$p = Paragraph::load($element['target\_id']);

`

$element is an array; probably a form array fragment, i.e.: for a [form or render element](https://api.drupal.org/api/drupal/elements) - in this case it contains a target\_id [child](https://git.drupalcode.org/project/drupal/-/blob/9.3.x/core/lib/Drupal/Core/Render/Element.php#L43-54) (i.e.: not a [property](https://git.drupalcode.org/project/drupal/-/blob/9.3.x/core/lib/Drupal/Core/Render/Element.php#L17-28)) key. The code gets the value at $element['target\_id'] and passes it to Paragraph::load(), i.e.: the [Paragraph module](https://www.drupal.org/project/paragraphs)’s [Paragraph](https://git.drupalcode.org/project/paragraphs/-/blob/8.x-1.x/src/Entity/Paragraph.php) [entity](https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Entity%21entity.api.php/group/entity_api/9.2.x)’s load() function, inherited from [\Drupal\Core\Entity\EntityBase::load()](https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Entity%21EntityBase.php/function/EntityBase%3A%3Aload/9.2.x) - while I don’t know the restrictions offhand, most Drupal entity IDs are positive integers > 0 or non-empty string (i.e.: which correspond to a UNIQUE column in a table in the database)

If this doesn’t seem to be working, I’d start by checking:

1. Whether there is a use Drupal\paragraphs\Entity\Paragraph; line at the top of the file - if this is missing, then Paragraph isn’t resolving, or is resolving to something else (at which all bets are off)
2. Whether $element is an array
3. If it is an array, whether $element has a child named target\_id
4. If $element['target\_id']  does not exist, whether it was supposed to be a property instead, i.e.: $element['#target\_id']
5. If $element['target\_id']  (or $element['#target\_id'] ) exists, whether it contains an integer or string (i.e.: not NULL, a sub-array, an object, file handle; nor something like a bool, float, etc. that can be cast/interpreted to an integer or string)
6. If $element['target\_id']  is an integer or a string, perhaps the integer or string isn’t actually a paragraph ID - i.e.: perhaps it contains a node/user/term/role/etc. ID instead - all sorts of interesting, weird, inconsistent behaviour can occur if this is the case

Me: it is somehow working, was trying to realize how the scope resolution works here and what Paragraph::load() does

if $element['target\_id'] returns, say, 5, and you thereby call Paragraph::load(5) , it will:

1. Load the storage backend for the Paragraph entity (which is usually SQL, i.e.: PostgreSQL)
2. Ask the storage backend to load the Paragraph with ID 5
3. The storage backend either returns NULL (i.e.: “there is no paragraph with ID 5”) or an instance of a Drupal\paragraphs\Entity\Paragraph object containing the data for the paragraph with ID 5

… and either NULL or the Drupal\paragraphs\Entity\Paragraph will be put in the variable $p

In Drupal’s default configuration, the storage backend for content entities like Paragraphs is SQL, i.e.: PostgreSQL in our case

(although in theory, as long as have a storage backend that conforms to [Drupal’s interface](https://api.drupal.org/api/drupal/core%21lib%21Drupal%21Core%21Entity%21EntityStorageInterface.php/interface/EntityStorageInterface/9.2.x) and is registered as a storage backend for paragraphs in services.yml, your storage backend could be whatever you want (e.g.: [Redis](https://redis.io/), [Memcached](https://www.memcached.org/), [Elasticsearch](https://www.elastic.co/elasticsearch/), [Solr](https://solr.apache.org/" \t "_blank), [MongoDB](https://www.mongodb.com/), or something really esoteric like using the Google API to access emails in a Gmail account whose email subjects are Paragraph IDs and email bodies contain XML-formatted paragraph data) (edited)

[4:51](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635281511039600)

(I don’t recommend “gmail as a storage backend” though, as malicious users could add paragraphs to your site by sending you an email - although I suppose that would be mitigated by the fact that no nodes on your site would refer to the spam; although I suppose even gmail accounts have maximum storage limits, so they could initiate a denial-of-service attack by spamming you and filling your mailbox quota) (edited)

there is a similar thing hapenning which is

$term = Term::load($languageField[0]['target\_id']);

[4:55](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635281723043700)

so again I assume this is for load the storage backend for the Term entity if I am not mistaken

[**Matthew Parker**](https://app.slack.com/team/U02CL6JPS94)  [4:56 PM](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635281764044100)

Yup, exactly

[**Alireza Tayari**](https://app.slack.com/team/U01NYBLA95G)  [4:56 PM](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635281802045400)

maybe for accessing the language code from the taxonomy

[**Matthew Parker**](https://app.slack.com/team/U02CL6JPS94)  [4:57 PM](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635281846046400)

That being said, I think Paragraph::load() or Term::load() isn’t the best way of doing things in production code (totally okay in tests though)

Personally, I’d do something like \Drupal::getContainer()->get('entity\_type.manager')->getStorage('paragraph')->load($element['target\_id'])  - or the equivalent dependency-injected thing :syringe:

[5:00](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635282048050800)

1. \Drupal refers to the global kernel object, instantiated at the beginning of every request
2. ::getContainer() gets the service container, for dependency injection
3. ->get('entity\_type.manager') loads the entity type manager service
4. ->getStorage('paragraph') loads the storage backend for paragraphs
5. ->load(…) asks the storage backend for the given paragraph object

[5:02](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635282150053000)

Asking the service container for the entity type manager, and then asking the entity type manager for the storage engine is how you would give Drupal the opportunity to load from some other storage engine (like Redis, or your email, or whatever)

[5:03](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635282213054400)

That’s the essence of dependency injection: ask the service container for something instead, which lets the service container use the default, or a substitute you specify

[5:06](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635282365057500)

To use a real-world analogy, suppose:

1. you are vegan
2. you are friends with a barista
3. the barista knows you’re vegan

… if you ask the barista (the service container in this case) for a latte, they’ll automatically substitute oat milk, (or almond milk if they’ve run out) in the coffee without you having to specify exactly  
(i.e.: which could fail if they’re out of the thing you specifically ask for)

In procedural code, e.g.: a legacy Drupal hook, we’d use \Drupal::getContainer()->get('entity\_type.manager')->getStorage('paragraph')->load($element['target\_id']) directlyBut in a class, we can use Dependency Injection better …

<?php

​

use Drupal\Component\DependencyInjection\Container;

​

class MyCoolClass {

private $paragraphStorage;

public function \_\_construct(EntityTypeManager $etm) {

$this->paragraphStorage = $etm->getStorage('paragraph');

}

public static function create(Container $container) {

return new static($container->get('entity\_type.manager'));

}

public function doMyStuff($element) {

(line 17) $p = $this->paragraphStorage->load($element['target\_id']);

// imagine the cool stuff you can do now!

}

}

Your example line of code is on line 17: in that, we get a class property named $this->paragraphStorage and call ->load($element['target\_id']) on it$this->paragraphStorage  got populated in the class constructor on line 9The class constructor was called in the ::create() function on line 13The service container calls the ::create() function and passes a copy of itself when you want to create a MyCoolClass

[5:21](https://ontariodigital.slack.com/archives/D02CYJCR751/p1635283301068100)

Put another way, the line \Drupal::getContainer()->get('entity\_type.manager')->getStorage('paragraph')->load($element['target\_id']) got broken up into several parts…\Drupal::getContainer()  is done by the service container, and gets passed as $container to ::create() on line 12->get('entity\_type.manager')  is called on line 13 - the new static(…) bit calls the constructor on line 8, passing in the entity type managerthe constructor calls ->getStorage('paragraph')  and stores the result in $this->paragraphStorageyour code calls ->load($element['target\_id'])