CEPs

Celery's Techincal Board

Aug 17, 2021

CONTENTS:

1 Celery Enhancement Proposals (CEPs) 1

- 1.1 Glossary . . . 1
- 1.2 Final CEPs . . . 4
- 1.3 Approved CEPs . . . 12
- 1.4 Draft CEPs . . . 12
- 1.5 Rejected CEPs . . . 70
- 1.6 Superseded CEPs . . . 71
- 1.7 Withdrawn CEPs . . . 71
- 1.8 CEP XXXX: CEP template . . . 71

2 Indices and tables 75

Index 77

CHAPTER

ONE

CELERY ENHANCEMENT PROPOSALS (CEPS)

Celery Enhancement Proposals are a formal way of proposing large feature additions to the Celery Project (http://www.celeryproject.org/).

See CEP 1 for details.

1.1 Glossary

Message Broker Enterprise Integration Patterns defines a Message Broker as an architectural building block that can receive *messages* from multiple destinations, determine the correct destination and route the message to the correct channel.

Message Enterprise Integration Patterns defines a Message as data record that the messaging system can transmit through a message channel.

Command Message Enterprise Integration Patterns defines a Command Message as a *Message* which instructs a worker to execute a task.

Event Message Enterprise Integration Patterns defines an Event Message as a *Message* which indicates that an event has occurred.

Document Message Enterprise Integration Patterns defines a Document Message as a *Message* containing data from a data source.

Service Activator Enterprise Integration Patterns defines a Service Activator as a one-way (request only) or two-way (request-reply) adapter between the *Message* and the service it invokes. The service can be a simple as a method call. The activator handles all of the messaging details and invokes the service like any other client, such that the service doesn't even know it's being invoked through messaging.

Idempotent Receiver Enterprise Integration Patterns defines an Idempotent Receiver as a component that can safely receive the same message multiple times but will produce the same side effects when facing duplicated messages.

Message Dispatcher Enterprise Integration Patterns defines a Message Dispatcher as a component that will consume messages from a channel and distribute them to performers.

Process Manager Enterprise Integration Patterns defines a Process Manager as a component that maintains the state of the sequence and determines the next processing step based on intermediate results.

Event Driven Consumer Enterprise Integration Patterns defines an Event Driven Consumer as a component that consumes a message as soon as it is delivered.

Circuit Breaker Martin Fowler defines a Circuit Breaker in the following fashion:

> The basic idea behind the circuit breaker is very simple. You wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all. Usually you'll also want some kind of monitor alert if the circuit breaker trips.

CAP Theorem The CAP theorem categorizes systems into three categories:

- CP (*Consistent* and *Partition Tolerant*) — At first glance, the CP category is confusing, i.e., a system that is consistent and partition tolerant but never available. CP is referring to a category of systems where availability is sacrificed only in the case of a network partition.
- CA (*Consistent* and *Available*) — CA systems are consistent and available systems in the absence of any network partition. Often a single node's DB servers are categorized as CA systems. Single node DB servers do not need to deal with partition tolerance and are thus considered CA systems. The only hole in this theory is that single node DB systems are not a network of shared data systems and thus do not fall under the preview of CAP.
- AP (*Available* and *Partition Tolerant*) — These are systems that are available and partition tolerant but cannot guarantee consistency.

Consistency A guarantee that every node in a distributed cluster returns the same, most recent, successful write. Consistency refers to every client having the same view of the data. There are various types of consistency models. Consistency in CAP (used to prove the theorem) refers to linearizability or sequential consistency, a very strong form of consistency.

Availability Every non-failing node returns a response for all read and write requests in a reasonable amount of time. The key word here is every. To be available, every node on (either side of a network partition) must be able to respond in a reasonable amount of time.

Partition Tolerant The system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

Fault Tolerance TODO

Network Resilience According to Wikipedia Network Resilience is:

> In computer networking: resilience is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation." Threats and challenges for services can range from simple misconfiguration over large scale natural disasters to targeted attacks. As such, network resilience touches a very wide range of topics. In order to increase the resilience of a given communication network, the probable challenges and risks have to be identified and appropriate resilience metrics have to be defined for the service to be protected.

Monitoring According to fastly monitoring is:

> The activity of observing the state of a system over time. It uses instrumentation for problem detection, resolution, and continuous improvement. Monitoring alerts are reactive–they tell you when a known issue has already occurred (i.e. maybe your available memory is too low or you need more compute). Monitoring provides automated checks that you can execute against a distributed system to make sure that none of the things you predicted signify any trouble. While monitoring these known quantities is important, the practice also has limitations, including the fact that you are only looking for known issues. Which begs an important question, "what about the problems that you didn't predict?"

Observability According to Wikipedia in the context of control theory Observability is:

> In control theory, observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs.

In the context of distributed systems observability is a super-set of *Monitoring*.

According to fastly the three pillars of observability are:

> Logs: Logs are a verbose representation of events that have happened. Logs tell a linear story about an event using string processing and regular expressions. A common challenge with logs is that if you haven't properly indexed something, it will be difficult to find due to the sheer volume of log data. Traces: A trace captures a user's journey through your application. Traces provide end-to-end visibility and are useful when you need to identify which components cause system errors, find performance bottlenecks, or monitor flow through modules. Metrics: Metrics can be either a point in time or monitored over intervals. These data points could be counters, gauges, etc. They typically represent data over intervals, but sometimes sacrifice details of an event in order to present data that is easier to assimilate.

Structured Logging Structured Logging is a method to make log messages easy to process by machines. A usual log message is a timestamp, level and a message string. The context describing the logged event is embedded inside the message string. A structured log message store their context in a predetermined message format which allows machines to parse them more easily.

JSON JSON stands for JavaScript Object Notation, which is a way to format data so that it can be transmitted from one place to another, most commonly between a server and a Web application.

stdout Stdout, also known as standard output, is the default file descriptor where a process can write output.

Service Locator Martin Fowler defines a Service Locator in the following fashion:

> The basic idea behind a service locator is to have an object that knows how to get hold of all of the services that an application might need. So a service locator for this application would have a method that returns a movie finder when one is needed.

GIL The Global Interpreter Lock, abbreviated as the GIL is a mutex which prevents executing threads in parallel if both are about to execute a python bytecode.

This is by design since Python has many atomic operations and maintaining individual locks on each object results in slower execution.

Depending on the implementation, a thread may be forced to release the GIL when a condition is met. In CPython's implementation of Python 3, a thread is forced to release the GIL after a it executes for a period of time.

A thread may also release the GIL voluntarily when it uses a system call or when a C extension instructs to do so.

IPC According to Wikipedia Inter-process Communication:

> refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests. Many applications are both clients and servers, as commonly seen in distributed computing.
>
> There are many approaches to IPC. Some of them are available in all operating systems, some are only available in specific operating systems.

Task A task is a unit of business logic that is completely independent and can be executed regardless of the execution platform.

Domain Model Martin Fowler defines a Domain Model in the following fashion:

> An object model of the domain that incorporates both behavior and data.

Domain Event Martin Fowler defines a Domain Event in the following fashion:

> I go to Babur's for a meal on Tuesday, and pay by credit card. This might be modeled as an event, whose event type is 'make purchase', whose subject is my credit card, and whose occurred date is Tuesday. If Babur's uses an old manual system and doesn't transmit the transaction until Friday, the noticed date would be Friday.

Things happen. Not all of them are interesting, some may be worth recording but don't provoke a reaction. The most interesting ones cause a reaction. Many systems need to react to interesting events. Often you need to know why a system reacts in the way it did.

By funneling inputs to a system into streams of Domain Event you can keep a record of all the inputs to a system. This helps you organize your processing logic, and also allows you to keep an audit log of the inputs to the system.

Serverless Computing TODO

Ubiquitous Language TODO

Result Backend TODO

Celery Master TODO

Celery Worker TODO

Celery Multi TODO

Celery Beat TODO

Flower TODO

Cell TODO

ETL TODO

Data Integration TODO

Python Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

CPython CPython is the reference implementation of the Python programming language. Written in C and Python, CPython is the default and most widely used implementation of the language.

PyPy PyPy is a replacement for CPython. It is built using the RPython language that was co-developed with it. The main reason to use it instead of CPython is speed: it runs generally faster.

1.1.1 Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

1.2 Final CEPs

CEPs that have been approved and fully implemented. See CEP 1 for details.

1.2.1 CEP 1: CEP Purpose and Guidelines

CEP 1

Author Omer Katz

Status Final

Type Process

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *What is a CEP?*
- *CEP Types*
- *CEP submission workflow*
  - *Pre-proposal*
  - *Forming the team*
  - *Submitting the draft*
  - *Discussion, development, and updates*
  - *Review & Resolution*
  - *Implementation*
- *CEP format*
  - *CEP Metadata*
  - *Auxiliary Files*
- *Reporting CEP Bugs, or Submitting CEP Updates*
- *Transferring CEP Ownership*
- *Differences between CEPs and PEPs*
- *Copyright*

What is a CEP?

CEP stands for Celery Enhancement Proposal. A CEP is a design document providing information to the Celery community, or describing a new feature or process for Celery. CEPs provide concise technical specifications of features, along with rationales.

We intend CEPs to be the primary mechanisms for proposing major new features, for collecting community input on issues, and for documenting design decisions that have gone into Celery.

The concept and implementation of CEPs (and this document itself!) is a nearly direct copy of Python's PEP process . If you're already familiar with PEPs, you should be able to quickly grok CEPs by reading the *differences between CEPs and PEPs*.

CEP Types

There are three kinds of CEPs:

1. A **Feature** CEP describes a new feature or implementation for Celery. Most CEPs will be Feature CEPs.
2. An **Informational** CEP describes a Celery design issue, or provides general guidelines or information to the Celery community but does not propose a new feature. Informational CEPs do not necessarily represent a community consensus or recommendation, so users and implementers are free to ignore Informational CEPs or follow their advice.
3. A **Process** CEP describes a process surrounding Celery, or proposes a change to (or an event in) a process. Process CEPs are like Feature CEPs but apply to areas other than the Celery framework itself. They may propose an implementation, but not to Celery's codebase; they often require community consensus; unlike Informational CEPs, they are more than recommendations, and users are typically not free to ignore them. Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in Celery development. Any meta-CEP is also considered a Process CEP. (So this document is a Process CEP).

CEP submission workflow

So, you'd like to submit a CEP? Here's how it works, and what to expect.

There are a couple of terms you should be familiar with before reading the rest of this document:

The Technical Board There are several reference in this CEP to the **Technical Board** (sometimes just "the Board"). This refers to Celery's Technical Board, the group of experienced and active committers who steer technical choices.

Core Developers Similarly, there are several references to **Core Developers** (sometimes "core devs"). This refers to the members of Celery's core team, and specifically those with commit access.

At a very high level, the CEP submission process looks like this:

1. *Pre-proposal* — someone has an idea and starts collecting early input and feedback to see if it's worth writing a CEP.
2. *Forming the team* — the CEP author rounds up the help they'll need to get the CEP considered.
3. *Submitting the draft* — the CEP author writes a rough draft of the CEP and submits it via pull request.
4. *Discussion, development, and updates* — the CEP and reference implementation are discussed, improved, and updated as feedback comes in.
5. *Review & Resolution* — the CEP is reviewed by the Technical Board and either accepted or rejected.
6. *Implementation* — the implementation of the proposed feature is completed by the CEP team.

For details on each step, read on.

Pre-proposal

The CEP process begins with a new idea for Celery. It is highly recommended that a single CEP contain a single key proposal or new idea. Small enhancements or patches usually don't need a CEP and follow Celery's normal contribution process.

The more focused the CEP, the more successful it tends to be. The Core Developers reserve the right to reject CEP proposals if they appear too unfocused or too broad. If in doubt, split your CEP into several well-focused ones.

The CEP Author (see below for the formal definition of an Author) should first attempt to ascertain whether the idea is CEP-able. Opening an issue on the celery/ceps repository is the best way to do so.

Vetting an idea publicly before going as far as writing a CEP is meant to save the potential author time. Many ideas have been brought forward for changing Celery that have been rejected for various reasons. Asking the Celery community first if an idea is original helps prevent too much time being spent on something that is guaranteed to be rejected based on prior discussions (searching the Internet does not always do the trick). It also helps to make sure the idea is applicable to the entire community and not just the author. Just because an idea sounds good to the author does not mean it will work for most people in most areas where Celery is used.

Forming the team

Once a CEP has been roughly validated, the author needs to fill out three vital roles. These roles will be required to get a CEP read, approved, and the code developed, so you need to identify up-front who will do what. These roles are:

Author The **Author** writes the CEP using the style and format described below (see *CEP format*), shepherds the discussions in the appropriate forums, and attempts to build community consensus around the idea.

Implementation Team The **Implementation Team** are the people (or single person) who will actually implement the thing being proposed. A CEP may have multiple implementers (and the best CEPs probably will).

Feature CEPs must have an implementation team to be submitted. Informational CEPs generally don't have implementers, and Process CEPs sometimes will.

Shepherd The **Shepherd** is the Core Developer who will be the primary reviewer of the CEP on behalf of the Celery team, will be the main point person who will help the Author assess the fitness of their proposal, and is the person who will finally submit the CEP for pronouncement by the Technical Board. When the implementation team doesn't contain someone who can commit to Celery, the Shepherd will be the one who actually merges the code into the project.

It's normal for a single person to fulfill multiple roles – in most cases the Author will be an/the Implementer, and it's not uncommon for the implementation team to include the Shepherd as well. It's unusual but acceptable for a single person to fulfill all roles, though this generally only happens when that person is a long-time committer.

Submitting the draft

Once the idea's been vetted and the roles are filled, a draft CEP should be presented to Celery-developers. This gives the author a chance to flesh out the draft CEP to make sure it's properly formatted, of high quality, and to address initial concerns about the proposal.

Following the discussion on Celery-developers, the proposal should be sent as a GitHub pull request to the celery/ceps repository. This PR should add a CEP to the `drafts/` directory, written in the style described below. The draft must be written in CEP style; if it isn't the pull request may be rejected until proper formatting rules are followed.

At this point, a core dev will review the pull request. In most cases the reviewer will be the Shepherd of the CEP, but if that's not possible for some reason the author may want to ask on Celery-developers to ensure that this review happens quickly. The reviewer will do the following:

- Read the CEP to check if it is ready: sound and complete. The ideas must make technical sense, even if they don't seem likely to be accepted.
- Make sure the title accurately describes the content.
- Check the CEP for language (spelling, grammar, sentence structure, etc.), markup, and code style (examples should match PEP 8).

If the CEP isn't ready, the reviewer can leave comments on the pull request, asking for further revisions. If the CEP's really in bad form, the reviewer may reject the pull request outright and ask the author to submit a new one once the problems have been fixed.

The reviewer doesn't pass judgment on CEPs. They merely do the administrative & editorial part (which is generally a low volume task).

Once the CEP is ready for the repository, the reviewer will:

- Merge the pull request.
- Assign a CEP number (almost always just the next available number), and rename the CEP file with the new number (e.g. rename `dep-process.rst` to `0001-dep-process.rst`)

Developers with commit access to the CEPs repo may create drafts directly by committing and pushing a new CEP. However, when doing so they need to take on the tasks normally handled by the reviewer described above. This includes ensuring the initial version meets the expected standards for submitting a CEP. Of course, committers may still choose to submit CEPs as a pull request to benefit from peer review.

Discussion, development, and updates

At this point there will generally be more discussion, modifications to the reference implementation, and of course updates to the CEP. It's rare for a CEP to be judged on the first draft; far more common is several rounds of feedback and updates.

Updates to a CEP can be submitted as pull requests; once again, a core developer will merge those pull requests (typically they don't require much if any review). In cases where the Author has commit access (fairly common), the Author should just update the draft CEP directly.

Feature CEPs generally consist of two parts, a design document and a reference implementation. It is generally recommended that at least a prototype implementation be co-developed with the CEP, as ideas that sound good in principle sometimes turn out to be impractical when subjected to the test of implementation.

CEP authors are responsible for collecting community feedback on a CEP before submitting it for review. However, wherever possible, long open-ended discussions on the relevant issue should be avoided. Strategies to keep the discussions efficient include: setting up a separate communication channel for the topic, having the CEP author accept private comments in the early design phases, setting up a wiki page, etc. CEP authors should use their discretion here.

Review & Resolution

Once the author has completed a CEP, the shepherd will ask the Technical Board for review and pronouncement. The final authority for deciding on a CEP rests with the Technical Board. They may choose to rule on a CEP as a team, or they may designate one or more board members to review and decide.

Having the shepherd (i.e. a core dev) rather than the author ask helps ensure that the CEP meets the basic technical bar before it's called for review. It also provides a fairly strong fitness test before the board is asked to rule on it, making board rulings fairly easy. If the core developer shepherd is happy, the board will likely be as well.

For a CEP to be accepted it must meet certain minimum criteria. It must be a clear and complete description of the proposed enhancement. The enhancement must represent a net improvement. The proposed implementation, if applicable, must be solid and must not complicate Celery unduly. Finally, a proposed enhancement must "fit" with Celery's general philosophy and architecture. This last category is the most imprecise and takes the most judgment, so if the Board rejects a CEP for lack of "fit" they should provide a clear explanation for why.

At this point, the CEP will be considered "Accepted" and moved to the `accepted` directory in the CEPs repo.

A CEP can also be "Withdrawn". The CEP author or a core developer can assign the CEP this status when the author is no longer interested in the CEP, or if no progress is being made on the CEP. Once a CEP is withdrawn, it's moved to the `withdrawn` directory for reference. Later, another author may resurrect the CEP by opening a pull request, updating (at least) the author, and moving it back to `draft`.

Finally, a CEP can also be "Rejected". Perhaps after all is said and done it was not a good idea. It is still important to have a record of this fact. Rejected CEPs will be moved to the `rejected` directory, and generally should be updated with a rationale for rejection.

CEPs can also be superseded by a different CEP, rendering the original obsolete. This is intended for Informational CEPs, where version 2 of an API can replace version 1.

Implementation

Finally, once a CEP has been accepted, the implementation must be completed. In many cases some (or all) implementation will actually happen during the CEP process: Feature CEPs will often have fairly complete implementations before being reviewed by the board. When the implementation is complete and incorporated into the main source code repository, the status will be changed to "Final" and the CEP moved to the `final` directory.

CEP format

To save everyone time reading CEPs, they need to follow a common format and outline; this section describes that format. In most cases, it's probably easiest to start with copying the provided CEP template, and filling it in as you go.

CEPs must be written in reStructuredText (the same format as Celery's documentation).

Each CEP should have the following parts:

1. A short descriptive title (e.g. "canvas-dsl"), which is also reflected in the CEP's filename (e.g. `0181-canvas-dsl.rst`).
2. A preamble – a rST field list containing metadata about the CEP, including the CEP number, the names of the various members of the *CEP team*, and so forth. See *CEP Metadata* below for specific details.
3. Abstract – a short (~200 word) description of the technical issue being addressed.
4. Specification – The technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.
5. Motivation – The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.
6. Rationale – The rationale fleshes out the specification by describing what motivated the design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

   The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.
7. Backwards Compatibility – All CEPs that introduce backwards incompatibilities must include a section describing these incompatibilities and their severity. The CEP must explain how the author proposes to deal with these incompatibilities. CEP submissions without a sufficient backwards compatibility treatise may be rejected outright.
8. Reference Implementation – The reference implementation must be completed before any CEP is given status "Final", but it need not be completed before the CEP is accepted. While there is merit to the approach of reaching consensus on the specification and rationale before writing code, the principle of "rough consensus and running code" is still useful when it comes to resolving many discussions of API details.

   The final implementation must include tests and documentation, per Celery's contribution guidelines.
9. Copyright/public domain – Each CEP must be explicitly licensed as CC0.

CEP Metadata

Each CEP must begin with some metadata given as an rST field list. The headers must contain the following fields:

CEP The CEP number. In an initial pull request, this can be left out or given as `XXXX`; the reviewer who merges the pull request will assign the CEP number.

Type `Feature`, `Informational`, or `Process`

Status `Draft`, `Accepted`, `Rejected`, `Withdrawn`, `Final`, or `Superseded`

Created Original creation date of the CEP (in `yyyy-mm-dd` format)

Last-Modified Date the CEP was last modified (in `yyyy-mm-dd` format)

Author The CEP's author(s).

Implementation-Team The person/people who have committed to implementing this CEP

Shepherd The core developer "on point" for the CEP

Requires If this CEP depends on another CEP being implemented first, this should be a link to the required CEP.

Celery-Version (optional) For Feature CEPs, the version of Celery (e.g. `5.0`) that this feature will be released in.

Replaces and Superseded-By (optional) These fields indicate that a CEP has been rendered obsolete. The newer CEP must have a `Replaces` header containing the number of the CEP that it rendered obsolete; the older CEP has a `Superseded-By` header pointing to the newer CEP.

Resolution (optional) For CEPs that have been decided upon, this can be a link to the final rationale for acceptance/rejection. It's also reasonable to simply update the CEP with a "Resolution" section, in which case this header can be left out.

Auxiliary Files

CEPs may include auxiliary files such as diagrams. Such files must be named `XXXX-descriptive-title.ext`, where "XXXX" is the CEP number, "descriptive-title" is a short slug indicating what the file contains, and "ext" is replaced by the actual file extension (e.g. "png").

Reporting CEP Bugs, or Submitting CEP Updates

How you report a bug, or submit a CEP update depends on several factors, such as the maturity of the CEP, the preferences of the CEP author, and the nature of your comments. For the early draft stages of the CEP, it's probably best to send your comments and changes directly to the CEP author. For more mature, or finished CEPs you can submit corrections as GitHub issues or pull requests against the CEP repository.

When in doubt about where to send your changes, please check first with the CEP author and/or a core developer.

CEP authors with git push privileges for the CEP repository can update the CEPs themselves.

Transferring CEP Ownership

It occasionally becomes necessary to transfer ownership of CEPs to a new author. In general, it is preferable to retain the original author as a co-author of the transferred CEP, but that's really up to the original author. A good reason to transfer ownership is because the original author no longer has the time or interest in updating it or following through with the CEP process, or has fallen off the face of the 'net (i.e. is unreachable or not responding to email). A bad reason to transfer ownership is because the new author doesn't agree with the direction of the CEP. One aim of the CEP process is to try to build consensus around a CEP, but if that's not possible, an author can always submit a competing CEP.

If you are interested in assuming ownership of a CEP, first try to contact the original author and ask for permission. If they approve, ask them to open a pull request transferring the CEP to you. If the original author doesn't respond to email within a few weeks, contact Celery-developers.

Differences between CEPs and PEPs

As stated in the preamble, the CEP process is more or less a direct copy of the PEP process (and this document is a modified version of PEP 1).

Relative to the PEP process, we made the following changes in CEPs:

- The workflow is GitHub based (rather than email-based as in PEP 1).

  This is a simple enough change, but has a number of ramifications for the details of how CEPs work, including:

  - CEPs use pull requests (and direct commits) as the workflow process.
  - CEPs use rST-style headers rather than RFC822 (because rST-style headers render properly on GitHub without additional tooling).
  - CEPs have document titles rather than title fields in the metadata (again, because of GitHub rendering).
  - CEP are organized into directories based on statuses (e.g. `draft/`, `accepted/`, `final/`, etc) so that additional tooling to create an index by status isn't needed.
  - CEP file names are more descriptive (e.g. `0181-canvas-dsl.rst`), again to avoid the need for additional tooling.
  - CEPs are "edited" (e.g. pull request approved) by any core developer, rather than an explicit "editor" role like the PEP editors.
- CEPs are pronounced upon by the Technical Board, rather than a BDFL (because Celery has no BDFLs).
- CEPs explicitly require identifying a few roles (Author, Implementation Team, and Shepherd) before submission and throughout the process. With PEPs, most are authored and implemented by the same person, but the same doesn't seem to be true of CEPs (so far), hence the "implementer" role. As for the "shepherd": the BDFL or BDFL-delegate tends to be much more hands-on than the Technical Board, so the role of commenting and critiquing will be fulfilled by the shepherd, rather than the board. Further, we've observed that features are tremendously unlikely to make it into Celery without a committer on board to do the detail-work of merging a patch.
- CEPs simplify the metadata somewhat, removing a few fields ("Post-History", etc.) and dropping a couple of statuses ("Active" gets merged into "Final", and "Deferred" merged into "Withdrawn").
- CEPs have "Feature CEPs" rather than "Standards Track" CEPs.
- CEPs may only be reStructuredText (there is no plain text option).

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

1.3 Approved CEPs

CEPs that have been approved by the Technical Board and are in the process of being implemented. See CEP 1 for details.

1.4 Draft CEPs

CEPs that are still in the process of being drafted and have not yet been approved or denied. See CEP 1 for details.

1.4.1 CEP XXXX: Celery 5 High Level Architecture

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Informational

Created 2019-04-08

Last-Modified 2019-04-08

Table of Contents

- *Abstract*
- *Specification*
  - *Message Types*
  - *Message Passing Protocol*
    - *Introduction to AMQP 1.0 Terminology*
      - *Connections*
      - *Sessions*
      - *Channels*
      - *Nodes (Components)*
      - *Links*
      - *Terminus*
    - *Implementation*

- *Canvas*
  - *Signatures*
  - *Primitives*
    - *Error Handling*
    - *Error Recovery*
    - *Chains*
    - *Groups*
    - *Chords*
    - *Maps*
    - *Starmaps*
    - *Chunks*
    - *Forks*
- *Workflows*
- *Observability*
  - *Metrics*
  - *Trace Points*
  - *Logging*
- *Network Resilience and Fault Tolerance*
  - *Fault Tolerance*
    - *Graceful Degradation*
    - *Retries*
    - *Health Checks*
    - *Circuit Breaking*
  - *Network Resilience*
- *Command Line Interface*
- *Dependency Inversion*
- *Worker*
  - *Configuration*
  - *Event Loop*
  - *Internal Tasks Queue*
  - *Internal Results Queue*
  - *Services*
  - *Internal Services*
    - *Task Execution*
    - *Consumer*

      - *Result Publisher*
      - *Maximal Concurrency Budget*
    - *Tasks*
      - *Deduplication*
      - *I/O Bound Tasks*
      - *CPU Bound Tasks*
      - *Boxed Tasks*
      - *Concurrency Budget*
    - *Internal Tasks*
      - *SystemD Notify*
      - *Retry Failed Boot Step*
    - *Boot Steps*
    - *Worker Health Checks*
    - *Worker Circuit Breakers*
    - *Inbox Queue*
  - *Publisher*
    - *Messages Backlog*
    - *Publisher Daemon*
    - *Publisher Internal Services*
      - *Message Publisher*
      - *Listener*
    - *Publisher Health Checks*
    - *Publisher Circuit Breakers*
  - *Router*
    - *Data Sources and Sinks*
      - *Data Sources*
      - *Data Sinks*
  - *Controller*
    - *Foreman*
      - *Development Mode*
      - *SystemD Integration*
      - *Other Integrations*
    - *Scheduler*
      - *Concurrency Limitations*
      - *Suspend/Resume Tasks*

        · *Task Prioritization*
        · *Resource Saturation*
        · *Rate Limiting*
        · *Periodic Tasks*
        · *Autoscaler*
      * *Controller Internal Services*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

Abstract

When Celery was conceived, production environments were radically different from today.

Nowadays most applications are (or should be):

- Deployed to a cloud provider's computing resources.
- Distributed (sometimes between data centers).
- Available or Consistent (We must pick one according to *CAP Theorem*).
- Network Partition Tolerant.
- Observable.
- Built with scalability in mind.
- Cloud Native - The application's lifecycle is managed using Kubernetes, Swarm or any other scheduler.

In addition, Celery lacks proper support for large scale deployments and some useful messaging architectural patterns.

Celery 5 is the next major version of Celery and so we are able to break backwards compatibility, even in major ways.

As such, our next major version should represent a paradigm shift in the way we implement our task execution platform.

Specification

Note: The code examples below are for illustration purposes only.

Unless explicitly specified, The API will be determined in other CEPs.

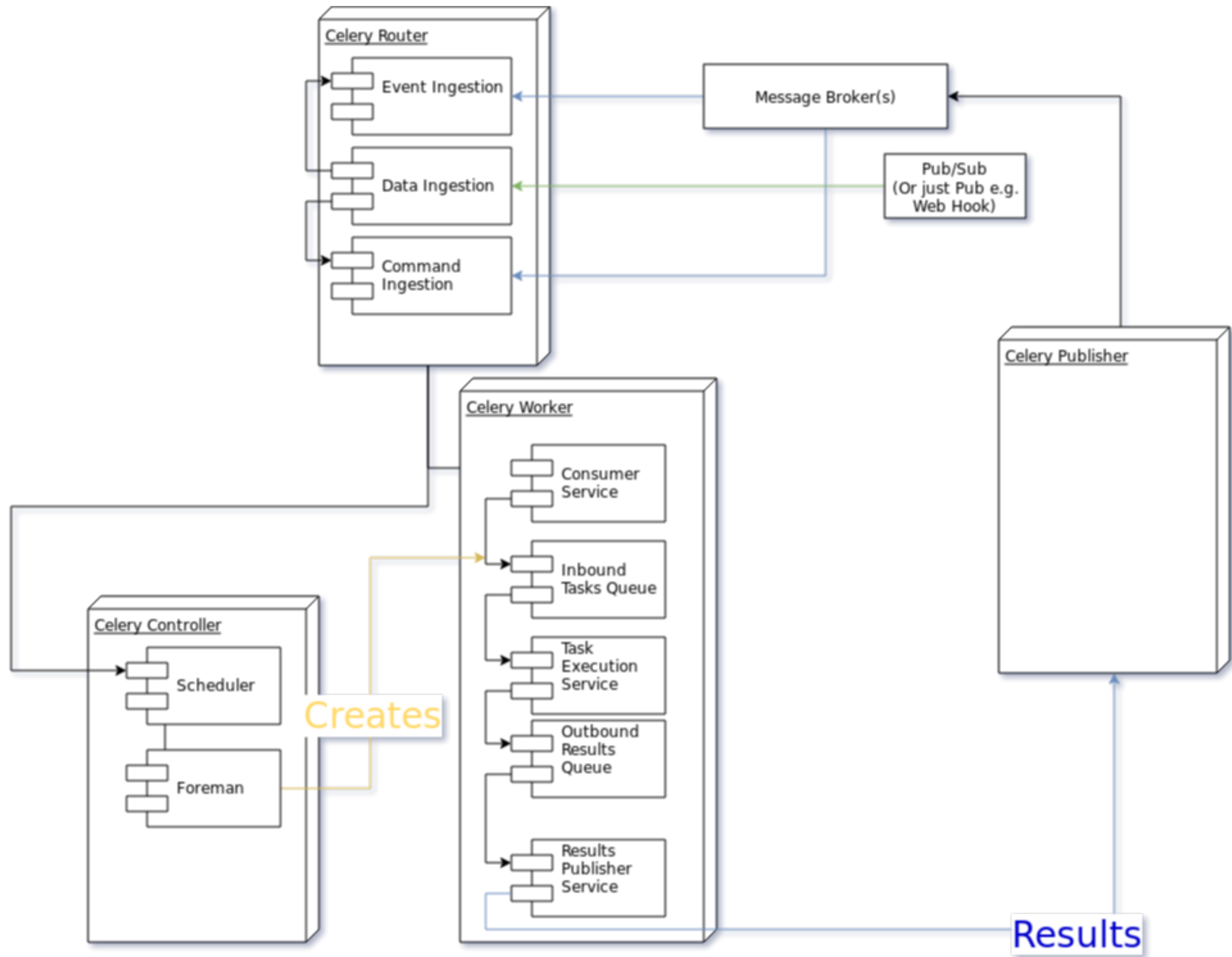


Fig. 1: High Level Architecture Diagram

Message Types

In relation to Celery *Command messages* are the messages we publish to the *Message Broker* whenever we want to execute a *Task*.

Document messages are the messages we get as a result.

Document messages may also be produced whenever we publish a serialized representation of a *Domain Model*.

```
>>> from celery import task
>>> @task
... def add(a, b):
...   return a + b
>>> result = add.delay(1, 2)  # Publish a command message
>>> result.get()  # Consume a Document message
3
```

Event messages are a new concept for Celery. They describe that a *Domain Event* occurred. Multiple tasks can be subscribed to an event.

```
>>> from uuid import UUID
>>> from celery import task, event
>>> from myapp.models import User, AccountManager
>>> @task
... def send_welcome_email(user_id, email):
...   send_email(email=email, contents="hello, welcome", subject="welcome") # Send a␣
↪welcome email to the user...
...   User.objects.filter(pk=user_id).update(welcome_email_sent=True)
>>> @task
... def notify_account_manager(user_id, email):
...   account_manager = AccountManager.objects.assign_account_manager(user_id)
...   send_email(email=account_manager.email, contents="you have a new user to attend to
↪", subject="Alert") # Send an email to the account manager...
>>> @event
... class UserRegistered:
...   user_id: UUID
...   email: str
>>> UserRegistered.subscribe(send_welcome_email)
>>> UserRegistered.subscribe(notify_account_manager)
>>> UserRegistered.delay(user_id=1, email='foo@bar.com')  # Calls both send_welcome_
↪email and notify_account_manager with the provided arguments.
```

These architectural building blocks will aid us in creating a better messaging system. To encourage *Ubiquitous Language*, we will be using them in this document and in Celery 5's codebase as well.

Message Passing Protocol

In previous versions of Celery the only protocol we defined was for the message format itself.

Whenever a client published a message to the broker using *task.delay()* Celery serialized it to a known format which it can use to invoke the task remotely.

This meant that you needed a special client library to publish tasks that Celery will later consume and execute. The architectural implication is a coupling between the publisher and the consumer which defeats the general use-case of message passing: decoupling the publisher from the consumer.

In addition, the communication protocol between the master process and the workers was an internal implementation detail which was not well defined or understood by the maintainers or users.

Introduction to AMQP 1.0 Terminology

AMQP 1.0 is a flexible messaging protocol that is designed to pass messages between nodes. Nodes can be queues, exchanges, Kafka-like log stream or any other entity which which publishes or consumes a message from another node.

AMQP 1.0, unlike AMQP 0.9.1, does not mandate the presence of a broker to facilitate message passing.

This feature allows us to use it to communicate effectively between the different Celery components without overloading the broker with connections and messages. It also allows users to invoke tasks directly without the usage of a *Message Broker*, thus treating a Worker as an actor.

Connections

Sessions

Channels

Nodes (Components)

Nodes (also called Components) are entities on either different processes which establish links.

Links

Terminus

Implementation

The python bindings to the qpid-proton library exposes all these features, can be used with trio and is supported on PyPy.

We will use it as a fundamental building block for our entire architecture.

Canvas

In Celery Canvas is the mechanism which users can use to define workflows dynamically.

In previous versions of Celery there are issues with the protocol which can cause Celery to generate messages too large for storage in *Message Brokers*. There are also issues with the API and multiple implementation problems.

In Celery 5 we're going to revamp the protocol, API and possibly the implementation itself to resolve these issues.

Signatures

Primitives

Error Handling

Error Recovery

Chains

Groups

Chords

Maps

Starmaps

Chunks

Forks

Workflows

A Workflow is a declarative *Canvas*.

Workflows provide an API for incrementally executing business logic by dividing it to small, self-contained tasks.

Unlike *Canvas*, a Workflow is immutable, static and predicable.

Observability

One of Celery 5's goals is to be *observable*.

Each Celery component will record statistics, provide trace points for application monitoring tools and distributed tracing tools and emit log messages when appropriate.

Metrics

Celery stores and publishes metrics which allows our users to debug their applications more easily and spot problems.

By default each worker will publish the metrics to a dedicated queue.

Other methods such as publishing them to StatsD is also possible using the provided extension point.

Trace Points

Celery provides trace points for application monitoring tools and distributed tracing tools.

This allows our users to spot and debug performance issues.

Logging

All log messages must be structured. *Structured logs* provide context for our users which allows them to debug problems more easily and aids the developers to resolve bugs in Celery.

The structure of a log message is determined whenever a component is initialized.

During initialization, an attempt will be made to detect how the component lifecycle is managed. If all attempts are unsuccessful, the logs will be formatted using *JSON* and will be printed to stdout.

Celery will provide an extension point for detection of different runtimes.

Example

If a component's lifecycle is managed by a SystemD service, Celery will detect that the JOURNAL_STREAM environment variable is set when the process starts and use it's value to transmit structured data into journald.

Whenever Celery fails to log a message for any reason it publishes a command to the worker's *Inbox Queue* in order to log the message again. As usual messages which fail to be published are stored in the *Messages Backlog*.

In past versions of Celery we've used the standard logging module. Unfortunately it does not meet the aforementioned requirements.

Eliot is a logging library which provides structure and context to logs, even across coroutines, threads and processes.

It is also able to emit logs to journald and has native trio integration.

Network Resilience and Fault Tolerance

Celery 5 aims to be network failure resilient and fault tolerant. As an architectural guideline Celery must retry operations **by default** and must avoid doing so **indefinitely and without proper limits**.

Any operation which cannot be executed either momentarily or permanently as a result of a bug must not be retried beyond the configured limits. Instead, Celery must store the operation for further inspection and if required, manual intervention.

Celery must track and automatically handle "poisonous messages" to ensure the recovery of the Celery cluster.

Fault Tolerance

Distributed Systems suffer from an inherent property:

> Any distributed system is unreliable.
>
> - The network may be unavailable or slow.
> - Some or all of the servers might suffer from a hardware failure.
> - A node in the system may arbitrarily crash due to lack of memory or a bug.
> - Any number of unaccounted failure modes.

Therefore, Celery must be fault tolerant and gracefully degrade its' operation when failures occur.

Graceful Degradation

Features which are less mission-critical may fail at any time, provided that a warning is logged.

This document will highlight such features and describe what happens when they fail for any reason.

Retries

In previous Celery versions tasks were not retried by default.

This forces new adopters to carefully read our documentation to ensure the fault tolerance of their tasks.

In addition, our retry policy was declared at the task level. When using Automatic retry for known exceptions Celery automatically retries tasks when specific exceptions are raised.

However the same type of exception may hold a different meaning in different contexts.

This created the following pattern:

```python
from celery import task
from data_validation_lib import validate_data

def _calculate(a, b):
  # Do something

@task(autoretry_for=(ValueError,))
def complex_calculation(a, b):
  try:
    # Code that you don't control can raise a ValueError.
    validate_data(a, b)
  except ValueError:
    print("Complete failure!")
    return

  # May temporarily raise a ValueError due to some externally fetched
  # data which is currently incorrect but will be updated later.
  _calculate()
```

An obvious way around this problem is to ensure that *_calculate()* raises a custom exception.

But we shouldn't force the users to use workarounds. Our code should be ergonomic and idiomatic.

Instead, we should allow users to declare sections as “poisonous” - tasks that if retried will surely fail if they fail at those sections.

```
from celery import task, poisonous
from data_validation_lib import validate_data

def _calculate(a, b):
  # Do something

@task(autoretry_for=(ValueError,))
def complex_calculation(a, b):
  with poisonous():
    validate_data(a, b)

  # May temporarily raise a ValueError due to some externally fetched
  # data which is currently incorrect but will be updated later.
  _calculate()
```

Not all operations are equal. Some may be retried more than others. Some may need to be retried less often.

Currently there are multiple ways to achieve this:

You can separate them to different tasks with a different retry policy:

```
from celery import task

@task(retry_policy={
  'max_retries': 3,
  'interval_start': 0,
  'interval_step': 0.2,
  'interval_max': 0.2
})
def foo():
  second_operation()

@task(retry_policy={
  'max_retries': 10,
  'interval_start': 0,
  'interval_step': 5,
  'interval_max': 120
})
def bar():
  first_operation()
  foo.delay()
```

Or you can wrap each code section in a try..except clause and call `celery.app.task.Task.retry()`.

```
@task(bind=True)
def foo(self):
  try:
    # first operation
  except Exception:
    self.retry(retry_policy={
      'max_retries': 10,
      'interval_start': 0,
```

(continues on next page)

(continued from previous page)

```
      'interval_step': 5,
      'interval_max': 120
    })

  try:
    first_operation()
  except Exception:
    self.retry(retry_policy={
      'max_retries': 10,
      'interval_start': 0,
      'interval_step': 5,
      'interval_max': 120
    })

  try:
    second_operation()
  except Exception:
    self.retry(retry_policy={
      'max_retries': 3,
      'interval_start': 0,
      'interval_step': 0.2,
      'interval_max': 1
    })
```

Those solutions are unnecessarily verbose. Instead, we could use a with clause if all we want to do is retry.

```
@task
def foo():
  with retry(max_retries=10, interval_start=0, interval_step=5, interval_max=120):
    first_operation()

  with retry(max_retries=3, interval_start=0, interval_step=0.2, interval_max=1):
    second_operation()
```

By default messages which cannot be re-published will be stored in the *Messages Backlog*.

Implementers may provide other fallbacks such as executing the retried task in the same worker or abandoning the task entirely.

Some operations are not important enough to be retried if they fail.

Example

We're implementing a BI system that records mouse interactions.

The BI team has specified that it wants to store the raw data and the time span between interactions. Since we have a lot of data already, if the system failed to insert the raw data into the data store then we should not fail. Instead, we should emit a warning. However, the time span between mouse interactions is critical to the BI team's insight and if that fails to be inserted into the data store we must retry it.

Such a task can be defined using the `optional` context manager.

```
@task
def foo(raw_data):
  # Using default retry policy
  with optional():
    # ignore retry policy and proceed
    insert_raw_data(raw_data)

  with retry(max_retries=10, interval_start=0, interval_step=5, interval_max=120):
    calculation = time_span_calculation(raw_data)
    insert_time_spans(calculation)
```

In case of a failure inside the optional context manager, a warning is logged.

We can of course be more specific about the failures we allow:

```
@task
def foo(raw_data):
  # Using default retry policy
  with optional(ConnectionError, TimeoutError):
    # ignore retry policy and proceed
    insert_raw_data(raw_data)

  with retry(max_retries=10, interval_start=0, interval_step=5, interval_max=120):
    calculation = time_span_calculation(raw_data)
    insert_time_spans(calculation)
```

Health Checks

Health Checks are used in Celery to verify that a worker is able to successfully execute a *task* or a *service*.

The *Scheduler* is responsible for scheduling the health checks for execution in each worker after each time the configured period of time lapses.

Whenever a health check should be executed the *Scheduler* instructs the *Publisher* to send the *<health check name>_expired* *Event Message* to each worker's *Inbox Queue*.

Workers which have tasks subscribed to the event will execute all the subscribed tasks in order to determine the state of the health check.

Health Checks can handle *Document Messages* as input from *Data Sources*.

This is useful when you want to respond to an alert from a monitoring system or when you want to verify that all incoming data from said source is valid at all times before executing the task.

In addition to tasks, Health Checks can also use *Services* in order to track changes in the environment it is running on.

Example

We have a task which requires 8GB of memory to complete. The worker runs a service which constantly monitors the system's available memory. If there is not enough memory it changes the task's health check to the **Unhealthy** state.

If a task or a service that is part of a health check fails unexpectedly it is ignored and an error message is logged.

Celery provides many types of health checks in order to verify that it can operate without any issues.

Users may implement their own health checks in addition to the built-in health checks.

Some health checks are specific to the worker they are executing on. Therefore, their state is stored in-memory in the worker.

Other health checks are global to all or a group of workers. As such, their state is stored externally.

If the state storage for health checks is not provided, these health checks are disabled.

Health Checks can be associated with tasks in order to ensure that they are likely to succeed. Multiple Health Check failures may trigger a *Circuit Breaker* which will prevent the task from running for a period of time or automatically mark it as failed.

Each Health Check declares its possible states. Sometimes it makes sense to try to execute a task anyway even if the health check occasionally fails.

Example

A health check that verifies whether we can send a HTTP request to an endpoint has multiple states.

The health check performs an OPTIONS HTTP request to that endpoint and expects it to respond within the specified timeout.

The health check is in a **Healthy** state if all the following conditions are met:

- The DNS server is responding within the specified time limit and is resolving the address correctly.
- The TLS certificates are valid and the connection is secure.
- The Intrusion Detection System reports that the network is secure.
- The HTTP method we're about to use is listed in the OPTIONS response's ALLOW header.
- The content type we're about to format the request in is listed in the OPTIONS response's ACCEPT header.
- The OPTIONS request responds within the specified time limits.
- The OPTIONS request responds with 200 OK status.

In addition, the actual request performed in the task must also stand in the aforementioned conditions. Otherwise, the health check will change it's state.

The health check can be in an **Insecure** state if one or more of the following conditions are met:

- The TLS certificates are invalid for any reason.
- The Intrusion Detection System has reported that the network is compromised for any reason.

It is up for the user to configure the *Circuit Breaker* to prevent insecure requests from being executed.

The health check can be in an **Degraded** state if one or more of the following conditions are met:

- The request does not reply with a 2xx HTTP status.
- The request responds slowly and almost reaches it's time limits.

It is up for the user to configure the *Circuit Breaker* to prevent requests from being executed after multiple attempts or not all.

The health check can be in an **Unhealthy** state if one or more of the following conditions are met:

- The request responds with a 500 HTTP status.
- The request's response has not been received within the specified time limits.

It is up for the user to configure the *Circuit Breaker* to prevent requests from being executed if there is an issue with the endpoint.

The health check can be in an **Permanently Unavailable** state if one or more of the following conditions are met:

- The request responds with a 404 Not Found HTTP status.
- The HTTP method we're about to use is not allowed.
- The content type we're about to use is not allowed.

Circuit Breaking

Celery 5 introduces the concept of *Circuit Breaker* into the framework.

A Circuit Breaker prevents a *task* or a *service* from executing.

Each task or a service has a Circuit Breaker which the user can associate health checks with.

In addition, if the task or the service unexpectedly fails, the user can configure the Circuit Breaker to trip after a configured number of times. The default value is 3 times.

Whenever a Circuit Breaker trips, the worker will emit a warning log message.

After a configured period of time the circuit is opened again and tasks may execute. The default period of time is 30 seconds with no linear or exponential growth.

The user will configure the following properties of the Circuit Breaker:

- How many times the health checks may fail before the circuit breaker trips.
- How many unexpected failures the task or service tolerates before tripping the Circuit Breaker.
- The period of time after which the circuit is yet again closed. That time period may grow linearly or exponentially.
- How many circuit breaker trips during a period of time should cause the worker to produce an error log message instead of a warning log message.
- The period of time after which the circuit breaker downgrades it's log level back to warning.

Example

We allow 2 **Unhealthy** health checks and/or 10 **Degraded** health checks in a period of 10 seconds.

If we cross that threshold, the circuit breaker trips.

The circuit will be closed again after 30 seconds. Afterwards, the task can be executed again.

If 3 consequent circuit breaker trips occurred during a period of 5 minutes, all circuit breaker trips will emit an error log message instead of a warning.

The circuit breaker will downgrade it's log level after 30 minutes.

Network Resilience

Network Connections may fail at any time. In order to be network resilient we must use retries and circuit breakers on all outgoing and incoming network connections.

In addition, proper timeouts must be set to avoid hanging when the connection is slow or unresponsive.

Each network connection must be accompanied by a *health check*.

Health check failures must eventually trip a *circuit breaker*.

Command Line Interface

Our command line interface is the user interface to all of Celery's functionality. It is crucial for us to provide an excellent user experience.

Currently Celery uses `argparse` with a few custom hacks and workarounds for things which are not possible to do with `argparse`. This created some bugs in the past.

Celery 5 will use Click, a modern Python library for creating command line programs.

Click's documentation explains why it is a good fit for us:

> There are so many libraries out there for writing command line utilities; why does Click exist?
>
> This question is easy to answer: because there is not a single command line utility for Python out there which ticks the following boxes:
>
> - is lazily composable without restrictions
> - supports implementation of Unix/POSIX command line conventions
> - supports loading values from environment variables out of the box
> - supports for prompting of custom values
> - is fully nestable and composable
> - works the same in Python 2 and 3
> - supports file handling out of the box
> - comes with useful common helpers (getting terminal dimensions, ANSI colors, fetching direct keyboard input, screen clearing, finding config paths, launching apps and editors, etc.)
>
> There are many alternatives to Click and you can have a look at them if you enjoy them better. The obvious ones are `optparse` and `argparse` from the standard library.
>
> Click actually implements its own parsing of arguments and does not use `optparse` or `argparse` following the `optparse` parsing behavior. The reason it's not based on `argparse` is that `argparse` does not allow proper nesting of commands by design and has some deficiencies when it comes to POSIX compliant argument handling.
>
> Click is designed to be fun to work with and at the same time not stand in your way. It's not overly flexible either. Currently, for instance, it does not allow you to customize the help pages too much. This is intentional because Click is designed to allow you to nest command line utilities. The idea is that you can have a system that works together with another system by tacking two Click instances together and they will continue working as they should.
>
> Too much customizability would break this promise.

Click describes it's advantages over argparse in its documentation as well:

> Click is internally based on optparse instead of argparse. This however is an implementation detail that a user does not have to be concerned with. The reason however Click is not using argparse is that it has some problematic behaviors that make handling arbitrary command line interfaces hard:
>
> - argparse has built-in magic behavior to guess if something is an argument or an option. This becomes a problem when dealing with incomplete command lines as it's not possible to know without having a full understanding of the command line how the parser is going to behave. This goes against Click's ambitions of dispatching to subparsers.
> - argparse currently does not support disabling of interspersed arguments. Without this feature it's not possible to safely implement Click's nested parsing nature.

In contrast to `argparse`, the Click community provides many extensions we can use to create a better user experience for our users.

Click supports calling *async* methods and functions using the *asyncclick* <https://github.com/python-trio/asyncclick>`_ fork which is likely to be important for us in the future.

Dependency Inversion

Currently Celery uses different singleton registries to customize the behavior of its' different components. This is known as the *Service Locator* pattern.

Mark Seemann criticized Service Locators as an anti-pattern for multiple reasons:

- It has API usage problems and maintenance issues.
- It violates encapsulation.
- It violates SOLID.

Using constructor injection is a much better way to invert our dependencies.

For that purpose we have selected the dependencies library.

Worker

The Worker is the most fundamental architectural component in Celery.

The role of the Worker is to be a *Service Activator*. It executes *Tasks* in response to *messages*.

A Worker is also an *Idempotent Receiver*. If the exact same *Message* is received more than once, the duplicated messages are discarded. In this case, a warning log message is emitted. The Worker maintains a list of identifiers of recently received *messages*. The number of *messages* is determined by a configuration value. By default that value is 100 *messages*.

Configuration

In previous versions of Celery we had the option to load the configuration from a Python module.

Cloud Native applications often use Etcd, Consul or Kubernetes Config Maps (among others) to store configuration and adjust it when needed.

Celery 5 introduces the concept of configuration backends. These backends allow you to load the Worker's configuration from any source.

The default configuration backend loads the configuration from a Python module.

Users may create their own configuration backends to load configuration from a YAML file, a TOML file or a database.

Once the configuration has changed, the Worker stops consuming tasks, waits for all other tasks to finish and reloads the configuration.

This behavior can be disabled using a CLI option.

Event Loop

In Celery 4 we have implemented our own custom Event Loop. It is a cause for many bugs and issues in Celery.

In addition, some I/O operations are still blocking the event loop since the clients we use do not allow non-blocking operations.

The most important feature of Celery 5 is to replace the custom Event Loop with Trio.

We selected it because of its design, interoperability with asyncio and its many features.

Trio provides a context manager which limits the concurrency of coroutines and/or threads. This saves us from further bookkeeping when a Worker executes *Tasks*.

Trio allows coroutines to report their status. This is especially useful when we want to block the execution of other coroutines until initialization of the coroutine completes. We require this feature for implementing *Boot Steps*.

Trio also provides a feature called cancellation scopes which allows us to cancel a coroutine or multiple coroutines at once. This allows us to abort *Tasks* and handle the aborted tasks in an idiomatic fashion.

All of those features save us from writing a lot of code. If we were to select asyncio as our Event Loop, we'd have to implement most of those features ourselves.

Internal Tasks Queue

The internal tasks queue is an in-memory queue which the worker uses to queue tasks for execution.

Each task type has its own queue.

The queue must be thread-safe and coroutine-safe.

Internal Results Queue

The internal results queue is an in-memory queue which the worker uses to report the result of tasks back to the *Router*.

The queue must be thread-safe and coroutine-safe.

Services

Services are stateful, long running tasks which are used by Celery to perform its internal operations.

Some services publish *messages* to brokers, others consume *messages* from them. Other services are used to calculate optimal scheduling of tasks, routing, logging and even executing tasks.

Users may create their own services as well.

Internal Services

The Worker defines internal services to ensure it's operation and to provide support for it's features.

The exact API for each service will be determined in another CEP.

This list of internal services is not final. Other internal services may be defined in other CEPs.

Task Execution

The `Task Execution` service is responsible for executing all Celery *tasks*.

It consumes tasks from the *Internal Tasks Queue*, executes them and enqueues the results into the *Internal Results Queue*.

The service supervises how many tasks are run concurrently and limits the number of concurrent tasks to the configured amount in accordance to the *Concurrency Budget*.

The service also attempts to saturate all of the available resources by scheduling as many as *I/O Bound Tasks* and *CPU Bound Tasks* as possible.

Consumer

The `Consumer` service consumes *messages* from one or many *Routers* or *Message Brokers*.

The service enqueues the consumed *messages* into the appropriate *Internal Tasks Queue* according to the task's type.

Result Publisher

The `Result Publisher` service consumes results from the *Internal Results Queue* and publishes them to the *Router*'s *Inbox Queue*.

Maximal Concurrency Budget

The `Maximal Concurrency Budget` service runs the user's concurrency budget strategies and notifies the *tasks* service of changes in concurrency.

Tasks

Tasks are short running, have a defined purpose and are triggered in response to messages.

Celery declares some tasks for internal usage.

Users will create their own tasks for their own use.

Deduplication

Some Tasks are not idempotent and may not run more than once.

Users may define a deduplication policy to help Celery discard duplicated messages.

Example

The `send_welcome_email` task is only allowed to send one welcome email per user.

The user defines a deduplication policy which checks with their 3rd party email delivery provider if that email has already been sent. If it did, the user instructs Celery to reject the task.

I/O Bound Tasks

I/O bound tasks are tasks which mainly perform a network operation or a disk operation.

I/O bound tasks are specifically marked as such using Python's *async def* notation for defining awaitable functions. They will run in a Python coroutine.

Due to that, any I/O operation in that task must be asynchronous in order to avoid blocking the event loop.

Some of the user's asynchronous tasks won't use trio as their event loop but will use the more commonly used asyncio event loop which we do support.

In that case, the user must specify the event loop they are going to use for the task.

CPU Bound Tasks

CPU bound tasks are tasks which mainly perform a calculation of some sort such as calculating an average, hashing, serialization or deserialization, compression or decompression, encryption or decryption etc. In some cases where no asynchronous code for the I/O operation is available CPU bound tasks are also an appropriate choice as they will not block the event loop for the duration of the task.

Performing operations which release the *GIL* is recommended to avoid throttling the concurrency of the worker.

CPU bound tasks are specifically marked as such using Python's *def* notation for defining functions. They will run in a Python thread.

Using threads instead of forking the main process has its upsides:

- It simplifies the Worker's architecture and makes it less brittle.

  Processes require *IPC* to communicate with each other. This complicates implementation since multiple methods are required to support *IPC* reliably across all operating systems Celery supports. Threads on the other hand require less complicated means of communication.

  In trio, we simply use a memory channel which is a coroutine and thread safe way to send and receive values.

- PyPy's JIT warms up faster.

  When using PyPy, using threads means that we get to keep our previous JIT traces and therefore JIT warmup will occur faster.

  If we'd use processes, each process has to warm up its own JIT which results in tasks being executed slower for a longer period of time.

Using threads for CPU bound tasks unfortunately has some downsides as well:

- Pure Python CPU bound workloads cannot be executed in parallel.

  In both CPython and PyPy the *GIL* prevents executing two Python bytecodes in parallel by design.

  This results in slower execution of Python code when using threads.

- The *GIL*'s implementation in CPython 3.x has a defect in design.

  According to a bug report the new GIL in Python 3 CPU bound threads may starve I/O threads (in our case the main thread).

  Note: This is not an issue with PyPy's implementation of the *GIL* according to Armin Rigo, PyPy's creator.

- Tasks are no longer isolated.

  Since we're mixing workloads to maximize our throughput a task which crashes the worker or leaks memory can crash the entire worker.

To mitigate these issues CPU Bound Tasks may be globally rate limited to allow the main thread to complete executing *I/O Bound Tasks*.

Boxed Tasks

To minimize the disadvantages of using threads in Python and workaround the shortcomings of the *GIL*, Celery also provides a new type of tasks called Boxed Tasks.

Boxed Tasks are processes which execute tasks in an isolated manner.

The processes' lifecycle is managed by the *Controller*.

Since Boxed Tasks are run separately from Celery itself, the program the process is running can be written in any language as long as it implements IPC in the same way the *Controller* expects.

Boxed tasks are a special kind of *I/O Bound Tasks*. They are executed the same way inside the worker but defined using a different API.

Concurrency Budget

Each worker has a concurrency budget for each type of task it can run.

The budget for each type of task is defined by a minimal and an optional maximal concurrency.

The maximal concurrency budget can be dynamic or fixed. Dynamic maximal concurrency strategies may be used to determine the maximum concurrency based on the load factor of the server, available network bandwidth or any other requirement the user may have.

Note:

If a user specifies a concurrency of more than 10 for *CPU Bound Tasks* a warning log message is emitted.

Too many threads can cause task execution to grind down to a halt.

If there are more tasks in the *Internal Tasks Queue* than what is currently the allowed maximum task concurrency we increase the current maximum by that number of tasks. After this increase, there will be a configurable cooldown period during which the worker will execute the new tasks. After the cooldown period, if there are still more tasks in the *Internal Tasks Queue* than the current maximum capacity we increase the maximum concurrency exponentially by a configurable exponent multiplied by the number of budget increases. The result is rounded up.

This process goes on until we either reach the maximum concurrency budget for that type of tasks or if the number of tasks in *Internal Tasks Queue* is lower than the current maximum concurrency.

If the current number of tasks is lower than the current maximal concurrency we decrease it to the number of tasks that are currently executing.

This algorithm can be replaced or customized by the user.

Internal Tasks

Celery defines internal tasks to ensure it's operation and to provide support for it's features.

The exact API for each task will be determined in another CEP.

This list of internal tasks is not final. Other internal tasks may be defined in other CEPs.

SystemD Notify

This task reports the status of the worker to the SystemD service which is running it.

It uses the sd_notify protocol to do so.

Retry Failed Boot Step

This task responds to a *Command Message* which instructs the worker to retry an optional *Boot Step* which has failed during the worker's initialization procedure.

The Boot Step's execution will be retried a configured amount of times before giving up.

By default this task's *Circuit Breaker* is configured to never prevent or automatically fail the execution of this task.

Boot Steps

During the Worker's initialization procedure Boot Steps are executed to prepare it for execution of tasks.

Some Boot Steps are responsible for starting all the *services* required for the worker to function correctly. Others may publish a *task* for execution to the worker's *Inbox Queue*.

Some Boot Steps are mandatory and thus if they fail, the worker refuses to start. Others are optional and their execution will be deferred to the *Retry Failed Boot Step* task.

Users may create and use their own Boot Steps if they wish to do so.

Worker Health Checks

Worker Circuit Breakers

Inbox Queue

Each worker declares an inbox queue in the *Message Broker*.

Publishers may publish *messages* to that queue in order to execute tasks on a specific worker.

Celery uses the Inbox Queue to schedule the execution of the worker's internal tasks.

Messages published to the inbox queue must be cryptographically signed.

This requirement can be disabled using a CLI option. Whenever the user uses this CLI option a warning log message is emitted.

While disabling the inbox queue is possible either through a configuration setting or a CLI option, some functionality will be lost. Whenever the user opts to disable the Inbox Queue a warning log message is emitted.

Publisher

The Publisher is responsible for publishing *messages* to a *Message Broker*.

It is responsible for publishing the *Message* to the appropriate broker cluster according to the configuration provided to the publisher.

The publisher must be able to run in-process inside a long-running thread or a long running co-routine.

It can also be run using a separate daemon which can serve all the processes publishing to the message brokers.

Messages Backlog

The messages backlog is a temporary queue of *messages* yet to be published to the appropriate broker cluster.

In the event where *messages* cannot be published for any reason, the *messages* are kept inside the queue.

By default, an in-memory queue will be used. The user may provide another implementation which stores the *messages* on-disk or in a central database.

Implementers should take into account what happens whenever writing to the messages backlog fails.

The default fallback mechanism will append the *messages* into an in-memory queue. These *messages* will be published first in order to avoid *Message* loss in case the publisher goes down for any reason.

Publisher Daemon

In sufficiently large deployments, one server runs multiple workloads which may publish to a *Message Broker*.

Therefore, it is unnecessary to maintain a publisher for each process that publishes to a *Message Broker*.

In such cases, a Publisher Daemon can be used. The publishing processes will specify it as their target and communicate the *messages* to be published via a socket.

Publisher Internal Services

The Publisher defines internal services to ensure it's operation and to provide support for it's features.

The exact API for each service will be determined in another CEP.

This list of internal services is not final. Other internal services may be defined in other CEPs.

Message Publisher

The `Message Publisher` service is responsible for publishing *messages* to a single *Message Broker*.

This service is run for each *Message Broker* the user configured the Publisher to publish messages to.

During the service's initialization it initializes a *Messages Backlog*. This will be the backlog the service consumes *messages* from.

The service maintains a connection pool to the *Message Broker* and is responsible for scaling the pool according to the pressure on the broker.

The connection pool's limits are configurable by the user. By default, we only maintain one connection to the *Message Broker*.

Listener

The `Listener` service is responsible for receiving messages and enqueuing them in the appropriate *Messages Backlog*.

During initialization the service starts listening to incoming TCP connections.

The service is only run in case the user opts to run the Publisher in *Publisher Daemon* mode.

Publisher Health Checks

The Publisher will perform health checks to ensure that the *Message Broker* the user is publishing to is available.

If a health check fails a configured number of times, the relevant *Circuit Breaker* is tripped.

Each *Message Broker* Celery supports must provide an implementation for the default health checks the Publisher will use for verifying its availability for new *messages*.

Further health checks can be defined by the user. These health checks allows the user to avoid publishing tasks if for example a 3rd party API endpoint is not available or slow, if the database the user stores the results in is available or any other check for that matter.

Publisher Circuit Breakers

Each *health check* has it's own Circuit Breaker. Once a circuit breaker is tripped, the *messages* are stored in the *Messages Backlog* until the health check recovers and the circuit is once again closed.

Router

The Router is a *Message Dispatcher*. It is responsible for managing the connection to a *Message Broker* and consuming *messages* from the *Message Broker*.

The Router can maintain a connection to a cluster of *message brokers* or even clusters of *message brokers*.

Data Sources and Sinks

Data Sources are a new concept in Celery. Data Sinks are a concept which replaces Result Backends.

Data Sinks consume *Document Messages* while Data Sources produce them.

Data Sources

Data Sources are *task* which either listen or poll for incoming data from a data source such as a database, a file system or an HTTP(S) endpoint.

These services produce *Document Messages*.

Tasks which are subscribed to Data Sources will receive the raw document messages for further processing.

Example

We'd like to design a feature which locks Github issues immediately after they are closed.

Github uses Webhooks to notify us when an issue is closed.

We set up a Data Source which starts an HTTPS server and expects incoming HTTP requests on an endpoint.

Whenever a request arrives a *Document Message* is published.

Data Sinks

A result from a *task* produces a *Document Message* which a Data Sink or multiple Data Sinks consume.

These *Document Messages* are then stored in the Sinks the task is registered to.

Example

We have a task which calculates the hourly average impressions of a user's post over a period of time.

The BI team requires the data to be inserted to BigQuery because it uses it to research the effectiveness of users posts.

However, the user-facing post analytics dashboard also requires this data and the team that maintains it doesn't want to use BigQuery because it is not a cost-effective solution and because they already use MongoDB to store all user-facing analytics data.

To resolve the issue we declare that the task routes it's results to two data sinks. One for the BI team and the other for the analytics team.

Each data sink is configured to insert the data to a specific table or collection.

Controller

The Controller is responsible for managing the lifecycle of all other Celery components.

Celery 5 is a more complex system with multiple components and will often be deployed in high throughput, highly available production systems.

The introduction of multiple components require us to have another component that manages the entire Celery cluster.

During the lifecycle of a worker the Controller also manages and optimizes the execution of tasks to ensure we maximize the utilization of all our resources and to prevent expected errors.

Note: The Controller is meant to be run as a user service. If the Controller is run with root privileges, a log message with the warning level will be emitted.

Foreman

The Foreman service is responsible for spawning the *Workers*, *Routers* and *Schedulers*.

By default, the Foreman service creates sub-processes for all the required components. This is suitable for small scale deployments.

Development Mode

During development, if explicitly specified, the Foremen will start all of Celery's services in the same process.

Since some of the new features in Celery require cryptographically signed messages Celery will generate self-signed certificates using the trustme library unless certificates are already provided or the user has chosen to disable this behavior through a CLI option.

SystemD Integration

Unless it is explicitly overridden by the configuration, whenever the Controller is run as a SystemD service, it will use SystemD to spawn all other Celery components.

Celery will provide the required services for such a deployment.

The Controller will use the sd_notify protocol to announce when the cluster is fully operational.

The user must configure the list of hosts the controller will manage and ensure SSH communication between the Controller's host and the other hosts is possible.

Other Integrations

Celery may be run in Kubernetes, Swarm, Mesos, Nomad or any other container scheduler.

Users may provide their own integrations with the Foreman which allows them to create and manage the different Celery components in a way that is native to the container scheduler.

The Controller may also manage the lifecycle of the *Message Broker* if the user wishes to do so.

Such an integration may be provided by the user as well.

Scheduler

The scheduler is responsible for managing the scheduling of tasks for execution on a cluster of workers.

The scheduler calculates the amount of tasks to be executed in any given time in order to make cluster wide decisions when autoscaling workers or increasing concurrency for an existing worker.

The scheduler is aware when tasks should no longer be executed due to manual intervention or a circuit breaker trip. To do so, it commands the router to avoid consuming the task or rejecting it.

Concurrency Limitations

Not all *Tasks* are born equal. Some tasks require more resources than others, some may only be executed once at a time due to a business requirement, other tasks may be executed only once per user at a time to avoid data corruption. At times, some tasks should not be executed at all.

The Scheduler is responsible for limiting the concurrency of such tasks.

A task's concurrency may be limited per worker or globally across all workers depending on the requirements. In case there are tasks which are limited globally, an external data store is required.

If a *task* is rate limited any concurrency limitations are ignored.

There are multiple types of limits the user can impose on a task's concurrency:

- **Fixed Limit**: A task can only be run at a maximum concurrency of a fixed number. This strategy is used when there is a predetermined limit on the number of concurrent tasks of the same type either because of lack of computing resources or due to business requirements.
- **Range**: A task can only be run at a maximum concurrency of a calculated limit between a range of numbers. This strategy is used to calculate the appropriate concurrency for a task based on some external resource such as the number of available database connections or currently available network bandwidth.
- **Concurrency Token**: A task can only be run at a maximum concurrency of either a **Fixed Limit** or a **Range** if it has the same Concurrency Token. A Concurrency Token is an identifier constructed from the task's *Message* by which we group a number of tasks for the purpose of limiting their concurrency. This strategy is used when the user would like to run one concurrent task per user or when a task may connect to multiple database instances in the cluster and the user wishes to limit the concurrency of the task per the available database connections in the selected instance.

A concurrency limitation of 0 implies that the task will be rejected and the queue it is on will not be consumed if possible.

The Scheduler may impose a concurrency limit if it deems fit at any time, these limits take precedence over any user imposed limit.

Suspend/Resume Tasks

Whenever a *Circuit Breaker* trips, the *Router* must issue an event to the Scheduler. The exact payload of the suspension event will be determined in another CEP.

This will notify the Scheduler that it no longer has to take this task into account when calculating the Celery workers cluster capacity. In addition this will set the task's *concurrency limitation* to 0.

The user may elect to send this event directly to the Scheduler if suspension of execution is required (E.g. The task interacts with a database which is going under expected maintenance).

Once scheduling can be resumed, the Scheduler sends another event to the *Router*. The exact payload of the resumption event will be determined in another CEP.

Task Prioritization

The Scheduler may instruct workers to prioritize tasks and to prefer consuming from specific queues first.

Priority based queues are only a partial solution to prioritizing tasks. Some *Message Brokers* don't support it. Those who do support priority based queues do not prioritize messages between queues.

This feature can be used to prefer to execute tasks which can be quickly executed first or to execute tasks which take a long time to complete first or to execute tasks which are rarely seen first.

Users may supply their own strategies for prioritizing tasks.

Resource Saturation

Celery provides the Resource Saturation *Task Prioritization* strategy to ensure we can utilize the full capacity of all the workers in the cluster.

The scheduler instructs each worker to prefer executing *I/O Bound Tasks* if the capacity of the worker for executing *CPU Bound Tasks* is nearing its maximum and vice versa.

Rate Limiting

A user may impose a rate limit on the execution of a *task*.

For example, we only want to run 200 *send_welcome_email()* *Tasks* per minute in order to avoid decreasing our email reputation.

Tasks may define a global rate limit or a per worker rate limit.

Whenever a *task* reaches it's rate limit, an event is published to the *Router*'s *Inbox Queue*. The event notifies the Router that it should not consume these tasks if possible. The exact payload of the rate limiting event will be determined in another CEP.

In addition the task is *suspended* until the rate limiting period is over.

Periodic Tasks

Previously, Celery used it's in-house periodic tasks scheduler which was the source of many bugs.

In Celery 5 we will use the APScheduler.

APScheduler has proved itself in production, is flexible and customizable and will provide trio support in 4.0, it's next major version.

In addition, APScheduler 4.0 will be highly available, a highly demanded feature from our users. This means that two Controller instances may exist simultaneously without duplicated *Tasks* being scheduled for execution.

The Scheduler only uses APScheduler to publish *Tasks* at the appropriate time according to the schedule provided by the user. Periodic tasks do not run inside the Scheduler.

Autoscaler

The Scheduler contains all the data required for making autoscaling decisions.

It is aware of how many tasks will be automatically rejected because they are *suspended* for any reason.

It is aware of how many *Periodic Tasks* are going to be scheduled in the future.

The Scheduler is aware for the maximum concurrency allowed for each worker and the *Concurrency Limitations* of specific tasks.

The Scheduler also periodically samples the queues' length.

Unfortunately, modeling such a queuing system is beyond the scope of Celery 5 due to the already large amount of new feature and changes in this version and our lack of knowledge in the math involved in such a model.

Instead we're going to provide the simple algorithm we use now in Celery 4 with some adjustments but allow room for extension.

In Celery 4 each worker checks if it should autoscale every second. This can cause a lot of thrashing as new processes are created and destroyed.

In Celery 5 after each autoscale event, there will be a cooldown period. The cooldown period increases exponentially until a configurable limit.

If the number of tasks in all the queues is larger than the current concurrency budget the Autoscaler publishes an event to all the routers. The routers will increase their prefetching multiplier as a response to this event.

The Scheduler will select which workers should increase their prefetching of tasks in order to reach the maximal concurrency budget.

Controller Internal Services

Motivation

Rationale

Backwards Compatibility

Reference Implementation

This document describes the high level architecture of Celery 5. As such, it does not have an implementation at the time of writing.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

1.4.2 CEP XXXX: Feature Release Schedule

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *Abstract*
- *Specification*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*

- *Reference Implementation*
- *Copyright*

This CEP provides a sample template for creating your own CEPs. In conjunction with the content guidelines in *CEP 1: CEP Purpose and Guidelines*, this should make it easy for you to conform your own CEPs to the format outlined below.

Note: if you are reading this CEP via the web, you should first grab the source of this CEP in order to complete the steps below. **DO NOT USE THE HTML FILE AS YOUR TEMPLATE!**

To get the source this (or any) CEP, look at the top of the Github page and click "raw".

If you're unfamiliar with reStructuredText (the format required of CEPs), see these resources:

- A ReStructuredText Primer, a gentle introduction.
- Quick reStructuredText, a users' quick reference.
- reStructuredText Markup Specification, the final authority.

Once you've made a copy of this template, remove this abstract, fill out the metadata above and the sections below, then submit the CEP. Follow the guidelines in *CEP 1: CEP Purpose and Guidelines*.

Abstract

This should be a short (~200 word) description of the technical issue being addressed.

This (and the above metadata) is the only section strictly required to submit a draft CEP; the following sections can be barebones and fleshed out as you work through the CEP process.

Specification

This section should contain a complete, detailed technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.

Motivation

This section should explain *why* this CEP is needed. The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.

Rationale

This section should flesh out the specification by describing what motivated the specific design design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

Backwards Compatibility

If this CEP introduces backwards incompatibilities, you must must include this section. It should describe these incompatibilities and their severity, and what mitigation you plan to take to deal with these incompatibilities.

Reference Implementation

If there's an implementation of the feature under discussion in this CEP, this section should include or link to that implementation and provide any notes about installing/using/trying out the implementation.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

1.4.3 CEP XXXX: Celery NextGen High Level Architecture

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Informational

Created 2020-05-27

Last-Modified 2020-05-27

Table of Contents

- *Abstract*
- *Specification*
  - *Diagram*
  - *Preface*
  - *Python Runtime*
  - *Message Types*
  - *Message Broker*
  - *Data Sources & Sinks*
    - *Data Sinks*
    - *Data Sources*
  - *Controller*
    - *Platform Manager*

    * *Foreman*
    * *Task Scheduler*
  - *Publisher*
  - *Router*
  - *Execution Platform*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

Abstract

When Celery was conceived, production environments were radically different from today.

Nowadays most applications are (or often should be):

- Deployed to a cloud provider's computing resources.
- Distributed (sometimes between data centers).
- *Available* or *Consistent* (If we store state, we must pick one according to *CAP Theorem*).
- *Network Partition Tolerant*.
- *Observable*.
- Built with scalability in mind.
- Cloud Native - The application's lifecycle is managed using Kubernetes, Swarm, or any other scheduler.
- Heterogeneous - Microservices may be written in different languages.

Also, Celery lacks proper support for large scale deployments and some useful messaging architectural patterns.

Celery 5 is the next major version of Celery, and so we can break backward compatibility, even in significant ways.

As such, our next major version should represent the beginning of a paradigm shift in the way we implement our task execution platform. Future major versions will drastically change how Celery works.

This document provides a high-level overview of the new architecture for the next generation of Celery major versions.

Specification

Note: From now on when we write Celery we refer to Celery NextGen.

Whenever we refer to a previous major version of Celery we will specify the version number.

Diagram

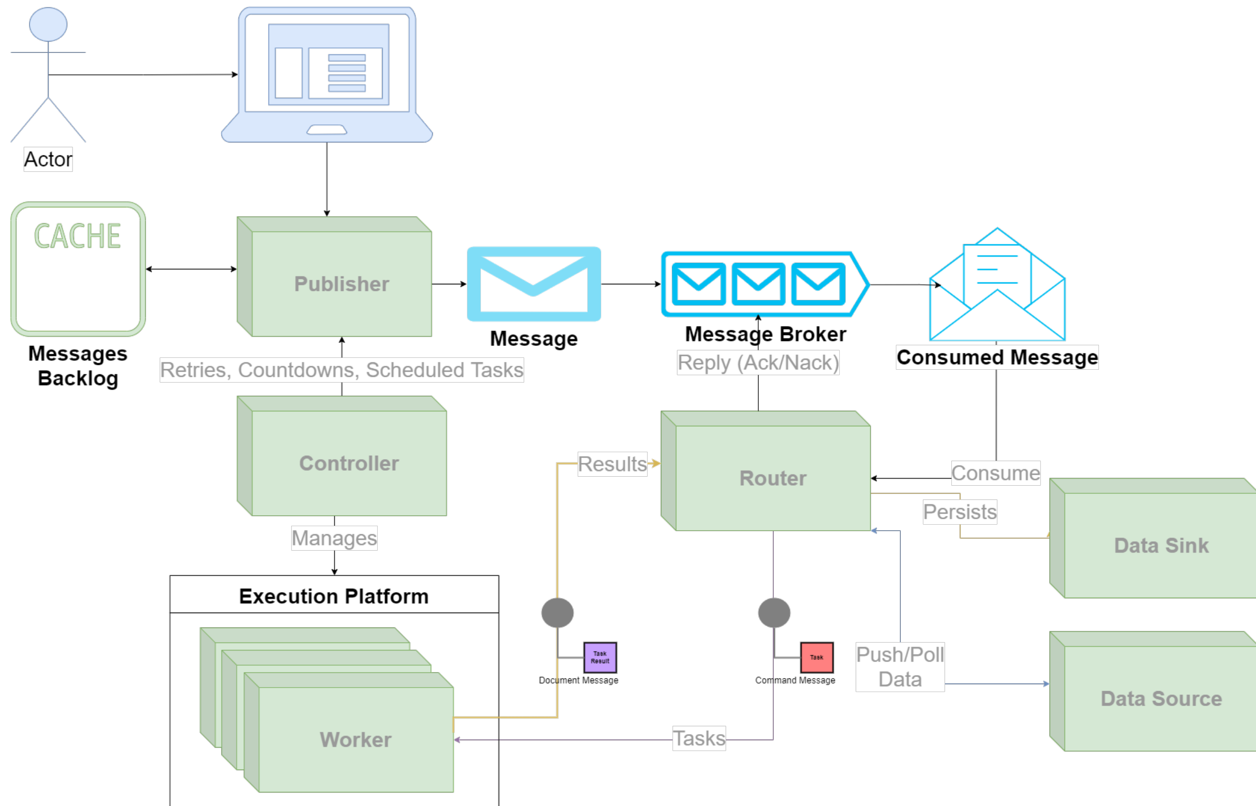


Fig. 2: High Level Architecture Diagram

Preface

In Celery 4.x we had the following architectural building blocks:

- A *Python* Runtime
- *Message Broker*
- *Result Backend*
- *Celery Master*
- *Celery Worker*
- *Celery Beat*

In addition we had a few optional architectural building blocks (some of them maintained by the community):

- *Celery Multi*
- *Flower*
- *Cell*

The only architectural building blocks that remain in Celery are the *Python* Runtime and the *Message Broker*. The rest are replaced by new ones which provide more functionality and flexibility for our users.

In the rest of this specification we will describe the architectural building blocks of Celery.

Python Runtime

Python 2 no longer receives support and security patches from the community as of January 1st, 2020. Therefore, Celery will drop support for Python 2 and will run on Python 3 and above.

Celery supports the *CPython* and *PyPy* Python runtimes. Other runtimes will be considered in different CEPs on a case-by-case basis.

As a general guideline, we will attempt to keep support for the latest Python version *PyPy* supports. However, if the need arises we will opt to use new Python language features and hope *PyPy* can catch up.

Message Types

In Celery 4.x we only have tasks which are serialized to *Command Messages* that we publish to the *Message Broker* whenever we want to execute a *Task*.

Document messages are the messages we got as a result. Those message were stored in the *Result Backend*. They had a specific format which only the Celery 4.x library knew how to parse.

In Celery, we now have new types of messages we can use.

Document Messages may now also be produced whenever we publish a serialized representation of a *Domain Model* to the *Message Broker*. These messages may be received from a Data Source or published directly from the application.

Event Messages are a new concept for Celery. They describe that a *Domain Event* occurred. Multiple tasks can subscribe to an event. Whenever we receive an *Event Message* we publish those tasks as *Command Messages* to the *Message Broker*.

These fundamental architectural building blocks will aid us in creating a better messaging system. To encourage *Ubiquitous Language*, we will be using them in this document when applicable and in Celery's codebase as well.

Message Broker

In Celery 4.x each *Celery Master* connected to only one *Message Broker* cluster.

This is no longer the case. Celery now allows connecting to multiple *Message Brokers* even if they are of clusters that use different implementations of a message broker.

Users can consume messages from a Redis cluster, a RabbitMQ cluster, and an ActiveMQ cluster if they so desire.

This feature is useful when, for example:

- The user migrates from a legacy system that uses other implementation of a *Message Broker*, but the new system uses a more modern one.
- The user wants to split the load between clusters.
- There's a security reason to publish some messages to a specific cluster.

On some *Message Broker* implementations the *Controller* will assist in managing the cluster.

Data Sources & Sinks

In Celery 4.x we had a *Result Backend* which was used to store task results and coordinate the execution of chords.

We extend the *Result Backend* concept further to allow new use cases such:

- *ETL*.
- *Data Integration*.
- Reporting.
- Taking action when data is inserted or updated.

In addition, like we did for the *Message Broker*, we now allow multiple data sources and sinks instead of one cluster of a *Result Backend*.

The responsibility for coordination of the execution of chords has moved to the *Execution Platform*.

Data Sinks

A data sink is where task results are saved.

A task result may be saved in more than one data sink (e.g. a Kafka Topic and S3).

The *Router* is responsible for routing task results to the correct data sink(s) and properly serializing them.

Data Sources

A data source is a anything that stores data. It can be a Kafka topic, a S3 bucket, a RDBMS or even your local filesystem.

Some data sources can notify Celery of incoming data. Others, Celery needs to poll periodically using the *Task Scheduler*.

The *Router* is responsible for listening to incoming data from the various data sources connected to it.

Whenever the Router receives incoming data it sends a *Document Message* to the *Publisher* which in turn will publish it to the *Message Broker*.

Controller

In Celery 4.x we provided a basic tool for controlling Celery instances called *Celery Multi*. We also provided *Celery Beat* for periodically scheduling tasks.

The Controller replaces those two components and extends their functionality in many ways.

The Controller is responsible for managing the lifecycle of all other Celery components.

Celery is a complex system with multiple components and will often be deployed in high throughput, highly available, cloud-native production systems.

The introduction of multiple components require us to have another component that manages the entire Celery cluster.

Instead of controlling Celery instances on one machine in a way that is agnostic to the production environment we're operating in the Controller now provides a *Platform Manager* which manages Celery instances on one or many machines.

The Controller also manages and optimizes the execution of tasks to ensure we maximize the utilization of all our resources and to prevent expected errors.

That is why the Controller is now responsible for auto-scaling Celery instances, rate-limiting tasks, task concurrency limitations, task execution prioritization and all other management operations a user needs to operate a Celery cluster.

Platform Manager

The Platform Manager is responsible for interacting with the production environment.

The platform itself can be anything Celery can run on e.g.: Pre-fork, SystemD, OpenRC, Docker, Swarm, Kubernetes, Nomad.

Each implementation of the Platform Manager will be provided in a different package. Some of them will be maintained by the community.

Foreman

The Foreman is responsible for deploying & managing the lifecycle of all Celery instances and ensuring they stay up and running.

It can spawn new instances of Celery processes, stop or restart them either on demand or based on policies the user has specified for auto-scaling.

It interacts with the *Platform Manager* to do so on the platform the Controller manages.

On some platforms, the Foreman can instruct the *Platform Manager* to deploy and manage the lifecycle of *Message Brokers*, Data Sources & Data Sinks.

Task Scheduler

The Task Scheduler is responsible for managing the scheduling of tasks for execution on a cluster of workers.

The scheduler calculates the amount of tasks to be executed in any given time in order to make cluster wide decisions when autoscaling workers or increasing concurrency for an existing worker.

The scheduler is aware when tasks should no longer be executed for any reason. To do so, it commands the router to avoid consuming the task or rejecting it.

Publisher

The Publisher is responsible for publishing *Messages* to a *Message Broker*.

It is responsible for publishing a *Message* to the appropriate *Message Broker* cluster according to the configuration provided to the publisher.

The publisher must be able to run in-process inside a long-running thread or a long running co-routine.

It can also be run using a separate daemon which can serve all the processes publishing to the message brokers.

Whenever the *Message Broker* cluster is unavailable or unresponsive, the Publisher stores the *Messages* in the Messages Backlog. The Publisher will later retry publishing the message.

Router

In Celery 4.x there was a *Celery Master* which was responsible for consuming *Messages*, storing results and executing canvases.

In Celery we now decouple the execution of the tasks completely from the routing of *Messages* to the *Execution Platform*.

The Router is a combination of the following architectural patterns: - *Event Driven Consumer* - *Message Dispatcher* - *Service Activator* - *Process Manager*.

The Router consumes *Messages* from a *Message Broker* cluster or multiple clusters, dispatches the consumed *Messages* to the *Execution Platform* for processing and awaits for the results of the task and stores them in the appropriate Data Sink(s).

When the execution of a task requires coordination in case of a workflow, an *Event Message* or an incoming *Document Message* from a Data Source the Router is responsible for the order of the execution of the task(s).

Execution Platform

In Celery 4.x we had *Celery Worker* which was responsible for executing tasks.

In Celery we now have an execution platform which will execute tasks and report their results back to the *Router*.

Celery will provide a default implementation but other implementations may be provided as well by the community or the maintainers.

Alternative implementations may provide task execution on *Serverless Computing* platforms, as Kubernetes Jobs for example or provide task execution for tasks written in a different language.

Motivation

We want to modernize Celery for the Cloud Native age. We need to keep Celery relevant for our users and help them in new ways. Therefore, we must adjust and evolve to meet the unique challenges of the Cloud Native age.

Also, we want to modernize the code to support Python 3+, which will allow us to remove workarounds, backports, and compatibility shims. Refactoring the codebase to support Python 3+ allows us to keep a slimmer, more maintainable codebase.

Furthermore, we'd like to resolve long-standing design bugs in our implementation.

Gradually evolving our codebase is currently not possible due to the many changes in technology since Celery was conceived. We need to move fast and break things until we match all our goals.

Rationale

This architecture moves Celery from the past to the present.

We have the following goals:

- Observability
- Cloud Nativity
- Flexibility
- Increased Task Completion Throughput

The proposed architecture will guarantee that those goals will be met.

Backwards Compatibility

As evident from this document, Celery NextGen is not backwards compatible with previous versions of Celery.

The syntax for defining tasks will remain the same so you can reuse your code with little to no adjustments.

Reference Implementation

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

1.4.4 CEP XXXX: Controller

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *Abstract*
- *Specification*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

This CEP provides a sample template for creating your own CEPs. In conjunction with the content guidelines in *CEP 1: CEP Purpose and Guidelines*, this should make it easy for you to conform your own CEPs to the format outlined below.

Note: if you are reading this CEP via the web, you should first grab the source of this CEP in order to complete the steps below. **DO NOT USE THE HTML FILE AS YOUR TEMPLATE!**

To get the source this (or any) CEP, look at the top of the Github page and click “raw”.

If you’re unfamiliar with reStructuredText (the format required of CEPs), see these resources:

- A ReStructuredText Primer, a gentle introduction.
- Quick reStructuredText, a users’ quick reference.

- reStructuredText Markup Specification, the final authority.

Once you've made a copy of this template, remove this abstract, fill out the metadata above and the sections below, then submit the CEP. Follow the guidelines in *CEP 1: CEP Purpose and Guidelines*.

Abstract

This should be a short (~200 word) description of the technical issue being addressed.

This (and the above metadata) is the only section strictly required to submit a draft CEP; the following sections can be barebones and fleshed out as you work through the CEP process.

Specification

This section should contain a complete, detailed technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.

Motivation

This section should explain *why* this CEP is needed. The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.

Rationale

This section should flesh out the specification by describing what motivated the specific design design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

Backwards Compatibility

If this CEP introduces backwards incompatibilities, you must must include this section. It should describe these incompatibilities and their severity, and what mitigation you plan to take to deal with these incompatibilities.

Reference Implementation

If there's an implementation of the feature under discussion in this CEP, this section should include or link to that implementation and provide any notes about installing/using/trying out the implementation.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

1.4.5 CEP XXXX: Publisher

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *Abstract*
- *Specification*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

This CEP provides a sample template for creating your own CEPs. In conjunction with the content guidelines in *CEP 1: CEP Purpose and Guidelines*, this should make it easy for you to conform your own CEPs to the format outlined below.

Note: if you are reading this CEP via the web, you should first grab the source of this CEP in order to complete the steps below. **DO NOT USE THE HTML FILE AS YOUR TEMPLATE!**

To get the source this (or any) CEP, look at the top of the Github page and click "raw".

If you're unfamiliar with reStructuredText (the format required of CEPs), see these resources:

- A ReStructuredText Primer, a gentle introduction.
- Quick reStructuredText, a users' quick reference.
- reStructuredText Markup Specification, the final authority.

Once you've made a copy of this template, remove this abstract, fill out the metadata above and the sections below, then submit the CEP. Follow the guidelines in *CEP 1: CEP Purpose and Guidelines*.

Abstract

This should be a short (~200 word) description of the technical issue being addressed.

This (and the above metadata) is the only section strictly required to submit a draft CEP; the following sections can be barebones and fleshed out as you work through the CEP process.

Specification

This section should contain a complete, detailed technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.

Motivation

This section should explain *why* this CEP is needed. The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.

Rationale

This section should flesh out the specification by describing what motivated the specific design design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

Backwards Compatibility

If this CEP introduces backwards incompatibilities, you must must include this section. It should describe these incompatibilities and their severity, and what mitigation you plan to take to deal with these incompatibilities.

Reference Implementation

If there's an implementation of the feature under discussion in this CEP, this section should include or link to that implementation and provide any notes about installing/using/trying out the implementation.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

1.4.6 CEP XXXX: Router

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *Abstract*
- *Specification*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

This CEP provides a sample template for creating your own CEPs. In conjunction with the content guidelines in *CEP 1: CEP Purpose and Guidelines*, this should make it easy for you to conform your own CEPs to the format outlined below.

Note: if you are reading this CEP via the web, you should first grab the source of this CEP in order to complete the steps below. **DO NOT USE THE HTML FILE AS YOUR TEMPLATE!**

To get the source this (or any) CEP, look at the top of the Github page and click "raw".

If you're unfamiliar with reStructuredText (the format required of CEPs), see these resources:

- A ReStructuredText Primer, a gentle introduction.
- Quick reStructuredText, a users' quick reference.
- reStructuredText Markup Specification, the final authority.

Once you've made a copy of this template, remove this abstract, fill out the metadata above and the sections below, then submit the CEP. Follow the guidelines in *CEP 1: CEP Purpose and Guidelines*.

Abstract

This should be a short (~200 word) description of the technical issue being addressed.

This (and the above metadata) is the only section strictly required to submit a draft CEP; the following sections can be barebones and fleshed out as you work through the CEP process.

Specification

This section should contain a complete, detailed technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.

Motivation

This section should explain *why* this CEP is needed. The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.

Rationale

This section should flesh out the specification by describing what motivated the specific design design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

Backwards Compatibility

If this CEP introduces backwards incompatibilities, you must must include this section. It should describe these incompatibilities and their severity, and what mitigation you plan to take to deal with these incompatibilities.

Reference Implementation

If there's an implementation of the feature under discussion in this CEP, this section should include or link to that implementation and provide any notes about installing/using/trying out the implementation.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

1.4.7 CEP XXXX: Execution Platform

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *Abstract*
- *Specification*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

This CEP provides a sample template for creating your own CEPs. In conjunction with the content guidelines in *CEP 1: CEP Purpose and Guidelines*, this should make it easy for you to conform your own CEPs to the format outlined below.

Note: if you are reading this CEP via the web, you should first grab the source of this CEP in order to complete the steps below. **DO NOT USE THE HTML FILE AS YOUR TEMPLATE!**

To get the source this (or any) CEP, look at the top of the Github page and click "raw".

If you're unfamiliar with reStructuredText (the format required of CEPs), see these resources:

- A ReStructuredText Primer, a gentle introduction.
- Quick reStructuredText, a users' quick reference.
- reStructuredText Markup Specification, the final authority.

Once you've made a copy of this template, remove this abstract, fill out the metadata above and the sections below, then submit the CEP. Follow the guidelines in *CEP 1: CEP Purpose and Guidelines*.

Abstract

This should be a short (~200 word) description of the technical issue being addressed.

This (and the above metadata) is the only section strictly required to submit a draft CEP; the following sections can be barebones and fleshed out as you work through the CEP process.

Specification

This section should contain a complete, detailed technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.

Motivation

This section should explain *why* this CEP is needed. The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.

Rationale

This section should flesh out the specification by describing what motivated the specific design design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

Backwards Compatibility

If this CEP introduces backwards incompatibilities, you must must include this section. It should describe these incompatibilities and their severity, and what mitigation you plan to take to deal with these incompatibilities.

Reference Implementation

If there's an implementation of the feature under discussion in this CEP, this section should include or link to that implementation and provide any notes about installing/using/trying out the implementation.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

1.4.8 CEP XXXX: Configuration

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *Abstract*
- *Specification*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

This CEP provides a sample template for creating your own CEPs. In conjunction with the content guidelines in *CEP 1: CEP Purpose and Guidelines*, this should make it easy for you to conform your own CEPs to the format outlined below.

Note: if you are reading this CEP via the web, you should first grab the source of this CEP in order to complete the steps below. **DO NOT USE THE HTML FILE AS YOUR TEMPLATE!**

To get the source this (or any) CEP, look at the top of the Github page and click "raw".

If you're unfamiliar with reStructuredText (the format required of CEPs), see these resources:

- A ReStructuredText Primer, a gentle introduction.
- Quick reStructuredText, a users' quick reference.
- reStructuredText Markup Specification, the final authority.

Once you've made a copy of this template, remove this abstract, fill out the metadata above and the sections below, then submit the CEP. Follow the guidelines in *CEP 1: CEP Purpose and Guidelines*.

Abstract

This should be a short (~200 word) description of the technical issue being addressed.

This (and the above metadata) is the only section strictly required to submit a draft CEP; the following sections can be barebones and fleshed out as you work through the CEP process.

Specification

This section should contain a complete, detailed technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.

Motivation

This section should explain *why* this CEP is needed. The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.

Rationale

This section should flesh out the specification by describing what motivated the specific design design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

Backwards Compatibility

If this CEP introduces backwards incompatibilities, you must must include this section. It should describe these incompatibilities and their severity, and what mitigation you plan to take to deal with these incompatibilities.

Reference Implementation

If there's an implementation of the feature under discussion in this CEP, this section should include or link to that implementation and provide any notes about installing/using/trying out the implementation.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

1.4.9 CEP XXXX: Observability

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *Abstract*
- *Specification*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

This CEP provides a sample template for creating your own CEPs. In conjunction with the content guidelines in *CEP 1: CEP Purpose and Guidelines*, this should make it easy for you to conform your own CEPs to the format outlined below.

Note: if you are reading this CEP via the web, you should first grab the source of this CEP in order to complete the steps below. **DO NOT USE THE HTML FILE AS YOUR TEMPLATE!**

To get the source this (or any) CEP, look at the top of the Github page and click "raw".

If you're unfamiliar with reStructuredText (the format required of CEPs), see these resources:

- A ReStructuredText Primer, a gentle introduction.
- Quick reStructuredText, a users' quick reference.
- reStructuredText Markup Specification, the final authority.

Once you've made a copy of this template, remove this abstract, fill out the metadata above and the sections below, then submit the CEP. Follow the guidelines in *CEP 1: CEP Purpose and Guidelines*.

Abstract

This should be a short (~200 word) description of the technical issue being addressed.

This (and the above metadata) is the only section strictly required to submit a draft CEP; the following sections can be barebones and fleshed out as you work through the CEP process.

Specification

This section should contain a complete, detailed technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.

Motivation

This section should explain *why* this CEP is needed. The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.

Rationale

This section should flesh out the specification by describing what motivated the specific design design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

Backwards Compatibility

If this CEP introduces backwards incompatibilities, you must must include this section. It should describe these incompatibilities and their severity, and what mitigation you plan to take to deal with these incompatibilities.

Reference Implementation

If there's an implementation of the feature under discussion in this CEP, this section should include or link to that implementation and provide any notes about installing/using/trying out the implementation.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

1.4.10 CEP XXXX: Celery Kubernetes Operator - Architecture Document

CEP XXXX

Author Gautam Prajapati

Implementation Team Gautam Prajapati

Shepherd Omer Katz, Asif Saif Uddin

Status Draft

Type Feature

Created 2020-08-30

Last-Modified 2020-08-30

Table of Contents

- *Abstract*
- *Specification*
  - *Scope*
  - *Diagram*
  - *Workflow*
  - *Components*
    - *Worker Deployment*
    - *Flower Deployment*
    - *Flower Service*
    - *Celery CRD(Custom Resource Definition)*
    - *Celery CR(Custom Resource)*
    - *Custom Controller*
  - *Async KOPF Handlers(Controller Implementation)*
    - *Creation Handler*
    - *Updation Handler*
  - *Autoscaling*
    - *Native Metrics(CPU, Memory Utilization)*
    - *Broker Queue Length(KEDA based autoscaling)*
  - *Deployment Strategy*
- *Motivation*
- *Rationale*
- *Reference Implementation*
  - *Want to Help?*
- *Future Work*

- *Copyright*

Abstract

Kubernetes is a popular deployment target nowadays. To run Celery applications on Kubernetes, there are manual steps involved like -

- Writing deployment spec for workers
- Setting up monitoring using Flower
- Setting up Autoscaling

Apart from that, there's no standard or consistent way to set up multiple clusters, everyone configures their own way which could create problems for infrastructure teams to manage and audit later.

This document proposes writing a Kubernetes Operator for automating management of Celery clusters. This CEP is currently written keeping Celery 4.X in mind. Controller implementation will differ for Celery 5 and is under discussion.

Specification

Scope

1. Provide a Custom Resource Definition(CRD) to spec out a Celery and Flower deployment having all the configuration options that they support.
2. A custom controller implementation that registers and manages self-healing capabilities of custom Celery resource for these operations -
   - CREATE - Creates the worker and flower deployments along with exposing a native Service object for Flower
   - UPDATE - Reads the CRD modifications and updates the running deployments using specified strategy
   - DELETE - Deletes the custom resource and all the child deployments
3. Support worker autoscaling/downscaling based on resource constraints(cpu, memory) and task queue length automatically.

Discussions involving other things that this operator should do based on your production use-case are welcome.

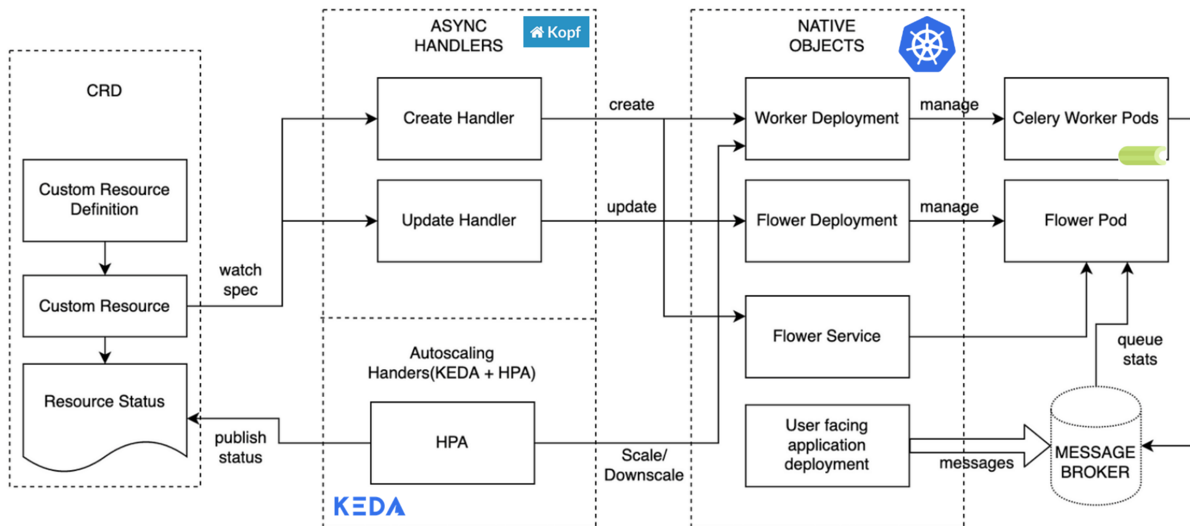
Diagram

Workflow

End user starts by writing and creating a YAML spec for the desired celery cluster. Creation event is listened by the Creation Handler(KOPF based) which creates deployment for workers, flower and a Service object to expose flower UI to external users.

Assuming we have broker in place, any user facing application can start pushing messages to broker now and celery workers will start processing them.

User can update the custom resource, when that happens, updation handler listening to the event will patch the relevant deployments for change. Rollout strategy can be default or to be specified by user in the spec.



Both creation and updation handlers will return their statuses to be stored in parent resource's status field. Status field will contain the latest status of the cluster children at all times.

User can choose to setup autoscaling of workers by resource constraints(CPU, Memory) or broker queue length. Operator will automatically take care of creating an HPA or use KEDA based autoscaling(See *Autoscaling* section below) to make that happen.

Components

Worker Deployment

A Kubernetes Deployment to manage celery worker pods/replicaset. These workers consume the tasks from broker and process them.

Flower Deployment

A Kubernetes Deployment to manage flower pods/replicaset. Flower is de-facto standard to monitor and remote control celery.

Flower Service

Expose flower UI to an external IP through a Kubernetes Service object. We should additionally explore Ingress as well.

Celery CRD(Custom Resource Definition)

CRDs are a native way to extend Kubernetes APIs to recognize custom applications/objects. Celery CRD will contain the schema for celery cluster to be setup.

We plan to have following objects in place with their high level description -

- **`common` - common configuration parameters for Celery cluster**
  - `image` - Celery application image to be run
  - `imagePullPolicy` - [Always, Never, IfNotPresent]
  - `imagePullSecrets` - to pull the image from a private registry
  - `volumeMounts` - describes mounting of a volume within container.
  - `volumes` - describes a volume to be used for storage
  - `celeryVersion` - Celery version
  - `appName` - App name for worker and flower deployments
  - `celeryApp` - celery app instance to use (e.g. module.celery_app_attr_name)
- **`workerSpec` - worker deployment specific parameters**
  - `numOfWorkers` - Number of workers to launch initially
  - `args` - array of arguments(all celery supported options) to pass to worker process in container (TODO: Entrypoint vs args vs individual params)
  - `rolloutStrategy` - Rollout strategy to spawn new worker pods
  - `resources` - optional argument to specify cpu, mem constraints for worker deployment
- **`flowerSpec` - flower deployment and service specific parameters**
  - `replicas` - Number of replicas for flower deployment
  - `args` - array of arguments(all flower supported options) to pass to flower process in the container
  - `servicePort` - Port to expose flower UI in the container
  - `serviceType` - [Default, NodePort, LoadBalancer]
  - `resources` - optional argument to specify cpu, mem constraints for flower deployment
- **`scaleTargetRef` - array of items describing auto scaling targets**
  - `kind` - which application kind to scale (worker, flower)
  - `minReplicas` - min num of replicas
  - `maxReplicas` - max num of replicas
  - **`metrics` - list of metrics to monitor**
    - `name` - Enum type (memory, cpu, task_queue_length)
    - **`target` - target values**
      - `type` - [Utilization, Average Value]
      - `averageValue/averageUtilization` - Average values to maintain

A more detailed version/documentation for CRD spec is underway.

Celery CR(Custom Resource)

Custom Resource Object for a Celery application. Multiple clusters will have multiple custom resource objects.

Custom Controller

Custom controller implementation to manage Celery applications(CRs). Contains the code for creation, updation, deletion and scaling handlers of the cluster. We plan to use open-source KOPF (Kubernetes Operator Pythonic Framework) to write this implementation.

Async KOPF Handlers(Controller Implementation)

This section contains brief overview of creation and updation handlers which are going to react on celery resource creation and updation respectively and return their status to be stored back as resource's status.

Creation Handler

Generates deployment spec for worker and flower deployments dynamically based on incoming parameters specified in custom celery resource. Also creates the flower service to expose flower UI. Status of each children is sent back to be stored under parent resource status field.

Additionally, it might handle the HPA object creation too if the scaling is to be done on native metrics(CPU and Memory utilization).

Updation Handler

Updates deployment spec for worker and flower deployments(and service - HPA) dynamically and patch them. Status of each children is sent back to be stored under parent resource status field.

Autoscaling

This section covers how operator is going to handle autoscaling. We plan to supporting scaling based on following two metrics.

Native Metrics(CPU, Memory Utilization)

If workers need to be scaled only on CPU/Memory constraints, we can simply create an HPA object in creation/updation handlers and it'll take care of scaling relevant worker deployment automatically. HPA supports these two metrics out of the box. For custom metrics, we need to do additional work.

Broker Queue Length(KEDA based autoscaling)

Queue Length based scaling needs custom metric server for an HPA to work. KEDA is a wonderful option because it is built for the same. It provides the scalers for all the popular brokers(RabbitMQ, Redis, Amazon SQS) supported in Celery.

KEDA provides multiple ways to be deployed on a Kubernetes cluster - Helm, Operator Hub and Yaml. Celery Operator can package KEDA along with itself for distribution.

Deployment Strategy

Probably the best way would be distribute a Helm Chart which packages CRD, controller and KEDA together(More to be explored here). We'll also support YAML apply based deployments.

Additionally, Helm approach is extensible in the sense that we can package additional components like preferred broker(Redis, RMQ, SQS) as well to start with Celery on Kubernetes out of the box without much efforts.

Motivation

Celery is one of the most popular distributed task queue system and Kubernetes is the de-facto standard for container-orchestration nowadays. Running and managing Celery applications on Kubernetes is a largely manual process. Having a Kubernetes operator to help setup and manage celery in a consistent and graceful way would benefit users immensely. We'd also be able to bridge the gap between application engineers and infrastructure operators who manually manage the celery clusters.

Rationale

Having this operator will automate the setup and management of a Celery cluster. It'll also enable Celery project which is written in Python to spearhead the development of production ready Kubernetes extensions which are generally written in Golang.

The rationale behind using KOPF and Python to building this operator is that there is a good learning curve to understand internals and write(also maintain) an operator with Go. It'll also motivate community to overcome the learning barrier and create useful libraries, tools and other operators while staying in Python ecosystem.

Reference Implementation

Github Repo - https://github.com/brainbreaker/Celery-Kubernetes-Operator Steps to try out are available in the Readme.

Also a talk in EuroPython 2020 describing this POC implementation and demo - Youtube

Want to Help?

If you're running celery on a Kubernetes cluster, your inputs to how you manage applications will be valuable. You could contribute to the discussion here.

Future Work

This issue lists open points for the operator

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

1.4.11 CEP XXXX: CLI Refactor

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-11-10

Last-Modified 2019-11-10

Table of Contents

- *Abstract*
- *Specification*
  - *CLI Context*
  - *Parameter Types*
  - *Plugins*
  - *Acceptance Test Suite*
- *Motivation*
- *Rationale*
  - *Alternative CLI Frameworks*
  - *Parameter Types*
- *Backwards Compatibility*
  - *User Options*
  - *Preload Options*
- *Reference Implementation*
- *Copyright*

Abstract

Celery's CLI infrastructure is based on a custom framework built using the `argparse` built-in module. This implementation has multiple design defects and a few bugs and as a result a developer cannot easily read the code or extend it.

As we're moving towards Celery 5, we are likely to add new sub-commands and/or enhance existing ones. Therefore, refactoring this part of the codebase will increase our future productivity and flexibility.

This CEP proposes to refactor our CLI implementation in order to make that part of the code developer friendly and user extensible.

Specification

The refactor replaces our custom `argparse` framework which can be found here with an implementation using Click.

CLI Context

Instead of sharing common functionality in the *Command* base class we introduce a class called `CLIContext` which provides access the Celery application, helper methods for printing informational messages or errors and other common functionality.

Parameter Types

The current implementation extracted common parsing and validation code of parameter types into custom types which can be reused across our CLI implementation.

Plugins

Click allows extending existing CLIs using setuptools's entrypoints using the click-plugins extension.

Frameworks which base themselves over Celery will now be able to extend or customize the CLI for their needs.

Acceptance Test Suite

The former implementation was only covered by unit tests which did not cover the entire surface of the implementation.

The new implementation will be covered by both unit tests and BDD-style acceptance tests.

Motivation

The main purpose of this refactor is to allow us to use an event loop using Python 3's *async/await* syntax without investing further in our custom CLI framework. Instead we opt to use a battle-tested solution which allows us to remove the entire custom framework entirely.

This allows us to delegate the maintenance overhead to others and reduce the surface of potential bugs introduced in Celery 5.

Argparse which our previous implementation was based on wasn't a good fit. We needed to create a framework around it to support sub-commands such as *celery worker* and nested sub-commands were not possible at all which means that each command that had them figured out how they should be implemented for itself. In addition, We had to implement

a REPL nearly from scratch for *celery amqp* while Click has a plugin which uses the python-prompt-toolkit library called click-repl.

This resulted in an implementation of ~3k LOC (without spaces or comments).

By using Click, our new implementation has only ~2.2K LOC (without spaces or comments). This 27% reduction makes the code easier to reason about and is simpler due to Click's API.

The Click ecosystem provides us with many features that argparse lacks such as "Did you mean" messages, automatic documentation using Sphinx and other user experience enhancing features which `argparse` lacks.

We use these extensions for enriching our CLI implementation.

Rationale

Alternative CLI Frameworks

Docopt was considered as part of this effort but was found insufficient for our needs.

While Doctopt does support sub-commands, it does not dispatching them to functions which requires us to write the same type of framework we wanted to avoid.

Furthermore, Docopt does not parse parameter types and leaves that to the implementor.

Docopt however does allow us to customize our help page better.

The aforementioned disadvantages outweigh the only advantage.

Parameter Types

Our previous implementation used to parse and validate some of the arguments during the actual execution of the command. No infrastructure was provided to share the implementation of parsing and validating such special arguments such as ISO-8601 date time strings or comma separated lists.

This resulted in violation of the DRY principle and more importantly the Single Responsibility Principle (SRP).

Violating SRP makes unit testing harder as there are more code paths to take care of. This violation also increases the difficulty of reasoning about the code in question for the same reason.

The current implementation separates the responsibility of parsing and validating arguments from the command invocation itself to small classes which are very easy to unit test and reason about.

Backwards Compatibility

This CEP is almost completely backwards compatible with our previous implementation.

The only changes in our API are around the CLI's customization.

User Options

User Options now pass the relevant Click Command object to the callbacks.

If you are using this feature you have to migrate your code from Argparse to Click.

In addition the API changed. Previously the following code was required:

```
def add_worker_arguments(parser):
    parser.add_argument(
        '--enable-my-option', action='store_true', default=False,
        help='Enable custom option.',
    ),
app.user_options['worker'].add(add_worker_arguments)
```

With this refactor you either need to set the relevant `user_options` key with a list of `click.Option``s or ``click.Argument`'s or provide a callback which will return those.

```
import click
app.user_options['worker'] = [click.Option('--enable-my-option', is_flag=True, help=
↪'Enable custom option')]
```

Preload Options

Preload options are User Options and are subject to the same breaking change.

In addition the signal's sender is now changed to the `click.Context` of the invoked command.

Reference Implementation

The reference implementation can be found at celery/celery#5718.

Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

1.5 Rejected CEPs

CEPs that have been rejected by the Technical Board. See CEP 1 for details.

1.6 Superseded CEPs

CEPs that have been replaced/superseded by newer CEPs. See CEP 1 for details.

1.7 Withdrawn CEPs

CEPs that have been withdrawn by their authors. See CEP 1 for details.

1.8 CEP XXXX: CEP template

CEP XXXX

Author Omer Katz

Implementation Team Omer Katz

Shepherd Omer Katz

Status Draft

Type Feature

Created 2019-04-03

Last-Modified 2019-04-03

Table of Contents

- *Abstract*
- *Specification*
- *Motivation*
- *Rationale*
- *Backwards Compatibility*
- *Reference Implementation*
- *Copyright*

This CEP provides a sample template for creating your own CEPs. In conjunction with the content guidelines in *CEP 1: CEP Purpose and Guidelines*, this should make it easy for you to conform your own CEPs to the format outlined below.

Note: if you are reading this CEP via the web, you should first grab the source of this CEP in order to complete the steps below. **DO NOT USE THE HTML FILE AS YOUR TEMPLATE!**

To get the source this (or any) CEP, look at the top of the Github page and click "raw".

If you're unfamiliar with reStructuredText (the format required of CEPs), see these resources:

- A ReStructuredText Primer, a gentle introduction.
- Quick reStructuredText, a users' quick reference.
- reStructuredText Markup Specification, the final authority.

Once you've made a copy of this template, remove this abstract, fill out the metadata above and the sections below, then submit the CEP. Follow the guidelines in *CEP 1: CEP Purpose and Guidelines*.

1.8.1 Abstract

This should be a short (~200 word) description of the technical issue being addressed.

This (and the above metadata) is the only section strictly required to submit a draft CEP; the following sections can be barebones and fleshed out as you work through the CEP process.

1.8.2 Specification

This section should contain a complete, detailed technical specification should describe the syntax and semantics of any new feature. The specification should be detailed enough to allow implementation – that is, developers other than the author should (given the right experience) be able to independently implement the feature, given only the CEP.

1.8.3 Motivation

This section should explain *why* this CEP is needed. The motivation is critical for CEPs that want to add substantial new features or materially refactor existing ones. It should clearly explain why the existing solutions are inadequate to address the problem that the CEP solves. CEP submissions without sufficient motivation may be rejected outright.

1.8.4 Rationale

This section should flesh out the specification by describing what motivated the specific design design and why particular design decisions were made. It should describe alternate designs that were considered and related work.

The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

1.8.5 Backwards Compatibility

If this CEP introduces backwards incompatibilities, you must must include this section. It should describe these incompatibilities and their severity, and what mitigation you plan to take to deal with these incompatibilities.

1.8.6 Reference Implementation

If there's an implementation of the feature under discussion in this CEP, this section should include or link to that implementation and provide any notes about installing/using/trying out the implementation.

1.8.7 Copyright

This document has been placed in the public domain per the Creative Commons CC0 1.0 Universal license (https://creativecommons.org/publicdomain/zero/1.0/deed).

(All CEPs must include this exact copyright statement.)

CHAPTER TWO

INDICES AND TABLES

- genindex
- search

INDEX

A

Availability, 2

C

CAP Theorem, 2
Celery Beat, 4
Celery Master, 4
Celery Multi, 4
Celery Worker, 4
Cell, 4
Circuit Breaker, 1
Command Message, 1
Consistency, 2
CPython, 4

D

Data Integration, 4
Document Message, 1
Domain Event, 3
Domain Model, 3

E

ETL, 4
Event Driven Consumer, 1
Event Message, 1

F

Fault Tolerance, 2
Flower, 4

G

GIL, 3

I

Idempotent Receiver, 1
IPC, 3

J

JSON, 3

M

Message, 1
Message Broker, 1
Message Dispatcher, 1
Monitoring, 2

N

Network Resilience, 2

O

Observability, 2

P

Partition Tolerant, 2
Process Manager, 1
PyPy, 4
Python, 4

R

Result Backend, 4

S

Serverless Computing, 4
Service Activator, 1
Service Locator, 3
stdout, 3
Structured Logging, 3

T

Task, 3

U

Ubiquitous Language, 4