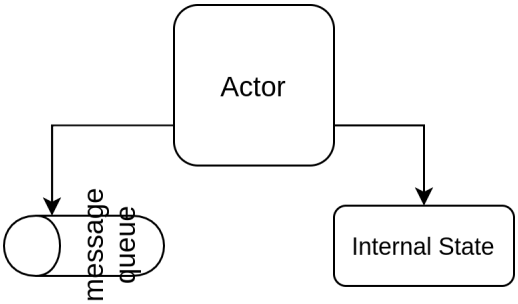
Libactors

libactors repository

> Libactors - asyncio concurrency framework without locks based on the actor model.

The actor model in computer science is a mathematical **model of concurrent computation** that treats *actors* as the universal primitives of concurrent computation. The communication in the system is done via messages, actors can send messages each other and react to them. In the actor model, each actor can handle only one message at a time, in other words, message handling is done under a hidden "lock".

Actor Model



An actor can be thought of as a simple worker; It has its own mailbox which is usually implemented using a message queue. Unlike us, actor is a lazy worker, if his mailbox is empty - he rests, waiting until a new message will be sent to him. Apparently, actors are males, since they cannot do two things in parallel. An actor will only handle a single message at a time, according to the insertion order (FIFO style) into his mailbox. When an actor handles a message it can:

- send messages to other actors
- create new actors
- modify its own state
- I/O and computational operations

Once the actor started handling a message, his only concern is to finish it and it will not rest nor skip to another message until he finishes handling the message completely.

Each actor holds its own private state, which only the actor itself may read, modify, or access. In fact, actors are not aware of other actor's states at all. This distinction results in the following property:

> *Since only one message is handled at a time, the actor can access and modify its state without worrying about it becoming corrupt because of race conditions.*

Think about it for a moment, if the only entity that can access and modify an actor's state is the actor itself and as long as there are no messages in the mailbox, the actor does nothing and even when there are messages in the queue, they are handled one at a time until completion, then it is guaranteed that the internal state is not accessible from two different threads at once and therefore there is no need to protect it using synchronization utilities (such as locks). Please make sure you understand these last few sentences, they are the core (and the reasoning) of this model.

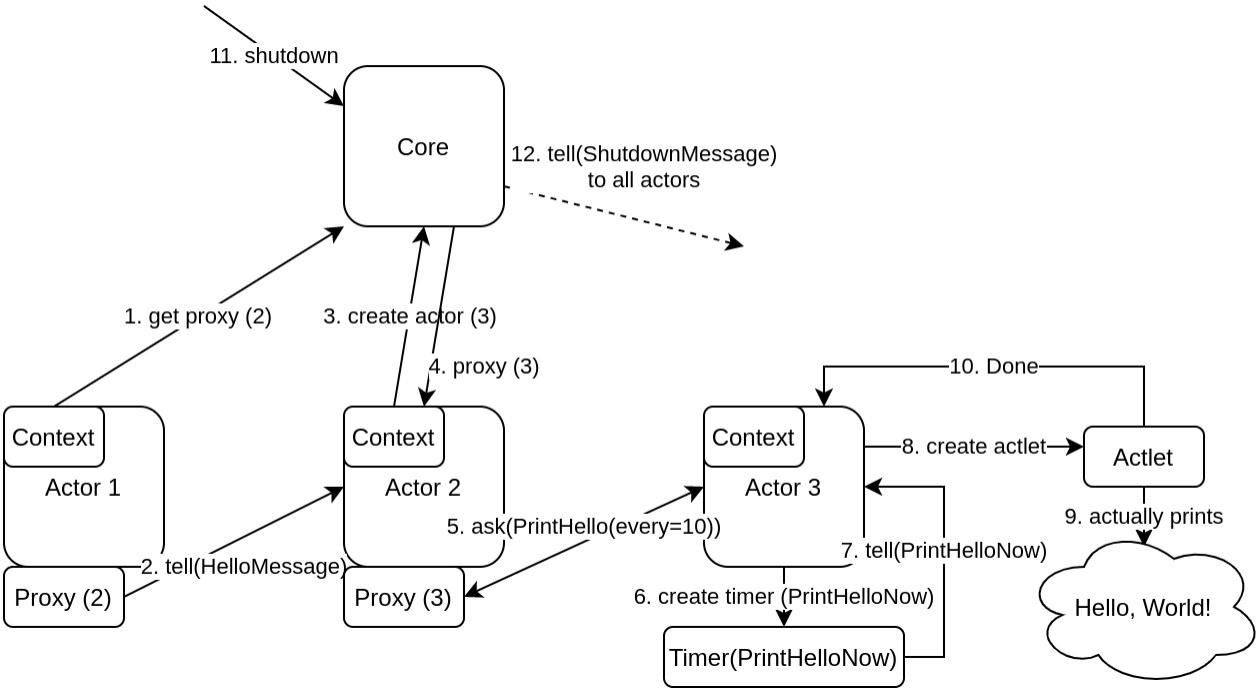
Why the actor model?

- Allows building responsive applications.
- Allows separation of concerns.
- Allows focusing on business logic without the need for re-validation of all flows.
    - On lock-based systems, each new flow demands placing locks in the correct locations and make sure mutual exclusion is preserved.

Open-source alternatives

- We examined the available open-source actor and event-based libraries (actor frameworks):
    - didn't find any framework that met our requirements.
        - asyncio based - threaded model might not be suitable for several dozen actors.
        - actively maintained.
    - however, as part of the examination we were exposed to several features which we have implemented as well (such as proxies, decorators...).

Tell it to me like a story



In the beginning, there was nothing but a *core* and its *context*. The core created some actors whether it was invoked directly (*core.create_actor*) or via a created actor's context (*context.create_actor*). When an actor is created the creation result is not the actor itself but a *proxy,* this proxy wraps the actor and exports functions for communicating with it. Each actor may hold several proxies (each one wraps exactly one actor) and it can use them to send messages asynchronously (*tell*) or synchronized request-response messages (*ask*). Each message is a serializable object (*mashumaro. DataClassJSONMixin*) that is wrapped in an *Envelope* (just like in the real world, the letter is composed of a message and an *envelope*). Messages can be sent from one actor to another, from the core to an actor, from the actor to itself, and from *actlet* to its actor. This system lived happily ever after (until the *co re* decides to shut down all its actors by sending them *ShutdownMessage*).

Components

Core

TODO: Core structure

Responsible of:

- actors lifetime
    - initialization
    - termination
- actors repository
    - lookup

Actors Lifetime

The core can create new actors.

1. The core first creates the actor object and then starts his internal engine.
2. When shutting down the core, the core sends *ShutdownMessage* messages to all created actors. Then it awaits until all actors are shut down completely.

Actors Repository

The core maintains all created actors in a map (name actor object), this allows an external object to ask the core for the communication details (*proxy*) of an actor.

Message & Envelopes

Serializable dataclasses object (mashumaro, json-serializable), **not protobuf!**
Any custom message must inherit from `libactors.Message`, this make sure that the declared message is json serializable.
Declaring a new message is simple as:

Declaring New Message

```
@dataclasses.dataclass(frozen=True)
class MyMessage(libactors.Message):
        name: str
        content: bytes
```

Notice that we freeze the message (`frozen=True`) since it does not make sense to modify the message once it was created.
Each sent message in the system is wrapped automatically by an envelope, which add additional metadata to it. Each envelope contains:

1. id (unique id)
2. sender id
3. receiver id
4. the message itself

Router

The messages router, it is used to registered function handlers for different messages.
When an actor receives a message it uses its *router* to fetch the correct function handler of this message.

The handlers must be:

1. Coroutine (async function)
2. Exact signature:

    ```
    async def handler(self, context, message):
        pass
    ```

The Message must:

1. not already registered in the same router
2. must inherit from *libactors.Message*

Actor

Actor consists of:

- unique id
- message queue
- message handlers
- actlets

- context

The framework does not assign any state to the actor, it is the actor's responsibility to maintain its state.
Code is better than words:

```
import dataclasses
import libactors

@dataclass.dataclass(frozen=True)
class MyMessage(libactors.Message):
        pass

class MyActor(libactors.Actor):

        async def initialize(self, context):
                print('initialized')

        @libactors.actor.register_handler(MyMessage)
        async def on_my_message(self, context, message: MyMessage):
                print(f'handling {message}')
```

A simple example for actor creation with a custom *initialize* function and a custom handler for *MyMessage* messages.
Notice that the reason there is an *initialize* function is because *__init__* functions cannot be async and it is very likely that we would need to initialize the actor using coroutines.

Context

A special object that supplies:

- logging and binding

  ```
  import dataclasses
  import libactors

  @dataclass.dataclass(frozen=True)
  class MyMessage(libactors.Message):
          pass

  class MyActor(libactors.Actor):

          async def initialize(self, context):
                  context.info('initialized')
                  # {'level': 'info', 'event': 'initialized'}
                  with context.bind(foo='bar'):
                          context.info('look now we got foo tag')
                          # {'level': 'info', 'event': 'look now we got foo tag', 'foo': 'bar'}
                  context.info('look now we do not')
                  # {'level': 'info', 'event': 'look now we do not'}

          @libactors.bind(foo='bar')
          async def on_my_message(self, context, message: MyMessage):
                  context.info(f'handling {message.__class__.__name__}')
                  # {'level': 'info', 'event': 'handling MyMessage', 'foo': 'bar'}
  ```

- core access
  - actor creation
  - proxy
- application should inherit and add more functionality:
  - e.g. external services API

```
import libactors

class MyContext(libactors.Context):
        def foo(self):
                return 'bar'

libactors.get_core().set_context(MyContext(...))
# Now any created actor will have MyContext instance and therefore can invoke foo method.
```

Proxy

A Proxy is the interface with which actors send messages to each other. The proxy also provides other utilities to control the life cycle of the underlying actor.

Actlets

Actlets are a slim version of the Actors, they can only handle a single message (once) and they do not require any boilerplate code.

Actlets run in their own "libactors context", meaning they have no access to the owning Actor and must receive all relevant information at their creation.

Once they finish running they must return a message (which the owning Actor can handle) with their result.

Timers

Timers can be used to send a message to self every interval. Timers are implemented using Actlets.