1. C# Questions

1.1 Explain the use of `ref` and `out` keywords in C# method parameters. When should each be used?

- **Detailed Answer**: The `ref` and `out` keywords in C# allow methods to modify parameters passed to them, with each serving a different purpose.
  - `ref` : When a parameter is passed with the `ref` keyword, it allows both the caller and method to access and modify an existing value. The parameter must be initialized before being passed to the method, as `ref` signifies that the method might read and modify the existing value.
  - `out` : The `out` keyword allows a method to return multiple values through its parameters. Unlike `ref` , `out` parameters don't need to be initialized by the caller before passing to the method, but the method must assign a value before it completes execution.
  - **Use Cases**: Use `ref` when a value needs to be updated and returned to the caller. Use `out` when returning multiple values from a method is required.

1.2 How would you load data partially from a database to display on a screen in a C# application?

- **Detailed Answer**: To load data partially (or implement pagination), use `Skip` and `Take` methods in LINQ to SQL or Entity Framework queries. This approach allows for loading a manageable subset of data at a time, which is especially helpful with large datasets.
  - **Example**: If displaying data page by page, use `dbContext.Records.Skip(pageNumber * pageSize).Take(pageSize)` to fetch only the required page of data.
  - **Considerations**: Implementing efficient pagination is essential to reduce memory and CPU load, ensuring faster response times and a better user experience.

1.3 A developer needs to implement only 9 out of 10 methods in an interface due to project needs. How can this be resolved while adhering to SOLID principles?

- **Detailed Answer**: This scenario points to the Interface Segregation Principle (ISP) of SOLID, which suggests that interfaces should be small and specific to the client's needs rather than large and general.
  - **Solution**: The developer can refactor the interface into smaller, more specific interfaces, each with only the required methods. For example, instead of an `IShape` interface with all methods, smaller interfaces like `IDrawable` and `ITransformable` can be created.
  - **Alternate Solution**: If the interface is from a third-party library and cannot be refactored, the developer may implement the interface and provide minimal or placeholder implementations (e.g., throwing `NotImplementedException`) for unnecessary methods, though this isn't ideal.

1.4 The code throws an `OutOfMemoryException` when reading data from a database in batches into a .NET List. How can this be resolved?

- **Detailed Answer**: To avoid `OutOfMemoryException` , especially when working with large datasets, it's best to use a streaming or lazy-loading approach.
  - **Streaming**: Instead of loading all data into a list at once, process items as they are retrieved, such as by using `yield return` or returning an `IEnumerable` collection.
  - **Batch Processing**: Process data in smaller batches that can be released from memory after processing, avoiding the accumulation of data in memory. Additionally, if Entity Framework is used, consider disabling tracking (`AsNoTracking`) to reduce memory usage.

1.5 How can you add a method to a sealed class from a third-party library to convert its data to a specific format?

- **Detailed Answer**: Since sealed classes cannot be inherited, extension methods provide a practical way to add functionality. Extension methods allow adding static methods to existing sealed classes without modifying the source.
  - **Implementation**: Define an extension method in a static class that operates on the sealed class type. For example, `public static string ToFormattedString(this ThirdPartyClass obj) { /* conversion logic */ }` .

- **Advantages**: Extension methods are non-intrusive, enabling you to add functionality without access to the original code.

---

2. .NET Core

2.1 What are the common approaches for implementing Web API versioning in .NET Core?

- **Detailed Answer**: API versioning helps manage different versions of an API to avoid breaking changes for existing clients.
  - **Approaches**:
    - **Query String Versioning**: Add version information as a query parameter, e.g., `api/products?version=1` .
    - **URL Path Versioning**: Version is added in the URL path, e.g., `api/v1/products` .
    - **Header Versioning**: The client specifies the version in a custom header.
    - **Media Type Versioning**: Use Accept headers with custom media types to specify the version.
  - **Tooling**: The `Microsoft.AspNetCore.Mvc.Versioning` package simplifies versioning in .NET Core by allowing easy configuration and management of versioned routes.

2.2 What steps should be taken to troubleshoot a 500 Internal Server Error in a .NET Core application?

- **Detailed Answer**:
  - **Enable Developer Exception Page**: In development, enable `app.UseDeveloperExceptionPage()` in `Startup.cs` to view detailed error messages.
  - **Check Logs**: Inspect logs in the configured logging provider (e.g., files, database) to locate detailed error information.
  - **Error Handling Middleware**: Add global error-handling middleware to catch and log exceptions before they return to the client.
  - **Dependency and Configuration Check**: Verify that all required services, configurations, and dependencies are correctly set up.

2.3 How can you configure multiple logging sources (file, Splunk, database) in a .NET Core application?

- **Detailed Answer**: .NET Core's built-in logging framework supports multiple providers, which can be configured in `Startup.cs` or `appsettings.json` .
  - **Setup**: In `ConfigureServices` , use `AddLogging` to add providers like `AddFile` , `AddSplunk` , and `AddDatabase` . Each provider can be customized with its own configuration in `appsettings.json` .
  - **Example**: Add `"Logging": { "LogLevel": { "Default": "Warning" }, "File": { ... }, "Splunk": { ... } }` to `appsettings.json` to configure different providers.

2.4 For a class with 100 methods used across multiple controller actions, what dependency injection lifetime scope would you choose?

- **Detailed Answer**: The choice depends on the intended usage and memory footprint of the class.
  - **Scoped Lifetime**: Creates a new instance per HTTP request, ensuring isolation across requests. Suitable if data/state shouldn't persist across requests.
  - **Singleton Lifetime**: Provides a single instance for the application's lifecycle, useful if the state needs to be shared across requests. However, avoid this for large objects with high memory usage.

2.5 How would you configure different data stores for different tenants in a .NET Core application?

- **Detailed Answer**:
  - **Tenant Identification**: Identify the tenant based on criteria such as domain name, headers, or API tokens.

- **Custom Configuration Provider**: Implement a custom configuration provider that loads connection strings and settings for each tenant.
- **Multi-Tenancy Libraries**: Some libraries, like `Finbuckle.MultiTenant`, provide pre-built solutions for managing multi-tenant data stores.

---

3. **Web API**

3.1 What HTTP status code is appropriate for a long-running Web API call?

- **Detailed Answer**: Use a `202 Accepted` status code for long-running API calls to inform the client that the request is being processed asynchronously. Follow up with status update endpoints to provide the client with progress or completion updates.

3.2 How can you apply conditional validation in a Web API, where validation depends on another property's value?

- **Detailed Answer**:
  - **Custom Validation Attribute**: Write a custom validation attribute that checks the value of another property to apply conditional logic.
  - **Model-Level Validation**: Implement `IValidatableObject` in the model and apply custom validation logic based on property dependencies.
  - **FluentValidation**: Libraries like `FluentValidation` provide a powerful syntax for building conditional validation rules.

3.3 Describe the scope and examples of unit tests and integration tests for a Web API that transfers funds between accounts.

- **Detailed Answer**:
  - **Unit Tests**: Test individual methods, such as `TransferFunds`, to verify fund transfer logic, input validation, and boundary conditions.
  - **Integration Tests**: Validate that all components (API endpoints, services, database interactions) work together correctly. Test scenarios like successful transfer, insufficient funds, and transaction rollbacks.

---

4. **Event-Driven Development**

4.1 For a notification system that sends notifications by email and SMS upon user registration, would you use a queue or topic? Why?

- **Detailed Answer**: Use a **topic** because it allows broadcasting a single message to multiple consumers (Email and SMS services) simultaneously. Queues are more suitable for point-to-point messaging, where each message is consumed by only one receiver.

4.2 Compare Kafka and RabbitMQ in terms of use cases and strengths.

- **Detailed Answer**:
  - **Kafka**: Best for high-throughput, low-latency streaming, commonly used in data pipelines and real-time analytics.
  - **RabbitMQ**: Focused on managing complex message routing, often used for event-driven microservices.

4.3 Describe the lifecycle management of messages in Kafka.

- **Detailed Answer**: Kafka retains messages for a specified time or until a size limit is reached (retention policy). Log compaction retains only the latest record for each key, managing storage efficiently.

5. **Development Methodology**

5.1 Do you approach your work with a design-first or code-first mindset? Which tools do you use for designing APIs or components?

- **Detailed Answer**:
  - **Design-First Approach**: Prioritize planning and designing APIs before implementation. Tools like Swagger or OpenAPI are used to define and document APIs, ensuring clarity for stakeholders.
  - **Code-First Approach**: Use tools like Entity Framework Code-First for rapid prototyping, where code drives the model definition.