Task Three

Overall Solution:

Firstly, to avoid over-engineering, I expand my idea step by step. I consider it as a .Net Web-Application project that offers a UI presentation and web API endpoint. UI representation for end-user and webApi endpoint for the scenario which negotiates with other services.

Solution components:

I see just one responsibility which take a backup from SalesForce and upload them in Azure and Aws. I ask you one question, is there any possibility in the future that we want to add another uploader? Maybe after a while you want to persist it in GCP or others. There is a narrow border between over-engineering and having an extensible design which becomes clear after asking questions from business experts and having a good understanding of the business. If we have an assumption that it can be extensible, we can use the Open-Close Principle that lets us without changing the code we can put beside the project another upload provider. If we want to make a run-time decision to select one of them, then we can use a strategy design pattern.

Let me expand my scenario, from now on we suppose our app is an instantiated module in a Microservice Service Mesh System and it is hosted behind Api-Getway and end-users can access it by webUi and other Microservices according to their requirement call its endpoint. Then concurrency issues are possible and async programming is useful. Then we should pay attention that if we register a service that has a state, as a singleton then their states get shared among other requests.

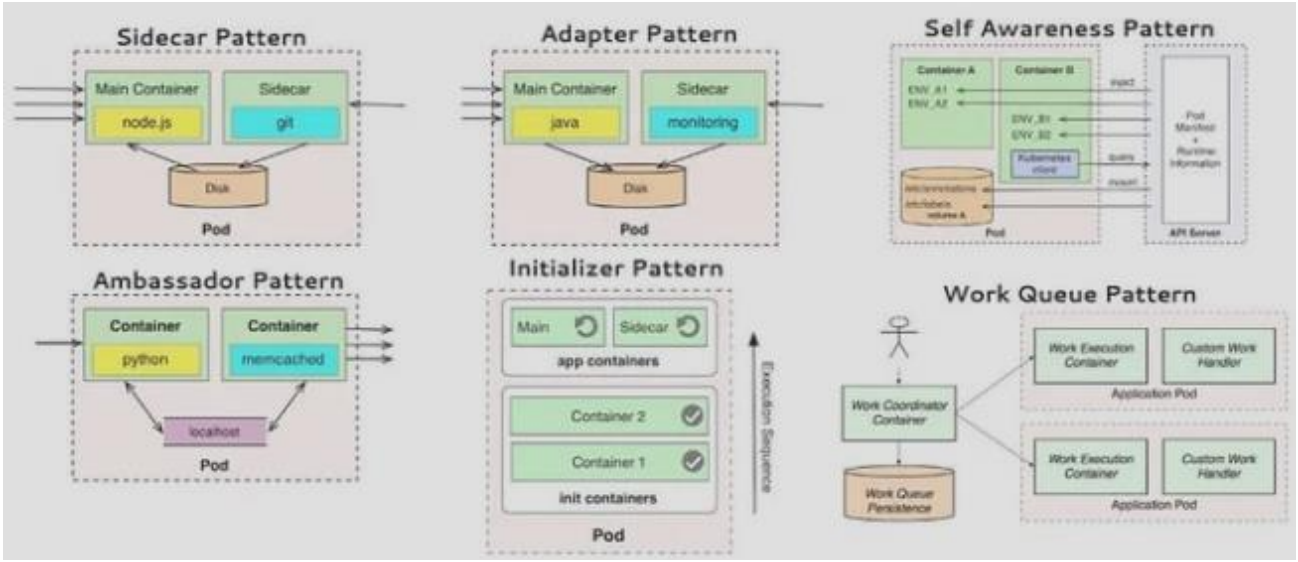
By following our scenario, we can provide a list of below features:

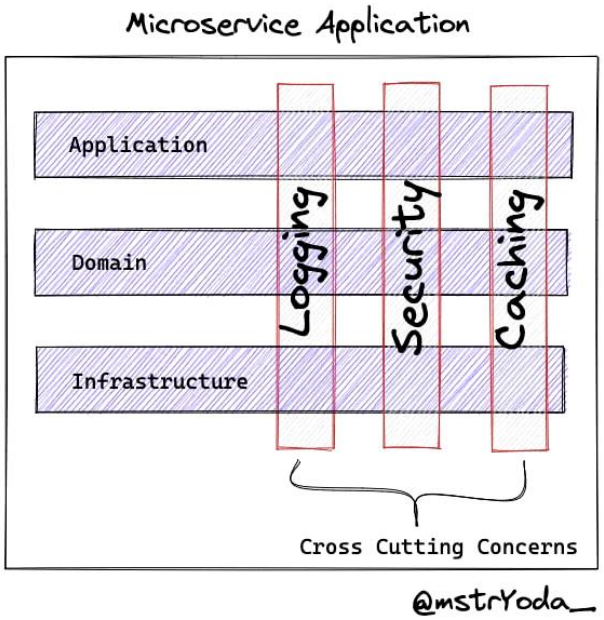
- a report of all ongoing backup processes and the history of all previous requests, enabling the customer to retry the failed ones. All ongoing processes can have an online progress bar.
- Represent some information about the duration of the backup process.
- Ability to define a schedule process which take a Cron expression from end-user. Using Quartz

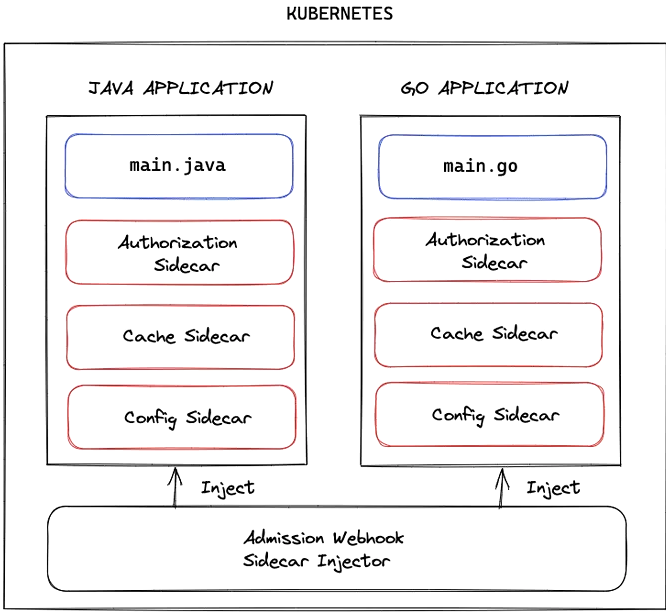
Also, we have some cross cutting concern such as:

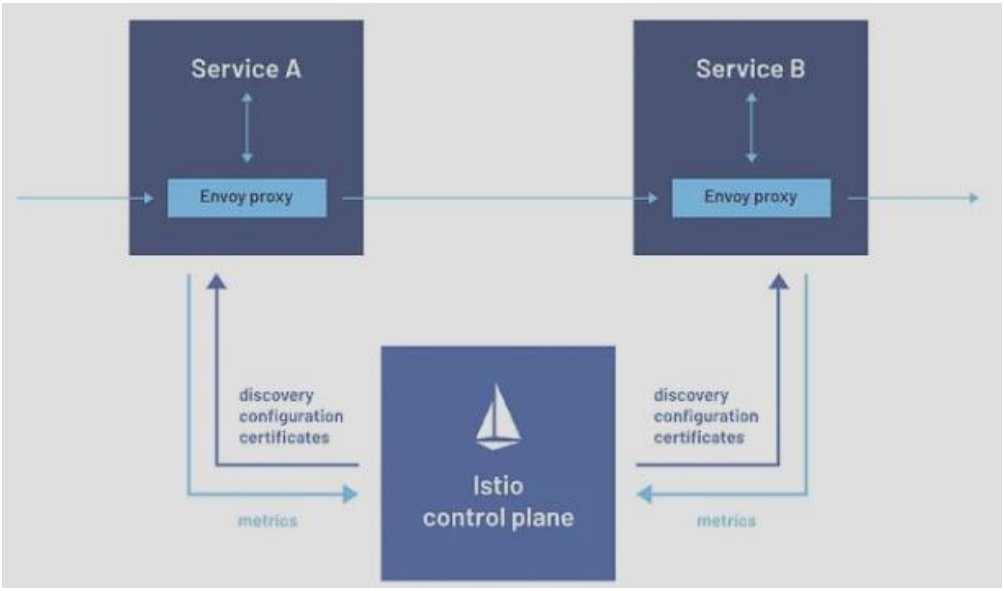
- Audit log
- Activity log
- Application log
- Monitoring
- Authentication, authorisation
- Configuration management
- Automatic instantiation
- Security
- Cache management
- Circuit Breaker

By following our assumption, we have a microservice that is equipped with a Service Mesh System (Side-Car Pattern) then we have already had a solution for all of them except Automatic instantiation that Dockerization and Kubernetes can handle.









Also, if we use one of the Cloud Platforms such as Azure, it can provide us with many tools and facilities such as its log that causes having a full-featured dashboard for Monitoring.

Backend Structure:

By taking into consideration all the above points I will empower the backend side by .net 8 and use the advantages of:

- IHostBuilder
- BackgroundService which named it as a worker.
- Channel as an internal queue
- The worker listens to the channel.

1. **Request**: A request comes from UI by calling the backup endpoint which is located in BackupController, it is a Restful API then uses post-http-verb. And support versioning

   Post httpVerb: https://salesforceBackup.NewYorker.com/api/V1/backeup

   ```
   BackupCommand: {
         SalesforceUsername
         SalesforcePassword
        SecurityTokenforSalesforce
        Salesforcehostnameforyourorg
        AWSaccesskey
        AWSsecretkey
       Azureaccountname
       Azuresharedkey
   }
   ```

2. **Processing Flow:** BackupController dispatch BackupCommand and in BackupApplicationHandler we have BackupCommandHandler which is responsible for handling this command and trying to download ScalesForcePage Content, then Put them in the Channel, then reply to the consumer that "Backup Is Processing". On the other hand, the worker is listening to the Channel, and as soon as a newly downloaded file comes, try to upload it to one of the Upload providers. When all files are uploaded, then push to the consumer that it gets finished.

3. **Scheduler**: Using Quartz for the scheduling mechanism which just needs to dispatch BackupCommand, it gets handled by BackupApplicationHandler

4. **ILogger**: If we use the could platform, ILogger is wrapped by the Could logger provider, Otherwise, I prefer wrapping it with Serilog, and making an Elastic configuration for Serilog, and then logging them in Elastic. Then using Prometheus for Monitoring.

5. **Business Flow form End-User Side**: list of End-User functionality
   - make a Query on BackupRecord Object.
   - Backup request
   - Retry Backup Request

6. **Database**:
   - for keeping track of the BackupRecord Object, correspondingly we have BackupRecordEntity.
   - for keeping track of the BackupScheduler Object, correspondingly we have BackupSchedulerEntity.

   I couldn't extract more objects in this domain that need persistence, we can impose more complexity on the business but keep it simple (KISS Principal). Per each backup request, we have one BackupRecord which has a state that show its status. And keep the data of the backup request, that can be used it for retry if the process fails.
   In BackupApplicationHandler BackupRecordRepository is injected and as soon as the command is going to be handled, firstly we add it to BackupRecordRepository

7. **SQL server and EF Core:** I prefer using SQL server and EF Core, we decorate our command handler with a decorator in which the Unit of work is injected and calls Savechange() at the end of each method of the command handler. That guarantee has a transactional behavior in our command handler.

8. **Checking Idempotency:** BackupRecord has an Id property which is fed by the hash of BackupCommand instance, then we use this Id as an Idempotency key, and in our BackupCommandHandler we would check if we have an ongoing process with this Id or not, it means we only let one ongoing backup process is going on for specific id.

9. **Authentication and Authorization**: we can divide this subject into two different parts.
   First, for the UI side, I suggest using Basic Authentication in ASP.NET Web API
   For restful Api Side, as we assume it is part of our service Mesh, generally they are equipped by SST for oAuth and OpenID such as using IdentityServer4. Then the only thing we need to implement is adding their middle-ware to our project and defining a new scope for our webApp in SST.

10. **How can ensure that the web application maintains or improves upon the functionality of the existing command-line application?**
    By taking a benchmark we can answer how much we have improved. We have a mature library for taking this benchmark. Also, Load Tests and Stress Tests can specify how much improvement we have. Generally, it is crystal clear that after using the below feature we can expect to have a huge improvement.
    - asynchronous programming
    - using an inside queue (channel)
    - BackgroundService
    - Automatic instantiation that Dockerization and Kubernetes can handle it.

    But the Benchmark in real situations, is the only measure and criteria that determine which one is better.

11. I can expand my idea more but again I have a concern about over-engineering and premature optimization. But let me extend the scenario that we have a very huge volume of backup files and countless requests, then we can present another design. I would consider Downloader, Uploader, and Backup as three different systems, which negotiate with each other by Kafka as a stream processor platform. In this design, Kafka is substituted instead of the .net channel. in this design I distribute these three (Downloader, Uploader, and Backup) services then we need Kafka as an external stream processor.

12. **Azure & AWS Data Storage:**

    1. **Azure:** Azure Blob Storage is a scalable, high-performance object storage solution for unstructured data, such as images, videos, documents, and backups. We have three types of Blobs
       - Block Blobs: Used for storing large amounts of unstructured data, such as text or binary data. Ideal for file uploads.
       - Append Blobs: Optimized for append operations, such as logging.
       - Page Blobs: Used for random read/write operations, typically used for VHD files and Azure virtual machine disks.

       I will use Block Blobs as a file storage.

    2. **AWS:** I didn't have any chance to check AWS Data Storage

Frontend Structure:

There is a range of products and technologies for the Front-End development side.

- Asp net MVC MVC patteren
- Angular Web Mvvm pattern
- React Web Mvvm pattern
- VueJs Web Mvvm pattern
- Wpf Windows Mvvm Pattern

Also, a wide range of products for Push Notifications.

- Signal R
- One signal
- Lightstreamer Web/Node.js Client SDKs

I think selecting one of the products is related to business requirements. If the business needs to have a SPA (single page Application) requirement, according to our team's knowledge, I select one of the web Mvvm technologies (Angular, React, VueJs).

1. Angular: If there isn't any knowledge concern in our team, I prefer choosing Angular. Because of below advantages:
    - it has a better implementation for two-way bounding principles,
    - provides us with more Reusability & Modularity which able Components and other parts to be divided into modules.
    - robustness, extensive ecosystem, and **strong community** support make it a go-to choice for many developers

2. ASP.net MVC: I think ASP.net MVC can be another option if refreshing the page is tolerable for end-users. it has a perfect integration with Basic Authentication in ASP.NET Web API template. By using ASP.NET Scaffolding we can generate most Front-end viewers and some basic web APIs and controllers which can expedite the development life cycle.

    Note: using Angular or Asp.net MVC depends on our business requirements and need to share it with Stakeholders and Product owners. With Angular, we can design a SPA that gives a better experience to the end users by providing smooth and responsive behavior. On the other hand, by ASP.net MVC we can bring it up very quickly.

3. push notifications: if we are based on Azure, SignalR is one of our best choices. Also, LightStreamer is a nodeJs free package that is so popular option for this purpose.

4. Responsiveness: I prefer using Bootstrap to provide responsiveness as a Powerful, extensible, and feature-packed frontend toolkit.

5. User interface design:

    1. we have a page for managing BackupRecords.

        - Show the list of BackupRecord
        - Send a request for a new backup.
        - See the "ongoing progress bar" (which shows the percentage of progress) in front of each BackupRecord that are in the state of ongoing.
        - Per each failed record we have a retry button,
        - Per each record we can add a description
        - see the result of progress (pending, ongoing, failed, progressed, retried)

    2. we have a page for managing the BackupScheduler

        Per each specific backup attribute, we can create a new BackupScheduler record

- Show the list of BackupScheduler
- Create a new Scheduler for specific backup attributes
- Crud functionality on each record

Test:

I didn't have any chance to add tests to these solutions. But for each elements, we can have a unit test:

- Downloader
- Uploader
- Backup
- AppSetting

We can use below tools and frameworks for Unit testing:

- Xunit: as a test framework
- Moq or NSubstitute: as a Mocking tools
- NBuilder: allows you to rapidly create test data

Integration test: we can use the below framework and tools

- Gherkin cucumber: for writing high-level test specification
- BDDfy and SpecFelow: for writing low-level test specification
- Selenium: for example submitting a Button from UI

//

Note: in my GitHub by this address you can see the below folders:

1- Reports: which contain the response of these three task.
2- MyDotNetFrameworkSalesForceBackup: the code-reviewed version of SalesForceBackup
3- MyDotNet8RefactoredSalesForceBackup: the .net8 version of the project which is a Work In Progress