php[architect]

May 2017
VOLUME 16

#200

APPROVED FOR PHP DEVELOPERS

Uncanny PHP

Visualization of Workflows in an Event Sourced Application

Look at the Vue From Here

Cryptography Best Practices in PHP

PHP Prepared Statements and MySQL Table Design

ALSO INSIDE

Education Station:
Qafoo Quality Analyzer

Leveling Up:
Code Review

Community Corner:
Become a Better Listener

Artisanal:
Project Creation

Security Corner:
An OWASP Update—The Top 10 for 2017

finally{}:
Happiness is a Boring Stack

GIANT ISSUE!

PHPARCH.COM

The polyglot webtech conference on a cruise ship!
Leaving from New Orleans and sailing the Caribbean.

July 16-23, 2017 — *Tickets $295*

www.codercruise.com

Sponsored by:

CISCO DEVNET

When I Work · Roave · CIOReview · VPSSERVER · Engine Yard · stickermule

Drupal Watchdog · ActiveState · concrete5 · php[architect] · contentful · GitHub

salesforce · php women · OmniTI · OSMI · INFINITE RED

CONTENTS

MAY 2017
Volume 16 - Issue 5

Uncanny PHP

Features

Columns

php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Uncanny PHP

Even before superhero movies were an annual summer staple, comic book heroes have been a part of my life. One of my earliest memories is of my dad taking me to the movie theater to see the first *Superman* movie with Christopher Reeve. I was born in Bolivia and we were still living there. Since I didn't speak English, much less read, I distinctly recall my Dad translating for me during the scene where Jonathan Kent counsels Clark on his destiny. As a kid, I would run around our house with a towel around me as a cape and later comic books were a welcome escape from the real world.

Because super hero's backgrounds and powers are often reimagined and updated, they've kept their appeal and place in our popular culture. One reboot of Spider-Man which had Peter Parker as a web developer at the Daily Bugle. Sure, not all of these ideas are great—like Batman's Shark Repellent. But the writers and artists are at least not afraid to try something new.

PHP's continual popularity, which you've read about in these pages many times, is due to a similar impetus to evolve. PHP 7 brought static and return type hints a while the later 5.x releases made it easier to use functional programming techniques. In its early days, PHP's popularity was boosted by how easy it was to extend and bring in new capabilities to your web applications. Even if some ideas, like `register_globals` and `magic_quotes` were the equivalent of shark repellent. As the language we use evolves, make sure you're staying relevant by expanding what's in your utility belt.

In this issue, Dustin Wheeler has an introduction to Workflows in *Visualization of Workflows in an Event Sourced Application*. In addition to writing about workflows, he shares a VueJS application to help visualize some examples. If you're not familiar with VueJS, John Congdon will help get you started in *Look at the Vue From Here*. Don't miss this article If you need to modernize your front-end skills. Check out *Cryptography Best Practices in PHP* by Enrico Zimuel to ensure you use PHP's cryptographic functions safely and correctly. Edward Barnard looks at when to bypass your ORM in *PHP Prepared Statements and MySQL Table Design*. See how to scale your application to handle high-volume workloads in a single leap.

Our columns this month are also packed full. In *Leveling Up: Code Review*, David Stockton summarizes the different ways you and your team can perform code reviews. More importantly, he'll share how to best integrate them into your day to day work. *Education Station: Qafoo Quality Analyzer* compliments David's column by looking at an easy-to-install application to help you automatically evaluate your code. In *Community Corner*, Cal Evans hands the reins over to Emily Stamey to write about how to *Become a Better Listener*. *Security Corner: An OWASP Update—The Top 10 for 2017* has Chris Cornutt looking at the proposed changes to the vulnerabilities web developers should be aware of. Joe Ferguson writes about starting a new Laravel project in *Artisanal: Project Creation*. See how to easily set up your routes, front end development, and testing suite. To cap off this issue, Eli White explains why *Happiness is a Boring Stack* in this month's *finally{}*. While new super powers are always fun, sometimes you need to stick to tried-and-true solutions for the problem at hand.

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

Stay in Touch

Don't miss out on conference, book, and special announcments. Make sure you're connected with us.

- Subscribe to our list: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download this Issue's Code Package:

http://phpa.me/May2017_code

Visualization of Workflows in an Event Sourced Application

Dustin Wheeler

Domain-driven design, Command Query Responsibility Segregation, and Event Sourcing are valuable strategies for modeling complex and chaotic domains, especially when interpretation of the domain may be evolving while the rules of it are well-known. In this article, we will investigate how CQRS/ES can be applied to visualize Workflow Management using PHP, Vue.js, and JointJS.

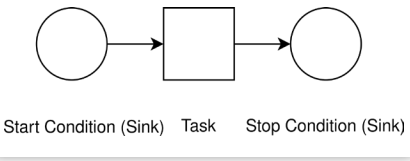
One of the projects I'm currently working on is a simplified Workflow Management System which can be used to orchestrate tasks required to provision arbitrary resources. Administrators author workflow nets specific to different types of resource actions (e.g. I may have a workflow for creating a WordPress site involving things like provisioning block storage, mounting it, downloading and configuring WordPress, configuring Apache, making requests to update DNS using an API, etc.). Also, during the life-cycle of that site, I may want to do other things to it: backups, retirement, etc. So we have **resources** (the WordPress site) which can have **actions** (create, backup, retire) performed on them. These higher-level **actions** are made up of individual **tasks** that must be orchestrated as part of workflow execution. We use workflow nets to describe the orchestration of **tasks** to carry out requested **resource actions**.

What Are Workflow Nets?

A workflow net is a specialization of Petri net (PT-net) used to describe business processes having a single defined beginning and end. A Petri net is a directed bipartite graph, in which nodes represent transitions and places. The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows). Connections between nodes of the same type are not allowed. Places are represented by circles and transitions by rectangles. Places may contain tokens, represented by colored dots. The placement of tokens throughout the graph is called a "marking."

There's more! Transitions are used to control the execution of a PT-net. Transitions "fire" when enabled. A transition is enabled if and only if every input place has at least one token. Firing a transition consumes one token from each input place and produces one token in each output place. Firing a transition may enable other transitions further down the graph, which causes them to fire. This process continues until there are no enabled transitions to fire.

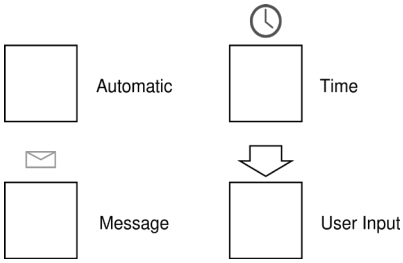
Figure 1.



Workflow nets (WF-net) place some additional constraints on the rules of a PT-net. First, a WF-net must have a single input and output place; a single beginning and ending. Additionally, the terminology is slightly different for transitions and places. In a WF-net, transitions are referred to as "tasks," and places are "conditions." The difference in terminology for transitions is important because tasks no longer simply fire when enabled. They must be triggered by some means:

- **Automatic**: The task behaves exactly as normal transitions in that they fire as soon as they are enabled. This trigger may be most useful in controlling process parallelization/synchronization.
- **User Input**: The task fires if enabled and if acceptable user input has been provided. That input may be added to "case attributes" which are key-values specific to an individual executing workflow. These might be used in routing a request.
- **Time**: The task fires after some amount of time has passed. This trigger is useful for preventing live-locking scenarios where a workflow may be stuck waiting on some worker that is sick!

Figure 2.



Building a Visualization

I know, it's a lot. Tactically, there are some learning curves which need to be flattened to a more easily digestible way of describing and debugging workflows. WF-nets are very easily visualized, and it would be awesome if we had a way of stepping through execution live. In the

short-term, something like this could support training, demonstration, and debugging of processes. In the long-term, visualization might support an over-arching monitoring goal; being able to see "hot-spots" in a process definition (where workers take longer than normal or are under significant load). This type of monitoring can directly influence process development for the better.

> *There are many routing primitives that can be used to implement different processes. Sequential routing allows for modeling processes where tasks have dependencies. Execution runs tasks in-order one after the other. Parallel routing allows specific tasks to be executed in parallel and synchronized before continuing. In the example above, both parallel tasks must complete before execution continues. Conditional routing allows exclusive OR-ing of tasks. Either one or the other will fire. If one task fires, then the tokens from its input condition are consumed, disabling the other tasks from firing.*

Ultimately, visualization of a `Workflow` is a projection of events occurring during the workflow's execution. That "story" might look something like:

1. `WorkflowStarted`: A process definition (the graph of conditions and tasks) is provided as well as the initial marking of the input place.
2. `TaskBecameEnabled`: An automatically triggered task is enabled.
3. `TaskWasFired`: The task fires in response.
4. `TaskBecameEnabled`: The firing of the previous task causes a user input triggered a task to become enabled.
5. `InputWasProvided`
6. `TaskWasFired`: When the task was enabled and input provided, it fires.
7. `WorkflowCompleted`: The workflow is complete when its output condition is marked.

This is the core narrative of a basic workflow. These events represent messages which are emitted by an executing `Workflow` aggregate to be projected into a read model (i.e. our JointJS visualization). Still, there remains the work of controlling the emission of these events as part of the domain. We'll cover both, starting with the domain.

> **Aside**: *In my experience, when working with event-sourced domains, I find it best to start with the events using a planning strategy like event storming. Doing so helps to form a contract of sorts between bounded contexts which can be used to split up work on your team.*

The Workflow Aggregate

> *The source for this project (in its entirety) can be found at mdwheele/phparch[1]. Implementation snippets will be kept brief for readability, but I encourage all to check out the project in its entirety.*

The workflow aggregate represents the transactional boundary for an executing workflow. An aggregate is a cluster of entities and value objects[2] that form a single logical concept. In this case, the workflow aggregate represents the places, conditions, arcs, and marking of a process definition as well as the behaviors which transform the workflow state over time. All aggregates will promote an entity of the aggregate to be the aggregate root. All references to the workflow aggregate will go through the workflow entity. In this way, the workflow ensures the integrity of the aggregate as a whole.

Aggregates can be modeled using some form of object-relational mapping to implement persistence between requests. In this way, patterns like Active Record or Data Mapper can be used to persist the aggregate root, and all relations would be encoded per implementation. Alternatively, aggregate state may be derived from domain

Figure 3.

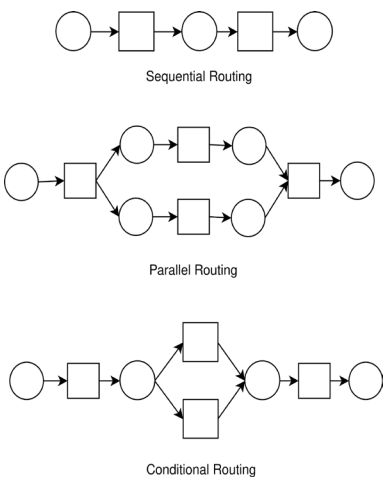
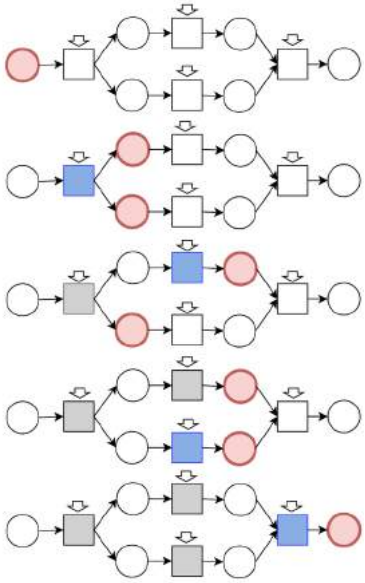


Figure 4.



Example of a workflow-net representing a parallel process where every task is triggered by user-input. As each task receives input (marked by blue) the task is fired, producing a new marking (marked by red). Tasks that have already fired are marked in grey for visibility purposes.

1 *mdwheele/phparch:* https://github.com/mdwheele/phparch-code

2 *entities and value objects:* http://phpa.me/Domain-driven-design

events having occurred during the aggregate's lifecycle (e.g., a workflow started, a task became enabled, a task fired, a workflow completed). This is how we will model our workflow.

Let's take a look at the functional pieces of an abstract `AggregateRoot`:

1. Afford the ability to record events which happen during a use-case.
2. Keep track of domain events that are uncommitted so they can be appended to an event store.
3. Afford the ability to reconstitute aggregates from a collection of domain events.

> *Keep in mind our implementation is purposefully simple to highlight some of the core concepts of Event Sourcing. There are several use-cases uncovered by this naive implementation (aggregate versioning / playhead, child events, etc.) which would be supported by a more robust framework (e.g. Broadway[3] .*

Recording Events

Implementations of `AggregateRoot` will expose a domain-specific API. Ultimately, clients of the aggregate send messages through method calls to that public API resulting in uncommitted domain events having to be appended to the event store. Internally, these methods will validate the incoming message against current state and if all is well, create a new event and `apply` it to mutate current state. Thus, we have an `apply` method:

```
# src/Support/AggregateRoot.php
protected function apply($event) {
   // We will cover the implementation of handle below.
   $this->handle($event);

   $this->uncommittedEvents[] = $event;
}
```

This method will take any domain event the Aggregate wishes to handle internally and if the event is handled successfully, appends it to an array of uncommitted events which can be retrieved.

Retrieving Uncommitted Events

Once an aggregate root has been acted upon during some use-case, the uncommitted domain events must be retrieved and persisted to an event store. This is implemented fairly simply:

```
# src/Support/AggregateRoot.php
public function getUncommittedEvents() {
   $events = $this->uncommittedEvents;
   $this->uncommittedEvents = [];

   return $events;
}
```

3 *Broadway: https://github.com/broadway/broadway)*

Reconstituting Aggregate State

Finally, we must be able to reconstitute aggregate state from domain events retrieved from an event store. This is implemented differently in many systems; usually depending on application-specific demands. That said, all implementations rely on some part of the public interface of the aggregate root receiving an iterable collection of domain events and building up state through processing those events one by one. Simply implemented, it can look like this:

```
# src/Support/AggregateRoot.php
public function initializeState(array $events) {
   foreach ($events as $event) {
      $this->handle($event);
   }
}
```

This implementation expects a factory to create an instance of the aggregate (injecting dependencies through the constructor, if needed) and then calling `initializeState` with an `array` of domain events. We then simply loop over them and delegate to `handle` for processing. Note that we do **not** append events as "uncommitted" because these are events which have already occurred. We're "re-hydrating" aggregate state based on those events to be able to exercise the model further and possibly produce more events.

Let's finally take a look at the `handle` method and then jump into the implementation of our `Workflow` aggregate. See Listing 1.

Listing 1

```
1. private function handle($event) {
2.    $method = $this->getApplyMethod($event);
3.
4.    if (!method_exists($this, $method)) {
5.       return;
6.    }
7.
8.    $this->$method($event);
9. }
```

Given any event, the `handle` method will attempt to map the object to a method that will mutate aggregate state. For example, if it was given an event named `WorkflowStarted`, `handle` might delegate to a private method named `applyWorkflowStarted` which would receive the event and update state. It is important these "apply methods" update state without exception. You do **not** want to be checking business rules inside these methods. Keep that work in the public API of your aggregate root and handle those concerns *before* applying new domain events.

Consider a case where you have a running model that guarantees certain invariants (business rules) are maintained. We're building up a history of facts that occurred (events) over time. Then, out of left field, the rules change! If I guarantee invariants in my apply methods, then there is a possibility

I won't be able to load past events to build up state because they aren't "valid" anymore. This gets into more advanced Event Sourcing topics which are out of the scope of this introductory article but are the main focus of event versioning patterns (type-based, weak schema, negotiated, copy-replace, etc.). In short, I cannot reiterate enough the importance of keeping methods that apply events to current state singularly focused on updating aggregate state used to enforce business rules.

Implementing the Workflow Aggregate

We will begin our implementation with a unit test which demonstrates how a workflow aggregate instance might be created as shown in Listing 2.

First, our implementation must provide a factory method to `start` a workflow and must also apply a single `WorkflowStarted` event used to encapsulate aggregate identity through a "case number" (domain concept specific to workflow). If both of these requirements are met, we assert the model is correct (Listing 3).

Our factory method creates an instance of `Workflow`, applies the `WorkflowStarted` event (which is given the generated "case number" as a UUID) and then returns the instance. Our apply method is very simple. It assigns `caseNumber` to fulfill the `getId` contract of `AggregateRoot`. At this point, a class responsible for persistence could call `getUncommittedEvents` and commit those events to an event store.

We also need to verify we can reconstitute a `Workflow` aggregate from a collection of events. In Listing 4, let's write a test case for that.

We have created a workflow instance from scratch, saved its uncommitted events to an event store and reloaded events to rehydrate our aggregate. With that out of the way, we can focus on the business rules governing workflow execution.

In practice, the workflow represents a specific execution of a process definition. This means our factory method should be updated to accept a `ProcessDefinition`. A process definition is the graph of conditions and tasks linked together with a single beginning (source) and end (sink). We will assume our process definitions are created for us, and we can just interact with them. In practice, there are tools specifically built to manage these definitions to form a Process Catalog. When a workflow is started, the source is marked, and output tasks from that source may become enabled. In response to being enabled, the task may fire, and if the output condition of that task is the process' sink, then the process will end.

Listing 2

```
1. public function workflows_can_be_started_and_are_assigned_a_generated_case_number(
2. ) {
3.     $workflow = Workflow::start();
4.
5.     $events = $workflow->getUncommittedEvents();
6.     $this->assertCount(1, $events);
7.
8.     $event = $events[0];
9.
10.    $this->assertInstanceOf(Workflow::class, $workflow);
11.    $this->assertInstanceOf(WorkflowStarted::class, $event);
12.    $this->assertEquals(
13.        $workflow->getId(), $event->caseNumber
14.    );
15. }
```

Listing 3

```
1. public static function start() {
2.     $instance = new static();
3.
4.     // WorkflowStarted is a POPO (Plain Old PHP Object) with a
5.     // single public property, $caseNumber, which is a Uuid.
6.     $instance->apply(new WorkflowStarted(Uuid::make());
7.
8.     return $instance;
9. }
10.
11. private function applyWorkflowStarted(WorkflowStarted $event) {
12.     $this->caseNumber = $event->caseNumber;
13. }
```

Listing 4

```
1. /** @test */
2. public function is_rehydrated_by_past_events_loaded_from_event_store(
3. ) {
4.     $caseNumber = 'some-case-number';
5.
6.     $events = [
7.         new WorkflowStarted($caseNumber)
8.     ];
9.
10.    $workflow = new Workflow();
11.    $workflow->initializeState($events);
12.
13.    $this->assertEquals($caseNumber, $workflow->getId());
14.    $this->assertCount(0, $workflow->getUncommittedEvents());
15. }
```

Listing 5

```
/** @test */
public function
the_most_basic_process_definition_executes_to_completion(
) {
    /*
     * The most basic process definition you can have:
     *
     *   C (source) -> Task -> C (sink)
     *
     */
    $definition = ProcessCatalog::basic();

    $case = Workflow::start($definition);

    $events = $case->getUncommittedEvents();

    $this->assertCount(4, $events);
    $this->assertEventStream(
        $events, [
        WorkflowStarted::class,
        WorkflowTaskEnabled::class,
        WorkflowTaskFired::class,
        WorkflowCompleted::class
    ]
    );
}
```

Listing 6

```
public static function start(ProcessDefinition $definition) {
    $instance = new Workflow();
    $marking = new Marking();
    $marking->mark($definition->getSource());
    $attributes = new AttributeStore();

    $instance->apply(
        new WorkflowStarted(
            Uuid::make(), $definition, $marking, $attributes
        )
    );

    foreach (
        $instance->getTasksEnabledBy(
            $definition->getSource()
        ) as $task
    ) {
        $instance->apply(
            new WorkflowTaskEnabled(
                $instance->caseNumber, $task->getId(),
                $task->getTriggerType()
            )
        );
    }

    $instance->tick();

    return $instance;
}
```

Let's code that up (see Listing 5), starting with a unit test.

When the workflow is started, the source condition is marked. This enables the only task in the definition, which will then immediately fire in response. When the task fires, it consumes a token from the source and produces one in the sink. Because the sink is marked, the workflow is completed. We assert there are four uncommitted events and they occur in a specific order. We could also make assertions on the contents of the Domain Events, but this is good enough, for now.

Our factory method requires quite a few changes. First, we must now track the process definition we're executing. We must also track the current marking as well as case attributes (which are runtime key-value stores which can be used for routing) as in Listing 6.

There's a lot going on here. A process definition is provided to start a workflow. We mark the workflow with the source of the process definition. The process definition, initial marking, and attributes are now captured by the `WorkflowStarted` event. After we apply `WorkflowStarted`, we must see if any tasks were immediately enabled as a result. If so, we apply a `WorkflowTaskEnabled` event. Note there is no matching "apply method" for this event. This is because this is an informational event which does not represent state change but may be interesting for clients of the workflow aggregate to consume. After checking for immediately enabled tasks, we instruct the workflow to `tick`. "Tick" is simply a means of having the workflow execute continuously until either there are no more enabled tasks which can fire, or the workflow is completed. In our example, the workflow's only task will immediately fire when `tick` is called (producing a `WorkflowTaskFired` event) and produce a token in the Workflow's Sink condition. The workflow is then completed and produces the `WorkflowCompleted` event.

Domain events aren't necessarily limited to the properties you place on them. If used to resolve state, they need to encapsulate the state transition caused by some behavior. You can "fatten" domain events with related information which may be interesting to consumers outside the aggregate. Because of this, it is important to consider consumers of these events in addition to the bare minimum for tracking aggregate state. Remember the state we track on aggregates is only used to guarantee domain invariants are maintained. Aggregates are part of the "write model." The separation of state between models can be a hard habit to break for many used to having one model of a domain (be it based on Active Record or Data Mapper).

Read Models

Almost uselessly put, a read model is a **model** specialized for **reads** rather than **writes**. However, that isn't the most useful definition. Think of a read model as the current *understanding* or *interpretation* of events which have occurred in the past. When we think of why software systems change, it

is usually because our interpretation shifts over time; **not** the rules that govern state change.

By separating these models conceptually, we are afforded an ability to implement models using technologies that make sense rather than attempting to shoehorn everything as a single monolith. For example, I may have a read model which provides a full-text search interface over employee records. That may be best implemented using something like Elasticsearch. My Elasticsearch-based read model becomes a projection of events having occurred in the past. Over time let's say we find the way we search that data is more about relations between records instead of simple full-text search.

Time passes, and now we want to find all actors from California who were in movies with Kevin Bacon and also attended chemistry classes with Michael Keaton in freshman year of college. If I had that ridiculous search requirement, I might choose to build my read model using Neo4j, a graph database. In either case, the events are the source of truth, and I *project* them into a read model which is a specific interpretation of facts having occurred in the past, optimized for the types of queries I currently need to make.

The "Simulation" Read Model

In our case, we wish to present a simulation of a workflow that is executing. A simulation is made up of several steps which can be paged through iteratively. These represent workflow progression visually and provide us a means of inspecting workflows while they're executing. This is important in diagnosing mistakes in process definition or routing rules. Our `Simulation` read model, shown in Listing 7, is a POPO (Plain Old PHP Object) that tracks a set of steps.

There is no requirement that a read model be a POPO. You may elect to use something like Doctrine to persist your model (in which case your read model would be a Doctrine entity), or you might decide to use Eloquent, an implementation of Active Record. Read models aren't terribly complex. It's just about providing easy access to state and fulfilling query requirements. In fact, you may decide your read model is the HTTP interface of an Elasticsearch cluster, and a projector simply populates that index using the same HTTP interface. It's up to you! *Our* example projects events into an in-memory plain ol' PHP object.

A projector is responsible for applying domain events from an event store to a particular read model. The technical means to accomplish this are very similar to the `handle` methods in our `AggregateRoot`. In fact, aggregate state can be thought of as a private read model used to validate business rules! Mind-blowing.

Here is what using our `SimulationProjector` might look like:

```
$simulation = new Simulation;
$projector = new SimulationProjec-
tor($simulation);

foreach (events_from_store() as $event) {
   $projector->handle($event);
}
```

Inside the `SimulationProjector`, apply methods similar to what we saw in the `Workflow` aggregate exercise and populate our `Simulation` read model (see Listing 8).

The rest of our workflow events are handled by other apply methods on `SimulationProjector`. Ultimately, we build up a `Simulation` that has retrievable steps we can iterate over. At this point, we can build a user interface which allows us to page over steps. In the sample project, I have built two types of user-interface. The first uses data URI encoded images (implemented via `SimulationProjector#screenshot`). There is a Twig template which makes a call to `Simulation#getSteps` and just renders out each step with attached information, including an image of current state. The other uses a serialization of Workflow state to expose a JSON-based API that is consumed by Vue.js integrated with JointJS to render a user interface. The rest of the article is focused on the latter interface, but I encourage you to check out the Twig-based implementation.

JSON-Based API, Vue.js, and JointJs Interface

First, we need a controller method which will produce a JSON document that represents our simulation. See

Listing 7

```
1. class Simulation
2. {
3.     private $steps = [];
4.
5.     public function add(Step $step) {
6.         $this->steps[] = $step;
7.     }
8.
9.     /**
10.     * @return Step[]
11.     */
12.    public function getSteps() {
13.        return $this->steps;
14.    }
15. }
```

Listing 8

```
1. protected function applyWorkflowStarted(WorkflowStarted $event) {
2.     $this->simulation->add(
3.         new Step(
4.             "Workflow Started",
5.             "The workflow case number is <kbd>{$event->caseNumber}</kbd>.",
6.             $this->screenshot(),
7.             $this->graph(),
8.             $event->attributes
9.         )
10.    );
11. }
```

Listing 9

```
1. public function showBasicSimulationJson() {
2.     $definition = ProcessDefinitionFactory::basic();
3.     $case = Workflow::start($definition);
4.     $events = $case->getUncommittedEvents();
5.
6.     $simulation = new Simulation();
7.
8.     $projector = new SimulationProjector($simulation);
9.     foreach ($events as $event) {
10.        $projector->handle($event);
11.    }
12.
13.    return new JsonResponse($simulation->asArray());
14. }
```

Listing 9.

Normally, our `SimulationProjector` would receive events as they occur during application execution and immediately persist them to a database. Controllers would simply ask the read model for a particular instance and it would re-hydrate from whatever is backing it. To keep the sample project simple, everything is done in memory (to not require a database to run). With that said, we do a little bit of work on the top of the controller method to execute a workflow, gather events and project them into our read model.

We return a JSON response capturing a collection of simulation steps. Next, we need to set up a Vue component that will query the endpoint. In the sample project, there are three Vue components:

1. `WorkflowSimulation` is responsible for providing the primary user interface and querying the API.
2. `GraphvizVisualization` is responsible for rendering Graphviz-based imagery based on steps received from the parent component
3. `JointVisualization` is responsible for integrating Vue with JointJS and receiving individual steps to render, similar to `GraphvizVisualization`.

The `WorkflowSimulation` allows us to choose between different simulation examples (simple vs complex) as well as how they are visualized (Graphviz vs. JointJS). When the simulation is changed in the user interface, it needs to query the API to get new data. We watch an attribute named `simulation` to determine when to pull new data using axios, a simple HTTP client:

```
// resources/assets/js/components/WorkflowSimulation.vue
watch: {
   simulation(sim) {
      axios.get('/api/simulation/' + this.simulation)
           .then(response => {
              this.steps = response.data
           });
   }
},
```

The workflow simulation Vue component tracks pagination of steps and provides the current step to child components for visualization. We use a Vue.js `prop` for that:

```
<div v-if="step">
    <graphviz-visualization
       v-if="form.visualization == 'graphviz'" :step="step"
    ></graphviz-visualization>
    <joint-visualization
        v-if="form.visualization == 'jointjs'" :step="step"
    ></joint-visualization>
</div>
```

Depending on which `visualization` is chosen, the appropriate child component is rendered and receives the current `step`. The JointJS visualization is more interesting simply because it has to integrate with a third-party JavaScript library which expects to run on its own. The strategy used to integrate JointJS equally extends to how you might pull a jQuery plugin into your Vue projects.

JointJS uses a "graph" and "paper" metaphor in its API. "Graphs" represent a specific model (or data structure). This could be a PT-net, or flow chart, or any of a number of pre-fabricated patterns the project support as well as custom graphs. "Paper" represents the mapping to SVG. This includes things like the DOM element JointJS owns as well as modeling surface dimensions. You also have to pass in a `model` to render, which in our case is the graph we manipulate.

So, when our Vue component is `mounted`, we want to set all this up as in Listing 10.

We store `graph` and `paper` as attributes on our component. Then we initialize `joint.dia.Paper`, passing in our `graph` as well as specifying the element our `paper` is attached to. This effectively scopes JointJS interaction with the DOM to that element, alone.

Now, when we make changes to `this.graph` we can tell `this.paper` to re-render, and we'll see an updated visualization. The implementation of rendering is a little lengthy for the article, but is available in the sample project. The **most important** part of this snippet is the integration of Vue to JointJS. By hiding the messiness of interacting with JointJS behind a nice interface provided by our Vue component, calling code can quickly visualize data structures without having to know the details of working with JointJS.

In our implementation, when the parent component `simulation` changes, new `steps` are retrieved from the API and the current `step` provided to the visualization changes. We watch for changes to the `step` prop and when they occur, we re-render the graph. We do this in a fairly brute-force manner by calling `this.graph.clear()` and re-building from scratch, finally making a call to `this.paper` to scale content to fit the dimensions supplied when the component was mounted. Check out the sample project for more details.

Whew!

It's important to keep in mind a lot of this is based on white papers out of academia and, to be honest, many of the Workflow Management Systems I've worked with have a steep learning curve because they are somewhat far removed from concrete applications. Additionally, this article is **dense** with terminology. To help wrap everything up, let's work through a summary of the terminology covered and then discuss how Workflow Management can be applied to the famous WordPress "5-minute install."

Ultimately, there are three taxonomies:

1. The language of domain-driven design (e.g. aggregate (root), entity, domain events, value objects, application/domain services, etc.)
2. The language of CQRS/ES (e.g. read model, event listeners, projector, saga, process manager, etc.)
3. The language of Workflow Management (tasks, conditions, source, sink, triggers, workflow, etc.)

Domain-Driven Design (DDD)

These are the building blocks of a modeling strategy and don't have much to do with Workflow Management at all. This modeling language is not much different than something like Model-ViewController (MVC) or Action-Domain-Responder (ADR). What I mean is it describes a set of building blocks used to communicate models (not that DDD is an architecture for web applications). DDD is a set of patterns used to reason about and structure a model. It can be *applied* in many domains (e.g. Workflow Management, building a blog, conference management, etc.) Domain-driven design can be done without models being sourced from domain events. In these cases, you could use Doctrine or any off-the-shelf ORM to persist the model between use-cases. The "model," in this case, does not refer to a singular (e.g. post, comment, etc.) but instead refers to the conceptual model; a model of your domain.

Listing 10

```
// resources/assets/js/components/JointVisualization.vue
import joint from 'jointjs'
export default {
    data() {
        return {
            graph: null,
            paper: null,
        }
    },
    mounted() {
        this.graph = new joint.dia.Graph
        this.paper = new joint.dia.Paper({
            el: this.$el.querySelector('.jointjs-visualization'),
            width: this.$el.offsetWidth,
            height: this.$el.offsetHeight,
            model: this.graph,
            gridSize: 1
        })
    }
}
```

Command Query Responsibility Segregation and Event Sourcing (CQRS/ES)

At its core, CQRS is the idea that a different domain model is used for writes than is used for reads. The write model receives commands, and the read model receives queries. The terminology for command and query are lifted directly from Command Query Separation (CQS), which is the idea an object's interface is divided into methods that either: return a result without state change **or** change state but do not return results; but not both. After separating reads from writes, the question becomes, "How does the read model receive updates from use-cases executing against the write?" One solution is to introduce domain events to the write model that parts of the read model subscribe to. It is common to use Event Sourcing in the write model to track state of the model, whose domain events can then be published easily to the read model. Event Sourcing is nothing more than a persistence pattern similar to Active Record or Data Mapper. When implementing Event Sourcing, we don't persist the *current state* of the model. Instead, we persist all *changes* to that model which can be used to arrive at current state; similar to an audit log.

Workflow Management

Workflow Management can be an application of an "event-sourced domain driven model." A "workflow" can be modeled as an "aggregate" where the "root" of that aggregate is the workflow entity. Workflows orchestrate resources to execute smaller tasks to fulfill a larger goal. So in the WordPress "5-minute install", I may have several tasks I want to apply resources to:

- download_wp_package
- create_db
- upload_wp_to_web_server
- configure_db
- run_wordpress_install_script

We can build a very linear workflow net from these tasks:

Source -> download_wp_package (T) -> C -> create_db (T) -> C -> … -> Sink

Alternatively, we could build a WF-net that branches where parallel processing can be applied. For example, I can download WordPress and create a database at the same time, but I need to wait to configure the database until those are done as well as the files uploaded to a web server. Workflows can be used to describe the orchestration of tasks, but

do not necessarily concern themselves with **how** that work is carried out. This concern is commonly handled by some sort of "work item delegation" model which would be subscribed (and also influence) our "workflow execution" model.

To quickly describe how such an integration might work, consider that we might set all of the above tasks to be triggered by user input. Then, our work item delegation model would subscribe to workflow execution; listening for "user-input triggered tasks that have become enabled" (remember `WorkflowTaskEnabled?`). When one of these tasks becomes enabled during workflow execution, it means that resources are required to complete the task. When the task becomes enabled, the work item delegation model creates a new work item and delegates it to workers who are capable. For example, let's say "download_wp_package" became enabled. We may have automated workers that can download the package. The work item would be assigned to them, and when they check in for jobs, they'd see the new work item and start working. They report to the system they are claiming the work item and when they finish, they report they have completed the activity.

The Workflow Management model subscribes to certain events emitted from work item delegation. In this example, when Workflow Management hears a "work item was completed," it sees what task in the workflow the work item was for and provides input to that task, which then fires and moves workflow state forward. This back and forth continues on-and-on until the workflow is complete.

A benefit of the work item delegation model is that workers are abstract. A worker could be human or automated. Take "run_wordpress_install_script" as an example. That task could be delegated to an automated agent which runs Selenium or `wp-cli` to complete the install script for WordPress. Alternatively, we might only delegate "run_wordpress_install_script" to a human resource. When that work item is delegated, the human might get an email, log into the application and see a list of work items they need to complete. When they finish the install script, they'd check-off they did so, and workflow state moves forward.

Conclusion

That's it; now you know everything there is to know about DDD, CQRS, Event Sourcing, and every other acronym under the sun!

In all seriousness, our industry is filled with terminology, techniques, and everything that comes with it. Sometimes, it's hard to recognize the things we don't know we don't know. My hope is that I have exposed you to a lot of different ideas and that you will follow up. I love building software, and I love talking about it! Please reach out on Twitter if you have any questions.

Dustin Wheeler is a Software Engineer at NC State University in Raleigh, North Carolina. He is an avid practitioner of Domain Driven Design and Test Driven Development. When he's not at work, he's attempting to manage the chaos that is his Great Dane, Sadie. He's losing. @mdwheele

Look at the Vue From Here

John Congdon

Vue, more formally known as Vue.js, is touted as *The Progressive JavaScript Framework*. Like other JavaScript frameworks such as jQuery, its goal is to make coding in the browser easier for developers by abstracting away all of the differences in browsers.

Why Vue?

I was recently involved in a discussion with a group of developers trying to pitch the need for an updated JavaScript framework to a client. The client asked some very good questions, one of which was, "How do we decide on a framework and know that it will be around for years to come?" They've been down the road of implementing other code from places, only to have them age and not be supported well in new browsers.

They also don't want to pick "new and shiny" just for the sparkle it may have. Developers are notorious for jumping on a bandwagon for a short period and then jumping off when the next new hotness comes along. I've seen lots of projects with scattered libraries all over the place, including the same library with different versions in the same codebase because nobody wants to go through and update all of the old code using an older version.

So, why Vue.js[1]? The biggest concern to have in a project is for the libraries you use to be maintained and usable into the future. It appears that because the Laravel framework has adopted Vue.js in its LTS code, Vue is here for the long haul. There should be enough people that want and need Vue to succeed; you should have little concern over the longevity. Plus the feature set already available and stable is outstanding.

My goal for this article is to whet your appetite for this great framework. There is plenty of fantastic online material for you to dive deeper and take your knowledge to the next level.

I am an avid Ultimate Frisbee player and love playing pick-up games at lunch time. Unfortunately, just like any other team pickup game, you have to have enough players show up to have a game. I thought a great little app we can build quickly and will take advantage of Vue.js is one called "Game On" which allows players to let the group know if they are available to play or not.

> *You can get a working demo of the application from diegodevgroup/phparch-vue-demo[2]*

Getting Started

For ease, I am not going to use a PHP framework. Another thing I am going to do to keep things simple is to forego any authentication.

The first thing we need to do is get Vue into our page. There are multiple installation options to choose from. For simple development, I am going to use their CDN which always has the latest published code (version 2.2.6 as of this writing). However, once you are ready for production, you will probably want to install the code with npm to make sure the code you write will not break with a new release of the codebase.

We will simply add the following to our HTML file to get started:

```
<script src="https://unpkg.com/vue"></script>
```

Each Vue application instance is bound to a part of the DOM identified by the `el:` attribute.

Data Binding

The first high-level concept to wrap your mind around is data binding. It took me a while to understand the concept when I fist started hearing this term. But once you get it, you start to see the power of Vue.js.

Listing 1

```
<div id="game-on">
    <h1> {{ title }} </h1>

    <span>Current List of players:</span>
    <ul>
        <li v-for="player in players">{{ player.name
}}</li>
    </ul>
</div>

<script src="https://unpkg.com/vue"></script>
<script>
    var game_on = new Vue({
        el: '#game-on',
        data: {
        title: 'Game On?',
            players: [
                { name: 'John Congdon' },
                { name: 'Eric Van Johnson' },
                { name: 'Kevin Reeves' }
            ]
        }
    });
</script>
```

1 *Vue.js: https://vuejs.org*

2 *diegodevgroup/phparch-vue-demo: http://phpa.me/phparch-vue-demo*

You need to start thinking about your app in a data-centric fashion. Data binding is a way of tying pieces within your HTML to a data object. Examples to come.

I've always worried about building up HTML with my data, whereas Vue mostly separates those concerns. Try to get out of the mindset of `<div>`'s, `<ul>`'s, and `<li>`'s, etc. and ensure you have a solid data structure.

Listing 1 shows two examples of data binding. The data is included statically in the code for demonstration purposes. The first part to point out illustrates the use of binding simple text in the heading of the page to a `title` attribute in the data structure. The second part shows how to build a list from an array of data.

Don't let this contrived example fool you; this is just to show how data binding works. While you may not need to set the `<h1>` tag via data, imagine the possibility of changing that text on the fly. I am going to jump the gun here, but I am sure you can guess what this code snippet will do:

```
game_on.$data.title = "Game Today"`
```

Handling User Input

So far, all we've seen is displaying data that is in some object. We are going to add one more very contrived example for demonstration purposes before we get into building the application.

This is where data binding shines. We are going to add an input field to the bottom of our HTML inside the game-on div. Notice how the `<h1>` tag and the input both have "Game On?" as their text? Change the value of the input. Behind the scenes, Vue is changing the data in the Vue.js object because it is bound, and now that data has changed, so is the `<h1>` tag also bound to that same data element.

```
Change the title: <input v-model="title">
```

The v-* Directives

Much of the beauty of Vue is it is intuitive and often gets out of your way. Most of the directives start with v- and can usually be understood just by reading them.

We've seen two examples in Listing 1.

`v-for` is a simple loop directive which allows us to go through an array. It will repeat whatever element it is connected to for each of the records in the array.

`v-model` is used to create two-way binding to form input elements. If the form changes the value, it is reflected in the Vue app, and vice-versa. If the Vue app's data changes, the form input is updated to reflect that change.

You will see them used throughout this article.

Event Magic

So far, we have a very simple application which doesn't do much. Let's change that. Listing 2 breaks this application up just a little bit more. Again, I put everything into a single HTML file just for demonstration purposes. In a real application, you will want to use separate files, broken down into components.

Let's follow through the code, and I will point out how this is now working.

Looking at lines 5 and 6, you will see we have some new tags. These are Vue component tags that will be replaced as the application gets up and running.

Listing 2

```
1. <html>
2. <div id="game-on">
3.    <h1> {{ title }} </h1>
4.
5.    <demo-form v-on:add-player="addPlayerToList"></demo-form>
6.    <player-list v-bind:player_list="players"></player-list>
7. </div>
8.
9. <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
10. <script src="https://unpkg.com/vue"></script>
11. <script type="text/x-template" id="demo-form-template">
12.    <div>
13.       <div>Name: <input v-model="addedPlayerName"></div>
14.       <div>Playing: <select v-model="addedPlayingOption">
15.          <option value="0" selected="selected">No</option>
16.          <option value="1">Yes</option>
17.       </select></div>
18.       <button v-on:click="respond">Respond</button>
19.    </div>
20. </script>
21.
22. <script>
23.    Vue.component('demo-form', {
24.       template: '#demo-form-template',
25.       data: function () {
26.          return {addedPlayerName: '', addedPlayingOption: 1};
27.       },
28.       methods: {
29.          respond() {
30.             var newPlayer = {
31.                name: this.addedPlayerName,
32.                playing: this.addedPlayingOption
33.             };
34.             this.addedPlayerName = '';
35.             this.addedPlayingOption = 1;
36.             axios.post('/vue_save.php', newPlayer);
37.             this.$emit('add-player', newPlayer);
38.          }
39.       }
40.    });
```

For Complete Listing see this month's code archive.

`demo-form` will be the form we use to add new players to our list. While `player-list` will be just that, our list of players and whether they have selected to play or not.

In my opinion, this is one of the beauties of VueJS. Our HTML is very small, and we can see almost immediately what is going on. Of course, the better your component names, the easier it is to read. For example, from the Vue documentation, you can probably figure out what is going on here:

```
<div id="app">
  <app-nav></app-nav>
  <app-view>
    <app-sidebar></app-sidebar>
    <app-content></app-content>
  </app-view>
</div>
```

Within our application, we will have a navigation section, a sidebar of some sort, and the main content.

Now, back to Listing 2. Lines 9 and 10 include axios[3] and Vue from a CDN. Remember, in production, you will want to install these libraries locally and use the minified production version. axios will be used to do some AJAX calls, but you are free to use any library you want. If you already have jQuery on your page, then there is no reason to use axios. For our purposes, bringing in the entire jQuery library just to make AJAX calls is overkill.

Lines 11-20 show one way of creating Vue templates. By putting HTML inside of a script tag, it will not be rendered by default. And setting the type to something like "text/x-template" will keep it from being interpreted as JavaScript. So, now the script tag is acting just like a text holder that can be referenced later, as you will soon see.

Lines 23-40 set up our demo-form component. `Vue.component` takes two arguments, a name and a configuration object. Inside our configuration object, we include a template, which is just pointing to our script template above by an ID query selector. In Vue components, data is a function which returns a data object.

The `methods` property includes a function that responds to the button click in our form. The `respond` function does the following:

1. Builds a quick object from the form data to be posted to the server.
2. Resets the form input to be used again.
3. Uses axios to post the data back to the server.
4. And finally, emits an event.

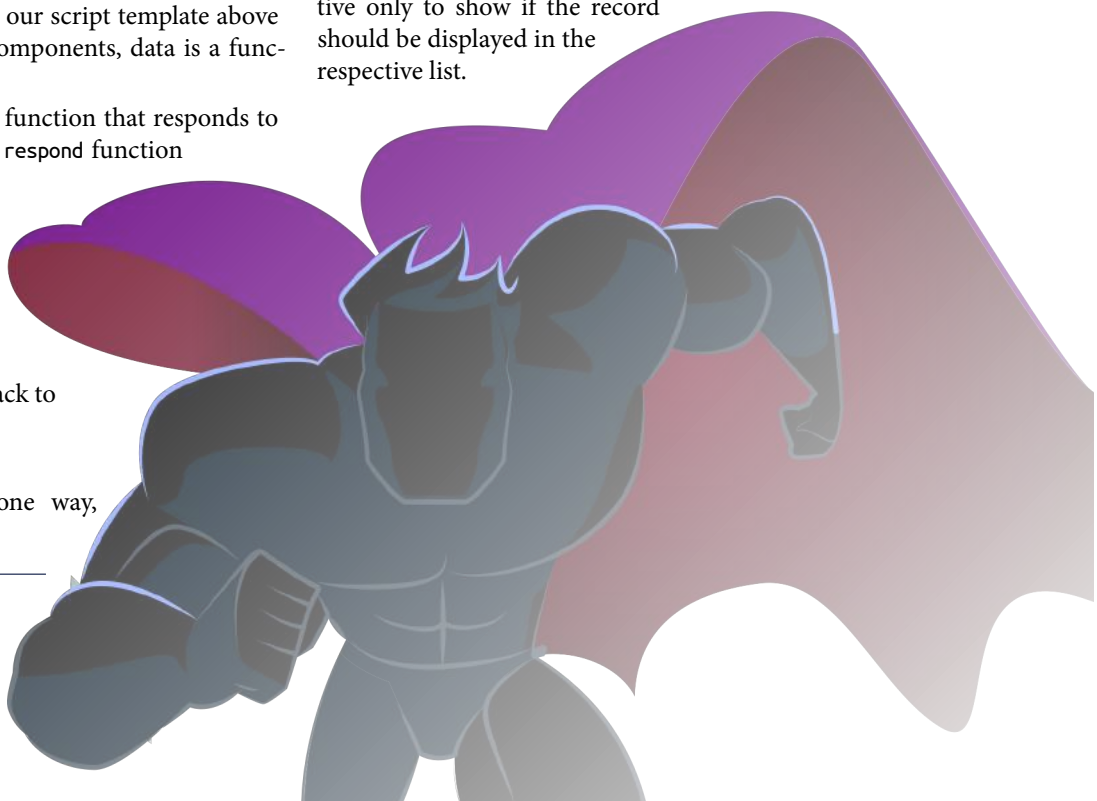
Component data binding is one way, from the application down to components via `props` (to be shown in the next component). The idea is to compartmentalize components, so they don't change the applications state. If you have lots of components, you don't want one of them to unknowingly change the data in the application and affect any of your other components. However, components can emit events, which the application can then catch and act on, which may be, in fact, as simple as changing application data. The difference in the latter scenario is there is more structure and purpose built when doing it this way.

Our `respond` method emits the `add-player` event. Now look more closely at line 5, which says when we see the event "add-player," call the `addPlayerToList` method in our application. This is a misnomer, in the sense I only allow the name to be on the list once. I want to demonstrate how updating data this way, also updates the list. So if the same name is used, we will just update the playing property. It's as simple as that; we'll see the `addPlayerToList` method shortly.

Our second component is the player-list on lines 41-61. This time, we are demonstrating how to include the template as a string on lines 42-59. Note that we are encasing the string in backticks "`" to make this a template literal because we are spanning multiple lines.

The last piece of this component are the `props`, in our case the player's data. This is the one-way binding from the application down to the component. Let's take another look at line 6 where we bind our application's players data to the components `player_list` data. These two can be the same (players), but I wanted to demonstrate how you could write a component completely separate from the rest of the application (or imported from someone else) and then just tie the application data to the component data.

One more thing to point out in the template. We are using the same data array in both lists and using the `v-if` directive only to show if the record should be displayed in the respective list.



3 *axios:* https://github.com/mzabriskie/axios

Finally we have our application code from lines 62-90. We are passing a configuration object into Vue. We've discussed the `el`, `data`, and `methods` properties above. The `created` item is just one of the many other configuration options you can pass in. It is part of the lifecycle hooks in Vue where the instance has been created, but not rendered yet. We use this opportunity to make an AJAX call via Axios to retrieve our player data. I could have just put the call directly into this function, but I also wanted to show you can call methods from your application. This should allow you to keep your code DRY. On a side note, you have the ability to inject code into any part of the lifecycle.

API Endpoints

You can use this code as is for a demo. You just need to provide the endpoints to get and save the data. For my demo, I had an endpoint which pulled the JSON out of a file, and another endpoint that saved the data back in. This was quick and dirty, but if you want all of the pieces to see this work, you need the code.

Listing 3 shows the endpoint for a `GET` request.

Listing 4 shows the endpoint for a `POST` request.

Conclusion

The goal was to give a basic overview of what Vue.js can do. There is so much more to learn to really become efficient. In my example, I used the longhand way of declaring directives, such as v-bind, but there is a more succinct and way of declaring v-bind and v-on events in Vue with shorthand notation. There is power in Vue I haven't even scratched the surface of myself, but luckily I have developers on my team that know Vue very well and can answer my questions quickly.

If you aren't as lucky as I am and you have a Laracasts account, I strongly recommend Jeffrey Way's *Learn Vue 2: Step By Step*[4]. If you don't have either, I strongly recommend you get a Laracasts account.

Resources

There are so many great sites for more reading and to continue learning.

1. Laracasts[5]
2. Vue.js Guide[6]
3. Develop Basic Web Apps with Vue.js[7]
4. VueX State Management Introduction[8]
5. Vue Cheatsheet[9]

Listing 3

```
1. header("Content-type: application/json");
2.
3. $data = file_get_contents('vue_data.js');
4. if ( !$data) {
5.    $data = '[]';
6. }
7.
8. echo $data;
```

Listing 4

```
1. <?php
2. $postdata          =
3.    json_decode(file_get_contents("php://input"));
4. $newPlayer         = new stdClass();
5. $newPlayer->name   = $postdata->name;
6. $newPlayer->playing = $postdata->playing;
7.
8. $data = json_decode(file_get_contents('vue_data.js'));
9. if (json_last_error() !== JSON_ERROR_NONE) {
10.    $data = [];
11. }
12.
13. $new = TRUE;
14. foreach ($data as $index => $player) {
15.    if ($player->name == $newPlayer->name) {
16.       $data[$index] = $newPlayer;
17.       $new          = FALSE;
18.    }
19. }
20. if ($new) {
21.    $data[] = $newPlayer;
22. }
23.
24. $fh = fopen('vue_data.js', 'w');
25. fputs($fh, json_encode($data));
26. fclose($fh);
```

John Congdon is CEO of DiegoDev Group, LLC, a web application development firm, in San Diego, CA. John is a co-organizer of San Diego PHP (SDPHP) and active member of the community. He is passionate about bringing his skills and those of the people around him to a higher level. @johncongdon

4 *Learn Vue 2: Step By Step:* https://laracasts.com/series/learn-vue-2-step-by-step

5 *Laracasts:* http://www.laracasts.com

6 *Vue.js Guide:* https://vuejs.org/v2/guide/

7 *Develop Basic Web Apps with Vue.js:* http://jrcon.me/vue-egghead

8 *VueX State Management Introduction:* http://jrcon.me/vuex-intro

9 *Vue Cheatsheet:* https://vuejs-tips.github.io/cheatsheet/

Cryptography Best Practices in PHP

Enrico Zimuel

How to use cryptography in PHP without getting hurt.

Security is a very important aspect of software development, particularly in a web application where every user can potentially be malicious. If you need to manage critical or sensitive data like passwords, API secret keys, credit card numbers, etc. you need to use cryptography.

But cryptography is very, very hard!

PHP as a language offers many cryptography tools, but the usage is not always simple. There are many open source libraries which can facilitate the cryptography implementation but you still need to know general best practices in order to use them.

In this article, I'll introduce some best practices for using cryptography in PHP.

Use Standard Algorithms

The first suggestion for cryptography is to **only use standard algorithms**. This is very important, never try to implement cryptography by yourself. Even if you are familiar with the topic, you are very likely to fail.

If cryptography is hard, its implementation in software is even harder. There are many examples in history where cryptographic protocols failed because of software implementation issues.

Perfect security does not exist in the real world. A fault or a new attack will be discovered sooner or later; it's only a matter of time. The only thing we can do to mitigate the risk of security issues is to use standard algorithms and protocols. They are the best solutions because they have been reviewed by many people over years of usage.

Most importantly, **always use open source algorithms for cryptography**. If you are using an unknown black box algorithm to protect your data, you are confusing secrecy with security. Secrecy only contributes to a false sense of security.

> *The argument that secrecy is good for security is naive, and always worth rebutting. Secrecy is beneficial to security only in limited circumstances, and certainly not with respect to vulnerability or reliability information. Secrets are fragile; once they're lost, they're lost forever. Security that relies on secrecy is also fragile; once secrecy is lost there's no way to recover security. Trying to base security on secrecy is simply bad design.*
>
> *– Bruce Schneier*

Which Standard Algorithms Should You Use?

Currently, I would recommend the following algorithms for general purpose usage.

- Symmetric-key algorithm: **AES**, a FIST 197 standard since 2001;
- Public-key algorithm: **RSA**, an industry standard algorithm used in many products;
- Hash function: SHA, in particular **SHA-256** or **SHA-512**; Note: do not use SHA-1 for cryptography!
- Key derivation algorithm: **PBKDF2** is a very popular algorithm (RFC 2898);
- Password storing: **bcrypt** is a very popular algorithm or **Argon2** the winner of Password Hashing Competition in July 2015

PHP supports all of these algorithms directly in the language or using the OpenSSL extension.

Most of these algorithms are also supported by the mcrypt extension, but I do not recommend it anymore because it has been deprecated from PHP 7.1.

If you want to use Argon2, you need to wait for PHP 7.2 or install the Halite project[1] which uses the libsodium crypto library.

Design With Security in Mind

This is more a mindset than practical advice but still important in daily work. Security is a process that should be considered in every single line of code, especially when related to cryptography.

> *A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.*
>
> *– Auguste Kerckhoffs*

This quote is very popular in cryptography, it's known as *Kerchoof's principle*. You should always assume an attacker knows every detail about your system.

> *The enemy knows the system.*
>
> *– Claude Shannon*

1 *Halite project:* https://github.com/paragonie/halite

Use bcrypt or Argon2 to Store Passwords

If you are using MD5 or SHA hashes to store users' passwords, **please don't do that**! Even if you salt the hash, you are not safe. A random salt can protect from a dictionary attack, but it cannot prevent brute-force attacks. Using a GPU (even a cheap one), you can decrypt passwords in seconds.

A reasonably secure way to store a password is to use the bcrypt algorithm. This one-way function can make brute-force attacks impractical because it is computationally slow. If someone wants to attack the algorithm, they need a lot of time (years) to generate all the values.

Starting with PHP 5.5 we can use two special functions for the bcrypt algorithm, they are `password_hash()` and `password_verify()`.

Here's an example of generating the hash value of a user's password:

```
$password='supersecretpassword';
$hash = password_hash($password,
PASSWORD_BCRYPT);
echo $hash;
```

This script will output something like this:

```
$2y$10$TZp7a29gDmtwa5Inch0Eq.INxx1tnjY9k-
6gWxwH/TUoX4uJYTxm76
```

The output of password_hash is a string of 60 bytes, with a header of seven bytes including the bcrypt specification (`$2y$`) and the cost parameter (`10$`). Each time you generate a hash, you will get a different output. This is because the algorithm automatically uses a random salt each time to improve the security.

To verify a hash value, we can use `password_verify()` with a given password:

```
echo password_verify($password, $hash)
    ? 'Ok' : 'Password incorrect';
```

You can also use the Argon2 hash function to protect your password. Argon2 is a key derivation function selected as the winner of the Password Hashing Competition in July 2015.

The Argon2 algorithm provides more security compared to bcrypt because it requires more RAM to generate the hash. That means an attacker will have more difficulty scaling a brute-force attack.

To use Argon2 in PHP, you need to install the libsodium extension. If you are interested in how to use it you can read Secure Password Storage in PHP[2].

PHP 7.2 will include Argon2 support directly in the core language. This means we will use it inside the `password_hash()` function using the `PASSWORD_ARGON2I` constant. Here is an example:

```
$hash = password_hash($password,
PASSWORD_ARGON2I);
```

Don't Use a Password as an Encryption Key

Never use a user's password as encryption key! A user's password is not random, and it doesn't have good entropy bits. Always use a key derivation function (KDF) to generate an encryption key starting from a user's password.

One of the most used key derivation function is the PBKDF2 algorithm.

PHP 5.5+ offers `hash_pbkdf2()` function for that.

Here is an example:

```
$password = 'supersecretpassword';
$salt = random_bytes(16);
$hash = hash_pbkdf2(
    "sha256", $password, $salt, 20000
);
var_dump($hash);
```

In this example, the `hash_pbkdf2()` function generates the hash values iterating SHA-256 20,000 times. The function also uses a random salt value that is very important for the security of the algorithm. I used the `random_bytes()` function of PHP 7. If you are using PHP 5, you can use the .

The output of the PBKDF2 is a string of 32 bytes in hex format (64 characters). If you want a binary string with a different size, you have to specify two additional parameters. For instance, if you need a hash value of 128 bytes in binary format you can use the following syntax:

```
$hash = hash_pbkdf2(
   "sha256", $password, $salt, 20000,
128, true
);
```

Another important parameter of the PBKDF2 algorithm is the number of rounds. I used the value 20,000 used in many applications; you should use the maximum number of rounds which is tolerable, performance-wise, in your application (e.g. LastPass uses 100,000 iterations).

Don't Use rand() or mt_rand()

The PHP functions `rand()` and `mt_rand()` do not generate cryptographically secure pseudo-random values. To be cryptographically secure a random number generator must pass tests of statistical randomness. Also, if part or all of its state is revealed, and attacker should still not be able to predict future results or reconstruct how previous values were generated.

The `rand()` function uses the `libc` library to generate pseudo-random numbers which is not secure for cryptography applications. It generates random numbers using a linear additive feedback method, with a short period, that is predictable.

Even the `mt_rand()` function is not secure from a cryptographically point of view. It uses the *Mersenne Twister* algorithm[3] to generate pseudo-random numbers. This function is faster than the rand(), and it produces pseudo-random numbers with a big period but is still deterministic, so it is predictable.

To generate a cryptographically secure random number with PHP 7 we can use `random_int()` or `random_bytes()` for a binary string.

If you are using PHP 5.x, you can use

2 *Secure Password Storage in PHP:* http://phpa.me/php-store-passwords

3 *Mersenne Twister algorithm:* http://phpa.me/mersenne-twister

Listing 1

```php
<?php
// Example of encrypt/decrypt functions with AES-256-CBC
// + HMAC-SHA256in encrypt-then-authenticate mode

function encrypt(string $text, string $key): string {
    $iv = random_bytes(16); // iv size for aes-256-cbc
    $keys = hash_pbkdf2('sha256', $key, $iv, 80000, 64, TRUE);
    $encKey = mb_substr($keys, 0, 32, '8bit');
    $hmacKey = mb_substr($keys, 32, NULL, '8bit');

    $ciphertext = openssl_encrypt(
        $text, 'aes-256-cbc', $encKey, OPENSSL_RAW_DATA, $iv
    );

    $hmac = hash_hmac('sha256', $iv . $ciphertext, $hmacKey);
    return $hmac . $iv . $ciphertext;
}

function decrypt(string $text, string $key): string {
    $hmac = mb_substr($text, 0, 64, '8bit');
    $iv = mb_substr($text, 64, 16, '8bit');
    $ciphertext = mb_substr($text, 80, NULL, '8bit');

    $keys = hash_pbkdf2('sha256', $key, $iv, 80000, 64, TRUE);
    $encKey = mb_substr($keys, 0, 32, '8bit');
    $hmacKey = mb_substr($keys, 32, NULL, '8bit');
    $hmacNow = hash_hmac(
        'sha256', $iv . $ciphertext, $hmacKey
    );
    if (!hash_equals($hmac, $hmacNow)) {
        throw new Exception('Authentication error!');
    }

    return openssl_decrypt(
        $ciphertext, 'aes-256-cbc', $encKey,
        OPENSSL_RAW_DATA, $iv
    );
}
```

the *paragonie/random_compat* library[4]. It's a polyfill for the PHP 7 `random_*` functions.

Use Authenticated Encryption

Encryption is not enough to protect data; you also need integrity and authenticity. Without integrity, you can alter encrypted data without any evidence. Without authentication, you cannot be sure the data is generated by a legitimate user.

Moreover, there are specific attacks on encrypted data without authentication, like the *Padding Oracle Attack*[5] discovered in 2001 by Serge Vaudenay. This attack works for ECB, CBC, and OAEP modes for all the block ciphers and can discover the encryption key in a few seconds!

We can add authentication to an encryption system using encrypt-then-authenticate approach or we can use authenticated encryption which offers authentication built-in to the algorithm.

The idea is to encrypt first and then authenticate the encrypted data using a *message authentication code* such as HMAC.

There is an example of encrypt-then-authenticate using OpenSSL in Listing 1.

PHP libraries are available to facilitate the encrypt-then-authenticate implementation. For instance, zendframework/zend-crypt[6] provides AES-HMAC-256 since PHP 5.3.3. Full disclosure: I'm the co-author of zend-crypt.

4 *paragonie/random_compat library:* https://github.com/paragonie/random_compat

5 *Padding Oracle Attack:* http://phpa.me/padding-oracle-attack

6 *zendframework/zend-crypt:* https://zendframework.github.io/zend-crypt/

Starting from PHP 7.1 you can use the authenticated encryption available in OpenSSL. In particular, we can use AES with 256 bit in GCM or CCM mode.

I suggest using the GCM mode which is three times faster than CCM; there are more details below.

The functions to encrypt and decrypt data in OpenSSL are the following:

```
string openssl_encrypt(
    string $data,
    string $method,
    string $password,
    [ int $options = 0 ],
    [ string $iv = "" ],
    [ string &$tag = NULL ],
    [ string $aad = "" ],
    [ int $tag_length = 16 ]
)

string openssl_decrypt(
    string $data,
    string $method,
    string $password,
    [ int $options = 0 ],
    [ string $iv = "" ],
    [ string $tag = "" ],
    [ string $aad = "" ]
)
```

The authentication hash is stored in the `$tag` variable. This value is filled by the `openssl_encrypt` function and returned as a reference.

The other optional parameter `$aad` represents additional authentication data you could use to protect the message against alterations, without the encryption part. For instance, if you need to encrypt an email leaving the header information

in plaintext (like the sender and the receiver) you can pass the header in `$aad`.

The last optional parameter `$tag_length` is the length in bytes of the hash value, that is 16 by default. This value is related to the encryption algorithm used.

To decrypt an authenticated message, you need to pass the `$tag` value to `openssl_decrypt` and optionally the additional authenticated data (`$aad`).

The Galois/Counter Mode (GCM)

GCM[7] is a mode of operation for symmetric key cryptographic block ciphers that provides encryption and authentication. This is an algorithm used in many applications like IPsec, SSH, and TLS 1.2. Used together with AES (AES-GCM) it is included in the NSA Suite B Cryptography. This algorithm is very fast because the execution can be parallelized. Moreover, the algorithm does not any patents and can be used without restrictions.

You can check if your *OpenSSL* extension supports the *GCM* mode using the `openssl_get_cipher_methods` function. If you see `-gcm` or `-GCM` at the end of a cipher name you can use the Galois/Counter Mode. You need to have at least OpenSSL version 1.1 to support this algorithm.

Listing 2 is an example of usage.

In Listing 2, if you need to store the encrypted value somewhere you need to store `$tag` and `$iv` concatenated with `$ciphertext`. This is because you need to pass these data again to decrypt the ciphertext. If you store `$tag` you need to also remember the size of this value. I suggest using the default value of 16 bytes to simplify the usage. The tag length for the GCM mode can be between four and 16 bytes.

If you want to use additional authenticated data you can pass the string to be authenticated in the `$aad` optional parameter of `openssl_encrypt`.

In this case, the result will be the concatenation of: `$tag`, `$iv`, `$aad`, and `$ciphertext`. You need to also store the `$aad` in plaintext to be able to perform the authentication, during the decryption of the message.

Counter With CBC-MAC (CCM)

Counter with CBC-MAC[8] (CCM) is another authenticated encryption mode for symmetric block ciphers. The CCM mode is also used in many applications like IPsec and TLS 1.2 and is part of the IEEE 802.11i standard.

You can use it using the `-ccm` suffix in the OpenSSL algorithm name. For instance, in Listing 2 you can have CCM mode replacing the first line with:

```
$algo = 'aes-256-ccm';
```

Listing 2

```
<?php
// Example of authenticated encryption using GCM
// Note: you need PHP 7.1+

$algo = 'aes-256-gcm';
$iv = random_bytes(openssl_cipher_iv_length($algo));
$key = random_bytes(32); // 256 bit
$data = random_bytes(1024); // 1 Kb of random data

$ciphertext = openssl_encrypt(
   $data, $algo, $key, OPENSSL_RAW_DATA, $iv, $tag
);

$decrypt = openssl_decrypt(
   $ciphertext, $algo, $key, OPENSSL_RAW_DATA, $iv, $tag
);
if (FALSE === $decrypt) {
   throw new Exception(
      sprintf("OpenSSL error: %s", openssl_error_string())
   );
}
printf(
   "Decryption %s\n", $data === $decrypt ? 'Ok' : 'Failed'
);
```

Use RSA With OAEP_PADDING

This is very specific advice; if you are using OpenSSL and you want to encrypt or decrypt using public key cryptography with RSA you need to use the `OPENSSL_PKCS1_OAEP_PADDING` padding mode to prevent the Bleichenbacher's CCA attack.

This is a padding oracle attack against RSA encryption which uses PKCS1v1.5, discovered by Daniel Bleichenbacher in 1998. This attack is more generally known as the "million message attack" due to the attack cost requiring a million messages to recover a plaintext.

Unfortunately, the default padding settings used by OpenSSL in PHP is `OPENSSL_PKCS1_PADDING` and is vulnerable to this attack.

If you are using the `openssl_public_encrypt()` or `openssl_public_decrypt()` functions remember to always use `OPENSSL_PKCS1_OAEP_PADDING` as follow:

```
bool openssl_public_encrypt(
   string $data, string &$crypted,
   mixed $key, OPENSSL_PKCS1_OAEP_PADDING


bool openssl_private_decrypt(
   string $data, string &$decrypted,
   mixed $key, OPENSSL_PKCS1_OAEP_PADDING
)
```

7 *GCM: http://phpa.me/galois-counter-mode*

8 *Counter with CBC-MAC: http://phpa.me/cbc-mac*

Use Base64 to Encode Data

If you need to exchange encrypted data with different systems—for instance, transmitting data over the internet—it is recommended to encode the data in Base64.

In PHP you can use the functions `base64_encode()` and `base64_decode()`. This encoding will guarantee your data will be stored correctly independently of the encoding system used in your environment.

Update Your Version of PHP

The last but not least best practice is to update your version of PHP. Work with the latest PHP version if you can; this is very important for security.

It's important to note the currently supported versions of PHP. As of March 2017, the only supported ones are PHP 5.6 (only security fixes), 7.0, and 7.1. If you are using an old version, you are exposing your applications to potential security issues!

As you know, PHP is a very popular language, and it's used by million of people worldwide. This means many bugs and security fixes (that is good!). Subscribe to the PHP internal mailing list if you want to be updated.

Conclusion

In this article, I reported some best practices for the usage of cryptography in PHP. As I wrote many times, cryptography is hard, and you need to be very careful using it in your application. If you can, hire an expert to review your code or ask for help in the PHP community.

References

For reference, I have a short list of books for more information about cryptography and how to apply it in real use cases.

- N.Ferguson, B.Schneier, and T. Kohno, *Cryptography Engineering*, John Wiley & Sons, 2010
- Ross J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley, 2008
- J.Katz, and Y. Lindell *Introduction to Modern Cryptography, Second Edition*, Chapman & Hall/CRC, 2014
- Wenbo Mao, *Modern Cryptography: Theory and Practice*, Prentice Hall, 2003

Enrico Zimuel is a Senior Software Engineer in the R&D department of Zend, a Rogue Wave Company. He's a core contributor of Apigility and Zend Framework open source projects. He's a TEDx and international speakers of 80+ conferences, and co-founder of PHP User Group Turin, in Italy.

PHP Prepared Statements and MySQL Table Design

Edward Barnard

When using a PHP framework, standard practice is to use an object-relational mapping (ORM) for database access. However, with high-volume logging and statistics-gathering, it pays to go "old school" with PHP prepared statements.

Meanwhile, when MySQL tables quickly grow by millions of rows, table storage space becomes an issue. Our table design must focus on keeping these tables more compact and efficient. Here too, prepared statements simplify both coding and table design.

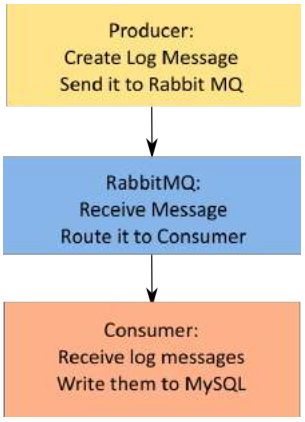
We'll be using CakePHP 3's excellent support for PHP prepared statements, but all concepts are native to PHP and apply to any project striking this use case.

Context

Picture a high-traffic website that records information in a MySQL database, but doesn't further use this specific information for the website itself. It's for offline analysis and reporting. Think of the situation as a one-way data pipeline, with information flowing from the web application to the MySQL database.

The *producer-consumer* programming pattern provides efficiency and high data throughput for this sort of one-way traffic. Our focus will be on the database, but to get there, we need to build a producer-consumer infrastructure. For this example, we'll produce and consume log messages which contain timing data.

Figure 1. Producer-consumer data flow



Infrastructure

This article focuses on the "consumer" portion of the traffic flow, that is, getting our log information into the MySQL database. To get there, however, we need to build our messaging infrastructure fir.

Create a Project

Create a new project according to the CakePHP 3 Quick Start Guide[1]:

```
composer create-project --prefer-dist cakephp/app prepared
```

> `cakephp/app` *requires the* `intl` *extension so make sure it's installed and enabled before you start.*

Listing 1. src/Shell/ProducerShell.php

```
1. <?php
2. namespace App\Shell;
3.
4. use App\DemoLogger\DemoLoggerUtil;
5. use Cake\Console\Shell;
6.
7. class ProducerShell extends Shell
8. {
9.     public function main() {
10.        DemoLoggerUtil::log('Line One', __FUNCTION__);
11.        DemoLoggerUtil::log('Line Two', __FUNCTION__);
12.
13.        DemoLoggerUtil::finalize();
14.        $this->verbose('Done.');
15.    }
16. }
```

1 *CakePHP 3 Quick Start Guide:* http://phpa.me/cakephp-quickstart

Data Flow

When I build a messaging system, I begin "upstream" and work my way "downstream." I first produce the message, then pass it through the messaging system (RabbitMQ), and finally push it to the database (or whatever the final destination might be). This means we'll build the producer first, and then the consumer.

Producer

Let's begin with the logging API. Listing 1 shows our "producer" class.

Notice the logging API; notice how we generate a log message. We call our logger as `DemoLoggerUtil::log(event, detail)`. When we are done producing log messages, we close out the RabbitMQ interface with `DemoLoggerUtil::finalize()`.

Listing 2. src/DemoLogger/DemoLoggerUtil.php

```
 1. <?php
 2. namespace App\DemoLogger;
 3.
 4. use App\GenerateToken\GenerateToken;
 5. use PhpAmqpLib\Channel\AMQPChannel;
 6. use PhpAmqpLib\Connection\AMQPStreamConnection;
 7. use PhpAmqpLib\Message\AMQPMessage;
 8.
 9. final class DemoLoggerUtil
10. {
11.     private static $exchange = 'demo';
12.     private static $routingKey = 'demo_log';
13.     private static $instance;
14.     private $meta;
15.
16.     /** @var AMQPStreamConnection */
17.     private $connection;
18.
19.     /** @var AMQPChannel */
20.     private $channel;
21.
22.     /**
23.      * DemoLoggerUtil constructor. All public methods are
24.      * static
25.      */
26.     private function __construct() {
27.         $this->setMeta();
28.         $this->connectRabbitMQ();
29.     }
30.
31.     private function setMeta() {
32.         $this->meta = [
33.             'begin' => sprintf('%.6f', microtime(TRUE)),
34.             'server' => gethostname(),
35.             'instanceCode' => GenerateToken::token(),
36.         ];
37.     }
38.
39.     private function connectRabbitMQ() {
40.         $this->connection = new AMQPStreamConnection(
41.             'localhost', 5672, 'guest', 'guest'
42.         );
43.         $this->channel = $this->connection->channel();
44.         $this->channel->exchange_declare(
45.             static::$exchange, 'direct', FALSE, FALSE,
FALSE
46.         );
47.     }
48.
49.     public static function finalize() {
50.         static::getInstance()->disconnectRabbitMQ();
51.     }
52.
53.     private function disconnectRabbitMQ() {
54.         $this->channel->close();
55.         $this->connection->close();
56.     }
57.
58.     /**
59.      * @return DemoLoggerUtil
60.      */
61.     public static function getInstance() {
62.         if (!static::$instance) {
63.             static::$instance = new static;
64.         }
65.         return static::$instance;
66.     }
67.
68.     public static function log($event, $detail = '') {
69.         $begin = sprintf('%.6f', microtime(TRUE));
70.         $trace = debug_backtrace(FALSE);
71.         $class = $trace[1]['class'];
72.         $function = $trace[1]['function'];
73.         unset($trace);
74.         $logEvent = [
75.             'begin' => $begin,
76.             'class' => $class,
77.             'function' => $function,
78.             'event' => $event,
79.             'detail' => $detail,
80.         ];
81.         static::getInstance()->flush($logEvent);
82.     }
83.
84.     private function flush(array $logEvent) {
85.         $payload = [
86.             'meta' => $this->meta,
87.             'event' => $logEvent,
88.         ];
89.         $message = new AMQPMessage(json_encode($payload));
90.         $this->channel->basic_publish(
91.             $message, static::$exchange,
static::$routingKey
92.         );
93.     }
94. }
```

Listing 3. First log message

```
{
  "meta": {
    "begin": "1491140360.062965",
    "server": "demo.local",
    "instanceCode": "duZtd5Q4SEaWM_yIMIxhEQ"
  },
  "event": {
    "begin": "1491140360.133589",
    "class": "App\\Shell\\ProducerShell",
    "function": "main",
    "event": "Line One",
    "detail": "main"
  }
}
```

Listing 4. Second log message

```
{
  "meta": {
    "begin": "1491140360.062965",
    "server": "demo.local",
    "instanceCode": "duZtd5Q4SEaWM_yIMIxhEQ"
  },
  "event": {
    "begin": "1491140360.135417",
    "class": "App\\Shell\\ProducerShell",
    "function": "main",
    "event": "Line Two",
    "detail": "main"
  }
}
```

Logging Utility

Listing 2 is the logging utility class. We'll touch on it lightly and focus on its output.

The logger generates a JSON-encoded string with `meta` and `event`. The first log message is in Listing 3.

By comparing the second log message in Listing 4, you can see that `meta` remains the same for all log messages, and `event` records the new information with timestamp.

This particular logger is aimed at understanding timing and performance in legacy PHP code. The main entry point is `log()` at line 68. Everything else is magic under the hood aimed at producing the messages shown.

It's worth noting:

1. Both calls to `microtime()` store the result as a string. Otherwise, `json_encode()` stores the value as a lower-precision number. When passing messages around, it's good practice to convert floating-point numbers to a type which won't be mangled, such as a string.

2. Once we have created the message, we pass it into the RabbitMQ exchange. Our class follows the same pattern as the official RabbitMQ Tutorial Four[2]. We won't be focusing on the RabbitMQ portion at all. It's here to show this technique's context: a producer-consumer messaging system.

Singleton Anti-Pattern

Much has been written about the Singleton design pattern[3] being an anti-pattern. Singletons can be difficult to test, and it can be difficult to guarantee we won't need multiple instances of the class/feature sometime in the unknown future. My production version of `DemoLoggerUtil` *does have unit tests; it contains a* `reset()` *method which deletes the singleton at the end of each test.*

Consumer

Let's work our way "downstream" in terms of data flow. Write just enough of our consumer to ensure we are receiving messages correctly. The below consumer matches RabbitMQ Tutorial Four (see above).

In the `ConsumerShell` class (Listing 5 on the following page), note *TODO 1* (line 40) and *TODO 2* (line 58). We'll be adding our database implementation there.

Run the Infrastructure

Finally, we can check our messaging infrastructure. Start the consumer first, *then* the producer. It's a subtle point outside of the scope of this article, but the RabbitMQ tutorials have the *consumer* create the destination queue. The destination must exist before the producer tries to send messages to it.

In a terminal window, start the consumer with the built-in "verbose" option:

```
bin/cake consumer --verbose

Welcome to CakePHP v3.4.4 Console
```

Nothing happens. By specifying "verbose" we get some version information (partially shown above), and then nothing. This is because our consumer is waiting for RabbitMQ to deliver our messages.

In a different terminal window, run the producer:

```
bin/cake producer

Welcome to CakePHP v3.4.4 Console
```

We see the same welcome message followed by the command line prompt (not shown). The producer invisibly did its thing, and stopped. Over on the consumer's window, however, we suddenly have output:

2 *RabbitMQ Tutorial Four:* http://phpa.me/rabbitmq-tutorial-4

3 *Singleton design pattern:* http://phpa.me/singleton-pattern

```
{"meta":{"begin":"1491168463.461577","server":"demo.local",
"instanceCode":"bK6968dQ_INDCstCFrC9RQ"},"event":{"begin":
"1491168463.532451","class":"App\\Shell\\ProducerShell",
"function":"main","event":"Line One","detail":"main"}}
{"meta":{"begin":"1491168463.461577","server":"demo.local",
"instanceCode":"bK6968dQ_INDCstCFrC9RQ"},"event":{"begin":
"1491168463.536446","class":"App\\Shell\\ProducerShell",
"function":"main","event":"Line Two","detail":"main"}}
```

The output is ugly to be sure, but it *is* what we expect. We expected to see two JSON-encoded strings which are our two log messages. Success!

First Table Design

When designing a messaging system, as I noted before, I prefer to work from "upstream" to "downstream." At this point, we have incoming live data. We can design a table to hold the result.

Our first table design is intentionally inefficient. We'll be evolving the design. First, we'll be mapping each incoming message to a single table row (see Listing 6).

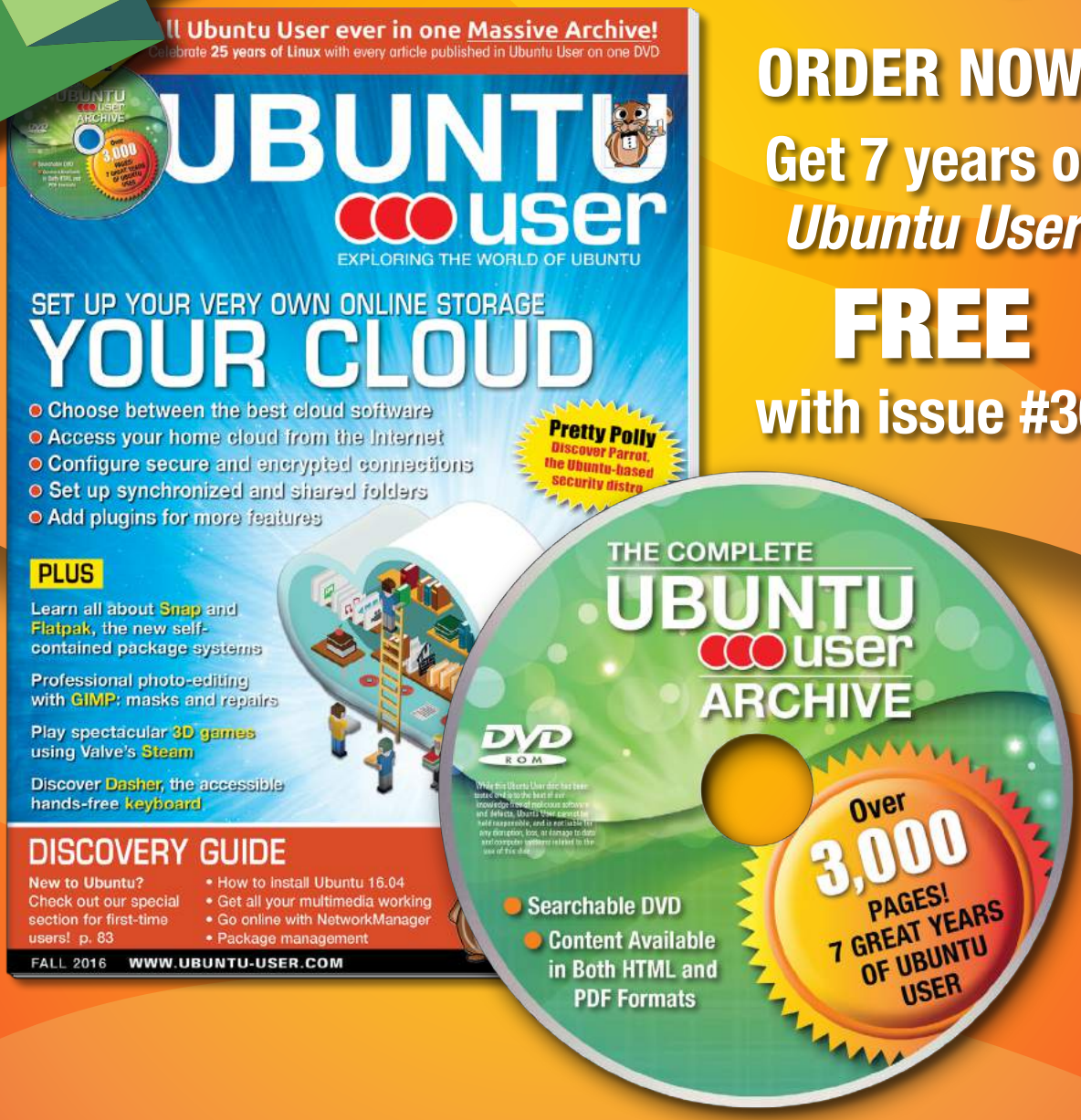
Listing 6. Table demo_log_v01

```
1. CREATE TABLE `demo_log_v01` (
2.   `id` bigint(10) unsigned NOT NULL AUTO_INCREMENT,
3.   `archive_month` tinyint(3) unsigned NOT NULL DEFAULT '1',
4.   `instance_code` char(22) NOT NULL DEFAULT '',
5.   `hostname` varchar(255) NOT NULL DEFAULT '',
6.   `instance_begin` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
7.   `event_time` timestamp NOT NULL DEFAULT '0000-00-00 00:00:00',
8.   `event_class` varchar(255) NOT NULL DEFAULT '',
9.   `event_function` varchar(255) NOT NULL DEFAULT '',
10.  `event` varchar(255) NOT NULL DEFAULT '',
11.  `detail` varchar(255) NOT NULL DEFAULT '',
12.  `created` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
13.  PRIMARY KEY (`id`,`archive_month`)
14. ) ENGINE=InnoDB DEFAULT CHARSET=utf8
15. /*!50100 PARTITION BY LIST (archive_month) ... */
```

Listing 5. Incomplete src/Shell/ConsumerShell.php

```
1. <?php
2. namespace App\Shell;
3.
4. use Cake\Console\Shell;
5. use PhpAmqpLib\Channel\AMQPChannel;
6. use PhpAmqpLib\Connection\AMQPStreamConnection;
7.
8. class ConsumerShell extends Shell
9. {
10.    private static $exchange = 'demo';
11.    private static $routingKey = 'demo_log';
12.    private $queueName;
13.
14.    /** @var AMQPStreamConnection */
15.    private $connection;
16.
17.    /** @var AMQPChannel */
18.    private $channel;
19.
20.    public function main() {
21.       $this->myInitialize();
22.       $this->consume();
23.    }
24.
25.    private function myInitialize() {
26.       $this->connectRabbitMQ();
27.       $this->prepareStatements();
28.    }
29.
30.    private function connectRabbitMQ() {
31.       $this->connection = new AMQPStreamConnection(
32.          'localhost', 5672, 'guest', 'guest'
33.       );
34.       $this->channel = $this->connection->channel();
35.       $this->channel->exchange_declare(
36.          static::$exchange, 'direct', FALSE, FALSE, FALSE
37.       );
38.       $this->queueName = static::$exchange . '_'
39.                        . static::$routingKey;
40.       $this->channel->queue_declare(
41.          $this->queueName, FALSE, TRUE, FALSE, FALSE
42.       );
43.       $this->channel->queue_bind(
44.          $this->queueName, static::$exchange,
45.          static::$routingKey
46.       );
47.    }
48.
49.    private function prepareStatements() {
50.       // TODO
51.    }
52.
53.    private function consume() {
54.       $callback = function ($msg) {
55.          $this->processMessage($msg->body);
56.       };
57.       $this->channel->basic_consume(
58.          $this->queueName, '', FALSE, TRUE, FALSE, FALSE,
59.          $callback
60.       );
61.       while (count($this->channel->callbacks)) {
62.          $this->channel->wait();
63.       }
64.       $this->channel->close();
65.       $this->connection->close();
66.    }
67.
68.    public function processMessage($payload) {
69.       $message = json_decode($payload, TRUE);
70.       $this->verbose($payload);
71.    }
72. }
```

Celebrating 25 Years of Linux!



ORDER NOW!

Get 7 years of *Ubuntu User*

FREE

with issue #30

***Ubuntu User* is the only magazine for the Ubuntu Linux Community!**

BEST VALUE: Become a subscriber and save 35% off the cover price!
The archive DVD will be included with the Fall Issue, so you must act now!

Order Now! Shop.linuxnewmedia.com

Oops! In designing this table, I realized my laptop is only running MySQL 5.5, which does not support microsecond timestamps. Since microsecond precision is not important for this article, we'll simply drop the fractional portion of the timestamp as we insert it into our table. We also save a couple of bytes per row by using the "timestamp" data type rather than "date time" data type.

Prepare Statements and Execute

Now that we have a table let's write the "old school" code to insert our messages into the table. In Listing 7, we've replaced the "TODO" lines with real code.

Prepare and Process

In `prepareStatements()` we begin by obtaining a MySQL database connection. We won't be using any framework Model classes. Next, we prepare the insert statement with placeholders[4].

In `processMessage()` we extract the received message and insert it into our table. We do this by creating a parameter list in the same order as our prepared insert statement.

To be sure, this looks a bit "clunky." I'm explicitly truncating some fields to their column width of 255 characters. I'm converting the microsecond-precision timestamp fields to a standard date-time format. The framework `Model` class could be doing that messy work for me.

However, *for this use case* we already have a strong gain. The prepare/execute cycle is *far* faster and more efficient than the framework `Model` class. The consumer is a long-running process. We could have thousands or millions of row inserts over the life of that one prepared statement. When you look at it that way, the slight "clunkiness" is more than acceptable.

Listing 7. Additional Consumer Code

```
1. private function prepareStatements() {
2.    /** @var Connection $connection */
3.    $connection = ConnectionManager::get('demo');
4.
5.    $sql = 'insert into demo_log_v01
6.       (archive_month, instance_code, hostname, instance_begin,
7.       event_time, event_class, event_function, event,
8.       detail, created) VALUES
9.       (month(curdate()), ?, ?, ?, ?, ?, ?, ?, ?, now())';
10.   $this->insertDemoLog = $connection->prepare($sql);
11. }
12.
13. public function processMessage($payload) {
14.    $message = json_decode($payload, TRUE);
15.    $this->verbose($payload);
16.    if (is_array($message) && count($message)) {
17.       $parms = [
18.          $message['meta']['instanceCode'],
19.          $this->varchar($message['meta']['server']),
20.          $this->timestamp($message['meta']['begin']),
21.          $this->timestamp($message['event']['begin']),
22.          $this->varchar($message['event']['class']),
23.          $this->varchar($message['event']['function']),
24.          $this->varchar($message['event']['event']),
25.          $this->varchar($message['event']['detail']),
26.       ];
27.       $this->insertDemoLog->execute($parms);
28.    }
29. }
30.
31. private function timestamp($microtime) {
32.    return date('Y-m-d H:i:s', (int)$microtime);
33. }
34.
35. private function varchar($string, $length = 255) {
36.    return substr($string, 0, $length);
37. }
```

Final Table Design

It's time to fast forward. It's time to create a more efficient table design *and* automate the building and usage of our prepared statements. Let's make some observations.

First observation. `Meta` remains the same for every message coming from a given process. We should have one row in a `meta` table with multiple rows in an `event` table (one row per received message). Each `event` table row points to its corresponding `meta` table row.

Second observation. `Server` is always the same. It's the hostname of the server running the producer. Even if the producer is potentially running on many in-house servers, that's still a small number compared to the millions of table rows containing the hostname.

The same observation applies to class, function, and event. We'll assume `detail` is different most every time.

Third observation. Not all tables need to be partitioned. The "meta" table `demo_instance` could have a large number of rows and will be partitioned, as will the "event" table `demo_event`. The other tables (server name, class name, function name, event name) will likely have well under a thousand rows each. We'll provision them for 65,535 rows (small int).

> *Partitioning a table distributes portions of individual tables across a file system based on a partitioning function. It makes it possible for tables to be larger than the limit imposed by the filesystem and can improve the performance of some queries. For more see Overview of Partitioning in MySQL[5].*

4 placeholders: http://php.net/pdo.prepared-statements

5 Overview of Partitioning in MySQL: http://phpa.me/mysql57-partition

Listing 8. LookupUtil.php

```php
<?php
namespace App\DemoLogger;

use Cake\Database\Connection;
use Cake\Database\StatementInterface;

final class LookupUtil
{
    private static $cacheLimit = 200;
    private static $instances = [];

    /** @var Connection */
    private $connection;
    private $table;

    /** @var StatementInterface */
    private $query;
    /** @var StatementInterface */
    private $insert;
    private $cache = [];

    /**
     * LookupUtil constructor.
     *
     * @param Connection $connection
     * @param string $table
     * @param array $dependencies For unit testing
     */
    private function __construct(
        Connection $connection, $table, array $dependencies
    ) {
        $this->connection = $connection;
        $this->table = $table;
        $this->setDependencies($dependencies);
        $this->prepareStatements();
    }

    private function setDependencies(array $dependencies) {
        foreach ($dependencies as $key => $value) {
            $this->$key = $value;
        }
    }

    private function prepareStatements() {
        if (!$this->query) {
            /** @noinspection SqlResolve */
            $sql = "select id from {$this->table}
             where `name` = ?";
            $this->query = $this->connection->prepare($sql);
        }
        if (!$this->insert) {
            $sql = "insert into {$this->table}
             (`name`, created) VALUES (?, now())";
            $this->insert = $this->connection->prepare($sql);
        }
    }

    public static function lookup(
        Connection $connection, $table, $value
    ) {
        $instance = static::getInstance($connection, $table);
        if (!array_key_exists($value, $instance->cache)) {
            return $instance->runLookup($value);
        }
        return $instance->cache[$value];
    }

    /**
     * @return LookupUtil
     */
    public static function getInstance(
        Connection $connection, $table, array $dependencies = []
    ) {
        if (!array_key_exists($table, static::$instances)) {
            static::$instances[$table] = new static(
                $connection, $table, $dependencies
            );
        }
        return static::$instances[$table];
    }

    private function runLookup($value) {
        if (count($this->cache) >= static::$cacheLimit) {
            $this->cache = [];
        }
        if (!$this->query) {
            // Should only happen when developing unit tests
            throw new \InvalidArgumentException(
                'No query for '
                . $this->table
            );
        }
        $parms = [substr($value, 0, 255)];
        $this->query->execute($parms);
        $row = $this->query->fetch('assoc');
        if (is_array($row) && count($row)) {
            $id = (int)$row['id'];
        } else {
            $this->insert->execute($parms);
             $id = (int)$this->insert->lastInsertId();
        }
        $this->cache[$value] = $id;
        return $id;
    }
}
```

Code Design

Now that we understand the situation, we can improve our code design.

First, in our example, the message `function` is always `main`. We might add several rows to table `demo_log`, but we only have the one row in `demo_function` (it contains `main`). Given the consumer is a long-running process, it would make sense to cache the row ID containing `main`. There would be no point in looking it up every time for thousands of rows when we already know the answer.

Second, four of the six tables are identical except for the table name. Knowing this, we can write a single utility class handling all such tables. See Listing 8.

Lookup Utility

The Lookup Utility is an array of singletons, with one singleton for each table. There is a small amount of extra code in the utility, allowing for unit testing.

Method `prepareStatements()` contains the meat of the matter. We create a PHP prepared statement unique to the table in question. We have one (singleton) instance for each table. Each table has `id` as the primary key, and `name` containing the variable text. When the row is not found, we've prepared the statement to insert it.

This utility assumes only one consumer is running at a time. When there is a chance of multiple processes, I would change the "insert" to "insert ignore" and then, after the insert, re-run the query rather than depending on the last insert ID. That approach should avoid race conditions. Since inserts are relatively rare (compared to writing the log/instance rows), there shouldn't be much overhead.

This utility also works as a database fixture (for testing purposes) without requiring any real database connection. When you are testing code which uses this utility, mock the query and insert statements. Load the instance cached with values to be returned. Once the correct dependencies have been injected, `LookupUtil` runs smoothly (and quickly) without any database access.

Listing 9. Consumer changes

```
1. private function prepareStatements() {
2.    /** @var Connection $connection */
3.    $this->db = ConnectionManager::get('demo');
4.
5.    $sql = 'SELECT id FROM demo_instance WHERE
6.       archive_month = ? AND instance_code = ? LIMIT 1';
7.    $this->queryInstance = $this->db->prepare($sql);
8.
9.    $sql = 'INSERT INTO demo_instance
10.      (archive_month, demo_server_id, instance_code,
11.      instance_begin, created) VALUES
12.      (month(curdate()), ?, ?, ?, now())';
13.   $this->insertInstance = $this->db->prepare($sql);
14.
15.   $sql = 'INSERT INTO demo_log (archive_month,
16.      demo_instance_id, demo_class_id, demo_function_id,
17.      demo_event_id, event_time, detail, created) VALUES
18.      (month(curdate()), ?, ?, ?, ?, ?, ?, now())';
19.   $this->insertLog = $this->db->prepare($sql);
20. }
21.
22. public function processMessage($payload) {
23.    $this->message = json_decode($payload, TRUE);
24.    $this->verbose($payload);
25.    if (is_array($this->message) && count($this->message)) {
26.       $detail = $this->message['event']['detail'];
27.       $detail = substr($detail, 0, 255);
28.       $parms = [
29.          $this->lookupInstance(),
30.          $this->lookup('demo_class', 'class'),
31.          $this->lookup('demo_function', 'function'),
32.          $this->lookup('demo_event', 'event'),
33.          $this->timestamp('event'),
34.          $detail,
35.       ];
36.       $this->insertLog->execute($parms);
37.    }
38. }
39.
40. private function lookupInstance() {
41.    $code = $this->message['meta']['instanceCode'];
42.    if (array_key_exists($code, $this->instanceCache)) {
43.       return $this->instanceCache[$code];
44.    }
45.    if (count($this->instanceCache) >= static::$cacheLimit)
46. {
47.       $this->instanceCache = [];
48.    }
49.    $parms = [date('n'), $code];
50.    $this->queryInstance->execute($parms);
51.    $row = $this->queryInstance->fetch('assoc');
52.    if (is_array($row) && count($row)) {
53.       $id = (int)$row['id'];
54.    } else {
55.       $server = $this->message['meta']['server'];
56.       $serverId = LookupUtil::lookup(
57.          $this->db, 'demo_server', $server
58.       );
59.       $begin = $this->timestamp('meta');
60.       $parms = [$serverId, $code, $begin];
61.       $this->insertInstance->execute($parms);
62.       $id = (int)$this->insertInstance->lastInsertId();
63.    }
64.    $this->instanceCache[$code] = $id;
65.    return $id;
66. }
67. private function timestamp($section) {
68.    $microtime = $this->message[$section]['begin'];
69.    return date('Y-m-d H:i:s', (int)$microtime);
70. }
71.
72. private function lookup($table, $field) {
73.    $value = $this->message['event'][$field];
74.    return LookupUtil::lookup($this->db, $table, $value);
75. }
```

In your unit test, call `LookupUtil::getInstance()` with your array of dependencies. `setDependencies()` creates and populates the singleton for that table. Then, whenever your code calls `LookupUtil::lookup()`, the utility returns your test values from memory.

Consumer

Our table structure is more complex but now far more compact and efficient. Our consumer can cache various values. Most of the ugly database code is hidden in `LookupUtil`. We can, of course, use `LookupUtil` for any other table which has the same structure.

Listing 9 is the changed portion of our consumer.

The new `prepareStatements()` creates both select and insert statements for the `demo_instance` table. Remember, when a single running process (the producer) generates several log messages, those messages will become one row in `demo_instance` and several rows in `demo_log`. We, therefore, need to check and see if we already have the `demo_instance` row, or if we need to create it. We'll be caching the result just like the `LookupUtil`.

The new `processMessage()` is under 20 lines of code but is writing to six different database tables. We are converting timestamps, keeping strings to 255 characters, and caching results, so we don't have to keep looking up the same thing. We limit each cache to 200 items, so we don't have memory creep.

Method `lookup()` at the end of the listing shows how we use the Lookup Utility. We extract the relevant field from our RabbitMQ message and pass it to the `LookupUtil`. The `LookupUtil` will, in turn, return the row ID or create the table row and then return the row ID.

Method `lookupInstance()` maintains our access to the `demo_instance` table. We follow the same pattern; we insert table rows as needed, and return the row ID of the requested row. We cache up to 200 row IDs.

Query

Our table design is optimized for efficient inserts rather than easy retrieval. It's helpful to provide a sample query which shows how to pull information from our tables. The query is in Listing 10.

Summary

This project addresses a specific use case, but it's a use case I do see in production. We created a one-way data pipeline from our main PHP web application to our MySQL database. I based this project on actual production code.

We simulated the main PHP application with our simple "producer" script. On a high-traffic site with lots of users, each of those page loads would be funneling messages through RabbitMQ to our single consumer. Even so, the messaging pattern is the same.

Listing 10. Sample Query

```
1. SELECT
2.     ds.`name` hostname ,
3.     di.instance_code ,
4.     di.instance_begin ,
5.     dc.`name` `class` ,
6.     df.`name` `function` ,
7.     de.`name` `event` ,
8.     dl.event_time ,
9.     dl.detail
10. FROM
11.     demo_log dl
12. INNER JOIN demo_instance di ON di.id = dl.demo_instance_id
13. AND di.archive_month = dl.archive_month
14. INNER JOIN demo_server ds ON ds.id = di.demo_server_id
15. INNER JOIN demo_class dc ON dc.id = dl.demo_class_id
16. INNER JOIN demo_function df ON df.id = dl.demo_function_id
17. INNER JOIN demo_event de ON de.id = dl.demo_event_id
18. ORDER BY
19.     dl.id DESC
20. LIMIT 10;
```

It's often the case that database tables have repeating information such as city, country, or username. In our example we had repeating host name, PHP class name, and so on. When you decide to use smaller tables off to the side and name them the same way, your code can take advantage of your table design pattern.

In your main PHP web application, by all means, take advantage of your framework's tools. Free yourself from having to deal with low-level concerns such as prepared statements.

On the other hand, when it's just you and your data pipeline, with nary a pesky user in sight, use the better tools for *this* job. MySQL statements take a substantial time to analyze, compile, and execute. When you prepare it once and use it thousands—or millions—of times, you have a great improvement in efficiency.

With practice, this whole process becomes easy. Learn the pattern and build the next pipeline. Enjoy!

Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others. @ewbarnard

Qafoo Quality Analyzer

Matthew Setter

Code. Is it science or an art? While it *is* a science, it is often treated as an art. Not sure what I mean? How often do you go on gut feelings and intuition that your code is of sufficiently high quality? How often, when referring to it do you use terms such as "I think," "It should," and "I guess?"

If you want to know how good your (or your team's) code is, then these terms don't cut it. You have to know how good it is, quantitatively. And to do that, you have to analyze it. In this month's edition of Education Station, I'm going to step you through an excellent library that can do help you do this: the Qafoo Quality Analyzer.

What Is It?

The Qafoo Quality Analyzer is, "*a tool to visualize metrics and source code*." It's used by the team at Qafoo[1] internally for code reviews for their customers. One way of looking at it is as an intelligent wrapper over several of the existing PHP code quality tools. Specifically, it incorporates:

- *PDepend*[2]
- *PHP Mess Detector*[3]
- *PHP_CodeSniffer*[4]
- *PHP Copy/Paste Detector*[5]
- *PHPLOC*[6]

With these, the quality analyzer can:

- Assess the lines of code in the project or a given file.
- Analyze code metrics (at the package, class and method level).
- Trace and graph code dependencies.
- Assess mess detector violations
- Assess code style violations.
- Discover code duplicates.

You might be thinking you could do all of this yourself. Well, yes, you could. However, there's a lot of work that you'd have to do manually. Secondly, you wouldn't have the unified interface and simplified setup along with the both the command line and web interfaces which the tool supports.

1 *Qafoo:* http://phpa.me/qafoo-analyzer
2 *PDepend:* http://pdepend.org
3 *PHP Mess Detector:* https://phpmd.org
4 *PHP_CodeSniffer:* http://phpa.me/php_codesniffer
5 *PHP Copy/Paste Detector:* http://phpa.me/php-cpd
6 *PHPLOC:* http://phpa.me/php-loc

What's Code Quality Analysis and Why Does It Matter?

I don't want to assume you're an old hand at code quality analysis, nor that it's something with which you're intimately familiar. So, let's have a quick introduction. A simplistic explanation of code quality analysis is that it is a standardized process which helps ensure the delivery of ever increasing, high quality, code.

The processes do this by measuring code against a range of metrics. These include:

- Code complexity
- Code smells
- Lines of code
- Code dependencies
- Code duplication

By doing so, it becomes easier to see where problems either are or might arise in the future. It is also simpler to know whether the quality of the code is improving or declining. There is a range of online services which provide this, such as Code Climate[7], and Phabricator[8]. However, the pricing might prove prohibitive—especially if you're working for a small organization or team. Given that, the Qafoo Quality Analyzer is an excellent jumping off point.

> David Stockton previously wrote about many of these tools and metrics in *Leveling Up: Using Code to Help You Code Better*[9] in the February 2016 issue.

Installation

Now that we've laid a solid foundation for what source code quality analysis is and why you would use it, let's install the analyzer so we can analyze a codebase. We're not going to cover every aspect of the analyzer—I want to leave something for you to experiment with. But we will cover several of the key aspects.

As with most PHP projects, there are several ways to install the project. But, as always, the one I favor is Composer. However, there's one little catch: there's no initial release yet. This might be of concern to some, yet not to others. I don't see it as a big problem. But it

7 *Code Climate:* https://codeclimate.com
8 *Phabricator:* https://www.phacility.com
9 Leveling Up: Using Code to Help You Code Better, http://phpa.me/2016febzine

might make it difficult to integrate with your project; mainly because you're going to have to lower the minimum stability setting in your project's `composer.json` to `dev`.

If you're comfortable with that, then add the following line to your `composer.json`:

```
"minimum-stability": "dev"
```

Then, in the `require-dev` section, add:

```
"qafoo/quality-analyzer": "dev-master"
```

When complete, save the file and run `composer update`. After a short time, you *should* have the library available in the `vendor` directory, and the binaries available in `vendor/bin`. If so, you're now ready to begin. Alternatively, you could create a separate project just for code analysis and remember to use the appropriate paths.

Once it's installed, you can see which analyzers are installed with:

```
vendor/bin/analyze list:analyzers
```

How to Use It

Assuming your code's located under `src` run the following command:

```
vendor/bin/analyze analyze src
```

This will run the analyzers over all the PHP files in the `src` directory and generate the analysis results. When it runs, you should expect to see output similar to the following:

```
Analyze source code in /Users/matt/health-monitor-api/src
 * Running source
 * Running coverage
 * Running pdepend
 * Running dependencies
 * Running phpmd
 * Running checkstyle
 * Running tests
 * Running cpd
 * Running phploc
 * Running git
 * Running gitDetailed
Done
```

This lists all of the analyzers run on your codebase. When it's complete, you then need to boot the dashboard so that you can see the results. To do that, run the following command, again from the root directory of your project:

```
vendor/bin/analyze serve
```

This will start PHP's built-in web server on port 8080. As it's running, open `http://localhost:8080/` in your browser of choice, and you'll see the main dashboard, as in Figure 1. I ran this on a small codebase of mine. Unfortunately, I don't have a large codebase to run it on, but I'm hoping this one is sufficient to be able to highlight enough of the tool's functionality.

You can see the dashboard is quite minimalist. There are three menus:

- **Source:** this takes you to the source view.
- **Metrics:** this shows the code analysis for its *size*, *metrics*, and *dependencies*.
- **Reports:** this shows the code analysis for the *mess detector*, *tests*, *checkstyle*, and *copy and paste*.

You can see the code rates pretty well. It has 196 lines of code, no code duplications, and no mess detector violations.

Figure 1.

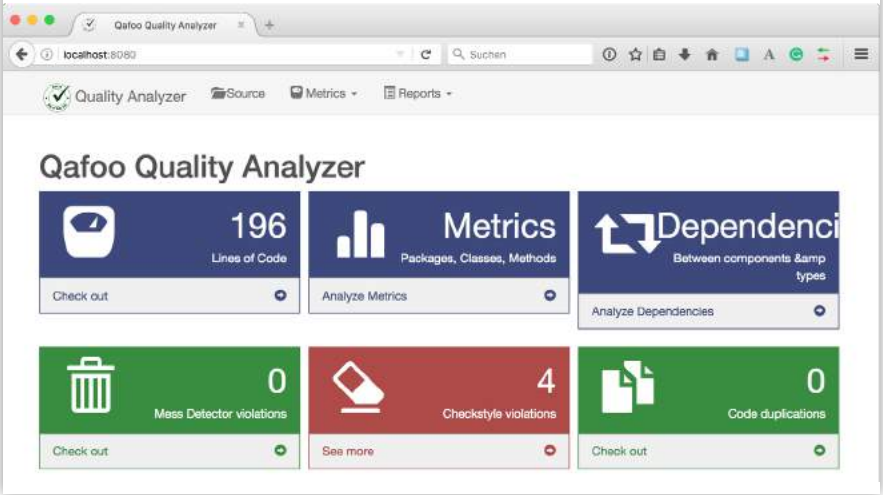
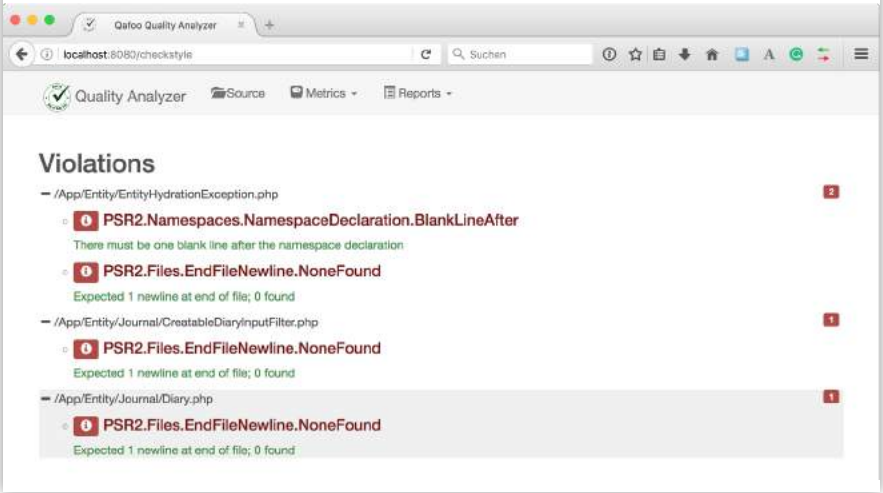


Figure 2.



Code Style Analysis

Let's start with the code style, by clicking `Reports -> Checkstyle`, we see the Checkstyle report. Here, it reports four checkstyle violations. I honestly wasn't expecting that. So, let's have a look at the results. In Figure 2, you can see the expanded view of the checkstyle violations.

They're not overly serious, if at all. But they're worth noting if as a team we have chosen a code style (in this case, PSR-2) and we're all meant to stick to it. As the adage goes: pick whatever style you want, be it good or bad, but regardless of what it is, **stick to it**.

That said, three files have no trailing newline. And one has more than one blank line after the namespace declaration. Investigating each file, I found all the violations were legitimate. No violations are reported after updating them as necessary, and re-running:

```
vendor/bin/analyze analyze src
```

You can see the revised violations dashboard in Figure 3.

Project Size

Now, let's have a look at the project size analysis results. To get there, click `Metrics -> Size`. Again, as the size of the assessed codebase isn't significant, I'm not expecting to see much. There, we see two graphs, which you can see in Figure 4. The first shows how many lines of code are commented and uncommented. The second shows the number of lines of code across *Traits*, *abstract classes*, *interfaces*, and *classes*.

The report shows that I have only 35 commented lines of code, versus 161 without comments. It also indicates I have six classes. Not altogether a bad result; though, I prefer to have as many code comments as is practical. Perhaps we're touching on a flame war, but I believe in as much commenting as is practical — *but no more*.

The reason is partly for the other developers, partly for the IDEs. I see this as further required as PHP is a loosely-typed language. Though, the type-hinting in PHP 7 has improved this situation immensely.

I've found it extremely helpful to at least always comment the function signatures and the return types, if nothing else. That way, if you're using an IDE, it is more intelligently able to help you. And for other developers, you can give them a better understanding of what to expect and why you wrote the function in the first place.

But, as this month's column's more about Qafoo Quality Analyzer, rather than my views on commenting, I'm going to hop off my soapbox and get back to the walk-through.

Code Dependencies

Now that we've looked at code size, what about dependencies? Qafoo provides an interesting interface for doing so. By clicking on `Metrics -> Dependencies`, you can see a condensed code dependency graph for my modest codebase, as in Figure 5.

Figure 3.

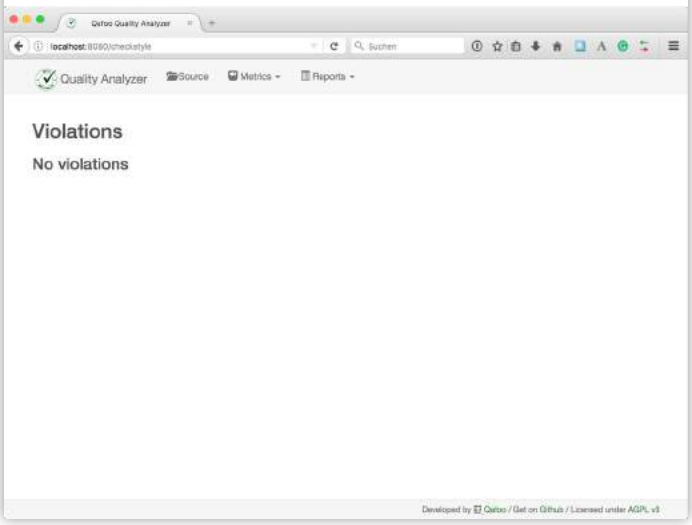


Figure 4.

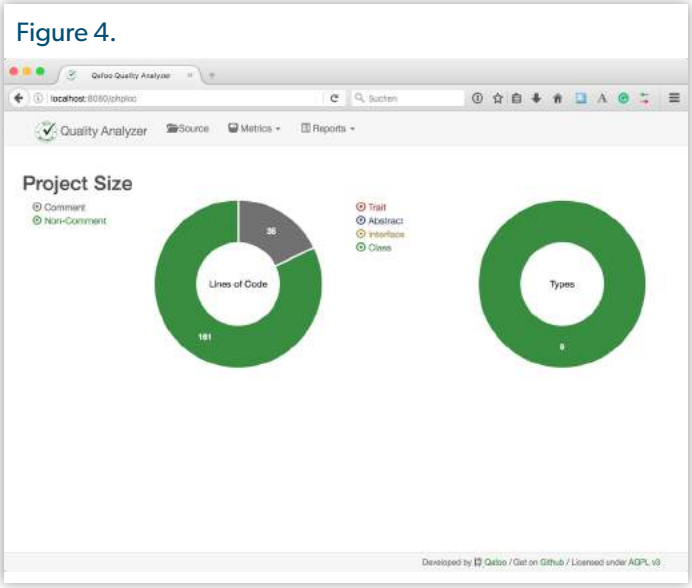
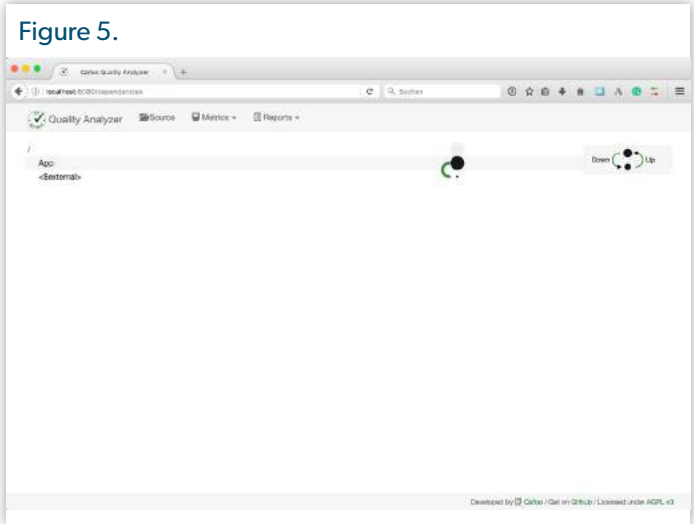


Figure 5.



By clicking on each class, you can progressively expand out the dependency graph and see which classes depend on which, until you've fully expanded the graph, as in Figure 6. In my graph, you can see:

- `HomePageFactory` depends on `HomePageAction`
- `Diary` depends on `EntityHydrationException`

It's not an overly complicated graph. But it's enlightening nonetheless.

Code Metrics

Let's look at code metrics, which you can see in Figure 7. To do so click `Metrics -> Metrics`. This initially shows the lines of code per file, and by clicking on the filename, you can see the file's source.

Down the left-hand side, though, you can see a range of other metrics, grouped by package, class, and method, which we can peruse. These include:

- Package code rank, number of functions, and Git commits
- Class logical lines of code, efferent coupling, and non-private weighted method count
- Method CRAP Index, Cyclomatic Complexity, and Maintainability Index

There are far too many to cover them all in the space I have available. So let's have a look at three; those being the method's Cyclomatic Complexity[10] and Weighted Method Count[11], as well as the class' efferent coupling[12]. If you don't know what those are, here's a quick introduction.

Efferent coupling: measures the number of data types a class knows about. A large efferent coupling can indicate a class is unfocused. It may also indicate brittleness since it depends on the stability of all the types to which it is coupled.

The Weighted Method Count: is a good indicator how much effort will be necessary to maintain and develop a particular class.

Cyclomatic complexity: is a software metric (measurement), used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code.

You can see the results of these tests in Figures 8-10.

Figure 6.

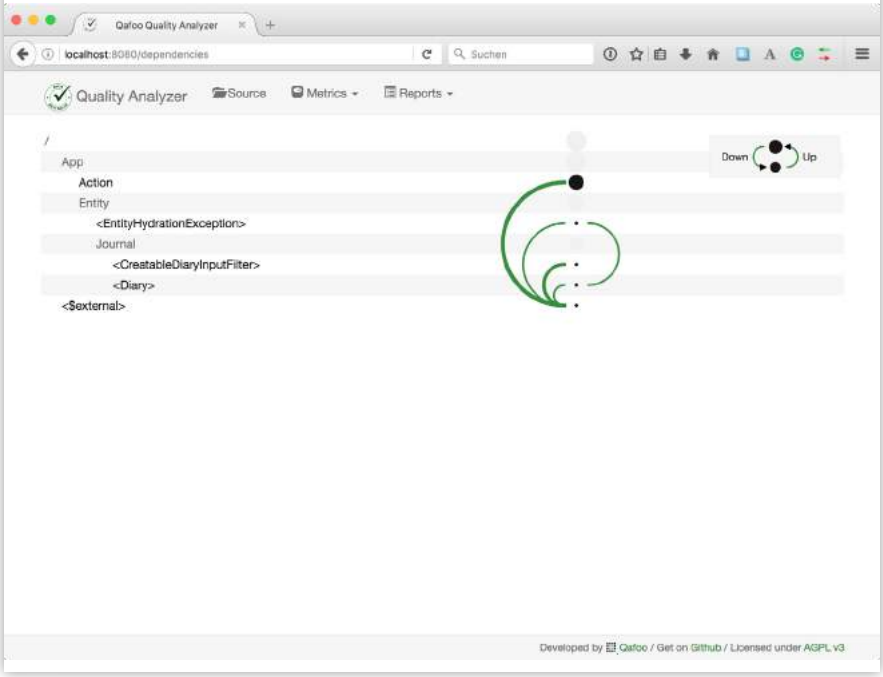
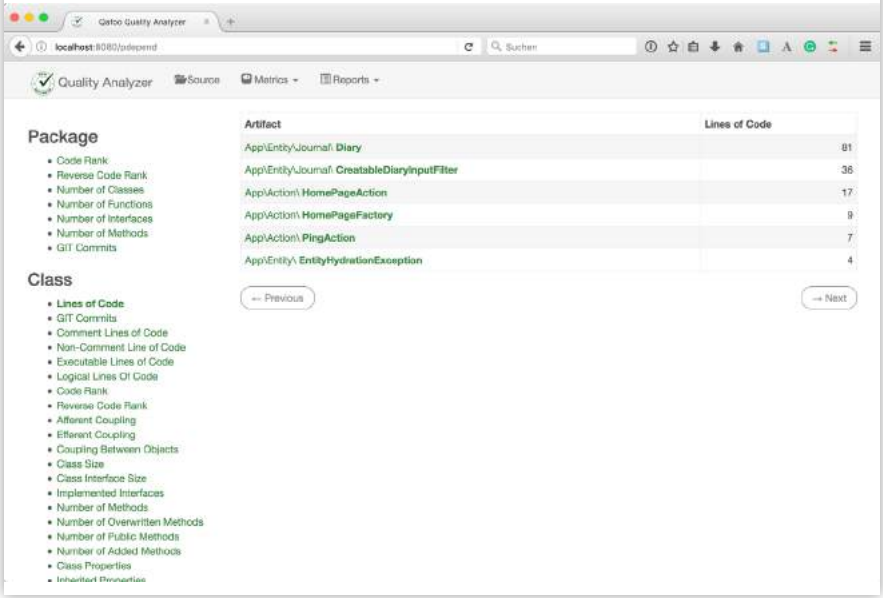


Figure 7.



10 *Cyclomatic Complexity:* http://phpa.me/cyclomaticComplexity

11 *Weighted Method Count:* http://phpa.me/weighted-method-count

12 *efferent coupling:* http://phpa.me/efferent-coupling

Figure 8.

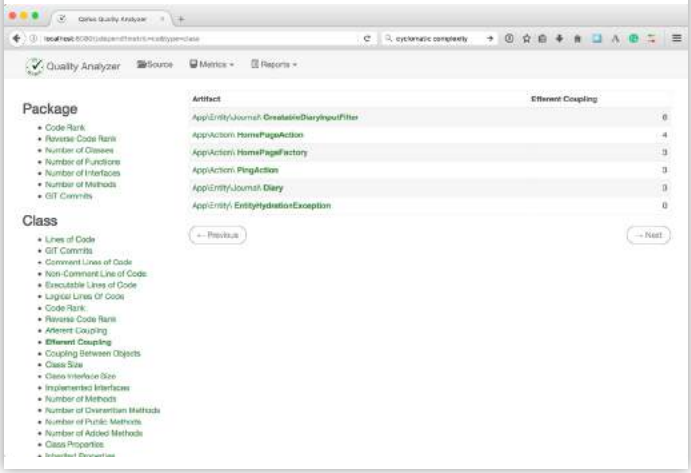
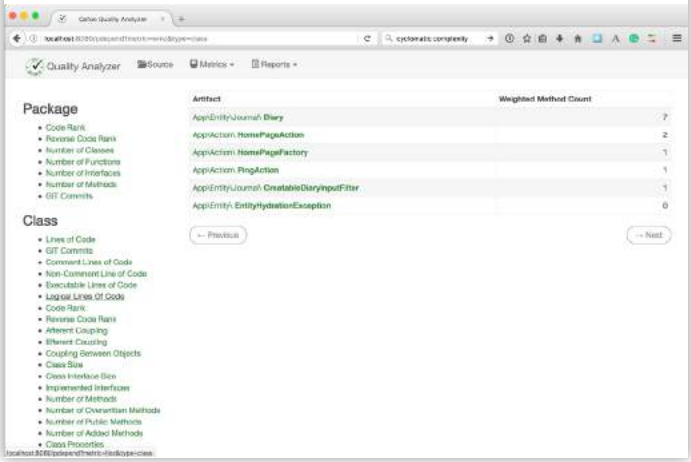


Figure 9.



Excluding Tests or Source Code

Now that we've run the default analysis let's see how to customize the analysis for a select group of tests. Specifically, let's see how to exclude some directories and analyzers.

Why would you do this? Here are four good reasons:

1. You're only interested in a few directories or types of analysis, not all of them.
2. You want to ease your team into code quality analysis, and not overwhelm them.
3. Your codebase is too big to analyze in a meaningfully short period.
4. Your codebase is so big that to analyze it all intermittently exhausts available memory.

If so, then use the `--exclude` switch to specify which directories to skip and `--exclude_analyzers`. We'll start with `--exclude`. For example, let's say you have the following directory structure:

```
/src
  /templates
  /modules
  /includes
  /partials
```

And let's say that all you're interested in is `/src/modules`. You could either run:

```
vendor/bin/analyze analyze --exclude=src/templates \
 --exclude=src/partials --exclude=src/includes src
```

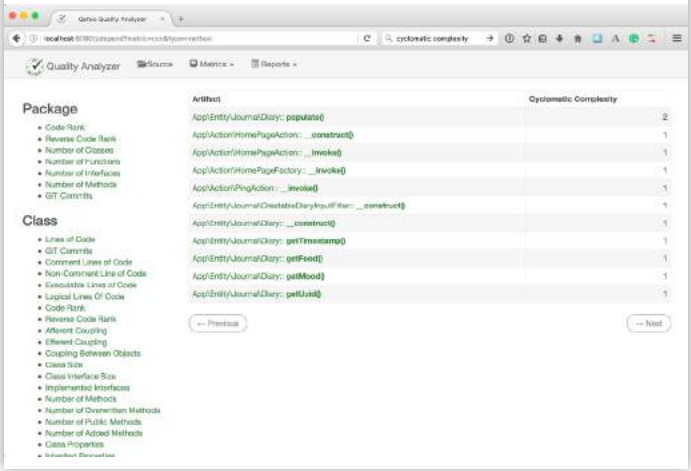
Or, you could run:

```
vendor/bin/analyze analyze src/modules
```

Either way, you have the option to filter out some directories if required.

If you only want to run a few analyzers, as I mentioned use the `--exclude_analyzers` switch. To use it, pass an enclosed, comma-separated list of the analyzers to run. Say you only wanted to run `pdepend`, `phpmd`, and `phploc`. You would do so by running the following command.

Figure 10.




```
vendor/bin/analyze analyze --exclude_analyzers="source,
coverage,dependencies,checkstyle,tests,cpd,git,gitDetailed" \
src/modules
```

What would be handy, instead of having to exclude the analyzers that you don't want, so that you run the ones that you do, would be to have an `--include_analyzers` switch instead. Perhaps I should submit a PR for that. I can easily see the case for both.

Little Gotchas Along the Way

I'm going to leave the tool coverage there, and finish up with some gotchas I encountered while experimenting the quality analyzer.

While the codebase I integrated the analyzer with was pretty straight-forward and the install worked the first time perfectly, yours might not. You may be working with a codebase that's still got a lot of legacy baggage, one not utilizing a lot of modern PHP's features. Alternatively, it may use Composer, but in a non-standard way, such as in a directory other than the standard, `vendor`.

If so, more than likely, you're going to run into problems running an analysis as, at least on the version I'm using, the tools are configured to expect the dependencies to be in certain locations.

If you can relate to one of these two situations and encounter some problems then here's one suggestion which worked for me. Initialize Composer in the project's parent directory and include the Quality Analyzer as that package's sole dependency. Then, you should be able to run the analysis as normal.

One other thing worth noting is depending on the size of the codebase and your PHP's default memory configuration, you may run into memory allocation exhaustion issues.

These aren't the fault of the Quality Analyzer. These are to do with the individual tools. If you do, have a look at the existing memory configuration and experiment with increasing it to see if that solves the issues.

In Conclusion

And that's been a rapid introduction to Qafoo's Quality Analyzer. While not perfect, from the experience I've had with it so far, I find it a very tidy package, one which makes code analysis quite straight-forward to do.

What's more, it lets any team get started, regardless of previous experience, for no money down; just a bit of time. Given the importance of ever improving code quality, I strongly encourage you to experiment with the Quality Analyzer and to begin learning about source code quality analysis, and how you can use it to improve the quality of your code.

Matthew Setter is an independent software developer, specializing in creating test-driven applications, and a technical writer http://www.matthewsetter.com/about/, focused on security and continuous development. When he's not coding or writing, he loves spending time with his family, cooking, and hiking (oh, and reading a bit too). @settermjd

Become a Better Listener

Cal Evans

This month I'm taking a break from yelling "GET OFF MY LAWN" so that someone who actually has something interesting can speak to you. This month my friend Emily Stamey is going to take over and talk about paying attention to each other. Do me a favor and pay more attention to her than you kids ever do to me.

Being unheard feels awful! You feel excluded from groups and unneeded. These feelings can lead to loneliness, anger, and depression. Our industry can be broadcast-heavy with very little listening. "Write your code this way." "Use this tool; it's the best!" Questions are often met with hostile responses or judgment. This clearly communicates to people their ideas don't matter!

You may believe you are a good listener, but before you move on, please consider there is a lot more to listening than being quiet while others talk. Listening isn't the same as overhearing a conversation. When you are "listening" to a coworker in a meeting, are you also planning your next comment? Are you waiting to continue your comments from earlier? If so, you aren't listening. The conversation you are planning in your head is distracting, and you are missing all or part of the other person's comments.

I enjoy Tara Brach's podcasts and guided meditations. She has a video called the *Sacred Art of Listening*[1]. In the video, she mentioned our attention spans are decreasing as we spend more of our lives on our phones. She asked people to attempt good listening and then describe it. To be listening well, you must first let go. Release the to-do list and be present. Approach listening with the openness of a "beginner's mind." This means you aren't listening while also thinking related ideas or comparing it to something else. You are listening with curiosity and hearing what the other person is saying. Your listening is free from judgment. Listening openly and fully, you practice empathy and establish trust, allowing you to truly learn.

There are many ways we decide whether to pay attention to someone, and many of those matrices are hidden in our brains as unconscious biases. Go90 released a video last year called *Blissful Thinking*[2] where a timer compared the length of time men and women spoke during a conversation between four people. The two men had only a passing interest in the topic while the two women were professionals in that area. The groups never learned that fact through conversation; the men monopolized the conversation and did very little listening.

Poor listening, however, isn't specific to one gender. Senior developers may be accustomed to talking and not listening to their junior devs. We give extra weight to some community members' ideas or call them thought leaders, sometimes discounting our own ideas and experience in the process. What can it do for your peers to be heard by you? What can you learn in the moments you aren't talking? It will change your life and the lives of others to practice good listening. It shows respect. It's a way to understand an entirely different point of view. The way you show you care is to give a person focused attention.

Many factors influence your communication style. How you grew up, your personality, and your preferred activities contribute to how you speak in groups. Some people dominate conversations and are quick to interrupt or finish others' sentences. Others may wait to hear everyone's ideas before speaking. We should be making room in our conversations and meetings for all types of communicators.

Jessica Price shared some great recommendations on Twitter. She suggested meeting leaders ensure everyone is heard from, sometimes prompting quieter team members or

1 *Sacred Art of Listening:* https://www.tarabrach.com/?p=6464

2 *Blissful Thinking:* http://phpa.me/blissful-thinking

cutting off the person filibustering the meeting. If you know you are a talker, you can practice making concise statements and then listening to your teammates who speak next. Listening requires focused attention. Remove distractions from your meetings. Gregg Pollack at Peers Conference shared that he encourages employees to call him out if he becomes distracted by his phone in meetings.

Being a good listener will help you become a better communicator everywhere. Apply these techniques at home, in meetings, and with your user groups. Encourage a quiet member of your user group to share something they're proud of. Put your phones away at meals and enjoy your family and friends. When you listen well, you show you value and care about someone else's ideas. I'm truly looking forward to speaking with and listening to you the next time we meet! <3

– Emily Stamey, @elstamey

Past Events

April

PHP Yorkshire
April 8, York, U.K.
https://www.phpyorkshire.co.uk

Lone Star PHP 2017
April 20–22, Addison, TX
http://lonestarphp.com

DrupalCon Baltimore
April 24–28, Baltimore, MD
https://events.drupal.org/baltimore2017

Upcoming Events

May

PHP Unicorn Conference (Online)
May 4
http://www.phpunicorn.com

phpDay 2017
May 12–13, Verona, Italy
http://2017.phpday.it

PHPKonf Istanbul
May 20, Istanbul, Turkey
http://phpkonf.org

PHP Tour 2017 Nantes
May 18–19, Nantes, France
http://event.afup.org

php[tek] 2017
May 24–26, Atlanta, Georgia
https://tek.phparch.com

PHPSerbia Conference 2017
May 27–28, Belgrade, Serbia
http://conf2017.phpsrbija.rs

International PHP Conference 2017
May 29–June 2, Berlin, Germany
https://phpconference.com

June

CakeFest 2017
June 9–10, New York, USA
https://cakefest.org

PHP South Coast 2017
June 9–10, Portsmouth, UK
https://2017.phpsouthcoast.co.uk

Dutch PHP Conference
June 29–July 1, Amsterdam, The Netherlands
https://www.phpconference.nl

September

Pacific Northwest PHP Conference
September 7–9, Seattle, Washington
http://pnwphp.com

PHP South Africa
September 27–29, Cape Town, South Africa
http://phpsouthafrica.com

October

ZendCon 2017
October 23–26, Las Vegas, Nevada
http://zendcon.com

November

php[world] 2017
November 15–16, Washington, D.C.
https://world.phparch.com

December

ConFoo Vancouvery 2017
December 4–6, Vancouver, Canada
https://confoo.ca/en/yvr2017/

These days, when not working with PHP, Cal can be found working on a variety of projects like Nomad PHP. He speaks at conferences around the world on topics ranging from technical talks to motivational talks for developers @calevans.

Code Review

David Stockton

It is said that to become a better writer, one should read a lot. In a related way, one of the best ways to get better at writing code is to read a lot of code. Not only is it a great way for you to level up your skills, but you can help improve your team and your code as well.

What Is Code Review?

Code review can come in a number of different shapes and sizes. Having a program evaluate your code is one means of code review. It also includes looking at someone else's code or when someone else looks at your code.

You may be familiar with some of the typical automated code review programs. If you're using an IDE like PhpStorm, you may have some these running all the time. Any sort of analysis of problems or potential problems on your code can count as automated code review. Syntax checking is a form of automated code review. This can come in the shape of a squiggly red line indicating there's a problem in your code which will prevent it from executing, or even from compiling. If you're running with strict types enabled, your IDE may be able to tell you about other problems like passing incorrect parameters or returning the wrong values from a method which defines a return type. Even if you're not using strict types, but are using DocBlock comments, many IDEs can help out as well.

PHP_CodeSniffer is a program which can be used to determine if your code conforms to your code formatting standards. In certain circumstances, executing tests against your code could also count as a form of code review. There are other options as well, such as the commercial service Code Climate. If you're interested in other static analysis tools, there's a large curated list available at exakat/php-static-analysis-tools[1].

1 *exakat/php-static-analysis-tools: http://phpa.me/static-analysis-tools*

Let's talk about some common ways code review is done by and for humans, and why many of those ways don't work for improving code quality.

Editors Note: *See this month's Education Station to see how to setup Qafoo for automatic code reviews.*

Email Code and Diffs

One common way to perform code review is to email around code or code diffs. After reviewing the code, any comments or suggested changes can be emailed back to the original developer. I did this for a number of years, but it's not terribly effective. First of all, emails are often ignored. Secondly, I've found that humans tend to interpret written communication, especially written communication which may be critical of something they've done, in the most negative way possible. This seems to be true not only for code reviews but in general. Comments indicating something may be done in a better way, or perhaps the provided code doesn't cover an edge case or may be less than optimal in a number of ways can be read as "your code sucks." Depending on the developer reading the comments, and the level they associate their self-worth with the code they write, the reaction can be anything from apathy to anger or self-loathing to wanting to re-evaluate their career of choice.

In the best of cases, the email may be taken in stride or even potentially ignored. If code review comments are regularly ignored, it may lead to the developers deciding code review is a waste of time. Why should they bother to comment on code when their comments are ignored? Why try to make the software better? Eventually, this leads to the whole process being abandoned and the software heading down a shame spiral of code rot and technical debt. So let's talk about another way to do code review.

Evaluating Diffs in a Repo

With modern repositories like GitHub, Bitbucket, and GitLab, both hosted and self-hosted, this method of review is less problematic than it was just a few years ago. Nowadays, many repositories allow for code review in the repo, but some still do not. In those rare cases, it means someone is looking at the code in the repo. This is done typically by a single developer, probably one of the more senior members of the team. It doesn't work if there's very much activity in the repository, so it typically only works if it's on a small team. And since only one person is doing the code review, it requires diligence and a commitment to ensure issues don't slip by. This could make it difficult for the reviewer to take time off work which leads to burnout. Commenting on changes in repositories like this is often done via email. Please see the previous section for problems with code review via email.

Even with many modern repositories, if the development team is not following a pull request workflow for changes, the repository may not provide any—or at least not any good ways—to comment on changes. If the repository provides for pull requests (or merge requests in some repositories' nomenclature), then

it typically provides for a way to comment on the code in the pull request. However, even if comments can be made on code that is not part of a pull request (perhaps comments made directly against a commit), there is also typically no good way to notify the developer of any issues or questions found. Then, there is little incentive to work on fixing those issues since the code has already been integrated into the main code line.

Group Code Review

Another common way of providing code review is done when a group of developers gathers around a television or a projector in a conference room to review code. This can be a single developer's code or all the code changed during a particular period, or anything in between. There are some problems with this approach. First, it's another meeting, which is time developers are not able to think about or build software. It takes up a lot of time—time which is multiplied by however many developers are in attendance.

> *The code reviews must be about the code. No personal attacks are allowed.*

If the code review is focused on a single developer's code, they can often feel like they are being attacked. With all other developers focused on pointing out problems or issues with the code under review, it may be necessary to include someone in the role of moderator to keep the code review productive. The moderator should ensure the comments and questions are directed towards the code and making the software better, not against the developer who wrote the code. In these cases, even though everyone in the room can see and interpret body language and signals, it can still be a very rough experience for both the developer and the reviewers.

Additionally, there may need to be someone in the role of the note taker. Since comments and questions are made verbally, someone needs to keep track of what was said and the responses as well as any changes that were deemed necessary by the team. Overall, a group code review is often not a good use of time. It can be effective sometimes, but more often than not it's a waste of time, and it's certainly less efficient than other methods.

Over-Shoulder Code Review

This method of code review leaves the original developer in control of the code review. This is typically done with the developer in charge of the mouse and keyboard. The reviewer in this situation is along for the ride as the developer clicks through the code and explains, often very briefly, anything they *think* may be of interest to the reviewer. There's typically little chance to review the code since all that's provided are fleeting glances and jumping between files. If the developer wants to hide certain areas of code because they know the reviewer wouldn't like it or because they are not proud of it, then the overall code quality can decrease even if the reviewer is doing their best to try to help.

As the developer is cycling through code, explaining to the reviewer what they are seeing, any time the reviewer needs to ask a question or make a comment they will be interrupting the developer. This can be frustrating for both parties.

If the code is reviewed this way by more than one developer, then the original developer is losing even more time. This tends to lead toward more rushing, more skipped code, and more glossed-over explanations.

Pull Request Code Review

With a distributed version control system (DVCS) like Git or Mercurial, a pull request code review is one of the better options for reviewing code. As mentioned earlier, modern tools and repository hosting options tend to include some level of pull request review capabilities. If you happen to be still using something like CVS or SVN, then your options are more limited.

With a DVCS, branching and merging is straightforward and easy. This leads to a more natural use of a pull request process. For repositories like CVS or SVN, branching and merging are more complicated, more likely to result in merge conflicts which means a pull request process is less likely. Developers tend to rush to get their code checked in quickly, so they don't have to deal with merge conflicts meaning code review is often skipped.

Repository software has advanced over the past few years including improvements in their ability to facilitate effective code reviews. For the remainder of this article, my comments will include these modern repositories and their ability to assist with code reviews as well as other stand-alone software designed specifically to help with code reviews.

Code Review Software

Code review software (including many modern pull request systems) helps immensely with good code review practices. The software is designed to make code review easy and effective. Because it's software, the reviews can happen when it's more convenient for the reviewer. With meetings and over-the-shoulder reviews, schedules must be coordinated. With review software, the reviews can occur between efforts to build features or fix bugs. Assuming each reviewer takes the same amount of time to review the code as they would have had in a group review, the code review can be more effective.

Code review software allows for reviewers to make comments on the changes and for other reviewers and the original developer to review these comments, as well as replying to or opening issues. Some pull request systems only allow comments to be linked to a single line of code which can lead to some confusion or ambiguity about what the comments are referring to. Better code review software can allow you to select a group of lines and can indicate the comment is about a block of code. If the comment is about more than a single line, it's very clear. It may not seem like much, but any reduction in cognitive load for understanding and resolving code review comments can be very helpful.

Additionally, comments can often be made on the entire changeset as a whole. The ability to reply to comments means conversations can be held, and recorded between the developer and the reviewers. If you recall, I text communication is often taken in the most negative way possible. To combat this, it may be necessary to create ground rules for code reviews. The code reviews must be about the code. No personal attacks are allowed. If the back-and-forth comments and replies grow beyond a certain size, get up and talk to the developer in person.

On teams I've lead, we've had to enact all of these. When I started at my current position, code review was not done so I introduced it. In order to keep things civil, we had to come up with these rules so we could continue to use code review to improve code quality as well as keeping the team working well together.

Approvals and "Ship It"

Not all repositories allow for a "ship it" or an approval, but they are necessary. Some teams have made emojis in comments stand for these approvals. The approvals are a reviewer's way of indicating they are happy with the code and feel it is ready to go to production. In some cases, there may be caveats, like, "Fix this minor thing and then ship it," or "Approved if this thing is done." In those cases, it's the reviewer's way of indicating the code is nearly good enough and the reviewer feels the minor changes don't warrant another review.

> *The code we wrote last week, last month, or last year will not be as good as the code we write today. The code we write today will not be as good as the code we build next week, next month, or next year.*

In some cases, there may be more than a few issues or the changes requested are complex, or the reviewer feels the developer's code may need additional review even after the changes. In rare cases, the reviewer may decide the code submitted for review should not, even with changes, be included into the main code line. Perhaps the code doesn't meet the original requirements, or possibly the requirements have changed so significantly that including the code would move the software in the wrong direction.

What We Do

On the teams I manage, each group of developers and QA have a set of required approvals that must be in place for the code to merge. We follow a pull request with code review process on all teams, but each team has different requirements. On all teams, for the code to merge we currently require it must be one commit ahead and zero behind. This means the pull request branch must be based on the latest code in the target branch. Merging the code is done automatically by software we built that receives events when pull requests are approved and builds finish. Each team must have passing builds from the continuous integration server. This means their unit tests and other integration tests must pass for the pull request branch.

For the code review side of things, each team has their own set of rules. For one team which practices continuous deployment, two approvals are needed for the code to merge. One of the approvals happens when the build finishes successfully. The other approval can come from any of the developers or the testers on the team. Once all of are in place, the code will be merged. We've also built in several DSLs or domain-specific languages that can assist with merging code. If a developer feels strongly there's a problem, they can add a comment like, "Don't merge until I approve." In that case, unless the developer who left the comment has also approved, that code will not be merged. If the original developer or any reviewer wants to make sure another specific developer has reviewed the code, they can leave a comment like, "Don't merge until Dan has approved."

If a pull request relies on another pull request to be merged first, perhaps in another repo, a comment can be left like, "Merge this once <url> has merged," where <url> is a link to the other pull request.

On other teams, the requirements to merge may include more developer approvals and approvals from someone on the QA team. This allows for developers to review and help improve each other's code while at the same time allowing the QA team to have control over what is merged and when. We still

have some of the review rules in place, but now the review process is healthy and very effective.

You Are Not Your Code

One of the most effective ways of ensuring our code reviews are effective is to make sure everyone on the team knows that code reviews happen because we all want to make our products better and improve the software. There are no personal attacks. Developers learn that while they may be proud of what they build, they are not their code. The code we wrote last week, last month, or last year will not be as good as the code we write today. The code we write today will not be as good as the code we build next week, next month, or next year.

I see it most often with junior developers who are just starting out, but also sometimes with senior developers, especially those who have spent a lot of their career working solo, rather than as a member of a team. If developers equate the code they produce with their own self-worth then code reviews feel like a critique of them as a person. If another developer is able to suggest improvements to the code, then that means the developer is not as good as they could be or as good as they thought they were. This is not a healthy attitude or reaction. For this reason, we make sure everyone knows we are all trying to help each other improve. The suggestions we make are to improve the software, not to point out a shortcoming in the developer. No matter how much we each know, every other person on the team knows something we do not, and we know something they do not. You are better at something than each of your colleagues, and they are better than you at something as well. It does not matter how junior or senior you are on the team.

Conclusion

Code review is one of the most important and most effective ways to improve as a developer. In reading, commenting, and critiquing code we not only provide our colleagues with valuable feedback which can help them consider things they may not have, but their comments can help us to improve our code as well. I find it very valuable for developers at all levels to review code produced by developers of all levels. This includes having junior developers review code written by the more senior developers. The reviews may be more questions about why things were done a certain way which can lead to more and better understanding of the code by the junior developer. For reviews from senior developers on code written by more junior developers, the questions can lead to questioning assumptions and making them aware of other ways of building the software they may not have been aware of. If you're not already practicing code review on your team, I cannot recommend getting it started highly enough. See you next time.

David Stockton is a husband, father and Software Engineer and builds software in Colorado, leading a few teams of software developers. He's a conference speaker and an active proponent of TDD, APIs and elegant PHP. He's on twitter as @dstockto, YouTube at http://youtube.com/dstockto, and can be reached by email at levelingup@davidstockton.com.

An OWASP Update—The Top 10 for 2017

Chris Cornutt

The world of the web is changing at such a rapid pace these days it's almost impossible to keep up. There's always some hot new language to learn or some game-changing tool you just have to implement right now. It's hard to figure out the good content and information from the deluge that bombards us every day as technology professionals. The security world is just a slice of this big picture, and even in that small of a section, there's still a going on. Not only are new techniques and tools being created and used every day, but attackers are finding new and even more clever ways to work around our defenses.

What we need is something we can use as a path to help narrow things down and provide a guide to some of the worst of the worst out there. A resource to help us focus on the things we, as web application developers, need to worry about in our day to day work. Fortunately, there's a group which has set one of its goals as doing just that. Enter the OWASP Top 10[1].

What is the OWASP and the Top 10?

You may already be familiar with the OWASP Foundation (Open Web Application Security Project) and their infamous Top 10 list. If you are, feel free to skip over this first part and jump down to the things that have changed in this new edition. OWASP[2] has been around for a long time and has several projects and events under their umbrella all with the same intent—to provide security information and instruction to web application developers to keep them informed and following good, secure development processes.

A large number of projects sit under the OWASP banner, both on the informative and functional sides of the house. The OWASP Top 10 is a list started several years ago to distill the wide world of application security into a set of ten items which should be at the forefront of developers' minds. Each time they do a refresh of the list they look at the most common vulnerabilities out in the wild, how often they're executed, and reassess the current list for changes.

What Topics Does It Cover?

The Top 10 is, well, a list of ten types of security issues a web application could face. It includes both single attack types (like XSS) and more nebulous concepts like, "Using Component with Known Vulnerabilities," (added with the last refresh). For each of the items on the list, the Top 10 gives a brief overview of the problem and breaks it down into measurements for criteria like ease of exploit, detectability, and the potential impact of the issue. Let's look at some examples.

Cross-Site Scripting (XSS)

XSS, or Cross-Site Scripting, is an attack where the attacker can inject content—usually code—and have it executed in the application as the logged in user. For example, say we're expecting a username to come on the URL like so:

```
http://supercoolsite.com?username=foo
```

Then, in our code we're using the `username` value directly:

```
<?php echo 'Hi there ' .
$_GET['username']; ?>
```

The vulnerability happens when an attacker figures out they can control the content of the page directly without it being filtered. They can then create a special link that will, instead of a valid username, inject JavaScript code, doing just about anything they want.

One of the main ways to fix this issue is to correctly escape the output that you're using prior to output. If the value is validated and filtered, you'll resolve 95% of your XSS issues.

Cross-Site Request Forgeries

Cross-Site Request Forgeries (CSRF) attacks occur when a submission isn't properly validated, and it's assumed it came from your own application. Imagine you have a form on your site that takes in information about a banking transfer: to/from accounts, amount, etc. Without some kind of CSRF protection, an attacker could use your credentials to perform a submission without you even knowing about it and transfer money from your account to one of their choosing.

On this one, the mitigation usually starts with the implementation of CSRF tokens into form submissions. These tokens are randomly generated strings which are validated on form submit to ensure the submission did, in fact, come from the correct user and not from a link on a random site.

1 *OWASP Top 10:* http://phpa.me/owasp-2017-draft-pdf

2 *OWASP:* https://www.owasp.org

Broken Authentication and Session Management

This item is a little less concrete than some of the others but it's also broader. While attacks like XSS and CSRF have a clearly defined approach (and usually resolution), broken authentication or bad session management could come in the shape of a lot of things. These are a few examples that fit in this category:

- Not correctly hashing and storing credentials.
- Inconsistent authentication practices across endpoints.
- Allowing session identifiers to be defined.
- Using shared credentials.

There's a lot of different auth related issues that could fall into this category. They can't get much more specific than this, though, because there are just too many variables at play. Authentication/session systems vary so widely across different groups and companies; there's just not a way to say, "This is how you solve this," for every situation. It's more about evaluating your current setup and finding out where the weaknesses are.

Of course, each issue above does have a solution. For example, by now you should know not to store passwords in plain text and instead use `password_hash()` and `password_verify`. Similarly, session identifiers should be sent via cookies and not as part of your application's URLs.

What's Different?

The OWASP Foundation periodically refreshes the Top 10 list to keep it up to date. They've chosen to make their latest release this year with a few category changes to adapt it to the world we work in every day. Two similar categories were combined, and two others were added with focuses on more recent trends and issues security professionals see in their work.

A4 and A7 Merged

The "Insecure Direct Object References" (A4) and "Missing Function Level Access Control" (A7) items were merged. In most modern web applications the "objects" that are referenced are more related to functionality than anything else. This trend has continued, and over the past few years, these two issue types have become more closely related in meaning. Here the "objects" come with functionality when they're accessed or used and them being insecure is just one type of the "missing function level access control" so they merged them together.

This newly merged result is the new "A4" on the list with a more general title: "Broken Access Control." This is the sister issue to "Broken Authentication and Session Management" (A2). This new A4 deals more with *authorization* instead of *authentication* like A2 does.

Figure 1. OWASP Top 10 for 2017

OWASP Top 10 – 2013 (Previous)	OWASP Top 10 – 2017 (New)
A1 – Injection	A1 – Injection
A2 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References - Merged with A7	A4 – Broken Access Control (Original category in 2003/2004)
A5 – Security Misconfiguration	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Sensitive Data Exposure
A7 – Missing Function Level Access Control - Merged with A4	A7 – Insufficient Attack Protection (NEW)
A8 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards - Dropped	A10 – Underprotected APIs (NEW)

Insufficient Attack Protection

One of the new items added to this latest refresh is something that's constantly on any security professional's mind—attack protection. Sure, secure development practices can help to reduce your overall risk, but there are still times you'll be attacked. Unfortunately, most groups are woefully unprepared to handle attacks when they come around, and their code is even worse off.

Let's look a the official definition from the new OWASP document:

> *"The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks."*

The key here is not that the people don't know how to handle an attack (that's a whole other set of problems) it's that the application itself doesn't know what to do when it's attacked. If it's really bad off it may not even know it's being attacked at all. This means implementing things like rate limiting, informed detection based on log information, or automatic responses when attacks are detected.

This is another one of those "it depends" type of Top 10 topics. It is more pointing out a large chunk of the applications don't have any self-awareness or know what to do when—not if, but when—a potential attack comes around.

Underprotected APIs

The other new issue this time is something that's been a long time coming; the addition of API security to the list. While APIs aren't strictly the normal web applications with potentially exploitable frontends, there's still just as much risk that comes with them. Again, we'll start with the quote from the new OWASP guide about this addition:

> *"Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps, that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities."*

The goal is to promote API security to a first class citizen right along side the regular web application development happening over the last several years. API's have become one of the most well-used parts of some applications. There are even services out there which are just APIs. In some services, however, the APIs are treated as a lesser security concern just because they're not the frontend the customer uses from day to day. They can be made up of just as many libraries and packages as the frontend, but because it's "just a backend tool" and only scripts/servers would be connecting to use it, its security falls by the wayside.

APIs aren't going anywhere and securing them is just as important as securing any other part of your application.

Open to Suggestions

This version of the Top 10 2017[3] isn't the final draft (as of the time of this being published it's close though), and OWASP is still looking for comments until the end of June 2017. OWASP is largely driven by volunteer work, and the Top 10 is just one product of that. They're not the end all, be all of the security world, however. If you're reading through this new version and find something that just doesn't quite make sense or you think could be clearer, drop the team an email at OWASP-TopTen@lists.owasp.org with your ideas and they'll get back with you.

The world of the web is evolving, and now the OWASP Top 10 is as well to keep pace. The growth of API-driven development and the huge increase in cyber attacks has spurred on a new level of issues in web applications. Be informed about these threats and be ready for when they come knocking on your door.

For the last 10+ years, Chris has been involved in the PHP community. These days he's the Senior Editor of PHPDeveloper.org and lead author for Websec.io, a site dedicated to teaching developers about security and the Securing PHP ebook series. He's also an organizer of the DallasPHP User Group and the Lone Star PHP Conference and works as an Application Security Engineer for Salesforce. @enygma

3 *Top 10 2017: http://phpa.me/owasp-top10-2017*

Project Creation

Joe Ferguson

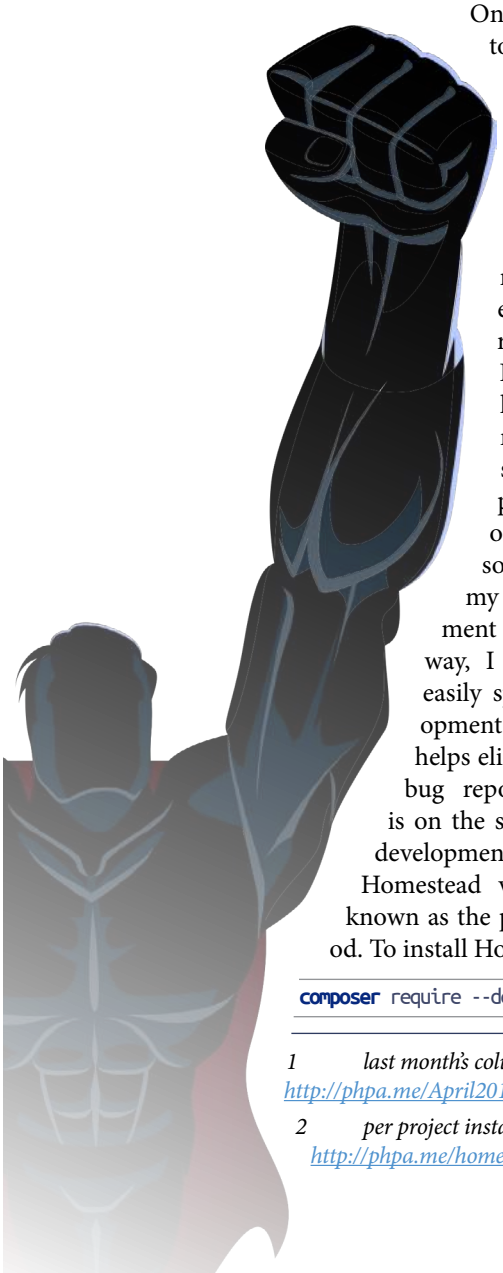
Every developer has a set way of starting a new project. Most frameworks have a linear path to getting started, and Laravel is no exception. With a few commands, you can quickly get started configuring routes, writing controllers, and saving data in a database.

You can install Laravel with Composer by:

```
composer create-project --prefer-dist laravel/laravel new-project
```

My preferred way to create a new project is by using the Laravel installer. You can get the installer by adding it to your system with Composer:

```
composer global require "laravel/installer"
```



Once installed you can use it to create a new project in a directory you specify. `larval new project` will create a new folder called `project` and install the latest version of Laravel.

The second command I run on all of my Laravel projects adds Homestead. As you read in last month's column[1] Homestead is my default local development environment. I install Homestead slightly different than many people. Often, I am working on projects I intend to open source later, so I always bundle my local development environment with my project files. This way, I know my contributors can easily spin up the same local development environment I am using. This helps eliminate many false errors and bug reports, and ensures everyone is on the same playing field during the development of the application. I install Homestead via Composer. This is also known as the per project installation[2] method. To install Homestead via Composer run:

```
composer require --dev laravel/homestead
```

1 last month's column: http://phpa.me/April2017issue

2 per project installation: http://phpa.me/homestead-per-project

One of the reasons I enjoy working with Laravel is its great developer experience. As web developers, we are often worried about the user experience (as we should be). Laravel always impresses me with the focus on making my job as a developer easier. Most full stack frameworks came about from the same reason: a developer was tired of writing boring boilerplate code for every project and decided to bootstrap pieces together. The power of packages and Composer allows that to happen even easier than it did years ago. Laravel allows developers to jump right into the business logic, the fun stuff of the application they're building.

Artisanal Frontend Development

My least favorite part of web development is frontend. I have never had an eye for design, and while I have developed a sense of what to look for in creating valuable user experiences, I do not enjoy it. I do like JavaScript, but I don't enjoy the rate of change of the frontend ecosystem. This is one of my favorite parts of Laravel: Laravel Mix. Mix is a collection of packages and pre-built workflows that take the pain out of nearly all the frontend bootstrapping of your application. Mix will give you Bootstrap, jQuery, and Vue support right out of the box. Coming in Laravel 5.5 there are even helper commands which will configure a Vue.js frontend or ReactJS. Laravel Mix removes a lot of the frustration of figuring how to do something today when it comes to frontend development.

Laravel 5.4 comes ready to work with Vue.js via Laravel Mix. All you need to get started is to run:

```
npm install
npm run dev
```

Alternatively, you can also use `yarn` instead of `npm` if you prefer. These two commands will install all the front end dependencies to the typical `node_modules` folder in your project and compile all the source assets from the `resources/assets` folder into compiled files in your `public/` folder. You will use the compiled versions of the CSS and JavaScript files in your templates and layout views. When you deploy to development, you will want to run `npm run production` so a few extra steps are taken to minify and optimize your files for production to ensure the best possible frontend performance of your application.

PHP[WORLD] 2017

This is a conference like no other. Designed to bring together all the communities linked by the PHP programming language.

Together as the PHP community, the sum is greater than the whole.

November 15-16, 2017
Washington, D.C.

world.phparch.com

Artisanal Testing

Once I have my development environment and frontend layout work configured, I start writing tests. I'm not a Test Driven Development (TDD) evangelist by any means. However, I know the value of tests and often find it easier to start writing basic tests I know will fail and then begin to work out the logic to make the tests pass. Some of you that are TDD veterans will know the Red, Green, Refactor mantra; I really like that approach.

Laravel Dusk is the newest testing package on the Laravel block, and it really takes the already impressive ease of testing to an entirely new level. Dusk utilizes the ChromeDriver[3] package to run tests in a real Chrome browser. When you run your Dusk test suite, you'll see Chrome open browser windows and execute the tests in front of you. It will send detailed error reports when something goes wrong. You can also run these tests from inside Homestead as of version 5.2.1. Obviously, because Dusk is using a real browser, these tests will take longer to process. Also, keep in mind these are *not* unit tests. Laravel Dusk tests are acceptance tests. Acceptance testing is often used to verify a product is working as expected. The tradeoff of using acceptance tests is they are slower and often don't show exactly where the problem may be. Acceptance tests are *not* a replacement for unit tests. They should be used together to ensure you have high test coverage of your application so when a bug does show up it is easier to find and fix.

One downside to Laravel Dusk is there is no way to reset the database state after every run. Laravel 5.3 test helpers

Listing 1

```
"autoload-dev": {
    "classmap": [
        "tests/TestCase.php",
        "tests/BrowserKitTestCase.php"
    ],
    "psr-4": {
        "Tests\\": "tests/"
    }
},
```

(now known as `BrowserKitTesting`) could use Traits that would reset the database after a test had completed, meaning you were free to do what you wanted with the database and it would be reset back the state it was before the tests ran. When I do acceptance testing, I like to test everything going in and out of the database is as I expect. This requires resetting the database after each run. You can manually seed a second test database with an SQL file import each time Dusk runs. This is how I've managed to utilize the power of Dusk's browser testing and still not have to worry about altering my database.

As mentioned, the Laravel 5.3 testing helper functionality has been renamed to a `BrowserKitTesting` package you can easily add to your project via:

```
composer require –dev laravel/browser-kit-testing
```

You will then need to update your `composer.json` as in Listing 1.

Now you can write the Laravel 5.3 style tests. Make sure your tests extend the `BrowserKitTestCase` class. You can also copy tests from Laravel 5.3 to 5.4 applications with this method.

The advantage of using the BrowserKit Testing package is you can easily write functional tests which do whatever you want to the database and easily clean up after themselves. Since they're not using the ChromeDriver, they also run slightly faster than the Dusk tests.

Artisanal Routing

Laravel 5.4 has four files in the `routes` folder to handle routes. If you are new to Laravel, you should only worry about `web.php` and `api.php` since these are the HTTP routes and API routes respectively. Once you get farther down the path with Laravel and need event broadcasting, you'll refer to the `broadcast.php`. Likewise you may never need `console.php` unless your application is leveraging several complex Artisan commands.

Laravel routing syntax is very expressive and easy to read (just like the entire framework). The example route can be seen here:

```
Route::get('/', function () {
    return view('welcome');
});
```

This is a simple GET route which returns a callback. This callback returns a view named `welcome.blade.php`. Because we are using the view helper in our callback, we do not have to specify the full path to the views folder, nor the entire filename of the view. Routes can use any of the HTTP verbs such as GET, POST, PATCH, PUT, or DELETE.

Route parameters allow you to capture a part of the URI to pass into your callback. A common route you would expect to return a display view of a widget may look like:

```
Route::get('/widgets/{id}', function ($id) {
    $widget = Widget::find($id);

    return view('widgets.view')
           ->with('widget', $widget);
});
```

This route passes the ID value from the URI to the callback, and the callback attempts to find the widget matching the ID from the URI with an ID in the database. If a match is found, the row is returned and passed to the widgets view to be rendered.

Naming routes is the easiest way to keep your routes organized so a refactor down the road is even easier. The following

3 *ChromeDriver:* http://phpa.me/chrome-webdriver

is an example of a named route:

```
Route::get('/widgets/{id}', function ($id) {
   $widget = Widget::find($id);

   return view('widgets.view')
         ->with('widget', $widget);
})->name('widget.view');
```

This allows you to easily route to this function by its name, such as when you want to redirect a user:

```
// Return a redirect to widget.view route
return redirect()->route('widget.view');
```

If you ever needed to change the URI of this route you would only have to update the route in the routes file, not in many different places in your application.

When building an application that needs a management or administration control panel, you would normally group all of those routes behind an `/admin` route. Such as `/admin`, `/admin/widgets` or `/admin/tasks`. This can be easily accomplished by using route prefixing to keep your routes clean and readable.

```
Route::group(['prefix' => 'admin'], function () {
    Route::get('/', function ()    {
        // Matches The "/admin" URL
    });
    Route::get('widgets', function ()    {
        // Matches The "/admin/widgets" URL
    });
});
```

You can also assign middleware to routes. We mentioned creating a route prefix for grouping routes behind an `/admin` prefix. Assuming these routes allow administrative tasks it would make sense to use the `auth` middleware to secure these routes. The `auth` middleware simply requires the request to be from a user that has logged in. This keeps unauthenticated users from accessing any of these routes.

```
Route::group(
   ['prefix' => 'admin', 'middleware' => 'auth'],
   function () {
      Route::get('/', function () {
          // Matches The "/admin" URL
      });
   }
);
```

Adding Middleware to Route Groups

By now your routes file is filling up with a lot of things which don't need to be there, like all of those callbacks. Callbacks are great for small things and even for testing ideas for logic before committing them to any needed abstraction layers. I often keep a `/test` route in many of my applications for this very reason during development. However, once you're happy with the functionality, you'll want to move those callbacks to a controller method which prevents your routes file from becoming a hard to read thousand line monstrosity. This also allows a clean separation of duties and results in much easier to read code.

We can refactor our earlier route by creating a `WidgetsController` via the Artisan command:

```
php artisan make:controller WidgetsController
```

The Artisan command will create a basic boilerplate controller for us so we can just drop in our logic from the callback. Listing 2 goes in the `app/Http/Controllers` folder.

Since we have moved our callback into a controller method named `viewWidget` we can now update our route:

```
Route::get(
   '/widgets/{id}', 'WidgetController@viewWidget'
)->name('widgets.index');
```

Now we can clean up our routes file and start abstracting our callbacks into controller methods.

Cross-Site Request Forgery Protection

Laravel routing automatically handles Cross-Site Request Forgery (CSRF) checking. Any route that points to POST, PUT, or DELETE must have a CSRF token in the form data. You only have to ensure you use the view helper `{{ csrf_field() }}` in your form, no need to do anything on the server side processing. Laravel will automatically reject the request if the tokens do not match.

Artisanal Databases

Whether you are a die-hard Postgres or MySQL fan, Laravel has you covered. Laravel ships with support for just about any flavor of database being used by modern (and in some cases legacy) web development.

The basics of connecting your Laravel application to a database is to create a migration. If you have never worked with database migrations, they are simply PHP class files that tell an ORM (Object Relation Mapper)—Eloquent, in our case—what to do with a database schema. A migration has two methods: `up` and `down`. The `up` method is executed when

Listing 2

```
1. <?php
2.
3. namespace App\Http\Controllers;
4.
5. class WidgetController extends Controller
6. {
7.    public function viewWidget($id)
8.    {
9.       $widget = Widget::find($id);
10.
11.      return view('widgets.view')
12.         ->with('widget', $widget);
13.   }
14. }
```

Listing 3

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateWidgetsTable extends Migration
{
    public function up() {
        Schema::create('widgets', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->text('description');
            $table->float('price', 8, 2);
            $table->timestamps();
        });
    }

    public function down() {
        Schema::dropIfExists('widgets');
    }
}
```

you run `php artisan migrate` and the `down` method is executed when you run `php artisan migrate`. Whatever you do in the `up` method should be reversed in the `down` method. An example migration to create our widgets table can be found here in Listing 3, found in the `database/migrations` folder.

The usefulness of the down method is frequently debated; some people advocate there is no practical reason you would ever roll *back* a migration in production but instead always roll forward to prevent data loss. Having worked on Laravel applications with sixty or more migration files spanning over a year, I can certainly agree you may never need all of those down statements. I always recommend getting comfortable with writing up and down methods. The down method will force you to think about what you're doing to the database in reverse and often times you may discover a bug or some other planned feature may have different needs.

To apply the changes in your migration run the command:

```
php artisan migrate
```

Now we are ready to move on to the model. A model is a class used to store and retrieve information about a particular database table. We can easily create a new model for our widgets table:

```
php artisan make:model Widget
```

Listing 4 shows our model stored in the `app/` folder.

We have added the protected array `fillable` to allow us to set those fields elsewhere in our application dynamically. If you do not set the fields and try to assign them you will get an error (empty data saved). Note: you do not have to specify the `timestamp` field in your model; the framework handles these fields for you automatically.

Now we have a migration which has created a table for us—a model that represents our table and will allow us to store and retrieve data—we can add a model factory. Model factories are somewhat new to the Laravel framework, and I find them extremely useful for creating sample data for your database based on your models which is even more useful for writing your tests. As you can guess, a model factory is a function you can call to create an instance of your model.

Model factories are defined in the `database/factories/ModelFactory.php` file. You can see an example of the `User` model here.

Listing 5 shows our model stored in the `database/factories/ModelFactory.php`.

We are leveraging the use of the package Faker[4] to create fake data for our widget. This callback will create a new instance of a widget and save it to

4 *Faker:* https://github.com/fzaninotto/Faker

Listing 4

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Widget extends Model
{
    protected $fillable = [
        'name',
        'description',
        'price',
    ];
}
```

Listing 5

```
/** @var \Illuminate\Database\Eloquent\Factory $factory */
$factory->define(
    App\Widget::class,
    function (Faker\Generator $faker) {
        return [
            'name' => $faker->word . ' Widget',
            'description' => $faker->paragraph(3),
            'price' => $faker->randomFloat(2, 20, 999999),
        ];
    }
);
```

the database using fake data. Faker will create a random word, paragraph description, and price for our new widget. This allows us to create sample data easily.

The first place I normally utilize model factories is in my database seeder. I don't want to use production data during development, so I combine model seeders to create fake data which mimics production data and uses database seeders to create a realistic data set across all of my models. You can create individual data seeder class files, or you can use the existing `DatabaseSeeder.php` from the `database/seeds` folder. To create 20 widgets in our application, we can use a model factory in the database seeder file: `database/seeds/DatabaseSeeder.php` shown in Listing 6.

Similar to running our migration, we can seed our database by running an Artisan command:

```
php artisan db:seed
```

Now, if we inspect the database, we can see 20 rows of sample widget data. You can run database seeds as many times as you want; they will continue adding new rows each time.

If something happens to your database you can easily reset everything with `artisan`:

```
php artisan:reset
```

This will rollback all of the migrations, and you can run the migrate and seed commands again to set up your data.

Listing 6

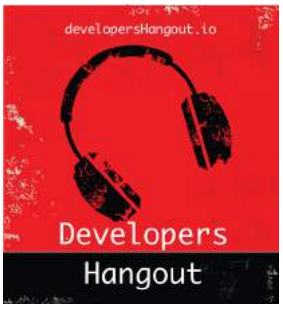
```
1. <?php
2.
3. use Illuminate\Database\Seeder;
4.
5. class DatabaseSeeder extends Seeder
6. {
7.     public function run() {
8.         factory(App\Widget::class, 20)->create();
9.     }
10. }
```

From here, you're ready to go forth and start building fresh baked Artisanal applications with Laravel. You can easily set up routes, controllers, and start accessing data in a database. I look forward to seeing what you build.

Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. @JoePFerguson

Welcome to php[architect]'s new MarketPlace! MarketPlace ads are an affordable way to reach PHP programmers and influencers. Spread the word about a project you're working on, a position that's opening up at your company, or a product which helps developers get stuff done—let us help you get the word out! Get your ad in front of dedicated developers for as low as $33 USD a month.

To learn more and receive the full advertising prospectus, contact us at ads@phparch.com today!



Listen to developers discuss topics about coding and all that comes with it.
www.developershangout.io

The PHP user group for the DC Metropolitan area
meetup.com/DC-PHP

Kara Ferguson
Editorial

Refactoring your words, while you refactor your code.

@KaraFerguson
karaferguson.net

technology : running : programming
rungeekradio.com

The Frederick Web Technology Group
meetup.com/FredWebTech

April Happenings

PHP Releases

PHP 7.1.4:
http://php.net/archive/2017.php#id2017-04-13-2

PHP 7.0.18:
http://php.net/archive/2017.php#id2017-04-13-1

News

Stefan Koopmanschap: To Exception or not to Exception

In the latest post to his site Stefan Koopmanschap offers some advice on when to use exceptions and when to avoid them (the result of a recent Twitter discussion). He goes on to define the term "program flow" and how that relates to the idea of using exceptions to control it. http://phpdeveloper.org/news/25129

Nikita Popov: PHP 7 Virtual Machine

Nikita Popov has a new post to his site sharing a look behind the curtain of how the PHP 7 virtual machine works, the latest version in the Zend Virtual Machine that powers the language. There's lots of information here and it's definitely interesting to see what happens inside the language to create the fast and functional PHP 7 applications we have now. http://phpdeveloper.org/news/25111

Exakat Blog: Moving from Array to Class

In a new post to the Exakat blog there's a proposal to replace uses of arrays with classes to make scripts more efficient and handle resources better behind the scenes. He mentions some of the recent changes in PHP 7 that make the use of classes over arrays a bit more advantageous. He then gets into how to take advantage of these efficiency benefits in moving from arrays to classes. http://phpdeveloper.org/news/25095

Hackernoon.com: Automatically Running PHPUnit With Watchman

On the Hackernoon site today Sebastian De Deyne has written up a tutorial showing you how to use Watchman to automatically run PHPUnit tests for your application when things change. Watchman is a tool from Facebook that watches files and directories for updates and execute actions based on the changes. In the setup he creates, Watchman is used to look for changes on files in either the project's `src/` or `tests/` directories and execute a bash script (code provided) that runs the tests and outputs the results. http://phpdeveloper.org/news/25094

Russell Walker: Is Best Practice Actually Poor Practice? Dependency Injection, Type Hinting, and Unit Tests

Russell Walker has a post to his site sharing his thoughts defending dependency injection, type hinting and unit testing against some of the common objections. He then breaks the rest of the post down into a few of the common objections and makes an attempt to set the record straight. http://phpdeveloper.org/news/25071

Fabien Potencier: Symfony 4: Monolith vs Micro

Fabien Potencier is back with a new post on his site following up this article about application composition and Symfony 4. In his latest post he compares two approaches to applications: micro versus macro. He talks about changes upcoming in Symfony 4 including the move away from the "symfony/symfony" package system and in with a component/bundle driven system. He gets into a specific example around the "symfony-framework" bundle. http://phpdeveloper.org/news/25067

Tomas Votruba: Why Is Doctrine Dying

In a recent post to his site Tomas Votruba shares some of his opinions about why he thinks that Doctrine is dying sharing three of the reasons he sees for this trend. He starts off by stating that Doctrine is "an awesome tool" but suggests that it is stuck in its legacy world and hasn't been able to evolve much past some of its original functionality. In his opinion this is because of the project's "system setup" not the code quality or maintainers. http://phpdeveloper.org/news/25065

Simon Holywell: PHP and Immutability—Part Two

Simon Holywell has continued his series looking at immutability and PHP in part two of his series improving on the code and classes from the previous post. He then moves on from the "simple" mutation method previously used (making a new immutable object when a property changes). http://phpdeveloper.org/news/25064

Happiness is a Boring Stack

Eli White

Does the title of this article sound familiar to you? Well, I fully admit that I have borrowed this month's title from a blog post, *Happiness is a Boring Stack*[1], which I recently read by Jason Kester, @jasonkester[2], with the same title. The article, in summary, talked about how Jason enjoyed staying on the cutting edge to keep advancing with the industry, but that on his projects he used the most basic boring stacks possible.

This concept resonated with me, as being one of the resident old curmudgeons of PHP[3] it is absolutely how I operate. In fact, it is precisely how I have pushed most every company I have worked for to run for years and to good effect.

Yes, the cutting edge is exciting, and it *is* in fact, important in our industry to stay abreast of where technology is going. If you do not keep up, you easily will fall behind. It is one of the reasons why I attend numerous conferences a year (as well as run 2-4 of them). Going to conferences is one of the best ways that I keep on top of what the industry, both narrow and broad, is doing.

I absolutely love conferences, because in a few days you can get a full dose of "ALL THE THINGS" thrown at you, without it impacting your day-to-day work in most cases.

However, like Jason, when it comes to building the next website that I am making? I am back to whatever my current "tried and true" solution is. Maybe it is that homegrown framework that I have used on the last X projects. Maybe it's a stock framework you've become familiar with such as Laravel or Symfony. Maybe it is tossing up a WordPress install and just getting it done quickly with some vetted plugins.

The fact is, there are lots of benefits of keeping your projects on "boring" stacks. First of all, it is good to make sure that the projects that are providing you money, are ones where you can understand and debug them intrinsically. You do not want to spend excessive time when on the clock trying to get a new feature out the door or on fixing the bug fix keeping your customers from paying you money!

> *"I also find doing the mundane, everyday things in life has a calming, creative influence on me."*
>
> — *Gail Tsukiyama*

Keeping your experimentation on the side as you learn about new things while focusing on doing what you know works and works well on the paying projects, gives you the best of both worlds.

However, perhaps one of the best reasons to do this, is that the 'boring' technology happens to be one of the longest running and most stable solutions out there. Heck, it is why PHP is such a popular choice for projects now because it has been around, everyone knows it, and there are some great base solutions built on top of it such as WordPress or Drupal.

I have been at companies where we were running absolute cutting edge technology. As we kept pushing the technology further than it was ever designed for we continually found bugs and undocumented limitations. It is a remarkably frustrating situation when you have built your entire solution upon a technology, making it an integral part of your workflow, just to find it begins to fail and there is nothing you can do about it.

If you have a boring stack, it means there is less technology for your fellow developers to learn as well. I have been at companies where just having the stack explained to you is a 2-hour long ordeal that involves charts and a powerpoint presentation. When all you asked was: "Where is the API endpoint stored?" Keeping things simple ensures that developers can understand what is going on, and perhaps just as important be familiar with all aspects of the project. Theres' nothing worse than having that 'one section' of your application only two people understand because it is written in a different language on top

1 *Happiness is a Boring Stack:* http://phpa.me/happiness-boring-stack

2 *@jasonkester:* https://twitter.com/jasonkester

3 *Apologies to the OG curmudgeon of PHP, Cal Evans,* @calevans

of a different tool that is experimental to begin with. (And then both of those people end up on vacation the day it breaks)

Of course, if you are hiring, there's a bit of a double-edged sword. On the one hand, if you are using a very stable technology (such as PHP), it means you will have no trouble finding programmers who know the technology. On the other hand, you may have trouble attracting certain types of developers who are craving working with the latest technology at all times. (Though I would argue that perhaps you do not want those people on your team anyhow, as they would never be satisfied with a 'boring stack' and constantly try to change it for purely change's sake)

Does this mean that you should never embrace new technology and move forward? Certainly not. However, when you make a decision to introduce a new technology on a project, it should be because it is absolutely the right tech for the job and your solution wouldn't be possible using your boring stack.

So here's to working on boring stacks, which pay the mortgage and keep the lights on. May we long be able to keep them going, while getting to enjoy the exciting new technology on the side. Preparing us for the future when the time is right to use them.

Don't feel bad about having a boring stack. Embrace it and love it, knowing that it will be there for you chugging along tomorrow, and for every tomorrow after that.

Eli White is a Conference Chair for php[architect] and Vice President of One for All Events, LLC. He thrives in making his work being as boring and mundane as possible, so that he can get it done and enjoy the exciting things in life (and technology) that there is to still learn! @EliW

SWAG

php[architect]

Our CafePress store offers a variety of PHP branded shirts, gear, and gifts. Show your love for PHP today.

www.cafepress.com/phparch

ElePHPants

Laravel and PHPWomen Plush ElePHPants

Visit our ElePHPant Store where you can buy purple or red plush mascots for you or for a friend.

We offer free shipping to anyone in the USA, and the cheapest shipping costs possible to the rest of the world.

www.phparch.com/swag