Test on refactored code

About me and my company

Mingwei Zhang(Ming)

- Working at foundU and trying to make things better.
- foundU provides workforce management and payroll solutions.
- Currently hiring for front/back-end devs, UX and tester.
- mingwei@foundu.com.au

Legacy code

- My first refactor experience from my first PHP project
  - Functions were included
  - No namespace and no classes(no “god class” but god php function files)
  - No composer
  - No tests
  - No documentation
  - What about yours? Tell me your horror stories.

Background about the project

- Using Laravel
- Small amount of PHPUnit coverage
- Roster and Payroll software that generates thouthands of payslips every week.

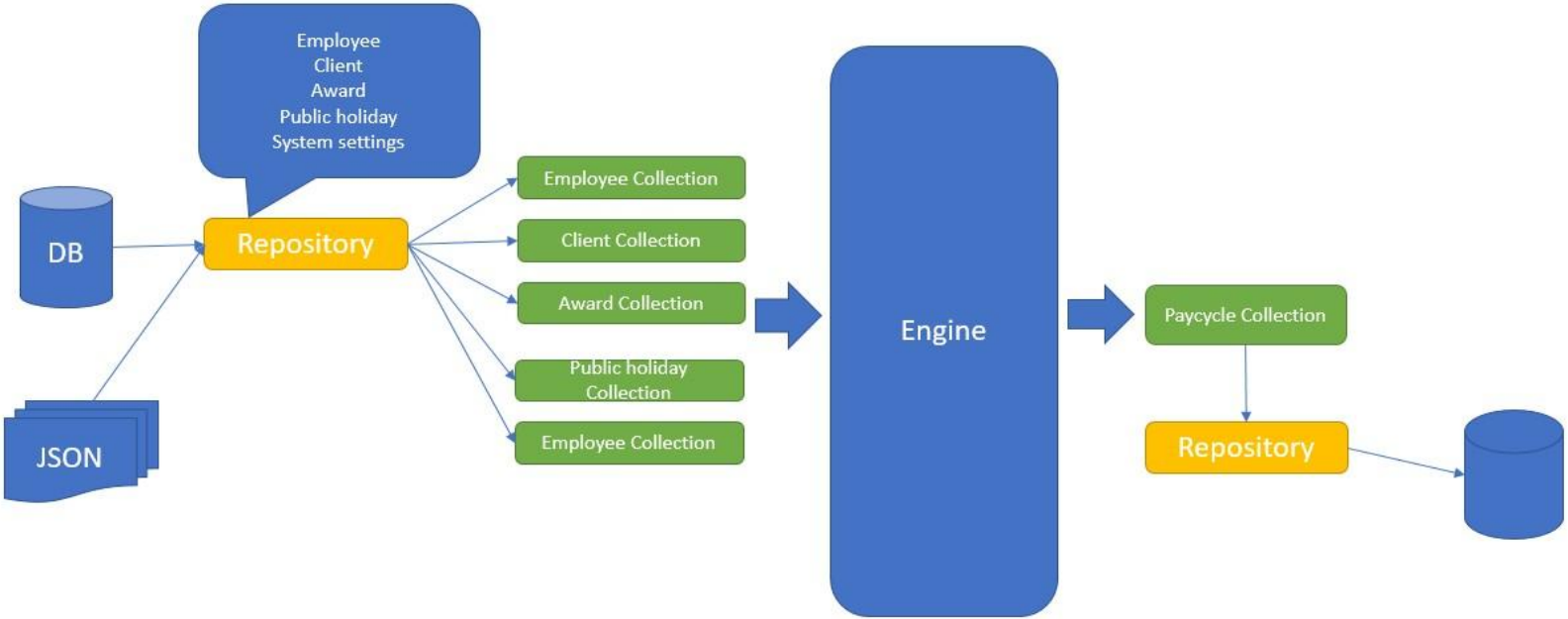
Why:

- Demands and changes makes it harder to maintain the software.
  - Bugs
  - Increasing amount of the tests
  - Slowness from the legacy code.
    - N + 1 queries
    - Repeated code
  - Hard to train other developers and taking lot of time to add new feature.
  - Impacted people’s payslips

Options

- New branch new code, build replacement code and try to run it against with production
  - Upsides
    - We can fully deprecate the old engine
  - Downsides
    - Hard to release it and know it is matching 100% with existing functionality
    - Hard to keep up with existing master branch
- Build code and run it with existing code
  - Upsides
    - Easy to merge and sync with main branch
    - Reuse some of the existing code and logics
  - Downsides
    - Have to deal with lot of code conflicts

Concept flow chart

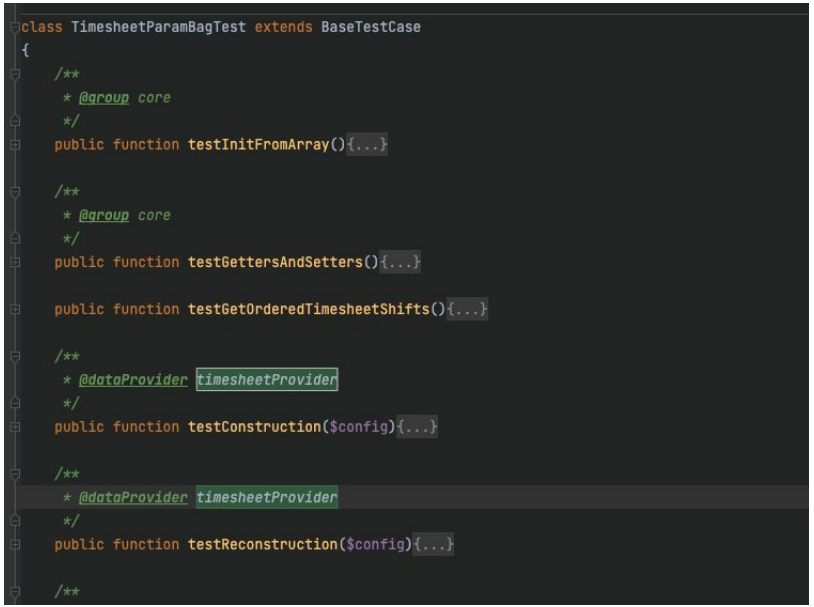


Plans

- New Architecture
  - Repository design pattern
    - All queries are in here. We can use replace it if we want to switch it to other DB types.
  - Services for pure business logics
    - We can inject the data into those services easily from tests.
  - ParamBag and Collections for object instances
    - No mutations on object. Have to go through setters.
- Tests
  - Unit tests (Bolts length tolerance)
  - Integration tests (Can turn the engine on)
  - Functional tests (Can turn the car on)

Unit tests

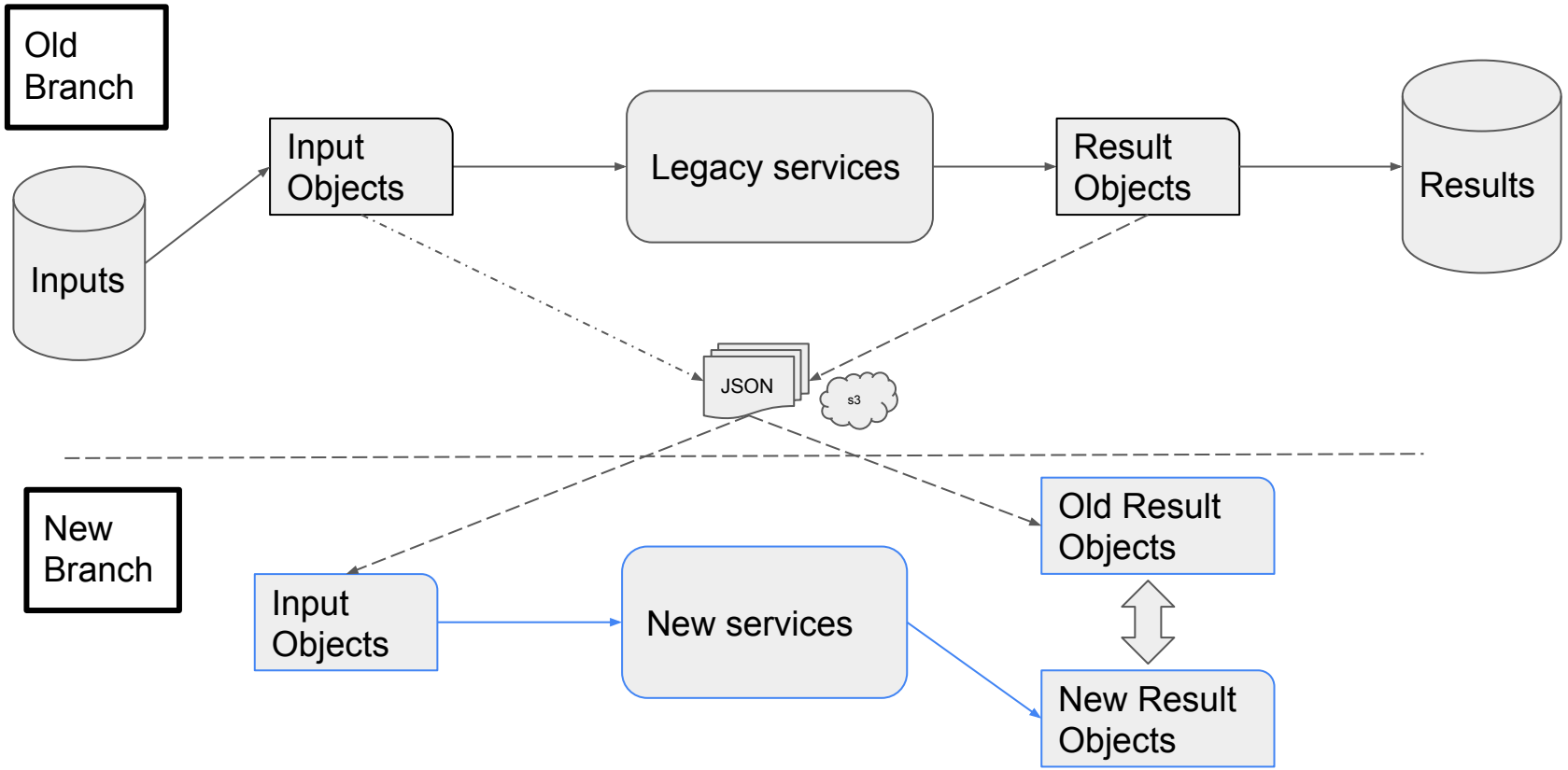
- Why
  - We use unit tests here to check whether each single function is working
- This step consists of
  - Testing constructor
  - Testing getter and setters
  - Testing computed methods
  - Testing custom methods



Integration tests

- Checks services
  - This is part of the business logics. For example:
    - Breakdown and convert the timesheets to interpreted shifts.
    - Apply the leaves
    - Add the allowances
- Checks the composite operations.
  - We introduced our scenario tests in here for our core service.
  - This step checks the important moving pieces and semi results such like interpreted timesheets, shifts and paying rates.
  - We don’t test the final outputs of the payslips
  - Approach 1
    - Use PHPUnit providers to write inputs and expected data.
  - Approach 2
    - Describe, write and store the parameters and expected results to JSON files.
    - Works similar to phpunit data providers, but it is easy to manage large amount of the scenarios.

Background before next test



Regression tests

1. On legacy branch, we do capture the inputs objects and output/result objects into JSONs on s3.
2. Switch to new project branch
3. Then initialize those data from JSONs on s3, and produce new results
    a. Input objects (timesheets and shifts)
    b. New output/result objects (payslips)
4. Then compare the output/results from “new services” with previous captured “result objects”.

Functional tests

- Existing captured scenarios will be baseline of product requirements.
  - Legacy logics are not always right. After assessment, we found some bugs from legacy logics.
- New features will need to be captured and tested for future regression tests
  - Pipeline will run all scenarios for any code changes in future.
  - Currently we have 17k+ payslip scenarios and it is still growing.

Release strategy

- From small customers to big customers.
- From minimum functionalities to complex features.
  - Easy stage
    - Interpret timesheets and shifts
    - Leaves
    - Wage
    - Tax
    - Superannuation
  - Medium level features
    - Allowance, meal allowance and shift allowances
    - Special award rules
  - Hard-core
    - Customized features

Lessons learned

- Don’t be distracted by unrelated things, unless that’s 200% necessary.
  - Hard to focus and consider edge cases if u constantly moving between different projects
- Code with “how can I test this” in mind
  - Try to build single purpose methods and classes

Q&A