Flarn — Fun Learning

Software Engineering Project

Supervised by Dr. Tomas Cerny

Dipta Das
Clinton Yeboah
Frimpong Boadu

Baylor University
Project GitHub: https://github.com/diptadas/flarn
Project Website: https://flarn.netlify.com
12-06-2019

Contents

1 Implemented features in brief

1.1 User Panel

1. User registration and login.
2. Email verification during registration.
3. Reset forgotten password.
4. Updating profile, uploading profile picture.
5. Deactivating your account.
6. Subscribe and unsubscribe a user.
7. Viewing list of all users and list of subscribed users.
8. Searching a user by full name.
9. Viewing profile of other users.
10. Viewing list of all problems and list of attempted problems.
11. Searching problems by title, category and difficulty.
12. Getting a random unsolved problem.
13. Attempting a problem.
14. Viewing your submissions, results, and gained points.
15. Reviewing an attempted problem (Star, unstar, and comment).
16. Viewing global and subscription rank.
17. Viewing list of your own activities.
18. Viewing list of subscription's activities (Stories).
19. Support page to contact with admins and report issues.
20. Clarifying privacy policy and licensing during registration.
21. About us page listing team members and supervisor.

1.2 Moderator Panel

1. Creating a problem.
2. Importing a problem from JSON.
3. Editing problems you created.

1.3 Admin Panel

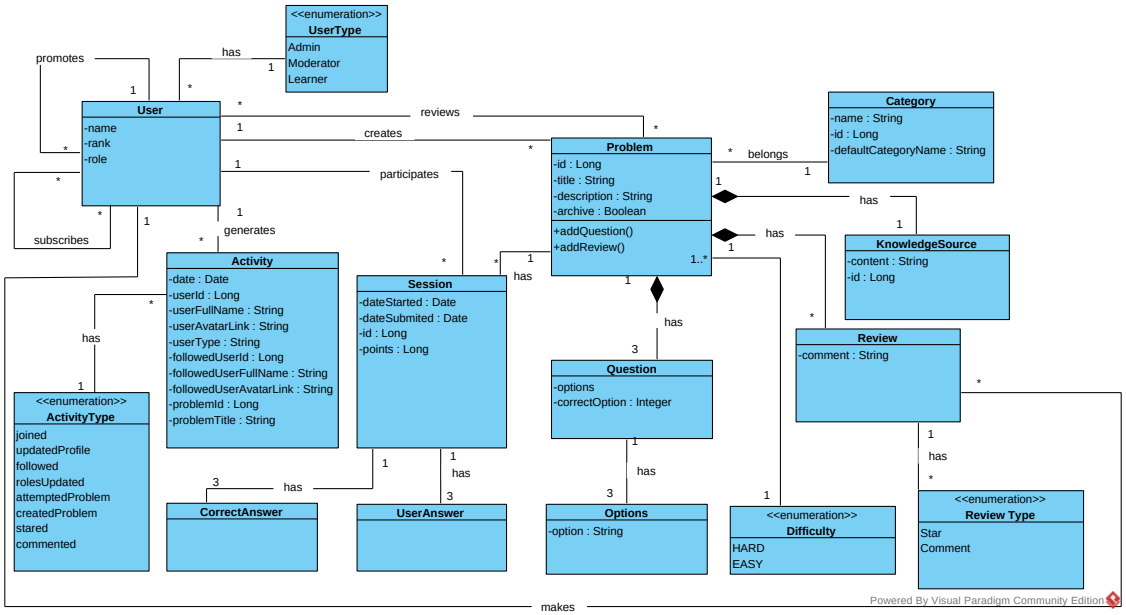
1. Managing problems - editing and archiving a problem.
2. Managing categories - creating a new category, editing existing category, and deleting a category.
3. Managing users - promoting a user.

2 Analysis and Design

During the analysis and design phase we have prepared following artifacts:

Iteration	Artifact	Numbers
Iteration 1	Use cases	10
	System sequence diagrams	10
	Domain model	01
	Business rules	04
	Traceability matrix	01
Iteration 2	Object constraint language	10
	Sequence diagrams	09
	Component diagram	01
	Deployment diagram	01
Iteration 3	Class diagram	01
	Activity diagram	01
	State diagram	01

2.1 Class Diagram



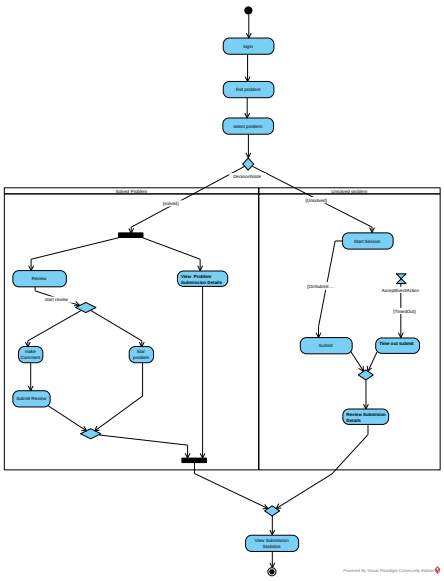
All classes above have accessor methods. We changed the followings from our previous domain model:

1. We removed the inheritance in user model, our actual implementation uses an attribute instead, however constraints on user roles is specified in the OCL submitted in iteration 2.

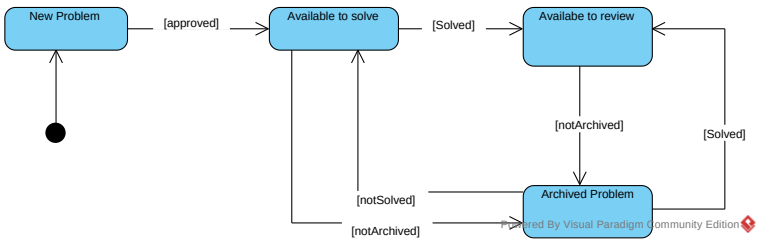
2. We also removed the User and Moderator package separations since we are now using UserType (explained above) within single User class instead of inheritance.

3. Instead of separate class for answers we are using an ordered list of correct answers and user answers in Session class. JPA internally puts that lists into separate table as a foreign key association.

2.2 Activity Diagram for Session

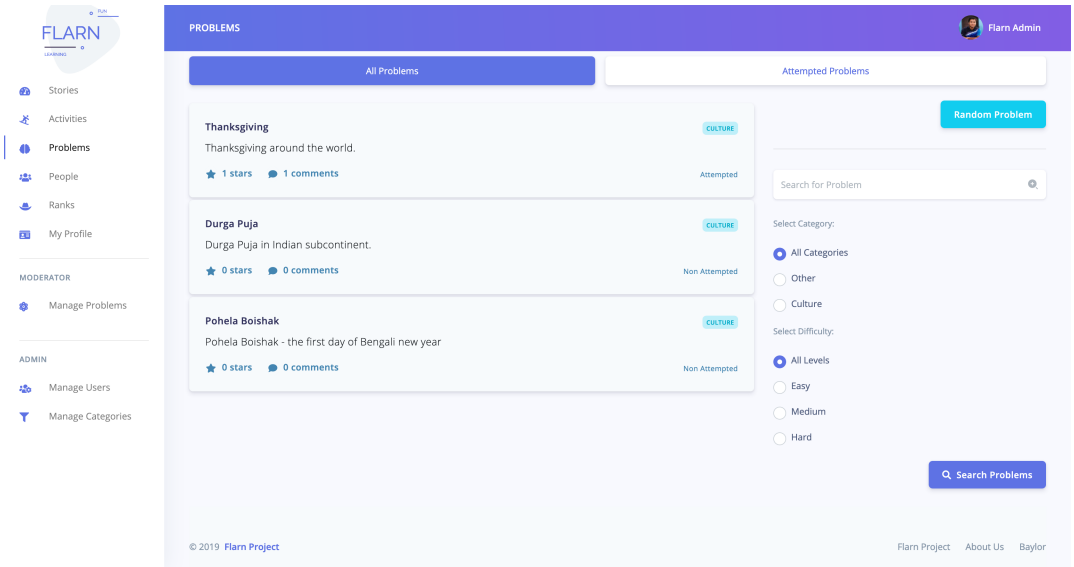


2.3 State Diagram for Problem (considering availability to a user)

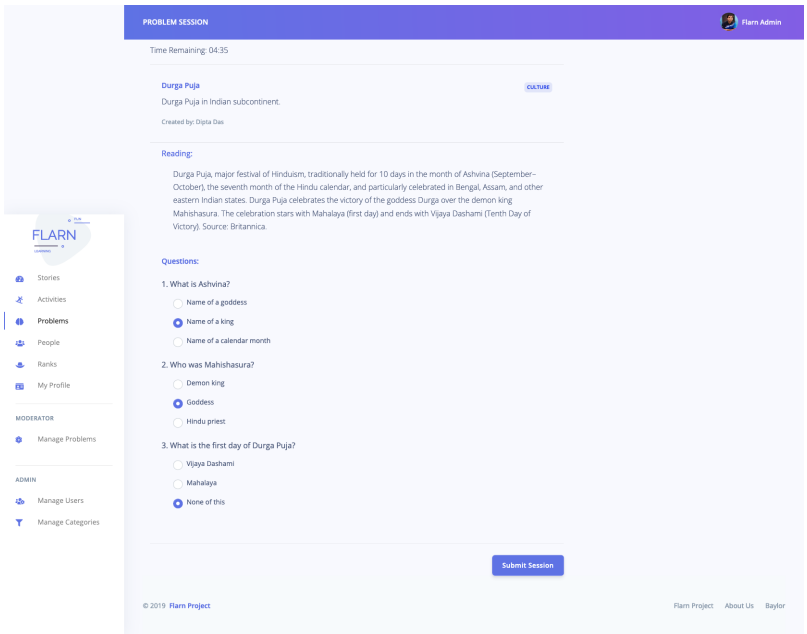


3 User Interface Overview

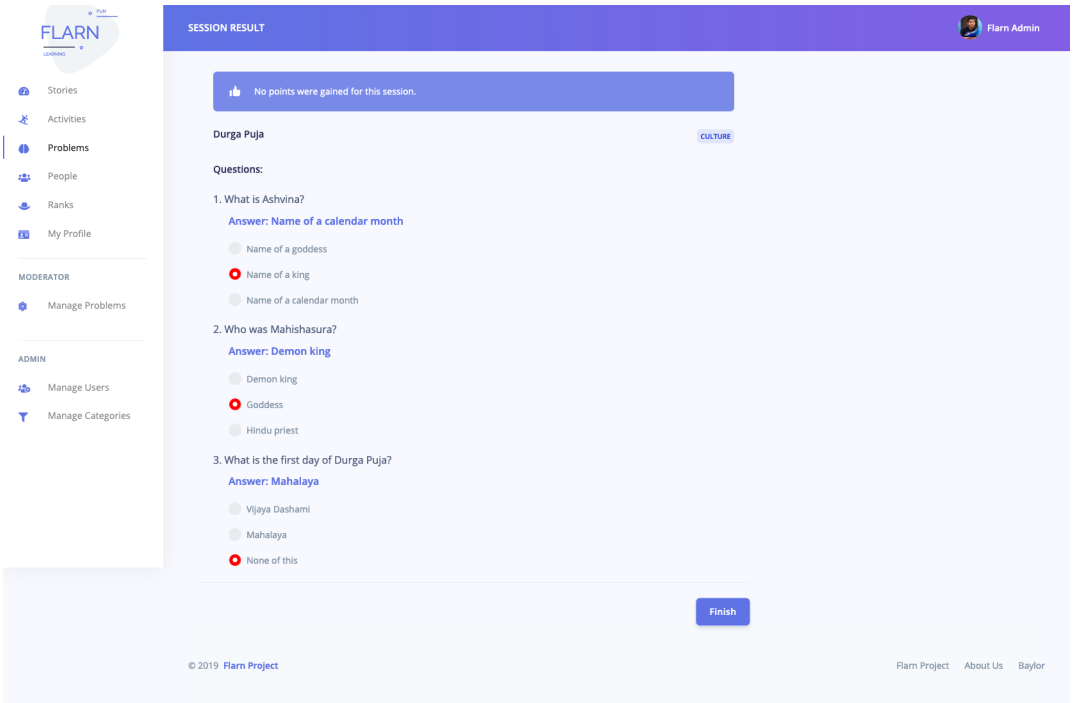
3.1 Problems Page



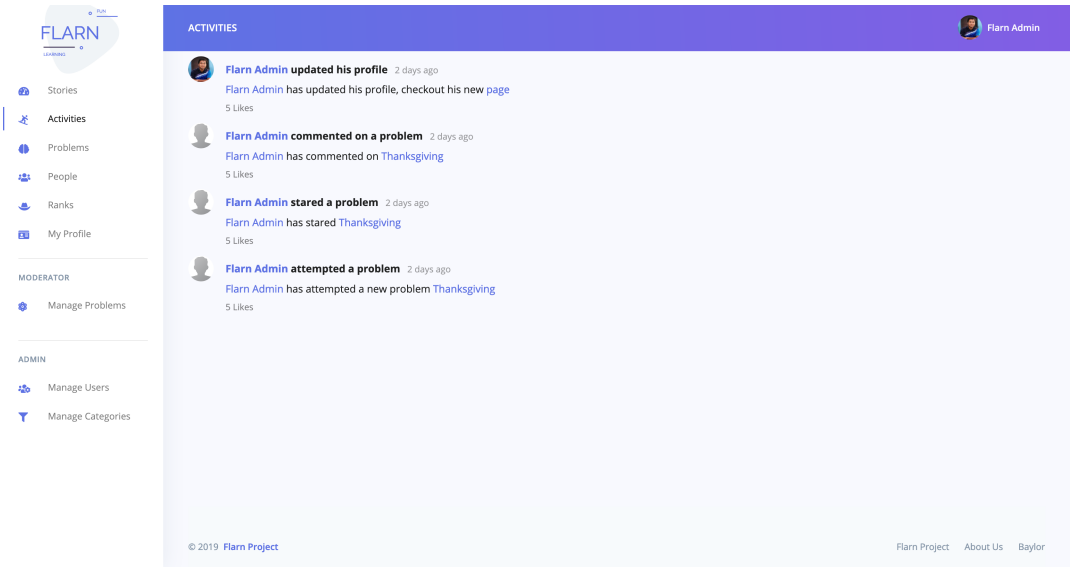
3.2 Problem Session Page



3.3 Session Results Page



3.4 Activities Page



4 Implementation

4.1 Backend

Our backend server is developed using Spring Boot.

Core maven dependencies:

1. Spring Boot Web Starter
2. Spring Boot Security Starter
3. Spring Boot ActiveMQ Starter
4. Spring Boot WebSocket Starter
5. PostgreSQL
6. H2 Database Engine
7. Lombok
8. Springfox Swagger UI
9. SendGrid Java
10. JUnit Jupiter Engine

4.2 Frontend

Our frontend client is developed using Vue JS. We choose VueJS over other frameworks (ReactJS, AngularJS) because VueJS has a lower learning curve and has better DOM optimizations. For more details, check Vue Comparison with other Frameworks. Frontend communicates with Backend server using REST API.

The Frontend has Runtime dependencies and Development dependencies.

For Development, we use the following tools:

1. @vue/cli-plugin-babel Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments
2. @vue/cli-plugin-router Router plugin to handle client-side routing and navigations.
3. @vue/cli-plugin-unit-mocha Run unit tests with mochapack + chai.
4. @vue/cli-plugin-vuex: Vuex plugin for vue-cli. Data/State management.

5. @vue/cli-service Standard Tooling for Vue.js Development.
6. @vue/test-utils Test utilities for vue-cli packages.
7. chai Chai is a BDD / TDD assertion library for testing.
8. sass CSS extension.
9. sass-loader Loads a Sass/SCSS file and compiles it to CSS.
10. shave JavaScript plugin that truncates multi-line text.
11. vue-template-compiler Compiles a template string and returns compiled JavaScript code.

For Runtime, we use the following dependencies:

1. vee-validate Template Based Form Validation Framework for Vue.js.
2. vue The Progressive JavaScript Framework.
3. filepond File uploader.
4. vue-router Official router for Vue.js.
5. vuex State management pattern + library for Vue.js applications.
6. underscore JavaScript utility library.
7. sass CSS extension.
8. timeago.js Library used to format date with '*** time ago' format.
9. axios Promise based HTTP client for the browser and node.js.

4.3 Backend Authentication and Authorization

We utilized Spring Security to enforce authentication and authorization in our backend server. Backend generates a JWT token when user successfully logged in and uses that token to authenticate consecutive rest calls.

We used role based access control to authorize access to certain controller methods. We implemented this using @RolesAllowed and @PreAuthorize annotations on controller methods. Few of our RBAC implementations are listed below:

Use case	Endpoint	Allowed Roles
Create problem	POST /problems	ADMIN, MODERATOR
Edit problem	POST /problems/update	ADMIN, MODERATOR
Archive problem	GET /problems/{id}/archive	ADMIN
Promote User	POST /users/type	ADMIN
Deactivate User	GET users/current/deactivate	USER

4.4 Frontend Authentication and Authorization

Unauthenticated users can only access selected pages. The default page for guest users is the **Login** page. Whiles the default page for authenticated users is the **Stories** page.

After users are authenticated, their token is stored securely in the browser **Session Storage**. This is then used for subsequent requests to the Backend until the user's session expire.

Also, users are restricted to a set of routes based on their authorizations.

4.5 Client-Side Form Validation

To improve user feedback and experience, we provide client-side instant form validations as users provide form inputs. This allows users to correct their mistakes as early as possible before finally submitting them.

4.6 REST API

We used Spring Boot's Rest Controller to implement our REST endpoints. We have implemented about 50 REST APIs in our backend according to the need of our frontend.

4.7 SpringFox Integration

We have integrated SpringFox for automated REST API documentation. This greatly improves our productivity as frontend developer can view the Swagger UI for API definitions instead of asking backend developers every time. The complete list of API documentations are attached in the appendix section.

4.8 Email Verification

During registration user need to verify his email. A confirmation code will be send to user's email after registration. Until email is verified, user account will remain disabled. We used free subscription of SenGrid to send those verification emails which allows 100 emails per day.

4.9 Reset Password

User can reset password of his account using forgot password option. He needs to provide the registered email. A confirmation code will be sent to that email which he can use to set a new password. We used the same mechanism as registration to generate and send confirmation codes through emails.

4.10 Support Page

User can contact the admins by providing useful feedbacks and reporting issues through the support page. It sends the user's message to all the admins of the application through emails. Apart from this it also sends a default reply to user's email that we have received his massage and we will contact him as soon as possible.

4.11 Websocket Implementation

We used a single websocket connection instead of regular REST calls when user starts solving a problem. Whenever user choose a answer for a question, frontend sends a message to backend through the websocket session. Backend remembers the associated user and problem for a specific session-id and stores the response every time a message is received. The final points is calculated when the session is closed. This helps us to solve following interesting issues that we faced when we implemented using regular REST:

1. User might close the browser window after viewing a problem, then start solving the problem again. Now since we mark the problem as attempted whenever the websocket connection is closed, user can not attempt the problem again.
2. There might be technical problems like internet connection drop. Now since we immediately store the user response after he selects any answer, he will get points for whatever he solved before the disruption.

4.12 Websocket Security

We provide an extra layer of security for the Websocket that integrates nicely with our existing Spring Boot Security. Users are requested to send their exiting token as the first message after a successful connection. After this authentication, further requests from the users will be processed normally.

4.13 JMS Implementation

Our verification email sending might observe some delay for few seconds due to high traffic in SenGrid API endpoints. It is a major issue for UI responsiveness. We solved this issue using JMS message queue. Instead of sending email directly, our backend sends the email object to the message queue and sends response to the frontend immediately. Finally, when the JMS client receives the message, it decodes it into email object and sends the email using SendGrid client. We used ActiveMQ for our JMS implementation.

4.14 Development and Production Profiles

We managed separate Spring Boot profiles for development and production environments. We used H2 database for development and PostgreSQL for produc-

tion. Also, in development we used "create-drop" to initialize database tables from scratch. However, in production we used "update". This ensures that our production DB will not loose data in case of server restarts.

Apart from this we maintained separate DataInitializer for dev and prod. We populated some dummy data in development to test the application. However, in production we only populated the admin account and we used environment variables to inject admin credential instead of hard-coded credentials.

4.15 Design Patterns

1. **Singleton**: The Singleton pattern is used in the following by applying the @Bean annotation:
   (a) DbConfig.dataSource()
   (b) DataJpaConfig.auditor()
   (c) SwaggerConfig.api()
   (d) SwaggerConfig.tryItOutConfig()
   (e) WebAppConfig.authenticationManagerBean()
   (f) WebAppConfig.corsConfigurer()
   (g) WebAppConfig.passwordEncoder()
2. **Builder**: The Builder pattern is used in the following:
   (a) DataJpaConfig.auditor()
   (b) SwaggerConfig.api()
   (c) WebAppConfig.configure()
   (d) WebAppConfig.corsConfigurer()
   (e) AuthenticationController.login()
   (f) EmailService.sendSupportEmail()
3. **Observer**: We have implemented WebSockets and JMS which follow the Observer pattern.
4. **Unit Of Work**: We have used EntityManager to initialize dummy data in Data Initiliazer. We used loops to create multiple copies of the users problems and categories. However, we flush the entitymanager only after the loop. This follows the Unit Of Work Design Pattern.

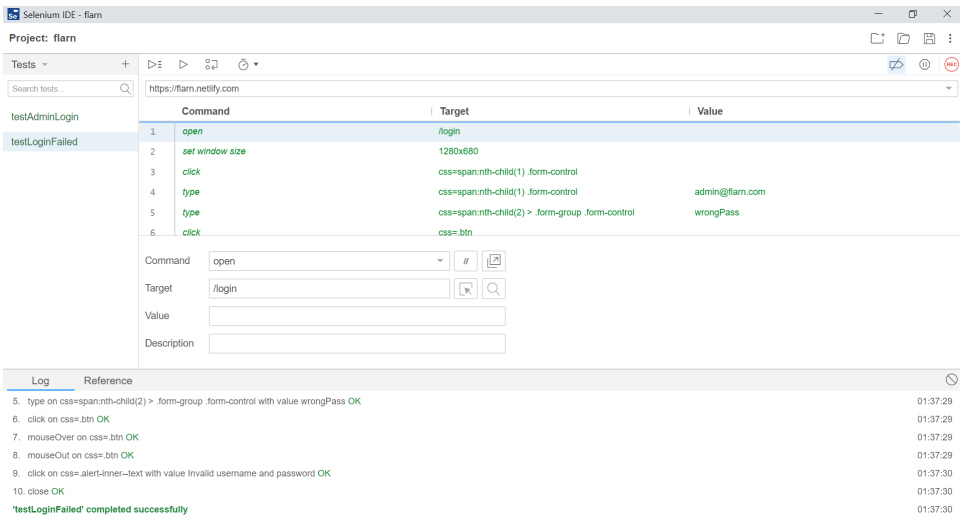
5 Testing And Debugging

5.1 Unit and Integration Test

We designed two general forms of test for our backend application, unit and integration test. These test covered our services and controller mostly. We used JUnit 5 (JUnit Platform + JUnit Jupiter) for these tests. In all, we have 43 unit and integration test, 32 test for our services and 11 test for the controllers.

5.2 Selenium End-to-End Testing

We have also done few end to end testing using selenium Selenium tests. We used Selenium IDE chrome extension to record the tests and export as Junit tests.



5.3 Stress And Load Testing

We performed some stress testing for our application, the different test are listed below:

1. Test the average responds time of our system when injected with a certain number of user at once.

2. Test the responds time of our system when injected with a certain number of user at overtime.

3. Test the responds time of our system when injected with a certain number of users at overtime, at some constant rate. Test two and three are similar to test one except that it injected users are ramped over some time period were conducted once.

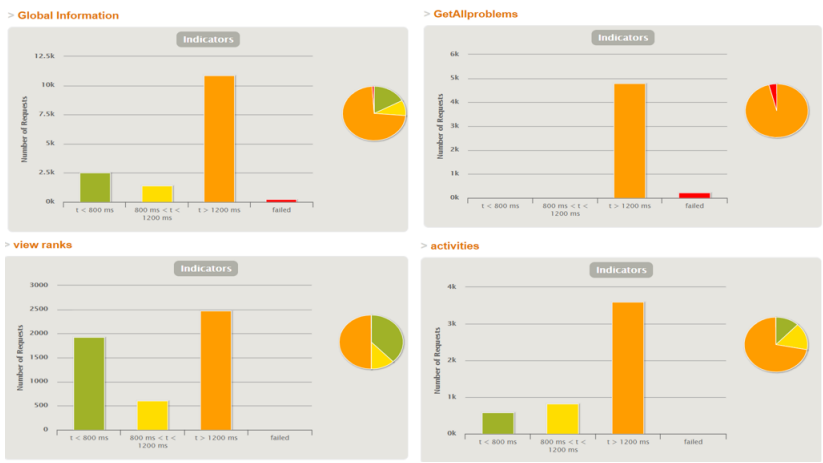
4. Test performance of uploading files in import problem section.

For test 1-3 experiment we tested the following endpoints in our scenarios:

1. https://flarn.netlify.com/problems (endpoint for getting all available problems)
2. https://flarn.netlify.com/users/current/activities (endpoint for getting the current users activities)
3. https://flarn.netlify.com/ranks (endpoint for getting ranking information)

A summary of the results for the first experiment is given below:

$timerange(ms)$	10	100	500	1000	2000	5000
$t < 800ms$	29.67	297	1086.32	$1,999.66$	3971.66	5002.33
$800ms < 1200ms$	0.30	0	186.33	41.33	26.33	1068
$t > 1200ms$	0	3	214	958.66	$1,999.33$	8794.33
$Failed$	0	0	0	0	0	135.33



The complete results for each tests can be found here.

5.4 Static Code Analysis

For code review, we used the PMD tool to ensure that we adhere to the best coding practices. There were some suggestions given by this code review tool which we also ignored. Below is a short list of suggestions and actions we took, we also give reasons why we chose to ignore some suggestions.

PMD report

Problems found

#	File	Line	Problem
1	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\FlarnApplication.java	10	All methods are static. Consider using a utility class instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning.
2	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\FlarnApplication.java	10	Header comments are required
3	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\FlarnApplication.java	11	Parameter 'args' is not assigned and could be declared final
4	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\FlarnApplication.java	11	Public method and constructor comments are required
5	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	12	Each class should declare at least one constructor
6	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	12	Header comments are required
7	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	14	Field comments are required
8	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	14	Found non-transient, non-static member. Please mark as transient or provide accessors.
9	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	16	Field comments are required
10	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	16	Found non-transient, non-static member. Please mark as transient or provide accessors.
11	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	18	Field comments are required
12	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	18	Found non-transient, non-static member. Please mark as transient or provide accessors.
13	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	21	Public method and constructor comments are required
14	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DBConfig.java	23	Local variable 'config' could be declared final
15	E:\Workspace\Java\Java ee\flarn-backend\backend\src\main\java\edu\baylor\flarn\config\DataJpaConfig.java	16	Each class should declare at least one constructor

The above fig shows, a screen shot of PMD report we run on the code, the full is attached to the submission details. Enlisted below are some problems we acknowledged (fixed and ignored):

1. Fixed unused imports issue.
2. Fixed package info missing issue.
3. Changed printStackTrace to log.error.
4. Removed unused local variables.
5. Refactored one line ArrayList initialization using doubled braces to add one by one.
6. Removed unchecked exception signatures.
7. Removed star imports.
8. Ignored too many methods in user controller.
9. Ignored excessive parameter lists.
10. Ignored Junit contains too many assertions, since we doing lot of assertions for each steps of integration tests.

6 Deployment

6.1 Backend

Our Backend application is hosted on **Heroku** using free dynos, which can be accessed at the following domain: Flarn Backend. Deploys are build from GitHub Source after any push to master. The application is rebuilt and the new version is made available online.

As an add-on for Heroku, we have a **Postgres** database running for our production server.

As at now, we have the following deploys:

Builds	Average time	Total time
11	52s	9m

6.2 Frontend

Our Frontend application is hosted on **Netflify**, which can be accessed at the following domain: Flarn Application. Deploys are build from GitHub Source after any push to master. The application is rebuilt and the new version is made available online. At at now, we have
As at now, we have the following deploys:

Builds	Average time	Total time
52	16s	7m

6.3 Environment Variables

We manage environment variables and secrets through the Heroku CLI which are stored safely and retrieved when needed by the various applications.

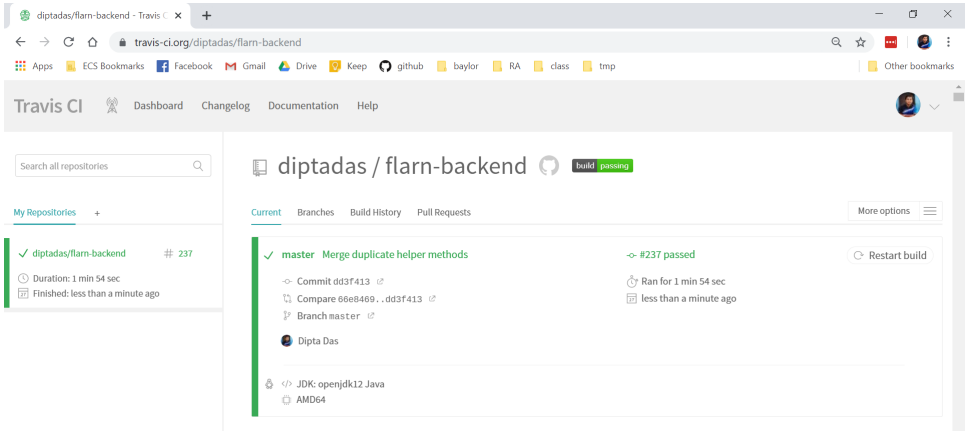
6.4 Monitoring

We monitor the production systems and receives various alerts if the services become unavailable. We employ a free online monitoring tool Uptime Robot.

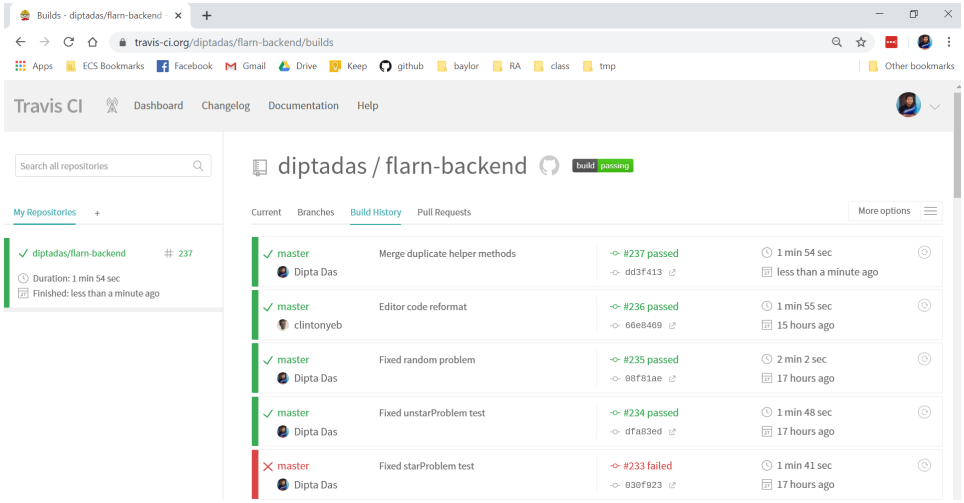
7 CI Integration

We have integrated travis-ci with our Github repository. It runs all the JUnit test whenever a commit is pushed to master branch.

7.1 Current Build



7.2 Build History



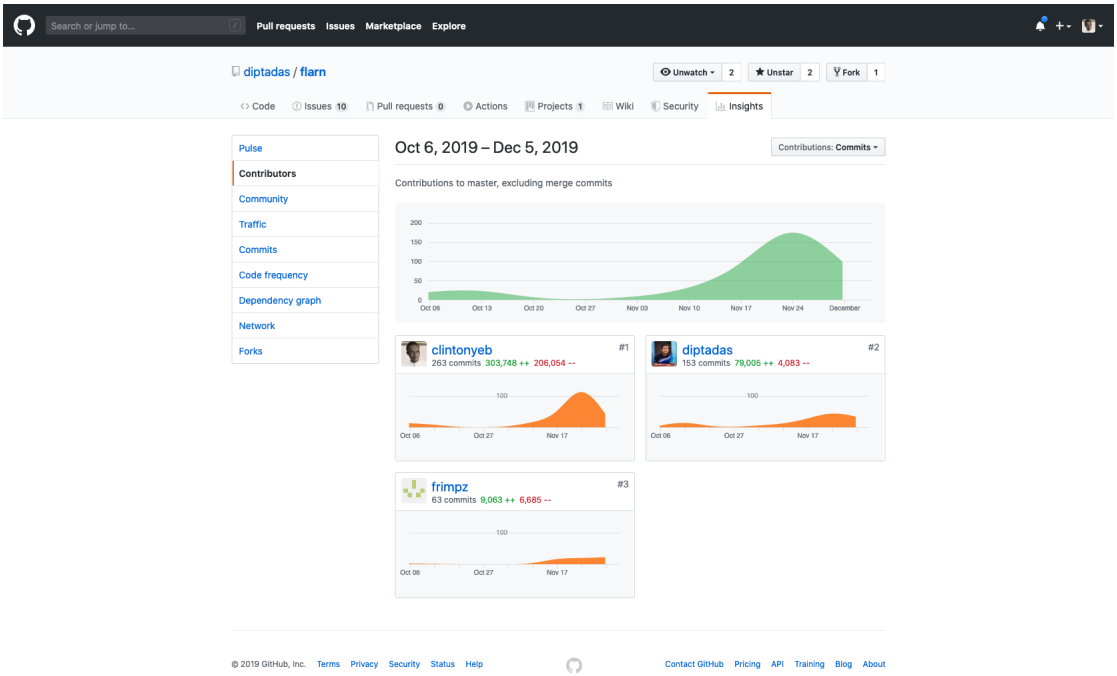
8 Git Statistics

Together we pushed about **530** commits in our Github repository. Followings are the number of commits for each contributors:

1. Clinton Yeboah: 257
2. Dipta Das: 153
3. Frimpong Boadu: 63

We closed about **60** issues and about **10** issues are remained open for future enhancements. Followings are the breakdown of number of issues in different categories:

1. First issue: 01
2. Bug: 16
3. Enhancement: 32
4. Documentation: 06
5. Question: 02
6. Future: 04



9 Value Estimation

Each of us spent about 85 hours on this project. Followings are the breakdown for each iteration:

1. Iteration 1: 10 hours
2. Iteration 2: 25 hours (8 hours per week)
3. Iteration 3: 50 hours (12 hours per week - midterm week + extra hours during thanksgiving)

Assuming average of $30 hourly wage for a software developer in US, total labour value is about $7650 (85*3*30).

We approximated the value of idea and innovation is about $5000, cost for software and tools is about $2000.

This leads to a total value of **$14650** for the whole project.

10 Appendix

10.1 Swagger API Documentation

You can view our API documentation here.

10.2 Generated Java Docs

You can view our java docs here.