

## InstaNet Project: Phase 2

Spring 2025

Deadline: Thursday 15<sup>th</sup> May **4PM**

### Introduction

In Phase 1, we laid the groundwork by setting up the development environment, installing essential dependencies, and developing a simple client-server application using socket programming. These steps were crucial in preparing us to build the real-time, interactive features required.

In this phase, we will design and implement a fully functional social media application named InstaNet, inspired by the core features of Instagram



InstaNet is a mock networked application where a program uses socket programming to send and receive text and images, emulating key features of the popular social media platform. What makes InstaNet a networked application rather than a standalone desktop program is its ability to enable communication between clients and a server, which you will help us implement with socket programming.

## Implementation:

To start the program you first have to download the dependencies just like in Phase 1. Phase 2 uses CustomTkinter and Pillow, so you will need to install it as a dependency:

```
customtkinter==5.2.2
```

```
Pillow==10.2.0
```

CustomTkinter is used to create modern, customizable graphical user interfaces (GUIs), and Pillow is used for opening, manipulating, and saving image files in Python.

Recall from phase 1, there are two ways to install the dependencies:

### 1) Using pip with requirements.txt:

- Open your terminal or command prompt.
- Navigate to the folder where your requirements.txt file is located.
- Run: `pip install -r requirements.txt`

### 2) Installing them individually (if needed)

- Run: `pip install customtkinter==5.2.2`
- Run: `pip install Pillow==10.2.0`

## Starting the Project

After installing the required dependencies, you will **not** be able to run the program immediately. You must first complete **Task 1** and **Task 2** below, which involve setting up the server and client socket connections.

Once task1 and task2 are completed, follow the steps below to launch the application:

### For Mac:

- Run: `Python3 server.py -port 5001`
- Run: `Python3 client.py -port 5001`

### For windows:

- Run: `Python3 server.py -port 5001`
- Run: `Python3 client.py -port 5001`

Note: Here, we are setting the server's source port number to 5001 and the client's destination port to 5001. Therefore, it is important to ensure that both the server and the client use the same port number when running the application.

## Login Instructions:

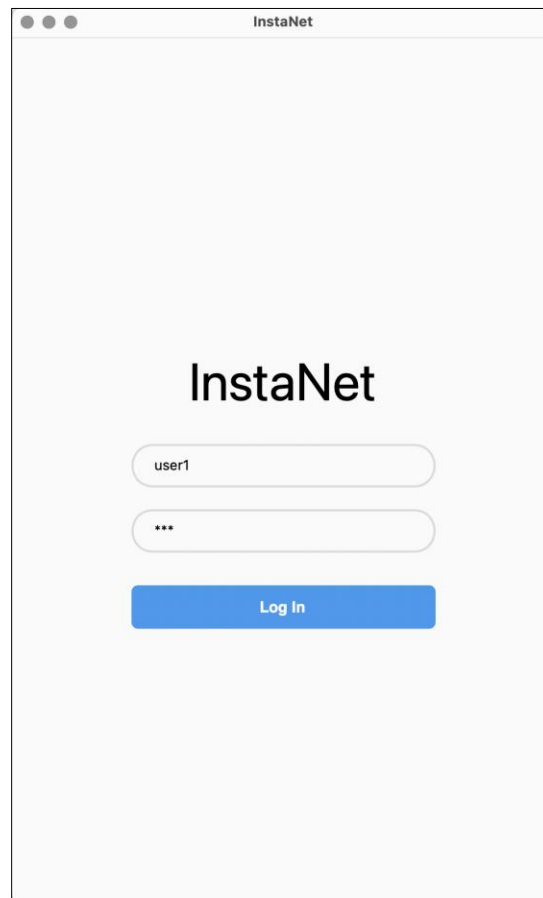
After running the above commands (but finish Task 1 and Task 2 first), the application will take you to the **login page**.

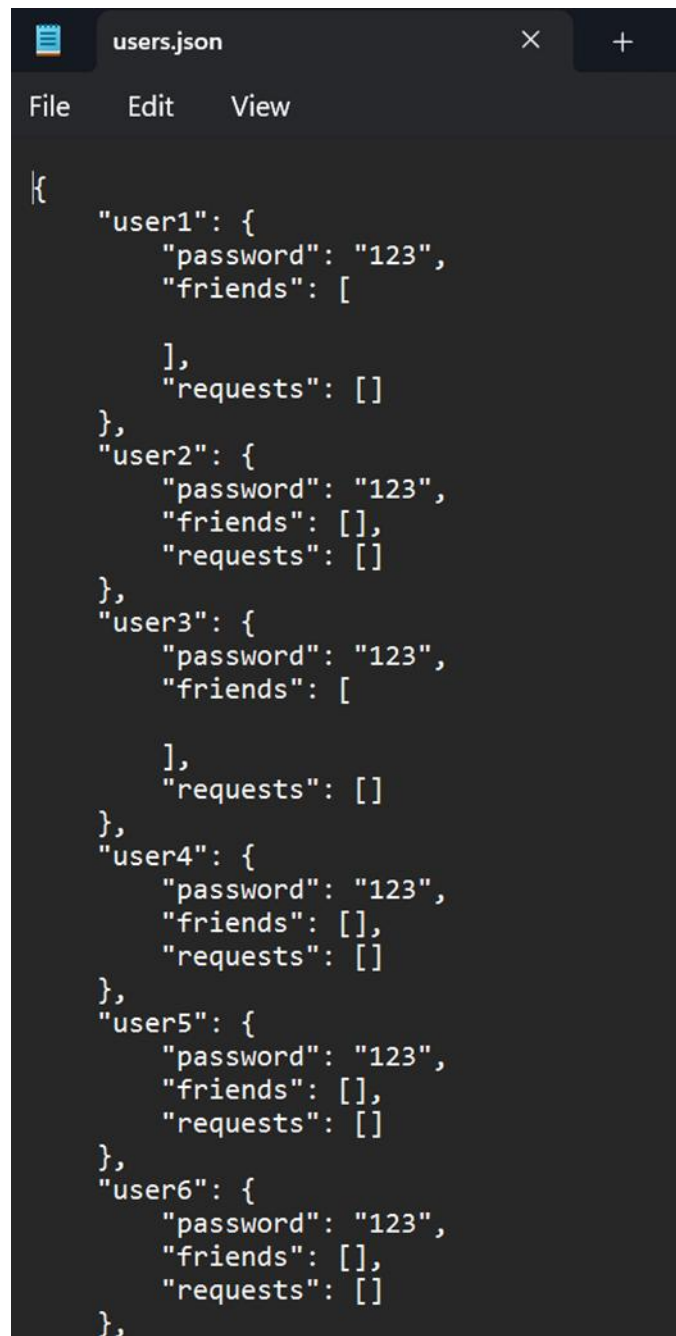
User credentials are stored in the following file:

```
data/user/users.json
```

You may update the credentials in that file or use the provided dummy credentials:

- **Username:** user1
- **Password:** 123

A screenshot of a web browser window titled "InstaNet". The page has a light gray background. In the center, the text "InstaNet" is displayed in a large, bold, black font. Below the title, there are two rounded rectangular input fields. The first field contains the text "user1". The second field contains three asterisks "\*\*\*". Below these fields is a blue rectangular button with the text "Log In" in white.



The image shows a code editor window with a tab labeled 'users.json'. The editor has a menu bar with 'File', 'Edit', and 'View'. The JSON content is as follows:

```
{
  "user1": {
    "password": "123",
    "friends": [

    ],
    "requests": []
  },
  "user2": {
    "password": "123",
    "friends": [],
    "requests": []
  },
  "user3": {
    "password": "123",
    "friends": [

    ],
    "requests": []
  },
  "user4": {
    "password": "123",
    "friends": [],
    "requests": []
  },
  "user5": {
    "password": "123",
    "friends": [],
    "requests": []
  },
  "user6": {
    "password": "123",
    "friends": [],
    "requests": []
  },
}
```

## Tasks:

These tasks are designed to help you understand how server–client communication works using Python sockets, with data exchanged as JSON messages. JSON is a lightweight format used to structure and transmit data (you can open them in notepad to see how they look).

## Summary of Tasks

To completely develop the network part of InstaNet, there are a total of six tasks to complete, as shown below. However, only four of these require new implementation. This is because Task 4 reuses the code from Task 3, and Task 6 reuses the code from Task 5, so no additional work is needed for those two.

Task	Objective	Key Implementation
Task 1: Setting Up the Server	Create a server that handles multiple clients	<code>create_server_socket()</code> using <code>bind()</code> , <code>listen()</code>
Task 2: Connecting the Client	Enable clients to connect to the server	<code>create_client_socket()</code> using <code>connect()</code>
Task 3: Sending and Receiving Friend Requests	Implement JSON-based friend request messaging	<code>send_json_message()</code> , <code>receive_json_message()</code>
Task 4: Implementing Text Private Messaging (DMs)	Reuse JSON messaging for private text chat	Reuse Task 3 functions
Task 5: Sharing Image Posts with Captions	Send images and captions as posts	<code>send_json_message_with_image()</code> , <code>receive_json_message_with_image()</code>
Task 6: Sending Image in DMs	Extend DMs to support images	Reuse Task 5 functions

You'll work with the `socket_utils.py` file to implement key functions for real-time, multi-client communication and request handling.

## Task 1: Setting Up the Server

### Objective:

Create a server that can handle multiple client connections simultaneously.

### Instructions:

- You need to **create** and **bind** a server socket to a host and port.
- Implement the `create_server_socket()` function inside `socket_utils.py`.
- The server should be able to listen for incoming connections and accept multiple clients concurrently.

### Hint:

In Python, you can use `socket.socket()` with `AF_INET` and `SOCK_STREAM` to create a TCP socket. Then use `bind((host, port))`, followed by `listen()` to allow incoming connections.

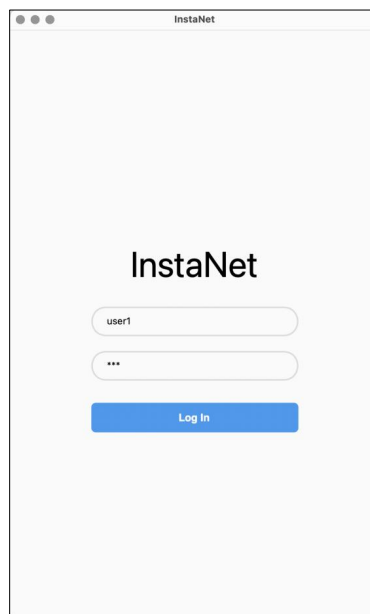
## Task 2: Connecting the Client

### Objective:

Enable a client to successfully connect to the server.

### Instructions:

- You need to **create** and **connect** a client socket to the server.
- Implement the `create_client_socket()` function inside `socket_utils.py`.



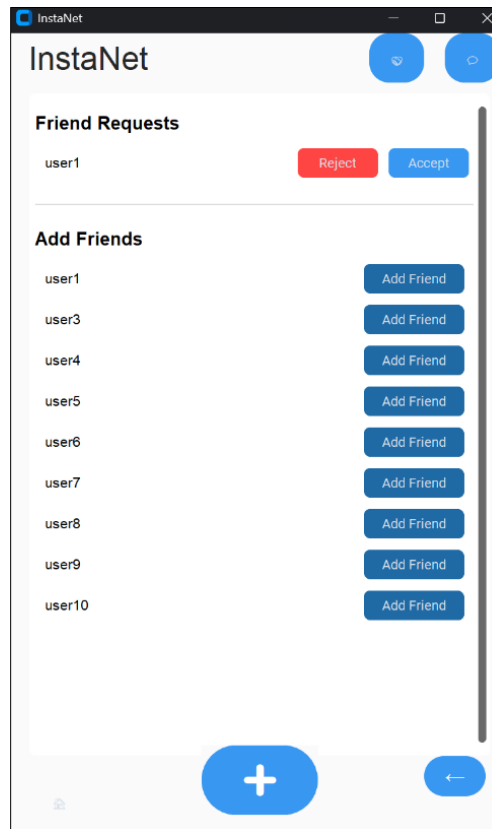
### Hint:

Create a TCP socket, then `connect((server_host, server_port))` to establish the connection. Ensure the host and port match what the server is listening on.

## Task 3: Sending and Receiving Friend Requests

### Objective:

Implement JSON-based friend request messaging



Your application should allow users to send friend requests. These requests are transmitted as JSON messages to the server. The process follows this sequence (user 1 sending a friend request to user 2):

1. The client (user 1) sends a friend request to the server as a JSON message.
2. The server stores the request and forwards it to the intended recipient (user 2).
3. The recipient (user 2) accepts or rejects the request and informs the server through another JSON message.
4. The server updates its database accordingly.

### Important:

The logic for accepting or rejecting requests is already implemented.

Your job is to implement the functions that send and receive the JSON messages.

**Instructions:**

- Implement *send\_json\_message()* in `socket_utils.py` to transmit a dictionary as a JSON-encoded message over a socket.
- Implement *receive\_json\_message()* in `socket_utils.py` to receive and decode a JSON message from a socket.

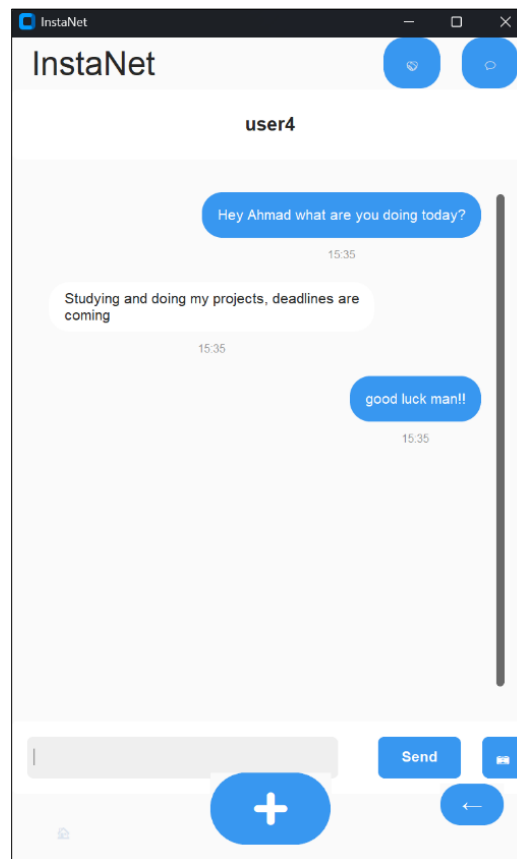
**Hint:**

- To send a message: Think about how you can convert a Python dictionary to a string that can be transmitted over a socket. What encoding format might be needed before sending?
- To receive a message: Consider how to read incoming bytes from a socket and convert them back into a Python object. Also, think about how to deal with incomplete or partial messages during reception.



## Task 4: Implementing Text Private Messaging (DMs)

Your application should support private messages (DMs), where friends can send private text messages to each other. Just like friend requests, these messages are exchanged between clients through the server using JSON.



This is what flow of the program looks like:

- A user sends a private message to a friend (text)
- The client encodes this into a JSON message and sends it to the server.
- The server receives the JSON message and updates the backend accordingly.
- It then forwards the message to the intended recipient.

this is what you implemented in **Task 3**. You're simply reusing the same JSON messaging structure for a different feature (DMs).

### Task:

- Implement (or reuse) *send\_json\_message()* in *socket\_utils.py* to transmit message data as a JSON-encoded socket message.
- Implement (or reuse) *receive\_json\_message()* in *socket\_utils.py* to receive and decode JSON messages sent from the server or client.

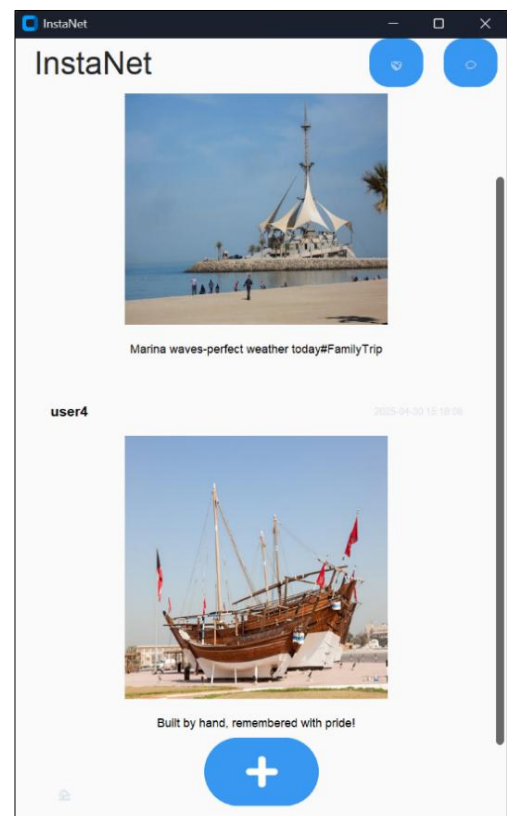
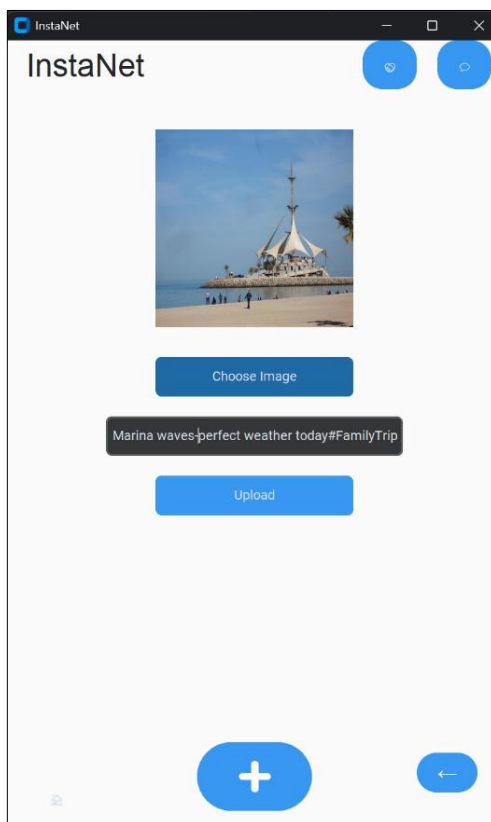
**Hint:**

If you completed Task 3 correctly, you don't need to write any new code here. Your existing implementations of `send_json_message()` and `receive_json_message()` should already support this functionality and enable private messaging out of the box. If they don't, there's likely an error in your code.

## Task 5: Sharing Image Posts with Captions

**Objective:**

Enable users to post images accompanied by captions as part of their activity feed or profile. These image posts must be transmitted from the client to the server using a structured JSON message, which includes both metadata (caption, username, etc.) and the image data.



When a user creates a post that contains an image and a caption, the following steps must occur:

1. The client packages both the caption (text) and the image (in binary or encoded format) into a JSON-compatible structure.
2. The packaged data is sent to the server using a dedicated function.
3. The server receives and decodes the message, stores the post details, and processes it accordingly (e.g., saving the image and metadata to storage).

### Task Requirements:

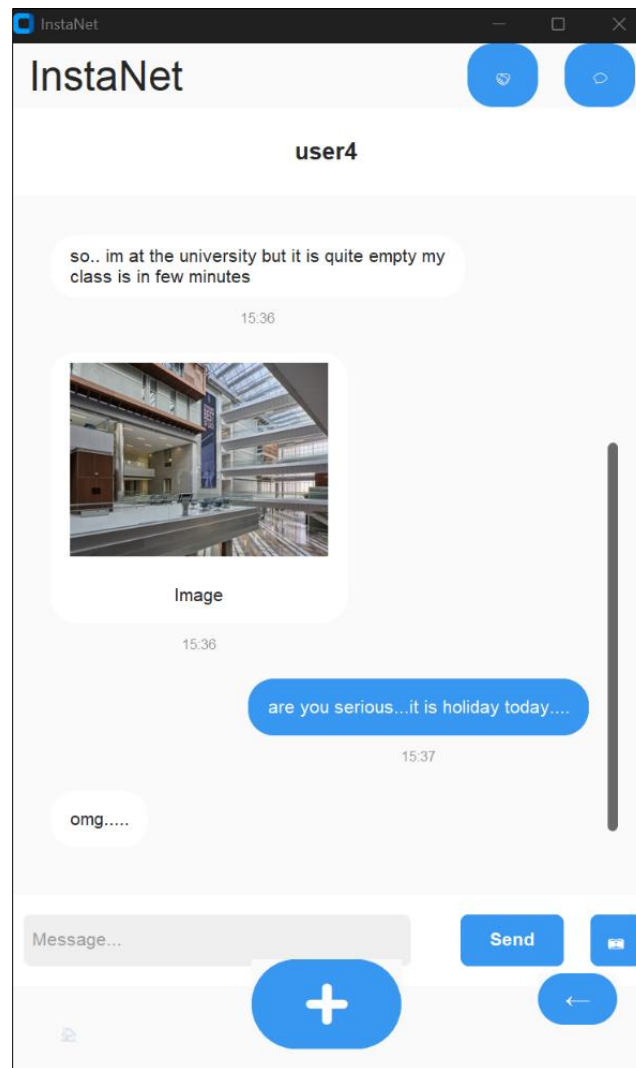
- Implement the function *send\_json\_message\_with\_image()* in the `socket_utils.py` file to transmit JSON messages containing image data along with captions.
- Implement the function *receive\_json\_message\_with\_image()* in the same file to receive and decode such messages from the client.

### Hint:

- To include image data in a JSON message, consider encoding the image using Base64 to convert the binary content into a text-compatible format.
- On the sender side, read the image file, encode it using Base64, and include it along with the caption in a JSON structure.
- On the receiver side, decode the Base64-encoded image data and handle it appropriately (e.g., save it to disk or display it).
- Ensure that the size of the message is manageable, as transmitting large images over sockets may require chunking or buffer handling.

## Task 6: Sending Image in DMs

Improve your private messaging (DM) feature to also support sending private images between users. Just like with posts, image messages should be sent through the server using the same structured format. The flow remains the same- clients package the image, send it to the server, and the server forwards it to the intended recipient.



This is what flow of the program looks like:

- A user sends an image message.
- The client encodes this into a JSON message and sends it to the server.
- The server receives the JSON message and updates the backend accordingly.
- It then forwards the message to the intended recipient.

This is what you implemented in **Task 5**. You're simply reusing the same JSON messaging structure for a different feature (DMs).

### Task:

- Implement (or reuse) *send\_json\_message\_with\_image()* in `socket_utils.py` to transmit message data as a JSON-encoded socket message.
- Implement (or reuse) *receive\_json\_message\_with\_image()* in `socket_utils.py` to receive and decode JSON messages sent from the server or client.

### Hint:

If you completed **Task 5** correctly, you don't need to write any new code here. Your existing implementations of *send\_json\_message\_with\_image()* and *receive\_json\_message\_with\_image()* should already support this functionality and enable private messaging out of the box. If does not there is an error in you code fix it.

## Submission

Submit your complete project code and include your name(s) as comments within the code. Additionally, submit multiple screenshots of your developed application, covering each task.

Your solution should:

- Work on a single computer (localhost).
- Complete all tasks from 1 to 6 successfully.
- Be written entirely inside the provided TODO blocks.

## Demo

You will be required to present and run your code during a demo session. During this session, you may be asked questions about your implementation. Failure to adequately answer these questions may result in a zero for the demo.

The session will last only 10 minutes, so come fully prepared. Detailed instructions and the demo schedule will be posted on Teams; please make sure to review them carefully in advance.