

Celery

Live de Python #159



1. Celery

Entendendo a biblioteca

2. Um problema

Vamos problematizar

3. Tarefas

Um primeiro contato

4. Lidando com erros

Entendo melhor as tasks

5. Debug

Sim, as vezes é necessário

6. Cadeias

Agora começa a ficar interessante



picpay.me/dunossauro



apoia.se/livedepython



PIX



Ajude o projeto



Ademar Peixoto, Alex Lima, Alexandre Harano, Alexandre Santos, Alexandre Souza, Alexandre Tsuno, Alysso Oliveira, Amaziles Carvalho, Anderson Araujo, Andre Rodrigues, André Rocha, Antonio Neto, Arnaldo Turque, Bianca Rosa, Bruno Oliveira, Caio Nascimento, Carlos Cardoso, Carlos Chiarelli, César Almeida, César Moreira, Davi Ramos, David Kwast, Diego Guimarães, Dilenon Delfino, Douglas Bastos, Edgard Sampaio, Elias Soares, Eugenio Mazzini, Everton Alves, Fabio Barros, Fabio Castro, Fabrício Coelho, Flavkaze Flavkaze, Franklin Silva, Fábio Serrão, Gabriel Simonetto, Gabriel Soares, Gabriela Santiago, Geandreson Costa, Guilherme Felitti, Guilherme Marson, Guilherme Ostrock, Haroldo Júnior, Henrique Machado, Hélio Neto, Isaac Ferreira, Israel Fabiano, Italo Silva, Jeison Sanches, Johnny Tardin, Jonatas Leon, Jorge Plautz, José Prado, Jovan Costa, João Lugão, João Schiavon, Juan Gutierrez, Jônatas Silva, Júlia Kastrup, Kaneson Alves, Leonardo Cruz, Leonardo Galani, Leonardo Mello, Lorena Ribeiro, Lucas Barros, Lucas Ferreira, Lucas Mello, Lucas Mendes, Lucas Teixeira, Lucas Valino, Luiz Lima, Maiquel Leonel, Maiquel Leonel, Marcela Campos, Marcelo Rodrigues, Maria Clara, Natan Cervinski, Nicolas Teodosio, Nilo Pereira, Otavio Carneiro, Patric Lacouth, Patricia Minamizawa, Patrick Gomes, Paulo Tadei, Pedro Andrade, Pedro Pereira, Peterson Santos, Reinaldo Silva, Rodrigo Campos, Rodrigo Ferreira, Rodrigo Vaccari, Ronaldo Silva, Rubens Gianfaldoni, Sandro Mio, Silvio Xm, Thiago Araujo, Thiago Borges, Thiago Bueno, Tyrone Damasceno, Valdir Junior, Victor Geraldo, Vinícius Bastos, Vinícius Ferreira, Wesley Mendes, Willian Lopes, Willian Lopes, Willian Rosa, Wilson Duarte



Obrigado você



Vamos no modo mais lento possível. Sim, estamos
munidos de diversos exemplos



Disclaimer



Caso você já tenha familiaridade com o assunto, CALMA



Disclaimer 2



Uma
apresentação
inicial

Celery

O que é o Celery?



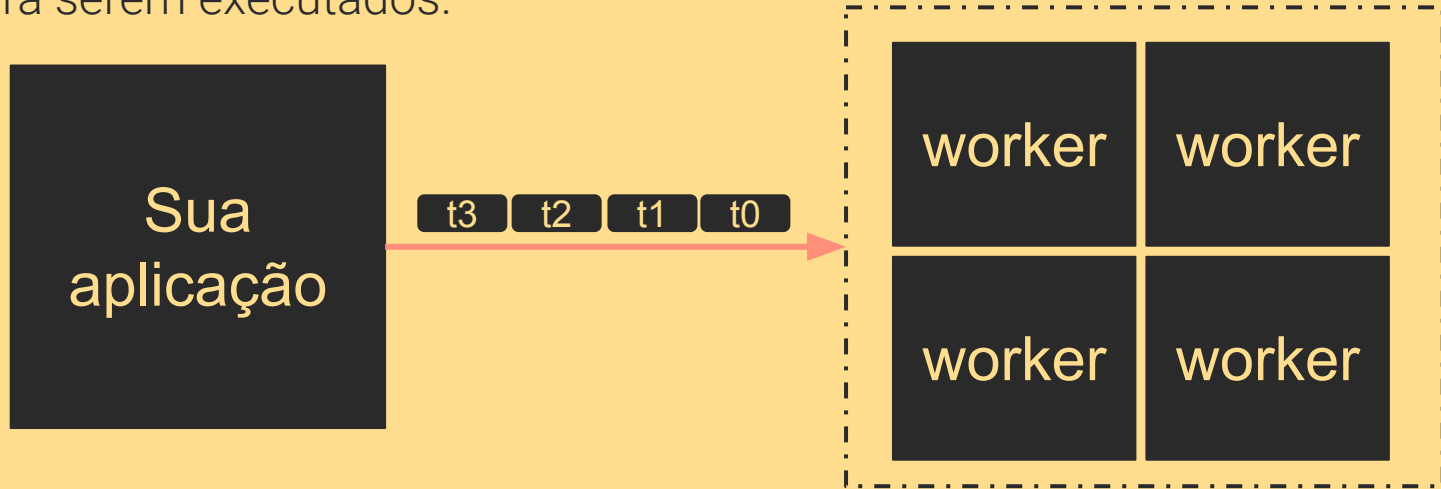
Celery é uma biblioteca para distribuir trabalhos de maneira assíncrona por meio de filas (***calma***)

- De código aberto (BSD-3)
- Atualmente na versão 5.0.5
- Sua primeira versão estável (1.0 é de Fev/2010)

Como o celery funciona?



O celery provê workers para executar tarefas em **background**, ou seja, você pode delegar blocos de código que "demoram" ou apresentam problemas para serem executados.



Como o celery funciona?



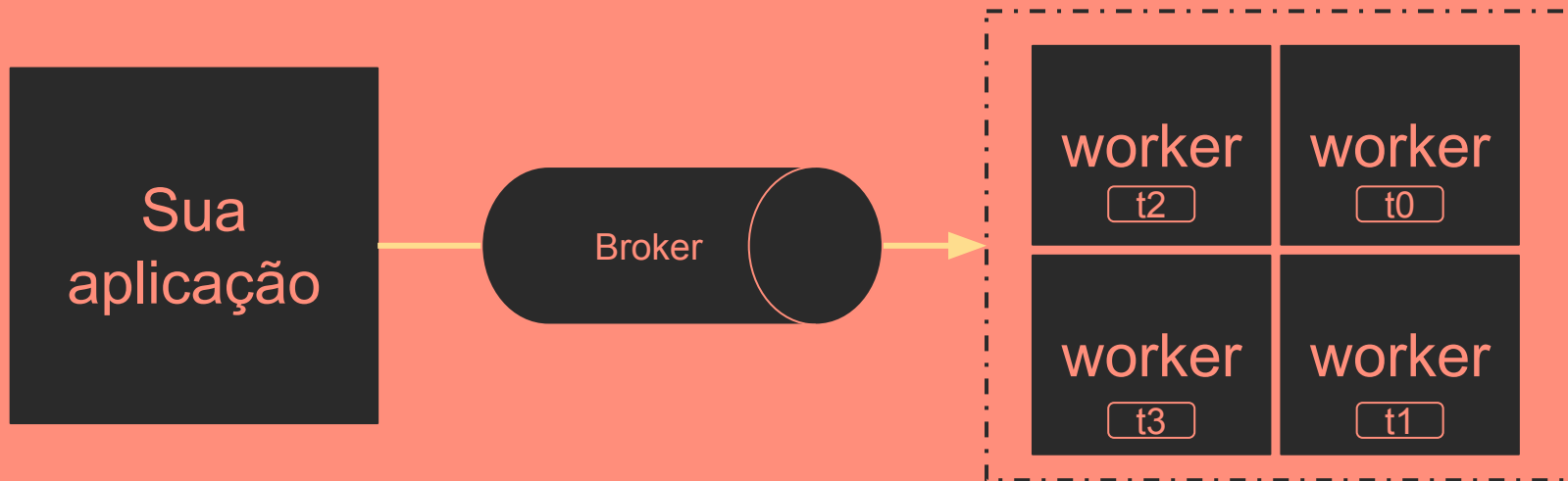
O celery provê workers para executar tarefas em background, ou seja, você pode delegar blocos de código que "demoram" ou apresentam problemas para serem executados.



Como o celery funciona?



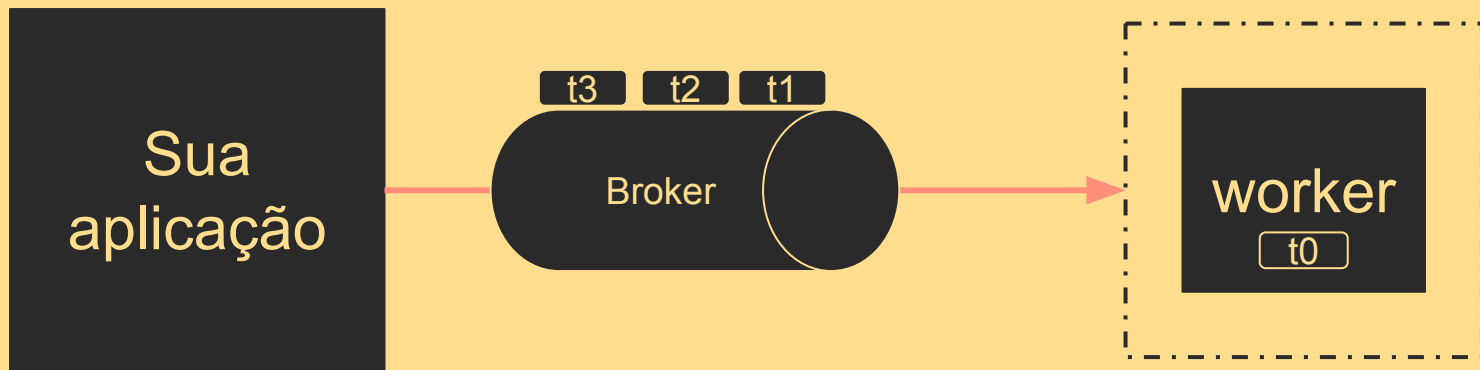
A integração com os workers se dá por um mecanismo de filas. Ou, como é mais conhecido, **broker**



Como o celery funciona?



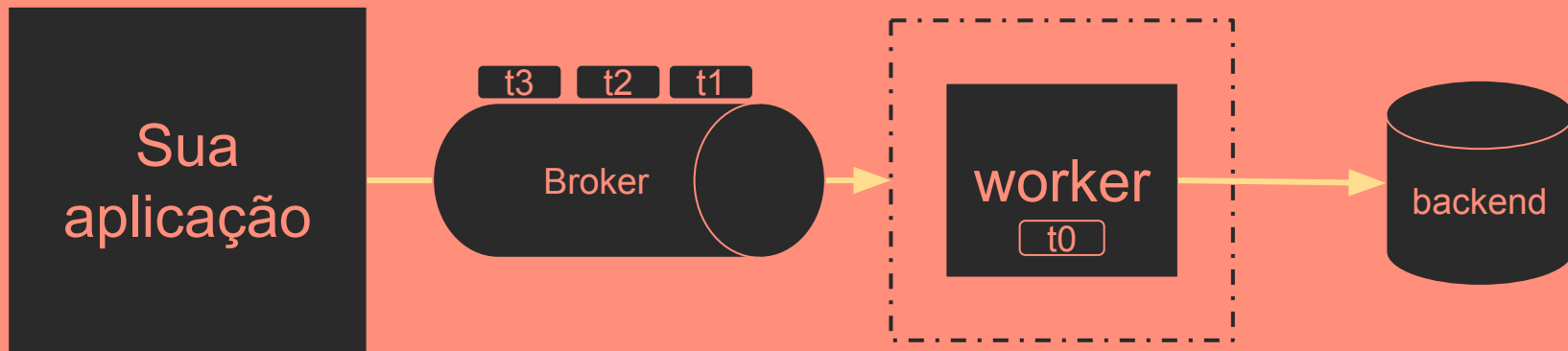
O mecanismo de filas garante que o worker executará somente uma tarefa por vez



Como o celery funciona?



Após finalizar a tarefa o celery pode ou não armazenar o resultado em um banco de dados, o banco de dados é chamado de **backend** na nomenclatura do celery



```
pip install celery
```



Então, bora instalar o celery



Configuração mínima do Celery



O celery é muito fácil de iniciar, porém ele precisa de um broker para começar a funcionar

```
1  from celery import Celery
2
3  app = Celery(
4      broker='pyamqp://guest@localhost//',
5  )
```

Suporte a brokers



O Celery tem suporte a diversos mecanismos de fila:

- RabbitMQ
- Redis
- Amazon SQS
- Apache Zookeeper (experimental)

Eu vou usar RabbitMQ



Existem diversas formas de instalar o RabbitMQ e ele é compatível com:

- Windows
- Linux
- Mac
- BSD
- Unix

<https://www.rabbitmq.com/download.html>

```
docker run -d -p 5672:5672  
rabbitmq
```



Eu vou usar docker <3



Com RabbitMQ rodando



Agora vamos criar nossa task de 'Olá Mundo' para ver se está tudo funcionando.

```
1 shell $ celery -A tasks worker
```

```
1 # tasks.py
2 from celery import Celery
3
4
5 app = Celery(
6     broker='pyamqp://guest@localhost//',
7 )
8
9
10 @app.task
11 def ola_mundo():
12     return 'Olá mundo'
```

Com RabbitMQ rodando



Agora vamos criar nossa task de 'Olá Mundo' para ver se está tudo funcionando.

```
1 shell $ celery -A tasks worker
```

--loglevel=INFO

```
1 # tasks.py
2 from celery import Celery
3
4
5 app = Celery(
6     broker='pyamqp://guest@localhost//',
7 )
8
9
10 @app.task
11 def ola_mundo():
12     return 'Olá mundo'
```

SIIMMM, hora de executar



Agora vamos ver essa belezinha rodando, deu trabalho, né? Eu sei

```
1  # app.py
2  from .tasks import ola_mundo
3
4  ola_mundo.delay()
```

pip install flower



Vamos ver isso de maneira bonita?



Subindo o flower



Para iniciar o flower é simples, podemos rodar esse comando no terminal.

Agora na porta 5555 temos um modo visual onde podemos ver as tasks e seus resultados

```
1  shell $ flower -A tasks
```

Um
problema

Para
contextualizar

Como sei que existem muitos conceitos envolvidos para entender o celery, vamos criar um problema fictício e trabalhar nele.

Por meio deste problema, vamos conseguir chegar no entendimento do que o celery significa e o que ele faz de fato



Disclaimer



0 problema



Imagine uma tarefa que leva tempo demais para ser feita. Como essa:

```
1  from time import sleep
2
3
4  def soma_com_delay(x, y):
5      sleep(10)
6      return x + y
```

0 problema



Imagine uma tarefa que leva tempo demais para ser feita. Como essa:

```
1 from time import sleep
2
3
4 def soma_com_delay(x, y):
5     sleep(10)
6     return x + y
```

Delay inserido para
demonstração

Exemplos mais concretos



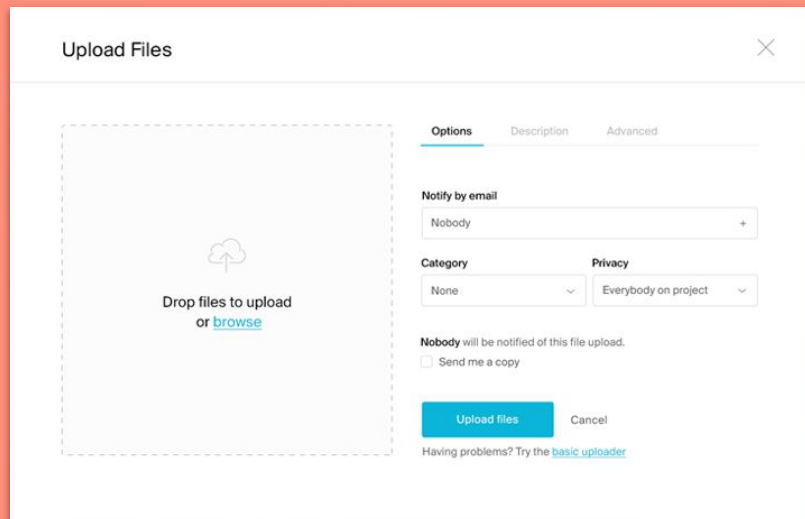
```
1  def receber_formulário():  
2      formulário = request.form()          # 1 segundo  
3  
4      processar_formulário(formulário)     # 5 minutos  
5  
6      return 'Formulário recebido com sucesso' # 1 segundo
```

Por que isso levaria 5 minutos?



Como uma coisa pode levar 5 minutos para acontecer?

Imagine que nesse formulário você tenha que enviar a foto de um documento seu.

A screenshot of a web application's 'Upload Files' dialog box. The dialog has a title bar with 'Upload Files' and a close button. The main area is divided into two sections. The left section is a large dashed box containing a cloud icon with an upward arrow and the text 'Drop files to upload or [browse](#)'. The right section contains settings for the upload. It has three tabs: 'Options' (selected), 'Description', and 'Advanced'. Under 'Options', there is a 'Notify by email' dropdown set to 'Nobody' with a plus icon to its right. Below that is a 'Category' dropdown set to 'None' and a 'Privacy' dropdown set to 'Everybody on project'. A note states 'Nobody will be notified of this file upload.' with a checkbox 'Send me a copy' which is unchecked. At the bottom of the right section are two buttons: 'Upload files' (highlighted in blue) and 'Cancel'. At the very bottom of the dialog, it says 'Having problems? Try the [basic uploader](#)'.

Seu documento

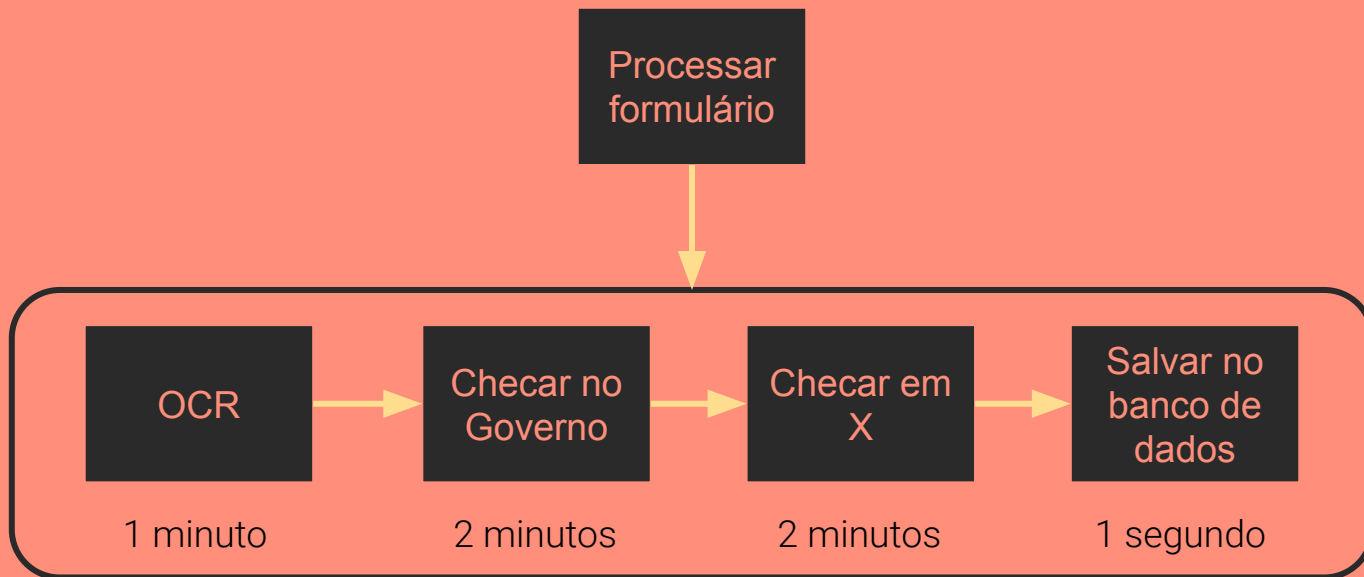


Seu documento



- Nome
- CPF
- Número de registro
- Nome do Pai
- Nome da Mãe

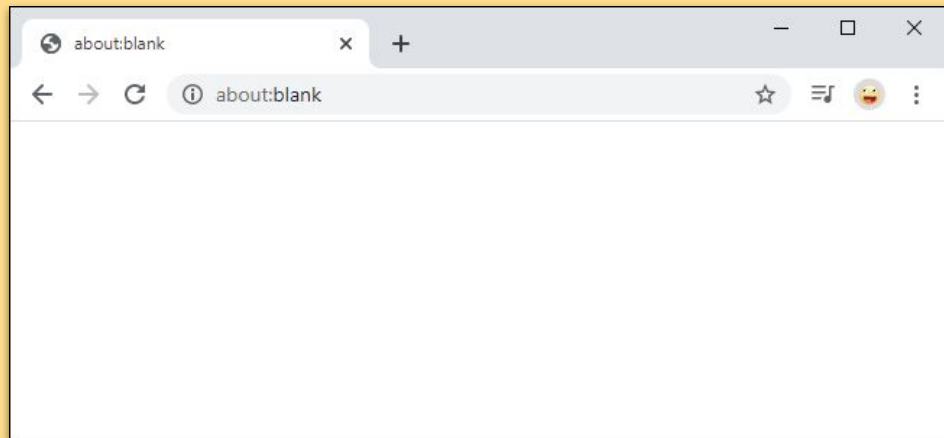
Seu processo



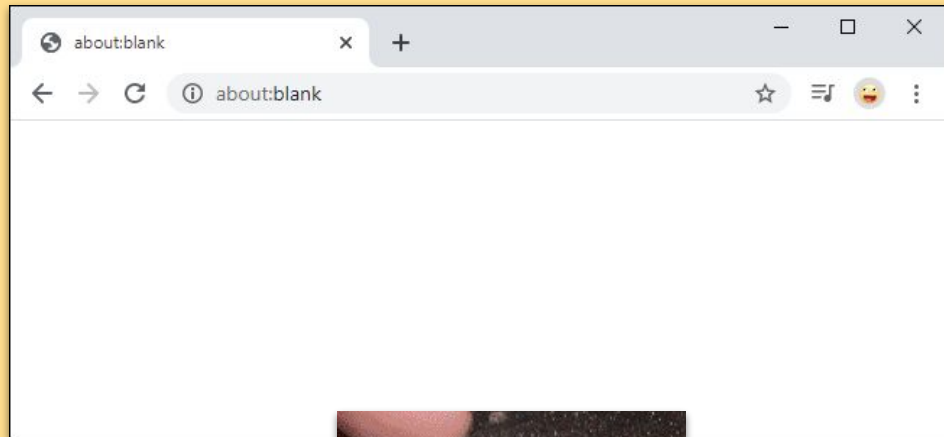
A pessoa esperando o formulário



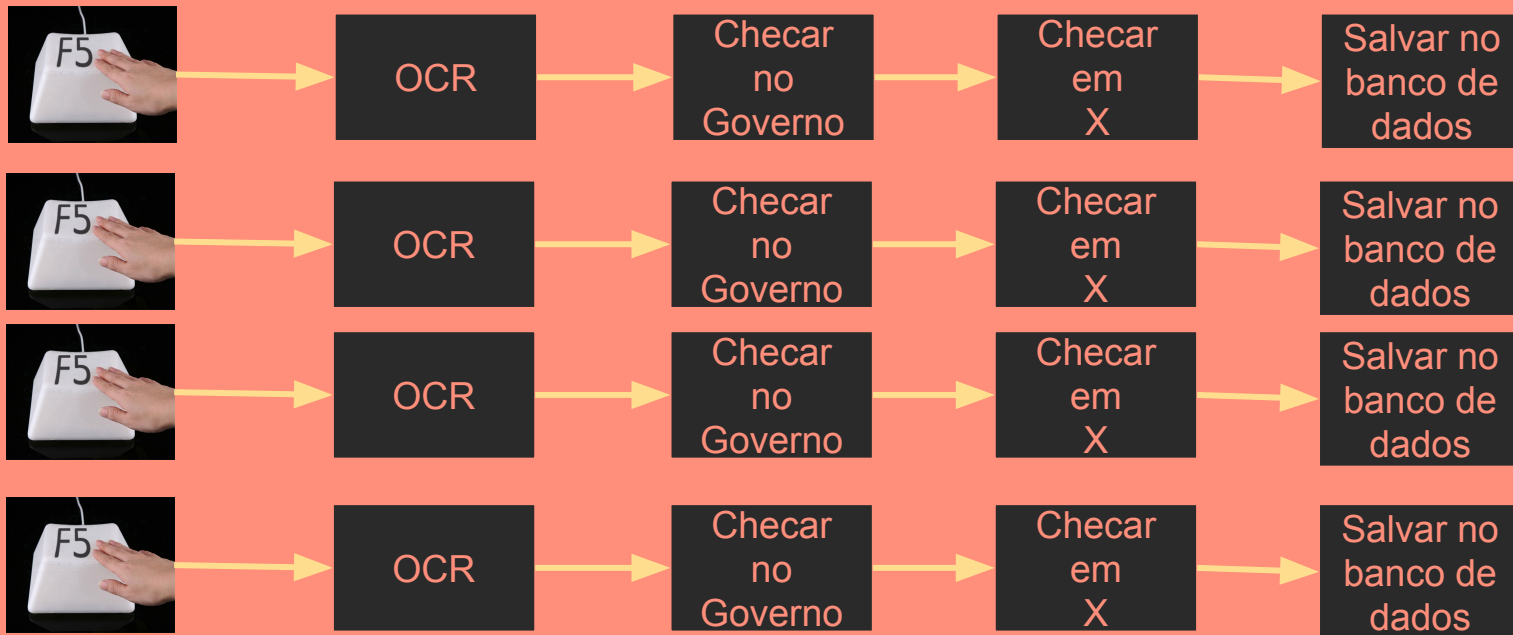
A tela que ela vê



O que ela faz?



Como o sistema responde?



No final, o que acontece?



Mas e se?



Beleza, 5 minutos





- A foto estiver ruim?
- O governo estiver fora do ar?
- Os dados estiverem errados?
- X não encontrar o dado?
- Se a resposta demorar mais que o normal?

Se der algo errado? Ainda assim vão ser 5 minutos?

OCR

Checar no
Governo

Checar em
X

Tarefas

Um primeiro
contato

Mudando o fluxo



Uma ideia que podemos aderir, é usar tarefas em background. Ou seja, executar em outro computador, por exemplo. Externalizando essa dependência e deixando a pessoa que usa o nosso sistema saber se algo aconteceu depois (**postponing**).

O caso do e-mail



Se lembra quando esquecemos nossa senha e o sistema nos diz que vai nos enviar um e-mail? Ele está fazendo exatamente isso.

```
1  def esqueci_minha_senha(email):  
2      user = buscar_no_banco(email) # 1 segundo  
3  
4      enviar_email_nova_senha(user) # Tarefa não bloqueante  
5  
6      return 'Você receberá um e-mail em instantes' # 1 segundo
```

Tá, mas como vamos fazer essa tarefa
ser executada em segundo plano?



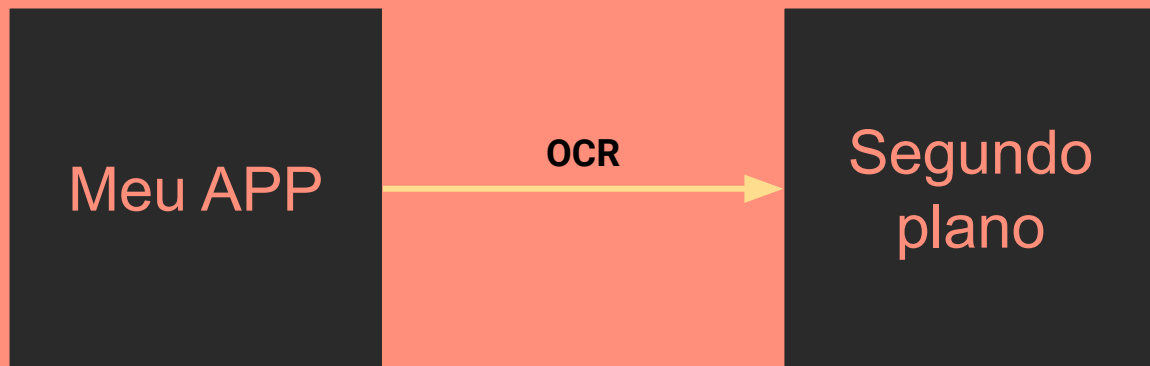
bla bla bla



Entendendo o fluxo



A ideia é termos **alguém** que consegue rodar a nossa tarefa em segundo plano.



Bora codar!

Vamos imaginar aquele formato de cadastro que usamos antes.

API:

live-159-external.herokuapp.com/

```
1  from dataclasses import dataclass
2
3
4  @dataclass
5  class Pessoa:
6      # Live de python #150
7      nome: str
8      telefone: str
9      documento: str
10
11
12  def cadastro(pessoa: Pessoa):
13      

OCR


14      

Checar  
no  
Governo


```

Erros

Entendo melhor
as tasks

Erros



Bom, erros podem acontecer a qualquer momento, não precisamos nem de um sistema distribuído para isso. Porém, as possibilidades de tratamentos de erros acabam ficando mais complexas a medida em que vamos adicionando mais componentes.



O decorador de task fornece vários mecanismos que nos auxiliam a manter a **confiabilidade** da nossa execução e ter um sistema mais **disponível**.

- bind
- max_retry
- default_retry_delay
- autoretry_for
- retry_backoff

@task(bind=True)

De todos os argumentos, acho que o **bind** é o mais interessante, pois ele permite que nós tenhamos acesso ao objeto **task** que faz o warper da nossa função

```
1  @app.task(bind=True)
2  def minha_task(self):
3      # Reexecuta a task (para caso de erro)
4      self.retry()
5
6      # Diz se foi chamado por outra task
7      self.request.called_directly
8
9      # Atualiza o estado da task
10     self.update_state(state='SUCCESS')
11
12     # Contém diversas informações interessantes
13     self.request
```

Retries



Bom, as vezes as coisas precisam rodar novamente, infelizmente.

```
1  @app.task(  
2      bind=True,  
3      max_retry=5,          # Tentará no máximo 5 vezes  
4      default_retry_delay=20, # tempo entre as tentativas  
5      # Auto retry caso algum erro na tupla aconteça  
6      autoretry_for=(TypeError,Exception),  
7  )  
8  def minha_task(self):  
9      self.retry() # reexecuta uma task
```

Backoff



A regra mais importante de quando os erros acontecem é dar um tempo. Imagine que o sistema esteja lento por conta um de um ataque, você forçando só piorará a situação do serviço.

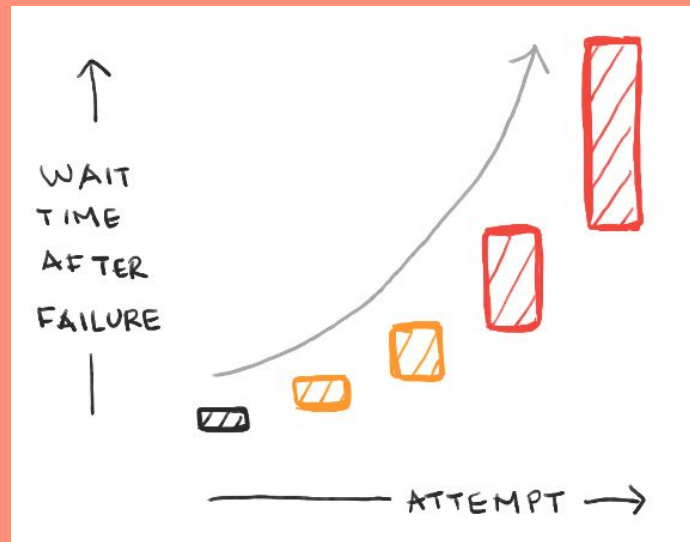
Ele também pode estar em manutenção, o que nos faria mandar diversos requests mesmo "sabendo" que não seremos respondidos

Backoff exponencial



Backoff é a arte de esperar um pouco mais entre uma request e outra.

Se o backoff estiver habilitado, a primeira tentativa será em 1 seg. A segunda em 2 seg, a terceira em 4 seg, ...



Backoff com celery



Com um número podemos iniciar a distribuição uniforme com um valor.

3, 6, 16, 24, 48, ...

```
1  @app.task(  
2      retry_backoff=True,  
3      # ou  
4      retry_backoff=3,  
5  )  
6  def minha_task(self):  
7      self.retry() # reexecuta uma task
```

Nomes



Embora essa seja a última, ela não é a menos importante, isso afeta drasticamente como iremos ler o nome de nossas tasks no futuro

```
1  @app.task(name='Task fododa')
2  def minha_task(self):
3      ...
```

As vezes é
necessário

Debug

Debug



Felizmente o celery já vem com um debugger nativo. Muitas vezes você pode executar a função de maneira simples, mas problemas acontecem, vocês sabem

```
1  from celery.contrib import rdb
2
3  @app.task
4  def task_para_debugar():
5      # Só em localhost
6      rdb.set_trace()
```


Debug



Após a task ser executada, o worker vai nos dizer qual a porta

```
1 from celery.contrib impo
2
3 @app.task
4 def task_para_debugar():
5     # Só em localhost
6     rdb.set_trace()
```

```
1 Remote Debugger:6907: Ready to connect: telnet 127.0.0.1 6907
2
3 Type `exit` in session to continue.
4
5 Remote Debugger:6907: Waiting for client...
6
7 [INFO/MainProcess] Events of group {task} enabled by remote.
8
```

Debug



```
1 Remote Debugger:6907: Ready to connect: telnet 127.0.0.1 6907
```

```
2
```

```
3 Type `exit` in occasion to continue  
4 dunossauro at babbage in ~/git/live_celery
```

```
5 Remote $ telnet localhost 6907
```

```
6 Trying ::1...
```

```
7 [INFO/M Connection failed: Conexão recusada
```

```
8 Trying 127.0.0.1...
```

```
Connected to localhost.
```

```
Escape character is '^['.
```

```
> /home/dunossauro/git/live_celery/app/tasks.py(44)ocr_documento()
```

```
-> 'image': standard_b64encode(image).decode('utf-8'),
```

```
(Pdb) █
```

Uma olhada breve

Chain

Chain

A ideia do chain é montar um pipeline de tasks. O resultado de uma task é o primeiro parâmetro de outro. Com isso podemos diminuir bastante a complexidade do nosso código.

A -> B

```
1  from celery import chain
2
3  @app.task
4  def a(x):
5      return x * 4
6
7  @app.task
8  def b(x, y):
9      return x + y
10
11  chain(a.s(1), b.s(2))()
12  # b(a(1), 2)
```

- Celery beat
 - agendamento de tarefas
- Pika
 - alternativa ao celery
- Backend
 - Armazenar resultados em bancos de dados
- Redis
 - Um banco de chaves e valores que pode ser usado como queue e como backend
- Chain



Do que não falamos?





picpay.me/dunossauro



apoia.se/livedepython



PIX



Ajude o projeto

