**Task.** Develop a prototype Skoltech Faculty Info System.
The system will work by reading the “email messages” written in natural language, and replying with the information about faculty members from this source. If the system is sufficiently confident that it understands the request, it will answer the email itself. If it does not understand the request, then it will leave it for a human to handle, replying with “Redirecting request to our staff.” (Systems like this might be helpful in a company that receives many repetitive emails such as requests for shipment tracking numbers.)

The following represents an example “email message” to handle. It is formed by a header and a body, both in plaintext. The header is separated from the body by a blank line.

```
----
From: Alexey Artemov <a.artemov@skoltech.ru>
To: Skoltech Faculty Info System <sifs@skoltech.ru>
Subject: Saying Hello
Date: Fri, 18 Sep 2020 09:55:06 +0300
Message-ID: <1234@local.machine.example>

This is a message just to say hello.
So, "Hello".
----
```

We are interested in requests for faculty information like this:

```
----
From: Alexey Artemov <a.artemov@skoltech.ru>
To: Skoltech Faculty Info System <sifs@skoltech.ru>
Subject: information on Alexander Bernstein
Date: Fri, 18 Sep 2020 09:55:06 +0300
Message-ID: <1234@local.machine.example>

Please provide me with a position of Alexander Bernstein
----
```

The goal of the system is to try to recognize that the request is about Alexander Bernstein, and the target information related to this person is his position at Skoltech. An example reply from the system may be structured like:

```
----
From: Skoltech Faculty Info System <sifs@skoltech.ru>
To: Alexey Artemov <a.artemov@skoltech.ru>
Subject: Re: information on Alexander Bernstein
Date: Fri, 18 Sep 2020 09:59:12 +0300
Message-ID: <1234@local.machine.example>

Position of Alexander Bernstein: Full Professor.
----
```

We will not send real emails for the sake of simplicity (though, implementing a real email server is relatively easy done in python). Instead, we will implement a simple asynchronous client-server application,  and a set of filters that will process the text message, determining whether it is able to reply, and finally replying using the same “email” format.

The following set of tasks will guide you through building the system.

**Problem 1.** Design and implement a set of email processing filters. Their goal is to decide if an email can be answered automatically, and in which form, or whether a human should be answering it. Please see the provided code to

- **Task A.** Implement a cleaning filter. The input to the filter is an “email message” (represented as a Python dict with a single ‘message’ key), and the output must be its stripped version without headers and non-alphanumeric characters (also represented as a Python dict).
- **Task B.** Implement a tagging filter. The input to the tagging filter is a cleaned “email message” (represented as a Python dict), and the output is a list of tags.
- **Task C.** Implement a name extraction filter. The input to the tagging filter is a cleaned “email message” (represented as a Python dict), and the output is a structured set of extracted names (represented as a Python dict).

**Problem 2.** Design and implement the client-server system architecture using ZeroMQ. To make sure everyone uses a different port, use this function to select a port for you: `hash(name_surname) % 64000 + 1024`.

- **Task A.** Create the client implementation. The client shall use a REQ socket to send messages to the server.
- **Task B.** Create the server implementation. The server shall use a REP socket to send messages to the client.

**Problem 3.** Document the system component architecture using UML notation. You may use online tools such as LucidChart or Draw.io.

- **Task A.** Enumerate components and connectors needed in the system design. Use the pipe-and-filter style and client-server style architectures.
- **Task B.** Create a component assembly diagram for the system.

**Problem 4 (hometask).** Modify the system in the following ways.

- **Task A.** Make the system asynchronous by letting the client “publish” an email, while the server (or servers) “subscribe” to the messages from the client. Note that in this way we require another process on the client side, as the client will no longer wait for server replies. Add this third process to check for messages from the server, via the same “publish>subscribe” model.
- **Task B.** Add another server to balance the load between the two server component instances. Make sure that the architecture is stable.
- **Task C.** Update the architectural UML diagram to reflect changes introduced by Task A and Task B above.