API Best Practices

Use JSON for sending and receiving data

Use noun instead of verbs in endpoints. **EX:** https://mysite.com/getPosts should be https://mysite.com/posts

Collections should use plural nouns. **EX:** https://mysite.com/post/123 should be https://mysite.com/posts/123

Use status codes for error handling

Use nesting on endpoints to show relationships. **EX:** https://mysite.com/posts/author

Use filtering, sorting, and pagination so retrieve data requested. **EX:** https://mysite.com/posts?tags=javascript

Use SSL for Security. **EX:** https://mysite.com/posts runs on SSL and http://mysite.com/posts does not run on SSL

Be Clear with Versioning. Common versioning system in semantic versioning. **EX:** 1.2.3 where 1 is the major version, 2 is the minor version, and 3 is the patch version. https://mysite.com/v1/

Provide accurate API documentation. **EX:** Swagger, Postman

HTTP Headers

HTTP headers let the client and the server pass additional information with an HTTP request or response.

Request headers contain more information about the resource to be fetched, or about the client requesting the resource.

Response headers hold additional information about the response, like its location or about the server providing it

Representation headers contain information about the body of the resource, like its MIME type, or encoding/compression applied.

Payload headers contain representation--independent information about payload data, including content length and the encoding used for transport.

Design Patterns - Basics

Types of Design Patterns	Creational, Structural, Behavioral

SOLID Principles

Single Responsibility Principle	A class changes for only one reason
Open/Closed Principle	A class should be open for extension, closed for editing
Liskov's Substitution Principle	Derived types should cleanly and easily replace base types
Interface Segregation Principle	Favor multiple single-purpose interfaces over composite
Dependency Inversion Principle	Concrete classes depend on abstractions, not vice-versa

What are design patterns?

Design patterns are solutions to software design problems you find again and again in real-world application development. Patterns are about reusable designs and interactions of objects.

Design Patterns - Singleton

Singleton is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code. Singleton has almost the same pros and cons a

Advantages: Singleton pattern can implement interfaces. Can be lazy-loaded and has Static Initialization. It helps to hide dependencies. It provides a single point of access to a particular instance, so it is easy to maintain.

Disadvantages: Unit testing is a bit difficult as it introduces a global state into an application. Reduces the potential for parallelism within a program by locking.

Design Patterns - Creational

Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation so that the same construction processes can create different representations.
Factory Method	Define an interface for creating an object, but let the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Singleton	Ensure a class only has one instance, and provide a global point of access to it.

Design Patterns - Structural

Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatibility interfaces.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Design Patterns - Structural (cont)

Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a system. Façade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context; it's indistinguishable from an instance of the object that's not shared.
Proxy	Provide a surrogate or placeholder for another object to control access to it.

Design Patterns - Factory Method

The **Factory Method** is one of the most known Design Patterns and often used when creating things with same behavior, but with different specifications.

EX: Things with same behaviour, but different specifications: Animal Factory, Logistics Factory (delivery method or transport vehicle)

Classes to be created cannot be determined in advance, therefore an abstract interface is provided which can be reached via a function

A class expects from its subclasses a specification of the products to be created

Specialization options for subclasses, as this pattern provides an extended version of the object compared to the direct creation of the object

Design Patterns - Behavioral

Chain of Resp.	Avoid coupling the sender of a request to its receiver by giving more then one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Design Patterns - Behavioral (cont)

Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients who use it.
Template Method	Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Dependency Injection Pattern

Dependency Injection (DI) is a design pattern used to implement **IoC**. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

Dependency Injection Pattern (cont)

The Dependency Injection pattern involves 3 types of classes:

Client Class: The client class (dependent class) is a class which depends on the service class.

Service Class: The service class (dependency) is a class that provides service to the client class.

Injector Class: The injector class injects the service class object into the client class.

Repository Design Pattern

The Repository Design Pattern in C# Mediates between the domain and the data mapping layers using a collection-like interface for accessing the domain objects. This isolates the data access code from the rest of the application.

Advantages: Changes can be done in one place. Testing the controller is easier as you don't have to test against the database.