Django Celery

GEEKYSHOWS YOUTUBE CHANNEL LEARNING NOTES

Source Code - https://github.com/satyam-seth-learnings/django_celery_learning/tree/main/Geekyshows

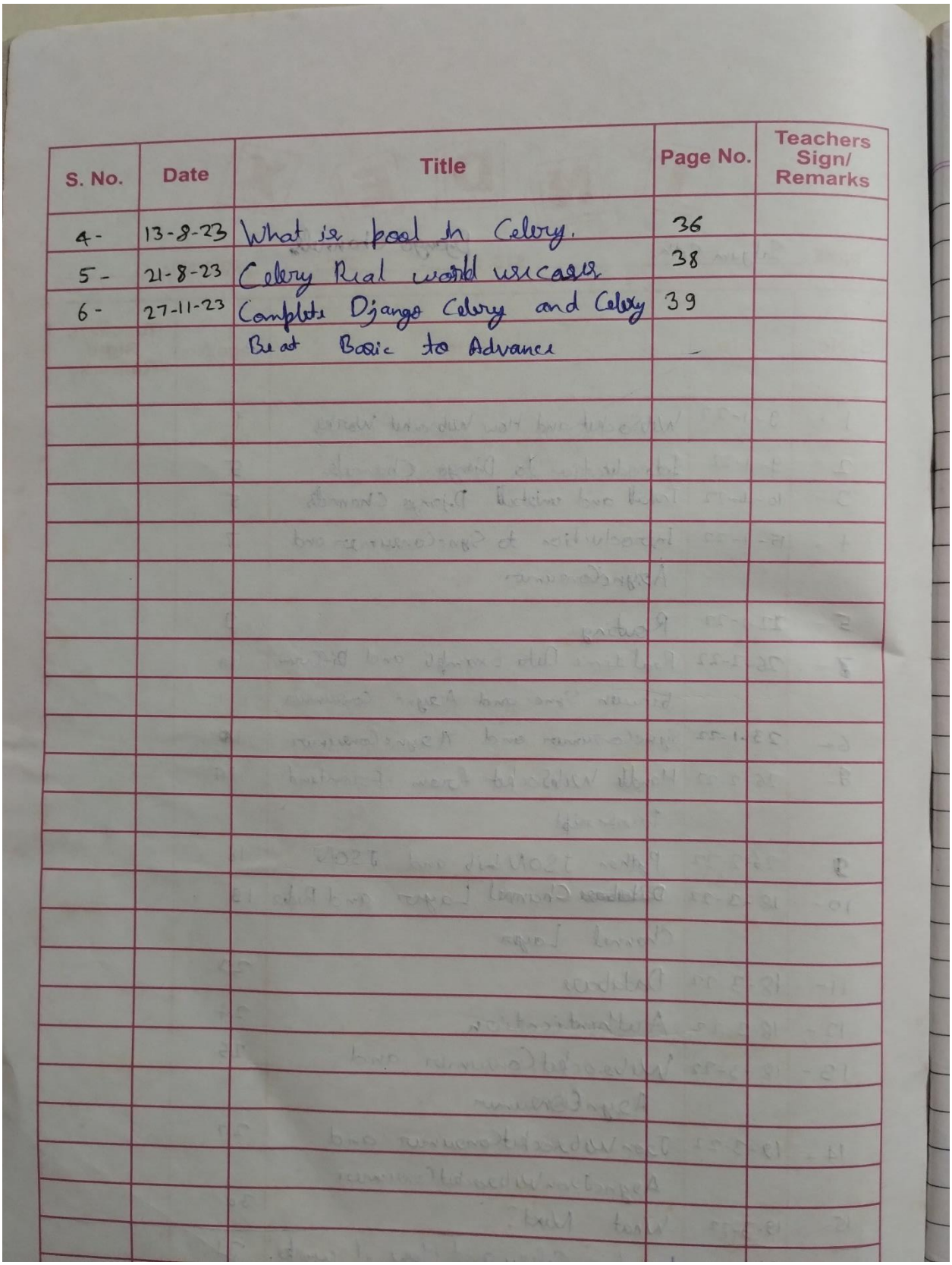
YouTube Link - https://www.youtube.com/playlist?list=PLbGui_ZYuhijN6wEQZ0DC-V5EdK4YMrYO

SATYAM SETH

27-05-2024

INDEX

NAME : Satyam Seth STD. : Django Channels SEC.: ROLL NO.:



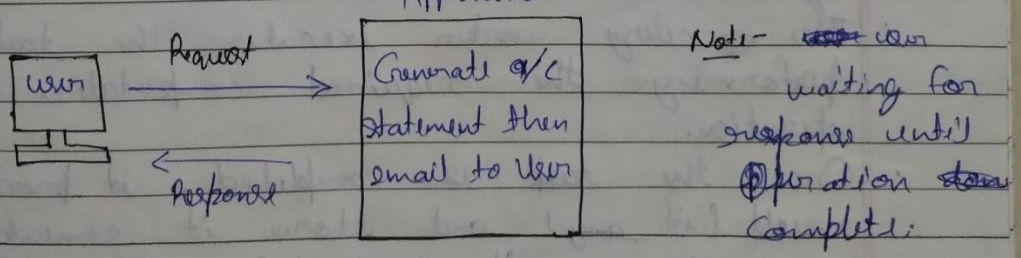
S. No.	Date	Title	Page No.	Teachers Sign/ Remarks
4-	13-8-23	What is pool in Celery.	36	
5-	21-8-23	Celery Real world use cases	38	
6-	27-11-23	Complete Django Celery and Celery Beat Basic to Advance	39	

ARFAT
Date 31-7-23
Page 31

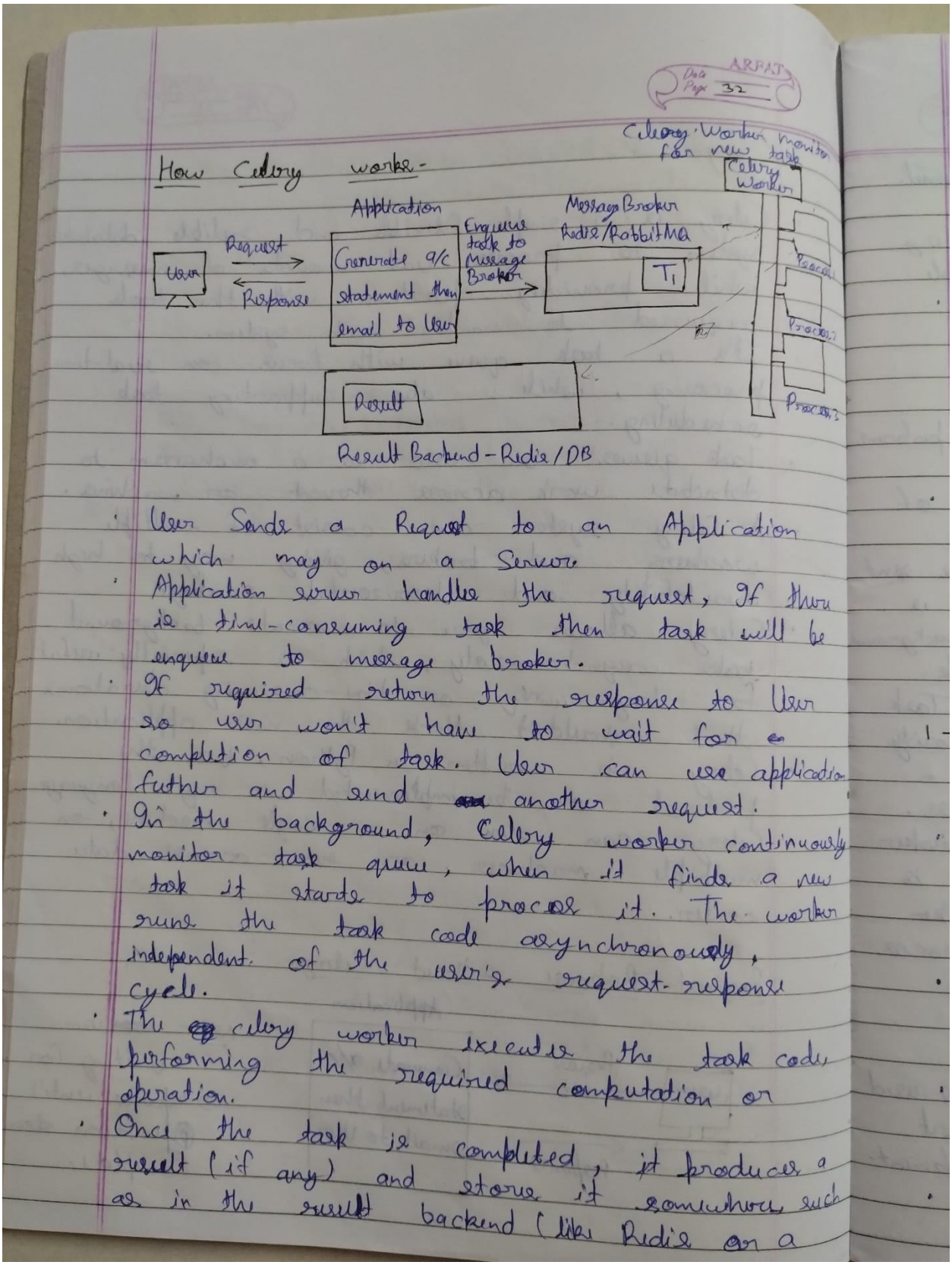
Celery

- Celery is a simple, flexible and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such system.
- It's a task queue with focus on real-time processing, while also supporting task scheduling.
- Task queues are used as a mechanism to distribute work across threads or machines.
- A Celery system can consist of multiple workers and brokers, giving way to high availability and horizontal scaling.
- Celery allows you to execute background tasks asynchronously, which is specially useful for long-running or time-consuming operations that shouldn't block the main application.
- Celery is written in Python, but the protocol can be implemented in any language.
- Celery can run on a single machine, on multiple machines, or even across data center.

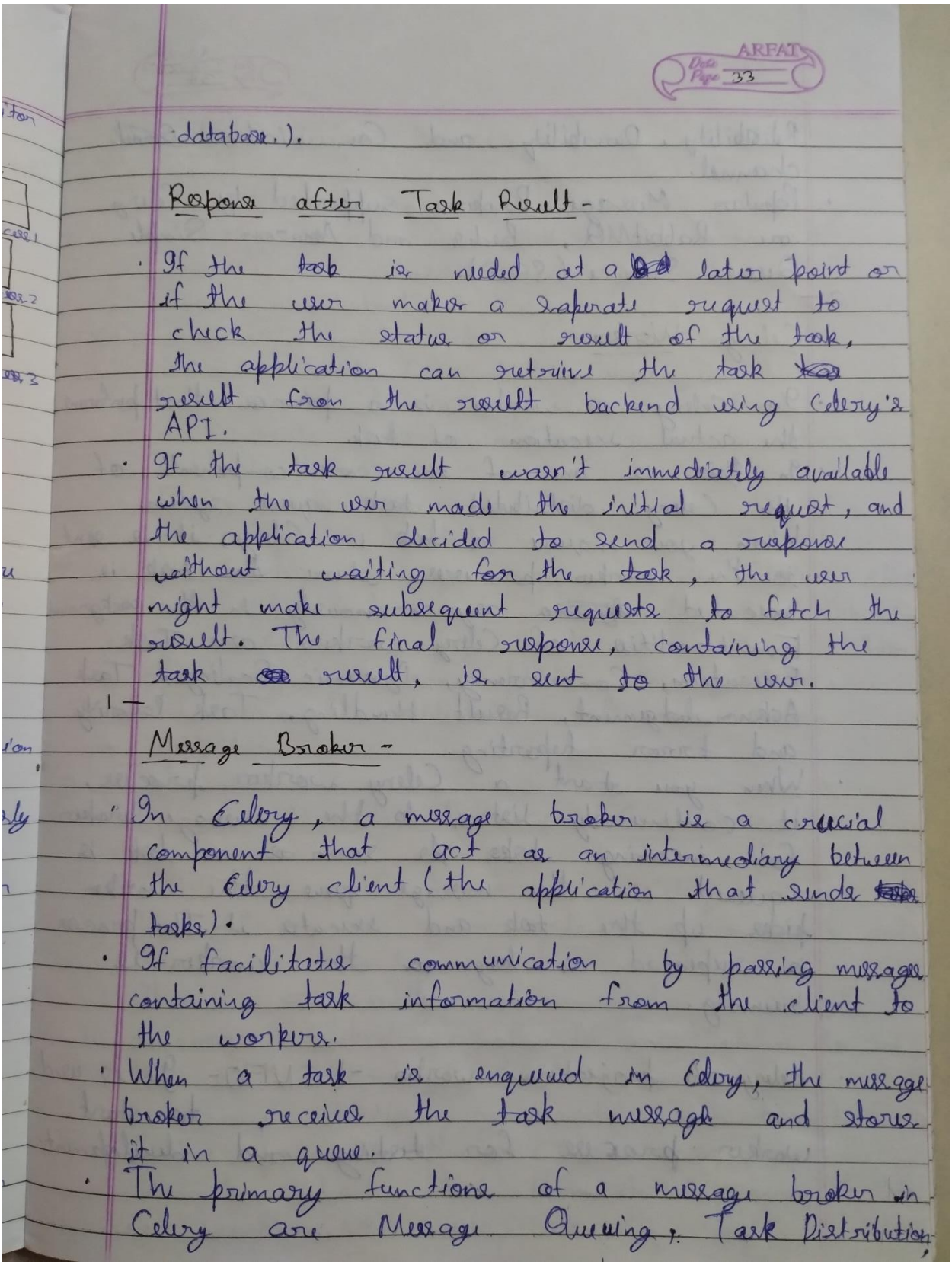
Request Response without Celery-



Note- User waiting for response until operation Completed.



- User Sends a Request to an Application which may on a Server.
- Application server handles the request, If there is time-consuming task then task will be enqueue to message broker.
- If required return the response to User so user won't have to wait for completion of task. User can use application futher and send another request.
- In the background, Celery worker continuously monitor task queue, when it finds a new task it starts to process it. The worker runs the task code asynchronously, independent of the user's request-response cycle.
- The celery worker executes the task code, performing the required computation or operation.
- Once the task is completed, it produces a result (if any) and stores it somewhere such as in the result backend (like Redis or a



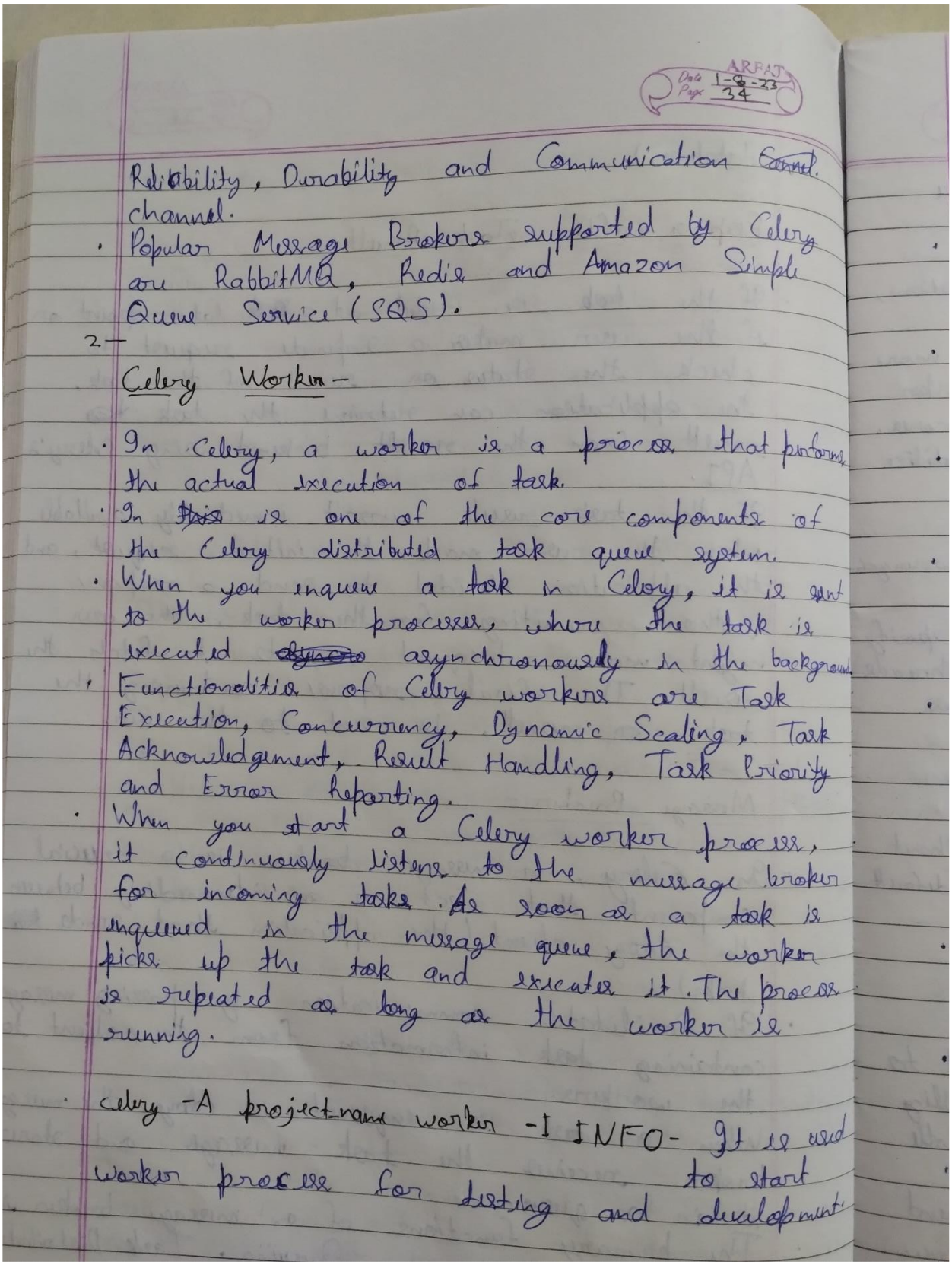
database.).

Response after Task Result -

- If the task is needed at a later point or if the user makes a separate request to check the status or result of the task, the application can retrieve the task result from the result backend using Celery's API.
- If the task result wasn't immediately available when the user made the initial request, and the application decided to send a response without waiting for the task, the user might make subsequent requests to fetch the result. The final response, containing the task result, is sent to the user.

Message Broker -

- In Celery, a message broker is a crucial component that act as an intermediary between the Celery client (the application that sends tasks).
- It facilitates communication by passing messages containing task information from the client to the workers.
- When a task is enqueued in Celery, the message broker receives the task message and stores it in a queue.
- The primary functions of a message broker in Celery are Message Queuing, Task Distribution,



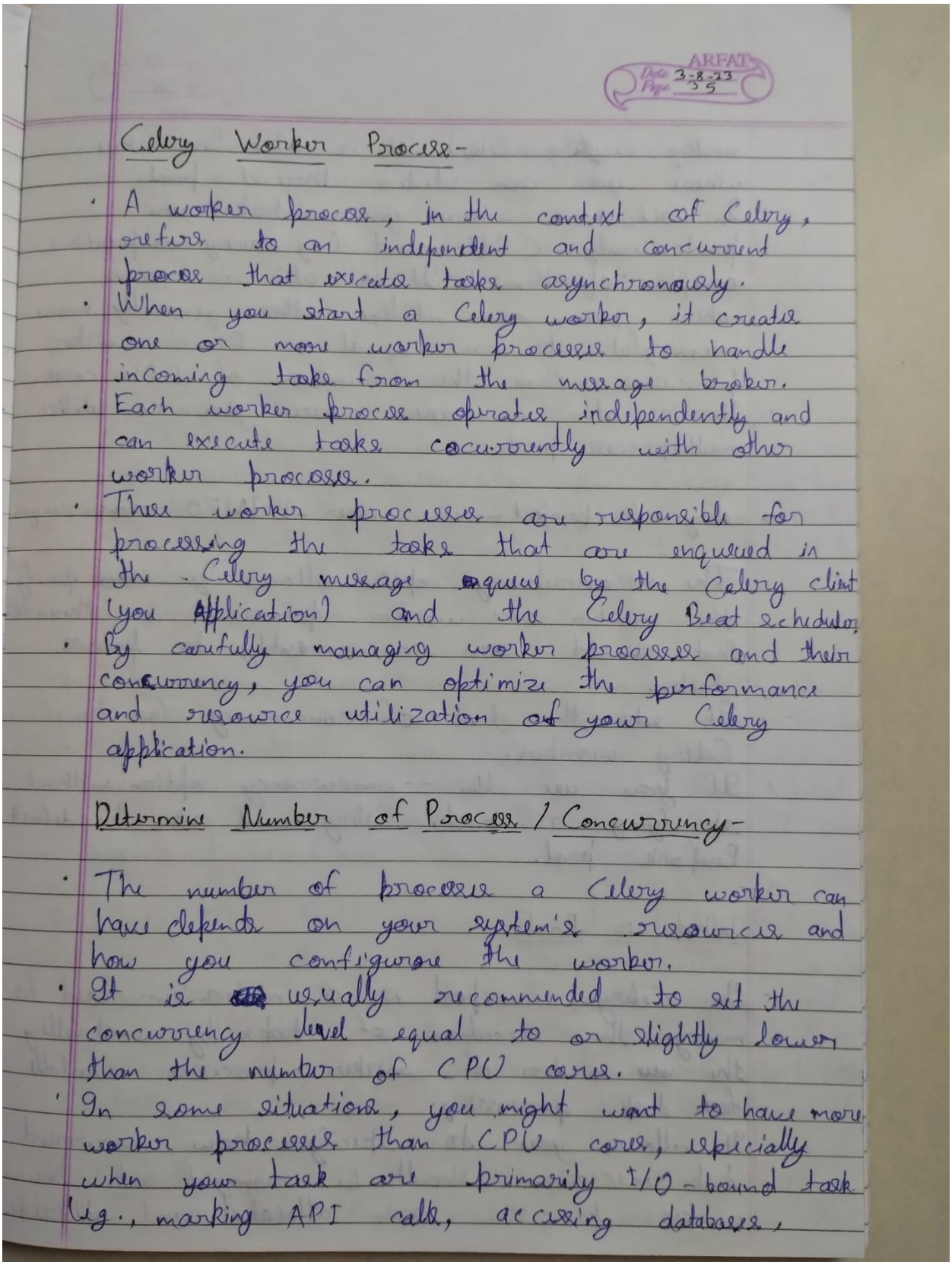
Reliability, Durability and Communication channel.
- Popular Message Brokers supported by Celery are RabbitMQ, Redis and Amazon Simple Queue Service (SQS).

2—

Celery Worker—

- In Celery, a worker is a process that performs the actual execution of task.
- It is one of the core components of the Celery distributed task queue system.
- When you enqueue a task in Celery, it is sent to the worker processes, where the task is executed asynchronously in the background.
- Functionalities of Celery workers are Task Execution, Concurrency, Dynamic Scaling, Task Acknowledgment, Result Handling, Task Priority and Error Reporting.
- When you start a Celery worker process, it continuously listens to the message broker for incoming tasks. As soon as a task is enqueued in the message queue, the worker picks up the task and executes it. The process is repeated as long as the worker is running.

- celery -A project_name worker -l INFO- It is used to start worker process for testing and development



Celery Worker Process -

- A worker process, in the context of Celery, refers to an independent and concurrent process that execute tasks asynchronously.
- When you start a Celery worker, it creates one or more worker processes to handle incoming tasks from the message broker.
- Each worker process operates independently and can execute tasks concurrently with other worker processes.
- These worker processes are responsible for processing the tasks that are enqueued in the Celery message queue by the Celery client (your Application) and the Celery Beat scheduler.
- By carefully managing worker processes and their concurrency, you can optimize the performance and resource utilization of your Celery application.

Determine Number of Process / Concurrency -

- The number of processes a Celery worker can have depends on your system's resources and how you configure the worker.
- It is usually recommended to set the concurrency level equal to or slightly lower than the number of CPU cores.
- In some situations, you might want to have more worker processes than CPU cores, especially when your task are primarily I/O-bound task (e.g., making API calls, accessing databases,

reading / writing file). It is also an area where you can utilize thread pool.
- In situations where your tasks are predominantly CPU-bound (eg. heavy computations, complex algorithms), it is ~~also an area where you can utilize thus~~ generally more meaningful to have equal or fewer worker processes than the number of CPU cores.
- It is also an area where you can utilize multi-process pool.

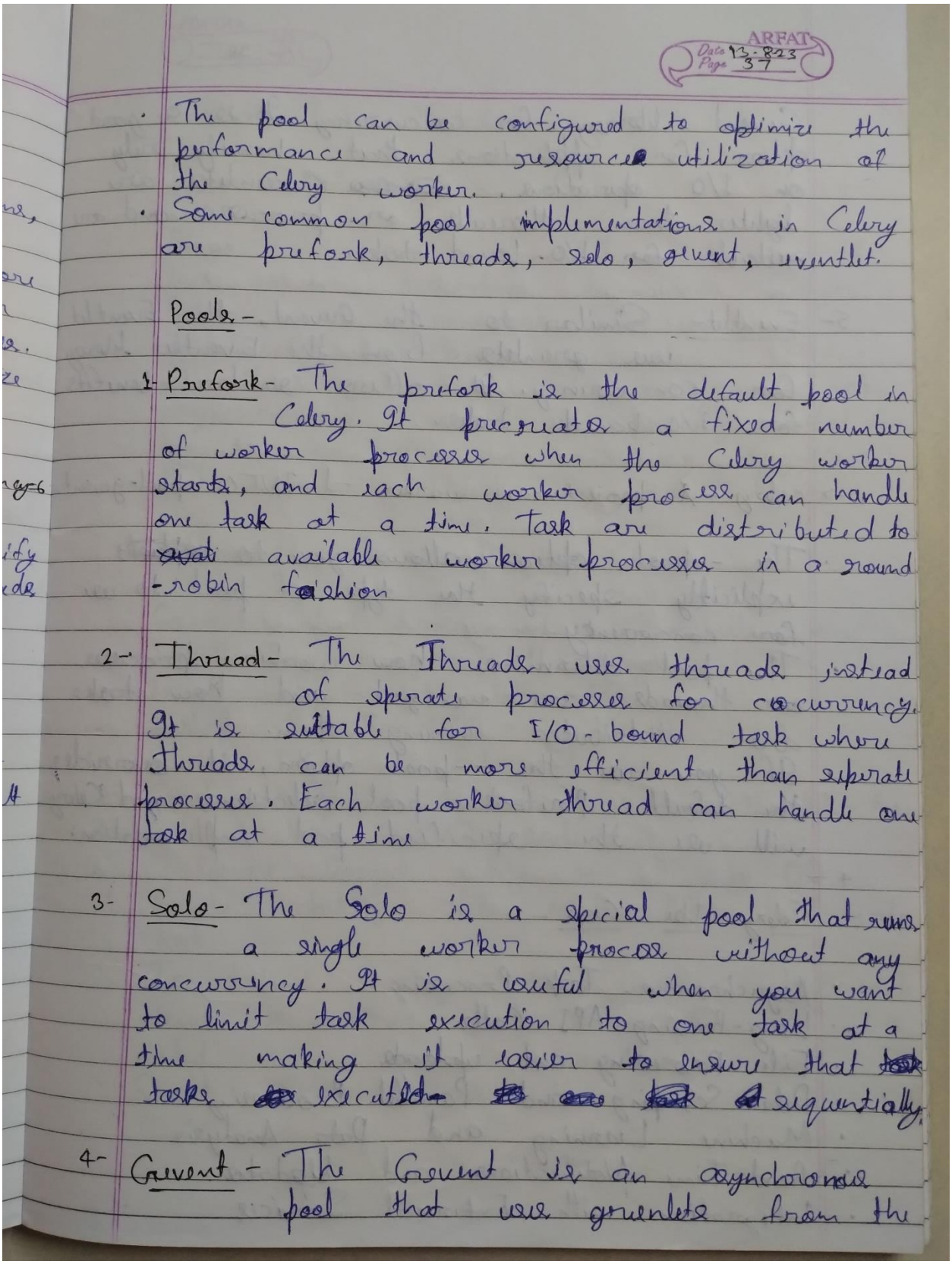
→ celery -A project_name worker -l INFO --concurrency

- The --concurrency option allows you to specify the number of worker processes or threads that should run concurrently to process tasks.
- It sets the level of concurrency for the Celery worker.
- If you use the --concurrency option without specifying a --pool, Celery will use the default Prefork pool.

3 —

What is Pool-

- In Celery, a pool is a mechanism used to manage the execution of tasks by controlling the ~~num~~ number of worker processes available for task processing.
- It allows you to specify how concurrent task execution should be managed and how worker processes are allocated and utilized.



- The pool can be configured to optimize the performance and resource utilization of the Celery worker.
- Some common pool implementations in Celery are prefork, threads, solo, gevent, eventlet.

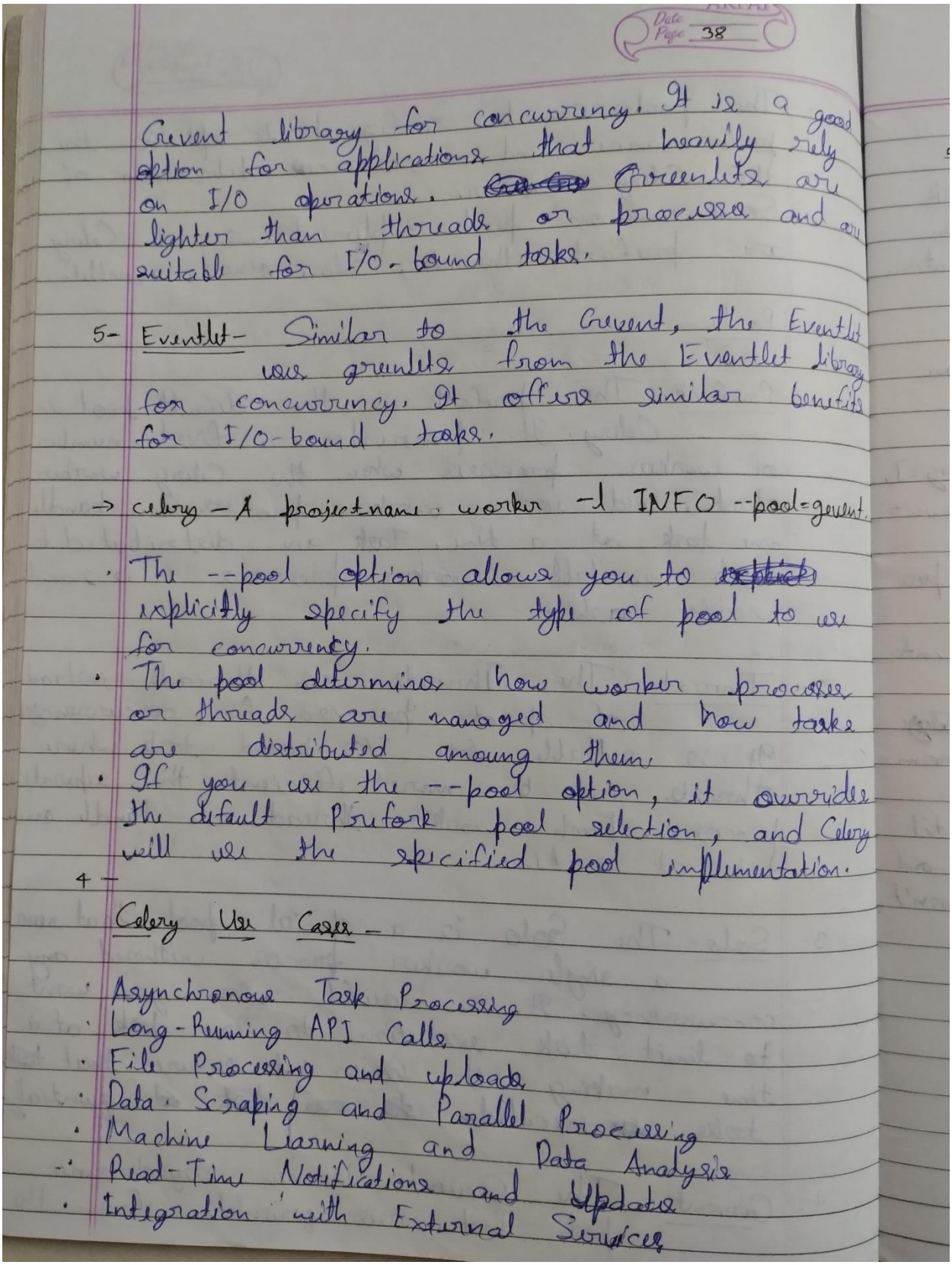
Pools -

1- Prefork - The prefork is the default pool in Celery. It creates a fixed number of worker processes when the Celery worker starts, and each worker process can handle one task at a time. Task are distributed to available worker processes in a round-robin fashion

2- Thread - The Threads use threads instead of separate processes for concurrency. It is suitable for I/O-bound task where threads can be more efficient than separate processes. Each worker thread can handle one task at a time

3- Solo - The Solo is a special pool that runs a single worker process without any concurrency. It is useful when you want to limit task execution to one task at a time making it easier to ensure that tasks executed sequentially.

4- Gevent - The Gevent is an asynchronous pool that use greenlets from the



Gevent library for concurrency. It is a good option for applications that heavily rely on I/O operations. Greenlets are lighter than threads or processes and are suitable for I/O-bound tasks.

5- Eventlet- Similar to the Gevent, the Eventlet uses greenlets from the Eventlet library for concurrency. It offers similar benefits for I/O-bound tasks.

→ celery -A projectname worker -l INFO --pool=gevent.

- The --pool option allows you to explicitly specify the type of pool to use for concurrency.
- The pool determines how worker processes or threads are managed and how tasks are distributed among them.
- If you use the --pool option, it overrides the default Prefork pool selection, and Celery will use the specified pool implementation.

4 -

Celery Use Cases -

- Asynchronous Task Processing
- Long-Running API Calls
- File Processing and uploads
- Data Scraping and Parallel Processing
- Machine Learning and Data Analysis
- Real-Time Notifications and Updates
- Integration with External Services

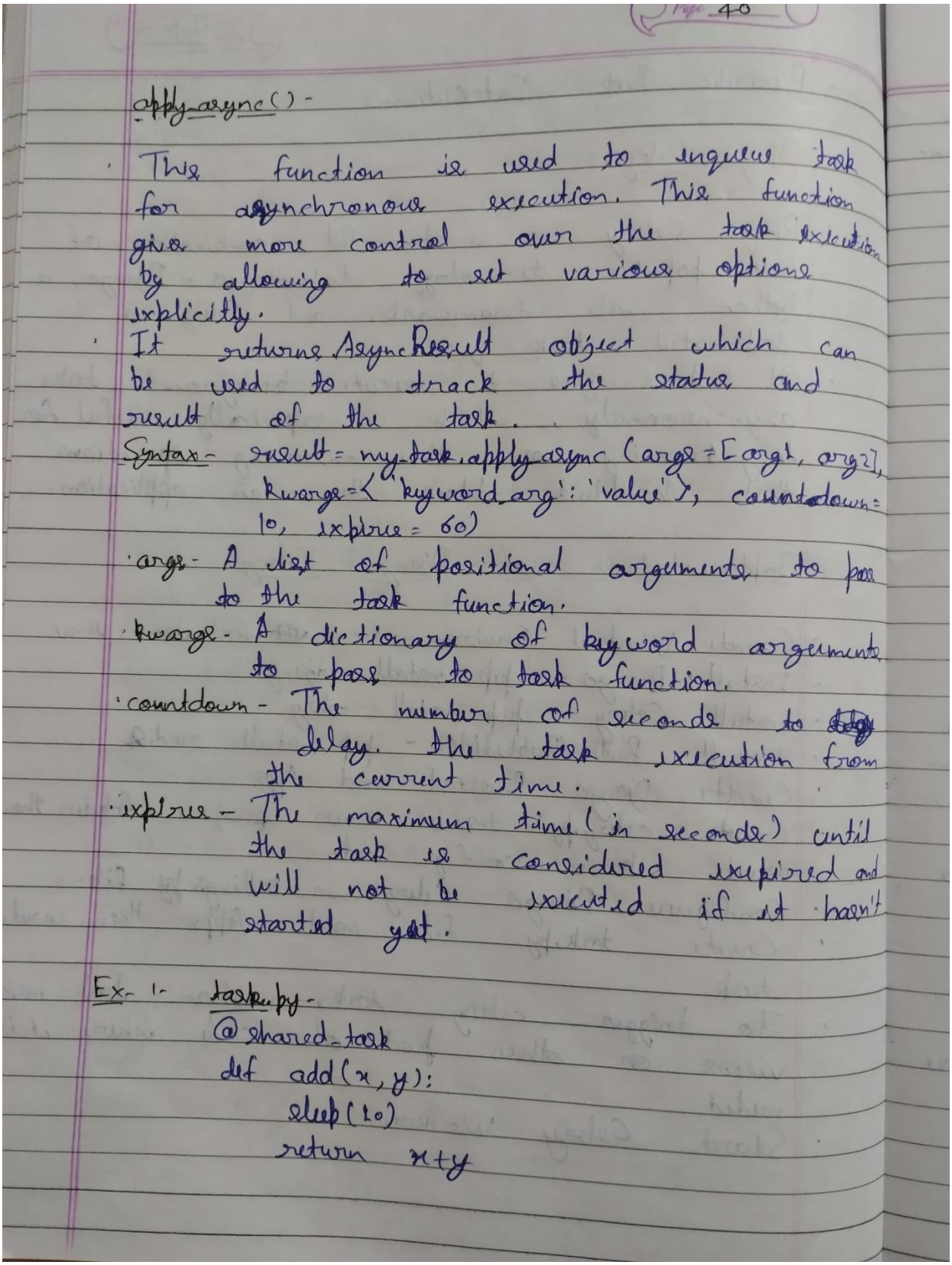
5 — Periodic Task Execution.

Django Celery -

- Django Celery is a powerful combination of two popular ~~technology~~ technologies - Django, a Python web framework, and Celery, a distributed task queue.
- It allows you to execute background tasks asynchronously, which is especially useful for long-running or time-consuming operations that shouldn't block the main application.

Implementation of Django Celery -

- Create Virtual Environment - virtualenv venv
- Install Django - pip install django
- Install Celery - pip install celery.
- Install Redis/RabbitMQ - pip install redis
- Create Django Project and Apps
- Create celery.py inside inner project folder then write basic config code.
- Configure Django Celery in settings.py file.
- Create task.py file inside Apps then write task.
- To trigger celery task, mention task inside views or other part of code where it is needed
- Start Celery Worker.



apply_async() -

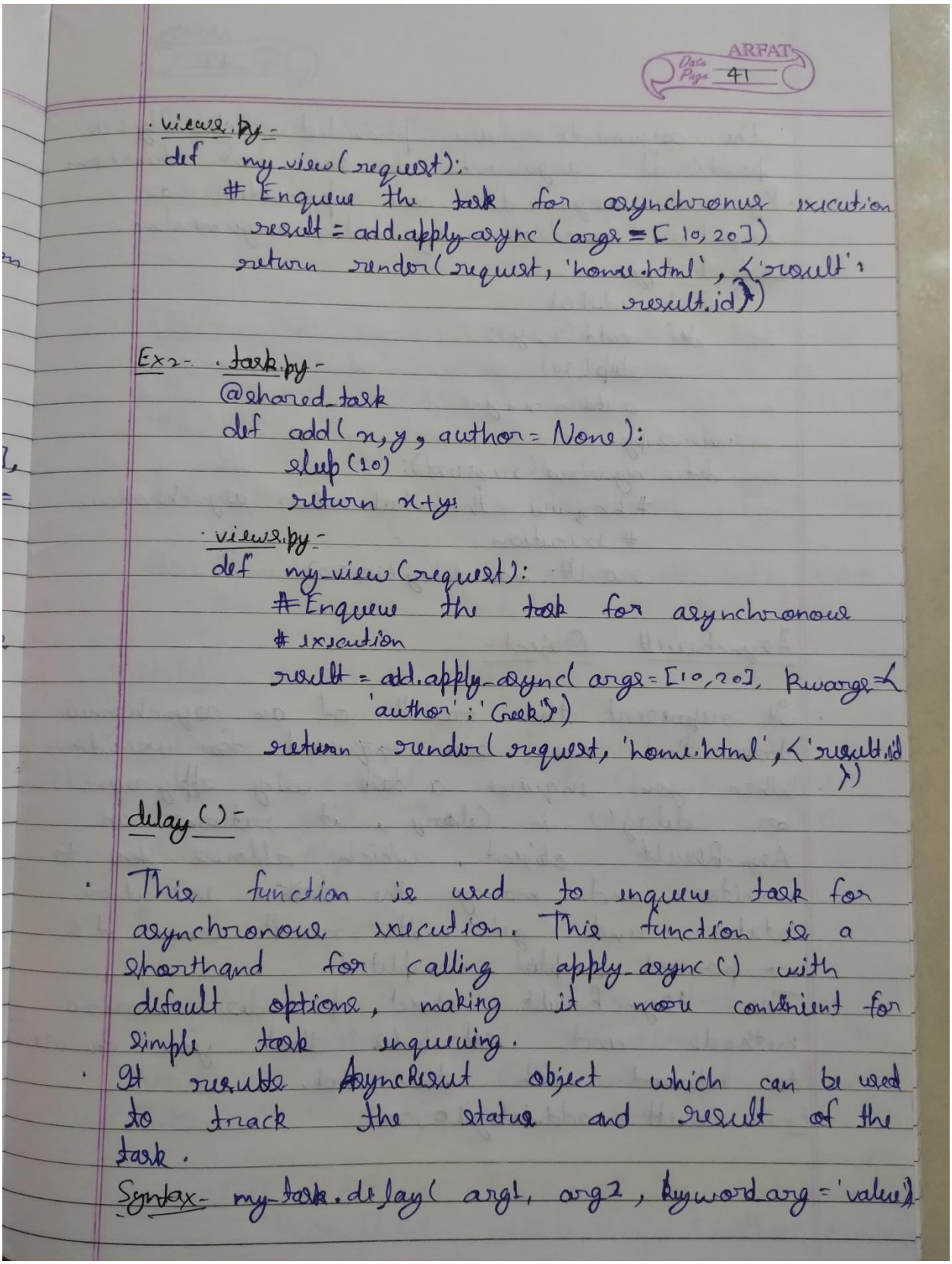
- This function is used to enqueue task for asynchronous execution. This function give more control over the task execution by allowing to set various options explicitly.
- It returns AsyncResult object which can be used to track the status and result of the task.

Syntax - result = my_task.apply_async (args=[arg1, arg2], kwargs={'keyword_arg': 'value'}, countdown=10, expires=60)

- args - A list of positional arguments to pass to the task function.
- kwargs - A dictionary of keyword arguments to pass to task function.
- countdown - The number of seconds to delay. the task execution from the current time.
- expires - The maximum time (in seconds) until the task is considered expired and will not be executed if it hasn't started yet.

Ex - 1- task.py -

```
@shared_task
def add(x, y):
    sleep(10)
    return x+y
```



· views.py -

```
def my_view(request):
    # Enqueue the task for asynchronous execution
    result = add.apply_async(args=[10, 20])
    return render(request, 'home.html', {'result': result.id})
```

Ex2 - · task.py -

```
@shared_task
def add(x, y, author=None):
    sleep(10)
    return x+y
```

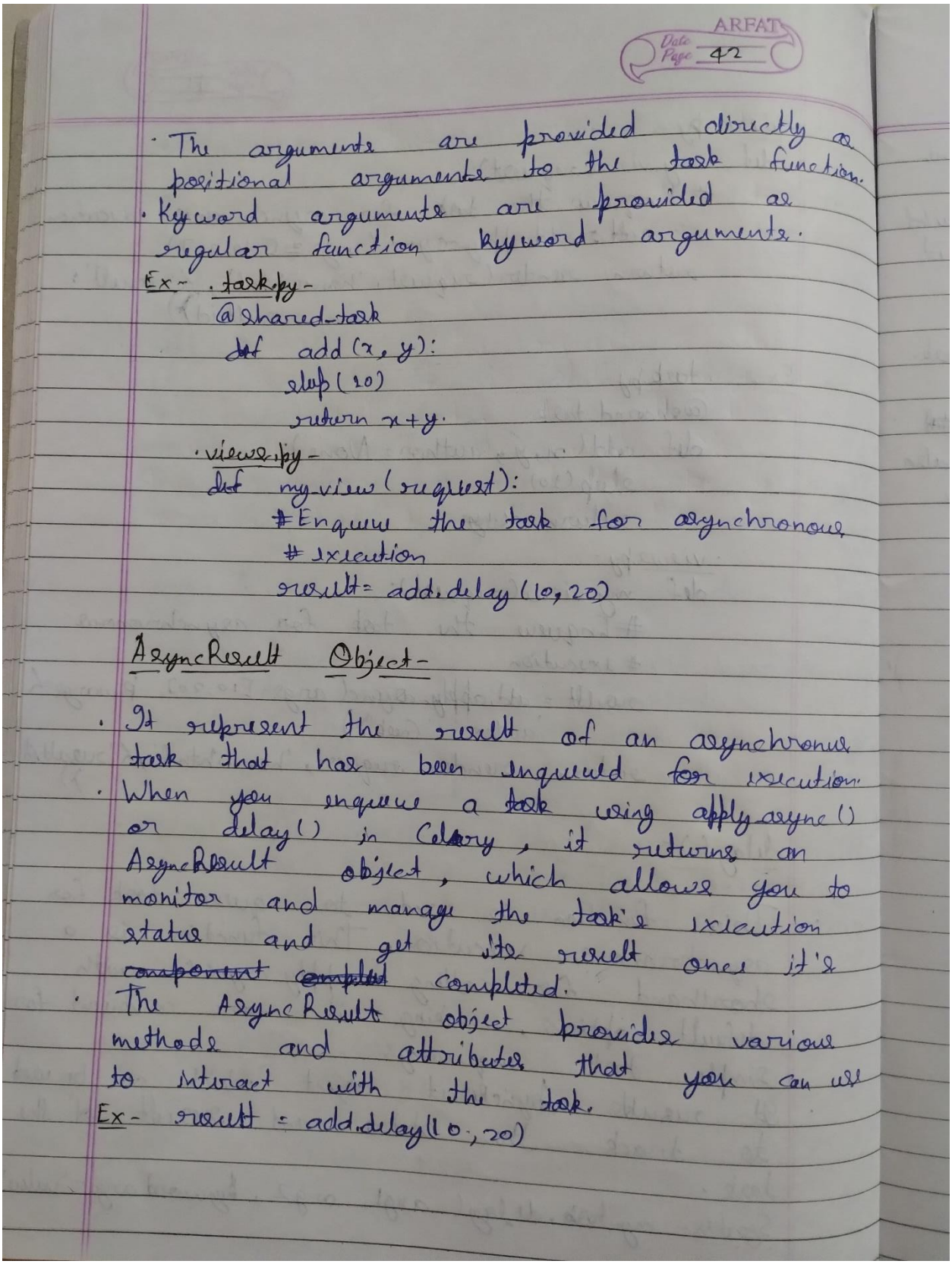
· views.py -

```
def my_view(request):
    # Enqueue the task for asynchronous
    # execution
    result = add.apply_async(args=[10,20], kwargs={'author':'Geek'})
    return render(request, 'home.html', {'result.id'})
```

delay() -

- This function is used to enqueue task for asynchronous execution. This function is a shorthand for calling apply_async() with default options, making it more convinient for simple task enqueuing.
- It results AsyncResult object which can be used to track the status and result of the task.

Syntax - my_task.delay(arg1, arg2, keyword_arg='value')



- The arguments are provided directly as positional arguments to the task function.
- Keyword arguments are provided as regular function keyword arguments.

Ex - . task.py -

```
@shared_task
def add(x, y):
    sleep(10)
    return x+y.
```

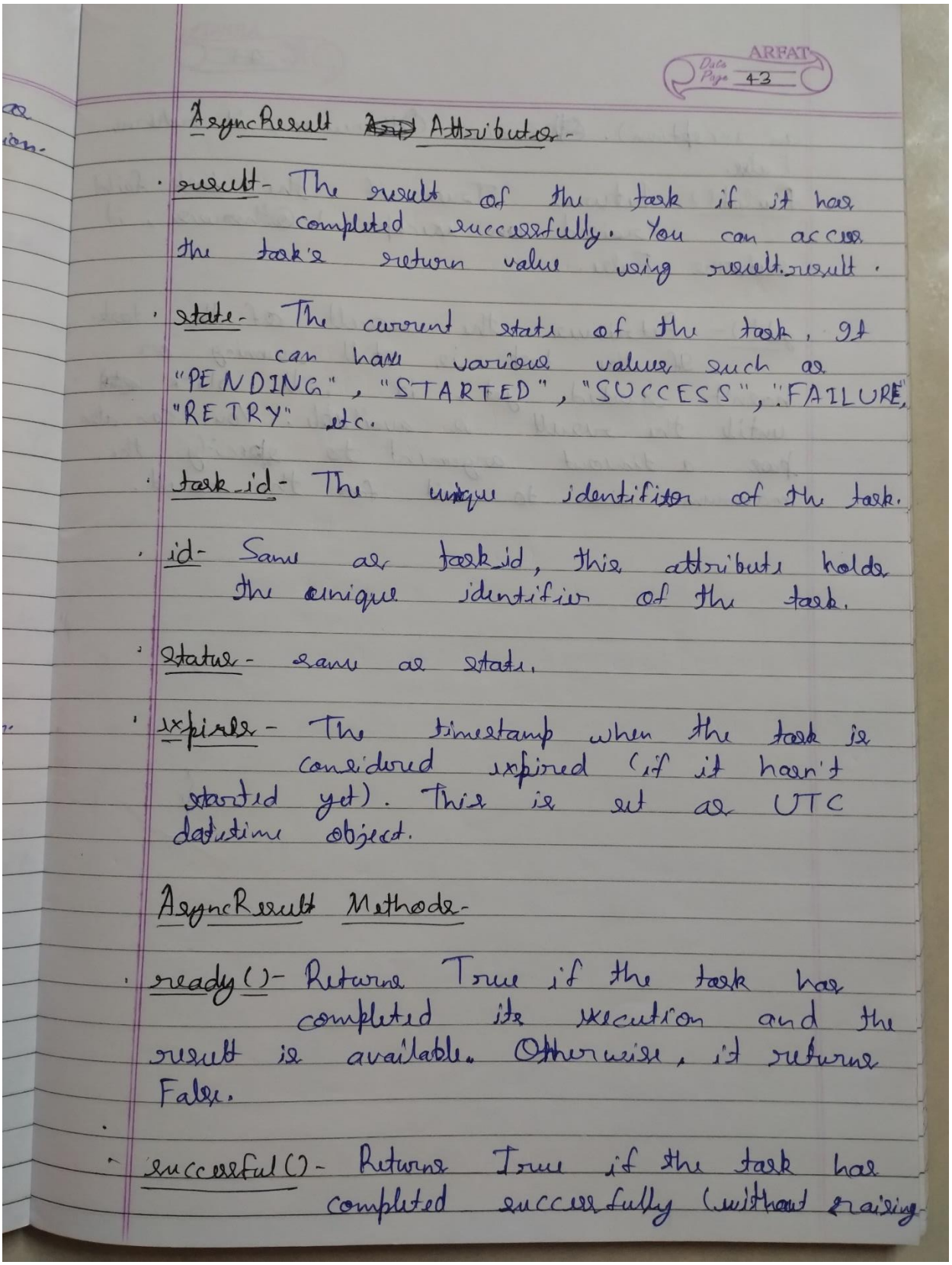
. views.py -

```
def my_view(request):
    # Enqueue the task for asynchronous
    # execution
    result = add.delay(10, 20)
```

AsyncResult Object -

- It represent the result of an asynchronous task that has been enqueued for execution.
- When you enqueue a task using apply_async() or delay() in Celery, it returns an AsyncResult object, which allows you to monitor and manage the task's execution status and get its result once it's ~~component~~ ~~completed~~ completed.
- The AsyncResult object provides various methods and attributes that you can use to interact with the task.

Ex - result = add.delay(10, 20)



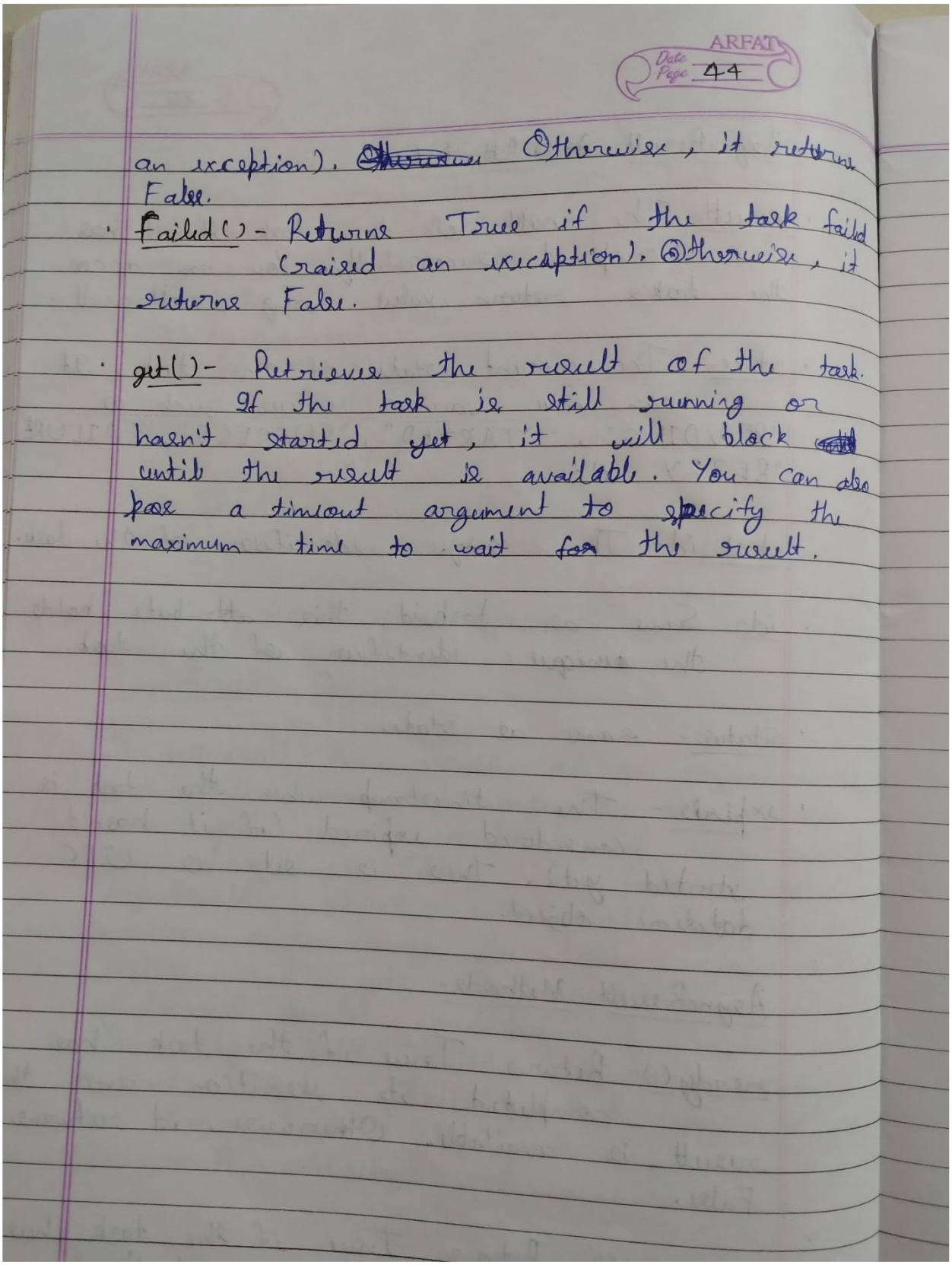
ARFAT
Date Page 43

AsyncResult Attributes -

- result - The result of the task if it has completed successfully. You can access the task's return value using result.result.
- state - The current state of the task, it can have various values such as "PENDING", "STARTED", "SUCCESS", "FAILURE", "RETRY" etc.
- task_id - The unique identifier of the task.
- id - Same as task_id, this attribute holds the unique identifier of the task.
- status - same as state.
- expires - The timestamp when the task is considered expired (if it hasn't started yet). This is set as UTC datetime object.

AsyncResult Methods -

- ready() - Returns True if the task has completed its execution and the result is available. Otherwise, it returns False.
- successful() - Returns True if the task has completed successfully (without raising



an exception). Otherwise, it returns False.

- failed() - Returns True if the task failed (raised an exception). Otherwise, it returns False.

- get() - Retrieves the result of the task. If the task is still running or hasn't started yet, it will block until the result is available. You can also pass a timeout argument to specify the maximum time to wait for the result.