Report for assignment 3

Anders Sjöbom, asjobom@kth.se
Albin Remnestål, albinre@kth.se
Johan Bergdorf, bergdorf@kth.se
Marcus Wengelin, wengel@kth.se
Robert Wörlund, worlund@kth.se

Project

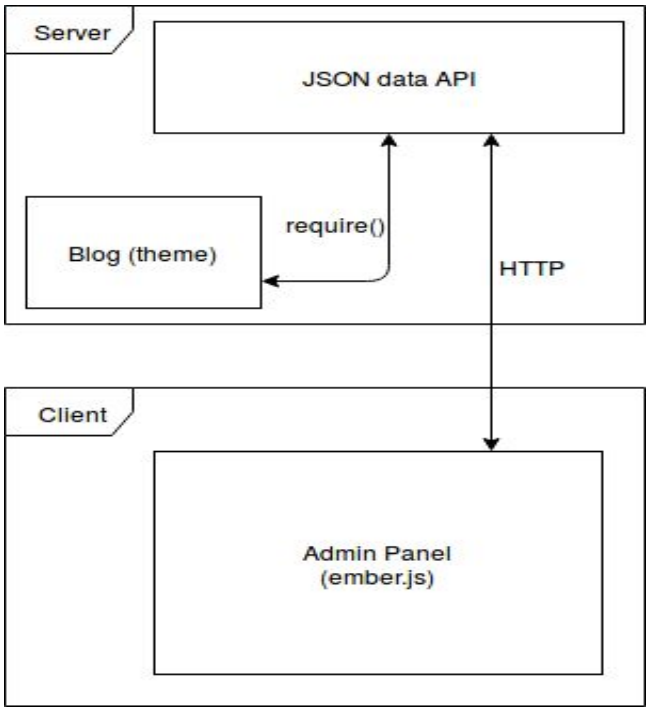
Name: Ghost

Refactor URL: https://github.com/remnestal/Ghost
Project URL: https://github.com/TryGhost/Ghost

Ghost is publishing software developed by a non-profit called the Ghost Foundation. It powers multiple blogs, magazines and journalists. Among their customers are Tinder, DuckDuckGo and Cloudflare. It includes features such as an editor, a content management system and more.

Architectural overview

The architectural overview of the program is summarized by the developers as follows;



(Source, https://docs.ghost.org/v1/docs/codebase-overview)

The code, as seen in the diagram above, can be placed into three categories. Firstly, the client, which consists of an administrative interface that can be used to interact with the server. The client is built in ember, which is a javascript framework based on the model-view-viewmodel pattern. Secondly, the server side application handles the rendering of the blog, and incorporates the selected theme. The third part, the JSON API, holds the primary functionality. It is meant as an abstraction layer to Ghosts internal structures, and handles permissions as well as formatting and removal of security-related data.

Other core server-side functionalities include *Data Models*, which provide access to databases such as MySQL, SQLite and postgreSQL. The data models add core properties to data being inserted into the database, such as authors (among others). The *Email* module contains all the email-related functionality within the Ghost platform. They can be sent using the JSON API. The *Config* is responsible for loading and validating the configuration file. The *Apps* module provide a modular way to add new functionality to Ghost. An App can request certain permissions at install, and can then use the JSON API to request and modify data, if it has the permissions to do so. The *Restful API* publicly exposes the JSON API according to the REST-principles.

The project also contains a substantial amount of tests: 2503 total tests divided in 4 categories/modules (319+1+1590+593). The tests live inside the /tests directory, and the tests are separated into three categories. There are *functional* tests, *integration* tests and *unit* tests. The test directory also contains a utils-directory for easier access to the api, among other things.

The purpose of the Ghost project is to provide a fully open-source, hackable publishing platform focused on the content. It is supposed to be easy to use, faster than other publishing platforms and equipped with many useful features, such as social media integration and a built-in markdown editor. Ghost is a non-profit organisation, and the authors do what they do because they believe in freedom of speech and journalism.

Dependencies

To get started with the project, there are some requirements. First of, Ghost is intended to be run on NodeJS, preferably version 6.9. Another requirement is Yarn, an alternate package manager for NodeJS. The packages required are:

- Knex-migrator (db migration tool)
- Grunt-cli (javascript task runner)
- Ember-cli (emberjs command line utility)
- Bower (package manager)

Selected issue

Title: Refactoring: API/Model layer - pass options object

URL: https://github.com/TryGhost/Ghost/issues/9127

The issue is about refactoring how the options object is passed around/handled in the API layer. The issue brings up a current problem of mixed design where an argument is passed by reference, and also returned in the same function. The problem exists because the system utilizes a pipeline tool which executes tasks sequentially, using Bluebird.reduce. The used function requires all tasks to return the options object as it passes it on as an argument to the next task.

The issue does not mention a concrete solution to the problem but rather presents where it exists and possible ideas on where to make changes. Some initial confusion regarding how to go about the issue resulted in us making contact with the developer of Ghost and creator of the issue. A solution was initially made utilizing callbacks instead of Bluebird.reduce, unfortunately this idea clashed with how the system uses Promises. Further discussion with Katarina, developer of Ghost, resulted in some clarifications regarding the refactor.

The refactor can be split up into three parts:

1. Store req.body in options.data before passing it to the API in apiHandler

   Previously the API handler passed two objects to the API on incoming requests. The options object as well as 'object' object which contained req.body. One step of the refactor was to store req.body in options.data rather than in its own object. This change resulted in a refactor of doValidate to use options.data instead of the previously distinct object for req.body that was passed to the function.

   It was quickly noted that options.data was also used for other things which would then result in it being overwritten in various functions. This was a problem Katarina was aware of but told us that is outside the scope of this issue.

2. Store the result of api_task.modelQuery in options.response

   The last task of the API performs a modelQuery and its result is returned directly. The refactor should instead make it so that the modelQuery result is stored in options.response. This makes sure we keep it consistent with relying on object modification by reference and thus the options object should not need to be returned after each api_task.

3. Replace pipeline so that is does not require the API tasks to return the options object

   Currently the pipeline chains all its given tasks using the function Bluebird.reduce, which iterates over the tasks and passes an initial specified argument to the first task and then the result of each task as an argument to the next task. This requires each task to return the options object and is the underlying cause of this mixed design the issue wants to tackle.

Implementation process

The issue that was picked contained a number of problems that the Ghost team wanted to be fixed by some refactorings. The main problem that we started focusing on was the replacement of the pipeline that is used to execute a number of tasks in sequence. The next part of the issue that we also focused on was to change how the so called options object that is passed around different layers of the ghost software. The ghost teams' goal was to only use and modify the options object via reference.

This section and the coming headlines will describe the process of our refactoring work with all the ups and downs.

Callback solution

The work started with us trying to implement a solution for how you could pass around the options object in a different way with callbacks. However after after we finally made contact with on of the ghost teams official employees and contributors Katarina Irrgang via their communication platform Slack we understood that they rather were interested in a solution that used promises instead of callbacks.

Trying to replace the pipeline and finding new refactors

In this part of the process we figured it would be a better idea to try and replace the pipeline before changing how the options object was passed around. After trying to replace the pipeline with a solution that would execute tasks in sequential with only a single argument (as the issue suggested) we found out that some tasks needed more than a single argument. We basically discovered that the developers from the ghost team were not exactly sure how the pipeline was used, there was inconsistencies. For example we found that a validation method for the http requests that was often run as the first task in the pipeline required two arguments. After consulting Katarina with this she understood the problem and made a suggestion for a new refactor to pass the second needed argument (http body of a request) for the validation via the options object. The suggestion was to pass the http body by adding it to the multi-purpose variable options.data.

Refactoring the validation of http requests

We accepted the new refactoring challenge and started the work on the refactor of the validation method. This resulted in a lot of changes as almost the entire API (except the mail, db and authentication parts) does validations as the first task of the pipeline execution. We sent the changes for a review to Katarina and she thought it looked great (see Figure 1). However when we ran the test suite there was a lot of failing tests. We found out that many of the tests failed due to the fact that options.data is used by other parts of the application. So using options.data seemed like a great idea at the moment since one of the main developers suggested this. But in

the end it turned out to be a not-so-great choice since it caused a lot of test fails. However when we ran the software server it seemed to work even though the tests were failing. We never had the time to discuss these problems further with the ghost developers.

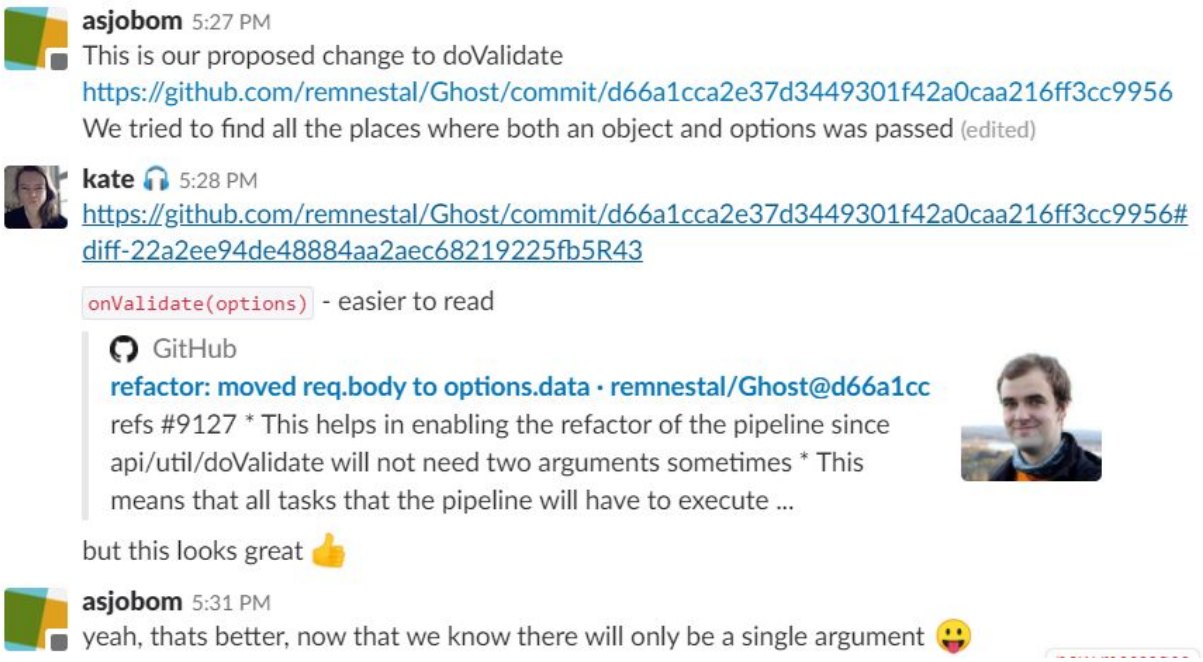


Figure 1. Chat with Katarina (ghost developer) on https://ghost.slack.com after validation refactor commit

Refactoring the pipeline

Since we worked in parallel we started the work on replacing the pipeline simultaneously as the we implemented the validation refactor (as described earlier). After having replaced the pipeline with a new implementation that does not pipe the values between the tasks but only executes them with a reference to the options object. When we combined this refactor with the validation refactor it did not introduce any new errors.

Refactoring the options object

The third refactor was to start looking at how you could change the passing of the options object. After some conversations with the community it was decided that they wanted the response to be added to options.response instead of just being returned as earlier (see Figure 2). We started the work on this refactoring and proposed a solution that we never had the time to discuss with the community.

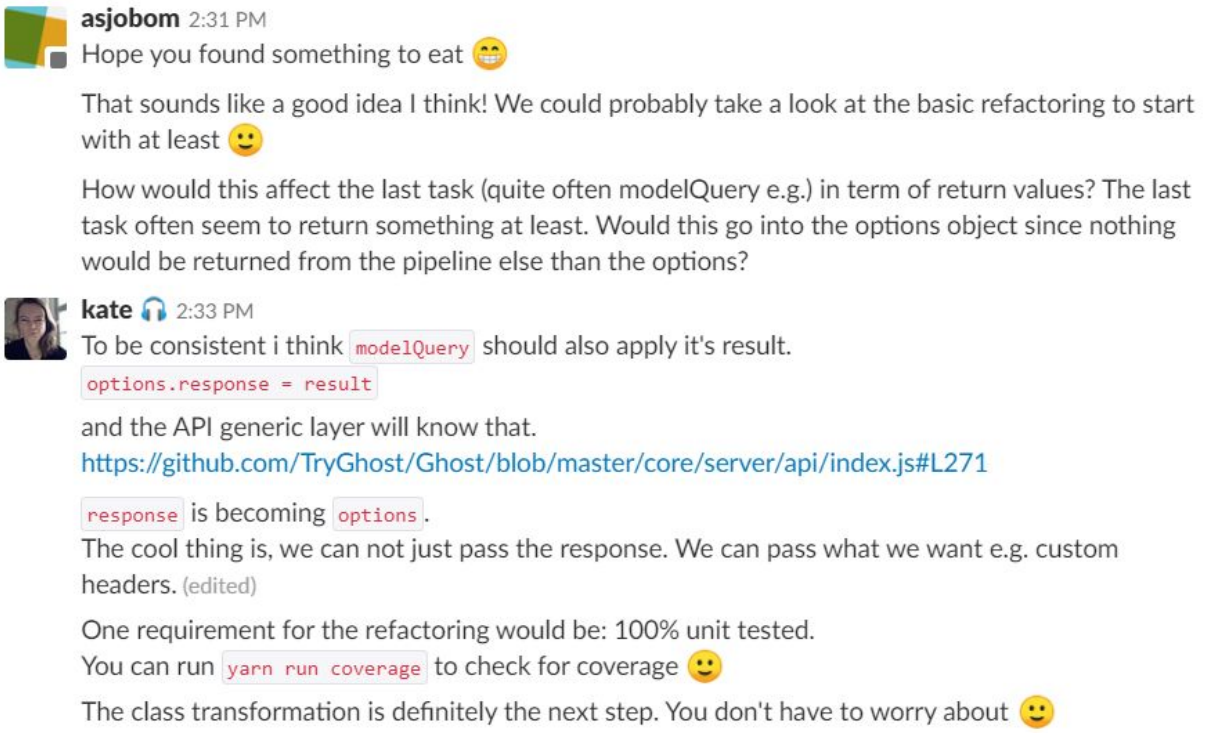
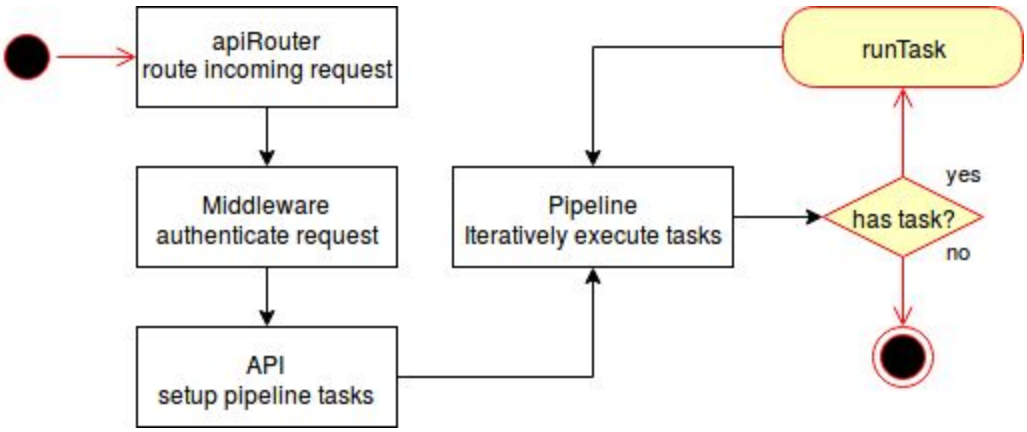


Figure 2. Chat with Katarina (ghost developer) on https://ghost.slack.com on deciding how to return the response after the pipeline execution

Technical description of implementation

The following UML diagram displays the control flow around the refactored code before the changes were made. The most important part is the pipeline, which is the piece of code we shall refactor. The flow of the API has not changed with our refactors and thus, the flow represented by the UML diagram still stands.



The changes made can be categorized in three parts, each described in further detail below:

- Replacing of the pipeline module
- Simplify the validation procedure to only use one argument consistently
- Enforce consistency to the way the *options*-object is modified and accessed throughout the API.

Simplify validation to receive a single argument

The previous version of api/utils/doValidate would sometimes be called by the pipeline with more than one argument, depending on the type of request being made. This was confusing and would have made the refactoring of the pipeline impossible, and hence the arguments were placed in the options-object instead. This idea was the result of a discussion with one of the maintainers, Katarina.

The refactor of the validation process was added through this commit.

Replacing the pipeline

The previous *pipeline* implementation sent the arguments through the different tasks, using the return value of the former as arguments for the latter, in accordance with the well-known pipeline design pattern. The main problem with this pipeline wasn't the implementation itself but rather that it encouraged inconsistent use throughout the API. It seems like the pipeline was first implemented to execute a list of functions sequentially and that the actual pipeline characteristic; to pipe the output of the previous task as input to the following, wasn't actually needed, but was simply a byproduct of borrowing the implementation from another project.

This probably wasn't an issue at first, but after some time other developers started writing code that depended on this side-effect of the pipeline, although that specific functionality wasn't sanctioned for use in that manner. Therefore, the current API implementations has a lot of inconsistencies regarding how the pipeline is utilised. Some API-handlers depend on the ability to pass the output of one task as input to another, while others simply use object modification (by reference) instead.

The developers of Ghost wanted the code to be consistent, and thus wanted us to enforce one way of using the pipeline. The final solution for the refactored pipeline would therefore remove the actual 'pipelining'-abilities of the module, and instead be implemented as a kind of queue that enforces in-order sequential execution and nothing more. This solution was first proposed in the issue and later improved after some discussions with the one of the main developers, Katarina.

The refactor of the pipeline was added through this commit.

Refactor use of options object

The first part of the original refactor issue was to change the way the so called *options*-object was used and referenced. This object is crucial to the success of the API-flow, since it both contains the original request to be read by the API-handlers and is used to store the response

object, successively appended by each handler. It is therefore passed through each handler, which in turn makes necessary changes to the object and response. However, just like how the *pipeline* was used inconsistently, the way the *options*-object was used varied greatly too.

The *options*-object was either modified directly by reference or replaced and returned by the API-handlers in question. Both these implementations actually works in the current state of the project because the pipeline inadvertently accepts the passing of objects between individual tasks executed by the pipeline. This is of course the main characteristic of a *pipeline* but it wasn't the way the developers initially intended the pipeline to be used, as described in the section above.

The original issue wasn't very clear about how the *options*-object was intended to be used, but after some discussions with the Ghost-community we arrived at the first of the observed implementation styles; to modify the object by reference rather than relying on returning and passing the object via the pipeline. This was a fairly large task, since most of the API uses the *options*-object and consisted of two main parts:

- The HTTP response that is constructed jointly by the API handlers should be stored in a object called *response*, inside the aforementioned *options*-object, i.e. in the object called "*options.response"*.
- The attributes of the *options*-object should be modified by reference and not be returned, since the replaced pipeline doesn't pass the returned *options* to the next task in the chain. This way, the next task in the chain will still have access to the changes made through the original *options*-object.

(Note: this problem is very closely coupled with the pipeline, and thus; it actually prompted the issue of replacing the pipeline in the first place.)

Onboarding experience

The build process went more or less as expected. The authors of the project provide a guide on exactly how to install the project at https://docs.ghost.org/v1/docs/working-with-ghost, which includes relevant information such as which versions of nodejs are supported and so forth.

Tests can be run with either "npm test" or "grunt test-all". After building the project and running the entire test-suite, we found that 318 out of 319, 1 out ot 1, 1585 out of 1590 and 583 out of 593 tests passed that are split up into 4 test categories. To run the development environment you use "grunt dev", ghost server is now running on localhost:2368.

Since it is a big project there are many functions and modules that are dependent on pipeline we had to try and understand the project well before trying to make a refactor. Also, none of us are proficient in javascript, node or bluebird which made it even more difficult to understand parts of the code. Even though we spend some time trying to understand as much as possible

we could probably have benefited from spending even more time to understand the codebase before trying refactoring the pipeline.

Functional requirements

The following functional requirements are written with respect to the IEEE 830 standard.

API

These requirements concern methods/objects related to the API layer of the system.

Requirement-1
Objects passed by reference shall be modified and not returned.
Related tests:
/test/functional/routes/api/*

Requirement-2
Each API task shall have access to the latest version of the options-object.
Related tests:
/test/functional/routes/api/*

Requirement-3
Results achieved from the pipeline execution of API tasks should be added to the options-object.
Related tests:
/test/functional/routes/api/*

Note: The options-object is not tested in isolation, but rather the entire API.

Pipeline

The following requirements are put on the implementation of the refactored *pipeline* function. (previous pipeline tests: /Ghost/core/test/unit/lib/promise/pipeline_spec.js)

Requirement-4
The pipeline shall execute tasks sequentially and in order.
Related tests:
/test/unit/lib/promise/sequential.js:22

Requirement-5
The pipeline shall accept a list of tasks as its first argument.
Related tests:
/test/unit/lib/promise/sequential.js:22

Requirement-6
The pipeline shall be able to accept an arbitrary amount of tasks to execute.
Related tests:
/test/unit/lib/promise/sequential.js:28

Requirement-7
The pipeline shall accept an object as its second argument.
Related tests:
/test/unit/lib/promise/sequential.js:22

Requirement-8
The pipeline shall return undefined if options parameter is undefined
Related tests:
/test/unit/lib/promise/sequential.js:34

Requirement-9
The pipeline shall always return a resolved promise.
Related tests:
/test/unit/lib/promise/sequential.js:*

Requirement-10
The pipeline shall be able to accept tasks that are promises.
Related tests:
/test/unit/lib/promise/sequential.js:40

Requirement-11
The pipeline shall invoke each of its tasks with the *options* -object as the only argument.
Related tests:
N/A

Test logs

Overall results with link to a copy of the logs (before/after refactoring):

Testlog of using “grunt test-all --force” before refactoring :
https://drive.google.com/open?id=1nFpSePJGs_W2T7F07sXZjRHyruk_Qjap
(0 failing but 16 “pending”)

Testlog of using “grunt test-all --force” after refactoring:
https://drive.google.com/open?id=11Pi3ZWOkmHnPQmKdoG_BjZTOwwpf9NHl (total tests failed = 250, 0 pending)

Tests added by the refactor

https://github.com/remnestal/Ghost/blob/master/core/test/unit/lib/promise/sequential.js
These tests were added to test the “new pipeline” sequential. They are based on the tests of the old pipeline and modified accordingly to fit the new sequential.js.

Known bugs

The options.data bug
The options.data object, which after the validation refactor contains the body of http-requests, gets overwritten in certain parts of the code, causing some tests to fail. We have not reported this bug in their issue tracker, since we have had a discussion with Katarina who explained that they are aware of the bug and plan to fix it. As can be seen in the issue Katarina suggested that further discussions are needed to decide on how to fix the problems with options.data. One example of a bug fix could be the to change the options object into a class as stated in the long-term-goals of the refactor.

Effort spent

Albin

Task	**Time (h)**
Looking for suitable refactoring issues	2
Studying the chosen issue	2.5
Familiarizing with the codebase	4.5
Project setup	1.5
Studying the API	3.5
Discussing the issue in project group	6.5
Reading up on key javascript features, such as issuing promises with the bluebird library	4
Identifying functional requirements	3
Reading up on the IEEE 830 standard	1
Pair-programming refactoring of *options* together with Johan	4.5
Report writing	5
TOTAL	**38**

Anders

Task	Time (h)
Research to find project	2
Research on issue	1
Project setup	2
Understanding project	6
Running tests	0.5
Doing code refactor that didn't work (callbacks)	5
Talking to the community	4
New code refactor (promises)	14
Report Writing	5
Discussions with group	1
TOTAL	**40.5**

Johan

Task	Time (h)
Research, project finding	2
Project setup	0.5
Javascript learning about callbacks, Promise etc	4
Project and codebase understanding	6
Documentation onboarding experience and overall experience	0.5
Playing with the codebase	1
Discussion within groups and pair programming on pipeline replacement	7
Testing and refactoring code options.respone pair programming with Albin	4.5
Peerreview documentation selected issue, requirements, architecture overview	1
Issue understanding	4
Report last fixes and additions	2
Testing logs and testing	1
TOTAL	33.5

Marcus

Task	Time (h)
Research to find project	2
Reading documentation	2.5
Project setup	3
Running tests	1
Reading code	2.5
Doing UML	2
Learning JavaScript	4
Writing testcases for the new pipeline	3
Writing report text	5
Discussions within group	5
TOTAL	30

Robert

Task	Time (h)
Research	2
Understanding Issue	1
Understanding API/code	6
Project Setup	1
Identify func-Requirements	1
Identify TestCases	1
Pair Programming Code Refactor	3
"Trying" to fix setup errors	1
Discussion	3
new code refactor	11
Report	3
TOTAL	33

Discussion

When we started the work on the project we planned to work on two refactorings (replacing the pipeline and changing how the options object is used). However we found a problem that needed to be addressed in a separate refactor (the validation refactor). This created more work than we anticipated.

The refactor of the pipeline was intended to create a new way of sequentially executing tasks in the api layer. We found out that the previous pipeline implementation (pipeline.js) was not only used by the api layer but also some other parts of the program. This resulted in us creating a new file sequential.js that contained the “new pipeline” and was used by the entire api layer. Since this refactor only should concern the api layer we left the old pipeline.js to not break the other parts of the code outside of the api.

After working with this issue for while and seeing it expand to more and more tasks we think it would probably be better to split the tasks of the issue to several refactor issues since there are quite many things to be done.

Overall experience

This assignment has been very challenging and offered many surprises over the course of the project. All in all, it was still a fun challenge and we’ve learned a lot in the process; both about the challenges of communication in a large open-source project and the difficulties of understanding someone else’s code. Especially if this code was written years ago and has laid dormant when the rest of the codebase has evolved.

These are our main takeaways:

- Different projects have different onboarding experience. Luckily for us, the setup process for *Ghost* was well documented in this document. Since we were new contributors (and new to the open-source community overall) this was a pleasant surprise!
- We now have much more respect for the amount of time and effort It takes to really understand the code base before you can do larger pieces of refactoring. Especially in projects of this size.
- We also have much more respect for the importance of good communication, either via issues or via other mediums such as Slack, Mail etc. If the problem is not described adequately and you have little or no chance of communicating with experienced developers in the project, faulty interpretations are sure to be made.
- We first assumed that a lot of the very old code was “untouchable”; that it is too important and too risky to change. This might be the case in old and big code bases when there is so much coupling going on so that the code gets locked in place. There may be some parts of the Ghost project that is starting to suffer these symptoms, but after further conversation with one of the main developers we learned that they’d long wanted to replace much of the old code.

- Looking for the cause of bugs in a large project written in a loosely typed language can be very difficult, since even simply bugs as typos are only detected during runtime. If this faulty code is placed in an obscure and poorly tested part of the code, it may never be found and documented properly.
- Don’t ignore/procrastinate refactoring. The longer you wait the more the pain.