INDIAN INSTITUTE OF TECHNOLOGY MADRAS

सिद्धिर्भवति कर्मजा

# IIT Madras

ONLINE DEGREE

**Modern Application Development - II**
**Professor. Nitin Chandrachoodan**
**Department of Electrical Engineering**
**Indian Institute of Technology, Madras**
**Web Servers**

(Refer Slide Time: 0:10)

Modern Application Developme

Web Servers

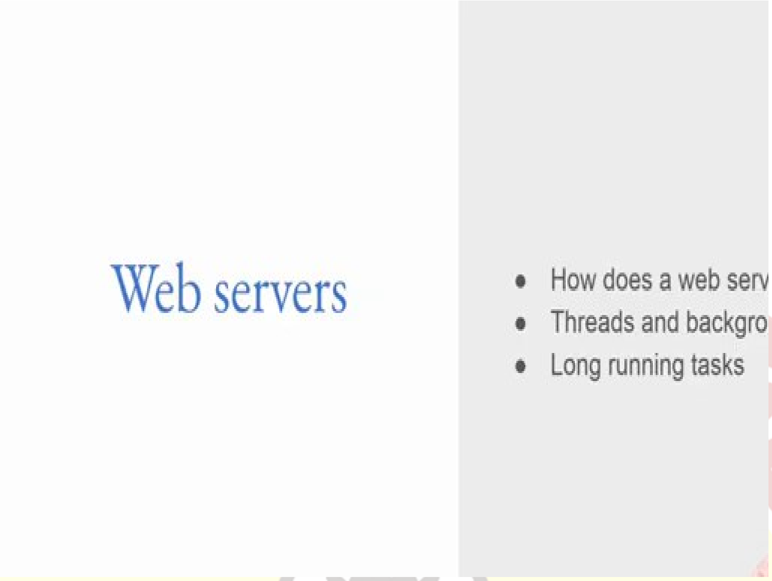
Hello everyone, welcome to Modern Application Development part 2.

(Refer Slide Time: 0:15)

Asynchronous Tasks

Hello everyone, this week we are going to look at the problem of how to schedule, or execute asynchronous tasks, that need to be performed as part of an overall web application.

(Refer Slide Time: 0:30)

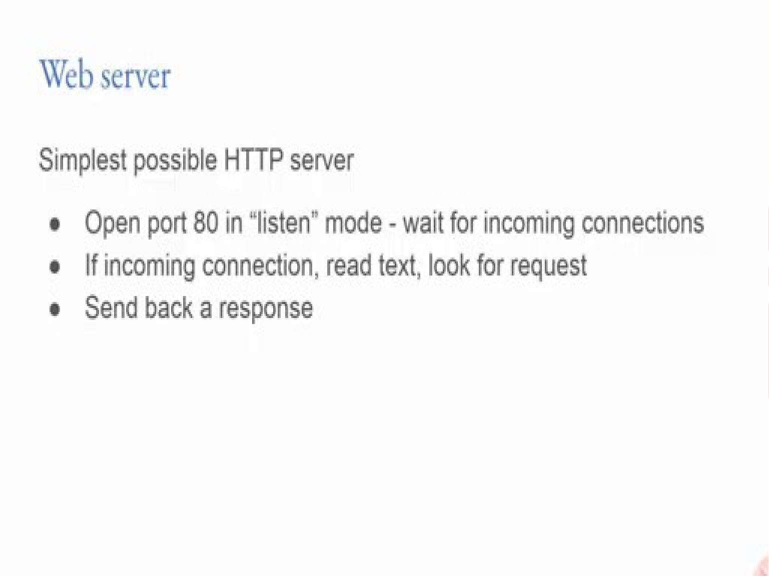


Web servers

- How does a web serv
- Threads and backgro
- Long running tasks

So, before we get into why we need to have a separate way of dealing with asynchronous tasks. Let us first get a slightly more in-depth understanding of how web servers work in the first place. And why we might need to look at different ways by which we need to execute long running tasks.

(Refer Slide Time: 0:48)



Web server

Simplest possible HTTP server

- Open port 80 in "listen" mode - wait for incoming connections
- If incoming connection, read text, look for request
- Send back a response

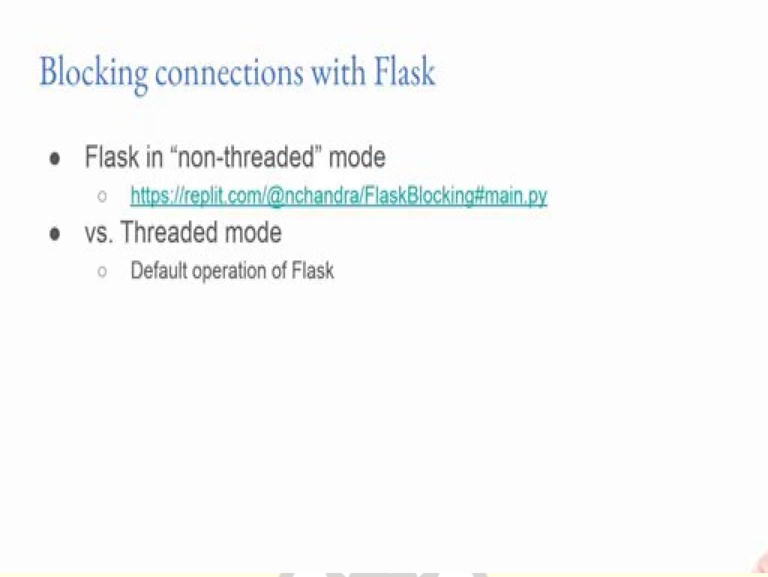
So, what is the simplest possible HTTP server, at the most basic level. And this is something that we had discussed in an early lecture, all that it does is, it opens a TCP listen mode connection, meaning that it is listening on a given port. What does listening mean exactly? For that you need to understand a little bit more about the networking stack, that is present inside the operating system.

The bottom line is that the operating system has some way of receiving packets over some network, whether it is ethernet, or Wi-Fi, or maybe some kind of a protocol over a dial up phone line, it does not matter at the end of the day you have some way by which certain packets internet protocol packets can be received by the operating system, those packets in turn can contain additional headers, that say that they are part of what is called the transmission control protocol TCP.

And there will be something in it which says that, this is a connection that is trying to connect to an application running on a given port, say port number 80, on the server machine. So, what the operating system does is? It listens meaning that, it looks for packets that have that port information and whenever it finds that a packet is received targeted to that port, it will send that packet onto whatever application you have written to listen on port 80, which in our case is a web server.

So, what does the web server do? It basically listens for incoming connections, once it has a connection, it reads the text, that is there in the that has been sent, the data which is expected to be in text format, it reads that information, sees if it makes sense. Is this an appropriate HTTP request, and based on that it generates some kind of a response, which it sends back to whoever was connected to it. So, that in some sense is the simplest possible HTTP server, all that it does is listen on port 80, interpret what it sees as the incoming data and respond appropriately.

(Refer Slide Time: 2:52)



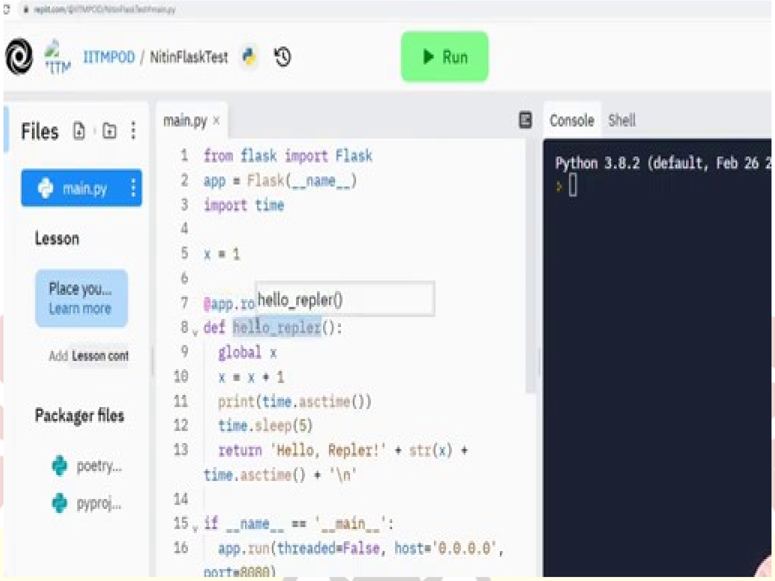
## Blocking connections with Flask

- Flask in "non-threaded" mode
  - https://replit.com/@nchandra/FlaskBlocking#main.py
- vs. Threaded mode
  - Default operation of Flask

So, in our case of course, we have been using flask quite extensively, as a means to implement a very basic form of a web server. And what we mean by that is, we write a python application and the python application behind the scenes is listening on a given port. And what it does is? It basically looks for where the incoming connections are, is it on port 80, or on some other port, it could be 8000, which I believe is the default for flask, or 5000, or something whatever it is, some port number.

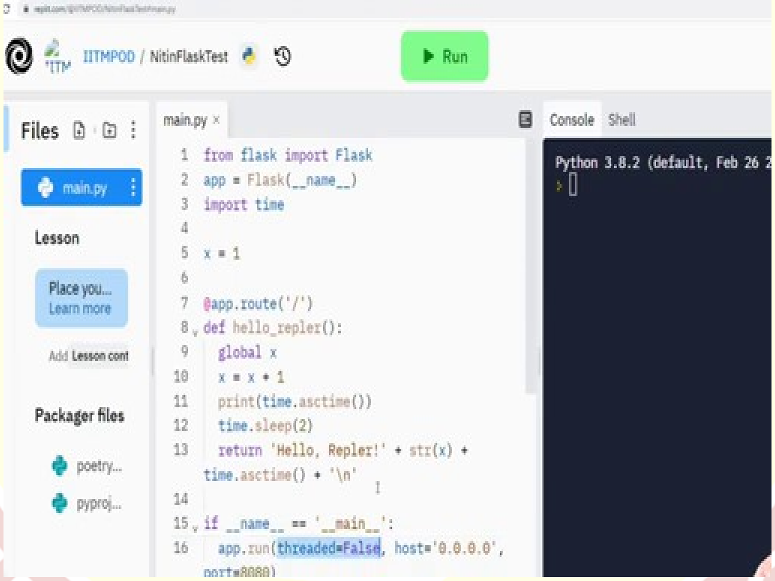
Whatever port it listens on the flask application, then interprets all the requests coming in and response. The response could be as simple as just sending back some text, or it could be rendering a template and sending back the complete HTML for that, or it could even be sending a file. But it is worth sort of looking a little bit more in detail on how that works. So, we will do that by looking at an example of a flask application.

(Refer Slide Time: 3:53)



```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.ro hello_repler()
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(5)
  return 'Hello, Repler!' + str(x) +
time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=False, host='0.0.0.0',
port=8080)
```

```
Python 3.8.2 (default, Feb 26 2
> 
```



```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(2)
  return 'Hello, Repler!' + str(x) +
time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=False, host='0.0.0.0',
port=8080)
```

```
Python 3.8.2 (default, Feb 26 2
> 
```

So, this is a very simple example of a flask application, it is on replit. So, hopefully most of all of you are familiar with how the thing works in general. And what we can see is, what is the code doing it just imports flask, it creates an app. It is also importing 1 more module in python, time.

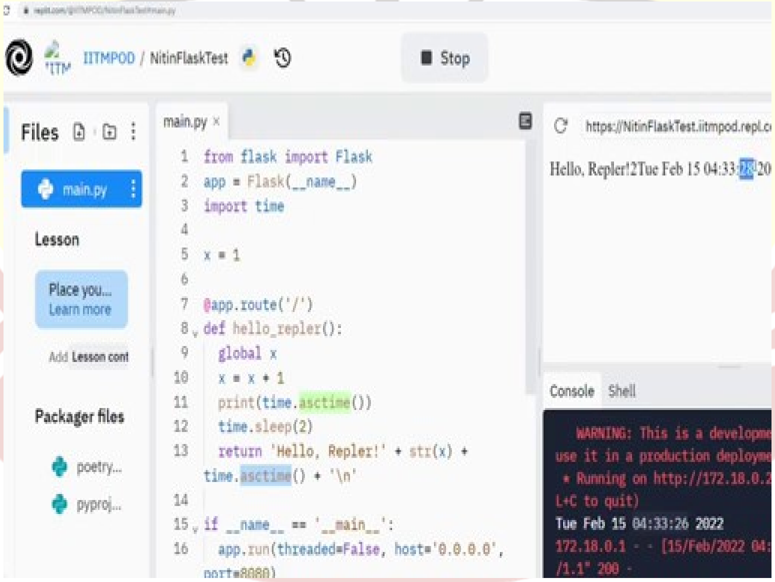
There is a global variable x, which is defined to be 1. And the main part over here is this app dot route, which basically says that any request, that says get slash is going to end up invoking this function. And what does the function do? It first of all make sure that you have access to this global variable x, increments x. This print time dot asc time will not go to the web client, that gets logged on the server side.

But then what it does is? It basically goes to sleep for a short while, let me just change this to 2 instead of 5, 5 seconds of sleep might be too long for a demo. And then it returns this string, hello repler, it converts x to a string and it also adds on the present time. And finally a back slashing, just to sort of convert it into something that gets printed properly on the browser.

So, what happens is, only this last this final string out here is the one that is going to be sent to the browser, it is not a valid HTML document, but on the other hand the browser is perfectly willing to accept just plain text and display it. Now, how am I going to run this? I basically have one piece of code over here, which says if name equal to main, that is to say if I am running this file directly app dot run.

I already created the flask application, host equal to 0, that is listen on all IP addresses, that the system is connected to, port equal to 8080, you could have left it at the defaults, but it does not matter in this case I just specify a port. And this crucial thing that I have over here is something called threaded equal to false. What exactly does that mean? That is something that we need to understand.

(Refer Slide Time: 6:02)



```
1  from flask import Flask
2  app = Flask(__name__)
3  import time
4
5  x = 1
6
7  @app.route('/')
8  def hello_repler():
9    global x
10   x = x + 1
11   print(time.asctime())
12   time.sleep(2)
13   return 'Hello, Repler!' + str(x) +
   time.asctime() + '\n'
14
15 if __name__ == '__main__':
16   app.run(threaded=False, host='0.0.0.0',
   port=8080)
```

Hello, Repler!2Tue Feb 15 04:33:28 20

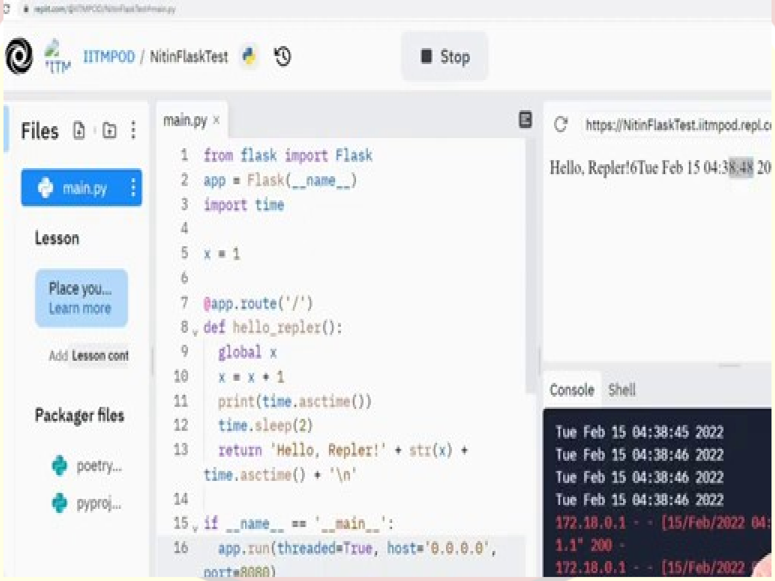
```
  WARNING: This is a developme
use it in a production deployme
 * Running on http://172.18.0.2
L+C to quit)
Tue Feb 15 04:33:26 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
```

So, what I am going to do is? I will run this code. And the first thing it needs to do is go ahead install the dependencies and you can see out here that, this Tuesday Feb 15, this information that has been printed on this line, on the right hand side in the console, that essentially corresponds to this print time dot asc time on line 11 of the code.

In other words, it did not go to the browser, what did go to the browser is displayed on the top right, in the dummy browser, that we have here and it basically says hello repler. It prints the value 2, because this global variable x has now been incremented before printing. So, it now has the value 2. And in addition to that, it basically prints the current time.

Now, look carefully at the numbers over here, what we have is, that the number that was printed on the console was 433 and 26 seconds, whereas what is printed over here is 433 and 28 seconds, of course, this is in some GMT, or some other time zone, so it is not like I am recording this at 4.30 in the morning.

(Refer Slide Time: 7:09)

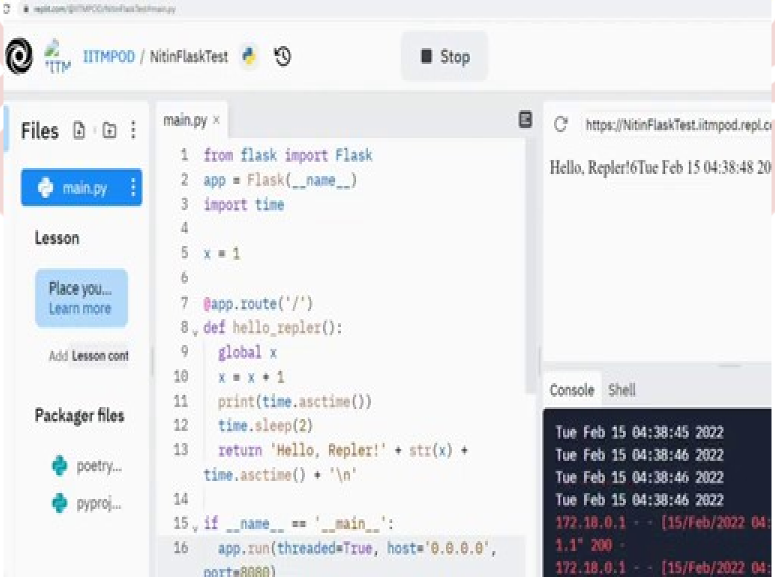


main.py

```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(2)
  return 'Hello, Repler!' + str(x) +
time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=True, host='0.0.0.0',
port=8080)
```

Hello, Repler!6Tue Feb 15 04:38:48 20

```
Tue Feb 15 04:38:45 2022
Tue Feb 15 04:38:46 2022
Tue Feb 15 04:38:46 2022
Tue Feb 15 04:38:46 2022
172.18.0.1 - - [15/Feb/2022 04:
1.1" 200 -
172.18.0.1 - - [15/Feb/2022
```

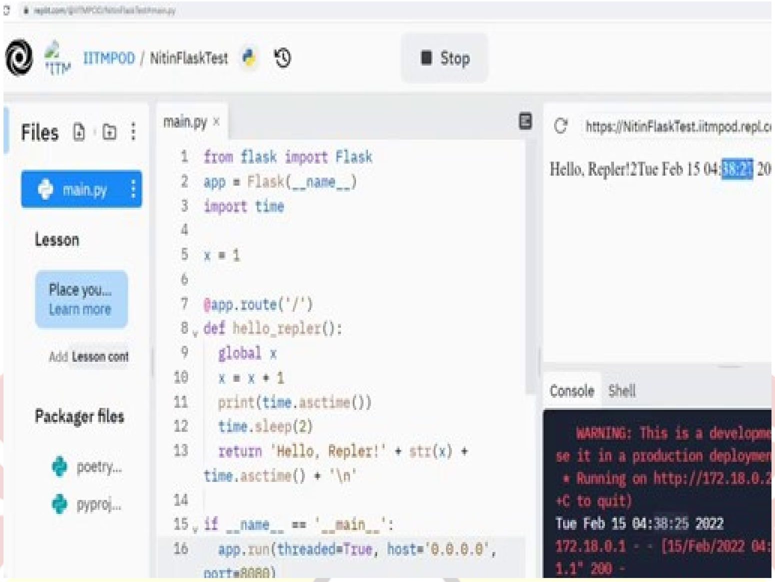


main.py

```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(2)
  return 'Hello, Repler!' + str(x) +
time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=True, host='0.0.0.0',
port=8080)
```

Hello, Repler!6Tue Feb 15 04:38:48 20

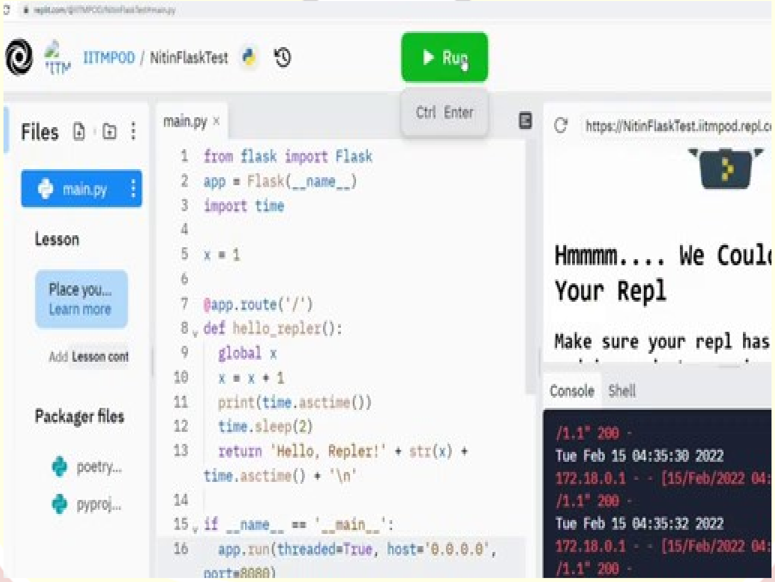
```
Tue Feb 15 04:38:45 2022
Tue Feb 15 04:38:46 2022
Tue Feb 15 04:38:46 2022
Tue Feb 15 04:38:46 2022
172.18.0.1 - - [15/Feb/2022 04:
1.1" 200 -
172.18.0.1 - - [15/Feb/2022 04:
```



```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(2)
  return 'Hello, Repler!' + str(x) + time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=True, host='0.0.0.0', port=8080)
```

Hello, Repler!2Tue Feb 15 04:38:27 20

```
 WARNING: This is a developme
se it in a production deploymen
 * Running on http://172.18.0.2
+C to quit)
Tue Feb 15 04:38:25 2022
172.18.0.1 - - [15/Feb/2022 04:
1.1" 200 -
```

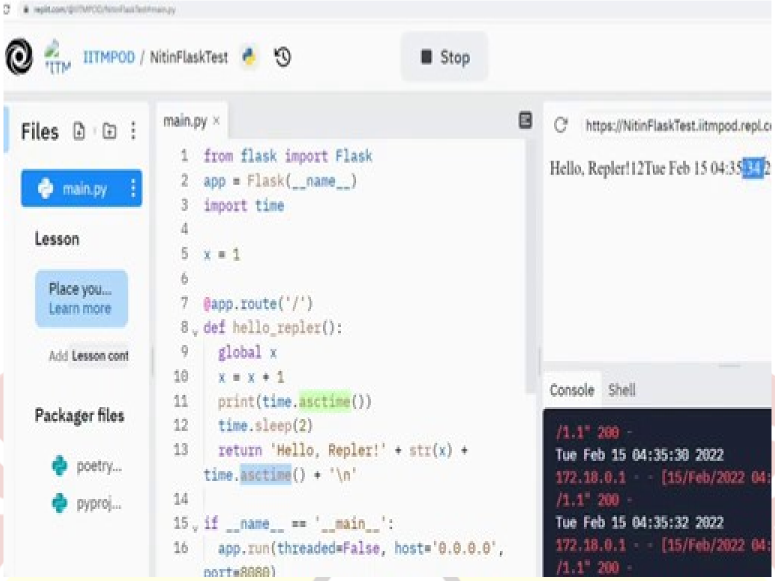


```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(2)
  return 'Hello, Repler!' + str(x) + time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=True, host='0.0.0.0', port=8080)
```

Hmmmm.... We Coul
Your Repl

Make sure your repl has

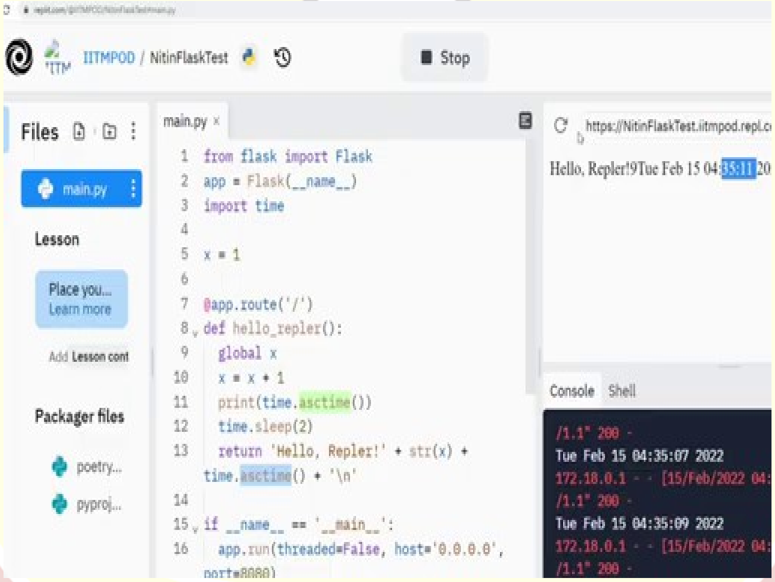
```
/1.1" 200 -
Tue Feb 15 04:35:30 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
Tue Feb 15 04:35:32 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
```



```python
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(2)
  return 'Hello, Repler!' + str(x) + time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=False, host='0.0.0.0', port=8080)
```

```
Hello, Repler!12Tue Feb 15 04:35:34 2
```

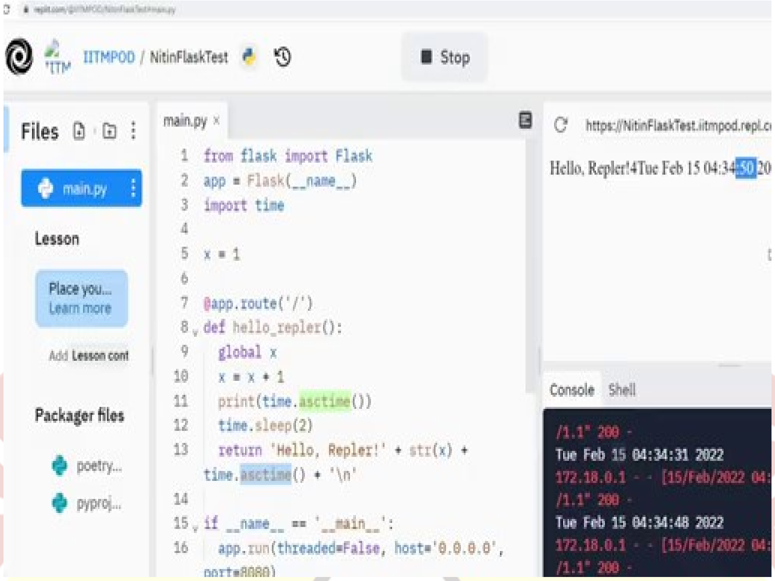
```
/1.1" 200 -
Tue Feb 15 04:35:30 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
Tue Feb 15 04:35:32 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
```



```python
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(2)
  return 'Hello, Repler!' + str(x) + time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=False, host='0.0.0.0', port=8080)
```

```
Hello, Repler!9Tue Feb 15 04:35:11 20
```

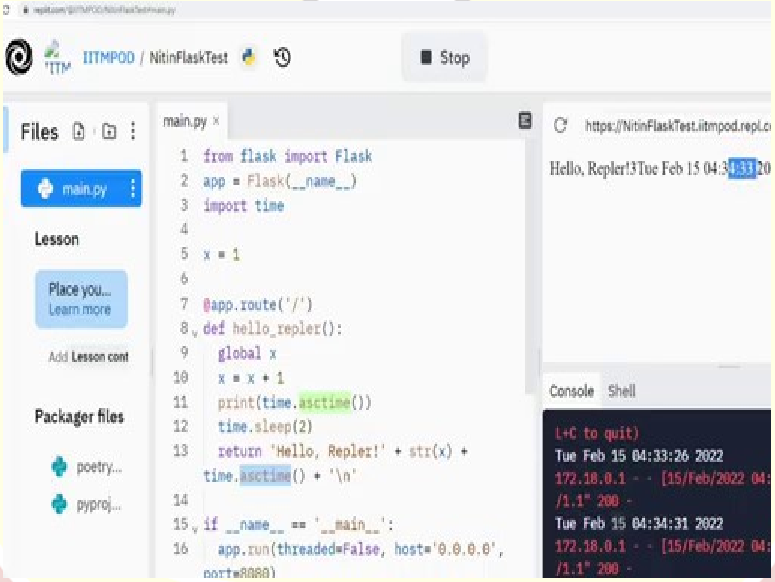
```
/1.1" 200 -
Tue Feb 15 04:35:07 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
Tue Feb 15 04:35:09 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
```



```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
    global x
    x = x + 1
    print(time.asctime())
    time.sleep(2)
    return 'Hello, Repler!' + str(x) + time.asctime() + '\n'

if __name__ == '__main__':
    app.run(threaded=False, host='0.0.0.0', port=8080)
```

```
Hello, Repler!4Tue Feb 15 04:34:50 20
```

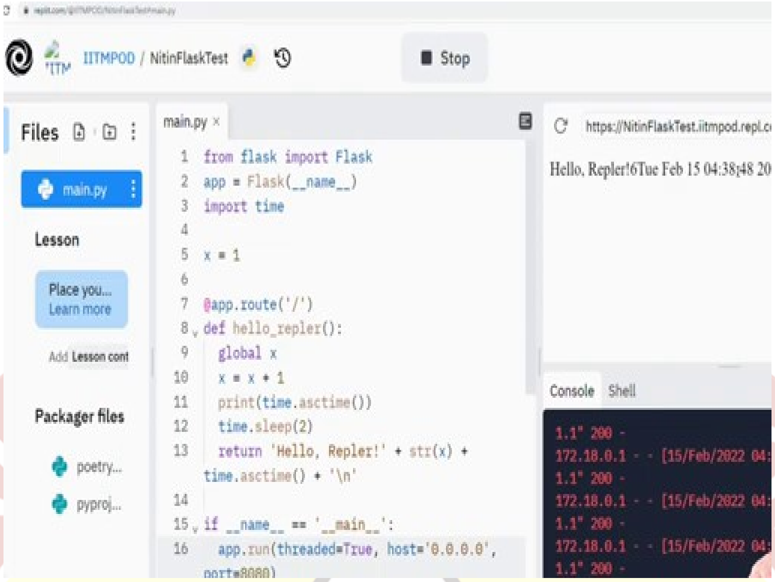
```
/1.1" 200 -
Tue Feb 15 04:34:31 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
Tue Feb 15 04:34:48 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
```



```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
    global x
    x = x + 1
    print(time.asctime())
    time.sleep(2)
    return 'Hello, Repler!' + str(x) + time.asctime() + '\n'

if __name__ == '__main__':
    app.run(threaded=False, host='0.0.0.0', port=8080)
```

```
Hello, Repler!3Tue Feb 15 04:33:33 20
```

```
L+C to quit)
Tue Feb 15 04:33:26 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
Tue Feb 15 04:34:31 2022
172.18.0.1 - - [15/Feb/2022 04:
/1.1" 200 -
```



```
from flask import Flask
app = Flask(__name__)
import time

x = 1

@app.route('/')
def hello_repler():
  global x
  x = x + 1
  print(time.asctime())
  time.sleep(2)
  return 'Hello, Repler!' + str(x) +
time.asctime() + '\n'

if __name__ == '__main__':
  app.run(threaded=True, host='0.0.0.0',
port=8080)
```

https://NitinFlaskTest.iitmpod.repl.c

Hello, Repler!6Tue Feb 15 04:38:48 20

```
1.1" 200 -
172.18.0.1 - - [15/Feb/2022 04:
1.1" 200 -
172.18.0.1 - - [15/Feb/2022 04:
1.1" 200 -
172.18.0.1 - - [15/Feb/2022 04:
1.1" 200 -
```

But now what happens? Let us say I click on refresh and it immediately prints out another number for 434 and 31, and sure enough 2 seconds later the data over here gets updated 434 and 33. And this x over here has got updated to the value 3, we can keep doing this each time immediately it prints out the number 48 over here and then 2 seconds later prints out a number 50 out here.

Now, let us try something which is I am going to click on this several times very quickly, you can see that it prints out once, I have stopped clicking. It then waits over there, it goes through it has got 5, 7, 9, you can see that those numbers have been getting incremented out here and then finally after the 9, only then the value up here got updated to 11. We can try it one more time, I am just going to click thrice in succession. And you can see as it prints 28 down here then 30 and then 32, and 2 seconds after that this updates to 34.

What happened? Basically, what is happening is, even though I clicked on reload several times, what happens each time that I click on reload, is that the browser sends out a request, a get request and that get request is received by the server.

But because I have this threaded equal to false, essentially what ends up happening is the second request coming from the browser is not even received by the server, the server is not ready to listen, which is why down here at the bottom you see that there is a pause, only 2 seconds later after it has sent the response for the first request, it listens to the next request. It then sends a response for the second request. Again is ready to listen to the third request. Then sends the response for the third request.

And at that point the browser has received all three responses, but it is sort of because it was so busy that as soon as it received 1 response, it then sent the next request. It had not bothered to update the screen until the final response came, which is why you see the browser screen updating only one time, it could have updated three times.

But it sort of uses a lazy approach to updating, which means that it update updates only once. Why did all this happen? Because of this threaded equal to false, which basically means that, while the web server is busy responding to one request, it simply cannot accept requests from anyone else. It does not matter how important the request is, it does not matter where it is coming from, whoever is sending the request will find that the web server is not even accepting a connection on that port, or rather it just waits over there, until it has finished whatever it is currently doing.

In this case, because the response over here had that time dot sleep 2, it literally means that the server is just sleeping, it is not doing any computation. But the server goes to sleep, it blocks for this duration 2 seconds, or you could make it any duration longer over here, if you want to see the effect more clearly. And as a result of that during that time no other requests can be answered. This is the default mode in which any kind of system, that listens on a port works.

Having said that you will notice that I explicitly had to add this threaded equal to false out here in the flask code. What happens if I change that? And either make it threaded equal to true, or I could even just remove that statement over there, it does not really matter. What happens when I give threaded equal to true, which is the default by the way for flask. And then I will just stop this and rerun the repl, once again it comes up with this the same kind of behavior, it prints out a number 25 on the bottom right hand side in the console. And the time printed on the top right in the browser is 27, 2 seconds later.

But now let me start clicking multiple times, you will see that each of those clicks responded immediately. And the responses to them. So, when did the clicks get registered with the server at 45, 46, 46, 46, so in other words three requests within 1 second were happily accepted by this web server. It did not say wait I am busy doing something else. But what is it actually doing? It is basically, listen it is accepted four requests in this case, all four of them in parallel go to sleep for 2 seconds.

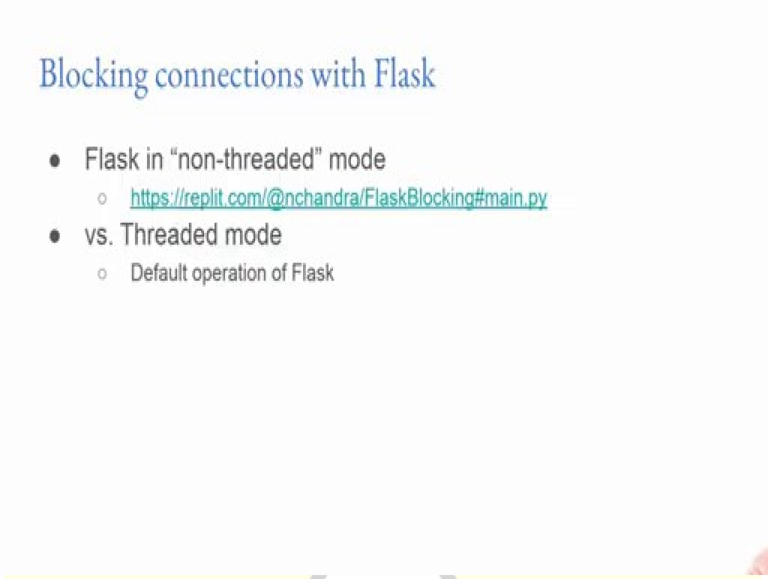
And after all four and as each one of them wakes up, it sends back the corresponding response, which means that as far as the browser on the right hand top is concerned, it has sent four requests, it gets back four responses, it sees which is the last response and prints the time corresponding to that. And you can see that you know the time was exactly 2 seconds later than each of the corresponding requests. There was 1 at 47 and 3 at 48. So, 48 is what gets printed out here.

So, what happened in this case is that the default mode of replit of a flask rather is to run in threaded mode, which means that the flask application is willing to accept requests even while another request is currently pending. It has not yet sent back a response for the first request, but is willing to accept the second request and the third and the fourth.

And all four of them run in parallel, which means that the sleep of 2 seconds that is happening in each one is happening in parallel. And all four responses come back very shortly after each other. There of course when I say parallel, they are still spaced apart by a few micro seconds, or milliseconds maybe. But it is too short for a human being to really make out the difference.

So, this fact that by default a web server will block, it does not work in threaded mode, certain web servers such as flask, apache, nginx and so on, explicitly have some way of handling multiple requests by going into threaded mode. And threaded mode simply means that multiple operating system threads are being launched, multiple processes are being created in order to handle these different requests. So, that is important to keep in mind, why for that we need to go back and look at what our understanding of long running jobs in a web server looks like.

(Refer Slide Time: 13:58)

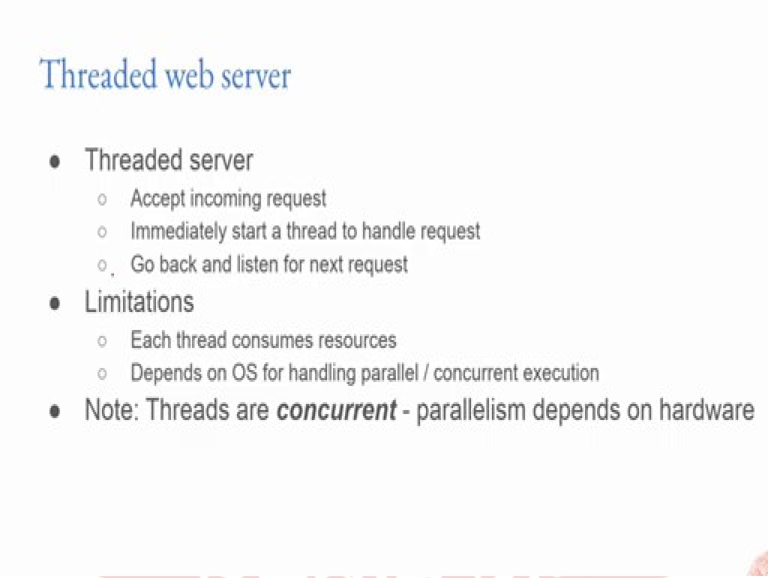


## Blocking connections with Flask

- Flask in “non-threaded” mode
  - https://replit.com/@nchandra/FlaskBlocking#main.py
- vs. Threaded mode
  - Default operation of Flask

So, we saw with the example that flask by default works in threaded mode, which is good. But in non-threaded mode, it would effectively just block until the next request came, until one request had completed before accepting the next request.

(Refer Slide Time: 14:14)



## Threaded web server

- Threaded server
  - Accept incoming request
  - Immediately start a thread to handle request
  - Go back and listen for next request
- Limitations
  - Each thread consumes resources
  - Depends on OS for handling parallel / concurrent execution
- Note: Threads are ***concurrent*** - parallelism depends on hardware

So, a threaded server in general not just flask, it would accept an incoming request, start a new thread to handle that request. And immediately go back and listen for the next request. So, in other words it has one part of the program, the web server program, whose only job is to listen for requests.

Does it actually handle the request, does it take care of sending back the information that is being asked for? No, all it does is hand it off to someone else. It is more or less like the receptionist at a building, or a hospital, all that they do is they ask the person, who you are, who do you want to see, and give you the direction to go.

And it is up to the actual person, or the manager, or the server, or the doctor who is sitting in the specific room, that you visit to take care of whatever your request is about. Which sounds great I mean after all that is precisely what you would like to see. I do not want to have a scenario where one request to a web server makes the entire system hang. There is a limitation of course nothing comes for free.

What happens is that each and every thread that is being created over here, consumes resources, some amount of memory is occupied, some slice of the time, that is available to the processor, now has to be allocated to this thread. So, if I suddenly end up with a situation, where there are let us say thousands of threads that are all alive at the same time, I might find that I am actually running out of memory, or my CPU is not able to jump between all these threads fast enough. And I actually start experiencing a slow down in my system.

The good thing of course is, this naturally lends itself to also splitting across multiple servers. So, at some point if I find that I am not able to handle this number of threads on one server, I already know how I could have sort of split it out among multiple threads on the same server. It just becomes a question of can these threads now run on different servers, do they all need to run on the same server, or can they be also split up among multiple servers.

And of course what happens in a threaded mode is that, it now depends on the OS for handling the concurrent execution, the concurrency of the system. Now, one thing to note over here is, there is as I mentioned at some point before, there is a difference between the notions of concurrency and parallelism.

Concurrent means two things are logically happening at the same time, meaning that they do not depend on each other, they could be processed in either order, either one before the other, or if you have the resources both at the same time, or I might even have a scenario where I can actually slice up the time.

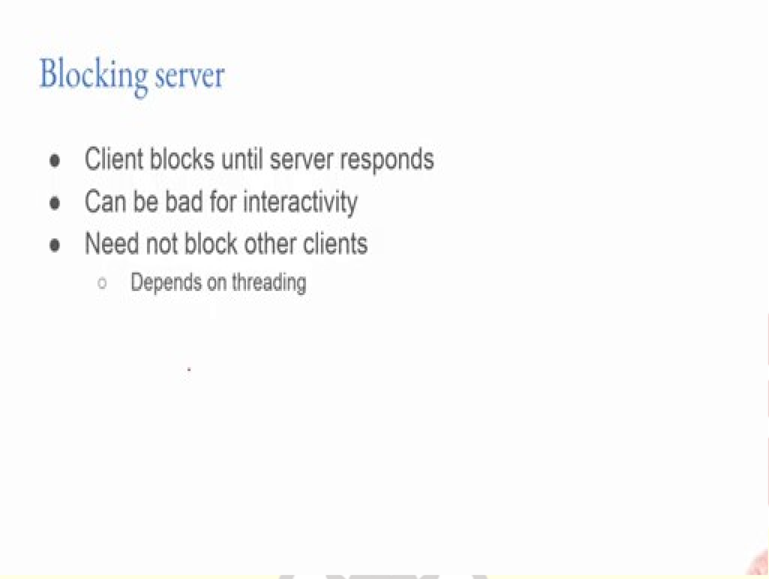
So, finally, do 1 millisecond of work on the first process, jump over and do 1 millisecond of work on the second, then come back and do 1 millisecond of work on the first, again go back to the second and do 1 millisecond of work, that sounds like a lot of jumping back and forth, I mean thousand times a second, remember that a CPU is running at a clock speed of 1 gigahertz, which means that in 1 millisecond it can actually execute 1 million clock cycles, 1 million instructions, or possibly even more in some cases.

So, that time multiplexing, where multiple thing you know the CPU jumps between two processes very rapidly. It can be done in such a way as to completely fool human beings into thinking, that they are happening at the same time. All that that means is, these threads are operating concurrently.

Parallelism depends on something else. Parallelism actually explicitly means there are two processors, one is executing one task, another is executing another task, which means that it is not just logical concurrency, but physical parallelism, that we are talking about over there. So, if you have something that is concurrent, and if you have multiple processors you will probably be able to benefit by using parallelism.

So, that is something to keep in mind, threads only imply concurrency, they do not automatically imply parallelism. Why is that important? Because that allows you to understand how you can generate thousands of threads, even when you do not have thousands of cores in your system, even on a laptop, that has only let us say four cores in your CPU, it is very easy to launch thousands of threads. And threaded web servers therefore are very powerful in terms of how they can handle large incoming, large numbers of incoming requests.

(Refer Slide Time: 18:53)



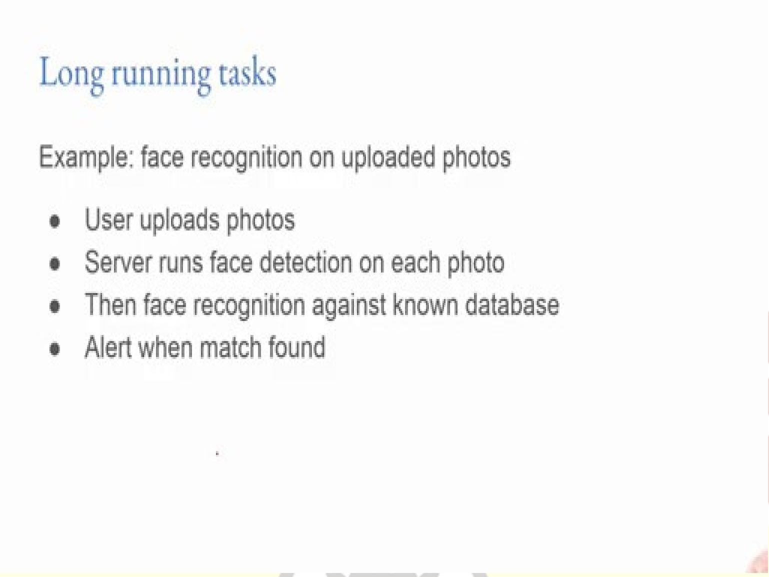
## Blocking server

- Client blocks until server responds
- Can be bad for interactivity
- Need not block other clients
    - Depends on threading

Now, a blocking server, on the other hand as opposed to a threaded server, basically what happens is that, a client will block until the server responds. And in general this is very poor for interactivity, you make a request, fine you might be willing to wait until a response comes.

But what it also means is? Everyone else around you has to wait until your response has come back, before they get a chance. It is like going to a bank, where there is only one person handling the cash counter, everyone has to wait. On the other hand, if you had like four cash counters, then that is great, I mean the moment that you go and stand in the queue, even if your request is taking a long time to process, the others could still go ahead and meet other people and get their work done.

(Refer Slide Time: 19:39)



## Long running tasks

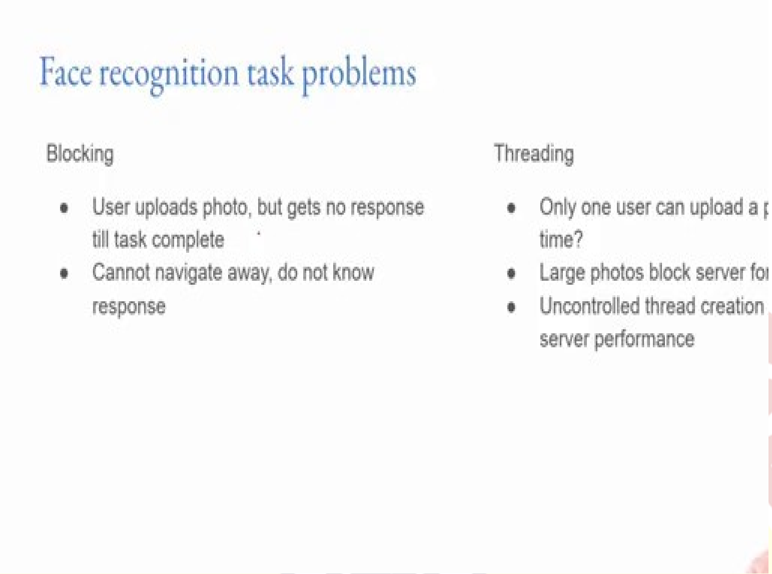
Example: face recognition on uploaded photos

- User uploads photos
- Server runs face detection on each photo
- Then face recognition against known database
- Alert when match found

Now, especially where this becomes a problem is, let us say that you have some kind of long running tasks. So, what do we mean by a long running task? Imagine that we have an application, where you can upload photographs. And it would automatically scan through the photo detect faces of people, and maybe do a face recognition against some known database.

It could be some kind of a fun application maybe just google photos, where it tries to find out who all your friends are, or it could be a database of criminal faces and used by the police, or the international police organizations. Now, it is supposed to run face detection on each photo, run it match against your known database and alert you when a face has been found.

(Refer Slide Time: 20:24)



Face recognition task problems

Blocking

- User uploads photo, but gets no response till task complete
- Cannot navigate away, do not know response

Threading

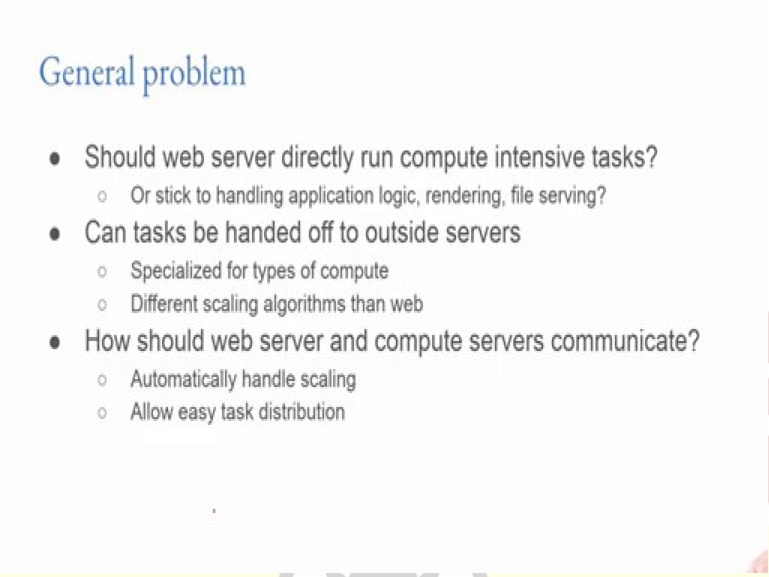
- Only one user can upload a p time?
- Large photos block server fo
- Uncontrolled thread creation server performance

The just imagine that this works in a blocking mode, what will happen is? The user uploads a photo, the server promptly starts its work. But the user does not even get a response until the task has been completed, you cannot navigate away, you cannot know what the response is, you just have to sit and wait over there probably with a loading screen, or something of that sort spinning away in your browser, waiting for a response from the server. But you do not even know how long it is going to take, there is absolutely no interactivity.

Now, imagine on top of that, that it was not even threaded, what happens is? Only one user can upload a photo at a time. And if you upload a large photo, the server just blocks for a long time. And of course what we could have is? If you do have it threaded, and a large number of people suddenly start uploading photos all at the same time, that can bring down the server, you run out of memory, you run out of CPU time and your server becomes slow and sluggish and not able to give responses quickly.

So, all kinds of problems can erase because of this. Why did this happen? Because the face recognition task was computationally intensive, it took a lot of time.

(Refer Slide Time: 21:39)



General problem

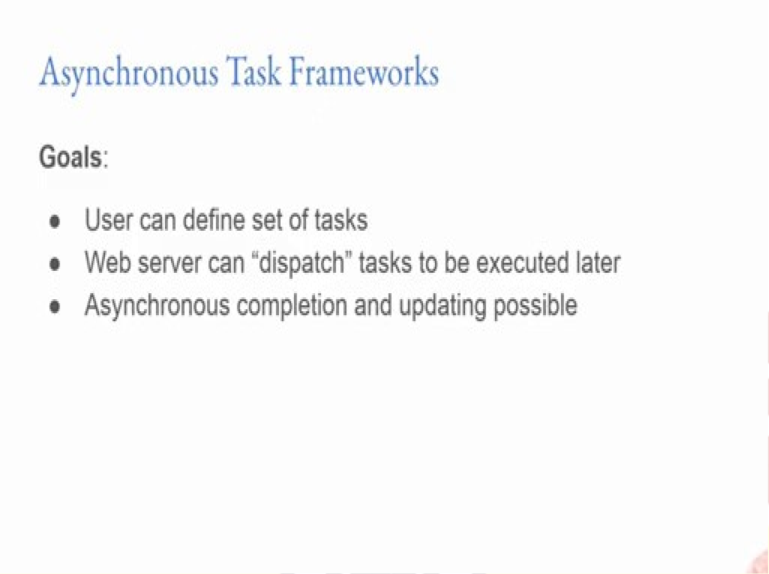
- Should web server directly run compute intensive tasks?
  - Or stick to handling application logic, rendering, file serving?
- Can tasks be handed off to outside servers
  - Specialized for types of compute
  - Different scaling algorithms than web
- How should web server and compute servers communicate?
  - Automatically handle scaling
  - Allow easy task distribution

So, it is a general problem should a web server directly run compute intensive tasks, that is the question that we are trying to answer here. Alternatively, what could the server do, like the receptionist at a hotel, or a hospital. The server just sticks to handle handling the application logic. And what it does is? It hands off any more complex tasks to other servers, whose job is to specialize for specific types of computation, it could be image recognition, face recognition.

But there is a server with maybe large amounts of memory, or maybe even a hardware coprocessor, whose job is just to do face recognition fast. And those other servers could scale differently than the front end, meaning that let us say I suddenly find that I have a flood of requests a lot of new photos being uploaded, do I really need to add more front-end servers, probably not I could just have one threaded server at the front end whose only job is to send back the HTML, or as and when it gets a photo being posted, it takes that photo copies it into some memory location, or into a file and hands off to someone else for their work.

So, this essentially brings us to this whole notion of having a task server, or something that can handle tasks, that is separate from the web server. And of course there are multiple questions that need to be answered over here, one of them is how should this web server the front end and these compute servers communicate, how do they coordinate their activities. And how can we set this up in such a way that we can automatically handle scaling, or distribution of the tasks to different servers.

(Refer Slide Time: 23:34)



## Asynchronous Task Frameworks

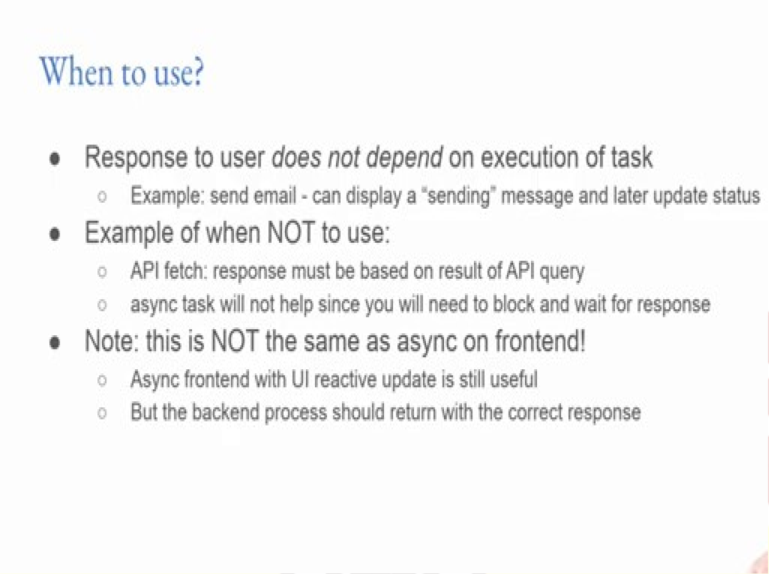
Goals:

- User can define set of tasks
- Web server can “dispatch” tasks to be executed later
- Asynchronous completion and updating possible

This is where the whole idea of asynchronous task frameworks comes in. The goals that we are trying to satisfy over here is that, the user should be able to define a set of tasks. It could be that a task is a single function, that basically takes an image as input, recognizes faces on it and gives back data.

But since the task will take a long time, the web server should be able to just dispatch the tasks, send off the information saying you know get this done and come back later. Which means that we are interested in asynchronous completion. And the updating of the results should be possible later.

(Refer Slide Time: 24:16)



## When to use?

- Response to user *does not depend* on execution of task
  - Example: send email - can display a "sending" message and later update status
- Example of when NOT to use:
  - API fetch: response must be based on result of API query
  - async task will not help since you will need to block and wait for response
- Note: this is NOT the same as async on frontend!
  - Async frontend with UI reactive update is still useful
  - But the backend process should return with the correct response

Now, what are scenarios in which you should use something of this sort. Primarily, it would be something, where the response to the user does not depend on the execution of the task. So, for example let us say that I click on a button and it says you know I am going to send you an email, confirming that you have some submitted a form, or done something.

Now, I just have to display that on the screen, I just display a message on the screen saying that, yes an email has been sent. But that does not mean the email has already actually been dispatched. The email just goes and sits in a queue and maybe the SMTP server, then takes its own time to actually figure out how to send it, where to send it, takes care of some other things in the background.

But the web browser the client just sees a message saying an email has been sent, later on they might see an update saying this transaction failed, you know the email could not actually be sent. But because in most cases, we expect the email to go through, you can probably safely say an email has been sent, go check your inbox. And if you did not receive it click here again.

On the other hand, let us say that you have something where you actually want the result of an API fetch command to update something on a screen. In such a scenario it probably does not make sense to use this kind of task, because you literally have to wait for the data and without that you cannot update what is on the screen.

So, asynchronous tasks will actually not help over here, because even though you might have sent off an asynchronous fetch instruction, you have to wait for the data, get the response and only then can you move on, you might of course you can still use threads, the question is can you use a completely asynchronous task to which you can just sort of hand off information and wait for the result.

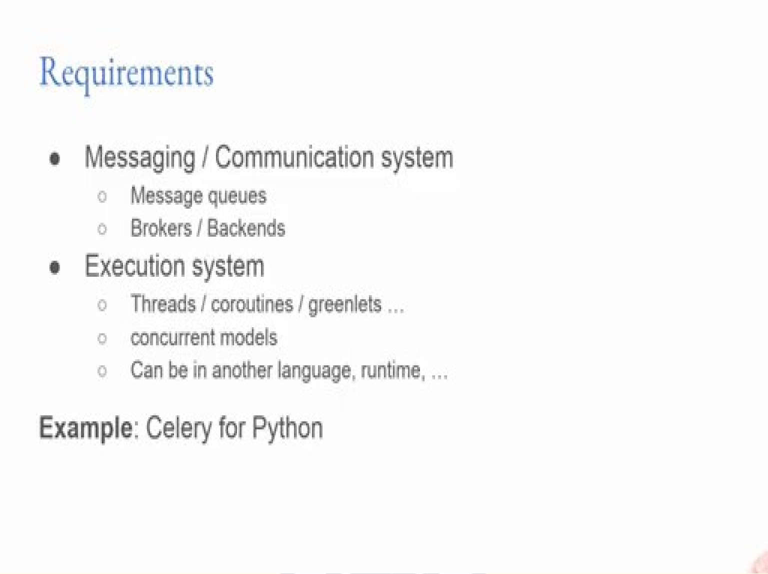
Now, one very important thing to keep in mind over here, which is worth emphasizing is that this entire notion of asynchronous tasks, that I am talking about is not the same as the use of async on the front end, by now we are all familiar with frameworks like VUE and other java script based frameworks, where you can perform asynchronous operations.

So, there are async functions and there is this notion of binding with VUE, where I can allow an async function to eventually update the data on the screen, as and when the data from the async function comes back reactively the screen will update, that is not exactly what we are talking about over here, that could of course be useful in other contexts.

So, an async function of that sort is useful when I do an API fetch. The API fetch takes a certain amount of time and then it updates what is on the screen. The async tasks that we are talking about over here are potentially even longer running than that. In an async UI update I still expect the data to come back reasonably fast, it is just that I want that interactivity therefore, I sort of let the fetch API call happen asynchronously and to update the screen later.

But that is not still as slow or as intensive as an image recognition task. This is something the asynchronous tasks, that we are talking about over here are things that are running on the back end. So, in other words where we are in looking at over here is when the back end needs to execute asynchronous tasks, and those tasks may then need to be collected their results need to be sort of waited for and observed and collected later.

(Refer Slide Time: 27:54)



## Requirements

- Messaging / Communication system
  - Message queues
  - Brokers / Backends
- Execution system
  - Threads / coroutines / greenlets ...
  - concurrent models
  - Can be in another language, runtime, ...

**Example**: Celery for Python

So, as you can imagine, we are talking about multiple servers, there is a web front end and there are other servers, whose job is to perform computation. They need to communicate with each other. So, some way by which they can sort of transfer messages between each other becomes important. And over here this whole notion of what are message queues and some kind of brokers, which are responsible for transferring messages between a requester and servers, that are actually capable of performing the tasks.

And of course the results come back from those servers, how are they collected and got back to the main backend server. And also how do we handle the entire execution system? Should we be using threads? Should we are there are other techniques called coroutines and greenlets, that are not as heavy, or as impactful on performance as launching a new operating system thread, they are lighter, they are a bit more easy to set up and tear down. But ultimately what happens is? You are trying to bring in some notion of concurrency.

And the important part as far as the execution system is concerned is, the part that is taking care of the actual tasks for you, need not even be in the same language as the actual server, that you are writing. So, in other words you could have your server in flask, that is in python.

And the computationally intensive tasks might have been written in C++, or in java, or in Go, or any other language, that is suited to performing the kind of tasks, that you need. One particular example of such a task management system is what is called celery for python, which we will be looking at in a little bit more detail later.