Plan

The function app is going to be the api for my hiking platform.

Step-by-step plan (not in order):

- Add some CRUD endpoints for users/hikes/mountains. Then some to add records for the hikes a user has done and ratings.
- Add blob storage functionality to upload photos for hikes.
- Get azurite and a db running in docker.
- Create a C# solution with a function app project, see https://learn.microsoft.com/en-us/azure/azure-functions/how-to-create-function-vs-code?pivot=progra... for more information
- Define a Mountain model with the following required properties:
    - Id
    - Name
    - Height
    - Location
- Create a mountains.json file with an array of at least 5 mountain objects
- Create a static class with a static method that reads the mountains.json file and returns a list of mountains
- Create a mountain service class that calls the static class to return the mountains as a list
- Define a mountains function (http) endpoint that calls the mountain service to return the list of mountains
- Add an xunit test project to the solution and write unit tests for the code written.
- Add a README that documents how to run the project and the unit tests.
- Use Postman to query your mountains endpoint Controller-Service-Repository. I've had a unique opportunity fall into... | by Tom Collings | Medium
- Add dependency injection to register and resolve (mountain) service(s). https://learn.microsoft.com/en-us/azure/azure-functions/functions-dotnet-dependency-injection
- Add a database next (Azure SQL)
    - Get Started with SQL Database Projects - SQL Server | Microsoft Learn
    - Use SSMS to write SQL queries and make a mountains table to match the object.
    - Install SQL Server Management Studio | Microsoft Learn
    - Lesson 1: Connecting to the Database Engine - SQL Server | Microsoft Learn

Notes

Dependency Injection (DI) is a **design pattern** in software engineering where an object receives the objects it depends on (its *dependencies*) from an external source, rather than creating them itself.

- Instead of a class saying *"I'll build my own Email Service"*, DI lets the class say *"I need something that can send emails"* and the system provides it.
- This approach **decouples** classes from concrete implementations, making code more modular, testable, and maintainable.

Why use DI?

- **Loose Coupling**: Classes depend on abstractions (interfaces), not concrete implementations.
- **Flexibility**: Swap implementations without changing the consuming class.
- **Testability**: Easily inject mocks or stubs for unit tests.
- **Maintainability**: Changes in dependencies don't ripple through the codebase.
- **Supports SOLID principles**, especially the Dependency Inversion Principle.

There are three common types:

1. **Constructor Injection** – Pass dependencies via the constructor (most common and recommended).
2. **Setter Injection** – Use public setters to inject dependencies.
3. **Interface Injection** – The dependency provides a method to inject itself into the client.

SSMS (SQL Server Management Studio) is a GUI tool for connecting to SQL Server or Azure SQL Database.

It lets you:

- **Dock into the database** (connect using your connection string).
- **Write and execute SQL queries directly** (e.g., CREATE TABLE, INSERT, SELECT).
- Inspect tables, run queries, and manage schema without deploying from VS Code.

What Does SOLID Stand For?

1. **S – Single Responsibility Principle (SRP)**
   A class should have **only one reason to change**, meaning it should focus on a single responsibility or purpose. This reduces coupling and makes the code easier to maintain.
   Example: A class that handles user authentication should not also manage database operations.

2. **O – Open/Closed Principle (OCP)**
   Software entities (classes, modules, functions) should be **open for extension but closed for modification**. You should be able to add new functionality without altering existing code.
   Example: Using inheritance or interfaces to add new behaviors without changing the original class.

3. **L – Liskov Substitution Principle (LSP)**
   Objects of a superclass should be replaceable with objects of its subclasses **without breaking the application**. This ensures correct behavior when using polymorphism.

4. **I – Interface Segregation Principle (ISP)**
   Clients should not be forced to depend on interfaces they do not use. Instead of one large interface, create smaller, more specific ones.
   Example: Separate interfaces for reading and writing operations instead of one big interface with both.

5. **D – Dependency Inversion Principle (DIP)**
   High-level modules should not depend on low-level modules; both should depend on **abstractions**. This promotes loose coupling and easier testing.
   Example: Use dependency injection to provide implementations at runtime rather than hardcoding them.