# ENTROPY WINS (HTTPS://WWW.ENTROPYWINS.WTF/BLOG/)

A blog on Software Architecture, Design and Craftsmanship

Clean Code (https://www.entropywins.wtf/blog/tag/clean-code/)

Craftsmanship (https://www.entropywins.wtf/blog/tag/software-craftsmanship/)

Architecture (https://www.entropywins.wtf/blog/tag/software-architecture/)

Testing (https://www.entropywins.wtf/blog/tag/testing/)

PHP (https://www.entropywins.wtf/blog/tag/php/)

Open Source (https://www.entropywins.wtf/blog/tag/open-source/)

MediaWiki (https://www.entropywins.wtf/blog/tag/mediawiki/)

Website home (https://www.entropywins.wtf/)

Professional.Wiki (https://professional.wiki/)

# Implementing the Clean Architecture

2016-11-24 Jeroen (https://www.entropywins.wtf/blog/author/jeroen/) 39 Comments (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comments)

Both Domain Driven Design and architectures such as the Clean Architecture and Hexagonal are often talked about. It's hard to go to a conference on software development and not run into one of these topics. However it can be challenging to find good real-world examples. In this blog post I'll introduce you to an application following the Clean Architecture and incorporating a lot of DDD patterns. The focus is on the key concepts of the Clean Architecture, and the most important lessons we learned implementing it.

## THE APPLICATION

The real-world application we'll be looking at is the Wikimedia Deutschland fundraising software. It is a PHP application written in 2016, replacing an older legacy system. While the application is written in PHP, the patterns followed are by and large language agnostic, and are thus relevant for anyone writing object orientated software.

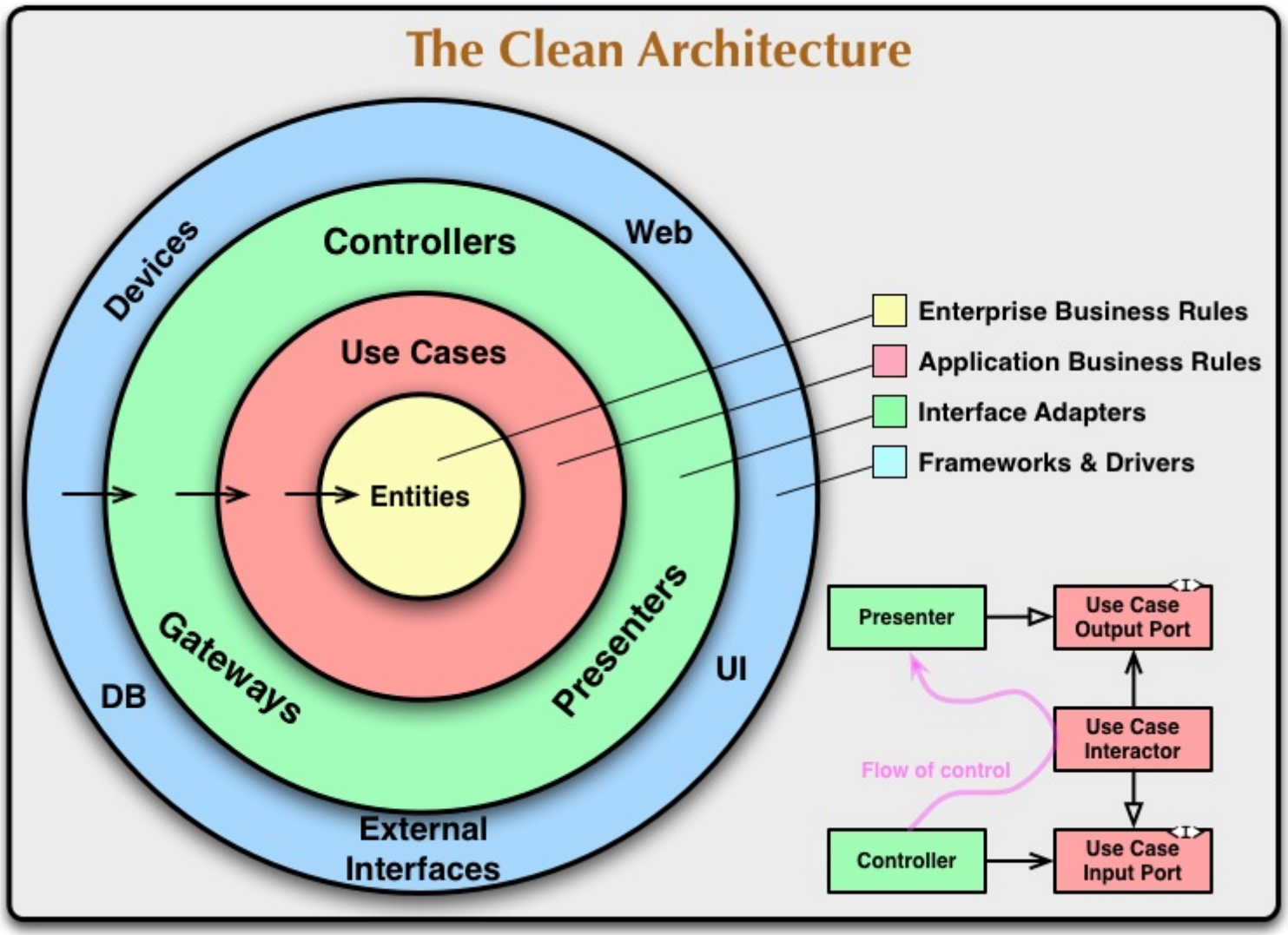
I've outlined what the application is and why we replaced the legacy system in a blog post titled Rewriting the Wikimedia Deutschland fundraising (https://www.entropywins.wtf/blog/2016/11/24/rewriting-the-wikimedia-deutschland-funrdraising/). I recommend you have a look at least at its "The application" section, as it will give you a rough idea of the domain we're dealing with.

## A FAMILY OF ARCHITECTURES

Architectures such as Hexagonal and the Clean Architecture are very similar. At their core, they are about separation of concerns. They decouple from mechanisms such as persistence and used frameworks and instead focus on the domain and high level policies. A nice short read on this topic is Unclebob's blog post on the Clean Architecture

(https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html). Another recommended post is Hexagonal != Layers (http://tpierrain.blogspot.de/2016/04/hexagonal-layers.html), which explains that how just creating a bunch of layers is missing the point.

## THE CLEAN ARCHITECTURE




(https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html)

The arrows crossing the circle boundaries represent the allowed direction of dependencies. At the core is the domain. “Entities” here means Entities such as in Domain Driven Design, not to be confused by ORM entities. The domain is surrounded by a layer containing use cases (sometimes called interactors) that form an API that the outside world, such as a controller, can use to interact with the domain. The use cases themselves only bind to the domain and certain cross cutting concerns such as logging, and are devoid of binding to the web, the database and the framework.

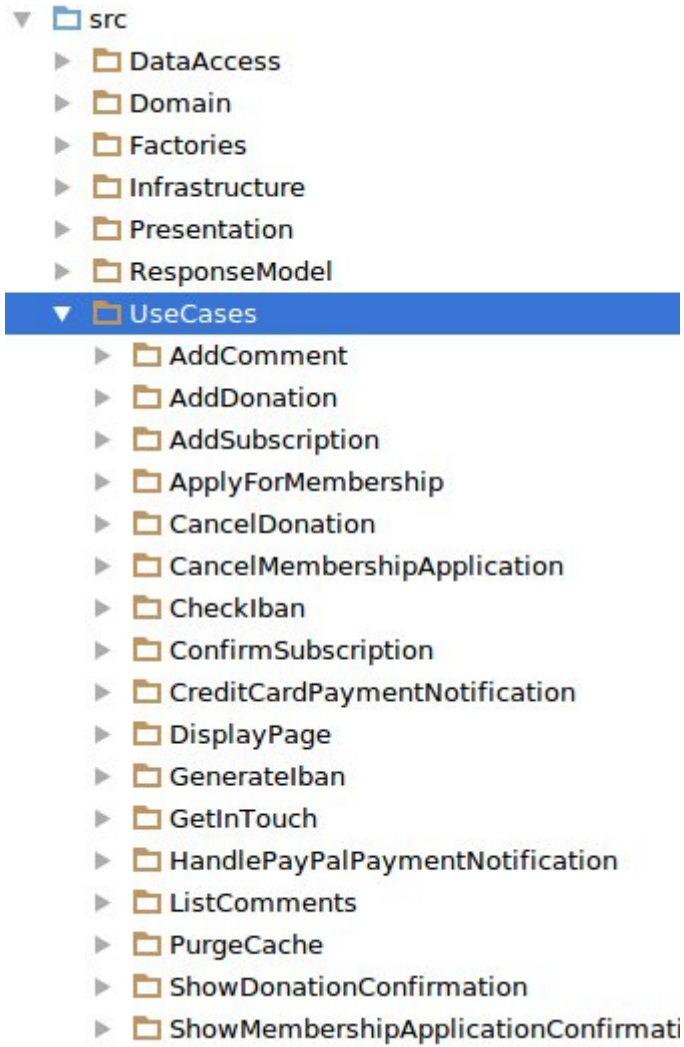
```
class CancelDonationUseCase {
    private /* DonationRepository */ $repository;
    private /* Mailer */ $mailer;

    public function cancelDonation( CancelDonationRequest $r ): CancelDonationResponse {
        $this->validateRequest( $r );

        $donation = $this->repository->getDonationById( $r->getDonationId() );
        $donation->cancel();
        $this->repository->storeDonation( $donation );

        $this->sendConfirmationEmail( $donation );

        return new CancelDonationResponse( /* ... */ );
    }
}
```

In this example you can see how the UC for canceling a donation gets a request object, does some stuff, and then returns a response object. Both the request and response objects are specific to this UC and lack both domain and presentation mechanism binding. The stuff that is actually done is mainly interaction with the domain through Entities, Aggregates and Repositories.

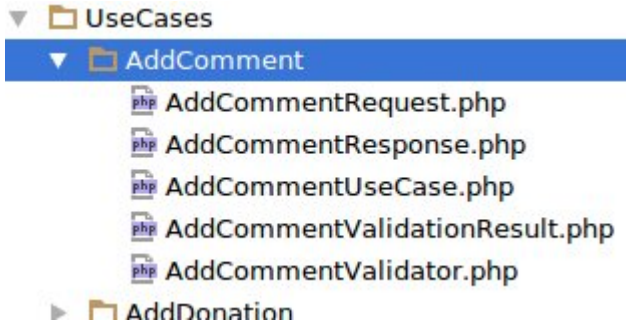
```php
$app->post(
    '/cancel-donation',
    function( Request $httpRequest ) use ( $factory ) {
        $requestModel = new CancelDonationRequest(
            $httpRequest->request->get( 'donation_id' ),
            $httpRequest->request->get( 'update_token' )
        );

        $useCase = $factory->newCancelDonationUseCase();
        $responseModel = $useCase->cancelDonation( $requestModel );

        $presenter = $factory->newNukeLaunchingResultPresenter();
        return new Response( $presenter->present( $responseModel ) );
    }
);
```

This is a typical way of invoking a UC. The framework we're using is Silex, which calls the function we provided when the route matches. Inside this function we construct our framework agnostic request model and invoke the UC with it. Then we hand over the response model to a presenter to create the appropriate HTML or other such format. This is all the framework bound code we have for canceling donations. Even the presenter does not bind to the framework, though it does depend on Twig.

If you are familiar with Silex, you might already have noticed that we're constructing our UC different than you might expect. We decided to go with our own top level factory, rather than using the dependency injection mechanism provided by Silex: Pimple. Our factory internally actually uses Pimple, though this is not visible from the outside. With this approach we gain a nicer access to service construction, since we can have a `getLogger()` method with `LoggerInterface` return type hint, rather than accessing `$app['logger']` or some such, which forces us to bind to a string and leaves us without type hint.




This use case based approach makes it very easy to see what our system is capable off at a glance.



And it makes it very easy to find where certain behavior is located, or to figure out where new behavior should be put.

All code in our `src/` directory is framework independent, and all code binding to specific persistence mechanisms resides in `src/DataAccess`. The only framework bound code we have are our very slim "route handlers" (kinda like controllers), the web entry point and the Silex bootstrap.

For more information on The Clean Architecture I can recommend Robert C Martins NDC 2013 talk (https://www.youtube.com/watch?v=Nsjsiz2A9mg). If you watch it, you will hopefully notice how we slightly deviated from the UseCase structure like he presented it. This is due to PHP being an interpreted language, and thus does not need certain interfaces that are beneficial in compiled languages.

## LESSON LEARNED: BOUNDED CONTEXTS

By and large we started with the donation related use cases and then moved on to the membership application related ones. At some point, we had a `Donation` entity/aggregate in our domain, and a bunch of value objects that it contained.

```
1 class Donation {
2     private /* int|null */            $id
3     private /* PersonalInfo|null */   $personalInfo
4     /* ... */
5 }
```

```
1 class PersonalInfo {
2     private /* PersonName */          $name
3     private /* PhysicalAddress */     $address
4     private /* string */              $emailAddress
5 }
```

As you can see, one of those value objects is `PersonalInfo`. Then we needed to add an entity for membership applications. Like donations, membership applications require a name, a physical address and an email address. Hence it was tempting to reuse our existing `PersonalInfo` class.

```
class MembershipApplication {
    private /* int|null */            $id
    private /* PersonalInfo|null */   $personalInfo
    /* ... */
}
```

Luckily a complication made us realize that going down this path was not a good idea. This complication was that membership applications also have a phone number and an optional date of birth. We could have forced code sharing by doing something hacky like adding new optional fields to `PersonalInfo`, or by creating a `MorePersonalInfo` derivative.

Approaches such as these, while resulting in some code sharing, also result in creating binding between `Donation` and `MembershipApplication`. That's not good, as those two entities don't have anything to do with each other. Sharing what happens to be the same at present is simply not a good idea. Just imagine that we did not have the phone number and date of birth in our first version, and then needed to add them. We'd either end up with one of those hacky solutions, or need to refactor code that has nothing to do (apart from the bad coupling) with what we want to modify.

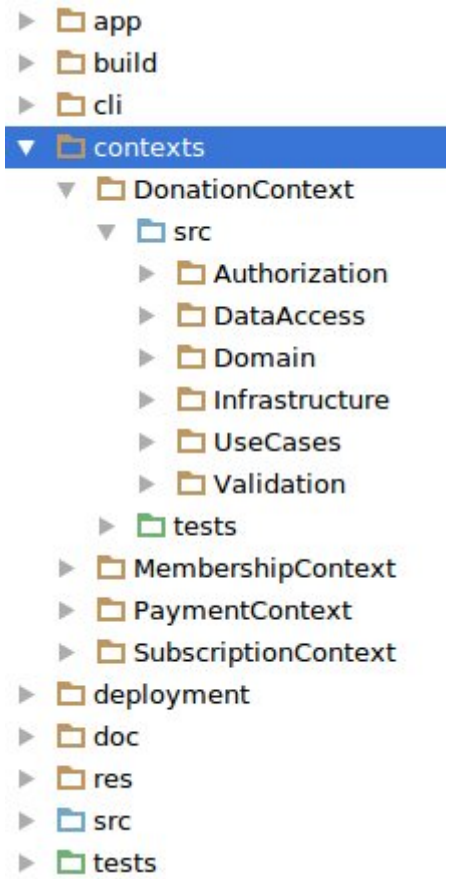
What we did is renaming `PersonalInfo` to `Donor` and introduce a new `Applicant` class.

```
class Donor {
    private /* PersonName */          $name
    private /* PhysicalAddress */     $address
    private /* string */              $emailAddress
}
```

```
class Applicant {
    private /* PersonName */          $name
    private /* PhysicalAddress */     $address
    private /* EmailAddress */        $email
    private /* PhoneNumber */         $phone
    private /* DateTime|null */       $dateOfBirth
}
```

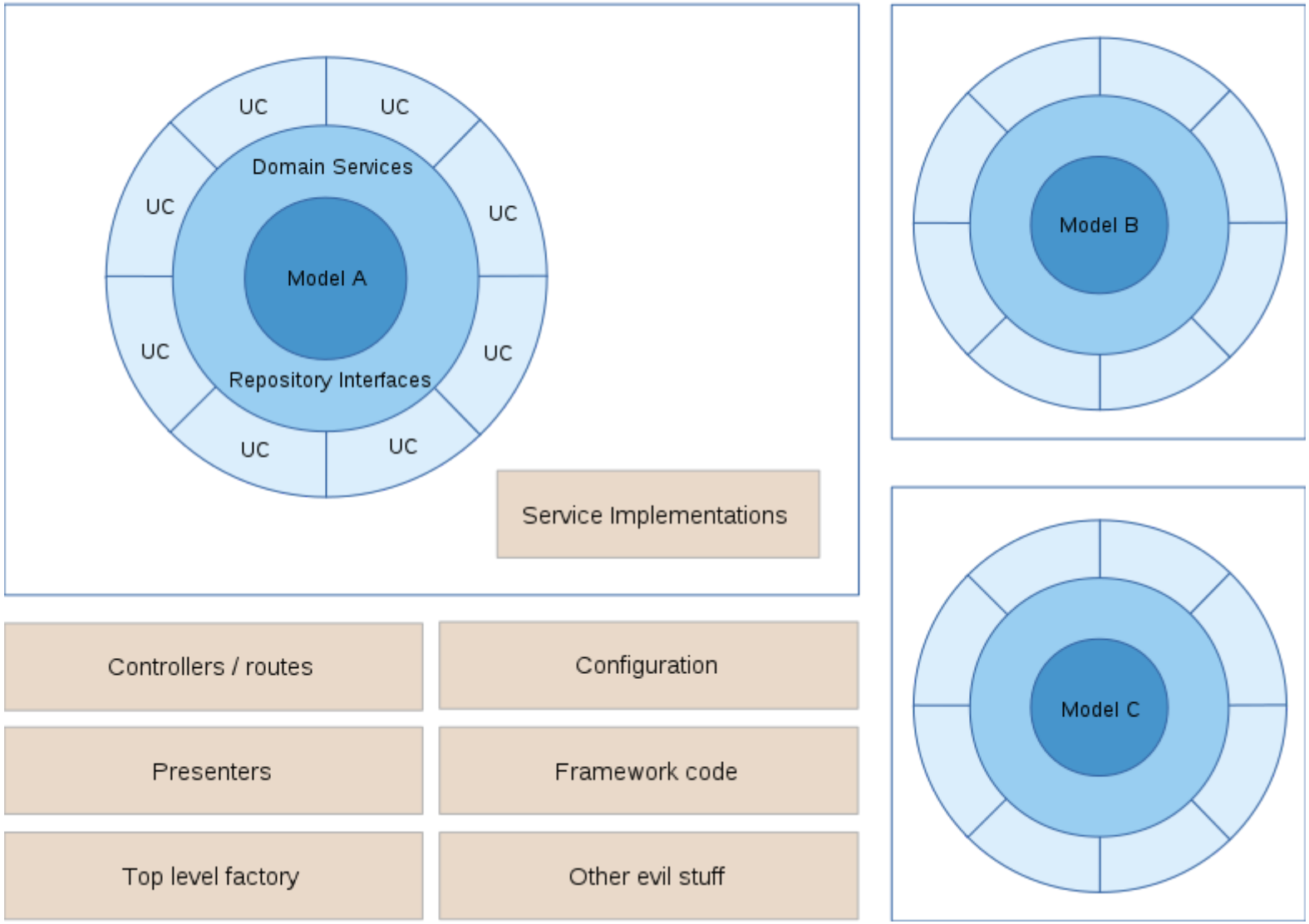
These names are better since they are about the domain (see ubiquitous language (https://www.agilealliance.org/glossary/ubiquitous-language/)) rather than some technical terms we needed to come up with.

Amongst other things, this rename made us realize that we where missing some explicit boundaries in our application. The donation related code and the membership application related code where mostly independent from each other, and we agreed this was a good thing. To make it more clear that this is the case and highlight violations of that rule, we decided to reorganize our code to follow the strategic DDD pattern of Bounded Contexts (http://martinfowler.com/bliki/BoundedContext.html).




This mainly consisted out of reorganizing our directory and namespace structure, and a few instances of splitting some code that should not have been bound together.

Based on this we created a new diagram to reflect the high level structure of our application. This diagram, and a version with just one context (https://www.entropywins.wtf/blog/2016/09/09/clean-architecture-diagram/), are available for use under CC-0.




## LESSON LEARNED: VALIDATION

A big question we had near the start of our project was where to put validation code. Do we put it in the UCs, or in the controller-like code that calls the UCs?

One of the first UCs we added was the one for adding donations. This one has a request model that contains a lot of information, including the donor's name, their email, their address, the payment method, payment amount, payment interval, etc. In our domain we had several value objects for representing parts of donations, such as the donor or the payment information.

```
class Donation {
    private /* int|null */            $id
    private /* Donor|null */          $donor
    private /* DonationPayment */     $payment
    /* ... */
}

class Donor {
    private /* PersonName */          $name
    private /* PhysicalAddress */     $address
    private /* string */              $emailAddress
}
```

Since we did not want to have one object with two dozen fields, and did not want to duplicate code, we used the value objects from our domain in the request model.

```
class AddDonationRequest {
    private /* Donor|null */          $donor
    private /* DonationPayment */     $payment
    /* ... */
}
```

If you've been paying attention, you'll have realized that this approach violates one of the earlier outlined rules: nothing outside the UC layer is supposed to access anything from the domain. If value objects from the domain are exposed to whatever constructs the request model, i.e. a controller, this rule is violated. Loose from the this abstract objection, we got into real trouble by doing this.

Since we started doing validation in our UCs, this usage of objects from the domain in the request necessarily forced those objects to allow invalid values. For instance, if we're validating the validity of an email address in the UC (or a service used by the UC), then the request model cannot use an `EmailAddress` which does sanity checks in its constructor.

We thus refactored our code to avoid using any of our domain objects in the request models (and response models), so that those objects could contain basic safeguards.

We made a similar change by altering which objects get validated. At the start of our project we created a number of validators that worked on objects from the domain. For instance a `DonationValidator` working with the `Donation` Entity. This `DonationValidator` would then be used by the `AddDonationUseCase`. This is not a good idea, since the validation that needs to happen depends on the context. In the `AddDonationUseCase` certain restrictions apply that don't always hold for donations. Hence having a general looking `DonationValidator` is misleading. What we ended up doing instead is having validation code specific to the UCs, be it as part of the UC, or when too complex, a separate validation service in the same namespace. In both cases the validation code would work on the request model, i.e. `AddDonationRequest`, and not bind to the domain.

After learning these two lessons, we had a nice approach for policy-based validation. That's not all validation that needs to be done though. For instance, if you get a number via a web request, the framework will typically give it to you as a string, which might thus not be an actual number. As the request model is supposed to be presentation mechanism agnostic, certain validation, conversion and error handling needs to happen before constructing the request model and invoking the UC. This means that often you will have validation in two places: policy based validation in the UC, and presentation specific validation in your controllers or equivalent code. If you have a string to integer conversion, number parsing or something internationalization specific, in your UC, you almost certainly messed up.

## CLOSING NOTES

You can find the Wikimedia Deutschland fundraising application on GitHub (https://github.com/wmde/FundraisingFrontend) and see it running in production (https://spenden.wikimedia.de/). Unfortunately the code of the old application is not available for comparison, as it is not public. If you have questions, you

can leave a comment, or contact me (https://www.entropywins.wtf/). If you find an issue or want to contribute, you can create a pull request (https://github.com/wmde/FundraisingFrontend/pulls). If you are looking for my presentation on this topic, view the slides (https://entropywins.wtf/slides/fun-architecture).

As a team we learned a lot during this project, and we set a number of firsts at Wikimedia Deutschland (http://software.wikimedia.de/), or the wider Wikimedia movement for that matter. The new codebase is the cleanest non-trivial application we have, or that I know of in PHP world. It is fully tested, contains less than 5% framework bound code, has strong strategic separation between both contexts and layers, has roughly 5% data access specific code and has tests that can be run without any real setup. (I might write another blog post on how we designed our tests and testing environment.)

Many thanks for my colleagues Kai Nissen (https://github.com/KaiNissen) and Gabriel Birke (https://github.com/gbirke) for being pretty awesome during our rewrite project.

## FURTHER READING



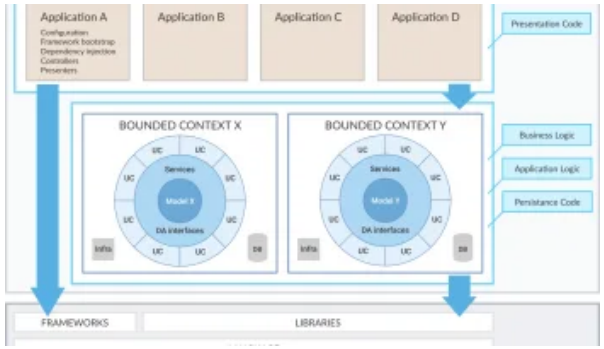
Other things to look at:

- Clean Architecture + Bounded Contexts (https://www.entropywins.wtf/blog/2018/08/14/clean-architecture-bounded-contexts/)
- Bounded Contexts in the Wikimedia Fundraising Software (https://www.entropywins.wtf/blog/2018/08/14/bounded-contexts-in-the-wikimedia-fundraising-software/)
- Clean Architecture: UseCase tests (https://www.entropywins.wtf/blog/2018/08/01/clean-architecture-usecase-tests/)

- Clean Architecture (https://www.goodreads.com/book/show/18043011-clean-architecture), Robert C. Martin, 2016



**RELATED**

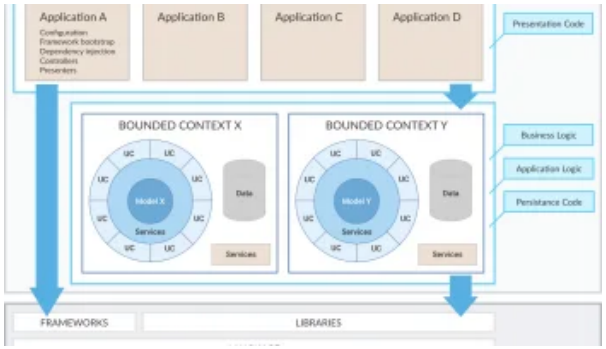



(https://www.entropywins.wtf/blog/2018/08/14/clean-architecture-bounded-contexts/)

Clean Architecture + Bounded Contexts (https://www.entropywins.wtf/blog/2018/08/14/clean-architecture-bounded-contexts/)

2018-08-14

In "Programming"




(https://www.entropywins.wtf/blog/2018/08/14/bounded-contexts-in-the-wikimedia-fundraising-software/)

Bounded Contexts in the Wikimedia Fundraising Software (https://www.entropywins.wtf/blog/2018/08/14/bounded-contexts-in-the-wikimedia-fundraising-software/)

2018-08-14

In "Programming"

Guidelines for New Software Projects (https://www.entropywins.wtf/blog/2018/05/09/guidelines-for-new-software-projects/)

In this blog post I share the Guidelines for New Software Projects that we use at Wikimedia Deutschland. I wrote down these guidelines recently after a third-party that was contracted by Wikimedia Deutschland

2018-05-09

In "Programming"

← Rewriting the Wikimedia Deutschland fundraising (https://www.entropywins.wtf/blog/2016/11/24/rewriting-the-wikimedia-deutschland-funrdraising/)

Final Rush Pro 5 → (https://www.entropywins.wtf/blog/2016/11/27/final-rush-pro-5/)

# 39 thoughts on “Implementing the Clean Architecture”

Pingback: Rewriting the Wikimedia Deutschland fundraising – Entropy Wins (https://www.entropywins.wtf/blog/2016/11/24/rewriting-the-wikimedia-deutschland-funrdraising/)

Pingback: Missing in PHP7: Value objects – Entropy Wins (https://www.entropywins.wtf/blog/2016/02/03/missing-in-php7-value-objects/)

Dmitriy (https://lessthan12ms.com) says:
2017-01-17 at 05:26 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354559)

Hi Jeroen,
you wrote cool stuff here, I enjoyed reading it!

While I am still reviewing your codebase, I wonder why you moved presenters to `src` folder while in my mind those are related to specific context, for example this presenter is working with specific context response: https://github.com/wmde/FundraisingFrontend/blob/master/src/Presentation/Presenters/CancelDonationHtmlPresenter.php (https://github.com/wmde/FundraisingFrontend/blob/master/src/Presentation/Presenters/CancelDonationHtmlPresenter.php)?

Reply →

Jeroen says:
2017-01-18 at 18:54 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354560)

Hey Dmitriy, that's a good question.

We did not move the presenters to `src`, but moved the contexts out of it. Still, the question of why we did not put the presenters together with the associated context remains valid.

We decided to limit our contexts to things bound to the domain model. That means the outermost "layer" of the contexts are the usecases, and that they remain fully framework independent.

From the literature I figured that what you suggest is the more common approach. Indeed we discussed reorganizing our codebase as such, but no compelling arguments where found that justify the effort.

I raised this question on the DDD CQRS list when we where talking about this in out team. Unfortunately we did not get any reply.
https://groups.google.com/forum/#!topic/dddcqrs/tWc6iZGhvUU
(https://groups.google.com/forum/#!topic/dddcqrs/tWc6iZGhvUU)

Reply →

Dmitriy (https://lessthan12ms.com) says:
2017-01-19 at 17:51 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354561)

I see, it makes sense why you did it.

P.s. You may also consider joining in the DDDinPHP maillist to reach out to more developers for feedback: https://groups.google.com/forum/#!forum/dddinphp (https://groups.google.com/forum/#!forum/dddinphp)

Reply →

---

Pingback: Why Every Single Argument of Dan North is Wrong – Entropy Wins (https://www.entropywins.wtf/blog/2017/02/17/why-every-single-argument-of-dan-north-is-wrong/)

---

Paul M. Jones (http://paul-m-jones.com) says:
2017-02-20 at 17:26 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354567)

This is a fantastic writeup. It's very nice to see that the Silex actions are almost point-for-point examples of Action-Domain-Responder . (Some of the actions might do with trivial refactorings to bring them even more in-line with the pattern, but overall it's very well done.)

Reply →

Paul M. Jones (http://paul-m-jones.com) says:
2017-02-20 at 17:27 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354568)

Action-Domain-Responder is here: http://pmjones.io/adr (http://pmjones.io/adr)

Reply →

Pingback: WikiMedia, Clean Architecture, and ADR | Paul M. Jones (http://paul-m-jones.com/archives/6535)

Paul M. Jones (http://paul-m-jones.com) says:
2017-02-21 at 16:02 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354571)

A longer reply here: http://paul-m-jones.com/archives/6535 (http://paul-m-jones.com/archives/6535)

An excerpt:

“The only place where Jeroen’s implementation deviates from ADR is that the Action code builds the presentation itself, instead of handing off to a Responder. (This may be a result of adhering to the idioms and expectations specific to Silex.) Because the rest of the implementation is so well done, refactoring to a separated presentation in the form of a Responder is a straightforward exercise. Let’s see what that might look like.”

Reply →

m1x0n (http://gravatar.com/m1x0n) says:
2017-02-28 at 10:29 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354573)

Good reading. Thanks for sharing code examples.

Reply →

Pingback: PHP Annotated Monthly – March 2017 | PhpStorm Blog (https://blog.jetbrains.com/phpstorm/2017/03/php-annotated-monthly-march-2017/)

Pingback: WikiMedia, arquitetura limpa e ADR – Agency Major (http://www.agencymajor.com.br/wikimedia-arquitetura-limpa-e-adr/)

Pingback: WikiMedia, arquitetura limpa e ADR – Tudo sobre PHP (http://php.lenonleite.com.br/2017/03/09/wikimedia-arquitetura-limpa-e-adr/)

Pingback: WikiMedia, arquitetura limpa e ADR | grupo IO Multi Soluções Inteligentes (https://grupoio.com.br/blog/digital/wikimedia-arquitetura-limpa-e-adr/)

Pingback: Clean architecture implemented as a PHP app | Dmitriy Lezhnev - PHP, LEMP, backend developer (https://lessthan12ms.com/clean-architecture-implemented-as-a-php-app/)

Pingback: Clean architecture links | Dmitriy Lezhnev - PHP, LEMP, backend developer (https://lessthan12ms.com/clean-architecture-links/)

Pingback: Generic Entity handling code – Entropy Wins (https://www.entropywins.wtf/blog/2017/05/24/generic-entity-handling-code/)

Pingback: The Fallacy of DRY – Entropy Wins (https://www.entropywins.wtf/blog/2017/09/06/the-fallacy-of-dry/)

NNPlaya (http://www.nnplaya.pl) says:
2017-09-28 at 14:52 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354679)

Hi! Thank you very much for this article and for sharing real world code. It helped me very much.

Reply →

---

Pingback: Introduction to Iterators and Generators in PHP – Entropy Wins (https://www.entropywins.wtf/blog/2017/10/16/introduction-to-iterators-and-generators-in-php/)

---

Todd Empcke (https://plus.google.com/+ToddEmpcke) says:
2017-11-13 at 01:58 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354705)

First, great article. I am a big fan of DDD and UncleBob both. I took the time to clone the git repo so I could browse it in PhpStorm and was really enjoying going though to see how you did the separation of concerns and boundaries etc. Then I came across https://github.com/wmde/FundraisingFrontend/blob/master/src/Factories/FunFunFactory.php (https://github.com/wmde/FundraisingFrontend/blob/master/src/Factories/FunFunFactory.php) … please explain, what the heck happened here???? there are 160 use statements at the top of that 1000+ line file … It looks like a dumping ground. Though I realize I am, it is not really my intention to be critical. I found this article searching for good examples of clean architecture in php and was happy I found your article.

Reply →

Jeroen says:
2017-11-17 at 01:32 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354706)

Hi Todd. You are right that this file has a lot of lines and many many dependencies. It is the top-level factory of our web application. It is the wiring where we inject the dependencies of our classes. This is single responsibility. And even though there is a lot of code, it is rather flat and uniform, making it easy to modify. We *could* split the top-level factory, though I suspect we'd be making a bad trade-off. Having a big top-level factory (or equivalent DI mechanism) is, as far as I am aware, not uncommon and typically not considered a problem.

Reply →

---

Pingback: Collaboration Best Practices – Entropy Wins (https://www.entropywins.wtf/blog/2018/02/22/collaboration-best-practices/)

---

Norman says:
2018-03-29 at 10:38 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354731)

Hi! Thank you very much for this post, it also helped me a lot!

Though, i have a question:
If we say the UseCases are the the only layer one can access to the domain:

there are several handlers in https://github.com/wmde/FundraisingFrontend/tree/master/app/RouteHandlers (https://github.com/wmde/FundraisingFrontend/tree/master/app/RouteHandlers)
that use domain-objects directly (or atleast construct them).

How does this conform to the clean-architecture?

Reply →

Jeroen says:
2018-04-04 at 23:03 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354733)

Hi Norman! There is more than one answer to that question and it depends on which code exactly you are taking about.

In this codebase we have some instances where this architecture rule has been violated where it should not have been. If you search for TODOs you are likely to find some of those. These violations where created due to our lack of experience with The Clean Architecture. The ones that remain are not high priority issues, as long as they are not made worse, and hence likely will stay as they are.

There might also be cases where we deliberately made an exception to this rule. There definitely are route handlers that use services directly rather than using a UseCase. If the UseCase just takes argument set x and passes it along to some service taking the same argument set, and thus not containing any application logic, having it is not necessary (and possibly harmful).

Reply →

Norman says:
2018-04-05 at 21:03 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354735)

Thank you for replying! (in such detail, too)

That explains everything and answered my question! (also confirming my assumption, why certain decisions were made). It makes sense, to not use a “useCase” if it is simply delegating simple data-sets! ☺

Reply →

Pingback: Software Craftsmanship at Wikimedia Deutschland: Rewriting the Wikimedia Deutschland fundraising application – Wikimedia Deutschland Blog (https://blog.wikimedia.de/2016/11/25/software-craftsmanship-at-wikimedia-deutschland-rewriting-the-wikimedia-deutschland-fundraising-application/)

Pingback: Clean Architecture: UseCase tests – Entropy Wins (https://www.entropywins.wtf/blog/2018/08/01/clean-architecture-usecase-tests/)

Pingback: Bounded Contexts in the Wikimedia Fundraising Software – Entropy Wins (https://www.entropywins.wtf/blog/2018/08/14/bounded-contexts-in-the-wikimedia-fundraising-software/)

Pingback: Clean Architecture + Bounded Contexts – Entropy Wins (https://www.entropywins.wtf/blog/2018/08/14/clean-architecture-bounded-contexts/)

Pierre (http://nogues.pro) says:
2018-10-10 at 13:41 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-354818)

That was a very good read about clean architecture. That's the first time I see multiple DDD bounded contexts in action and it is nice. Regarding the previous comments, where controllers reuse directly domain service instead of using UseCase, well it's a tradeoff, I faced similar situation and I don't think there is a right answer, sometime it doesn't worth it to add delegate boilerplate. Accessing directly the domain from low level is not a big issue as accessing directly low level from domain. Very good job, your codebase is a reference.

Reply →

---

Pingback: Applications as Frameworks - Entropy Wins (https://www.entropywins.wtf/blog/2019/02/28/applications-as-frameworks/)

---

Valentin Tudor Mocanu (http://agiledesign.org) says:
2019-07-18 at 16:28 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-355389)

Very interesting ! We have take some very similar decisions while implementing Clean Architecture. We also have differences ☺

Reply →

Michael (http://seven-kingdoms.ru/) says:
2019-09-07 at 07:59 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-355421)

Why are you so firm that outer layer could only access one layer under them?
Why View layer (controllers) can not have access to Domain Entities?
Uncle Bob's Dependency Rule says only that nothing from outer layers should be used in the inner layers.
Your restriction is not mentioned.

Reply →

Jeroen says:
2019-09-13 at 09:03 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-355424)

Hey Micheal. I also mentioned this restriction in this post https://www.entropywins.wtf/blog/2018/08/14/clean-architecture-bounded-contexts/ (https://www.entropywins.wtf/blog/2018/08/14/clean-architecture-bounded-contexts/) though likewise did not explain it much.

I got that idea from Uncle Bob. Since this is quite long ago I do not remember where, though as far as I recall he mentioned it in several places.

The basic idea here is that there should be no logic in the presentation layer. And there should be no presentation concerns in the domain model. Without the boundary that is formed by the Use Case layer, it is easy for such violations to happen. By having the Use Case layer as a boundary you also get a nicely defined public API to your domain model. Refactoring the model is a lot easier if only the Use Cases use it rather than also any number of presentation concerns, possibly in different applications.

Reply →

Marian says:
2020-02-27 at 18:48 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-355485)

Hey guys,
You've done a great job and I would say that your project is one of the best PHP + Clean projects which are available online. Additionally you put a lot of effort into the BC separation – something that is completely missing in the hundreds of "clean" projects around. The presenter part is also visible and well thought. So … gut gemacht 😉

Do you have any plans to get rid of this spaghetti class any time soon?
https://github.com/wmde/FundraisingFrontend/blob/master/app/Routes.php (https://github.com/wmde/FundraisingFrontend/blob/master/app/Routes.php)

I am pretty sure you can organize this much better.
The other huge file – FunFun – unfortunately it is too late to fix it but just as a suggestion – there is the “Pure DI” which could have probably fit well into the project.

Anyways, excellent job one more time!

Reply →

**Jeroen** says:
2020-02-28 at 08:34 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-355486)

Hi Marian! Thanks for the kind words.

I would not classify either Routes.php or FunFunFactory as spaghetti. Yes, these two contain a lot of code. Yes, they contain a lot of things that could be separated. Most of the time that means there is a problem. However for these cases the code is (1) highly uniform and (2) highly independent.

If you put the code from Routes.php in dedicated files, the structure would not change. The benefit would just be that you have one route per file. And not a clear cut benefit at that. Having things in one file has its upsides as well. We did put a number of routes in dedicated files because we “needed” dedicated classes for them.
https://github.com/wmde/FundraisingFrontend/tree/master/app/Controllers (https://github.com/wmde/FundraisingFrontend/tree/master/app/Controllers)

You could double the size of FunFunFactory and it would make essentially no difference. Same goes with reducing its size 10x. I used this top level factory approach in many projects and find working with one that has 5 methods just as easy as one with 500. Using a DIC (like you have in Symfony for instance) has its ups and its downs. I’m not convinced it is the better approach, so I tend to default to the simpler one used in this project.

Reply →

Marian says:
2020-02-28 at 12:01 (https://www.entropywins.wtf/blog/2016/11/24/implementing-the-clean-architecture/#comment-355487)

Hi Jeroen,
Thanks for the quick reply ☺

I noticed that you have controllers for some of the actions and I assume that it depends on the controller complexity. Still I would prefer to keep the “entry point” of a PHP application as short as possible since every request should go through it. I know that we have OpCache which works quite well but I think it is nice if we could pass the control to the specific code as fast as possible.

For the dependency injection I meant just using DI but not necessarily with a DI-Container. I know that it is quite convenient to pass the big “bucket” to every controller so that they can ask for whatever they want (and therefore transitively depending on maaaaany classes) but the other approach is that every controller specifies the dependencies it needs and they are passed as constructor arguments by the responsible dependency provider.

P.S. I hope you don’t get my comments as a criticism. I myself try to build a similar clean solution and I just see how common our problems/challenges/decision points are ☺

Reply →

## LEAVE A REPLY

Enter your comment here...

This site uses Akismet to reduce spam. Learn how your comment data is processed (https://akismet.com/privacy/).

Enjoying my blog or open source projects? Consider giving something small back! (https://github.com/sponsors/JeroenDeDauw) <3

Search …

## NEWSLETTER

Sign up below to receive news on my upcoming Clean Architecture book, including a discount:

Your email address

SUBSCRIBE

## RECENT POSTS

New MediaWiki blog (https://www.entropywins.wtf/blog/2019/10/26/new-mediawiki-blog/)

Applications as Frameworks (https://www.entropywins.wtf/blog/2019/02/28/applications-as-frameworks/)

Readable Functions: Guard Clause (https://www.entropywins.wtf/blog/2019/01/14/readable-functions-guard-clause/)

## ARCHIVES

October 2017 (https://www.entropywins.wtf/blog/2017/10/)

September 2017 (https://www.entropywins.wtf/blog/2017/09/)

June 2017 (https://www.entropywins.wtf/blog/2017/06/)

May 2017 (https://www.entropywins.wtf/blog/2017/05/)

April 2017 (https://www.entropywins.wtf/blog/2017/04/)

February 2017 (https://www.entropywins.wtf/blog/2017/02/)

January 2017 (https://www.entropywins.wtf/blog/2017/01/)

December 2016 (https://www.entropywins.wtf/blog/2016/12/)

November 2016 (https://www.entropywins.wtf/blog/2016/11/)

October 2016 (https://www.entropywins.wtf/blog/2016/10/)

September 2016 (https://www.entropywins.wtf/blog/2016/09/)

August 2016 (https://www.entropywins.wtf/blog/2016/08/)

July 2016 (https://www.entropywins.wtf/blog/2016/07/)

June 2016 (https://www.entropywins.wtf/blog/2016/06/)

May 2016 (https://www.entropywins.wtf/blog/2016/05/)

April 2016 (https://www.entropywins.wtf/blog/2016/04/)

March 2016 (https://www.entropywins.wtf/blog/2016/03/)

February 2016 (https://www.entropywins.wtf/blog/2016/02/)

January 2016 (https://www.entropywins.wtf/blog/2016/01/)

December 2015 (https://www.entropywins.wtf/blog/2015/12/)

November 2015 (https://www.entropywins.wtf/blog/2015/11/)

October 2015 (https://www.entropywins.wtf/blog/2015/10/)

September 2015 (https://www.entropywins.wtf/blog/2015/09/)

August 2015 (https://www.entropywins.wtf/blog/2015/08/)

June 2015 (https://www.entropywins.wtf/blog/2015/06/)

February 2015 (https://www.entropywins.wtf/blog/2015/02/)

September 2014 (https://www.entropywins.wtf/blog/2014/09/)

August 2014 (https://www.entropywins.wtf/blog/2014/08/)

July 2014 (https://www.entropywins.wtf/blog/2014/07/)

May 2014 (https://www.entropywins.wtf/blog/2014/05/)

April 2014 (https://www.entropywins.wtf/blog/2014/04/)

March 2014 (https://www.entropywins.wtf/blog/2014/03/)

February 2014 (https://www.entropywins.wtf/blog/2014/02/)

January 2014 (https://www.entropywins.wtf/blog/2014/01/)

December 2013 (https://www.entropywins.wtf/blog/2013/12/)

November 2013 (https://www.entropywins.wtf/blog/2013/11/)

October 2013 (https://www.entropywins.wtf/blog/2013/10/)

September 2013 (https://www.entropywins.wtf/blog/2013/09/)

July 2013 (https://www.entropywins.wtf/blog/2013/07/)

June 2013 (https://www.entropywins.wtf/blog/2013/06/)

March 2012 (https://www.entropywins.wtf/blog/2012/03/)

December 2011 (https://www.entropywins.wtf/blog/2011/12/)

November 2011 (https://www.entropywins.wtf/blog/2011/11/)

September 2011 (https://www.entropywins.wtf/blog/2011/09/)

August 2011 (https://www.entropywins.wtf/blog/2011/08/)

July 2011 (https://www.entropywins.wtf/blog/2011/07/)

June 2011 (https://www.entropywins.wtf/blog/2011/06/)

May 2011 (https://www.entropywins.wtf/blog/2011/05/)

February 2011 (https://www.entropywins.wtf/blog/2011/02/)

January 2011 (https://www.entropywins.wtf/blog/2011/01/)

December 2010 (https://www.entropywins.wtf/blog/2010/12/)

October 2010 (https://www.entropywins.wtf/blog/2010/10/)

August 2010 (https://www.entropywins.wtf/blog/2010/08/)

July 2010 (https://www.entropywins.wtf/blog/2010/07/)

June 2010 (https://www.entropywins.wtf/blog/2010/06/)

May 2010 (https://www.entropywins.wtf/blog/2010/05/)

April 2010 (https://www.entropywins.wtf/blog/2010/04/)

March 2010 (https://www.entropywins.wtf/blog/2010/03/)

February 2010 (https://www.entropywins.wtf/blog/2010/02/)

January 2010 (https://www.entropywins.wtf/blog/2010/01/)

December 2009 (https://www.entropywins.wtf/blog/2009/12/)

November 2009 (https://www.entropywins.wtf/blog/2009/11/)

October 2009 (https://www.entropywins.wtf/blog/2009/10/)

September 2009 (https://www.entropywins.wtf/blog/2009/09/)

August 2009 (https://www.entropywins.wtf/blog/2009/08/)

July 2009 (https://www.entropywins.wtf/blog/2009/07/)

June 2009 (https://www.entropywins.wtf/blog/2009/06/)

May 2009 (https://www.entropywins.wtf/blog/2009/05/)

April 2009 (https://www.entropywins.wtf/blog/2009/04/)