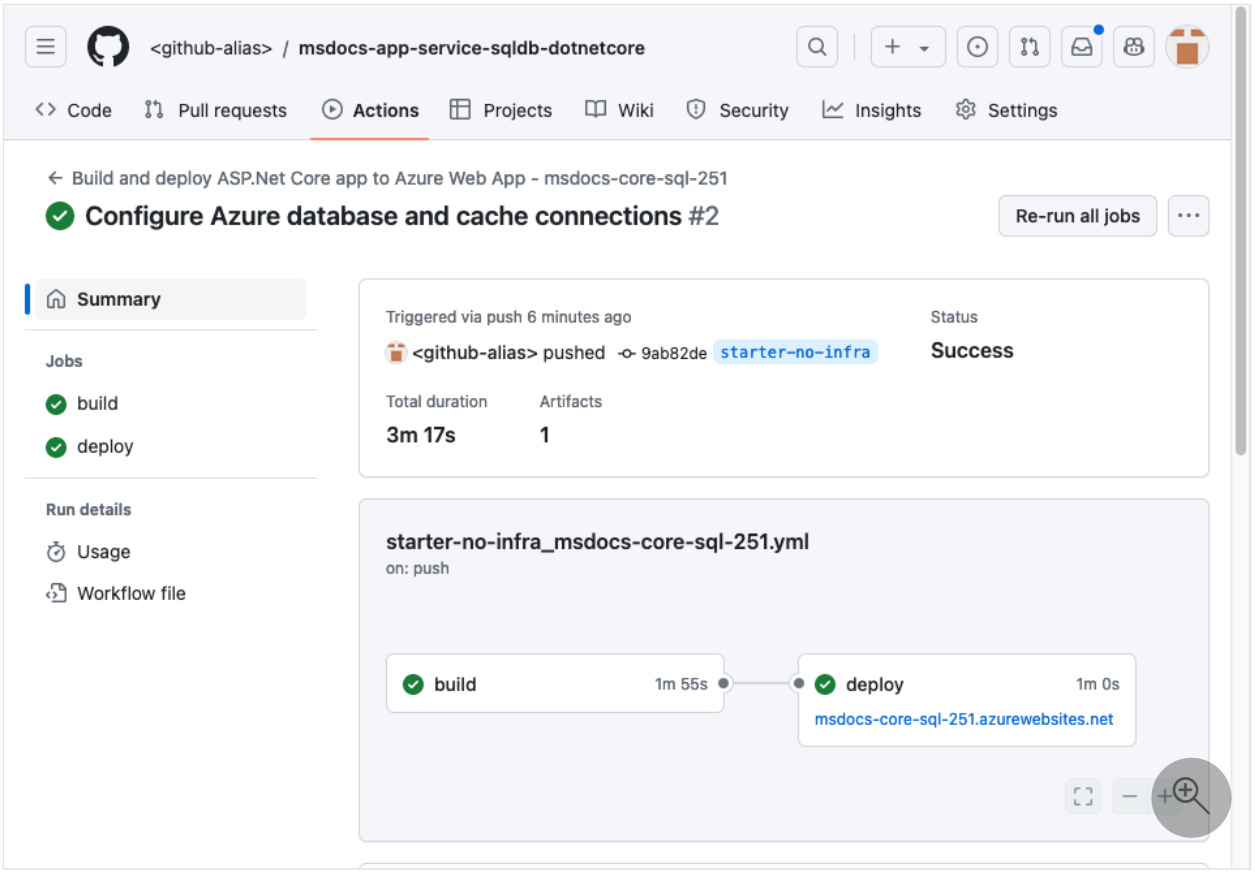
Step 7: Back in the Deployment Center page in the Azure portal:

1. Select the **Logs** tab, then select **Refresh** to see the new deployment run.
2. In the log item for the deployment run, select the **Build/Deploy Logs** entry with the latest timestamp.



Step 8: You're taken to your GitHub repository and see that the GitHub action is running. The workflow file defines two separate stages, build and deploy. Wait for the

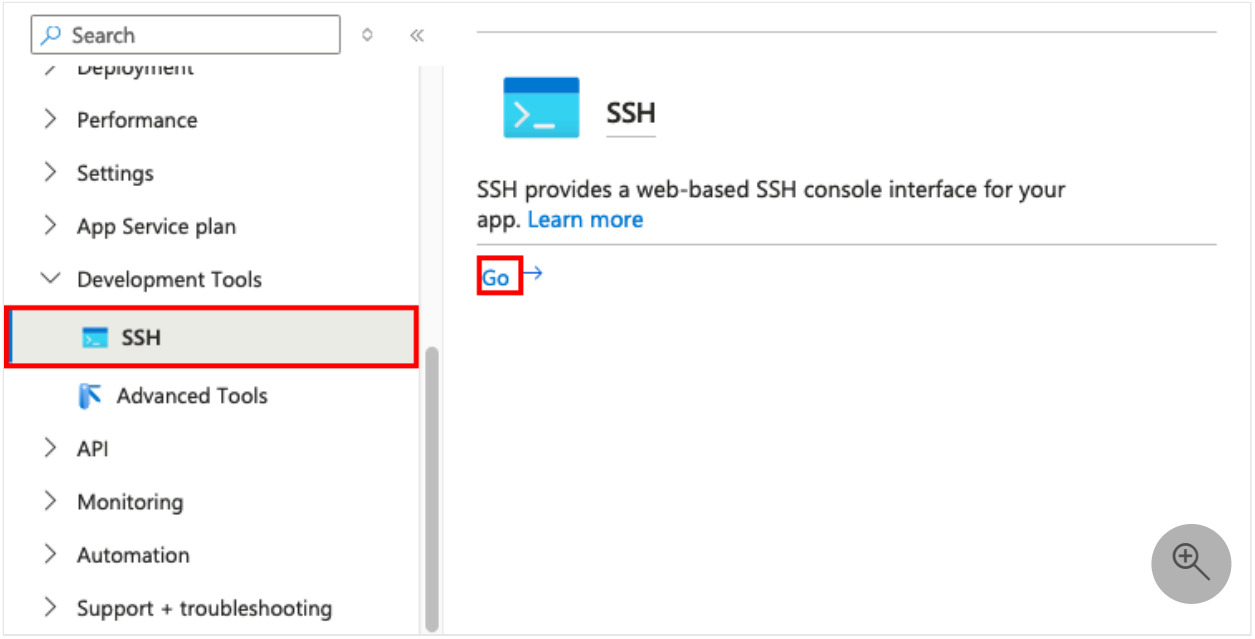
GitHub run to show a status of **Success**. It takes about 5 minutes.



5. Generate database schema

With the SQL Database protected by the virtual network, the easiest way to run dotnet database migrations is in an SSH session with the App Service container.

Step 1: Back in the App Service page, in the left menu, select **Development Tools** > **SSH**, then select **Go**.



Step 2: In the SSH terminal:

1. Run `cd /home/site/wwwroot`. Here are all your deployed files.
2. Run the migration bundle that the GitHub workflow generated, with the command `./migrationsbundle -- --environment Production`. If it succeeds, App Service is connecting successfully to the SQL Database. Remember that `--environment Production` corresponds to the code changes you made in *Program.cs*.

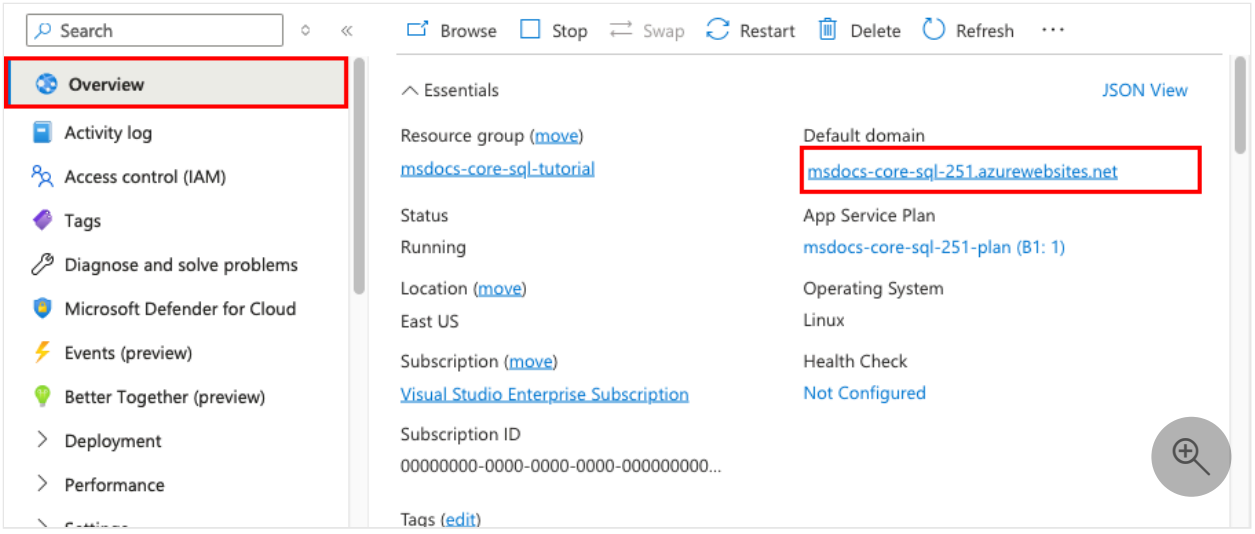
In the SSH session, only changes to files in `/home` can persist beyond app restarts. Changes outside of `/home` aren't persisted.

Having issues? Check the Troubleshooting section.

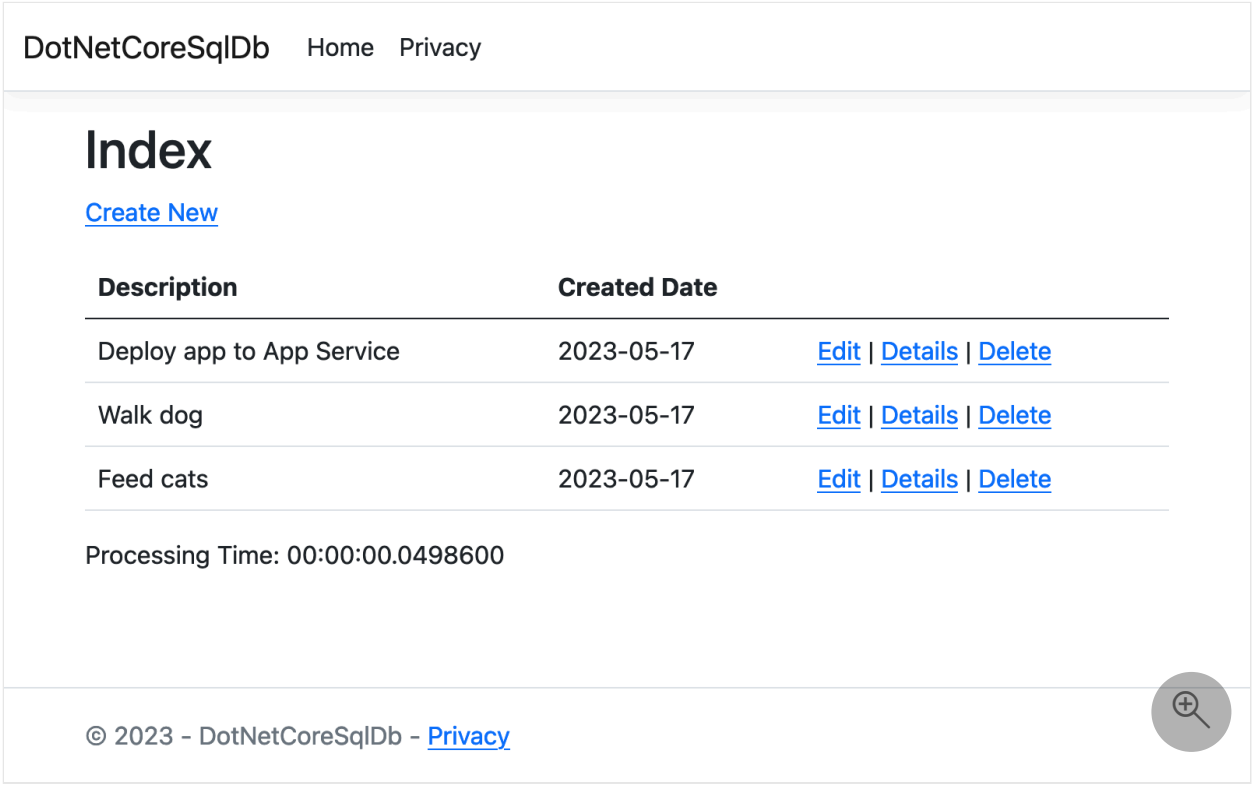
6. Browse to the app

Step 1: In the App Service page:

1. From the left menu, select **Overview**.
2. Select the URL of your app.



Step 2: Add a few tasks to the list. Congratulations, you're running a secure data-driven ASP.NET Core app in Azure App Service.



> **Tip**
>
> The sample application implements the cache-aside pattern. When you visit a data view for the second time, or reload the same page after making data changes,