Cheatsheets / **ASP.NET I**

ASP.NET: Databases

Database Model

Entity Framework uses C# classes to define the database model. This is an in-memory representation of data stored in a database table. Several model classes combine to form the schema for the database.
Each property maps to a column in a database table. The bottom line in the example shows a type of `Continent` which implies a relationship to another table.

```
using System;

public class Country
{
  public string ID { get; set; }
  public string ContinentID { get; set; }
  public string Name { get; set; }
  public int? Population { get; set; }
  public int? Area { get; set; }
  public DateTime? UnitedNationsDate {
get; set; }

  public Continent Continent { get; set; }
}
```

Database Context

The Entity Framework *database context* is a C# class that provides connectivity to an external database for an application. It relies on the `Microsoft.EntityFrameworkCore` library to define the DB context which maps model entities to database tables and columns.

The `DbContextOptions` are injected into the context class via the constructor. The options allow configuration changes per environment so the Development DB is used while coding and testing but the Production DB would be referenced for real work.

The `DbSet` is an in-memory representation of a table or view which has a number of member methods that can return a `List<T>` of records or a single record.

```
using Microsoft.EntityFrameworkCore;

public class CountryContext : DbContext
{
  public
CountryContext(DbContextOptions<CountryCon
text> options)
      : base(options)
  {
  }

  public DbSet<Country> Countries { get;
set; }
  public DbSet<Continent> Continents {
get; set; }

  protected override void
OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Country>
().ToTable("Country");
    modelBuilder.Entity<Continent>
().ToTable("Continent");
  }
}
```

DbSet Type

The Entity Framework type `DbSet` represents a database table in memory. It is typically used with a `<T>` qualifier. The type, or `T`, is one of your database model classes. The `ModelBuilder` binds each database table entity to a corresponding `DbSet`.

`DbSet` has a number of member methods that can return a `List<T>` of records or a single record.

```
using Microsoft.EntityFrameworkCore;

public class CountryContext : DbContext
{
  public
CountryContext(DbContextOptions<CountryCon
text> options)
      : base(options)
  {
  }

  public DbSet<Country> Countries { get;
set; }
  public DbSet<Continent> Continents {
get; set; }

  protected override void
OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Country>
().ToTable("Country");
    modelBuilder.Entity<Continent>
().ToTable("Continent");
  }
}
```

Entity Framework Configuration

In ASP.NET Core, a database may be connected to a web app using Entity Framework. There are four common steps for any setup:

1. Define one or more database model classes and annotate them
2. Define a database context class that uses DbSet to map entities to tables
3. Define a database connection string in **appsettings.json**
4. Add the Entity Framework service in `Startup.ConfigureServices()`

Database Connection String

The Entity Framework context depends on a database connection string that identifies a physical database connection. It is typically stored in **appsettings.json**. You can define multiple connection strings for different environments like Development, Test, or Production. Each database product has specific requirements for the syntax of the connection string. This might contain the database name, user name, password, and other options.

```
{
  "ConnectionStrings": {
    "CountryContext": "Data
Source=Country.db"
  }
}
```

Creating the Schema

Entity Framework provides command-line tools that help manage the connected database. Use these commands in the bash shell or Windows command prompt to create an initial database file and schema. This will read the context class and evaluate each database model represented by a `DbSet`. The SQL syntax necessary to create all schema objects is then generated and executed.

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

Model Binding

In ASP.NET Core, *model binding* is a feature that simplifies capturing and storing data in a web app. The process of model binding includes retrieving data from various sources, converting them to collections of .NET types, and passing them to controllers/page models. Helpers and attributes are used to render HTML with the contents of bound page models. Client- and server-side validation scripts are used to ensure integrity during data entry.

Adding Records

The Entity Framework context `DbSet` member provides the `Add()` and `AddAsync()` methods to insert a new record into the in-memory representation of the corresponding database table. A batch of multiple records can also be added in this fashion.

The record is passed from the browser in the `<form>` post back. In this case a `Country` member is declared with a `[BindProperty]` attribute so the entire record is passed back to the server.

Use the EF context `SaveChanges()` or `SaveChangesAsync()` methods to persist all new records to the database table.

```
// Assuming Country is of type Country
// Assuming _context is of a type
inheriting DbSet

public async Task<IActionResult>
OnPostAsync(string id)
{
  if (!ModelState.IsValid)
  {
    return Page();
  }

  await
_context.Countries.AddAsync(Country);

  await _context.SaveChangesAsync();

  return RedirectToPage("./Index");
}
```

Saving Changes

The Entity Framework context `DbSet` member provides the `Attach()` method to update an existing record, the `Add()` method to insert a new record, and the `Remove()` method to delete an existing record. Any combination of multiple records can batched before saving.

Use the EF context `SaveChanges()` or `SaveChangesAsync()` methods to persist all inserted, updated, and deleted records to the database table.

```
// Assuming Country is of type Country
// Assuming _context is of a type
inheriting DbSet

public async Task<IActionResult>
OnPostAsync(string id)
{
  // update
  _context.Attach(Country).State =
EntityState.Modified;

  // insert
  await
_context.Countries.AddAsync(Country);

  // delete
  Country Country = await
_context.Countries.FindAsync(id);

  if (Country != null)
  {
    _context.Countries.Remove(Country);
  }

  // all three methods must be followed by
savechanges
  await _context.SaveChangesAsync();

  return RedirectToPage("./Index");
}
```

Finding Records

The Entity Framework context `DbSet` member provides the `Find()` and `FindAsync()` methods to retrieve an existing record from the in-memory representation of the database table. Assign the result of this method to a local member in the page model.
This method generates the appropriate SQL syntax needed to access the record in the database table.

```
// Assuming Country is of type Country
// Assuming _context is of a type
inheriting DbSet

public async Task<IActionResult>
OnGetAsync(string id)
{
  if (id == null)
  {
    return NotFound();
  }

  Country Country = await
_context.Countries.FindAsync(id);

  return Page();
}
```

Deleting Records

The Entity Framework context `DbSet` member provides the `Remove()` method to delete an existing record from the in-memory representation of the database table. Any combination of multiple record deletions can be batched before saving.

Use the EF context `SaveChanges()` or `SaveChangesAsync()` methods to persist all deletions to the database table.

```
// Assuming Country is of type Country
// Assuming _context is of a type
inheriting DbSet

public async Task<IActionResult>
OnPostAsync(string id)
{
  if (id == null)
  {
    return NotFound();
  }

  Country Country = await
_context.Countries.FindAsync(id);

  if (Country != null)
  {
    _context.Countries.Remove(Country);
  }

  await _context.SaveChangesAsync();

  return RedirectToPage("./Index");
}
```

Updating Records

The Entity Framework context `DbSet` member provides the `Attach()` method to update an existing record in the in-memory representation of the corresponding database table. A batch of multiple records can also be updated in this fashion.

The record is passed from the browser in the `<form>` post back. In this case a `Country` member is declared with a `[BindProperty]` attribute so the entire record is passed back to the server.

Use the EF context `SaveChanges()` or `SaveChangesAsync()` methods to persist all updated records to the database table.

```
// Assuming Country is of type Country
// Assuming _context is of a type
inheriting DbSet

public async Task<IActionResult>
OnPostAsync(string id)
{
  if (!ModelState.IsValid)
  {
    return Page();
  }

  _context.Attach(Country).State =
EntityState.Modified;

  await _context.SaveChangesAsync();

  return RedirectToPage("./Index");
}
```

Valid Model State

Entity Framework database models accept annotations that drive data validation at the property level. If you are using the `asp-validation-for` or `asp-validation-summary` HTML attributes, validation is handled client-side with JavaScript. The model is validated and the `<form>` post back won't occur until that model is valid.

Sometimes the client-side validation will not be available so it is considered best practice to also validate the model server-side, inside the `OnPostAsync()` method. This example checks for `ModelState.IsValid` and returns the same page if it is false. This effectively keeps the user on the same page until their entries are valid.

If the model is valid, the insert, update, or delete can proceed followed by `SaveChangesAsync()` to persist the changes.

```
public async Task<IActionResult>
OnPostAsync()
{
  if (!ModelState.IsValid)
  {
    return Page();
  }

  _context.Continents.Add(Continent);

  await _context.SaveChangesAsync();

  return RedirectToPage("./Index");
}
```

Validation Attribute

The `asp-for` attribute in an `<input>` element will render HTML and JavaScript that handle the display and data entry for a field based on the model annotations. The JavaScript will set the valid flag on the field.

The `asp-validation-for` attribute in a `<span>` element will display any error message generated when the property annotations are not valid.

In this example, the `<span>` be rendered as this HTML:

```
<span class="field-validation-valid" data-valmsg-
for="Continent.Name" data-valmsg-replace="true"></span>
```

```
<div>
  <label asp-for="Continent.Name"></label>
  <div>
    <input asp-for="Continent.Name" />
    <span asp-validation-
for="Continent.Name"></span>
  </div>
</div>
```

[Display] Attribute

The `[Display]` attribute specifies the caption for a label, textbox, or table heading.
Within a Razor Page, the `@Html.DisplayForName()` helper defaults to the property name unless the `[Display]` attribute overrides it. In this case, `Continent` is displayed instead of the more technical `ContinentID` .

```
using
System.ComponentModel.DataAnnotations;

public class Country
{
  [Display(Name = "Continent")]
  public string ContinentID { get; set; }
}
```

[DisplayFormat] Attribute

The `[DisplayFormat]` attribute can explicitly apply a C# format string. The optional `ApplyFormatInEditMode` means the format should also apply in edit mode.
`[DisplayFormat]` is often used in combination with the `[DataType]` attribute. Together they determine the rendered HTML when using the `@Html.DisplayFor()` helper.

```
using
System.ComponentModel.DataAnnotations;

public class Country
{
  [DisplayFormat(DataFormatString = "
{0:N0}", ApplyFormatInEditMode = true)]
  public int? Population { get; set; }
}
```

[DataType] Attribute

The `[DataType]` attribute specifies a more specific data type than the database column type. In this case, the database table will use a `DateTime` column but the render logic will only show the date.
The `@Html.DisplayFor()` helper knows about types and will render a default format to match the type. In this case, the HTML5 browser date picker will appear when editing the field.

```
using
System.ComponentModel.DataAnnotations;

public class Country
{
  [DataType(DataType.Date)]
  [DisplayFormat(DataFormatString = "
{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
  public DateTime? UnitedNationsDate {
get; set; }
}
```

[Required] Attribute

The `[Required]` attribute can be applied to one or more properties in a database model class. EF will create a `NOT NULL` column in the database table for the property.
The client-side JavaScript validation scripts will ensure that a non-empty string or number is valid before posting the record from the browser on inserts and updates.

```
using
System.ComponentModel.DataAnnotations;

public class Country
{
  [Required]
  public string Name { get; set; }
}
```

[RegularExpression] Attribute

The `[RegularExpression]` attribute can apply detailed restrictions for data input. The match expression is evaluated during data entry and the result returns true or false. If false, the model state will not be valid and the optional `ErrorMessage` will display.
In a Razor page, the `@Html.DisplayFor()` helper only shows the data in the field. The `asp-validation-for` attribute on a `<span>` tag displays the `ErrorMessage`.

```
using
System.ComponentModel.DataAnnotations;

public class Country
{
  [RegularExpression(@"[A-Z]+",
ErrorMessage = "Only upper case characters
are allowed.")]
  public string CountryCode { get; set; }
}
```

[StringLength] Attribute

The `[StringLength]` attribute specifies the maximum length of characters that are allowed in a data field and optionally the minimum length. The model will not be flagged as valid if these restrictions are exceeded. In this case, the `ContinentID` must be exactly 2 characters in length.
In a Razor Page, the `@Html.DisplayFor()` helper only shows the data in the field. The client-side JavaScript validation scripts use the `asp-validation-for` attribute on a `<span>` tag to display a default error message.

```
using
System.ComponentModel.DataAnnotations;

public class Country
{
  [StringLength(2, MinimumLength = 2)]
  public string ContinentCode { get; set;
}
}
```

[Range] Attribute

The `[Range]` attribute specifies the minimum and maximum values in a data field. The model will not be flagged as valid if these restrictions are exceeded. In this case, `Population` must be greater than 0 and less than the big number!
In a Razor page, the `@Html.DisplayFor()` helper only shows the data in the field. The client-side JavaScript validation scripts use the `asp-validation-for` attribute on a `<span>` tag to display a default error message.

```
using
System.ComponentModel.DataAnnotations;

public class Country
{
  [Range(1, 10000000000)]
  public int? Population { get; set; }
}
```

Select List Items Attribute

```
<select asp-for="Country.ContinentID" asp-
items="Model.Continents"></select>
```

The `asp-items` attribute in a `<select>` element generates `<option>` tags according to the model property specified. It works in conjunction with the `asp-for` attribute to display the matching option and set the underlying value on a change.

```
using Microsoft.AspNetCore.Mvc.Rendering;

public SelectList Continents { get; set; }

public async Task<IActionResult> OnGetAsync(string id)
{
  Continents = new SelectList(_context.Continents,
nameof(Continent.ID), nameof(Continent.Name));
}
```

The `SelectList` type is declared in the page model code and assigned to a `new SelectList()` where each record in `Continents` grabs the `ID` as the `<option>` value and `Name` as the `<option>` display text.

The included `<select>` in the example would render as this HTML:

```
<select class="valid" id="Country_ContinentID"
name="Country.ContinentID" aria-invalid="false">
  <option value="NA">North America</option>
  <option value="SA">South America</option>
  <option value="EU">Europe</option>
  <!-- etc -->
</select>
```

LINQ Queries

The Entity Framework `DbSet` entities can manage complex queries using C# LINQ syntax. This is referenced from the `System.Linq` library.
All of the `Where()` and `OrderBy()` clauses are evaluated in the final statement that calls `ToListAsync()`. EF evaluates all options and generates a SQL `SELECT` statement with corresponding `WHERE` and `ORDERBY` clauses.

```
using System.Linq;

var countries = from c in
_context.Countries
          select c;

countries = countries.Where(c =>
c.Name.Contains("Russia"));

countries = countries.Where(c =>
c.ContinentID == 3);

countries = countries.OrderBy(c =>
c.Name);

List<Country> Countries = await
countries.ToListAsync();
```

DisplayNameFor Helper

The `@Html.DisplayNameFor()` tag helper is used to display the friendly name for a property in a database model. By default, this will match the property name. If a `[Display(Name = "Code")]` annotation is applied to the property in the model class, that string is used instead

```
// Example database model
public class Continent
{
  [Display(Name = "Code")]
  public int ID { get; set; }
}
```

```
<!-- In .cshtml file -->
<div>
  @Html.DisplayNameFor(model =>
model.Continent.ID)
</div>

<!-- Rendered HTML in browser -->
<div>
  Code
</div>
```

Print Share