Specing out teamwork

Symfony Berlin User Group

Tonight

1. Contribution Guidelines
2. Conventional Commits & Automation
3. Testing Strategy Document

Stiven Llupa
Senior Development Lead

Symfony *aficionado*

- Working with Symfony since 2013, Sonata on/off since 2015 and API Platform since 2019.
- Overseeing DLC since 2018

1

Contribution Guidelines

“SCRUM is great for managing complex projects with evolving requirements by promoting collaboration, flexibility, and iterative progress through short development cycles.”

cit. ChatGPT

Contribution Guideline

Project management methodologies like SCRUM and Kanban dictate workflows at an organizational level, development teams can **also define** their own best practices to streamline contributions, enhance collaboration, and improve delivery timelines.

- Define the contribution process
  - branching
  - commit messages
  - pull/merge request flow

- Standards
  - paradigms and design patterns
  - coding standards
  - testing standards
  - when applicable: ORM standards

Contribution Guideline

Contributing to [Project-Name]

This document describes the process of contributing to [Project-Name] source code base as well as the standards that will be applied when evaluating contributions. Please be aware that additional agreements will be necessary before we can accept changes from contributors.

Contribution process

References to roles are made throughout this document. These are not intended to reflect titles or long-term job assignments; rather, these are used as descriptors to refer to members of the development team performing tasks in the check-in process. These roles are:

- *Author*: The individual who has made changes to files in the software repository, and wishes to check these in.
- *Reviewer*: The individual who reviews changes to files before they are checked in.
- *Integrator*: The individual who performs the task of merging these files.

Branching

We follow the format of branch names:

- `project-0000/name-of-change`: Common branch naming for all new branches
- `hotfix/name-of-fix`: Branch naming for prioritized changes applied directly to `main`
- `chore/name-of-chore`: Branch naming for a non-customer related improvement

As a general rule of thumb keep the `name-of-change` concise and not overly descriptive. We do not particularly work with the branch name; our commit messages are the value holders when it comes to ticketing.

Contribution Guideline

Pull requests

When development is complete for a ticket, the first step toward merging it is to open a Pull Request. The contributions SHOULD meet code, test, and commit message standards as described below.

- Pull requests MAY be assigned to an *Integrator*.
- Pull requests MUST be assigned to a *Reviewer*.
- Code review SHOULD take place using discussion features within the pull request.

Opened merge pull SHOULD be marked by following ways:

- If the pull request is not ready for review yet, it MUST be labeled with `Work in progress`
  - Additionally, you can use GitHub interface to mark the pull request as Draft.
- If the pull request is ready to be reviewed, it MUST be labeled with `Ready for code review`
- If you are a *Reviewer* and you finish reviewing, including no-feedback, you MUST **Approve** the pull request
  - If the pull request is approved by at least 1 *Reviewer* you SHOULD mark it with the label `Reviewed`
  - If the pull request has outstanding feedback you MUST mark it with the label `Waiting for update`
  - If the pull request has `merge conflicts` you MUST mark it with the label `Waiting for update`

Pull request author MUST take care of the following things:

- Author SHOULD be responsible for their pull request, making sure their pull request will not be forgotten.
- Author SHOULD react on discussion points related to the pull request, updating the code or answering the comments from reviewers.
- Author MUST mark feedback points (comments) as resolved after requested changes were applied (if feedback required changes).

A pull request MUST have at least **1 approval** and marked as `Reviewed` to be merged. *Integrator* MAY so choose to expedite *Pull requests* rules under their own discretion (Hotfix, Deployment, etc).

Contribution Guideline

Standards

Code solution standards

- SHOULD use SOLID principles whenever possible to apply.
- SHOULD use design patterns whenever possible to apply.
  - Refactoring Guru
  - Design Patterns PHP
- SHOULD Avoid close coupling between classes (Law of Demeter)

Code style standards

- MUST use Symfony Coding Standards
- MUST use short array notation
- SHOULD use all possible type hinting available in used PHP version
- Business logic MUST NOT go in
  - Controllers
  - Commands
  - Doctrine Events
- SHOULD avoid double typed primitive variables. No `null` reference on `string`, `bool`, `array`, `int` etc
- MUST NOT execute native `SQL` queries in an application context.
  - They bypass all OOP related events and result in corrupted data.
  - They break OWASP Top 10 for PHP.
- MUST NOT use container injection.
- SHOULD NOT leave todo comments when pull request is submitted for code review.
  - `@todo` comments MAY only exist during `Work in progress` state and SHOULD be fixed by *Author* as part of the progress.
  - If todo section is unavoidable, *Author* MUST create a "technical debt" ticket providing all necessary information with instructions and link.

Contribution Guideline

Doctrine ORM standards

This project uses Doctrine ORM. We use Doctrine's definition of `Entity` to differentiate objects that contain persistable properties from other objects, namely `Model`.

Mapping

Entities' ORM metadata are mapped using PHP Attributes as listed in Doctrine's Attributes Reference.

Identifiers

When creating a new entity, the following rules for identifiers apply:

- All entities MUST use **numeric primary keys** as their internal Doctrine identifier by default.
- Entities MUST NOT expose their numeric primary key on a project API level.
- Entities that are also API resources SHOULD have a secondary **UUID7 unique index** by default.
- Entities that are also API resources MUST expose only their UUID7 unique index on a project API level.
- UUID7 fields MUST be stored as binary data. UUID7 fields MUST NOT be stored as VARCHAR.

Contribution Guideline

Doctrine ORM standards

Properties

It is RECOMMENDED to use rich entities as often as possible, but rich entities MUST NOT change framework property mutators prefixes, or extractors prefixes, used by Symfony and/or Doctrine such as: `get`, `set`, `is`, `has`, `add`, `remove`.

```php
class Foo
{
    public function getId(): ?int
    {
        //This function is auto-generated by Doctrine Bridge and MUST NOT be changed
    }

    public function addBar(Bar $bar): Foo
    {
        //This function is auto-generated by Doctrine Bridge and MUST NOT be changed
    }

    public function getBarWithBuzz(): Bar
    {
        //This function is not auto-generated by Doctrine Bridge and MAY be changed
    }

    public function resolveQux(int $rounds): Collection
    {
        //This function is not auto-generated by Doctrine Bridge and MAY be changed
    }
}
```

Contribution Guideline

Doctrine ORM standards

Schema migrations

The project uses Doctrine Migrations to facilitate and standardise the migration instruct set for DevOps. The following rules MUST be applied on development that requires schema changes:

1. Expected schema changes per commit type

- A `fix` MUST NOT introduce changes where columns or tables will be dropped.
- A `fix` SHOULD NOT introduce changes where columns or tables will be created.
- A `fix` SHOULD NOT introduce changes where foreign keys, index-es will be dropped.
- A `fix` MAY introduce changes where foreign keys, index-es will be created.
- A `feat` MUST NOT introduce changes where columns or tables will be dropped.
- A `feat` MAY introduce changes where columns or tables will be created.
- A `feat` SHOULD NOT introduce changes where foreign keys, index-es will be dropped.
- A `feat` MAY introduce changes where foreign keys, index-es will be created.
- A `chore` MAY introduce changes where columns or tables will be dropped.
- A `chore` MAY introduce changes where columns or tables will be created.
- A `chore` MAY introduce changes where foreign keys, index-es will be dropped.
- A `chore` MAY introduce changes where foreign keys, index-es will be created.
- A `ci`, `docs`, `test` commit MUST NOT introduce changes where columns or tables will be dropped.
- A `ci`, `docs`, `test` commit MUST NOT introduce changes where columns or tables will be created.
- A `ci`, `docs`, `test` commit MUST NOT introduce changes where foreign keys, index-es will be dropped.
- A `ci`, `docs`, `test` commit MUST NOT introduce changes where foreign keys, index-es will be created.

Contribution Guideline

Doctrine ORM standards

Schema migrations

2. Expected migration instructions per Pull Request

With every `Pull Request` that introduces schema changes the *Author* is responsible to generate the final Migration file. This file MUST use the autogenerated version number, and it has to be fully functional. Criteria of a fully functional migration:

- The `up()` and `down()` functions execute correctly.
- If there is new data to be introduced a simple single insert instruction is preferred. This should be limited to a handful of rows. Anything in the triple digits or above territory requires a data-migration plan.
- For more complex new data do not use `preUp()`, `postUp()`, `preDown()` and `postDown()`. A data-migration plan has to be implemented accordingly.

3. Expected migration file format

The `doctrine:migrations:diff` command generates a migration by comparing your current database to your mapping information. This file is auto-generated, please modify to your needs!

2

Conventional Commits

Conventional Commits

The Conventional Commits specification is a lightweight convention on top of commit messages.

It provides an easy set of rules for creating an explicit commit history; which makes it easier to write automated tools on top of. This convention dovetails with **SemVer**, by describing the features, fixes, and breaking changes made in commit messages.

- GitHub search becomes good, actually?!
- Your client likes to point fingers?
  - point them right back!

Conventional Commits

Commit message standards

This project uses Conventional Commits. A quick summary is as follows:
The Conventional Commits specification is a lightweight convention on top of commit messages. It provides an easy set of rules for creating an explicit commit history; which makes it easier to write automated tools on top of. This convention dovetails with SemVer, by describing the features, fixes, and breaking changes made in commit messages.

The commit message SHOULD be structured as follows:

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Conventional Commits

Commit message standards

`<description>` MUST always start with a verb, capitalised, in Simple Present tense, e.g, `Add ability to authenticate with serial number`. The commit contains the following structural elements, to communicate intent to the consumers of your library:

- **fix**: a commit of the type fix patches a bug in your codebase.
  this correlates with PATCH in Semantic Versioning
- **feat**: a commit of the type feat introduces a new feature to the codebase.
  this correlates with MINOR in Semantic Versioning
- **BREAKING CHANGE**: a commit that has a footer `BREAKING CHANGE:`, or appends a **!** after the type/scope, introduces a breaking API change. A `BREAKING CHANGE` can be part of commits of any type.
  this correlates with MAJOR in Semantic Versioning

Types other than **fix:** and **feat:** are allowed, for example **chore:**, **ci:**, **docs:**, and **test:**.
A scope MAY be provided to a commit's type, to provide additional contextual information and is contained within parenthesis, e.g., `feat(device): Add ability to authenticate with serial number`.
Footers other than `BREAKING CHANGE: <description>` MAY be provided and follow a convention similar to git trailer format. It is REQUIRED to add the ticket reference in footers, e.g., `Refs: PROJECT-1234`.

3

Testing Strategy Document

partially stolen from **I**nternational **S**oftware **T**esting **Q**ualification **B**oard

Testing Strategy

Document

A high-level description of the test levels to be performed and the testing within those levels for an organization or programme (one or more projects).

Testing Strategy

Test standards

Tests are REQUIRED

- Tests MUST follow the KISS principle.
- MUST add `ApiTestCase` or `WebTestCase` tests if you need to test client interaction.
- MUST add `KernelTestCase` tests if you need to test a service class.
- SHOULD add `TestCase` if you need to test a no-dependency class.
- PHPUnit test suite naming conventions SHOULD be compliant with `testdox` output format:
  - `testWithdrawingMoneyOnInvalidAccountWillThrowException`
  - `testAdultsHaveAgeAboveEighteen`
  - `testStudentIsNotAdmittedWhenMissingParentalInformation`
- Tests should use behavior-driven code coverage as a primary metric by focusing on evaluating expected behaviors and functionalities, prioritizing real-world use cases over traditional lines of code covered metrics
- Contributors should modify existing tests that are affected by changes of existing code to ensure they still accurately validate the functionality and behavior of the modified code.
- Pull requests containing test adaptations due to existing code changes must undergo a thorough code review to ensure compliance with the guidelines and maintain overall code quality.

quick recap

Define your contribution process

Rely on existing Standards

Copy and implement already proven methodologies. Try them out, keep what you like!

Change when necessary

Automate and relax

Automate it and never look back.
Be cool 😎

Links

cuz we need links

- NASA OpenMCT
- Conventional Commits
  - changelog tool
  - commit checker
- Testing Strategy

Vielen Dank