WIN BIG CACHE OUT

By: Ashley Hutson

WHO AM I?

- DevOps guru
- Software Engineer
  - PHP(first and foremost)
  - JS/Node
  - GoLang
- Online Teacher

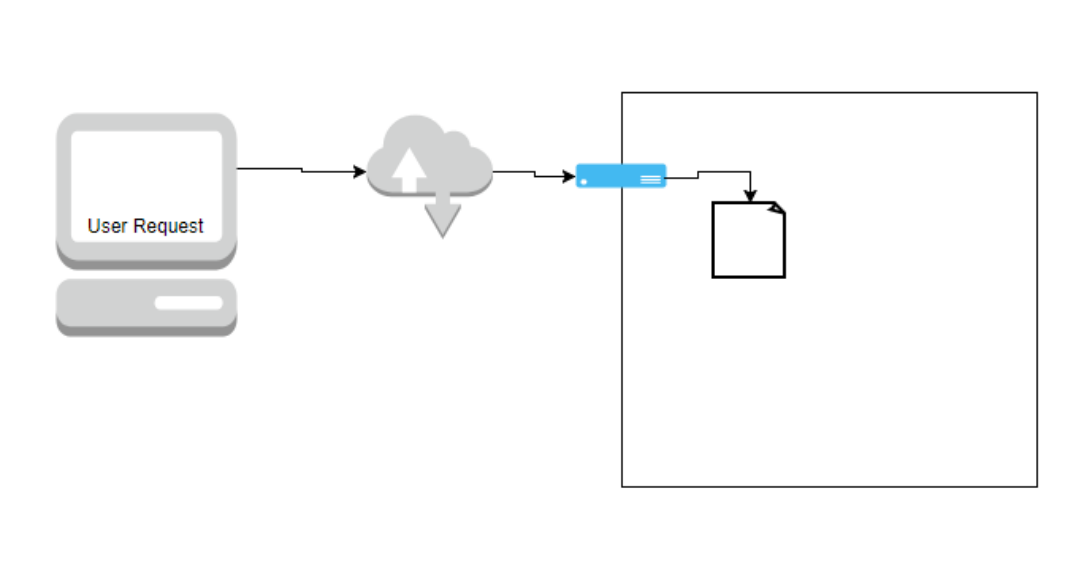
DIFFERENT TYPES OF CACHING

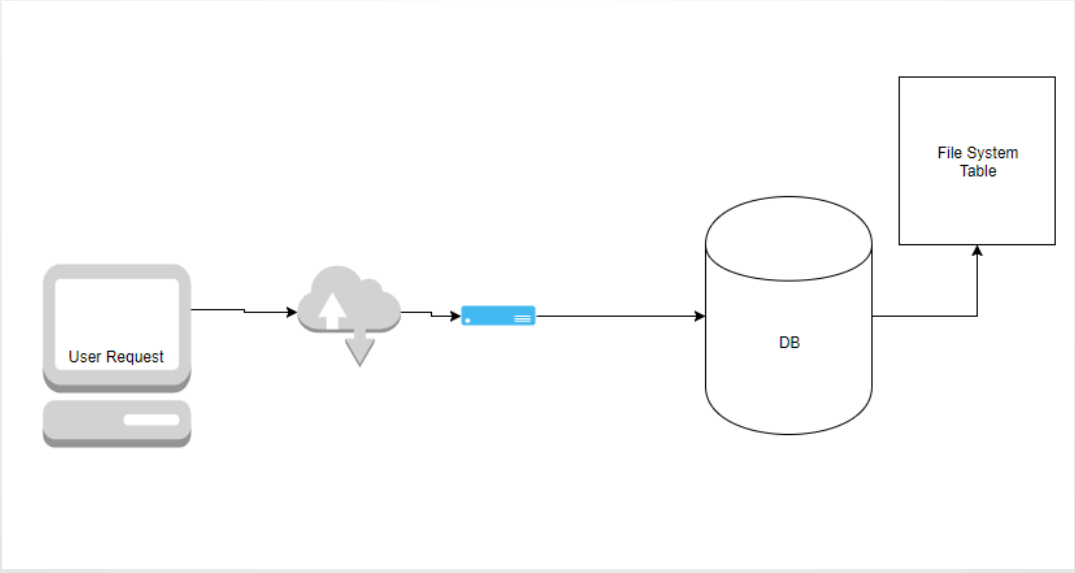
You have two basic options

File based (flat file, database(non memory))

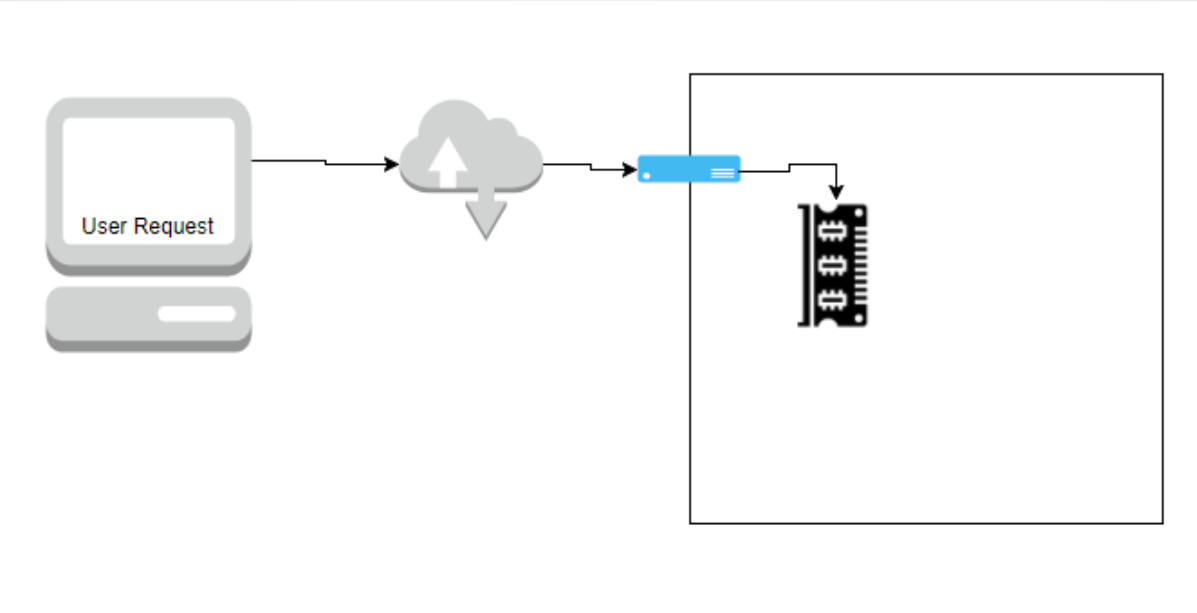
Memory based(redis, memcached)

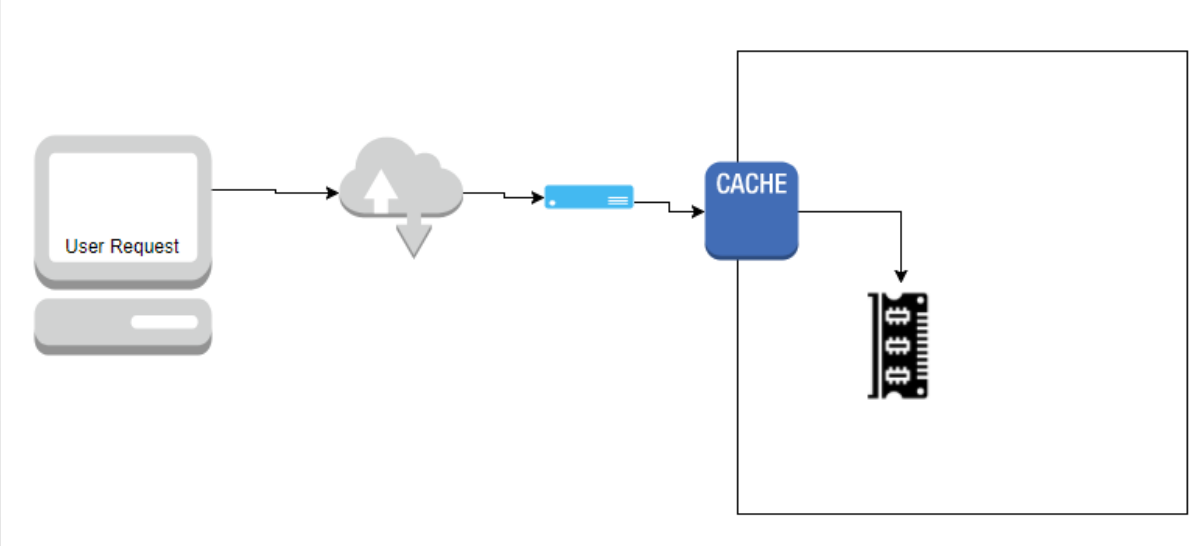
FILE BASED





MEMORY BASED





PROS

- Faster response times
- Save on processing costs
- Save money on bandwidth costs

CONS

- Overcacheing
- Knowing exactly what can be cached takes time
- Extra pieces to maintain

SESSION CACHE

KEY BASED CACHE

WEB CACHE

SESSION CACHING

- This is used for calls that have state.
- Many applications in PHP use sessions, especially many CMS platforms.
- Save time and allow for scalability.

CHANGE AT PHP INI

```
<?php

// Update these lines in your php.ini
session.save_handler = redis
session.save_path    = tcp://127.0.0.1:6379
```

CHANGE IN CODE

```php
<?php

// Set session handler in code redis as an example
ini_set('session.save_handler', 'redis');
ini_set('session.save_path',    'tcp://127.0.0.1:6379');

// Don't forget to sesssion_start()

session_start();
```

DO NOT USE DATABASES FOR SESSIONS AT SCALE

This is the best way to crash your database if running with other data side by side

OR

You will run with to much extra costs by running a full database to run sessions.

KEY BASED CACHING

Take time and make sure to build a plan for key strategies.

REDIS

- Store up to 512MB per entry
- Has persistence built in
- Has clustering built in (3.0+)
- Handles memory better

MEMCACHED

- Store up to 1MB per entry
- No persistence built in
- No scaling tools out of the box
- Does not give back outdated memory

KEY MANAGEMENT STRATEGIES

- Strong keys
- Easy to recreate
- No overlaps possible

STRONG KEYS

- Use combinations of values
- Use everything you have access to

```php
<?php

// data returned from api call of animals
$arr = [
    'id' => '12',
    'name' => 'poodle',
    'description' => 'a dog',
];

// Worst key
$key = time() . $arr['id'];

// Bad key
$key = 'dog' . $arr['name'];

// Good Key but to strong and not specific enough
$key = $arr['id'] . $arr['name'] . $arr['description'];

// Perfectly good Key
$key = 'dogs' . $arr['id'];
```

STRONG KEYS (CONT)

- Use SQL
- This method is a bit of overkill.

```php
<?php
$sql = 'select * from animals where id = 11';

// Good key using your sql query
$key = 'dogs' . $sql;
```

EASY TO RECREATE

- Values should not constantly be changing
- If there are to many variations, you may just not be able to cache

EASY TO RECREATE

- Values should not constantly be changing
- If there are to many variations, you may just not be able to cache

WHAT TO DO IF NOT POSSIBLE BUT CACHEABLE?

EASY TO RECREATE

- Values should not constantly be changing
- If there are to many variations, you may just not be able to cache

WHAT TO DO IF NOT POSSIBLE BUT CACHEABLE?

```php
<?php

// data keeps changing but still cacheable
$arr = [
    'id' => '12',
    'name' => 'poodle',
    'description' => 'a dog',
];

// Cache based on time example of 2 minutes
$ttl = 60 * 2;
```

PHPFASTCACHE

```php
<?php

use phpFastCache\CacheManager;

$cache = CacheManager::Memcached();

$key = generateKeyFunction();

// try to get from Cache first.
$resultsItem = $cache->getItem($key)

if(!$resultsItem->isHit()) {
    // Add in time based arguments here as well
    $resultsItem->set($cURL->get("http://www.youtube.com/api/json/url/keyword/page"))->
    $cache->save($resultsItem);
}

foreach($results as $video) {
    // Output Your Contents HERE
}
```

DOCTRINE CACHE

```php
<?php
// This here is a redis client Can be many alternatives
$redis = new Redis();
$redis->connect('redis_host', 6379);

$cacheDriver = new \Doctrine\Common\Cache\RedisCache();
$cacheDriver->setRedis($redis);

// get id and name from input
$key = 'objType' . 'id' . 'name';;

$obj = $cacheDriver->get($key);

if(null !== \json_decode($obj)){
    return $obj;
}

// Fetch data from source and set and return

$obj = fetchDataFunction();
$key = 'objType' . $obj->id . $obj->name;

// Serialize the object for most cases
$cacheDriver->save($key, \json_encode($obj));
```

DOCTRINE CACHE ALSO SUPPORTS

- Memcached
- Files
- APC
- Xcache
- Write your own

DOCTRINE CACHE ALSO SUPPORTS

- Memcached
- Files
- APC
- Xcache
- Write your own

Be warned though it is very heavy weight

USE RAW ALTERNATIVES

```php
<?php
// This here is a redis client Can be many alternatives
$client = new \Predis\Client();

// get id and name from input
$key = 'objType' . 'id' . 'name';;

$obj = $client->get($key);

if(null !== \json_decode($obj)){
    return $obj;
}

// Fetch data from source and set and return

$obj = fetchDataFunction();
$key = 'objType' . $obj->id . $obj->name;

// Serialize the object for most cases
$client->set($key, \json_encode($obj));
```

PREDIS

- Built completely in PHP
- A bit slower (less of an issue if using phpiredis extension)
- Easier code support

PHPREDIS

- C based extension
- Faster as it runs in C
- Harder to setup and manage

PHPIREDIS

- Phpiredis is an extension for PHP 5.x and 7.x based on hiredis that provides a simple and efficient client for Redis and a fast incremental parser / serializer for the RESP protocol.

```
git clone https://github.com/nrk/phpiredis.git && cd phpiredis && \
phpize && ./configure --enable-phpiredis && \
make && make install && \
echo "extension=phpiredis.so" > /path/to/php/conf.d/
```

https://github.com/nrk/phpiredis

BENCHMARKS

group	name	count	time	single	cost
SET	Redis	50000	1.7460916042328	3.4921832084656E-5	100 %
SET	Predis	50000	1.7508194446564	3.5016388893127E-5	100 %
EXISTS	Redis	50000	1.5275919437408	3.0551838874817E-5	100 %
EXISTS	Predis	50000	1.5728967189789	3.1457934379578E-5	103 %
GET	Redis	50000	1.5822939872742	3.1645879745483E-5	100 %
GET	Predis	50000	1.6208076477051	3.2416152954102E-5	102 %
DEL	Redis	50000	1.5144150257111	3.0288300514221E-5	100 %
DEL	Predis	50000	1.6063549518585	3.212709903717E-5	106 %
HMSET	Redis	50000	1.8996987342834	3.7993974685669E-5	103 %
HMSET	Predis	50000	1.847585439682	3.695170879364E-5	100 %
HMGET	Redis	50000	1.9222128391266	3.8444256782532E-5	100 %
HMGET	Predis	50000	1.9729597568512	3.9459195137024E-5	103 %
ALL	Redis	50000	10.907937288284	0.00021815874576569	100 %
ALL	Predis	50000	11.096460580826	0.00022192921161652	102 %

https://github.com/cheprasov/php-redis-client-vs-predis-test

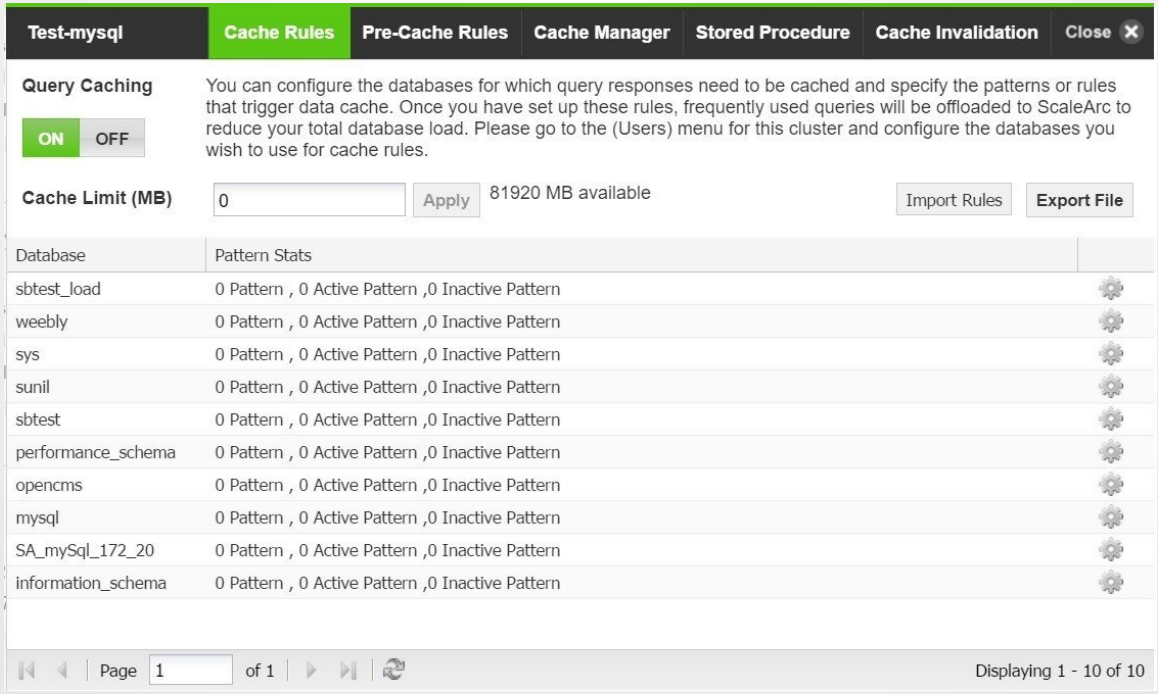
API USAGE

```php
use GuzzleHttp\Client;
use GuzzleHttp\HandlerStack;
use Kevinrob\GuzzleCache\CacheMiddleware;

// Create default HandlerStack
$stack = HandlerStack::create();

// Add this middleware to the top with `push`
$stack->push(new CacheMiddleware(), 'cache');

// Initialize the client with the handler option
$client = new Client(['handler' => $stack]);
```

https://github.com/Kevinrob/guzzle-cache-middleware

THIRD PARTY TOOLS

- ScaleArc/Heimdall
- Please someone build something like this but open source
- Heimdall used to be open :(



QUERY CACHE

- Cache only the query itself so it does not need to be regenerated
- Databases sometimes maintain their own for you.

RESULTS CACHING

- Fully caching objects
- This must be done manually and not handled by your cache storage.

CACHE BUSTING

- Always make sure to clear the cache
- Strategies around this topic are varried

If you are using Doctrine, there is a Second Level Cache still experimental

http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/second-level-cache.html

EXAMPLES OF OVERCACHING

- Using caching for logged in users
- Caching components that are constantly changing
- All of this can goes back to using better keys!

WEB BASED CACHING

- Browser Caching
- Whole Page Caching
- Partial Page Caching

HTTP CACHE HEADERS REQUEST

- Cache-Control: max-age=<seconds>
- Cache-Control: max-stale=<seconds>
- Cache-Control: no-cache
- Cache-Control: only-if-cached

HTTP CACHE HEADERS RESPONSE

- Cache-Control: max-age=<seconds>
- Cache-Control: must-revalidate
- Cache-Control: public
- Cache-Control: private
- Cache-Control: no-cache
- Cache-Control: only-if-cached

WHOLE PAGE CACHING

VARNISH/NGINX (ETC)

- Sits between End User and Web Server
- Serves whole web page effortlessly
- Can be configured for key usage at web layer or configured for time based
- Use microcache to save on calls in seconds

IN MEMORY/FILE

- Use Redis using values as keys for page generation
- Best if using templating like twig

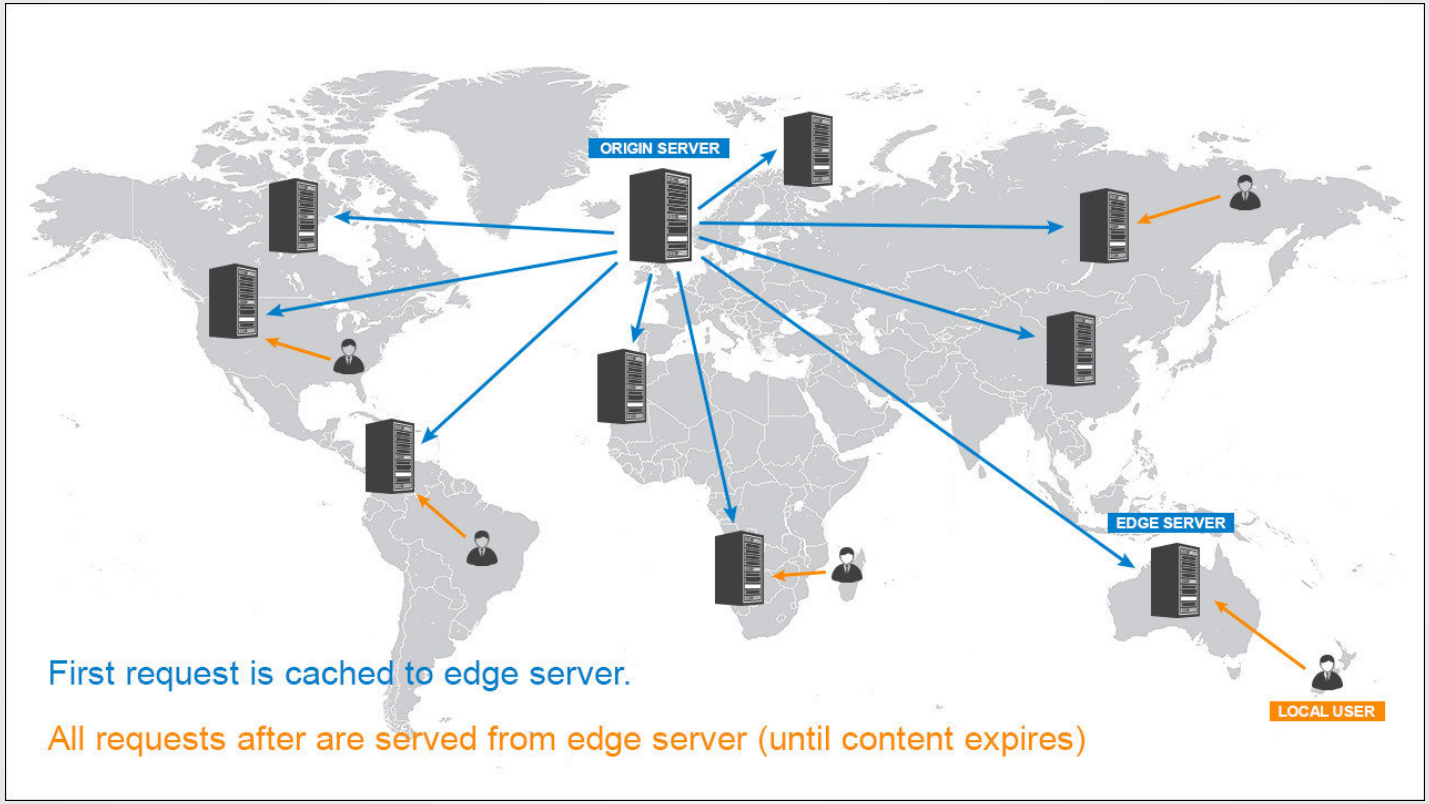
CLOUD CDN CACHE

- Can save files not even in your infrastructure
- Saves computing resources for managed cost
- Examples: Cloudflare/Cloudfront/MaxCDN

DNS CACHING

- All servers do it
- Can cause issues with some DNS based databases

CDN CACHING EXPLAINED



*https://www.crondose.com/2016/07/how-does-a-cdn-work/

CMS CACHING

- Wordpress
- Drupal
- Joomla
- Plugins Plugins Plugins

PARTIAL PAGE CACHING

USING TWIG CACHING

https://github.com/asm89/twig-cache-extension

```
// twig you can also set time to live values for your cache
{% cache %}
    {% for block in entry.myMatrixField %}
        <p>{{ block.text }}</p>
    {% endfor %}
{% endcache %}
```

USING KEYS

```php
<?php
// This here is a redis client Can be many alternatives
$redis = new Redis();
$redis->connect('redis_host', 6379);

$cacheDriver = new \Doctrine\Common\Cache\RedisCache();
$cacheDriver->setRedis($redis);

// get id and name from input
$key = 'objType' . 'id' . 'name';;

$obj = $cacheDriver->get($key);

if(null !== $obj){
    return $obj;
}

// prerender

$obj = $this->render(
            'article/recent_list.html.twig',
            array('articles' => $articles)
        );
$cacheDriver->save($key, $obj);
```

KEEP CALM ITS STORY TIME

BUILDING A SECOND LEVEL CACHE

- A second level cache is acts as a session layer for your objects.

WHAT I STARTED WITH

- All cache keys were only based on http requests and using sql as keys.
- Made cache invalidation near impossible.
- Had to rely completely on ttls

```php
$fullQuery = 'query generated by illuminate';

Cache::fetch(
    $redisClient,
    \sha1($fullQuery),
    function () use ($client, $fullQuery) {
        $response = $client->get('/my/api/call', [
            'query' => $fullQuery,
        ]);

        return [
            1 * Time::MINUTE,
            \json_decode((string) $response->getBody()
        ];
    }
);
```

WHAT I STARTED WITH

- A base cache function to use to simplify the process.

```php
public static function fetch(PredisCachePool $pool, string $cacheKey, callable $fetchFreshData)
{
    if ($pool->hasItem($cacheKey)) {
        return $pool->getItem($cacheKey)->get();
    }

    // Expect to receive a tuple
    [$ttl, $value] = $fetchFreshData();

    $item = $pool->getItem($cacheKey);
    $item->set($value);
    $item->expiresAfter($ttl);
    $pool->save($item);

    return $value;
}
```

CREATING A TTL FILE

1. Figure out how long you can possibly want to hold a stored piece of data
2. Create a central location or set of files for your cache

```php
<?php

namespace Application\Enum;

use Skyzyx\UtilityPack\Time;

class ObjectCacheTtl
{
    public const CAT = 1 * Time::MINUTE;
    public const DOG = 1 * Time::MINUTE;
}
```

Highly recommend making your code readable, even your ttl file.

CREATING A CACHE KEY FILE

1. Create a single enum to manage object or data types

```php
namespace Application\Enum;

class ObjectType
{
    public const DOG = 'dog';

    public const CAT = 'cat';
}
```

CREATING A CACHE KEY FILE

1. Design all of your keys a name
2. Give all prefixes for cache invalidation

```php
namespace Application\Enum;

class ObjectCache
{
    // Start Dog ------
    public const DOG = 'mysql.' . ObjectType::DOG . '.%s.%s';

    public const DOG_ALL = 'mysql.' . ObjectType::DOG . '.ALL.%s';

    public const DOG_PREFIX = 'mysql.' . ObjectType::DOG . '%s.*';

    public const DOG_ALL_PREFIX = 'mysql.' . ObjectType::DOG . '.ALL.*';
}
```

NOW WE ARE COOKING

- Now able to update the keys used for those calls.

```
$dogId = $this->getDogIdFromApi();
$query = $this->getQueryInformation();

$raw = Cache::fetch(
    $redis,
    \sprintf(
        \Application\Enum\ObjectCache::DOG,
        E\ObjectType::POUND. '.' . $pound->id,
        $dogId
    ),
    static function () use ($client, $dogId, $query) {
        $response = $client->get(\sprintf('/my/api/to/call/%s', $dogId), [
            'query' => $query,
        ]);

        return [
            ObjectCacheTtl::DOG,
            \json_decode((string) $response->getBody()),
        ];
    }
);
```

CACHE INVALIDATION

- Create standard way to invalidate

```
public function deleteCachePatterns(string ...$prefixPattern): void
{
    foreach ($prefixPattern as $pattern) {
        // Delete the single cached entity
        foreach (new Keyspace($this->redisClient, $pattern) as $key) {
            $this->redisClient->del($key);
        }
    }
}
```

CACHE INVALIDATION

- On create invalidate your ALL cache

```
$pound = $this->getPound();

/** @var CacheBuster $cacheBuster */
$cacheBuster   = $container['CacheBuster'];
$cacheKeyPound = E\ObjectType::DOG. '.' . $pound->id;
$allDogPattern = \sprintf(E\ObjectCache::DOG_ALL_PREFIX, $cacheKeyPound);

$cacheBuster->deleteCachePatterns($allDogPattern);
```

CACHE INVALIDATION

- On update and delete invalidate ALL and Single

```
$pound = $this->getPound();
$dog   = $this->getDog();

/** @var CacheBuster $cacheBuster */
$cacheBuster   = $container['CacheBuster'];

$cacheKeyPound = E\ObjectType::POUND. '.' . $pound->id;
$dogPattern  = \sprintf(
    E\ObjectCache::DOG_PREFIX,
    $cacheKeyPound,
    $dog->id
);

$allDogPattern = \sprintf(E\ObjectCache::DOG_ALL_PREFIX, $cacheKeyPound);

$cacheBuster->deleteCachePatterns($dogPattern, $allDogPattern);
```

QUESTIONS!?

I'LL TOTALLY ANSWER ANYTHING I CAN

https://joind.in/talk/c6c36