# Celery

## A Distributed Task Queue

Oskar Hollmann

User Technologies

26.10.2016

# Motivation

- time demanding tasks are a pain in web apps
  - HTTP request can easily time out
  - it's not acceptable to block the client for too long
  - client may not care about the result
- either we return an URL to the client who polls it later to get the result
- or we push it through web sockets
- how can we achieve that?

# Motivation

Node.js to the rescue!

# Motivation

*Node.js is cool, but...*

- Why use Node.js and struggle with async code when async operations are seldom needed in a web app?
- Async code is not enough – we might need to distribute the tasks, run them periodically, ...
- Node.js programmers are a rare commodity.
- What we actually need is **an asynchronous task queue**.
- Examples: RabbitMQ, JMS, Celery, ...


# Use Cases of Distributed Task Queues

1. Non-blocking task execution.
2. Task execution with failure recovery.
3. Concurrent task execution for single-threaded apps.
4. Distribute task to other machines.
5. Handle complex task workflows with dependencies.
6. Periodic tasks.

# Task Queue in Web App

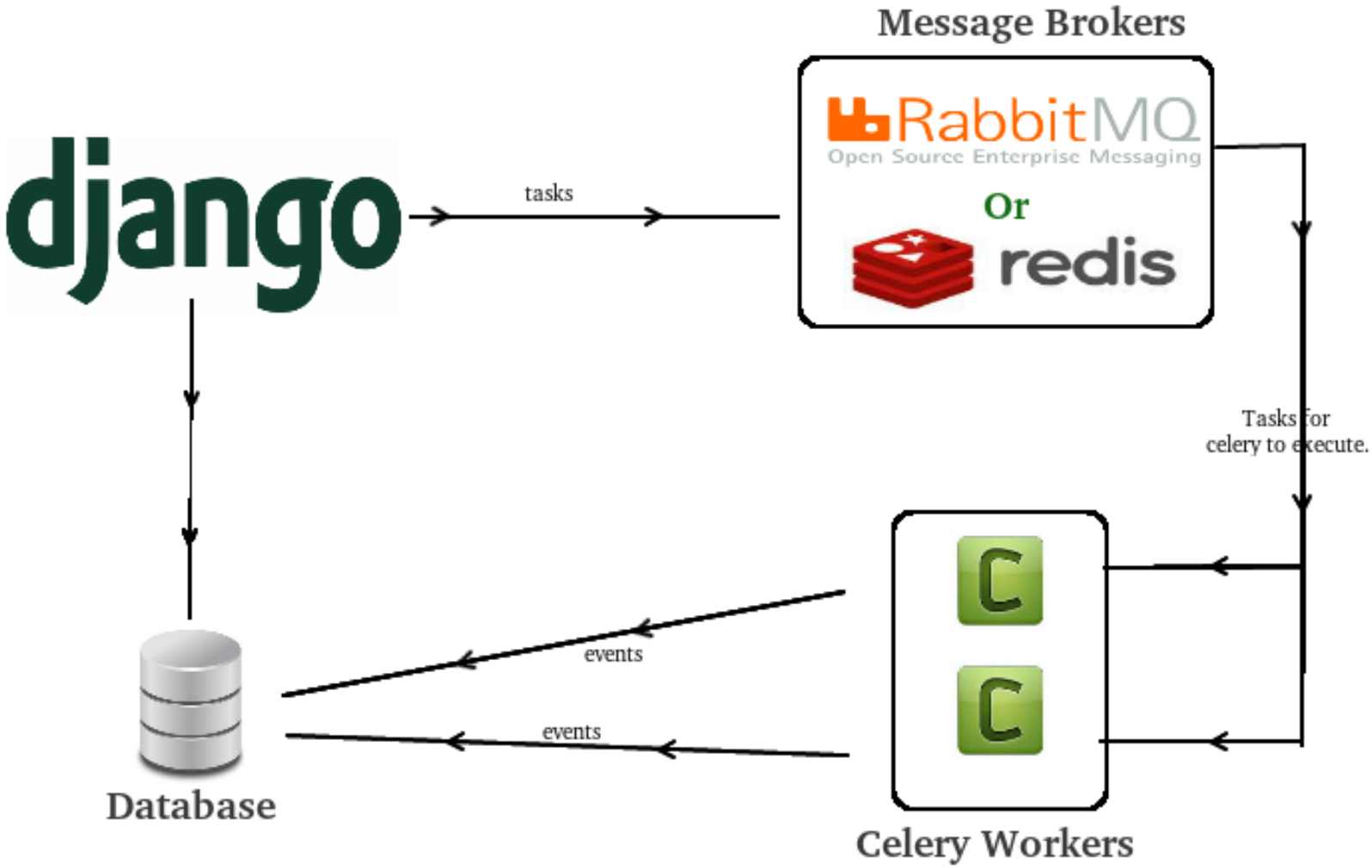


Figure: Source en.proft.me

# Celery is

- distributed task queue written in Python
- bindings for: PHP, Ruby, NodeJS and more
- different message broker transports: Redis, RabbitMQ, MongoDB and more
- arbitrary number of queues and workers

# Workers and Queues

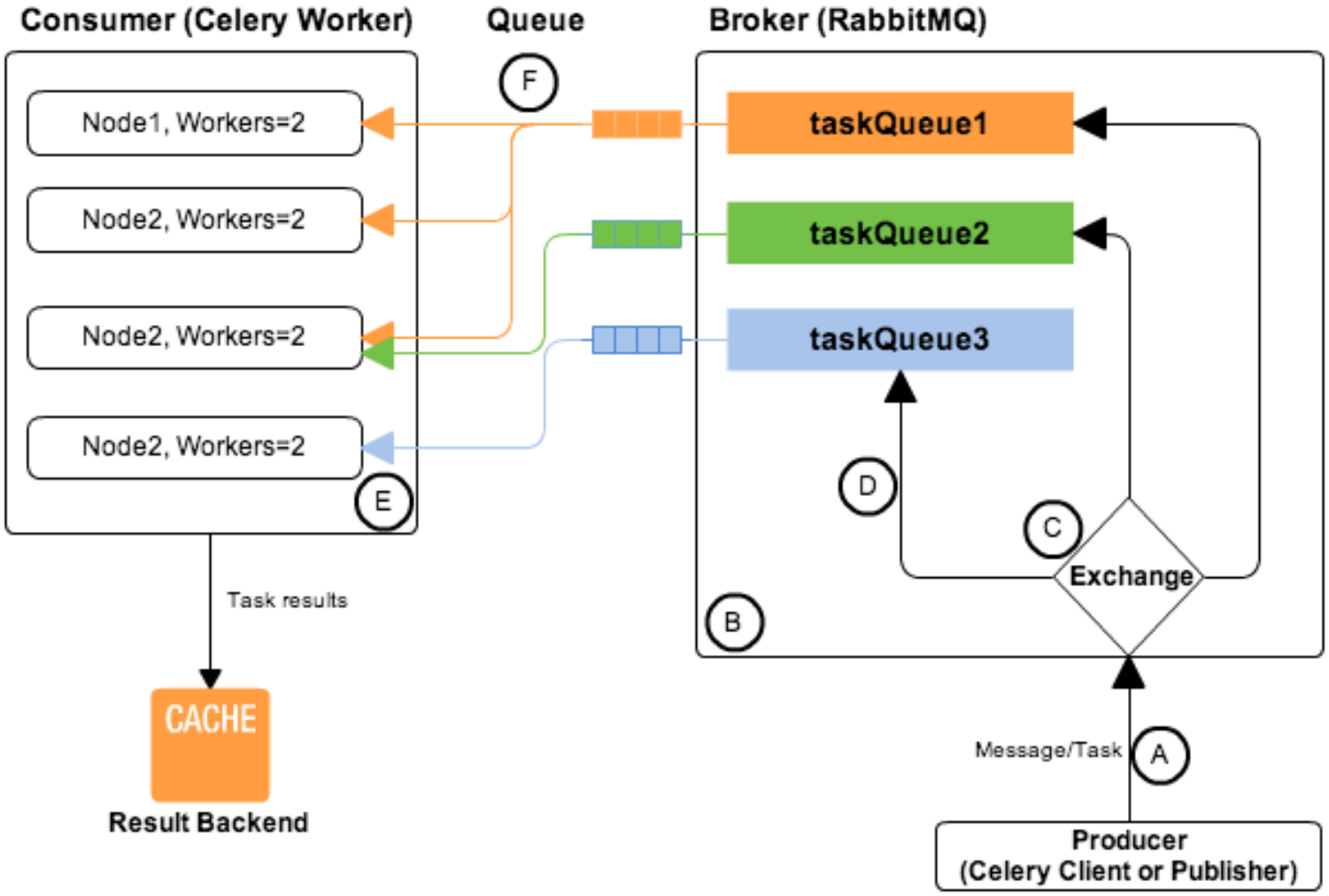


Figure: Source abhishek-tiwari.com

# Minimal Example – Create Celery App

```python
from celery import Celery

app = Celery(
    'tasks',
    broker='redis://localhost:6379/1',
    backend='redis://'
)

# Decorator creates a Celery task from a regular function
@app.task
def add(x, y):
    return x + y
```

# Minimal Example – Calling Tasks

```
1  from celery_example import add
2
3  # Fire up the task
4  add.delay(1, 4)
5
6  # Or a more sophisticated way
7  res = add.apply_async(args=(2, 4), queue='my_queue')
8
9  res.status  # Get status of the task
10 res.get()  # Wait for the result
```

# Task Orchestration with Signatures

- Celery supports complex workflows using different orchestration primitives
- **group, chain, chord, map, starmap, and chunks**
- for these, we need to be able to create a task instance with the required parameters but not execute it

```
1  from celery import signature
2
3  # Signature is a task with fixed parameters that
4  # can be called later (similar to partial in FP)
5  signature('tasks.add', args=(2, 2))
6
7  # Signature shortcut
8  s = add.s(2, 2)
9  s.delay()
```

# Supported Workflows

```python
@app.task
def xsum(numbers):
    return sum(numbers)

# 1. Chain passes result of the task to the next one.
(add.s(2, 2) | add.s(4) | add.s(8))().get() # → 16

# 2. Group executes a set of tasks in paralel.
group(add.s(i, i) for i in range(4))() # → [0, 2, 4, 6]

# 3. Chord is a group that takes a callback.
chord((add.s(i, i) for i in range(4)), xsum.s())() # → 12

# 4. Map applies task to a list of args in 1 message.
xsum.map([1, 2, 3], [7, 10]]) # → [6, 17]
```

# Example of a Complex Workflow

```
1  @app.task
2  def create_user(username, first, last, email):
3      ...
4  @app.task
5  def import_contacts(user):
6      ...
7  @app.task
8  def send_welcome_email(user):
9      ...
10
11 new_user_workflow = (
12     create_user.s() | group(import_contacts.s(),
13                             send_welcome_email.s()))
14
15 new_user_workflow.delay('asterix', 'Asterix', 'Gaul',
16                         'asterix@rychmat.eu')
```

# Problem Specification

- We have an online payment method where each order must go through non-trivial scoring process.
- Problems with synchronous code:
  1. Scoring may take up tu a minute.
  2. The computation is resource-heavy and must not affect processing of new orders.
  3. To increase throughput of the app, different scoring tasks must be run concurrently.
  4. Scoring cannot run in parallel for one customer.

# Implementation in Celery

```
1  # Setup Celery in Django settings
2  BROKER_URL = 'redis://localhost:6379/0'
3  BROKER_TRANSPORT = 'redis'
4  CELERY_SCORING_WORKERS = 10
5  CELERY_QUEUES = [
6      Queue(str(i), Exchange(str(i)), routing_key=str(i))
7      for i in range(CELERY_SCORING_WORKERS)]
8
9  # Define the scoring task
10 @app.task(name='run_scoring', time_limit=3600)
11 def execute_scoring(order):
12     ...
13 # Run scoring when new order arrives
14 execute_scoring.apply_async(
15     args=(order,),
16     queue=str(customer.pk % CELERY_WORKERS_COUNT))
```


# Thanks for your attention!

Code examples adapted from Celery docs.