

# MODERNIZE YOUR PRODUCTIVITY

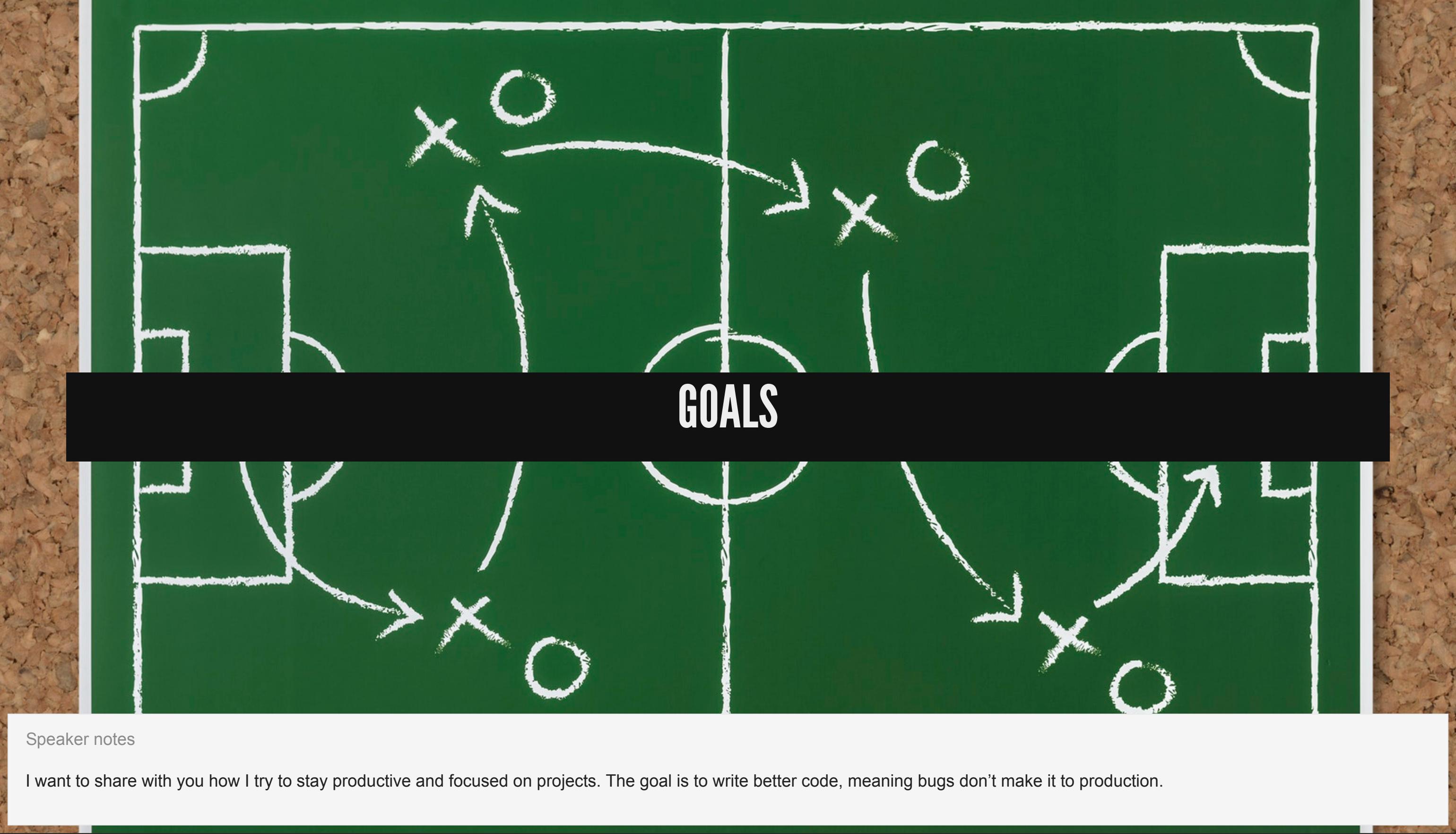
Oscar Merida, @omerida

September 20, 2018

**HELLO**

# SLIDES

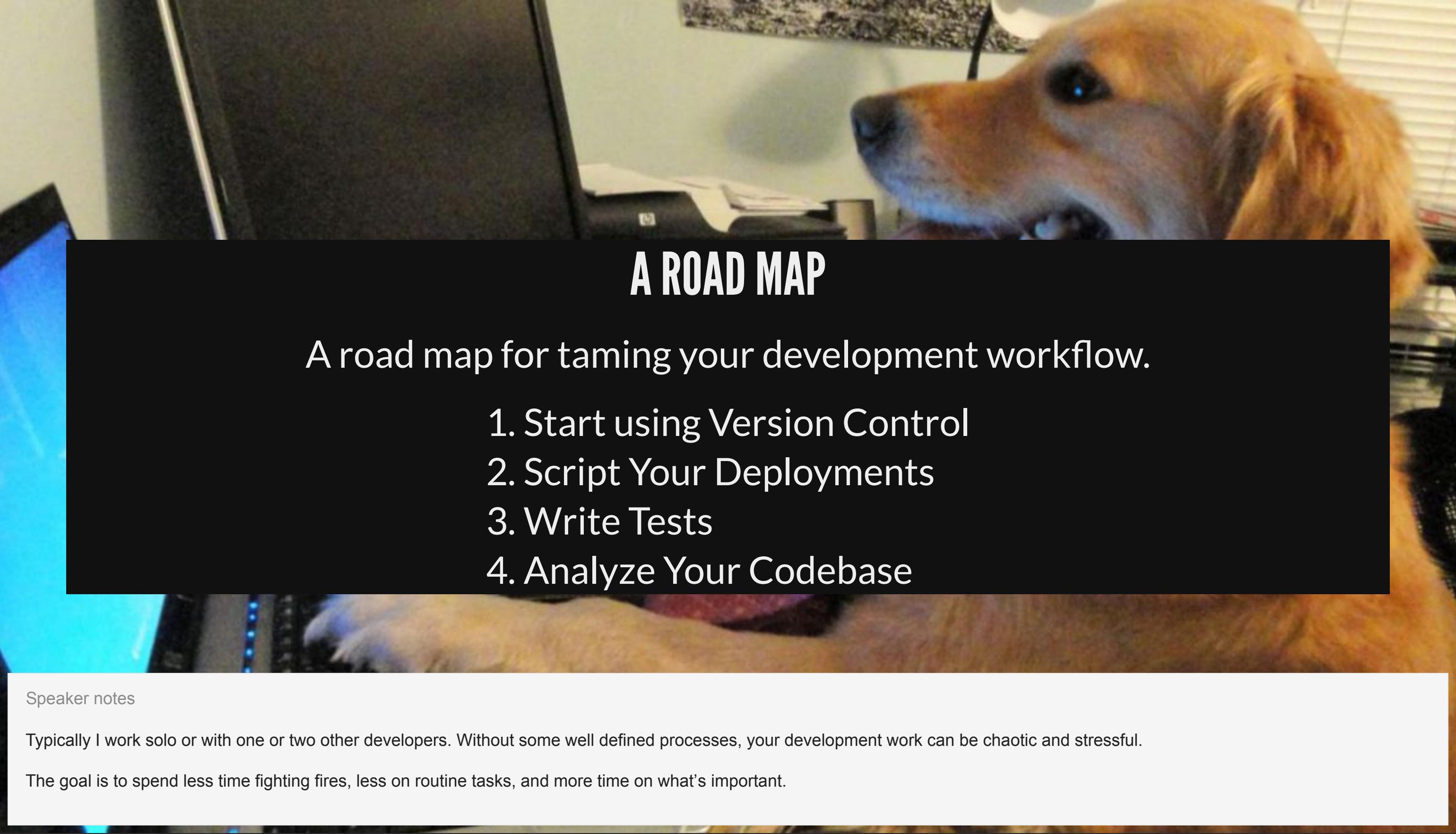
<http://phpa.me/modernize-productivity>



# GOALS

## Speaker notes

I want to share with you how I try to stay productive and focused on projects. The goal is to write better code, meaning bugs don't make it to production.

A golden retriever dog is sitting at a desk, looking towards the left. In the background, there is a computer monitor and a keyboard. The dog's head is in the upper right corner, and its front paws are visible near the keyboard in the lower left. A black semi-transparent box is overlaid on the center of the image, containing white text.

# A ROAD MAP

A road map for taming your development workflow.

1. Start using Version Control
2. Script Your Deployments
3. Write Tests
4. Analyze Your Codebase

## Speaker notes

Typically I work solo or with one or two other developers. Without some well defined processes, your development work can be chaotic and stressful.

The goal is to spend less time fighting fires, less on routine tasks, and more time on what's important.

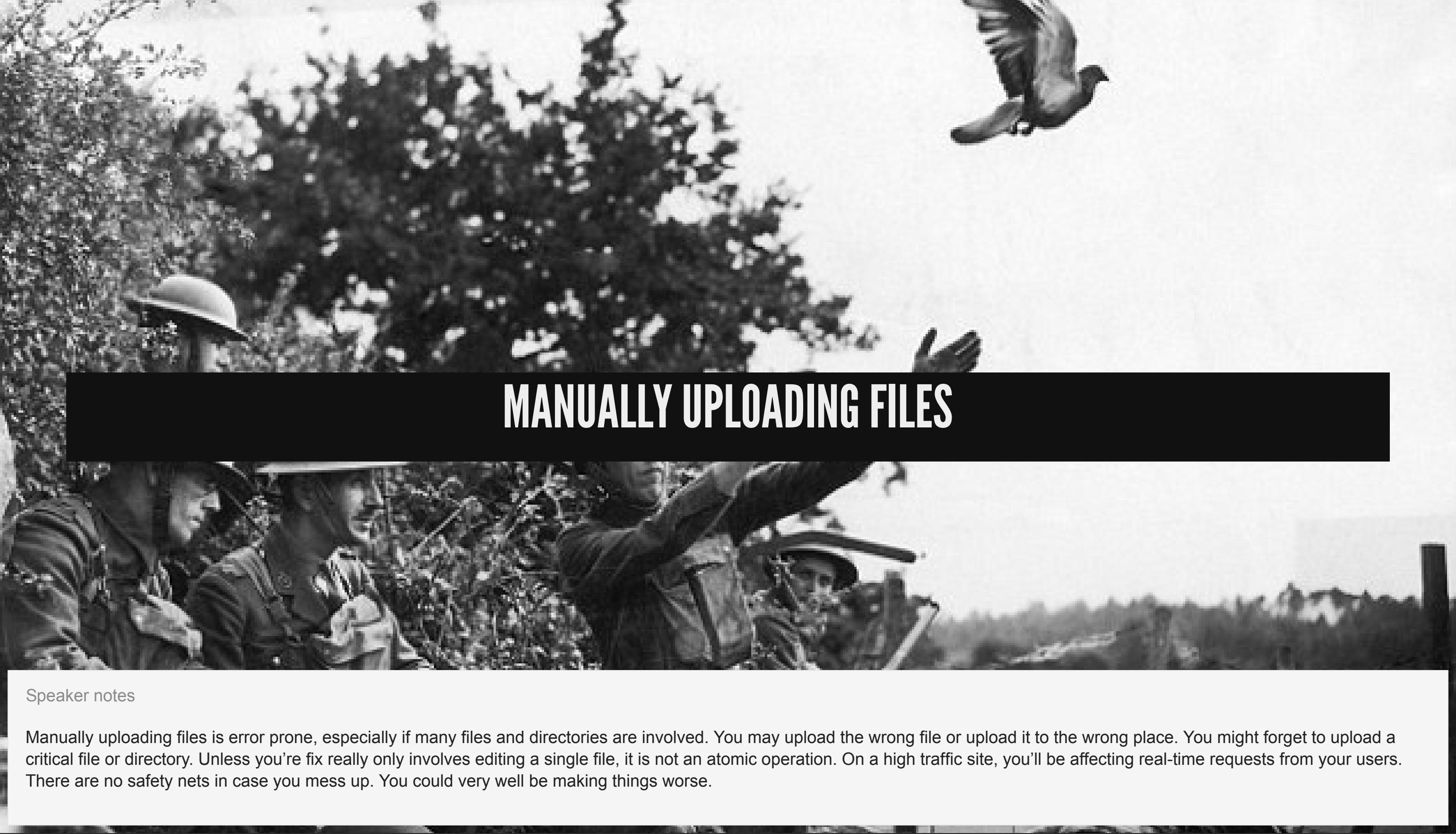
# **THE BAD OLD DAYS / POOR PRACTICES**

# EDITING IN PRODUCTION



## Speaker notes

You should never edit code directly on your production site, either to fix a bug or try to identify the cause. If you can't replicate a bug locally, how do you know you're fixing the right thing? How are you going to write a test or properly document the fix? How are you going to prevent the bug from happening again in the future? You need to be able to control the INPUTs to your application and the environment it runs in (PHP version, Web Server, etc) to diagnose the causes of a bug or unexpected behavior. But ultimately, the problem is that this is highly risky (you could alter or delete data) and disruptive (you could affect behavior for real users).



# MANUALLY UPLOADING FILES

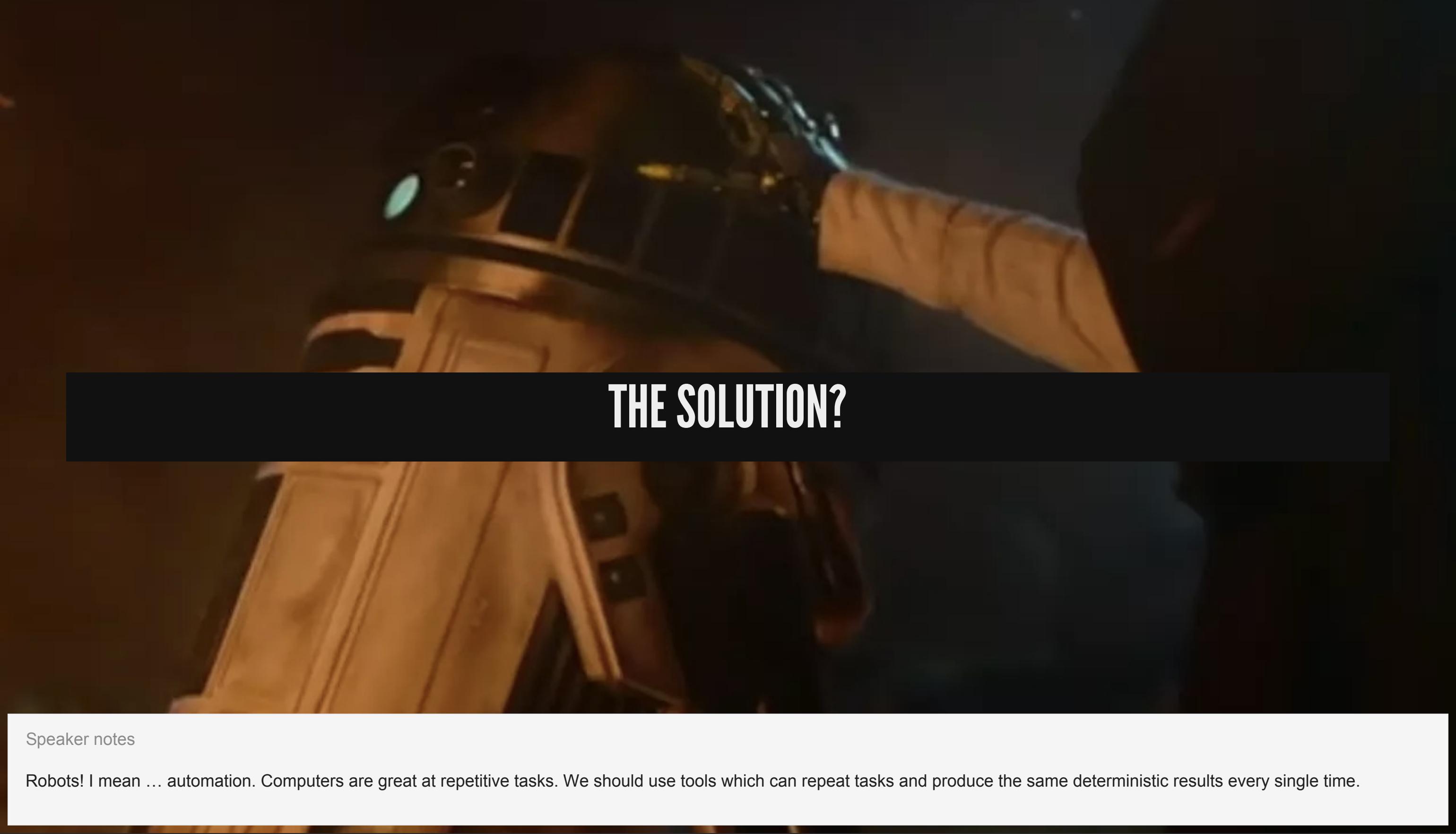
## Speaker notes

Manually uploading files is error prone, especially if many files and directories are involved. You may upload the wrong file or upload it to the wrong place. You might forget to upload a critical file or directory. Unless you're fix really only involves editing a single file, it is not an atomic operation. On a high traffic site, you'll be affecting real-time requests from your users. There are no safety nets in case you mess up. You could very well be making things worse.

# NO SOURCE OF TRUTH

## Speaker notes

If your production server is your canonical representation of your application's code ... how do you replicate the environment or know your dependencies—like what version of PHP you use and if it's safe to upgrade to a new version? How do you analyze the codebase? How do you track what changes have been made over time, and who made them? For security, can you find all the passwords and API tokens used by your application? Are they outside the web root?



# THE SOLUTION?

Speaker notes

Robots! I mean ... automation. Computers are great at repetitive tasks. We should use tools which can repeat tasks and produce the same deterministic results every single time.

Master

Develop

Feature

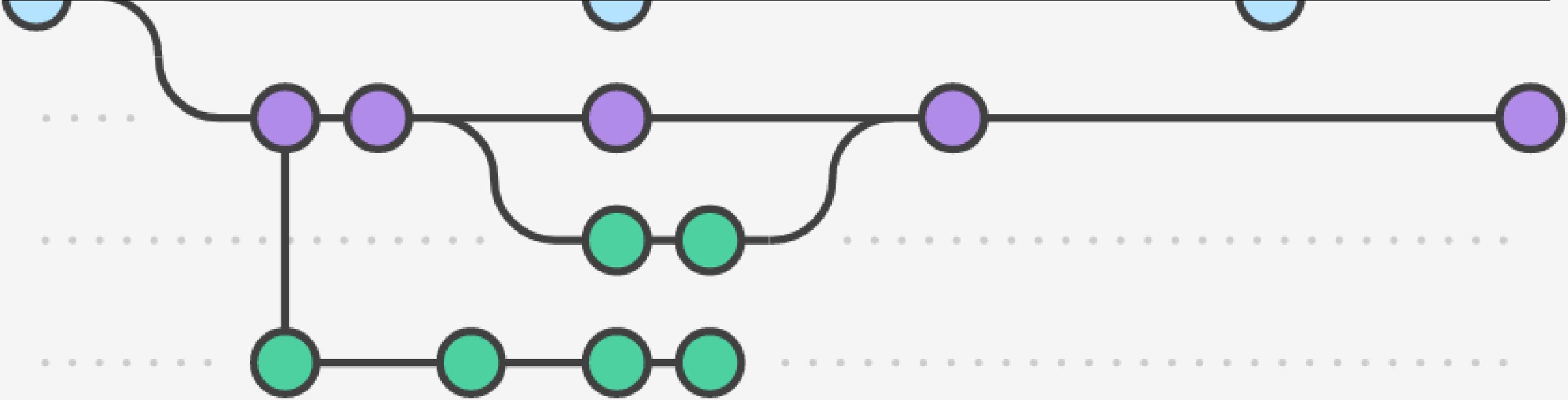
Feature

v0.1

v0.2

v1.0

# VERSION CONTROL



# START USING GIT

<https://git-scm.com>

Create a local repo.

```
git init
```

## Speaker notes

A local repository saves snapshots of your codebase on your own machine, eventually you'll want to share those changes elsewhere, but you don't need that at the start.

Note git is not GitHub (or gitlab), you don't need the latter to start using git.

# TRACKING CHANGES

Add a changed file, commit the change.

```
git add foo.php bar.php  
git commit -m "Fixed SQL injection bug"
```

## Speaker notes

If you're comfortable at the command line, basic git usage comes with a little practice. I use the Git integration with PhpStorm, but any decent IDE or code editor will have a plugin for Git integration

# BRANCHES

Create a branch and start working on it

```
git checkout -b 53-db-updates
```

Merge a branch

```
git checkout master  
git merge 53-db-updates
```

## Speaker notes

A branch lets you work on a single task—or experiment in some direction—without affecting your “main” codebase. You can try things out, and if they work you merge it back to the master branch. You can share a branch with other repositories, or it could never leave your local repo.

If you’re just starting, this is sufficient. Eventually, if you have a lot of commits and merges, you’ll want to squash commits and rebase your changes to keep your commit history cleaner.

# SHARING CODE

Setup a repository to push to. [GitLab](#) has free, private repos.

```
git remote add gitlab https://server/namespace/project.git
```

After you commit:

```
git push gitlab
```

To retrieve changes

```
git pull gitlab
```

## Speaker notes

A typical workflow is to have a “blessed” repository which receives the changes made by all collaborators. A push will send changes from one or more branches to those branches on the shared repo. From there, another user can pull (fetch+merge) those changes to their repository.

# A GIT WORKFLOW

Use branches to reflect your deployment environments. `master` could be your development branch and `production` your live site.

1. When working on new features or bugfixes, create a feature branch.

`118-new-account-email`

2. When work is ready for testing or live, merge the feature branch into `master` for testing then to `production` for release.

## Speaker notes

There are many ways to work with git. They all have their strengths and weaknesses, but I like gitflow for my projects.

It takes a little discipline, but following this even if you're the only developer keeps your work more organized. You can easily work on multiple requests without mixing them together. Imagine your working on adding a new feature to an ecommerce site when a critical bug is found in the shopping cart checkout proces. You have to stop to fix the bug, so sales don't stall. If you use feature branches, each of these tasks is isolated. There's no risk of your new feature affenting the new fix and you can deploy each independently.

For this to work, it's also important to write good commit messages and to have an issue or ticket for every task. Down the road, its helpful to lookup an issue for a given commit to get context and this is also useful if you have to prepare reports about what you've worked on.

# GIT HOOKS - PHP LINTING

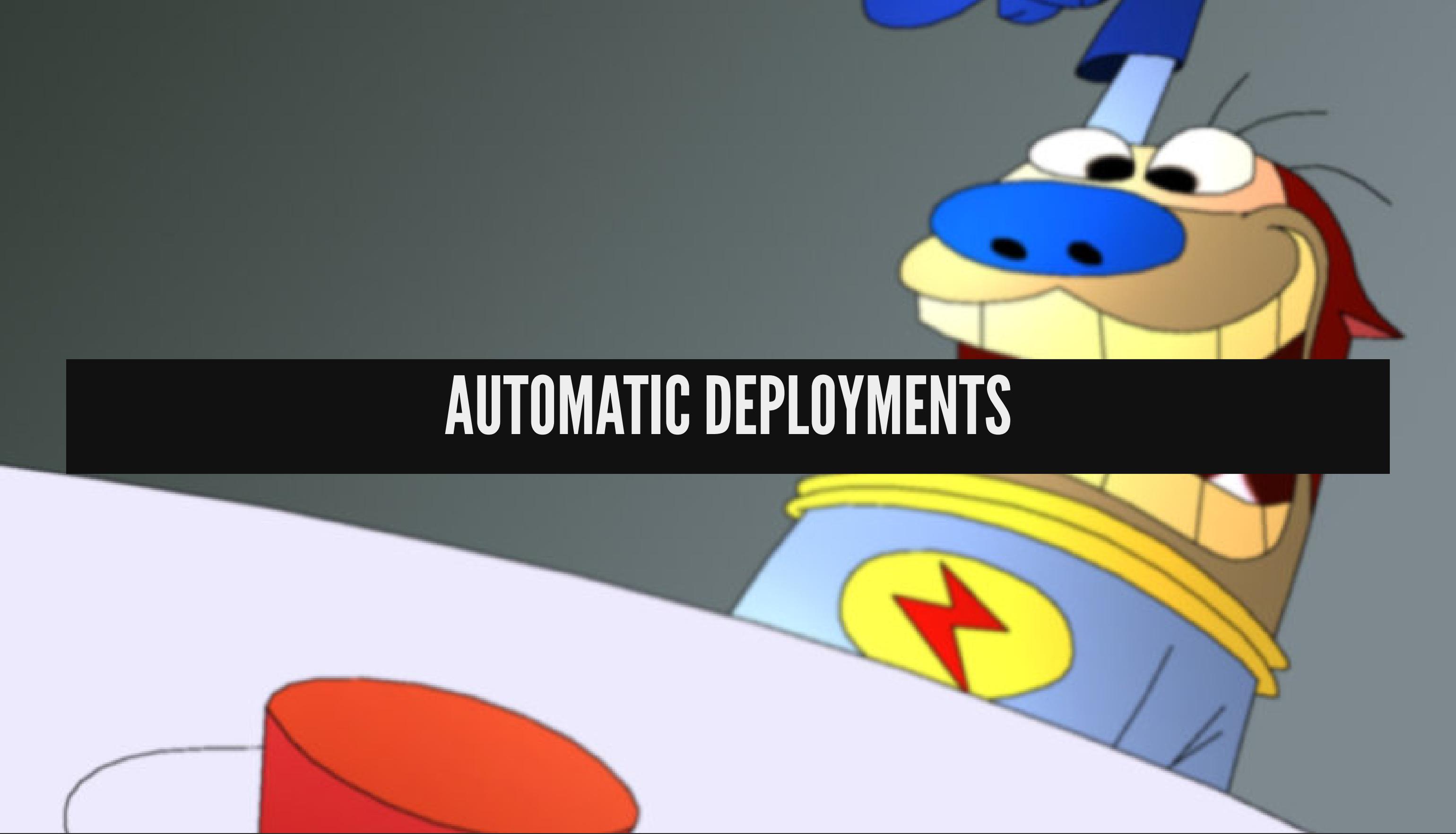
Custom scripts run when specific events happen. Use `.git/hooks/pre-commit` to ensure there are no syntax errors.

```
parse_error_count=0
for path in ${*:$arg_lookup_start}
do
    php -l "$path" 1> /dev/null
    if [ $? -ne 0 ]; then
        parse_error_count=$((parse_error_count + 1))
        php_errors_found=true
        if [ "$check_all" = false ]; then
            echo "Stopping at the first file with PHP Parse errors"
            exit 1
        fi
    fi
done;
```

From <https://github.com/hootsuite/pre-commit-php>

## Speaker notes

php-git-hooks can install hooks useful for PHP projects that not only lint your code, but also check the code style and run many of the quality tools we'll see later. Unlike SVN, git hooks have to be installed separately on each repository, so it takes a little more setup and effort.

A cartoon character with a large blue nose, wide eyes, and a yellow body with a blue lightning bolt on his chest. He is wearing a red cape and a blue hat. The character is looking down at a red cylindrical object on a white surface. The background is a solid grey color.

# AUTOMATIC DEPLOYMENTS

# RSYNC

A “straightforward” first step.

```
rsync -rzihc --delete -e "ssh -i $KEYFILE" --exclude=".idea"  
  --exclude=".git" \ --exclude="sites/default" \  
  --exclude="sites/all/modules/devel" \  
  ./web/ "$USER"@"$HOST":$DEST_DIR  
ssh "$USER"@"$HOST" -i $KEYFILE "cd $DEST_DIR ; drush cc all"
```

## Speaker notes

If you can use SSH to log in to a server, you can use rsync to transfer files. It's very efficient since it uploads only files which have changed.

It can be tedious to get the switches correct, you'll want to inventory what directories to exclude (upload folders for example). It can be very quick, but still not atomic. There's also no provision for running post deployment script to clear caches, or run database migrations. You'll end up creating a shell script for this.

# GIT PULL

Let's use Git to synchronize changes.

```
git pull gitlab
```

## Speaker notes

You can use git hooks to run tasks after a successful pull. To make this work consistently you should never be editing files on the target, otherwise you may have to manually merge the changes and at some point push them back to the remote. Downsides include depending on your git repo's availability in order to deploy. So if you use GitHub and it's having connectivity issues, you have to wait for them to resolve it.

Also, make sure you're not exposing your `.git` directory in your webroot, since this is a security vulnerability.

# DEPLOYER - RECIPES

Uses a recipe to define tasks to run on a remote host. <https://deployer.org>

```
set('repository', 'git@domain.com:username/repository.git');  
set('shared_files', [...]);  
set('shared_dirs', ['var/log', 'var/sessions']);  
set('shared_files', ['.env']);  
set('writable_dirs', ['var']);
```

## Speaker notes

Install it with Composer. You define your hosts, stages like testing and production, and tasks to run to deploy your code, like database migrations.

# DEPLOYER - TASKS

Define tasks that are part of deploying code

```
desc('Migrate database');
task('database:migrate', function () {
    run('{{bin/console}} doctrine:migrations:migrate --allow-no-migration');
});
desc('Clear cache');
task('deploy:cache:clear', function () {
    run('{{bin/console}} cache:clear --no-warmup');
});
```

# DEPLOYER - DEPLOY

```
dep deploy
```

```
desc('Deploy project');  
task('deploy', [  
    'deploy:info',  
    'deploy:prepare',  
    'deploy:lock',  
    'deploy:release',  
    'deploy:update_code',  
    'deploy:shared',  
    'deploy:vendors',  
    'deploy:writable',  
    'deploy:cache:clear',  
    'deploy:cache:warmup',  
    'deploy:symlink',  
    'deploy:unlock',  
    'cleanup',  
])
```

## Speaker notes

Deployer will checkout your code to your servers, run composer install, and more. Shared folders are not changed between releases, new code goes in a releases directory. Once all the files are ready a symlink is switched to point at the current release. In the end, this switch is very fast and means one request will always be served by a specific release. To make it atomic, install [https://github.com/etsy/mod\\_realdoc](https://github.com/etsy/mod_realdoc)

# ROCKETEER

Another PHP based task runner and deploy tool.

<http://rocketeer.autopergamene.eu>

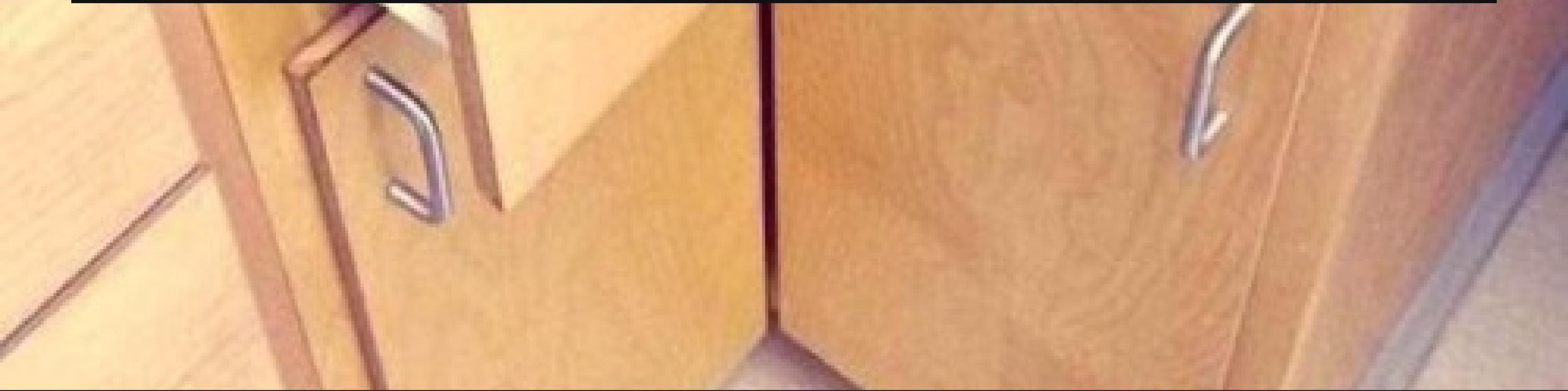
```
rocketeer deploy
```

## Speaker notes

Based on capistrano but integrates better with PHP projects. Like deploy, it'll create three folders in to deploy your application : releases and shared. A current symlink will point to the latest deployed releases. When you deploy, it can checkout your code from git, run composer install, and more.



**TESTING**



# BEHAT

Used for Behavior-Driven-Development (BDD).

## Speaker notes

Write features in plain english, which Behat can execute for acceptance. Under the hood, it uses Mink to fetch pages and test for various elements.

# BEHAT - GETTING STARTED

1. Tackle one or two trivial tasks
2. Identify critical tasks and write feature tests.

## Speaker notes

A good first task is ensuring your footer displays and the copyright year on your page is the current year. Don't tackle a complex task to start with—a simple task will ensure you configure your testing environment correctly (and gives you an easy win).

Then, start writing tests for critical tasks. A critical task is anything which would make your boss (or your boss's boss) notice. For [phparch.com](http://phparch.com), one critical flow is the shopping cart and checkout workflow. There are others for applying sales tax correctly, talking to our backend API, and more.

As your test suite grows, group critical tasks by similarity, this can come in handy when you only need to run tests for your API endpoints, for example.

# BEHAT - USER REGISTRATION

Feature: New User Registration

Scenario: A new user can create an account

Given I am on the register page

Then I should see the register form

Then I enter information for a new user in the form

Then I should be on "/account"

Then I should see my Recent Digital Purchases

## Speaker notes

This is a Behat feature. The goal is that any stakeholder could write a feature and define how something is supposed to work in plain English.

# BEHAT - USER REGISTRATION

```
/**
 * @Then I enter information for a new user in the form
 */
public function iEnterInformationForANewUserInTheForm() {
    // use the factory to create a Faker\Generator instance
    $faker = Faker\Factory::create();
    $this->page()->fillField('email', $faker->email);
    $password = $faker->password(8);
    $this->page()->fillField('password-1', $password);
    $this->page()->fillField('password-2', $password);
    $this->page()->fillField('first-name', $faker->firstName());
    $this->page()->fillField('last-name', $faker->lastName);
    $this->page()->pressButton('register');
}
```

## Speaker notes

On the flip side, once a step in a feature is defined, you can create PHP code to define the steps required in the browser for it. Here we are using Faker to get example data, then fill in registration fields, and press the submit button. The Mink class has many helper functions for interacting with an HTML page.

# BEHAT - STRIPE CHECKOUT

Feature: Magazine Subscriptions

```
@javascript @stripe
```

```
Scenario: Expired cards are declined
```

```
Then I click the single issue purchase button
```

```
Then I should be on the basket page
```

```
Then I should see the following in my basket:
```

```
| item          | price | qty |  
| php[architect] | $6.00 | 1   |
```

```
Then I am on the payment page
```

```
Then I click on the stripe payment button
```

```
Then I enter an expired card in stripe
```

```
Then I should see "Your card was declined."
```

## Speaker notes

You're not limited to just interacting with basic HTML. Use the `@javascript` annotation to tell Behat to use Selenium or PhantomJS instead of the default client.

# BEHAT - STRIPE CHECKOUT

```
// #CSS id of stripe popup
$this->getSession()->switchToIFrame('stripe_checkout_app');
$iframe = $this->getSession()->getPage();
// fill out user address fields & submit
// ...

// this number goes through stripe but then will fail
$iframe->fillField('Card number', '40000000000000341');
$iframe->fillField('Expiry', '12/22');
$iframe->fillField('CVC', '123');
$iframe->find('css', 'button[type=submit]')->click();
```

## Speaker notes

Defining a step which depends on Javascript is not much different than what we saw earlier. Here we look for Stripe's iframe to enter a known-bad credit card number and then submitting it to see the response returned. If you can investigate a page's DOM via Chrome or Firefox's Inspector, you can test it's behavior.

Chrome is being controlled by automated test software.



Magazine

Books

Training

Conferences

Swag

Register

Login



Login

# IMPLEMENTATION

```
..... 70
..... 140
..... 210
...F-.....FF-.F..... 280
.....F-

--- Failed steps:

67 scenarios (62 passed, 5 failed)
292 steps (284 passed, 5 failed, 3 skipped)
6m50.33s (13.80Mb)
```

Login

## Speaker notes

You shouldn't budget for testing as a separate task. It has to become a part of how you work. New bug reported? Write a test case for it before you fix it. Don't let managers cut your testing time out of your estimates. This seems expensive at first, but expanding your test suite pays off quickly in preventing regressions and ensuring your site's critical features always work.

# UNIT TESTING

- PHP Unit, <https://phpunit.de>
- Codeception, <https://codeception.com>

## Speaker notes

Codeception users PHP unit and provides a BDD layer for feature/integration testing.

Unit testing is meant to test a “unit” of code, typically a function or method. We want to verify that given a specific set of inputs, our code produces the expected output. To be unit-testable, your code should avoid using static calls and make use of dependency injection. This makes it easier to replace classes which talk to services like a database or an api with a mock class which behaves consistently and executes quickly.

# UNIT TESTING EXAMPLE

```
public function testLoginWrongPassword() {
    $this->expectException(\Vesta\Exception\ReportableException::class);
    $this->expectExceptionMessage('Login failed. Please try again.');
```

```
    $dbMock = $this->getDBMock(); // setup pdo and statement

    $testUser = $this->createMock(\User::class); // User mock
    $userFactory = $this->createMock(\Vesta\User\UserFactory::class);
    $userFactory->method('load')->willReturn($testUser);

    $auth = new \Vesta\User\Authenticator($dbMock, $userFactory);
    $success = $auth->login('oscar@musketears.me', 'notreallymypassword');
```

```
}
```

## Speaker notes

Here, we mock two dependencies: a PDO connection which simulates fetching user information and a factory class which creates a User object (from some data source). Mocks are used to mimic another classes behaviors. If our unit test breaks, we can be confident it is not the dependency that failed, it is our own logic.

# WHICH TESTS?

1. Write Behat/Feature tests first
2. Then write Unit tests as you refactor and fix classes and methods

## Speaker notes

Feature tests run slow, however you should start with them because they capture how your application behaves (or is supposed to behave).

Once you can rely on Behat to catch regressions and errors, you can start refactoring your underlying classes to decouple them and improve your application architecture. Write unit tests (which are quicker) to test the refactored behavior.

# QA TOOLS

# STATIC ANALYZERS

More than just PHP linting. These tools analyze your code without executing it.

## Speaker notes

Static analysis tools will automatically scan your codebase and look for things like undefined classes being used, correct number and type of parameters passed to functions or methods, correct value returned, phpdoc blocks match function signatures & returns, evaluate cyclomatic complexity, and more. Some can also look for parts of your code which might break if you upgrade from PHP 5 to 7.

# EXAKAT

<https://www.exakat.io>

Speaker notes

Exakat can also identify unused resources, supplement security reviews with automated checks.

# PHPSTAN

<https://github.com/phpstan/phpstan>

## Speaker notes

phpstan can ratchet up it's strictness via defined levels, so you can start fixing the easiest stuff first.

# PHAN

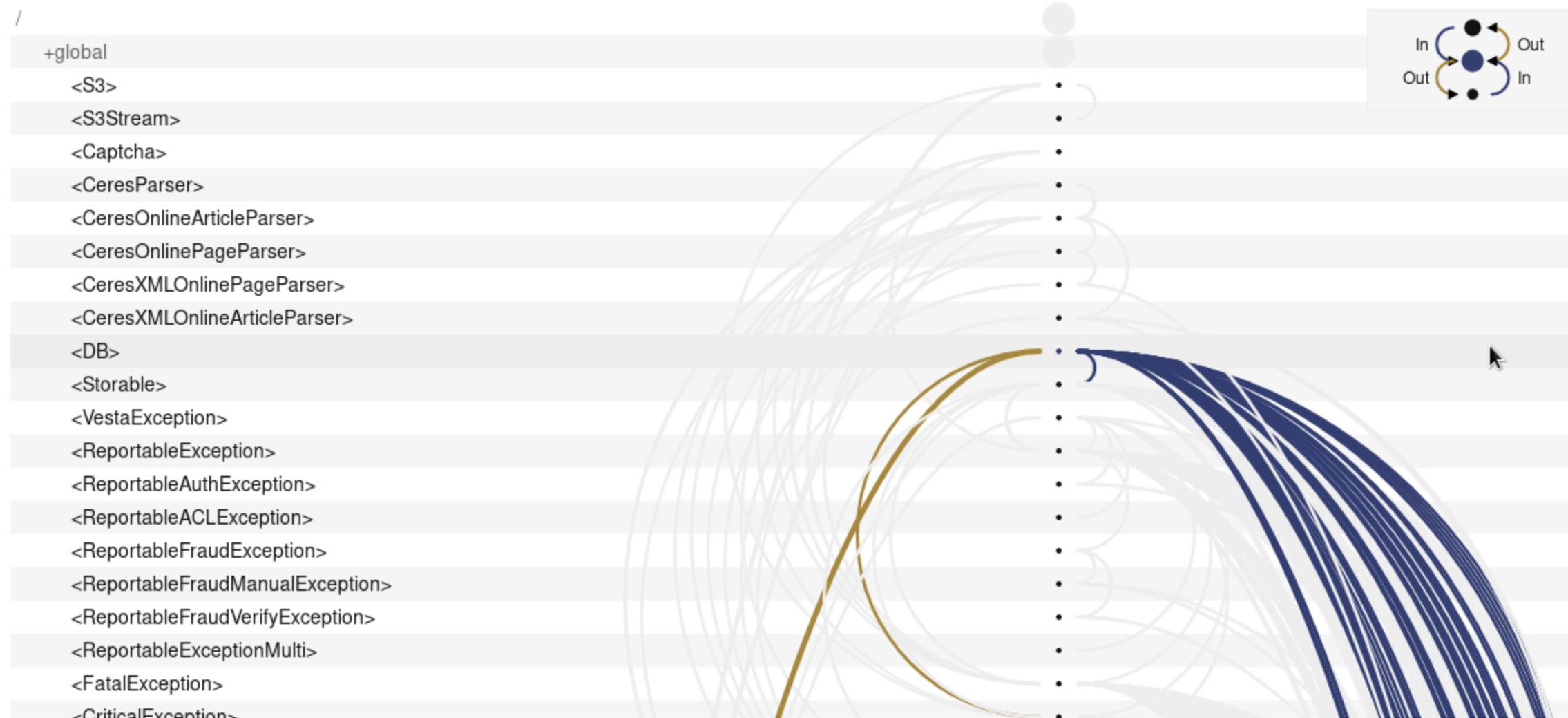
<https://github.com/phan/phan>

## Speaker notes

phan created by Rasmus Lerdorf and Andrew Morrison focus on finding incorrect code instead of proving the correctness of code. Can check for unreachable statements, validate PCRE regexes, check coding style conventions, and output results in different formats to help integration with other build/analysis services.

# DEHPEND

Identify dependencies in your code <https://dephpend.com>



## Speaker notes

depHPend analyses your apps dependencies. See which classes depend on others to focus refactoring efforts.

# PARSE

Scan for security vulnerabilities

<https://github.com/psecio/parse>

## Speaker notes

Checks for enabling display errors, use of `eval`, usages of `$_REQUEST` and `$GLOBALS` and more.

# PHP CODESNIFFER

Enforce a coding standard.

[https://github.com/squizlabs/PHP\\_CodeSniffer](https://github.com/squizlabs/PHP_CodeSniffer)

```
— /inferno/session/session.php 70 2
```

- **PSR1.Files.SideEffects.FoundWithSymbols**  
A file should declare new symbols (classes, functions, constants, etc.) and cause no other side effects, or it should execute logic with side effects, but should not do both. The first symbol is defined on line 2 and the first side effect is on line 269.
- **PSR1.Classes.ClassDeclaration.MissingNamespace**  
Each class must be in a namespace of at least one level (a top-level vendor name)
- **PSR2.Classes.ClassDeclaration.OpenBraceNewLine**  
Opening brace of a class must be on the line after the definition
- **Squiz.Scope.MethodScope.Missing**  
Visibility must be declared on method "register"

Speaker notes

Don't waste time arguing over a coding style. Just agree on one and stick to it. You can then use PHP CodeSniffer to audit it.

# PHP COPY/PASTE DETECTOR

Detect reused code.

<https://github.com/sebastianbergmann/phpcpd>

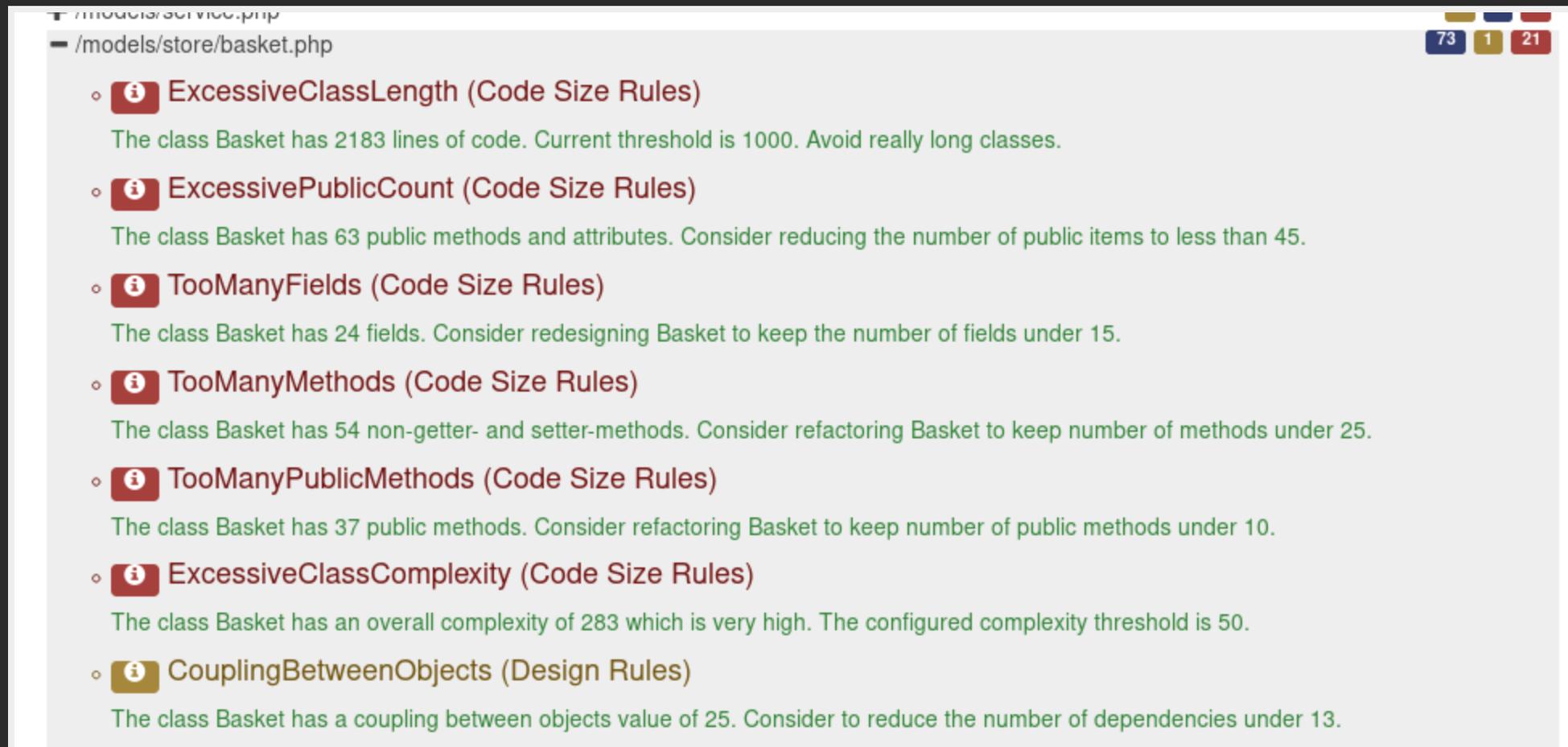
## Code Duplications

+ /inferno/ceres/geshi/geshi/cpp.php ↔ /inferno/ceres/geshi/geshi/c_mac.php	76 tokens	13 lines
+ /inferno/ceres/geshi/geshi/cpp.php ↔ /inferno/ceres/geshi/geshi/c_mac.php	148 tokens	22 lines
+ /inferno/ceres/geshi/geshi/cpp.php ↔ /inferno/ceres/geshi/geshi/c_mac.php	115 tokens	47 lines
+ /inferno/ceres/geshi/geshi/php.php ↔ /inferno/ceres/geshi/geshi/php-brief.php	141 tokens	62 lines
+ /inferno/ceres/geshi/geshi/applescript.php ↔ /inferno/ceres/geshi/geshi/perl.php	80 tokens	38 lines
+ /inferno/ceres/geshi/geshi/scheme.php ↔ /inferno/ceres/geshi/geshi/lisp.php	78 tokens	29 lines
+ /inferno/ceres/geshi/geshi/java5.php ↔ /inferno/ceres/geshi/geshi/actionscript-french.php	156 tokens	52 lines
+ /inferno/ceres/geshi/geshi/ocaml.php ↔ /inferno/ceres/geshi/geshi/ocaml-brief.php	95 tokens	16 lines
+ /inferno/ceres/geshi/geshi/java.php ↔ /inferno/ceres/geshi/geshi/d.php	92 tokens	43 lines
+ /inferno/ceres/geshi/geshi/mpasm.php ↔ /inferno/ceres/geshi/geshi/asm.php	71 tokens	36 lines
+ /inferno/ceres/geshi/geshi/caddcl.php ↔ /inferno/ceres/geshi/geshi/cadlisp.php	87 tokens	45 lines
+ /inferno/ceres/geshi/geshi/html4strict.php ↔ /inferno/ceres/geshi/geshi/cfm.php	438 tokens	110 lines
+ /inferno/ceres/geshi/geshi/c_mac.php ↔ /inferno/ceres/geshi/geshi/objc.php	183 tokens	27 lines

# PHP MESS DETECTOR

Find overly complicated code.

<https://phpmd.org>



The screenshot shows the output of PHPMD for a file named `/models/store/basket.php`. It lists several violations for the `Basket` class:

- ExcessiveClassLength (Code Size Rules)**: The class `Basket` has 2183 lines of code. Current threshold is 1000. Avoid really long classes.
- ExcessivePublicCount (Code Size Rules)**: The class `Basket` has 63 public methods and attributes. Consider reducing the number of public items to less than 45.
- TooManyFields (Code Size Rules)**: The class `Basket` has 24 fields. Consider redesigning `Basket` to keep the number of fields under 15.
- TooManyMethods (Code Size Rules)**: The class `Basket` has 54 non-getter- and setter-methods. Consider refactoring `Basket` to keep number of methods under 25.
- TooManyPublicMethods (Code Size Rules)**: The class `Basket` has 37 public methods. Consider refactoring `Basket` to keep number of public methods under 10.
- ExcessiveClassComplexity (Code Size Rules)**: The class `Basket` has an overall complexity of 283 which is very high. The configured complexity threshold is 50.
- CouplingBetweenObjects (Design Rules)**: The class `Basket` has a coupling between objects value of 25. Consider to reduce the number of dependencies under 13.

In the top right corner of the screenshot, there are three colored boxes with numbers: a blue box with '73', a yellow box with '1', and a red box with '21'.

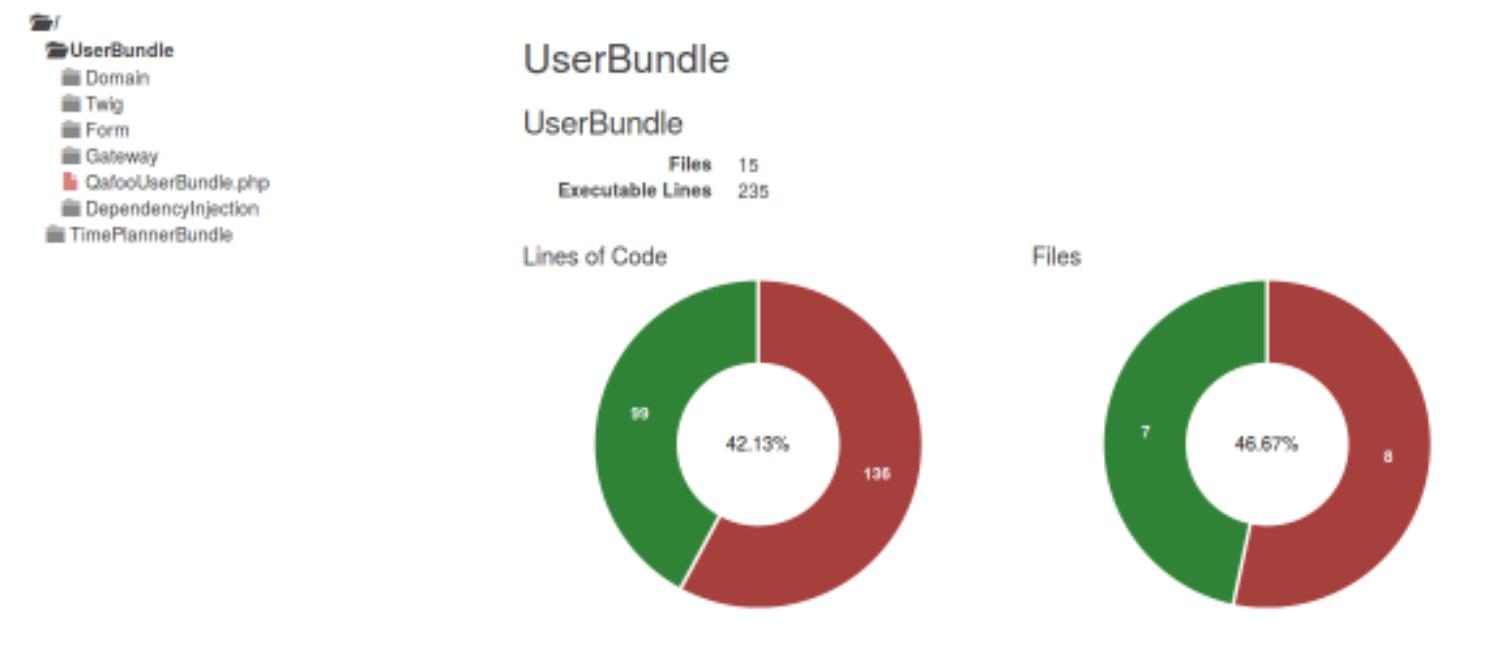
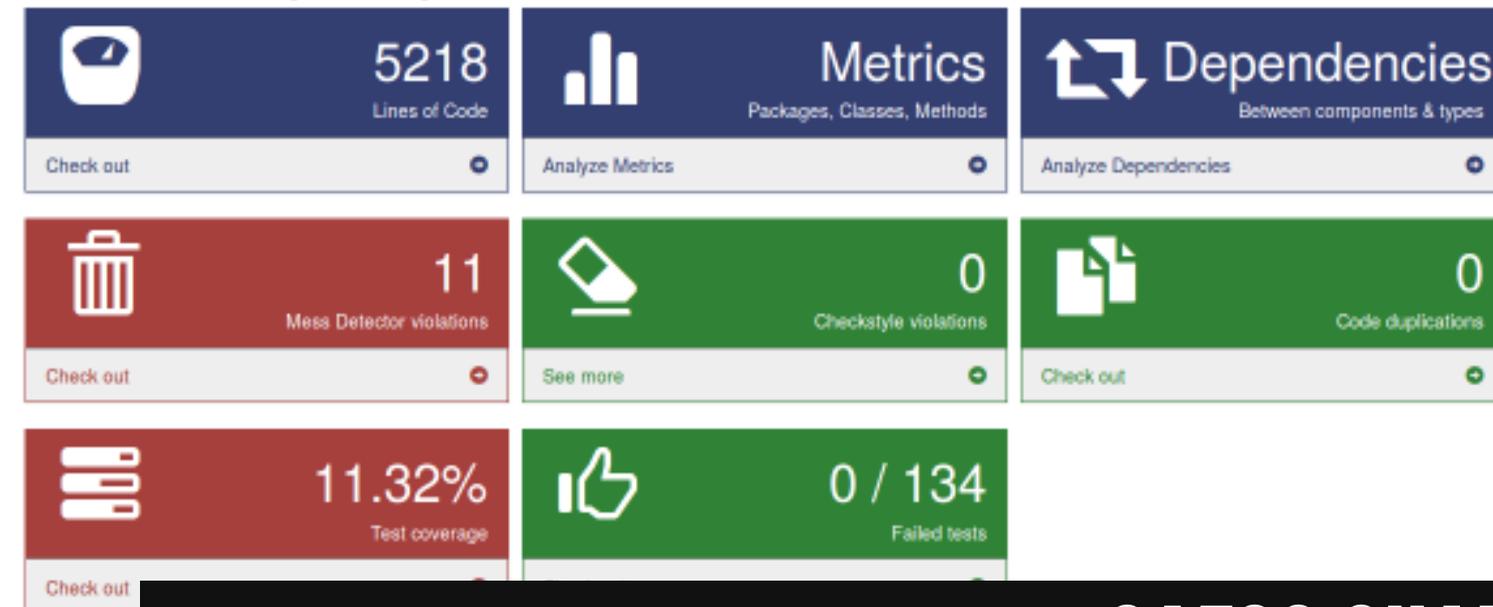
Speaker notes

PHPMD will look for possible bugs, overcomplicated expressions, and more. It offers rules grouped into things like “Clean Code”, “Design Rules”, and more.

**SO MANY MORE**

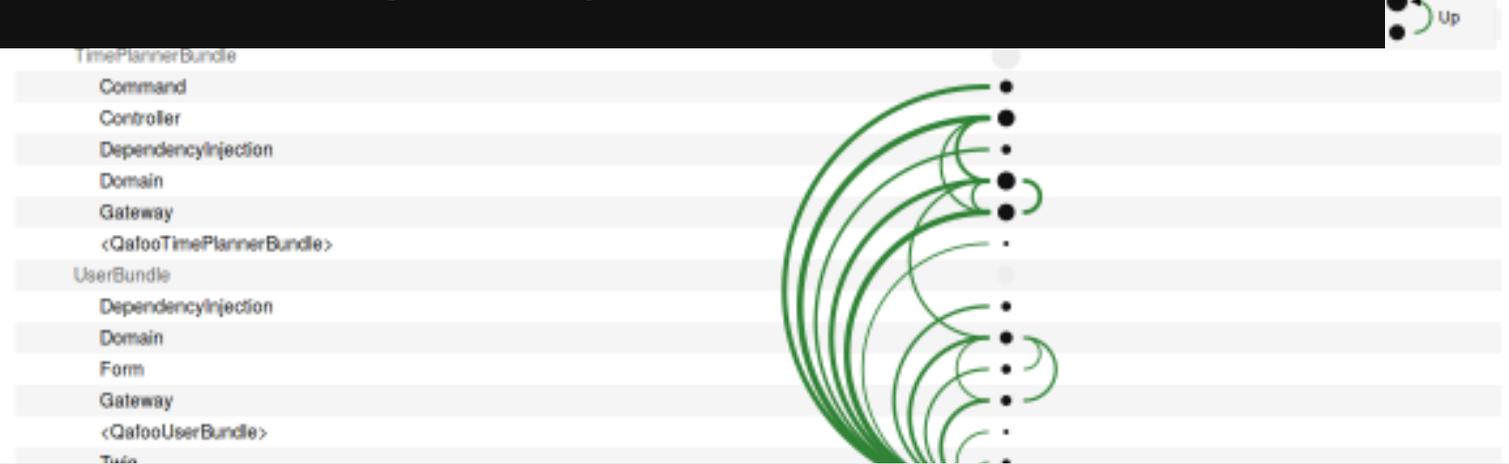
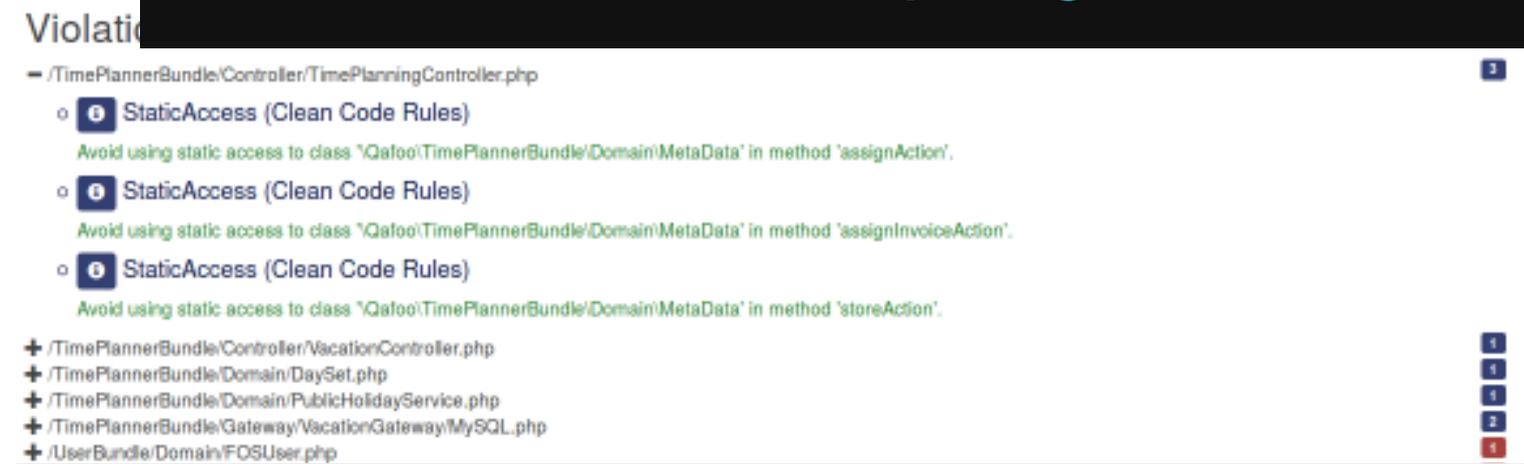
<https://phpqa.io/index.htm>

# Qafoo Quality Analyzer



# QAFOO QUALITY ANALYZER

<https://github.com/Qafoo/QualityAnalyzer>



## Speaker notes

Consolidates reporting on various metrics using many of the tools mentioned earlier. It's easy to setup and have it analyze your codebase and then provide a dashboard consolidating many of the tools mentioned earlier.

**AUTOMATION**

# BAKING DEV ENVIRONMENTS

- Docker + Docker-compose
- Vagrant + Ansible

## Speaker notes

Setting up a development environment should be a pain point for only one person on your team. Use the same tool across your team to ensure everyone's machine works the same so there are no surprised. That is, "It worked on my machine."

# COMPOSER SCRIPTS

```
"scripts": {  
    "check": [  
        "@cs",  
        "@test"  
    ],  
    "cs-check": "phpcs",  
    "cs-fix": "phpcbf",  
    "test": "vendor/bin/codecept run",  
    "test-coverage": "phpunit --colors=always --coverage-clover clover.xml",  
    "upload-coverage": "coveralls -v"  
},
```

<https://www.masterzendframework.com/series/tooling/composer/automation-scripts/>

## Speaker notes

Besides installing and managing project dependencies, Composer can run scripts for you. This is handy for saving and sharing often used commands to run tests or static analysis tools.

# SCAFFOLDING TOOLS

- Symfony, <https://symfony.com/doc/current/console.html>
- Laravel, <https://laravel.com/docs/5.6/artisan>
- Drupal, <https://drupalconsole.com>
- WordPress, <https://wp-cli.org>

## Speaker notes

If you work with a particular framework or CMS, learn to use the command line tools available for it. You can save yourself time (and prevent errors) when you have to define routes, clear caches, export data, and more. See Steve Grunwell's PHP CLI talk for more.

# SYMFONY CONSOLE

<https://symfony.com/doc/current/components/console.html>

```
book
  book:checkmd           Checks markdown for common errors
  book:gitlab:tickets    Crete issues for gitlab book repo
draft
  draft:upload          Uploads an article to Draftin.
fin
  fin:paypal:sheet      Transform Paypal related CSV files to an Excel sheet for reconciling.
  fin:stripe:sheet      Transform Stripe related CSV files to an Excel sheet for reconciling.
mag
  mag:acceptance:emails Draft Acceptance Emails.
  mag:article-outreach  Prepare content for article emails.
  mag:article:csv        Export article data as CSV.
  mag:code:archive       Generates code archive for an issue.
  mag:convert:docx       Converts a Word document to markdown iwth pandoc.
  mag:count:words        Calculate word count in a file.
  mag:cover-titles       Generates images for social media use.
  mag:fetch:joindin      Fetches information about an event from JoindIn.
  mag:fetch:news         Fetch markdown summary of news.
  mag:init:issue         Generates magazine directory structure and XML file.
  mag:make:pdf           Stitch an issue PDF.
  mag:make:summary       Generates an article summary from magazine XML file.
  mag:make:summaryhtml   Generates an HTML article summary from markdown.
  mag:outreach:tickets   Makes one or more Unfuddle tickets for magazine outreach from a CSV.
  mag:scrub:article      Converts a Word document to markdown iwth pandoc.
  mag:update:template    Updates the content for a mailchimp campaign.
marketing
  marketing:issue:tweets Generate article tweets for an issue.
```

## Speaker notes

You can use the Symfony Console component to put PHP to use at the command line. I use this to automate a lot of frequent tasks. Almost everything has an API nowadays, so there's no technical limit here. If you do a task regularly (monthly), invest some time in automating as much as you can.

# THANK YOU

Feedback: <https://joind.in/talk/6e37f>

[@omerida](#)

I publish php[architect], a monthly magazine for PHP developers.

[www.phparch.com](http://www.phparch.com)

php[world] is our fall conference in Washington D.C.

[world.phparch.com](http://world.phparch.com)