November 17, 2022 · 12 mins, 2275 words

CRUDdy By Design by Adam Wathan: Summary, Examples, Opinions

Back in 2017, Adam Wathan made a great talk at Laracon called CRUDdy By Design, which has been recommended and reshared online for years. I decided to summarize it in text form, so instead of watching all 40 minutes, you could read this quicker, also with more examples and my own opinion.

First, I do recommend watching the full video anyway, embedded below.

Let's consider this article just a "compact summary", but if you want the full version with details, those 40 minutes are still worth it.

And before you ask: yes, that video is from 2017. And yes, it's still hugely relevant, despite some small syntax changes.

The Main Point: Everything is CRUD

Let's get straight to the primary point of Adam's talk.

Instead of creating custom non-resourceful methods in Controllers, you should create separate new Controllers with resourceful methods.

A typical example would be a `PodcastController` which has typical methods like `index()`, `store()`, and others, and you want to add a new method `subscribe()`.

The problem with the `subscribe()` method is that it is a **custom** action: with adding those, such Controllers may soon grow too big and get hard to read/maintain.

Instead, according to Adam, you should create a `SubscriptionController` or `PodcastSubscriptionController` with the method `store()`.

If we limit each Controller to **only** CRUDdy methods, it's easier to maintain.

CRUDdy methods are the same ones from `Route::resource()` in Resource Controllers:

- index()
- create()
- store()
- show()
- edit()
- update()
- destroy()

In other words, if you want to create a method that is not in the list above, there's a chance of better creating a new separate Controller with one of those methods above, thinking about which of those 7 methods fits best.

Or, even shorter, **every Controller should be a Resource Controller**.

Repeating the same thing in Adam's words: **Never Write Custom Actions**.

Three Examples From Adam's Talk

Notice: I will be rephrasing/summarizing examples a bit, and will not go 100% with the video.

Example 1. Nested Resource Controllers

Let's take a look at this example:

```
class PodcastController extends Controller {
    public function index() { /* ... */ }

    public function create() { /* ... */ }

    // ...

    public function listEpisodes($id) {
        // list episodes of a specific podcast
    }
}
```

Then, in the Routes:

```
Route::get('podcasts/{id}/episodes',
    [PodcastController::class, 'listEpisodes']);
Route::resource('podcasts', PodcastController::class);
```

According to Adam, listing the children records should have its own method in its own Controller, specifically for **episodes**. And not even that, to manage the CRUD of episodes

by a particular podcast, we should have a **PodcastEpisodeController**:

```
class PodcastEpisodeController extends Controller {
    public function index(Podcast $podcast) { /* ... */ }

    public function create(Podcast $podcast) { /* ... */ }

    public function store(Podcast $podcast, Request $request) { /* ... */ }

    public function show(Podcast $podcast, Episode $episode) { /* ... */ }

    public function edit(Podcast $podcast, Episode $episode) { /* ... */ }

    public function update(Podcast $podcast, Episode $episode) { /* ... */ }

    public function destroy(Podcast $podcast, Episode $episode) { /* ... */ }
}
```

As you can see, `Podcast $podcast` is a parameter in all the methods, and it will be automatically resolved if you use this function in Routes:

```
Route::resource('podcasts', PodcastController::class);
Route::resource('podcasts.episodes', PodcastEpisodeController::class);
```

You can read more about so-called Nested Resource Controllers in the official Laravel docs.

Example 2. Resource Controller for Pivot Table

Next, Adam is discussing a case of **subscribing** to the podcast. If we follow a non-CRUDdy way, it would look like this:

```
class PodcastController extends Controller {
    // ... other methods

    public function subscribe($podcastId) {
```

```
        // Pivot table "podcast_user" operation
        auth()->user()->podcasts()->attach($podcastId);

        return redirect()->route('podcasts.index');
    }
}

// Routes file
Route::post('podcasts/{podcastId}/subscribe', [
    PodcastController::class, 'subscribe'
]);
```

Looks good, right? But again, to avoid too many custom action methods in the Controller, let's think about how we can make it into a CRUDdy method.

We're actually creating a new record in the "podcast_user" table. So, it should be a `store()` method instead. But store of what? PodcastUserController?

Not necessarily. What if we stick to the same names as things are called on the page, and call it a **Subscription**, instead of a pivot table "podcast_user"?

It's actually a set of changes:

- We rename the pivot table from "podcast_user" to "subscriptions"
- We create a separate Eloquent Model for it called Subscription
- And then, we create a SubscriptionController to store this data

```
class SubscriptionController extends Controller {
    public function store(Request $request) {
        Subscription::create([
            'user_id' => auth()->id(),
            'podcast_id' => $request->podcast_id,
            // ... maybe more fields potentially
        ]);

        return redirect()->route('podcasts.index');
    }
}

// Routes file
```

```
Route::post('subscriptions', [
    SubscriptionController::class, 'store'
]);
```

This is quite a fundamental change, as Adam points out. In this case, we actually **discovered a new entity** in our project. We rephrased it from being just an intermediate not-so-important table to a proper Model which we can work with in the future, with more functionality.

And again, we transformed the code to being CRUDdy.

Example 3. Publish, Update, or New Controller?

Let's imagine you have an endpoint to **publish** the podcast. The straightforward way would be this:

```
class PodcastController extends Controller {
    // ... other methods

    public function publish(Podcast $podcast) {
        $podcast->update(['published_at' => now()]);
        return redirect()->route('podcasts.index');
    }
}

// Route file:
Route::put('podcasts/{podcast}/publish',
    [PodcastController::class, 'publish']);
```

But this is where we introduce the **custom** action and violate the CRUDdy principle.

Let's think about what CRUDdy method would be the best fit for this action.

Inside our function, we're **updating** the record, right? So why don't we do this? Then, we also can use the `update()` method for UNpublishing the podcast.

```
class PodcastController extends Controller {
```

```
    // ... other methods

    public function update(Podcast $podcast, UpdatePodcastRequest $request) {
        if ($request->published_at) {
            $podcast->update([
                'published_at' => $request->published_at
            ]);
        } else if ($request->unpublish) {
            $podcast->update([
                'published_at' => NULL
            ]);
        } else {
            $podcast->update($request->validated());
        }

        return redirect()->route('podcasts.index');
    }
}

// Route file:
Route::resource('podcasts', PodcastController::class);
```

This way, we don't even need a separate method of `publish()`, and also we can pass `published_at` as a PUT parameter, to possibly publish the podcast in the future/past. Looks cool, right?

But this is where Adam suggests we make a step further.

Notice: Personally, I disagree with the following approach below, but will still explain it.

The problem with this is that we violate a single responsibility principle – we have one `update()` method for a totally different purpose: visually on the page, Publish will be just one button, and Update may be a huge form with 10+ fields.

So what Adam is suggesting is to treat the published podcast as a separate resource, with its own resource controller like **PublishedPodcastController**. If we transform it into the CRUDdy logic, we're **storing** the fact that the podcast is published, or **destroying** that fact:

```
class PublishedPodcastController extends Controller {
    public function store(Podcast $podcast, Request $request) {
```

```
        $podcast->update([
            'published_at' => $request->published_at
        ]);

        return redirect()->route('podcasts.index');
    }

    public function destroy(Podcast $podcast) {
        $podcast->update([
            'published_at' => NULL
        ]);

        return redirect()->route('podcasts.index');
    }
}

// Route file:
Route::post('podcasts/{podcast}/publish',
    [PublishedPodcastController::class, 'store']);
Route::delete('podcasts/{podcast}/unpublish',
    [PublishedPodcastController::class, 'destroy']);
```

This way, we have separated the CRUD of Podcast and its own CRUD of Published Podcast. So we have a "virtual resource" of Published Podcast, although we don't store it like this directly in the database or Eloquent model.

And this is where personally I disagree with this approach of Adam.

What if we need more similar updates? Not only Publish, but also Activate/Deactivate, Make Premium/Free, or something else. Will we create controllers like ActivatedPodcastController, PremiumPodcastController, and others? It just doesn't feel right to me.

Also, although we have fewer methods in each Controller, which is a noble goal, it doesn't feel natural to read. A new developer, while looking at PublishedPodcastController, may assume that there should be a DB table `published_podcasts` or Eloquent model `PublishedPodcast`, which is wrong.

So, in my personal opinion, separating state-based updates as separate Controllers introduces more confusion than adds benefits.

What I would do? Probably would stick to the `update()` method, just separated the Controller logic to some Service method which would take care of different updates, depending on the request. Something like this:

```
class PodcastController extends Controller {
    // ... other methods

    public function update(Podcast $podcast,
                           UpdatePodcastRequest $request,
                           PodcastService $podcastService) {

        // All the logic of publish/update/etc should be inside the Service
        $podcastService->update($podcast, $request);

        return redirect()->route('podcasts.index');
    }
}
```

This way, we still stay CRUDdy, and Controllers are pretty manageable.

Another similar example Adam provided is updating a cover image for the podcast. In the same fashion, it becomes a separate Controller – `PodcastCoverImageController` with the method `update()`. Not sure how I feel about this, I would probably leave it in the same `update()` for the `PodcastController` – again, this is just my personal opinion.

A Step Further: Invokable Controllers

Adam's idea, which actually originated from Basecamp source code analysis, was to have the lowest amount of methods per Controller, increasing the number of Controllers themselves.

So, shorter Controllers were the goal, right? Then, why wouldn't we use Single Action controllers?

I'm not entirely sure if those didn't exist in Laravel at the time of Adam's talk, but we could easily transform the idea of, for example, publishing a podcast, into this:

```
class PodcastPublishController extends Controller {

    public function __invoke(Podcast $podcast) {
        $podcast->update([
            'published_at' => $request->published_at
        ]);

        return redirect()->route('podcasts.index');
    }
}

// Route file:
Route::post('podcasts/{podcast}/publish',
    PodcastPublishController::class);
```

Similarly, we could transform some/all custom actions mentioned in the talk, to their own "invokable" Single Action Controllers.

Would it work? Yes. Would it lower the ratio of average-methods-per-controller? Absolutely. Would it be a better solution? Sometimes.

Personally, I rarely used such Controllers, just because I tend to think that a Controller is a set of methods around one entity or Eloquent Model, and it makes more sense to me to have multiple actions in Controller, than just artificially limit my code to only ONE method per Controller.

But, as with many things in coding, it's your personal preference. If you like these action-based Controllers, feel free to use them more often.

Criticism: The World is NOT CRUDdy?

We've covered the talk with examples, and you should understand the main idea by now. We kinda "force" our app to be a CRUD, even though it doesn't all look like CRUD from the beginning.

So, aren't there some cases where this logic would NOT work?

If you search for "cruddy by design" on Reddit, you will find quite a few discussions with a skeptical look at this pattern:

- Cruddy by design, bad?
- Is Cruddy by Design unfeasible when dealing with antithetical actions such as approve/reject?
- "Cruddy by Design" – Adam Wathan – Laracon US 2017

There's also a YouTube video from another Laracon, by Adel, called "The world is NOT cruddy by design":

That talk for me personally didn't deliver the message and failed to provide a valid counter-message to Adam's talk. But, along with the Reddit topics above, it raises a valid question: *"What if we're trying to force a CRUD on the world which is not exactly a CRUD in real life?"*

Example questions from Reddit:

- we make *everything* look like CRUD. But where is the real CRUD? the real crud is obscured because of all of this fake CRUD. Where do we *actually* create a model? all controllers *look as if*. To figure out which controller *actually* does CRUD, you are now f*ed
- From my point of view, the world is not Cruddy by design. I just "publish" the podcast. Not create a publish-podcast pseudo-entity. User "registers" self in some app. Not create.

To that, there are the answers by other Reddit users, mostly stating one simple truth: **don't blindly follow any pattern**.

- Like most other programming concepts I don't waste trying to make every part of my application fit into that specific concept. I would never follow them as unbreakable rules (unless that's a client requirement) and use them more like core guidelines.
- I really like his cruddy talk, and it helped me on many projects. But like the other said, nothing is absolute. If it doesn't make sense to you, then there is no point.
- Short answer: do what feels right for you.

Conclusion/Opinion: To CRUD Or Not To CRUD?

In my experience, trying to stick to standards is almost always a good thing. If you use typical methods like store/edit/update, it would be much easier for **other** developers in the future to understand the code.

That said, if you're trying to artificially force those methods to be CRUDdy, by creating "fake" virtual resources and not attaching them to real Eloquent models, to me it's a dangerous path.

The world is complex. It's not just "CRUDdy or not CRUDdy". It's both. It's a mix. It also has multiple interpretations of the same thing. That's why sometimes programming is more like art than science: how can you deliver your message to the future generations (aka, future teammates, including yourself) so it would be still understood in the years ahead?

So, there's no right or wrong answer. Probably, the most often reply I give to multiple questions on my social media and email, applies here, too: *"It's your personal preference"*.

Al Nahian

5 hours ago

Now that's a great insight! Thanks for the article.

Leave a reply

Leave a comment

You can use Markdown

Comment

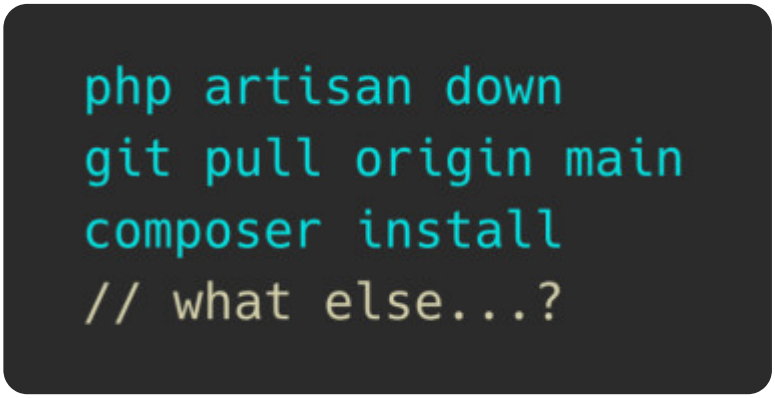
Recent Premium Tutorials

October 20, 2022 · 25 mins, 4982 words · PREMIUM

Laravel Breeze with User Areas: Student, Teacher, Admin

November 03, 2022 · 11 mins, 2046 words · ⭐ PREMIUM

Laravel: Simple Two-Factor Auth OTP via Email and SMS

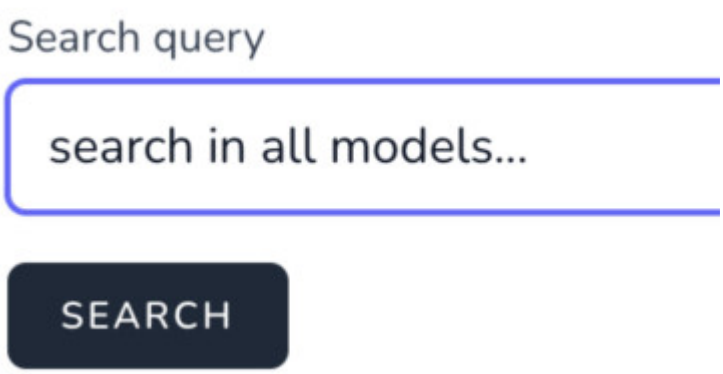


November 24, 2022 · 6 mins, 1121 words · ⭐ PREMIUM

Laravel Deployment Script: 4 Steps to Add Changes on Server

November 22, 2022 · 11 mins, 2077 words · ⭐ PREMIUM

Dealing With Money in Laravel/PHP: Best Practices



December 15, 2022 · 9 mins, 1747 words · ⭐ PREMIUM

Laravel Multiple Model Search: Queries, Scout, Packages

April 25, 2023 · 15 mins, 2867 words · ⭐ PREMIUM

Livewire Like/Dislike Component for Social Networks: Step-by-Step