

**IIT Madras**  
ONLINE DEGREE

**Modern Application Development - II**  
**Professor. Nitin Chandrachoodan**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Madras**  
**Asynchronous Tasks with Celery**

Hello everyone. Welcome to Modern Application Development - Part 2. We are now going to look at some way of executing asynchronous tasks and in particular we will be looking at Celery which is a library or a module that is available for Python that allows us to do this in a systematic manner.

(Refer Slide Time: 00:29)

**Task Queues**

How can we manage large numbers of long running tasks without interfering with ability to respond to user requests

- User request handler "pushes" task onto a queue
  - First-in-first-out: give priority in order that tasks issued, or can have separate priorities
- Separate queue managers to handle execution of tasks and returning results

Asynchronous: in general no guarantees of timely response.

So, the first question is, what exactly do we mean by a task queue? We are looking at something where we can manage large numbers of long running tasks, without interfering with the ability of the web server to manage user requests or respond to user requests. So, the user request handler, in other words, should be able to push tasks into a queue.

So, what exactly do I mean by a user request handler this is something that is actually receiving the incoming web requests, the http request from clients. And all that it should really need to do, should be to just push a task onto a queue, give priority in the order that the tasks were issued, or alternatively you might even have some other priority mechanism.

You might have a scenario where even though it tries to maintain first in first out to the extent possible. There is some other high priority queue, which means that when an urgent message comes in that is actually sent in on a different channel and therefore gets processed even before the regular messages.

Now, the user request handler may not always be the web front end. It might be your flask application, in turn needs to send out an email message. And what it does is, rather than actually trying to send out the email message as part of the flask application, it triggers off a task that could actually handle sending out the message.

So, we are sort of once again, separation of concerns. There is a queue manager that handles the execution of these tasks and returning results. And one of the things that we can understand from all of the discussion so far is that we need to have some kind of asynchronous operation happening out here. In general, there is no guarantee of a timely response. We want to be able to handle this scenario in a way that does not require the complete execution of the task before returning immediately.

(Refer Slide Time: 02:24)

**Task Queues**

How can we manage large numbers of long running tasks without interfering with ability to respond to user requests

- User request handler "pushes" task onto a queue
  - First-in-first-out: give priority in order that tasks issued, or can have separate priorities
- Separate queue managers to handle execution of tasks and returning results

Asynchronous: in general no guarantees of timely response.

So, asynchronous task execution. You could have some mechanisms that are just language supported. So, Python does have some notion of async io. Asynchronous input output and of course, we are familiar with Java script, the async await kind of coding style. Now, what we are interested in over here is, there are a few different things that we want to be able to add on things like a guarantee of completion.

Now, Python async io if I directly write code in that, I do not normally get something of that sort, all that I can say is, there will be a request, it will be done asynchronously. But handling things like should the request be retried, what happens if a server fails? Will it automatically try for a certain number of times? Unless you have an API, that sort of takes care of that kind of functionality for you and certain things like the fetch APIs could have timeouts and could have certain additional things that are capable of doing it.

Normally this kind of behavior should be handled by a separate runtime and what I mean by a separate runtime is it is not just something which is a function that you would normally call. It is actually something that has its own way of handling tasks and can take care of reliably executing and getting back with results, after a certain amount of time even in the face of potential failures. So, these are all things that you are looking for, in something like a task queue management system.

(Refer Slide Time: 04:00)

**General Principles**

- Pushing a task onto a queue should be faster than executing the task
  - Else not worth using a queue - just finish the task
- There should be enough worker resources to empty the queue eventually
  - Else build up backlog and eventually overflow

So, a couple of general principles. When you are building some kind of a queue management system or a task queuing system, the first and foremost is, the time required for pushing a task onto a queue must be very less. It should be very quick and efficient to put a task onto a queue; in particular, it should be faster and simpler than actually just executing the task.

So, in other words, if you try and create a task whose job is to add 3 plus 4 and return the result, you are certainly going to end up taking longer in actually pushing the task onto a queue, then having some other server pull out the task from the queue, execute it and get back with the result. Then it would have taken to just add 3 plus 4 and return the result in your primary server itself.

In the same way, let us say that I am actually just trying to do a crud operation on a database. I need to create a new entity corresponding to let us say a new book, in a book management system. I am probably better off just going ahead and creating the entry directly in the database, as part of the crud operation in the flask and returning.

But, on the other hand, let us say that I have some mechanism where I create a new book and as a result of that I then need to go and send out alerts to a whole number of people who are interested in the genre of books. It is maybe, it is the latest in the Harry Potter series, the book was released I need to send out emails to all the people who were interested and who had maybe either placed pre-orders or wanted to be alerted the moment the book came out.

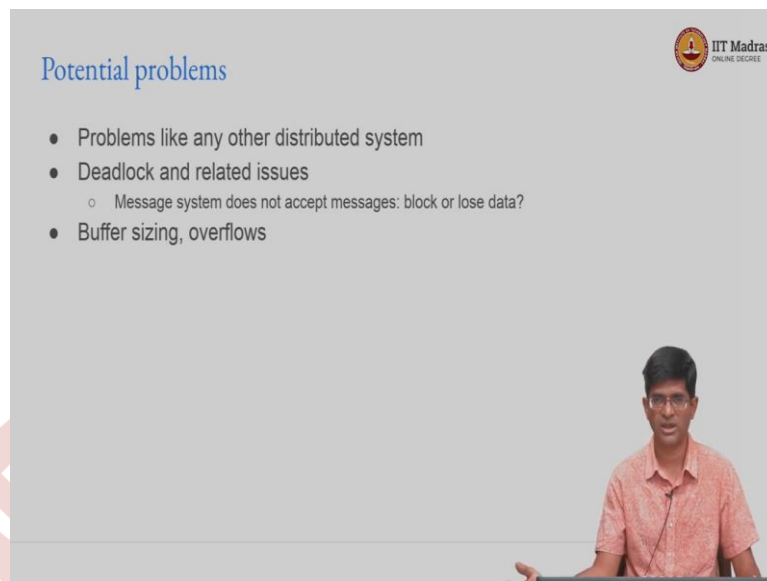
Now, I create the book in my database, should I also be sending out all of these emails immediately? Almost certainly the answer is no. Because sending out those emails, in fact even just updating the database, saying all these people need to be informed about it, will probably take a long time.

So, how do you handle this? Some part of it. It becomes a question of balance. some part of it needs to be done online, meaning that as soon as the request comes in, you create the entry in the database, but then you fire off a bunch of tasks, saying update all the other people, push tasks to send out emails to a whole bunch of other people. Maybe go and update other things such as the list of best-selling books or the latest things in the library, all of those information now need to get updated.

All of those rather than having them in the main code, you push them to be updated separately. And now, the other thing of course is that you are pushing things onto a queue, you need to make sure that you are not just building up a backlog. The queue should be something that will empty eventually.

Unless you have workers that can actually pull out tasks from the queue, execute them and clear up any backlogs that might be coming along, there is no point in really having some kind of a queuing system like this. So, these are a couple of general principles but very important to keep in mind, when you are deciding, even whether to use a queue for a system.

(Refer Slide Time: 07:18)



Potential problems

- Problems like any other distributed system
- Deadlock and related issues
  - Message system does not accept messages: block or lose data?
- Buffer sizing, overflows

Of course, there are a bunch of potential problems that can crop up, when you use queuing systems like this. It is, these I am not going into detail over here, because to really understand them you need to get into the design of distributed systems. But like any other distributed system, there are a bunch of problems. One of the most common terms that you can think of or might be aware of is this thing called deadlock.

Now, what happens in deadlock is that you could end up in a scenario where you are, essentially, some system loop has formed. So, there is a server A that is depending on some output from server B. It has sent out a request to server B, but server B is not able to perform its functionality, because it is in turn waiting for something else to either clear up. Maybe the queue of data that has been sent back to A, has not got cleared up, or something else has happened, but B has essentially got stuck.

The result is A and B are both sort of waiting for each other and the system has become deadlocked there is no way out of this. It is, this usually results from some, in some ways, it could be just a question of bad design of the system. But deadlock is not a very trivial thing to catch either, it actually requires fairly careful system level design.

So, you need to be able to get hold of, why it could have happened and make sure that you have designed in such a way that you are able to get past it. There are also issues; after all, you are using some kind of buffers or cues in order to communicate between the different servers. How big should the buffer be? How much data should it be capable of accepting?



Is there a limit on the number of messages that can be put into a queue? Is there a limit on the size of an individual message? What if I am trying to post messages that actually each message has a large image that needs to be processed? What if one of the images is too large and does not fit in the buffer? So, you could have those kind of scenarios where essentially, your buffer or the queue is not adequate for what you want.

The other is, I am generating requests too fast and after a while my buffer fills up. I do not have sufficient workers to drain out the queue quickly. After a while, when additional requests come, I need to drop them. So, I have buffer overflow, I start losing information. So, these are potential problems that can arise in the implementation of such systems.

As I said, it is not unique to web applications; this is true of any kind of distributed system. So, in order to truly understand, what is happening over there, you need some advanced knowledge of or rather at least pay attention to principles of distributed system design as well.

(Refer Slide Time: 10:06)

Scenario: Push Queue

- Client pushes task onto server queue
- Server *should* start operation "immediately"
  - May be delayed based on availability of resources etc
- Closer to "real-time" operation
- Example:
  - Update friend list in social media application: push updates to DB for all friends
  - Send emails: push emails onto queue to be sent out individually

There are a couple of interesting scenarios that we can look at over here. which again, so, Google's app engine gives these as different examples of the way that queues can be implemented. One of them is what is called a Push Queue. And what happens in a push queue is, basically the client pushes a task onto the server queue and the expectation is that the server should logically, at least, start the operation immediately.

Of course, it may be delayed based on the availability of resources, but in general the idea is that as soon as I put something onto a queue, if the server is not doing anything else, it should

immediately start working on that task. Now, examples of this could be that let us say, I add a new friend in a social media application, it should go and update the database and there should be updates for in multiple places probably..

It could be of course replication of the database, but it could also be that the number of friends or the network of friends or who are the second order friends of a given person, all of this also needs to get updated. And in general what I would expect is that since I have pushed in that request it would all get processed reasonably, quickly.

Another example is that I want to send out emails. So, some new book has been registered with a book management system, I want to send out books emails to everyone who subscribed to similar kinds of books. So, what do I do? I push the emails onto a queue and the expectation is that as and when the email server is free, it should immediately be sending out those messages. So, I am just pushing the message out there.

(Refer Slide Time: 11:44)

**Scenario: Pull Queue**

- Clients can push tasks at any time
- Server "polls" queue at regular intervals
- Better suited to "batch-mode" operation
  - Generally not real-time
- Example:
  - Batch update of high scores in gaming server: periodic updates
  - Dashboard updates - process many log entries in batch and update periodically

There is another alternative where I could think of something called a Pull Queue. And over here, what would happen is, somebody puts messages onto a queue, but the server does not guarantee that it is immediately going to process them. What it will do is it will pull the queue. Pull means that it would just look for information that is present on the queue, but it may do it only once in a while, at some regular intervals.

Now, this is very well suited for, what is called batch mode operation. What happens over there is that let us say, you have high scores for some games that are being played by multiple different people and I need to maintain a leaderboard. As and when the games get completed,

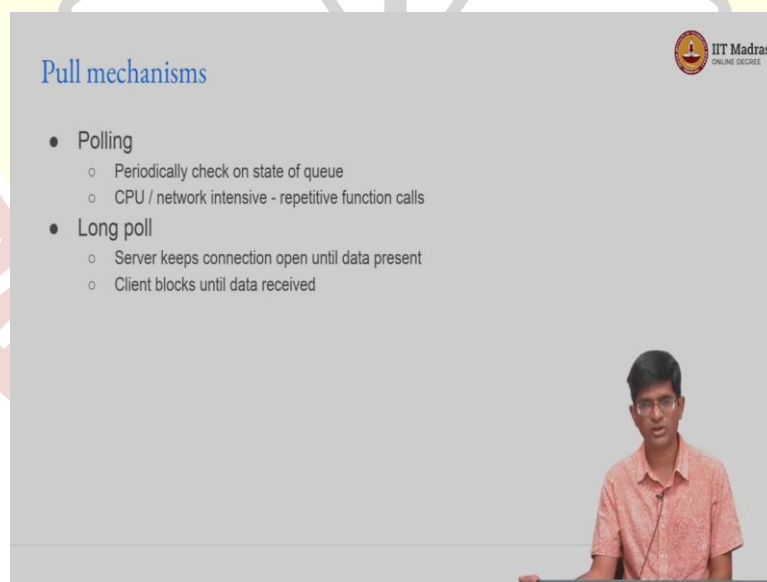


each of the clients basically pushes a message with the latest score onto the queue. Should the server be immediately processing each and every one of them? Or why not just wait for 5 seconds, collect everything that came in the last 5 seconds and do one global update and push information out or just prepare the latest updated information.

Why is that better, because each of these individual messages is very small and if I keep waking up and trying to process each one of them, it will probably end up becoming a large load on the server. But on the other hand if I wake up once every second or once every 5 seconds I can just process a large number of such messages very quickly and at the end of the day it does not have sort of a critical real time kind of effect on the final result.

So, dashboard applications, dashboards that are showing the present state of various things, they usually have some kind of an update interval. Maybe once in every second, if you are really concerned about it or it could be once in every 30 seconds. So, if I have a dashboard that is only updating once every 30 seconds, cannot my server also just wake up, once every several seconds, look at the queue, process everything over there and create the new set of data to go on to the dashboard on the next update. So, that is a notion of that is an example of what we would call a Pull Queue.

(Refer Slide Time: 13:48)



**Pull mechanisms**

- Polling
  - Periodically check on state of queue
  - CPU / network intensive - repetitive function calls
- Long poll
  - Server keeps connection open until data present
  - Client blocks until data received

Now, for implementing pull, there are a couple of different mechanisms. One is called polling where the server would essentially just sort of keep on periodically checking on the state of a queue. It would go and check the queue, is there any data for me? No. Go back, sleep for let us say a few milliseconds, come back again and check, is there any data? No. it can be very CPU or network intensive, because what happens is that the server is busy.

For those of you, who have worked with computer hardware, you would know that once again interrupts. There is a notion of interrupts versus polling in order to implement communication over there, once again. Now, what happens is that the interrupt basically means that the CPU just does not bother with anything; it waits to be explicitly interrupted, in some way, and told that it needs to do something. That is the equivalent of essentially an event driven system in a distributed system.

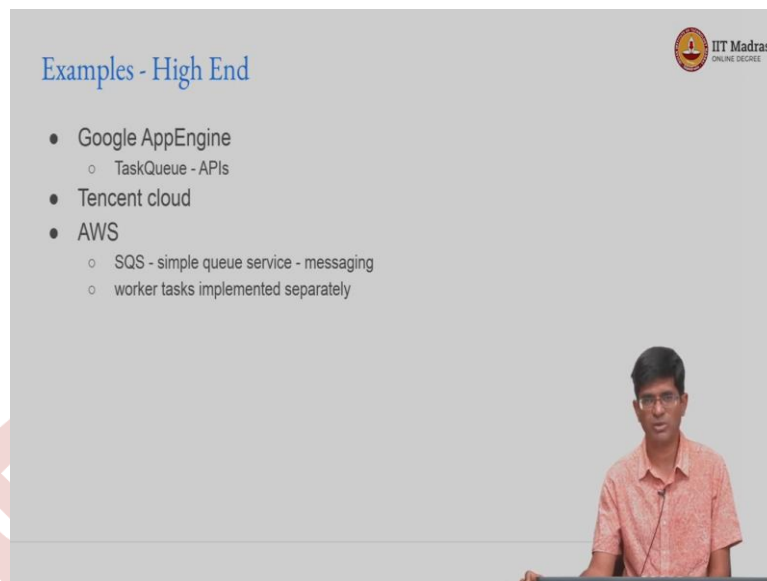
Polling on the other hand, is relatively easier to implement from a software point of view. Because all that it means is the client that is doing the work, just needs to keep on polling the server and checking if something is present. There is a variant of, so, which one should you use? Whether you should use polling or some kind of a push interrupt based mechanism, is potentially open to debate. I mean you need to sort of look at your application, look at the scenario and decide which one to go with.

But, there is an interesting variant of polling that is called a Long Poll. which is, the client makes a connection. The server simply does not send back a response, it creates an open thread and leaves it over there and it sends back data on that thread and closes the connection only when it actually has data.

What does a client do? The client actually has one physically open connection, now, the entire time. But, it all that it is doing is doing what is called a Blocking Read. It is just waiting for the data. In terms of something like the java script async await, the client is awaiting data, but that does not necessarily mean that it is actually busy or that it is actually doing any work.

There could be other mechanisms by which the client is told okay, fine, you can go to sleep, I will wake you up when there is actually data ready for you. So, long poles can be used essentially to just have a constant connection and the moment the server has some information, it sends it back and looks like an instantaneous update on the client side. So, these are sort of different pull mechanisms that could be implemented in practice.

(Refer Slide Time: 16:21)



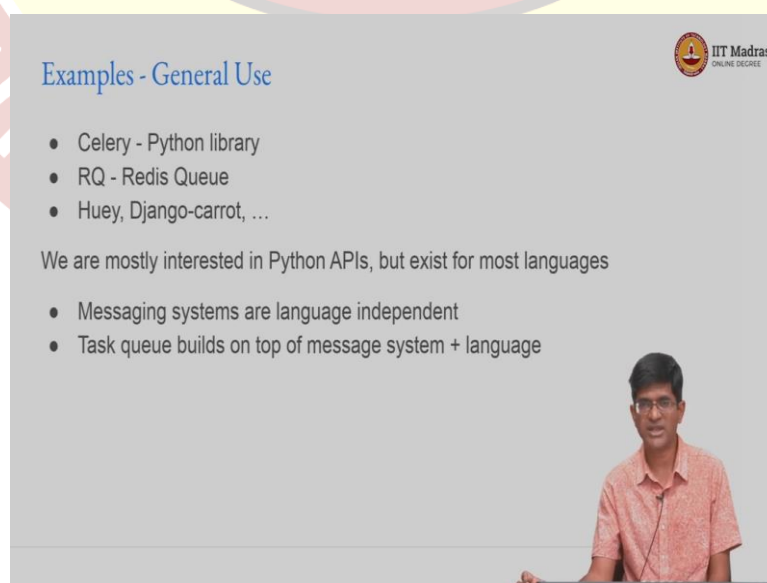
The slide is titled "Examples - High End" in blue text. It features a list of cloud services and their associated task queue APIs. In the bottom right corner, there is a small video feed of a male speaker wearing a red patterned shirt and glasses. The IIT Madras logo is in the top right corner.

- Google AppEngine
  - TaskQueue - APIs
- Tencent cloud
- AWS
  - SQS - simple queue service - messaging
  - worker tasks implemented separately

So, there are several examples of such task queues. there are some high end task queues like Google app engine has APIs for Python task queue and various other languages as well. Tencent cloud, which is the Chinese cloud provider, they also have their own documentation and information about where things can be implemented.

AWS, amazon web services, they use something called the simple queue service, primarily meant for messaging. And you can sort of implement your own worker tasks on top of that. So, the question is, do you explicitly need a task queue or, should you just build your own on top of a messaging system?

(Refer Slide Time: 17:00)



The slide is titled "Examples - General Use" in blue text. It lists several Python libraries for task queues. Below the list, a paragraph states that they are mostly interested in Python APIs but exist for most languages. Another list follows, discussing messaging systems. In the bottom right corner, there is a small video feed of the same male speaker. The IIT Madras logo is in the top right corner.

- Celery - Python library
- RQ - Redis Queue
- Huey, Django-carrot, ...

We are mostly interested in Python APIs, but exist for most languages

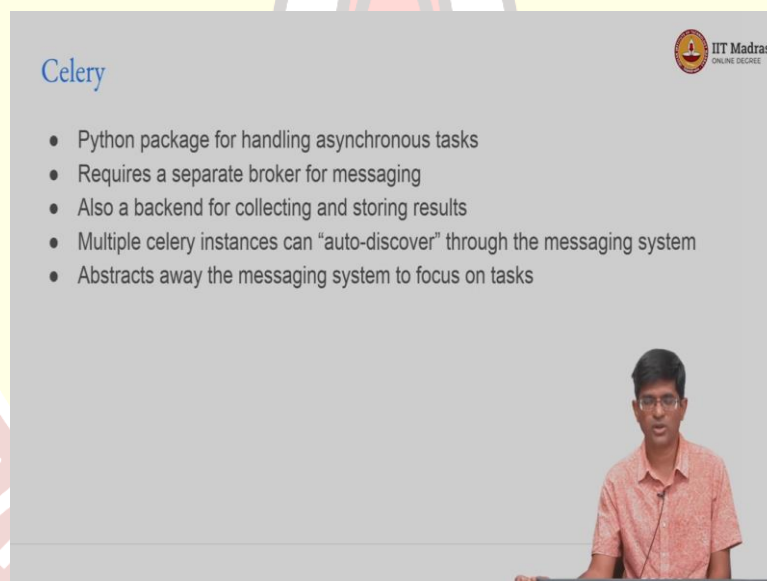
- Messaging systems are language independent
- Task queue builds on top of message system + language

In the case of open source, there is this notion of Celery, which is a library in Python. There are several others as well, there is something called RQ or Radius Queue, Huey, Django-Carrot, all of these are different variants that are available for Python. In our case, we are primarily interested in Python APIs. These things exist for most languages.

And the interesting thing that you need to keep in mind is the messaging system, the underlying messaging system, we had mentioned RabbitMQ and Redis, these messaging systems are mostly language independent. So, Redis is not really something that can only be used from Python, it can be used from php, from C, from Go, Java, pretty much any language has bindings to Redis.

But, the task queue on the other hand, when we talk about something like Celery that is essentially a framework that is being built, combining a message system with language features, in order to give us useful functionality.

(Refer Slide Time: 18:02)



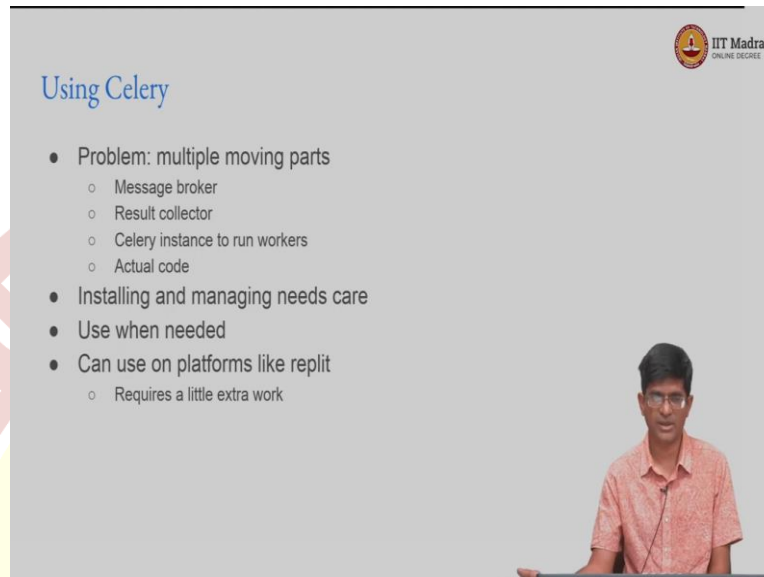
The slide is titled "Celery" in blue text. It features a list of five bullet points: "Python package for handling asynchronous tasks", "Requires a separate broker for messaging", "Also a backend for collecting and storing results", "Multiple celery instances can 'auto-discover' through the messaging system", and "Abstracts away the messaging system to focus on tasks". In the top right corner, there is a small logo for "IIT Madras ONLINE DEGREE". In the bottom right corner, there is a small video inset showing a man in a red shirt speaking.

- Python package for handling asynchronous tasks
- Requires a separate broker for messaging
- Also a backend for collecting and storing results
- Multiple celery instances can "auto-discover" through the messaging system
- Abstracts away the messaging system to focus on tasks

So, Celery is a Python package for handling asynchronous tasks. It requires a separate broker for messaging and it also requires a backend for collecting and storing results. Now, multiple Celery instances can auto discover. That is very interesting. Effectively, what is happening is that, you need to set up a messaging system once, but after that you can have multiple Celery workers that are sort of spinning up tasks and taking care of messaging through the messaging system and even discovering, who other, what are the other workers that are present, at any given point in time..

And what it does is that it allows you to abstract away the messaging system. It does not care whether you are using RabbitMQ or Redis or some other things, as long as they are supported by Celery. And the user can now focus on implementing tasks.

(Refer Slide Time: 18:55)



The slide is titled "Using Celery" in blue text. In the top right corner, there is a logo for "IIT Madras ONLINE DEGREE". The main content consists of a bulleted list:

- Problem: multiple moving parts
  - Message broker
  - Result collector
  - Celery instance to run workers
  - Actual code
- Installing and managing needs care
- Use when needed
- Can use on platforms like replit
  - Requires a little extra work

In the bottom right corner of the slide, there is a small video inset showing a man with glasses and a red shirt speaking.

So, finally we come to the question of, how do we use something like Celery? And the biggest problem in some sense with trying to use a library like Celery is, so far, we were able to write Python code. And just pretty much run it. The entire program that we were writing was just something that we had in Python and we could go ahead and run the whole thing.

The problem with a message, a task queue framework is, it needs a message broker, and Python by itself is only an interpreter, all that it does is, it runs your program. It does not by itself provide a separate program that can handle messages and send them between different processes that are running on the system.

It similarly does not have some mechanism for a result collector which means that in order to implement a Celery based system, you will need to create your own code but in addition to that you would need to be running a message broker; you would have some other kind of database or some mechanism that can collect the results. And you would also need to have an instance of Celery that is running the workers and actually going to implement the tasks that you ask of it.

Installing and managing something like this, needs a fair amount of care. Because there are these different processes that need to be managed. If any one of them is not running properly, you need to bring it down restart it. Make sure that all of them are running at any given point

in time, one of them does not run out of memory and so on. A lot of other problems come in that you would not have been concerned with if your only job was to run a simple piece of Python code. Which means that as I said earlier, task queues are not something to sort of take up trivially, you do not just add a task queue for the fun of it, you do it, when you need it.

Of course, there might be some fairly simple use cases, where you actually need one. An example is, sending emails. So, even if, all that you want to do is really send out emails asynchronously from your application, you might be better off using a task queue to explicitly make sure that the job of sending the emails is handled reliably by a mechanism that is independent of your code.

Now, one final word, when especially when you want to use it on a platform like replit, it does require a little extra work and the reason for that is because replit, by itself, is creating a virtual machine, on which you can run. You do have full access to that virtual machine for the duration when you are running your code, but it is a virtual machine, which means that it restricts what kind of applications you can actually install on it. Which means that, for example, running something like a redis server in the background on replit is not a trivial thing to do. It can be done, but it does require a decent amount of extra work.

