* Reasons to use a Data Server
  + Note that Angular does include an in-memory Web API package that can be used in place of a real data server
  + This replaces the HttpClient’s HttpBackend and emulates CRUD operations over a RESTful API by intercepting and redirecting Angular HTTP requests
  + But, the data sever will let us see what’s going on in the backend, which is the point of this book
* The Data Model
  + We will go about creating a data model to represent our city data so we can store, retrieve, and manipulate data through the front-end
  + Introducing Entity Framework Core
    - We are going to use Entity Framework Core (EF Core) as the Object Relational Mapper (ORM) for ADO.NET
  + The SQL Server Data Provider
    - Note one of the packages we will need (Microsoft.EntityFrameworkCore.SqlServer) is the MS SQL Database Provider for EF Core
    - This provides an interface for the entire MS SQL Server database family
  + Data Modeling Approaches
    - Model-First
      * This approach uses some visual designer to create the model; this could be e.g. XML-based DataSet Schema (XSD) or the Entity Designer Model XML visual interface (EDMX)
      * Once the model is created EF will create the domain classes and the necessary SQL script
      * This is nice for large data structures because it generates the DB schema and the class diagram as a whole
      * It also allows us to use the model to issue updates when the DB changes
      * But, there can be data loss in the case of updates due to the auto-generated SQL scripts
      * Also, precise control over your model using the design tools can be difficult
    - Database-First
      * This is basically the opposite of Model-First
      * Here, we build the SQL scripts that are used to create the DB first, then we provide this to EF to generate the model and classes accordingly
      * This is obviously good if you have an already existing DB that has data in it
      * Also, you have fine control over the DB model and can avoid data loss in the case of updates
      * But, updating the DB directly can be tricky when you need to deal with multiple DB instances (e.g., dev, test, prod)
      * Also, you won’t have control over the model classes that are auto-generated by EF (you will need to be very familiar with EF conventions to work with these classes)
    - Code-First
      * This is EF Core’s flagship approach in all the recent versions
      * Here we define the model in standard classes without the need for a design tool, XML mappings, or autogenerated code
      * We give the classes to EF and it will generate the DB accordingly
      * This also makes use of a fluent API that allows convention over configuration to handle most common scenarios, while still allowing for custom attribute-based implementations
      * Good knowledge of C# and EF are required here though; and there is still a possibility for data loss during migrations
* Creating the Entities
  + Defining the Entities
    - An entity here is a class that maps to a specific DB table
    - This allows us to work with data in a strongly-typed OO fashion using properties to access specific data columns
    - The DBMS Server (e.g., SQL Server hosted in Azure) hosts our database; the ORM Framework (EF Core here) takes the tables from this DB and transforms them into C# classes that we can use in the backend
    - We will create entities for City and Country
    - The City will have a foreign key reference to the Country that it belongs to
    - For this we will use several data annotations
      * Required
      * Key
      * ForeignKey
    - Note that the **binding** enforced by the ForeignKey attribute will be enforced via a constraint (as long as the DB supports it)
  + Defining Relationships
    - For our dataset, we will have a one-to-many relationship between the country and cities
    - We will have a country property in the city entity to identify the parent country for the city
    - We will have a cities property in the country to identify all the child cities that are in the country
    - Note that these properties will be **virtual** to allow for **lazy loading**
    - Also note that we only need a foreign key in the child entity so that we can point to the parent (we don’t need an FK/FK-list in the parent)
  + Connecting to SQL Server
    - We create a SQL database (a PaaS offering) in Azure
    - This requires creating a new SQL Server instance, which we then connect to via our management tool of choice (e.g. Azure Data Studio)
* Creating the Database using Code-First
  + Now we have our entity classes, we have a DBMS instance with a database created, but we have not gone through all the steps to create and fill in the DB
  + To do this we need to still
    - Set up a Database Context
    - Enable code-first migration support in our project
  + Setting up the DbContext
    - The Microsoft.EntityFrameworkCore.DbContext class is responsible for enabling us to interact with data as objects
    - This includes populating the objects with data from the database, keeping track of changes, and persisting them to the DB during CRUD operations
    - We need to create our own custom DbContext class for our project; this is our own class inheriting from DbContext
    - This includes adding DbSet<TEntity> properties to the child class for the objects that we want to interact with in the DB
    - We also override the OnModelCreating() method; we use the base implementation, then explicitly map our entity classes to specific tables in our DB
  + Database Initialization Strategies
    - We need to determine how we want to keep track of changes that we will make to our data model
    - Previously in EF, you needed to choose one management pattern (known as database initializers or DbInitializers) for code-first
    - These were pretty complicated though, and required extensive knowledge of EF
    - This pattern has now been streamlined in EF Core; there are no DbInitializers and no automatic migrations
    - Database initialization is now handled almost exclusively by PowerShell commands
* Creating the Database
  + Updating the appsettings.json File
    - We need to add the connection strings to our appsettings to use in the startup process
    - We then add a DbContext to our available services via Services.AddDbContext<TDbContext>(options) to add our custom DbContext
    - The options provided specify that we are going to use a SqlServer and then set the connection string from the configuration
* Adding the Initial Migration
  + To do this we first need to globally install the dotnet-ef dotnet tooling
  + We then go into our project and run **dotnet ef migrations add “Initial”** to create an initial migration to create the database tables
* Updating the Database
  + This basically means creating/updating the database to sync its contents (tables, constraints, etc.) with the rules/patterns defined in the DbContext and the data annotations in the Entity classes
  + The first migration creates the DB from scratch, subsequent ones will update the DB accordingly
  + To do this with the dotnet tooling, we run **dotnet ef database update**
  + Understanding Migrations
    - Data models can be expected to change several times after their initial development for a variety of reasons
    - When we do this the data model will become out-of-sync with the code-first database
    - This is typically fine for dev environments since here we can typically drop and recreate the entire database
    - This obviously is not an option for production; and these migrations are meant to solve this issue and allow you to change the data model in the DB
  + Is Data Migration Required?
    - Data migration can be very useful, but also very complicated to understand and to implement
    - So, if we decide we don’t want to use them (either at the beginning, or later on) we can switch to a database-first approach
    - So we can start to manually design, create, and/or modify the tables; EF will continue to work as long as entity class property types match to DB fields 100%
* Populating the Database
  + We can now work to populate the database with the data that we grabbed from the public data source
  + We need to implement one of EF Core’s data seeding strategies
    - Model data seed
    - Manual migration customization
    - Custom initialization logic
  + We will use the custom initialization logic here with a controller that we can use whenever we need to seed the DB
* Implementing the other Controllers
  + We add a SeedController to seed our DB with data from the Excel worksheet
  + We add controllers for City and Country entities using the **dotnet-aspnet-codegenerator** dotnet tooling