Development, Containerization, and Deployment of a Redaction Microservices-Based Application

Author: Maria Magdalena Michail

Date: 22/12/2024

Course: Cloud Engineering

Institution: Athens Tech College

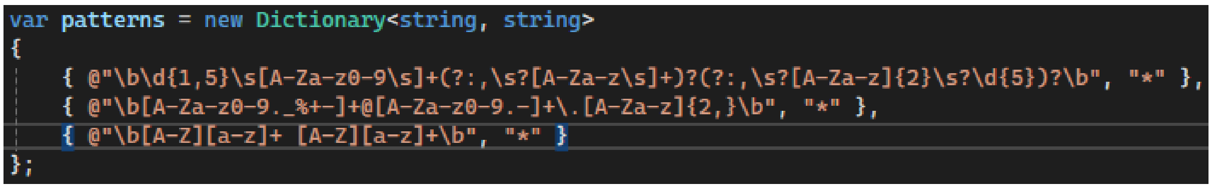
Table of Contents

Part A: Development of the Redaction Microservice and Web App

1. Design and Development of the Redaction Microservice

- **Functionality:**
    - **Redaction logic:**

Regular Expressions (Regex) for detecting names, email addresses, and street addresses.



```
var patterns = new Dictionary<string, string>
{
    { @"\b\d{1,5}\s[A-Za-z0-9\s]+(?:,\s?[A-Za-z\s]+)?(?:,\s?[A-Za-z]{2}\s?\d{5})?\b", "*" },
    { @"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b", "*" },
    { @"\b[A-Z][a-z]+ [A-Z][a-z]+\b", "*" }
};
```

The `RedactSensitiveInformation` method is designed to identify and redact sensitive information from text by replacing it with asterisks (*). It uses a dictionary of regex patterns, with each pattern tailored to detect specific types of sensitive data:

1. **Email addresses**: Matches common email formats and replaces them with `*`.
2. **Full names**: Matches names with a capitalized first and last name (e.g., "John Doe").
3. **Addresses or similar structured text**: Matches patterns that resemble addresses (e.g., "123 This Street" or "Other Street 12, City").

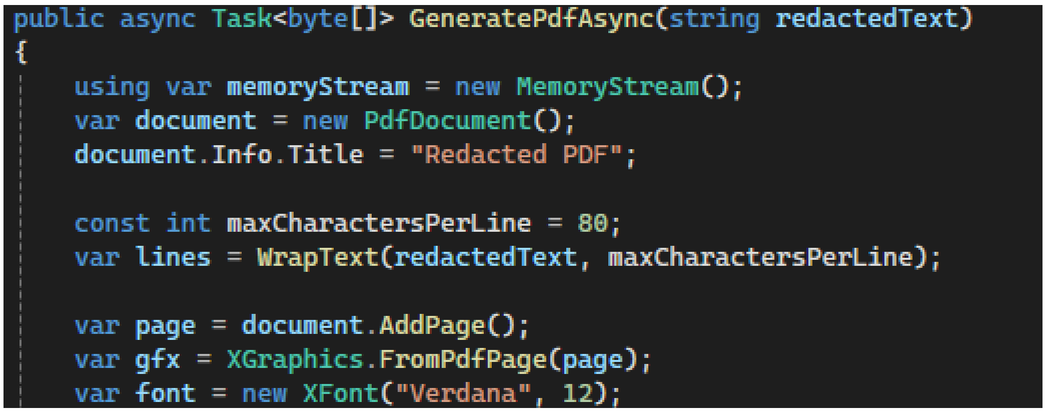
The method loops through these patterns, applying `Regex.Replace` to remove sensitive details while leaving non-sensitive text untouched. The resulting text is returned, fully redacted.

-
    - **PDF generation using the library PdfSharp:**

The `GeneratePdfService` class generates PDF files from input text, ensuring proper formatting and layout across pages. The class uses the PdfSharp library to create and manage PDF documents.

Key Components:

1. **`GeneratePdfAsync(string redactedText)`**: This is the main method that generates the PDF. It takes a string (`redactedText`) and processes it by splitting it into lines that fit within the specified page width. The text is then drawn on the PDF using `XGraphics` from the **PdfSharp** library.
    - **MemoryStream**: The PDF document is saved to a `MemoryStream`, which will be returned as a byte array. This byte array can then be used to save the PDF file or return it via an API.



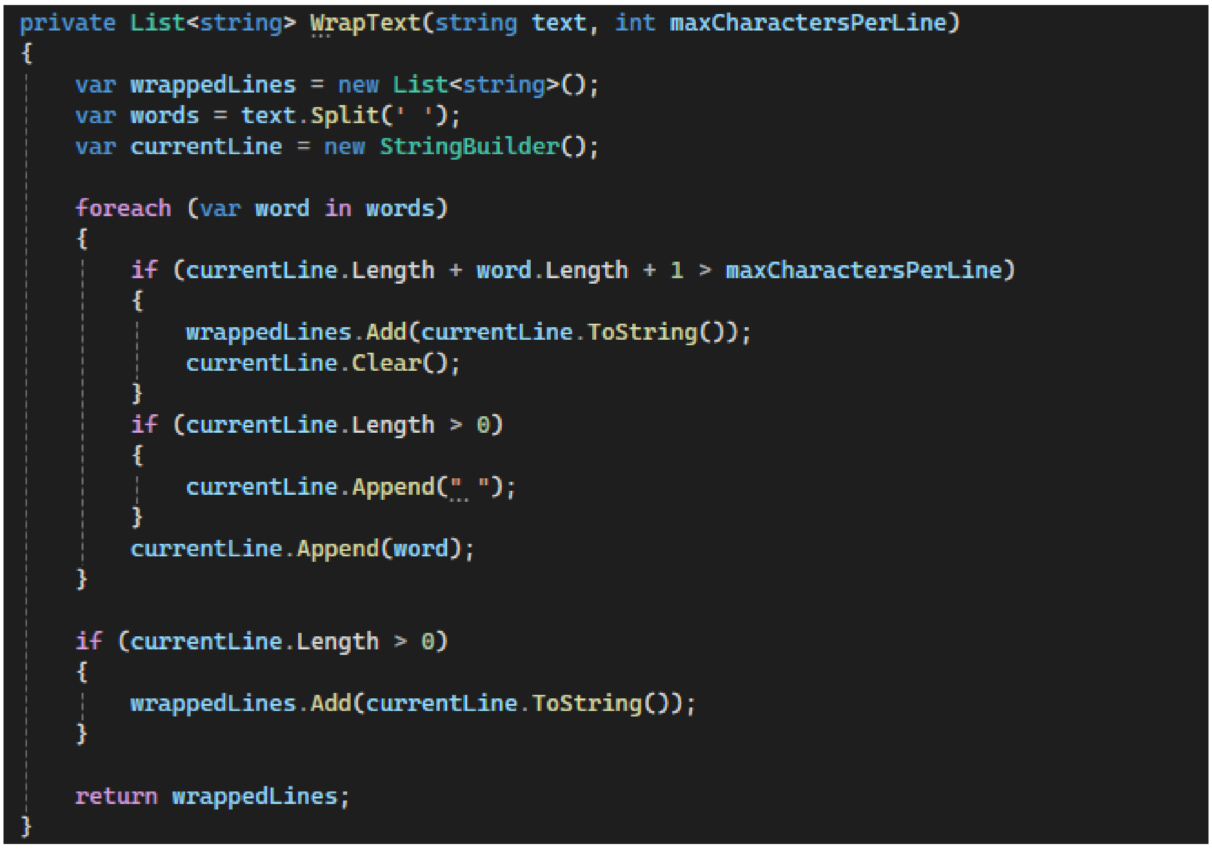
```
public async Task<byte[]> GeneratePdfAsync(string redactedText)
{
    using var memoryStream = new MemoryStream();
    var document = new PdfDocument();
    document.Info.Title = "Redacted PDF";

    const int maxCharactersPerLine = 80;
    var lines = WrapText(redactedText, maxCharactersPerLine);

    var page = document.AddPage();
    var gfx = XGraphics.FromPdfPage(page);
    var font = new XFont("Verdana", 12);
```

- **Text Wrapping**: The text is wrapped to a specified line length (`maxCharactersPerLine`). The wrapping occurs via the `WrapText` method.
- **Margins and Positioning**: The `leftMargin` and `topMargin` constants define the margins of the PDF. The `yPosition` keeps track of where to place the next line of text on the page. If the current page becomes full, a new page is added to the document.

2. **`WrapText(string text, int maxCharactersPerLine)`**: This helper method is responsible for breaking the input text into multiple lines, ensuring that each line does not exceed the specified character limit (`maxCharactersPerLine`). It splits the text into words and adds them to the current line until the maximum length is reached. If the line exceeds the limit, it adds the current line to the list and starts a new one.



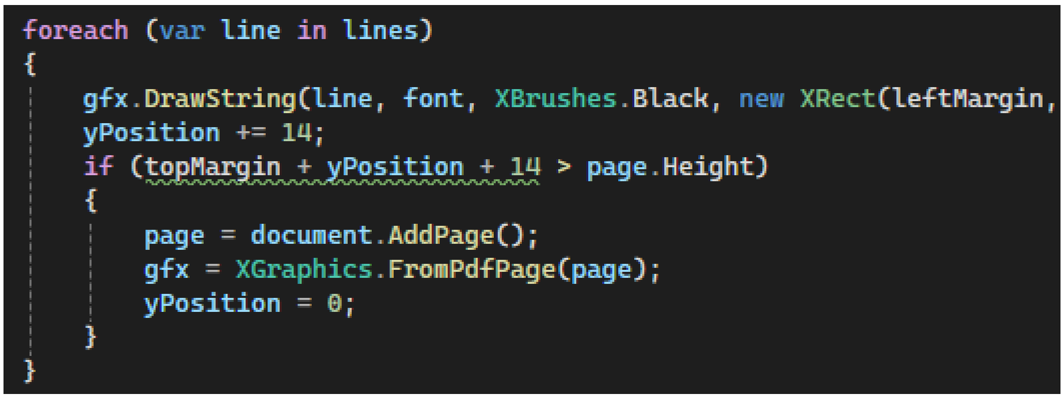
```
private List<string> WrapText(string text, int maxCharactersPerLine)
{
    var wrappedLines = new List<string>();
    var words = text.Split(' ');
    var currentLine = new StringBuilder();

    foreach (var word in words)
    {
        if (currentLine.Length + word.Length + 1 > maxCharactersPerLine)
        {
            wrappedLines.Add(currentLine.ToString());
            currentLine.Clear();
        }
        if (currentLine.Length > 0)
        {
            currentLine.Append(" ");
        }
        currentLine.Append(word);
    }

    if (currentLine.Length > 0)
    {
        wrappedLines.Add(currentLine.ToString());
    }

    return wrappedLines;
}
```

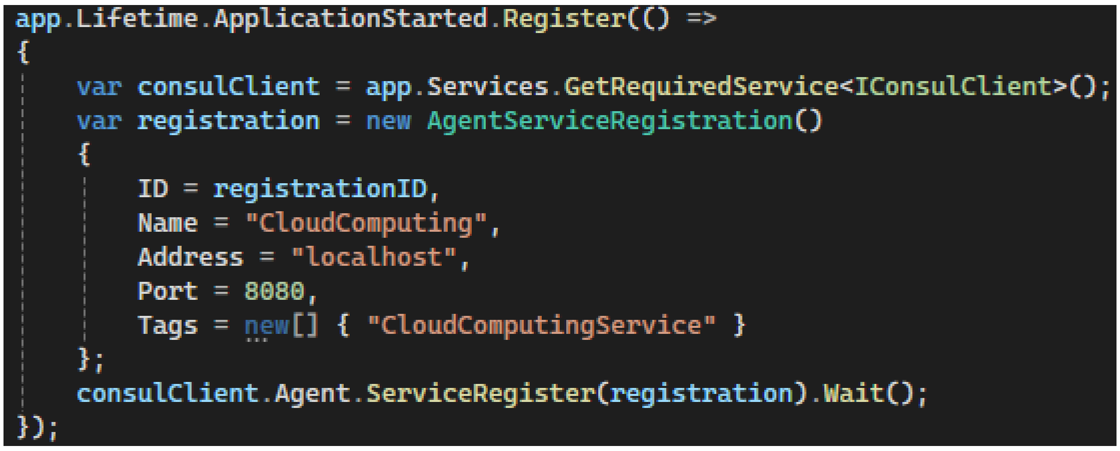
3. **Handling Multiple Pages**: If the text continues beyond the first page, the method adds new pages and continues drawing text on the new pages, ensuring that text is properly spaced out across multiple pages if needed.



```
foreach (var line in lines)
{
    gfx.DrawString(line, font, XBrushes.Black, new XRect(leftMargin,
    yPosition += 14;
    if (topMargin + yPosition + 14 > page.Height)
    {
        page = document.AddPage();
        gfx = XGraphics.FromPdfPage(page);
        yPosition = 0;
    }
}
```

- **Implementation Details:**
    - **Controller Endpoints:**
        - **`CreatePdf`**: Accepts text, redacts sensitive information, generates a hashed PDF, saves it, and returns a response with a link to retrieve the PDF.
        - **`GetPdfByHash`**: Fetches and returns the PDF file corresponding to the provided hash if it exists.

        - `ListPdfs`: Lists all generated PDF hashes.
    - **Interfaces:**
        - The application relies on two services—`IRedactionService` for redacting sensitive data and `IGeneratePdfService` for PDF creation. These services are implemented via interfaces. The controller delegates tasks to these services, keeping the logic separate from the presentation layer.
    - **Logging:**
        - Logging is used to track application behavior making it easier to debug and monitor the system. In this code, logging captures invalid requests or errors.
- **Service Registration:**
    - This application uses Consul to manage service discovery and monitor health within a microservices architecture. Consul serves as a centralized registry where services register themselves during startup, making them easily discoverable by other services. It also helps maintain the system's reliability by performing periodic health checks on registered services.

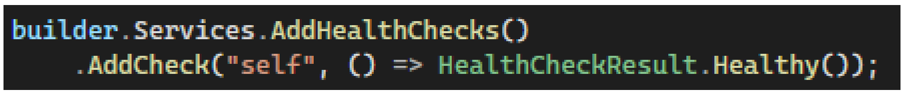


```
app.Lifetime.ApplicationStarted.Register(() =>
{
    var consulClient = app.Services.GetRequiredService<IConsulClient>();
    var registration = new AgentServiceRegistration()
    {
        ID = registrationID,
        Name = "CloudComputing",
        Address = "localhost",
        Port = 8080,
        Tags = new[] { "CloudComputingService" }
    };
    consulClient.Agent.ServiceRegister(registration).Wait();
});
```

When the application starts, it registers itself with a Consul agent, making it discoverable to other services that need to communicate with it.

Health Checks for Monitoring

To ensure the health and availability of the registered services, Consul performs regular health checks on each service. The application exposes a health check endpoint (`/health`), which is queried by Consul to verify that the service is running correctly. This health check is configured in the application using:



```
builder.Services.AddHealthChecks()
    .AddCheck("self", () => HealthCheckResult.Healthy());
```

The health endpoint is exposed to allow Consul to ping the service and confirm its status. In the code, we use:

```
app.MapHealthChecks("/health");
```

This setup ensures that Consul can check the health of the service by calling the `/health` endpoint, and only healthy services are registered in the service registry.

Deregistration of the Service

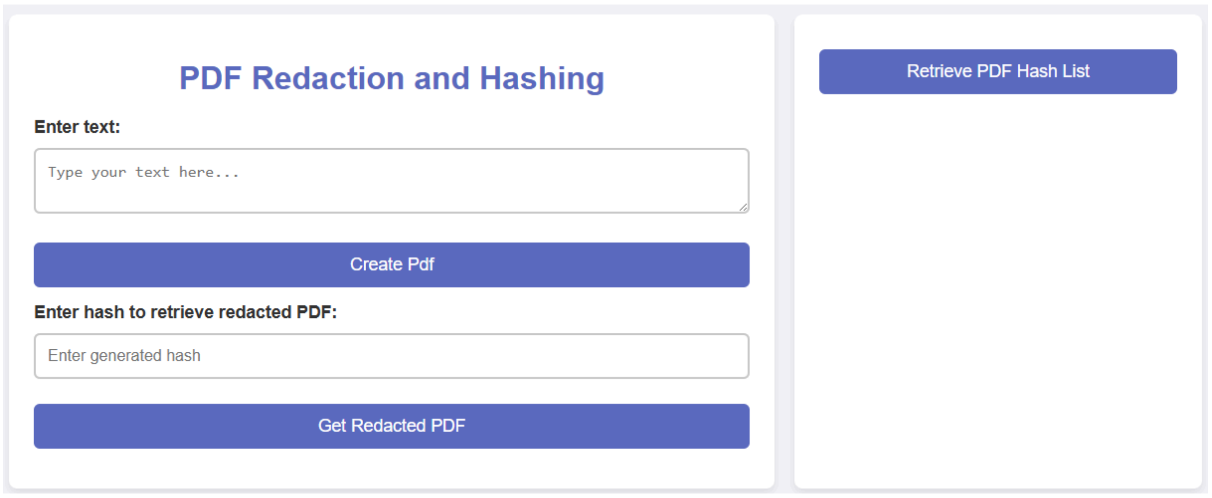
When the application stops or shuts down, it is important to remove the service from the Consul registry. This is achieved using the `ApplicationStopped` event, which ensures that the service is deregistered upon termination:

```
app.Lifetime.ApplicationStopped.Register(() =>
{
    var consulClient = app.Services.GetRequiredService<IConsulClient>();
    consulClient.Agent.ServiceDeregister(registrationID).Wait();
});
```

2. Design and Development of the ASP.NET Core Web App

UI Design:

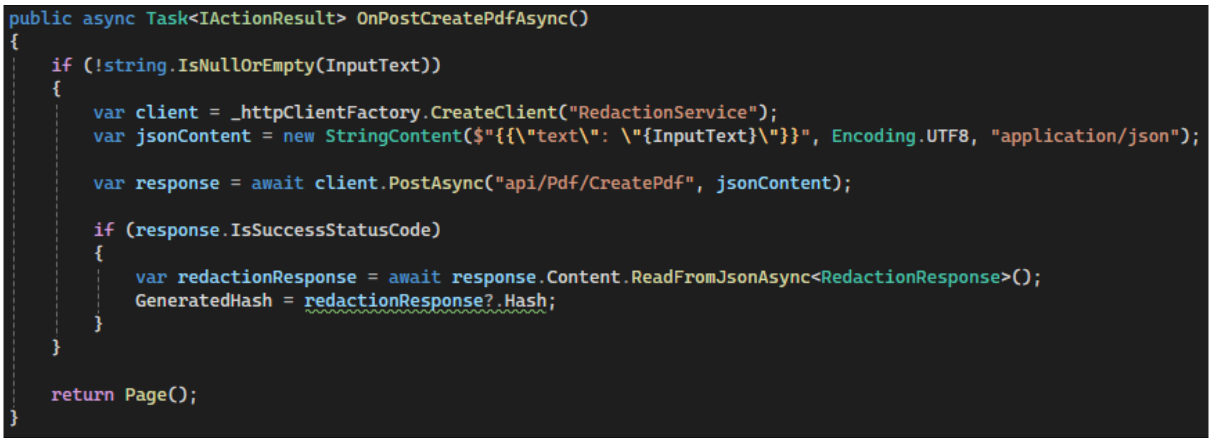
- Input form for submitting text to be redacted and create a PDF.
- Button to retrieve all PDF file names.
- Input for a hash to retrieve a PDF.



Communication with the Microservice:

The APIs for this project were developed using a RESTful design to manage PDF creation and retrieval processes. The backend APIs were built to handle three functionalities: generating PDFs, retrieving PDFs by their hash, and listing all available PDF hashes.

- **PDF Creation API**: A `POST` endpoint was created at `api/Pdf/CreatePdf` to accept JSON payloads containing text to be redacted. The backend processes the text, redacts sensitive information, generates a hash for identification, and returns the hash to the frontend along with a link to retrieve the PDF.



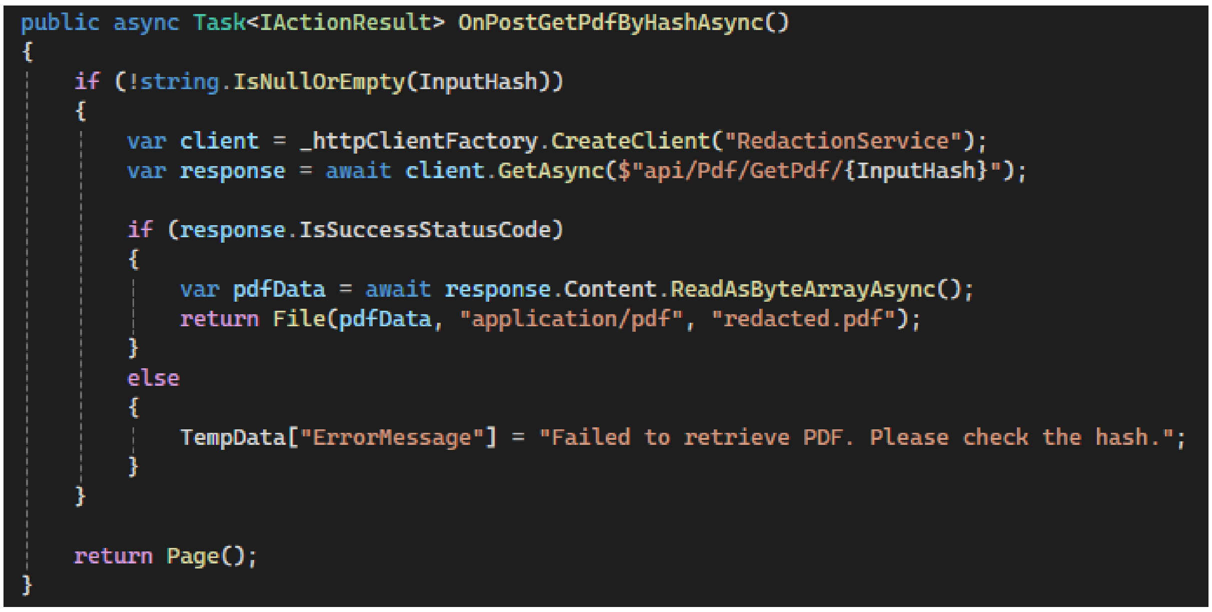
```
public async Task<IActionResult> OnPostCreatePdfAsync()
{
    if (!string.IsNullOrEmpty(InputText))
    {
        var client = _httpClientFactory.CreateClient("RedactionService");
        var jsonContent = new StringContent($"{{\"text\": \"{InputText}\"}}", Encoding.UTF8, "application/json");

        var response = await client.PostAsync("api/Pdf/CreatePdf", jsonContent);

        if (response.IsSuccessStatusCode)
        {
            var redactionResponse = await response.Content.ReadFromJsonAsync<RedactionResponse>();
            GeneratedHash = redactionResponse?.Hash;
        }
    }

    return Page();
}
```

- **PDF Retrieval by Hash**: A GET endpoint at `api/Pdf/GetPdf/{hash}` allows fetching PDFs using their unique hash. The backend verifies the hash, retrieves the corresponding file, and returns it as a downloadable response.

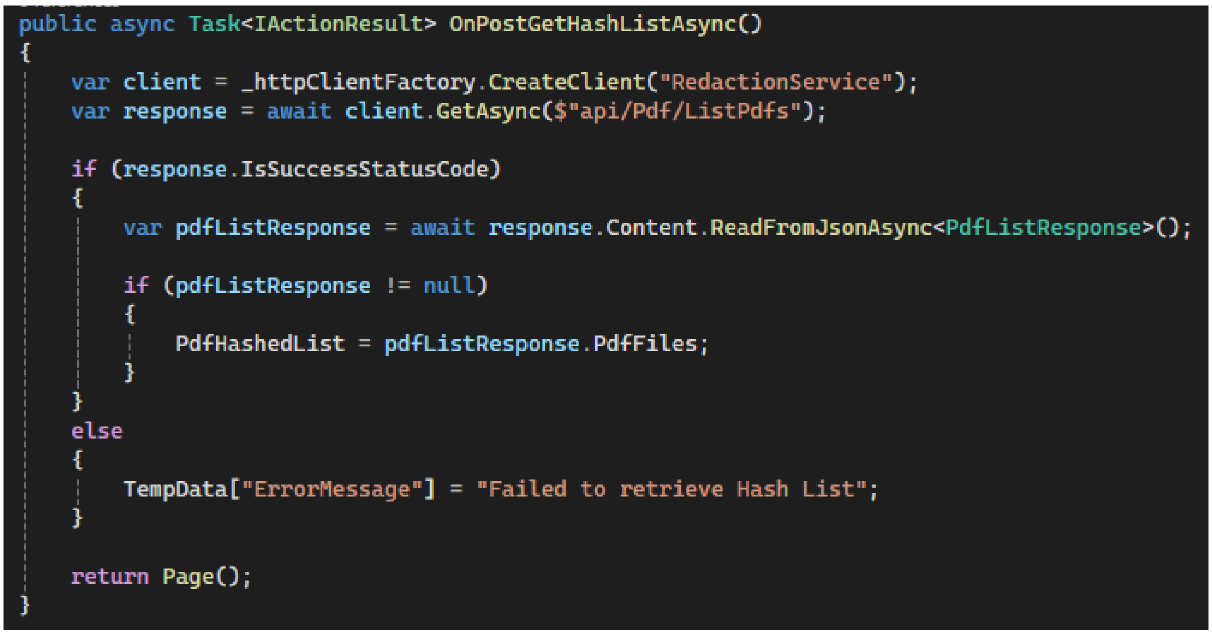


```
public async Task<IActionResult> OnPostGetPdfByHashAsync()
{
    if (!string.IsNullOrEmpty(InputHash))
    {
        var client = _httpClientFactory.CreateClient("RedactionService");
        var response = await client.GetAsync($"api/Pdf/GetPdf/{InputHash}");

        if (response.IsSuccessStatusCode)
        {
            var pdfData = await response.Content.ReadAsByteArrayAsync();
            return File(pdfData, "application/pdf", "redacted.pdf");
        }
        else
        {
            TempData["ErrorMessage"] = "Failed to retrieve PDF. Please check the hash.";
        }
    }

    return Page();
}
```

- **Hash List API**: A GET endpoint at `api/Pdf/ListPdfs` was implemented to list all available PDF hashes. The backend checks the directory for existing files and returns their hashes in a JSON format.



```
public async Task<IActionResult> OnPostGetHashListAsync()
{
    var client = _httpClientFactory.CreateClient("RedactionService");
    var response = await client.GetAsync($"api/Pdf/ListPdfs");

    if (response.IsSuccessStatusCode)
    {
        var pdfListResponse = await response.Content.ReadFromJsonAsync<PdfListResponse>();

        if (pdfListResponse != null)
        {
            PdfHashedList = pdfListResponse.PdfFiles;
        }
    }
    else
    {
        TempData["ErrorMessage"] = "Failed to retrieve Hash List";
    }

    return Page();
}
```

Binding APIs to the Frontend

The frontend was implemented using Razor Pages in an ASP.NET Core application. A `HttpClientFactory` was used to enable efficient communication with the backend microservice. Key functionalities were exposed to users via HTML forms.

- **PDF Creation**: On submitting the input text, the `OnPostCreatePdfAsync` method serializes the input into a JSON payload and sends it to the `CreatePdf` API. Upon success, the generated hash is displayed on the page.
- **Retrieving PDFs by Hash**: Users can enter a hash in a form, triggering the `OnPostGetPdfByHashAsync` method, which makes a GET request to the `GetPdf` API. If successful, the PDF file is returned and downloaded.
- **Listing Hashes**: The `OnPostGetHashListAsync` method fetches available hashes using the `ListPdfs` API. The response is deserialized into a model and displayed as a list in the frontend.

Error handling is integrated into each step, displaying appropriate messages for failed API calls or invalid inputs.

Part B: Containerization and Deployment

1. Containerization of Microservices

The containerization setup includes Dockerfiles for the Redaction Microservice (`CloudComputing`) and Web App (`CloudComputingWeppApp`), along with a `docker-compose.yml` for orchestration.

Redaction Microservice Dockerfile

- **Base Stage**: Uses the runtime image `aspnet:8.0`, sets `/app` as the working directory, and exposes port 80.
- **Build Stage**: Uses .NET SDK to restore dependencies, build the project, and output binaries to `/app/build`.
- **Publish Stage**: Packages the application for deployment into `/app/publish`.
- **Final Stage**: Copies publish artifacts from the previous stage and defines the entry point as `CloudComputing.dll`.

Web App Dockerfile

Follows a similar multistage structure as the Redaction Microservice but targets `CloudComputingWeppApp`.

docker-compose.yml

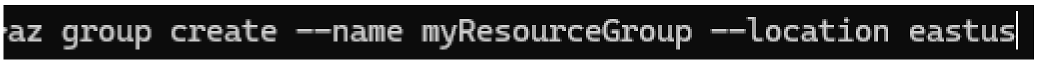
- **Consul Service**: Provides service discovery, exposing port 8500 and including a health check.
- **Web API**:
  - Builds from `CloudComputing/Dockerfile`.
  - Maps port 8080 to 80 and depends on Consul.
  - Passes Consul's URI as an environment variable.
- **Web App**:
  - Builds from `CloudComputingWeppApp/Dockerfile`.
  - Maps port 8081 to 80 and depends on the Web API.
- **Network**: Uses a bridge network to enable inter-service communication.

2. Cloud Deployment and Scaling

Cluster Creation:

The application is deployed on Azure Kubernetes Service (AKS) to ensure scalability and fault tolerance. The AKS cluster includes two nodes.

1. First we created an AKS cluster with Azure CLI:



```
az group create --name myResourceGroup --location eastus
az aks create --resource-group myResourceGroup --name myAKSCluster --node-count 2
--enable-addons monitoring --generate-ssh-keys
```

- `--node-count 2`: Specifies a minimum of two nodes for high availability.
- `--enable-addons monitoring`: Adds monitoring tools for resource metrics.
- `--generate-ssh-keys`: Automatically generates SSH keys for secure access.

2. Connect to the Cluster: After creating the cluster, we obtained its credentials:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

This command configures `kubectl` to interact with my AKS cluster.

Deploy Containerized Microservices:

Redaction Microservice (web-api):

Pushed Container Images to Azure Container Registry (ACR): Before deploying, the container images need to be stored in a registry:

```
az acr create --resource-group myResourceGroup --name myContainerRegistry --sku Basic
az acr login --name myContainerRegistry
docker tag web-api:latest mycontainerregistry.azurecr.io/web-api:v1
docker push mycontainerregistry.azurecr.io/web-api:v1
```

web-api-deployment.yaml:

Deployment:

- Runs 3 replicas for handling increased traffic. The `replicas: 3` setting ensures the microservice scales to handle higher loads.

Service:

- Exposes the service externally through Azure Load Balancer.

Apply the manifests with:

```
kubectl apply -f web-api-deployment.yaml
kubectl apply -f web-api-service.yaml
```

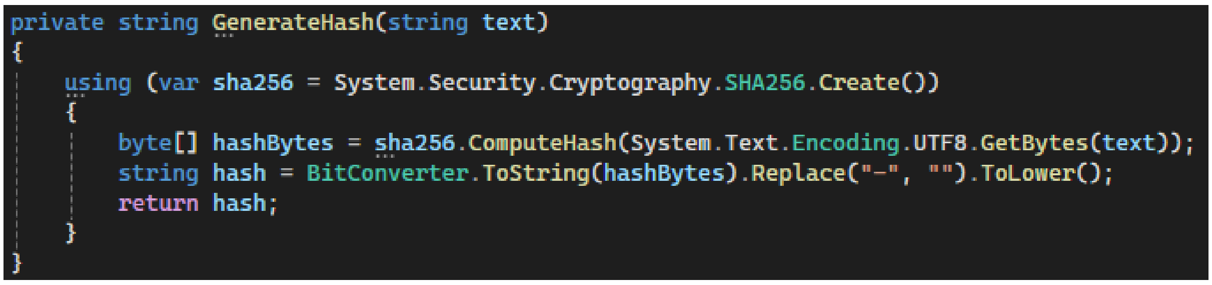
Web App (web-app):

The above process also applies for Web App. Deployment and Service follow a similar configuration, using `myacr.azurecr.io/web-app:v1` as the container image.

3. Enhancing the Redaction Microservice

- **Hashing Functionality:**

Extend the `POST /CreatePdf` endpoint for generating hashes to set as the file name.



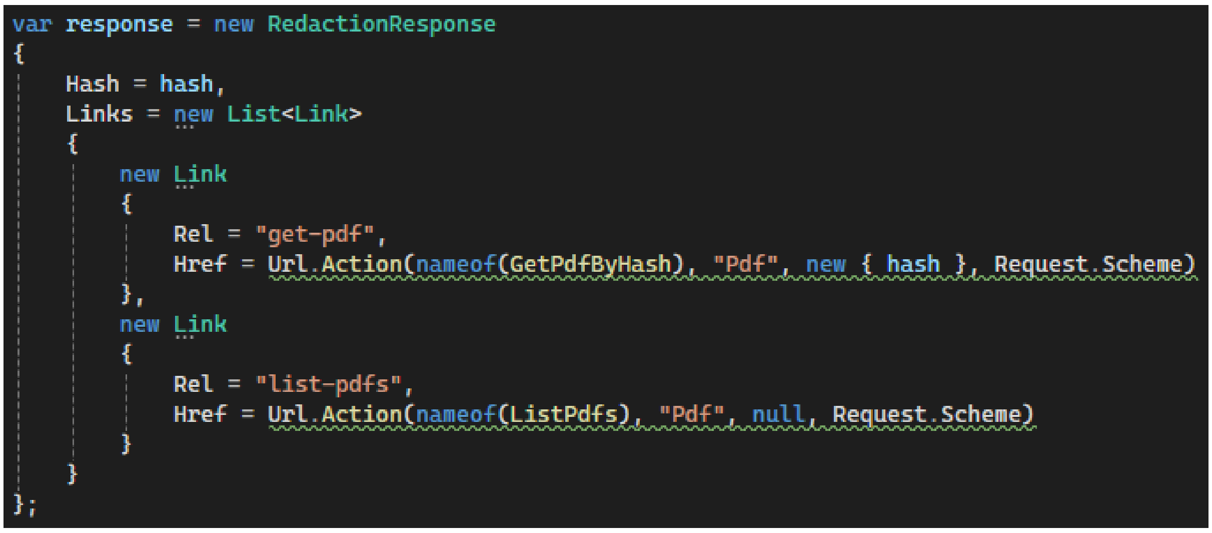
```
private string GenerateHash(string text)
{
    using (var sha256 = System.Security.Cryptography.SHA256.Create())
    {
        byte[] hashBytes = sha256.ComputeHash(System.Text.Encoding.UTF8.GetBytes(text));
        string hash = BitConverter.ToString(hashBytes).Replace("-", "").ToLower();
        return hash;
    }
}
```

This method accepts a text input and returns a cryptographic hash of the text using the SHA-256 hashing algorithm.

- **HATEOAS Implementation:**

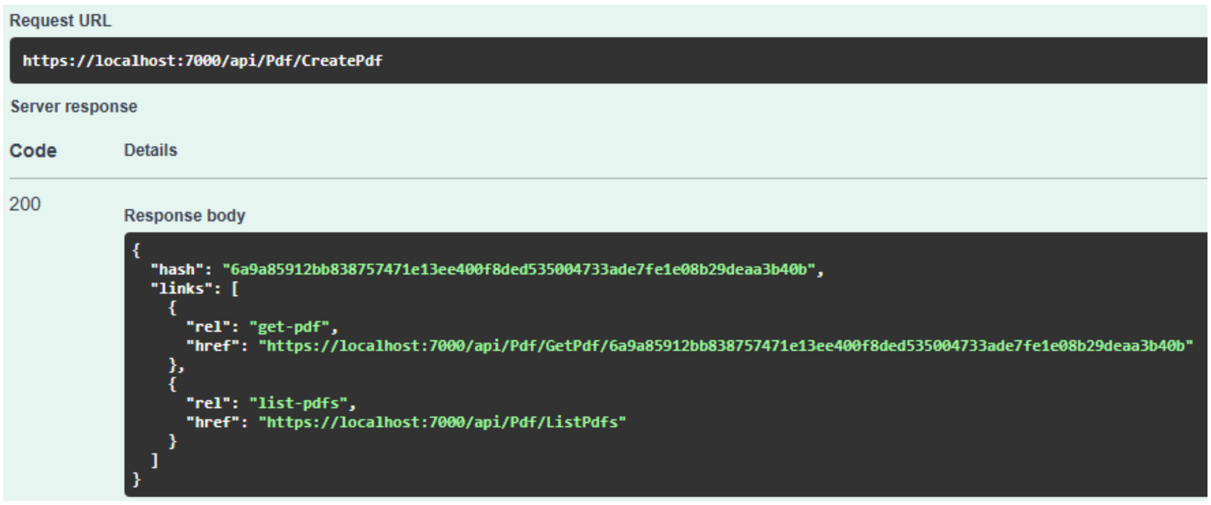
HATEOAS (Hypermedia as the Engine of Application State) is incorporated to improve API discoverability and usability. This approach allows clients to dynamically explore available actions by following links included in API responses, minimizing the need for external documentation.

In the `CreatePdf` endpoint, HATEOAS is applied by adding a hypermedia link to the response. After the text is redacted and a unique hash is generated for the PDF, the system creates and stores the file. The response then provides a `Links` collection containing URLs that enable clients to retrieve the PDF using its hash and access a list of all available PDFs. This design ensures seamless navigation through the API's functionality. This is demonstrated in the following block:



```
var response = new RedactionResponse
{
    Hash = hash,
    Links = new List<Link>
    {
        new Link
        {
            Rel = "get-pdf",
            Href = Url.Action(nameof(GetPdfByHash), "Pdf", new { hash }, Request.Scheme)
        },
        new Link
        {
            Rel = "list-pdfs",
            Href = Url.Action(nameof(ListPdfs), "Pdf", null, Request.Scheme)
        }
    }
};
```

And the response is shown below:



Request URL

```
https://localhost:7000/api/Pdf/CreatePdf
```

Server response

Code Details

200

Response body

```
{
  "hash": "6a9a85912bb838757471e13ee400f8ded535004733ade7fe1e08b29deaa3b40b",
  "links": [
    {
      "rel": "get-pdf",
      "href": "https://localhost:7000/api/Pdf/GetPdf/6a9a85912bb838757471e13ee400f8ded535004733ade7fe1e08b29deaa3b40b"
    },
    {
      "rel": "list-pdfs",
      "href": "https://localhost:7000/api/Pdf/ListPdfs"
    }
  ]
}
```

4. Rolling Update of the Redaction Microservice

A rolling update allows seamless deployment of a new version of the Redaction Microservice without causing downtime.

Build and Push the New Docker Image:

After making enhancements to the microservice, we created a new version of the Docker image:

```
docker build -t mycontainerregistry.azurecr.io/web-api:v2 .
docker push mycontainerregistry.azurecr.io/web-api:v2
```

This command tags and pushes the updated image (v2) to Azure Container Registry (ACR).

Update Kubernetes Deployment:

We modified the `web-api` deployment manifest to use the new image version:

```
    spec:
      containers:
      - name: web-api
        image: myacr.azurecr.io/web-api:v2
        ports:
        - containerPort: 80
```

We applied the updated manifest:

```
kubectl apply -f web-api-deployment.yaml
```

Kubernetes detects the change in the container image and triggers a rolling update.