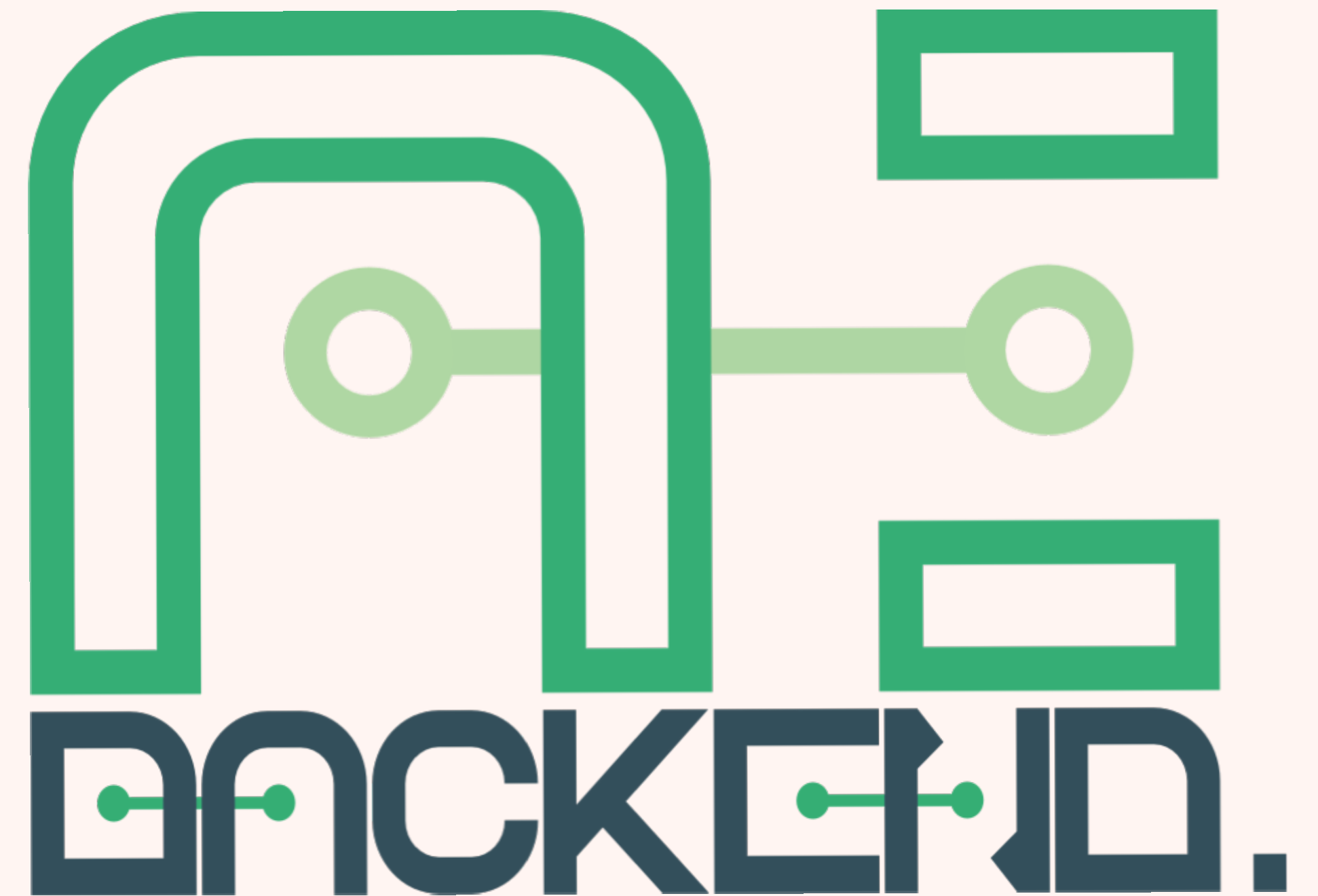

BACKEND.AI CODE BASE SEMINAR



VERSION INFO

- Backend.AI manager: 21.9.0.dev0
 - Backend.AI agent: 21.9.0.dev0
 - Backend.AI common: 21.9.0.dev0
 - Backend.AI client-py(Python SDK): 21.9.0.dev0
 - Backend.AI storage proxy: 21.9.0.dev0
-

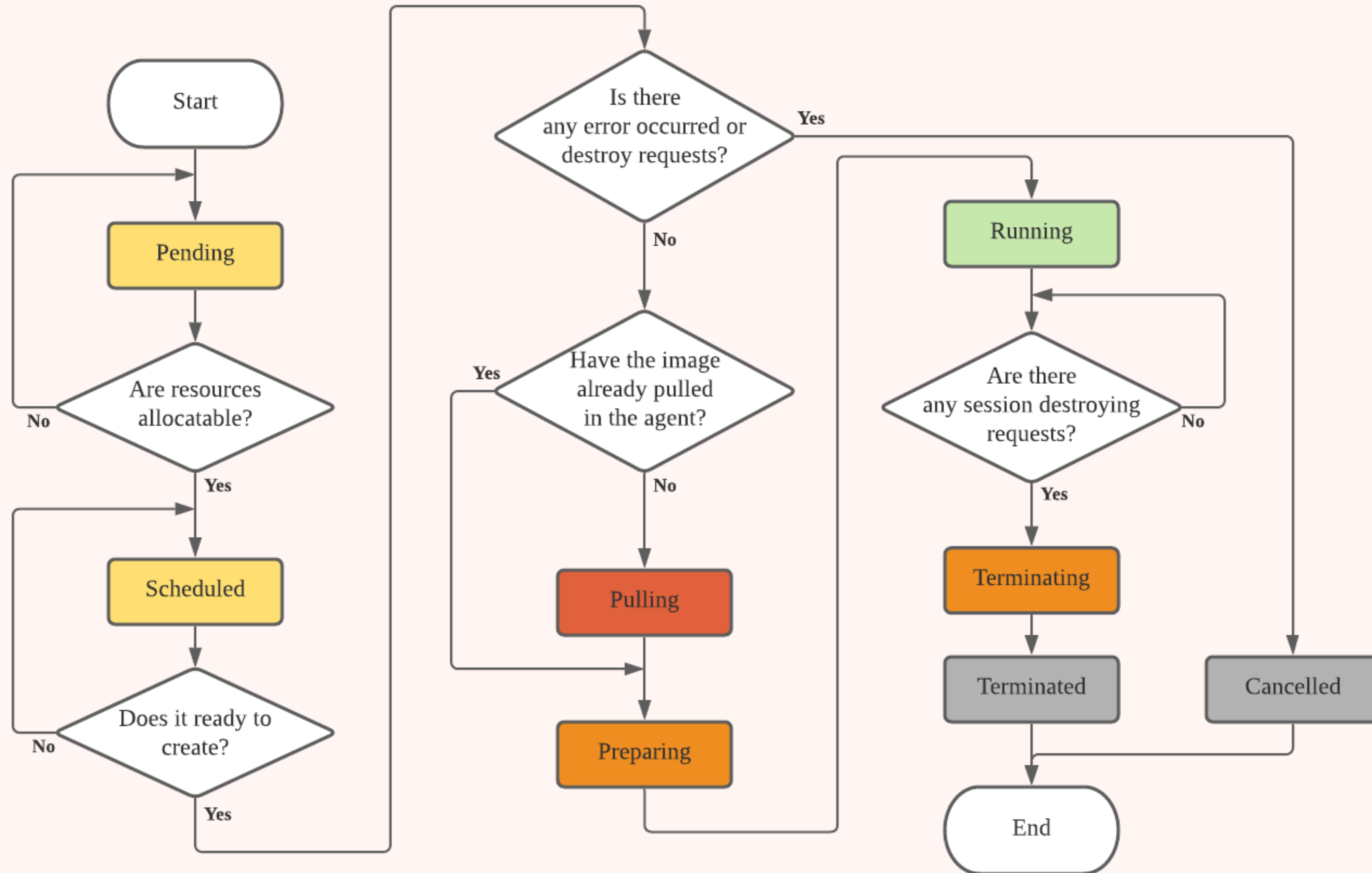
OVERALL ARCHITECTURE



KERNEL STATUS

- PENDING
 - CANCELLED
 - SCHEDULED
 - PULLING
 - PREPARING
 - RUNNING
 - TERMINATING
 - TERMINATED
-

KERNEL STATUS FLOW

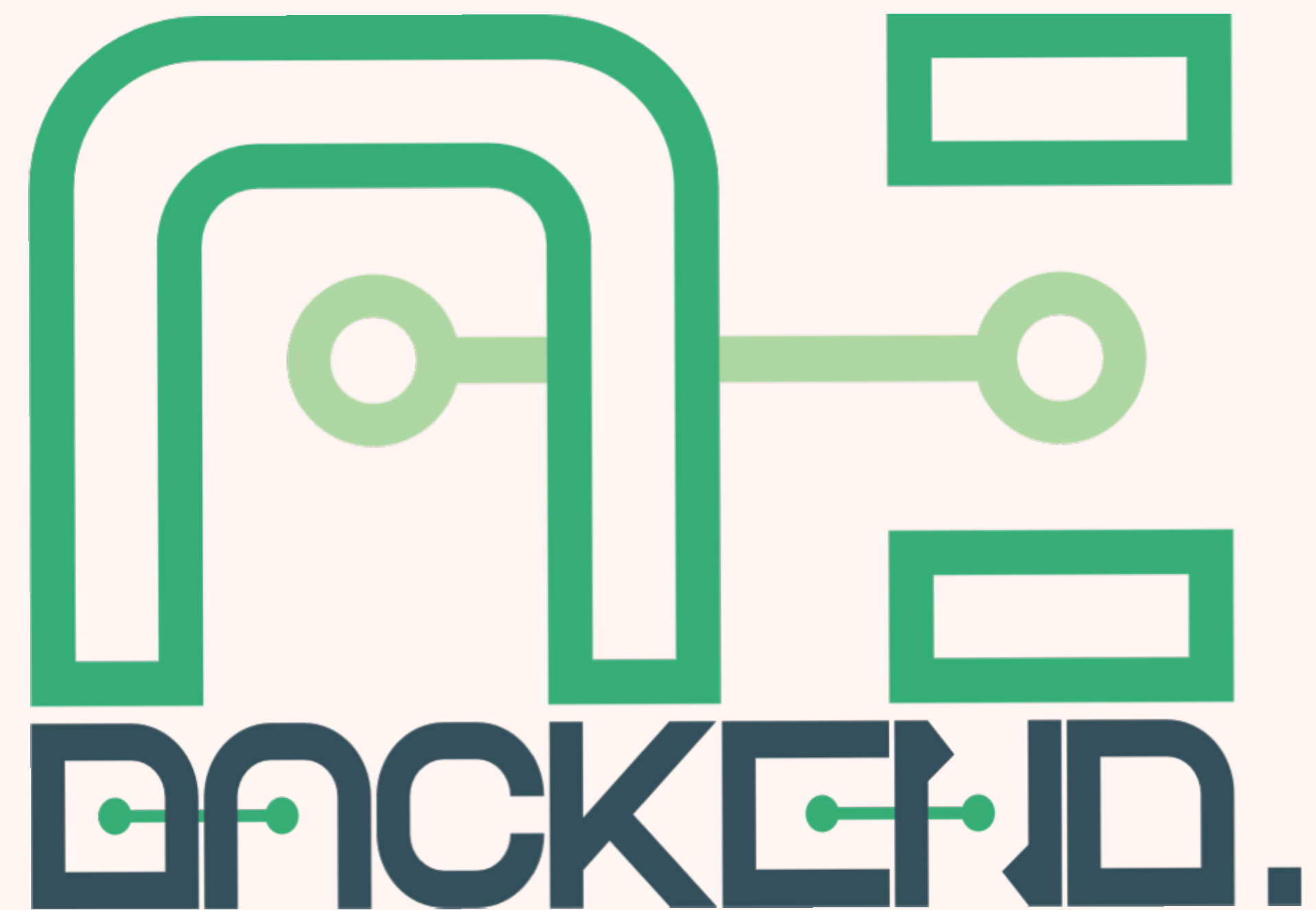


“SESSION DESTRUCTION” 은 대체 무슨 의미인가?

- 세션 정보는 세션 종료가 완료된 후에도 남아있습니다.
- 이는 세션에 대응되는 커널의 상태 값이 "CANCELLED" 또는 "TERMINATED"로 변경됨을 의미합니다.
- 세션 종료에는 세가지 과정이 있습니다:
 1. 세션 종료 준비하기
 2. 세션 종료하기
 3. 처리 결과 보내기

HOW DOES IT WORKS?

1. 세션 종료 준비하기



"SESSION DESTRUCTION" 요청하기

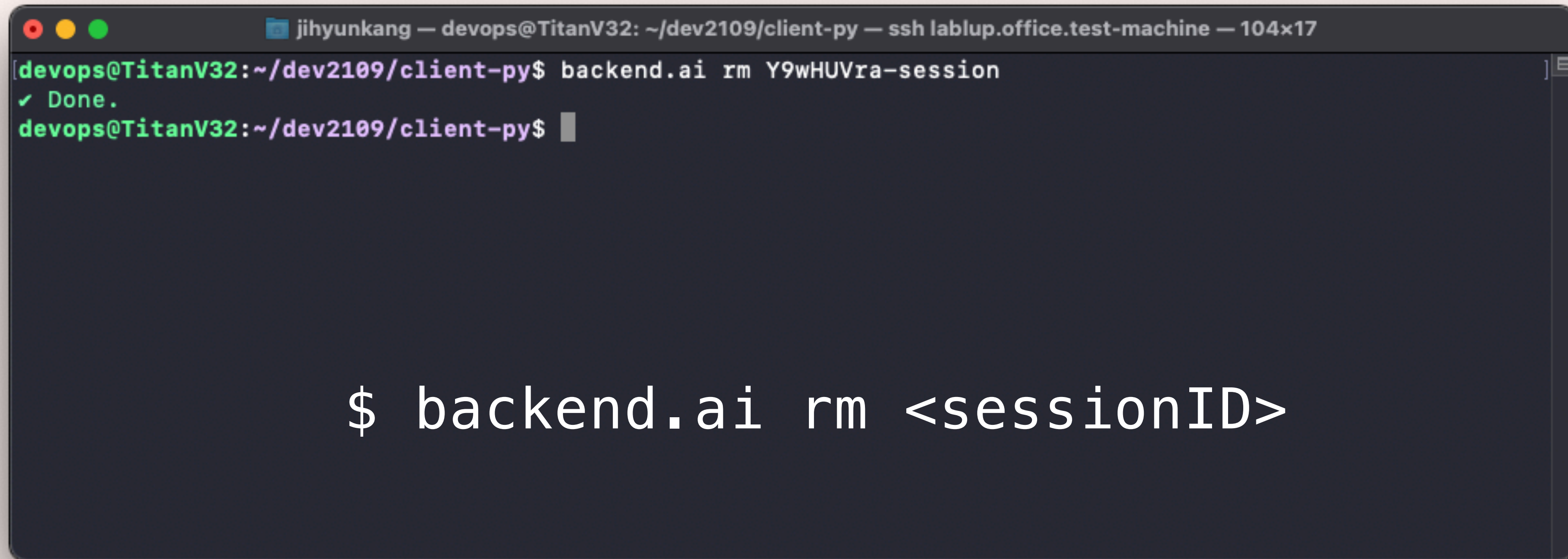
➤ WebUI (Javascript SDK): `destroy()` `../src/lib/backend.ai-client-node.ts`

| | # | User ID | SessionInfo | Status | Control |
|-------------------------------------|---|------------------|---|---------|---------|
| <input checked="" type="checkbox"/> | 1 | admin@lablup.com | Y9wHUVra-session Python 3.8 UBUNTU18.04 | RUNNING | >_ |

parameter: sessionId {string}, ownerKey {string | null} , forced {boolean}

“SESSION DESTRUCTION” 요청하기

- Client-py(Python SDK): `terminate()` `../cli/run.py`

A terminal window with a dark background and light text. The title bar shows 'jihyunkang — devops@TitanV32: ~/dev2109/client-py — ssh lablup.office.test-machine — 104x17'. The terminal content shows a command being executed: 'backend.ai rm Y9wHUVra-session', followed by a confirmation message '✓ Done.' and a new prompt 'devops@TitanV32:~/dev2109/client-py\$'.

```
jihyunkang — devops@TitanV32: ~/dev2109/client-py — ssh lablup.office.test-machine — 104x17
devops@TitanV32:~/dev2109/client-py$ backend.ai rm Y9wHUVra-session
✓ Done.
devops@TitanV32:~/dev2109/client-py$
```

`$ backend.ai rm <sessionID>`

Argument: `sessionId {string}`, `-f` or `--forced` argument option as forced

“SESSION DESTRUCTION” 요청하기

- `manager/api/session.py` 모듈*
- `manager/api/session_template.py` 모듈

`session` 모듈의 `destroy()` 보다 사용 빈도가 낮음.

템플릿의 레코드 기반으로 세션을 종료함.

세션 종료는 SQL Update 명령어로 완료됨

따라서 여기서 다루지 않겠음.

➤ `manager/api` 의 `session` 모듈에 있는 "DELETE" REST API function을 호출함.

```
def create_app(default_cors_options: CORSOptions) -> Tuple[web.Application, Iterable[WebMiddleware]]:
    ... app = web.Application()
    ... app.on_startup.append(init)
    ... app.on_shutdown.append(shutdown)
    ... app['api_versions'] = (1, 2, 3, 4)
    ... app['session.context'] = PrivateContext()
    ... cors = aiohttp_cors.setup(app, defaults=default_cors_options)
    ... cors.add(app.router.add_route('POST', '', create_from_params))
    ... cors.add(app.router.add_route('POST', '/_/create', create_from_params))
    ... cors.add(app.router.add_route('POST', '/_/create-from-template', create_from_template))
    ... cors.add(app.router.add_route('POST', '/_/create-cluster', create_cluster))
    ... cors.add(app.router.add_route('GET', '/_/match', match_sessions))
    ... session_resource = cors.add(app.router.add_resource(r'/{session_name}'))
    ... cors.add(session_resource.add_route('GET', get_info))
    ... cors.add(session_resource.add_route('PATCH', restart))
    ... cors.add(session_resource.add_route('DELETE', destroy))
```


- 종료 요청은 'forced' 와 사용자의 'role'에 따라 거부될 수 있습니다.
- registry 모듈의 `destroy_session()` 을 호출합니다.

```
@server_status_required(READ_ALLOWED)
@auth_required
@check_api_params(
    ...t.Dict({
    ...    ...t.Key('forced', default='false'): t.ToBool(),
    ...    ...t.Key('owner_access_key', default=None): t.Null | t.String,
    ...}))
async def destroy(request: web.Request, params: Any) -> web.Response:
    ...root_ctx: RootContext = request.app['_root.context']
    ...session_name = request.match_info['session_name']
    ...if params['forced'] and request['user']['role'] not in (UserRole.ADMIN, UserRole.SUPERADMIN):
    ...    ...raise InsufficientPrivilege('You are not allowed to force-terminate')
    ...requester_access_key, owner_access_key = await get_access_key_scopes(request)
    ...log.info('DESTROY (ak:{0}/{1}, s:{2}, forced:{3})',
    ...    ...requester_access_key, owner_access_key, session_name, params['forced'])
    ...last_stat = await root_ctx.registry.destroy_session(
    ...    ...functools.partial(
    ...        ...root_ctx.registry.get_session,
    ...        ...session_name, owner_access_key,
    ...        ...# domain_name=domain_name,
    ...    ...),
    ...    ...forced=params['forced'],
    ...    ...)
    ...resp = {
    ...    ...'stats': last_stat,
    ...    ...}
    ...return web.json_response(resp, status=200)
```

- `handle_kernel_exception()` 감싸져 있는데, 이는 kernel로 부터 오는 모든 에러를 핸들링하기 위함입니다.
- 'reason' 파라미터는 다음과 같은 값들을 갖습니다:
 - user-requested* (default)
 - force-terminated
 - self-terminated
 - ...
- 세션의 상태는 종료할 수 있는지, 아닌지를 결정합니다.
- 세션을 종료할 수 없을 경우, exception이 발생합니다.

```
... async def destroy_session(  
...     self,  
...     session_getter: SessionGetter,  
...     *,  
...     forced: bool = False,  
...     reason: str = 'user-requested',  
... ) -> Mapping[str, Any]:  
...     """  
...     Destroy session kernels. Do not destroy  
...     PREPARING/TERMINATING/ERROR and PULLING sessions.  
...  
...     :param forced: If True, destroy PREPARING/TERMINATING/ERROR session.  
...     However, PULLING session still cannot be destroyed.  
...     :param reason: Reason to destroy a session if client wants to specify it manually.  
...     """  
...     async with self.db.begin_readonly() as conn:  
...         session = await session_getter(db_connection=conn)  
...         if forced:  
...             reason = 'force-terminated'  
...         hook_result = await self.hook_plugin_ctx.dispatch(  
...             'PRE_DESTROY_SESSION',  
...             (session['session_id'], session['session_name'], session['access_key']),  
...             return_when=ALL_COMPLETED,  
...         )  
...         if hook_result.status != PASSED:  
...             raise RejectedByHook.from_hook_result(hook_result)  
...  
...     async with self.handle_kernel_exception(  
...         'destroy_session', session['id'], session['access_key'], set_error=True,  
...     ):

```


kernel 상태가 **PENDING** 일 때:

- 'status' 값을 CANCELLED 로 변경
- 'status_info' 값을 parameter 'reason' 으로 변경
- 'status_changed' 값을 변경하고, 현재 시간 값을 'terminated' 값에 적용
- **KernelCancelledEvent** 이벤트 생성
- 세션이 multi-container 모드로 생성되었을 경우:
하나의 메인 커널과 다른 커널들로 구성이 되었을 경우를 뜻한다. 이 때 메인 커널이 PENDING 상태에
서 있는 상태에서 세션 종료 요청을 받았을 때, 다른
커널들의 상태는 CANCELLED가 되며,
SessionCancelledEvent 이벤트를 생성함

```
.....for kernel in grouped_kernels:
.....    if kernel['status'] == KernelStatus.PENDING:

.....        async def _update() -> None:
.....            async with self.db.begin() as conn:
.....                await conn.execute(
.....                    sa.update(kernels)
.....                    .values({
.....                        'status': KernelStatus.CANCELLED,
.....                        'status_info': reason,
.....                        'status_changed': now,
.....                        'terminated_at': now,
.....                    })
.....                    .where(kernels.c.id == kernel['id']))
.....                )

.....        await execute_with_retry(_update)
.....        await self.event_producer.produce_event(
.....            KernelCancelledEvent(kernel['id'], '', reason)
.....        )

.....        if kernel['cluster_role'] == DEFAULT_ROLE:
.....            main_stat = {'status': 'cancelled'}
.....            await self.event_producer.produce_event(
.....                SessionCancelledEvent(
.....                    kernel['session_id'],
.....                    kernel['session_creation_id'],
.....                    reason,
.....                )
.....            )
.....        )
```

kernel 상태가 **PENDING** 일 때:

- PULLING 프로세스가 끝날 때 까지 종료될 수 없
- `GenericForbidden` exception 을 응답으로 받게 됨.

```
...elif kernel['status'] == KernelStatus.PULLING:  
    ...raise GenericForbidden('Cannot destroy kernels in pulling status')
```

📌 더 알아보기:

왜 kernel이 **PULLING** 상태일 때 종료할 수 없나요?

agent 내부에서 실행되는 docker daemon에 많은 부하를 주어 실행되던 container에 악영향을 끼칠 수 있기 때문입니다. 실제 컨테이너는 agent에 생성되어 있기 때문에, manager는 커널 생성 전 agent의 메타데이터를 확인해서 사용할 이미지가 pulled(downloaded) 되어 있는지, 아닌지 확인합니다. pulled 되어있지 않을 경우 manager는 먼저 RPC call을 요청하여 이미지를 다운로드 받도록 요청합니다. 이 작업은 agent 내부에서 실행되는 docker에서 이미지를 pulling 할 수 있게 합니다. 다운로드 과정에서, 사용자가 agent에 직접 접근하여 docker kill 명령어 또는 pulling 프로세스를 halt 시킬 수 있을 경우, pulling 상태일 때 종료가 가능합니다.

하지만, 이는 부수효과를 일으키게 되는데, 특히 이런 halting(killing) 작업은 매우 무거우며 (많은 오버헤드가 수반됨), 이는 docker 데몬 자체에 영향을 줄 수 있습니다. 특히 agent에서 실행중인 다른 docker container가 응답이 없는 상태로 만들 수도 있습니다.

따라서, Backend.AI 의 정책상 PULLING 상태에 있는 kernel은 종료할 수 없습니다.

kernel 상태가 SCHEDULED, PREPARING, TERMINATING, ERROR일 때:

- forced 값이 false일 때는 종료할 수 없습니다. 이 경우, **GenericForbidden** exception 를 보냅니다.
- forced 값이 true 일 때, kernel의 **container_id** 값을 확인하고 **destroyed_kernels** (agent 내 삭제할 실제 컨테이너들을 담은 목록) 또한, kernel 테이블의 레코드의 status 값을 **TERMINATED** 로 바꾸고, keypairs 테이블의 **concurrency_used** 값에서 1을 빼고, **KernelTerminatedEvent** 를 생성합니다.

```
... elif kernel['status'] in (
...     KernelStatus.SCHEDULED,
...     KernelStatus.PREPARING,
...     KernelStatus.TERMINATING,
...     KernelStatus.ERROR,
... ):
...     if not forced:
...         raise GenericForbidden(
...             'Cannot destroy kernels in scheduled/preparing/terminating/error status'
...         )
...     log.warning('force-terminating kernel (k:{}, status:{})',
...                 kernel['id'], kernel['status'])
...     if kernel['container_id'] is not None:
...         destroyed_kernels.append(kernel)
...
...     async def _update() -> None:
...         async with self.db.begin() as conn:
...             if kernel['cluster_role'] == DEFAULT_ROLE:
...                 # The main session is terminated;
...                 # decrement the user's concurrency counter
...                 await conn.execute(
...                     sa.update(keypairs)
...                     .values({
...                         'concurrency_used': keypairs.c.concurrency_used - 1,
...                     })
...                     .where(keypairs.c.access_key == kernel['access_key'])
...                 )
...             await conn.execute(
...                 sa.update(kernels)
...                 .values({
...                     'status': KernelStatus.TERMINATED,
...                     'status_info': reason,
...                 })
...             )
```


kernel 상태가 다른 상태 (RUNNING)일 때:

- keypairs 테이블의 **concurrency_used** 값에서 1을 뺍니다.
- kernel 테이블의 레코드의 status 값을 TERMINATED 로 바꿉니다.
- **KernelTerminatingEvent**를 생성합니다.

```
..else:

    ..async def _update() -> None:
    ..    async with self.db.begin() as conn:
    ..        if kernel['cluster_role'] == DEFAULT_ROLE:
    ..            # The main session is terminated;
    ..            # decrement the user's concurrency counter
    ..            await conn.execute(
    ..                sa.update(keypairs)
    ..                .values({
    ..                    'concurrency_used': keypairs.c.concurrency_used - 1,
    ..                })
    ..                .where(keypairs.c.access_key == kernel['access_key'])
    ..            )

    ..        await conn.execute(
    ..            sa.update(kernels)
    ..            .values({
    ..                'status': KernelStatus.TERMINATING,
    ..                'status_info': reason,
    ..                'status_data': {
    ..                    "kernel": {"exit_code": None},
    ..                    "session": {"status": "terminating"},
    ..                }
    ..            })
    ..            .where(kernels.c.id == kernel['id'])
    ..        )

    ..    await execute_with_retry(_update)
    ..    await self.event_producer.produce_event(
    ..        KernelTerminatingEvent(kernel['id'], reason),
    ..    )
```

kernel 객체의 `agent_addr` 값이 비어있을 경우,
TERMINATED 상태인 것으로 간주해
`destroyed_kernels` 에 추가됩니다.

```
if kernel['agent_addr'] is None:
    ... await self.mark_kernel_terminated(kernel['id'], 'missing-agent-allocation')
    ... if kernel['cluster_role'] == DEFAULT_ROLE:
    ...     ... main_stat = {'status': 'terminated'}
else:
    ... destroyed_kernels.append(kernel)
```


KernelCancelledEvent, KernelTerminatedEvent,
그리고 KernelTerminatingEvent 는 콜백 함수
`enqueue_kernel_termination_status_update()` 에서 처리됩니다.

```
async def events_app_ctx(app: web.Application) -> AsyncIterator[None]:
    ... root_ctx: RootContext = app['_root.context']
    ... app_ctx: PrivateContext = app['events.context']
    ... app_ctx.session_event_queues = set()
    ... app_ctx.task_update_queues = set()
    ... event_dispatcher: EventDispatcher = root_ctx.event_dispatcher
    ... event_dispatcher.subscribe(SessionEnqueuedEvent, app, enqueue_session_creation_status_update)
    ... event_dispatcher.subscribe(SessionScheduledEvent, app, enqueue_session_creation_status_update)
    ... event_dispatcher.subscribe(KernelPreparingEvent, app, enqueue_kernel_creation_status_update)
    ... event_dispatcher.subscribe(KernelPullingEvent, app, enqueue_kernel_creation_status_update)
    ... event_dispatcher.subscribe(KernelCreatingEvent, app, enqueue_kernel_creation_status_update)
    ... event_dispatcher.subscribe(KernelStartedEvent, app, enqueue_kernel_creation_status_update)
    ... event_dispatcher.subscribe(SessionStartedEvent, app, enqueue_session_creation_status_update)
    ... event_dispatcher.subscribe(KernelTerminatingEvent, app, enqueue_kernel_termination_status_update)
    ... event_dispatcher.subscribe(KernelTerminatedEvent, app, enqueue_kernel_termination_status_update)
    ... event_dispatcher.subscribe(KernelCancelledEvent, app, enqueue_kernel_termination_status_update)
    ... event_dispatcher.subscribe(SessionTerminatedEvent, app, enqueue_session_termination_status_update)
    ... event_dispatcher.subscribe(SessionCancelledEvent, app, enqueue_session_creation_status_update)
    ... event_dispatcher.subscribe(SessionSuccessEvent, app, enqueue_batch_task_result_update)
    ... event_dispatcher.subscribe(SessionFailureEvent, app, enqueue_batch_task_result_update)
    ... event_dispatcher.subscribe(BgtaskUpdatedEvent, app, enqueue_bgtask_status_update)
    ... event_dispatcher.subscribe(BgtaskDoneEvent, app, enqueue_bgtask_status_update)
    ... event_dispatcher.subscribe(BgtaskCancelledEvent, app, enqueue_bgtask_status_update)
    ... event_dispatcher.subscribe(BgtaskFailedEvent, app, enqueue_bgtask_status_update)

    ... yield
```

`enqueue_kernel_termination_status_update()`를 보면 kernel 정보, 더 나아가 클라이언트 쪽에서 "세션"이라고 부르는 정보를 업데이트 하기 위해 `session_event_queues` 라고 부르는 이벤트 큐에 kernel 정보를 추가하는 함수임을 알 수 있습니다.

```
async def enqueue_kernel_termination_status_update(
    app: web.Application,
    agent_id: AgentId,
    event: KernelCancelledEvent | KernelTerminatingEvent | KernelTerminatedEvent,
) -> None:
    root_ctx: RootContext = app['_root.context']
    app_ctx: PrivateContext = app['events.context']

    async def _fetch():
        async with root_ctx.db.begin() as conn:
            query = (
                sa.select([
                    kernels.c.id,
                    kernels.c.session_id,
                    kernels.c.session_name,
                    kernels.c.access_key,
                    kernels.c.cluster_role,
                    kernels.c.cluster_idx,
                    kernels.c.domain_name,
                    kernels.c.group_id,
                    kernels.c.user_uuid,
                ])
                .select_from(kernels)
                .where(
                    (kernels.c.id == event.kernel_id)
                )
            )
            result = await conn.execute(query)
            return result.first()

    row = await asyncio.shield(_fetch())
    if row is None:
        return
    for q in app_ctx.session_event_queues:
        q.put_nowait((event.name, row._mapping, event.reason))
```

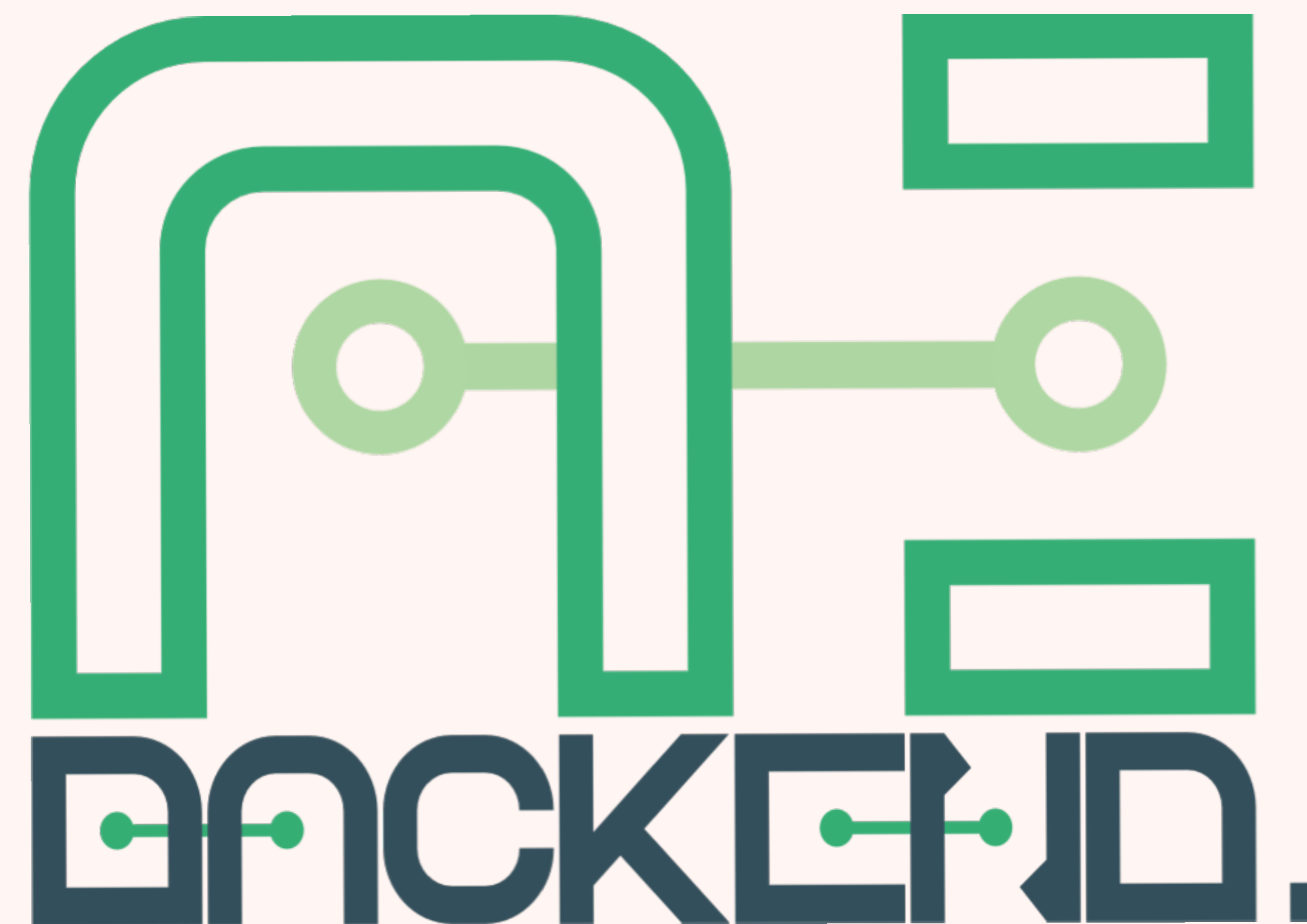

destroyed_kernel 함수에서는 중첩함수 `_destroy_kernels_in_agent()` 가 `destroy_kernel()` using `RPCContext` 객체를 통해 `destroy_kernel()` 를 호출합니다. 이 부분이 바로 kernel과 대응되는 컨테이너를 종료하는 실제 요청이라고 볼 수 있습니다.

```
.....else:
.....destroyed_kernels.append(kernel)

.....async def _destroy_kernels_in_agent(session, destroyed_kernels) -> None:
.....nonlocal main_stat
.....async with RPCContext(
.....destroyed_kernels[0]['agent'],
.....destroyed_kernels[0]['agent_addr'],
.....None,
.....order_key=session['session_id'],
.....) as rpc:
.....rpc_coros = []
.....for kernel in destroyed_kernels:
.....# internally it enqueues a "destroy" lifecycle event.
.....if kernel['status'] != KernelStatus.SCHEDULED:
.....rpc_coros.append(
.....rpc.call.destroy_kernel(str(kernel['id']), reason)
.....)
```

HOW DOES IT WORKS?

2. 세션 종료하기



agent/server.py 모듈에 있는 `destroy_kernel()` 는 kernel 종료를 위해 `inject_container_lifecycle_event()` 를 호출합니다.

```
@rpc_function
@collect_error
async def destroy_kernel(
    self,
    kernel_id: str,
    reason: str = None,
    suppress_events: bool = False,
):
    async with self._destroy_sema:
        log.info('rpc::destroy_kernel(k:{0})', kernel_id)
        done = asyncio.Event()
        await self.agent.inject_container_lifecycle_event(
            KernelId(UUID(kernel_id)),
            LifecycleEvent.DESTROY,
            reason or 'user-requested',
            done_event=done,
            suppress_events=suppress_events,
        )
        await done.wait()
    return getattr(done, '_result', None)
```


agent/agent.py 모듈의

`inject_container_lifecycle_event()` 은 check whether `container_id` 인자와 `kernel_object` 컨테이너 값이 동일한지 확인하고, 동일할 경우 모든 값을 `ContainerLifecycleEvent` 객체로 담아 `container_lifecycle_queue` 에 추가합니다.

```
async def inject_container_lifecycle_event(
    self,
    kernel_id: KernelId,
    event: LifecycleEvent,
    reason: str,
    *,
    container_id: ContainerId = None,
    exit_code: int = None,
    done_event: asyncio.Event = None,
    clean_event: asyncio.Event = None,
    suppress_events: bool = False,
) -> None:
    try:
        kernel_obj = self.kernel_registry[kernel_id]
        if kernel_obj.termination_reason:
            reason = kernel_obj.termination_reason
        if kernel_obj.clean_event is not None:
            # This should not happen!
            log.warning('overwriting kernel_obj.clean_event (k:{})', kernel_id)
            kernel_obj.clean_event = clean_event
        if container_id is not None and container_id != kernel_obj['container_id']:
            # This should not happen!
            log.warning('container_id mismatch for kernel_obj (k:{}, c:{}) with event (c:{})',
                        kernel_id, kernel_obj['container_id'], container_id)
            container_id = kernel_obj['container_id']
        except KeyError:
            pass
        await self.container_lifecycle_queue.put(
            ContainerLifecycleEvent(
                kernel_id,
                container_id,
                event,
                reason,
                done_event,
                exit_code,
                suppress_events,
            )
        )
    )
```


`container_lifecycle_queue` 는 `agent/agent.py` 모듈의 `asyncio` 의 `Queue` 객체입니다. 모든 이벤트는 FIFO 알고리즘이 적용되고 있는 `process_lifecycle_events()` 에서 처리됩니다. 종료 작업은 이벤트 값이 `LifecycleEvent.DESTROY` 와 같을 때 계속 되는데, 이 때 `_handle_destroy_event()` 를 호출하게 됩니다.

```
class AbstractAgent(aobject, Generic[KernelObjectType, KernelCreationContextType], metaclass=ABCMeta):  
  
    loop: asyncio.AbstractEventLoop  
    local_config: Mapping[str, Any]  
    etcd: AsyncEtdc  
    local_instance_id: str  
    kernel_registry: MutableMapping[KernelId, AbstractKernel]  
    computers: MutableMapping[str, ComputerContext]  
    images: Mapping[str, str]  
    port_pool: Set[int]  
  
    redis: aioredis.Redis  
    zmq_ctx: zmq.asyncio.Context  
  
    restarting_kernels: MutableMapping[KernelId, RestartTracker]  
    terminating_kernels: Set[KernelId]  
    timer_tasks: MutableSequence[asyncio.Task]  
    container_lifecycle_queue: asyncio.Queue[ContainerLifecycleEvent | Sentinel]
```

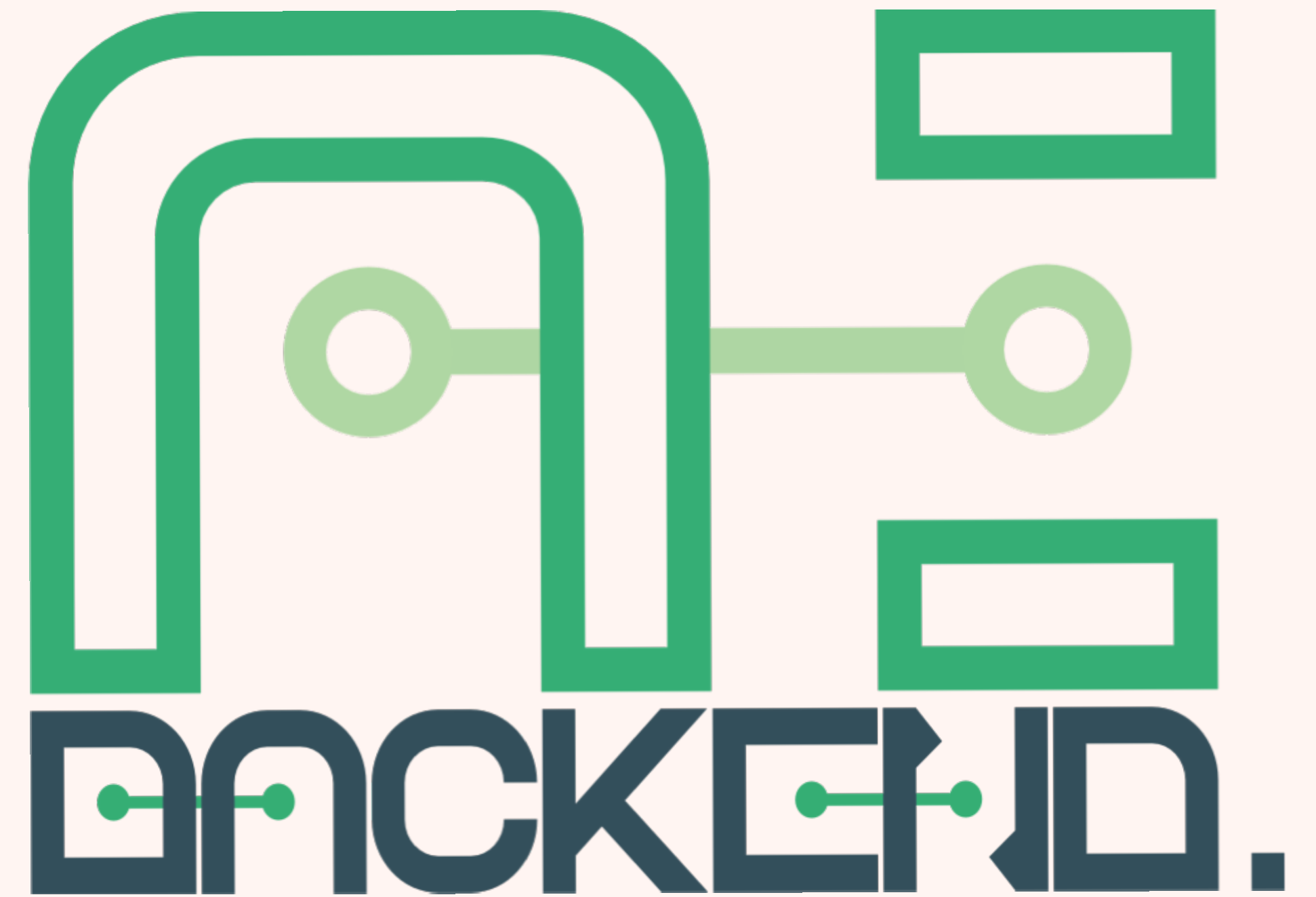
```
async def process_lifecycle_events(self) -> None:  
    async with aiotools.TaskGroup() as tg:  
        while True:  
            ev = await self.container_lifecycle_queue.get()  
            if isinstance(ev, Sentinel):  
                with open(ipc_base_path / f'last_registry.{self.local_instance_id}.dat', 'wb') as f:  
                    pickle.dump(self.kernel_registry, f)  
                return  
            # attr currently does not support customizing getstate/setstate dunder methods  
            # until the next release.  
            if self.local_config['debug']['log-events']:  
                log.info(f'lifecycle event: {ev!r}')  
            try:  
                if ev.event == LifecycleEvent.START:  
                    tg.create_task(self._handle_start_event(ev))  
                elif ev.event == LifecycleEvent.DESTROY:  
                    tg.create_task(self._handle_destroy_event(ev))  
                elif ev.event == LifecycleEvent.CLEAN:  
                    tg.create_task(self._handle_clean_event(ev))  
            else:  
                log.warning('unsupported lifecycle event: {!r}', ev)
```

`_handle_destroy_event()`에서는 `agent/docker/agent.py` 모듈의 `destroy_kernel()`을 호출하게 되는데, 여기서 마침내 컨테이너를 삭제하게 됩니다. 단, `container_id` 값이 빈 값이 아닐 때만 삭제가 됩니다.

```
async def destroy_kernel(
    self,
    kernel_id: KernelId,
    container_id: Optional[ContainerId],
) -> None:
    if container_id is None:
        return
    try:
        container = self.docker.containers.container(container_id)
        # The default timeout of the docker stop API is 10 seconds
        # to kill if container does not self-terminate.
        await container.stop()
    except DockerError as e:
        if e.status == 409 and 'is not running' in e.message:
            # already dead
            log.warning('destroy_kernel(k:{0}) already dead', kernel_id)
            await self.rescan_resource_usage()
        elif e.status == 404:
            # missing
            log.warning('destroy_kernel(k:{0}) kernel missing, '
                        'forgetting this kernel', kernel_id)
            await self.rescan_resource_usage()
        else:
            log.exception('destroy_kernel(k:{0}) kill error', kernel_id)
            self.error_monitor.capture_exception()
```

HOW DOES IT WORKS?

3. 처리 결과 보내기



실제 컨테이너를 종료하게되면, kernel의 마지막 정보를 담고있는 redis 서버의 `last_stat` 값을 `main_stat` 이라는 값으로 받게 되는데, 이 때 데이터 형식은 JSON 으로 받습니다.

redis 서버에서 Since there's some chance to get empty value of `last_stat` 에 빈 값을 보내줄 수 있기 때문에, 설정파일에 나와있는 최대 재시도 횟수에 따라 값 요청을 하게 됩니다.

마침내, `main_stat` 이 goes into return value of of manager/registry.py 모듈의 `destroy_session()` 의 리턴 값으로 들어가게 되고, 순차적으로 and sequentially to in manager/api/session.py 모듈의 `destroy()` 에 리턴 값으로 들어가게 되어, 클라이언트(WebUI, Client SDK for Python, etc.) 에 요청 결과로서 전달되는 것으로 세션 종료 흐름이 마무리 됩니다.

```
.....try:
.....    await asyncio.gather(*rpc_coros)
.....except Exception:
.....    log.exception(
.....        "destroy_kernels_in_agent(a:{}, s:{}): unexpected error",
.....        destroyed_kernels[0]['agent'],
.....        session['session_id'],
.....    )
.....    for kernel in destroyed_kernels:
.....        last_stat: Optional[Dict[str, Any]]
.....        last_stat = None
.....        try:
.....            raw_last_stat = await redis.execute_with_retries(
.....                lambda: self.redis_stat.get(str(kernel['id']), encoding=None),
.....                max_retries=3)
.....            if raw_last_stat is not None:
.....                last_stat = msgpack.unpackb(raw_last_stat)
.....                last_stat['version'] += 2
.....            except asyncio.TimeoutError:
.....                pass
.....            if kernel['cluster_role'] == DEFAULT_ROLE:
.....                main_stat = {
.....                    **({last_stat if last_stat is not None else {}},
.....                    'status': 'terminated',
.....                )
.....            if destroyed_kernels:
.....                per_agent_tasks.append(_destroy_kernels_in_agent(session, destroyed_kernels))
.....            if per_agent_tasks:
.....                await asyncio.gather(*per_agent_tasks)
.....            await self.hook_plugin_ctx.notify(
.....                'POST_DESTROY_SESSION',
.....                (session['session_id'], session['session_name'], session['access_key']),
.....            )
.....            if forced:
.....                await self.recalc_resource_usage()
.....            return main_stat
```

THANK YOU '😊'🌸
