Testing Policy Document

Testing Policy Document

Ctrl Alt Defeat

Contents

1 Quality Requirements Testing **2**
- 1.1 Usability . . . 2
- 1.2 Availability . . . 5
  - 1.2.1 Quantification . . . 5
  - 1.2.2 Results . . . 5
- 1.3 Security . . . 6
- 1.4 Performance . . . 7
- 1.5 Scalability . . . 7
- 1.6 Modifiability . . . 7

2 Code Coverage **8**

3 Types of testing **9**

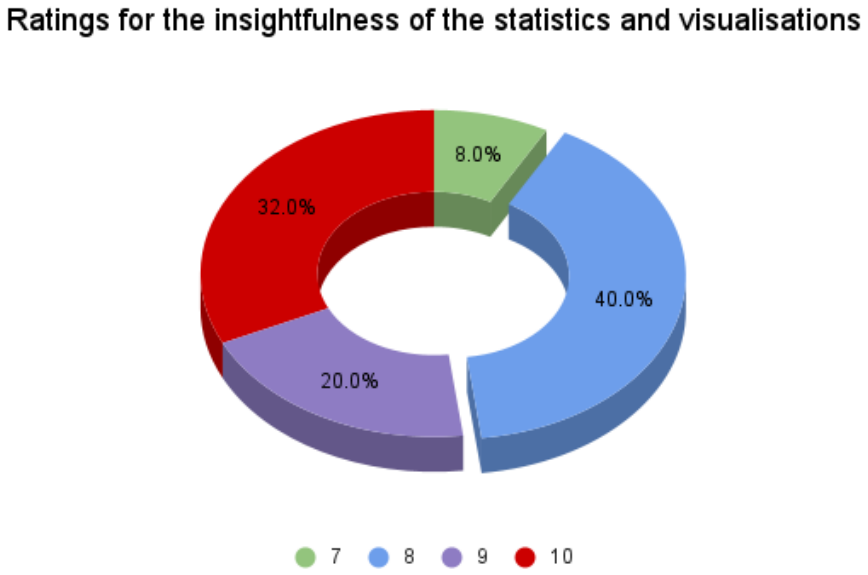
4 Choice of testing tools/frameworks **10**
- 4.1 Frontend Testing . . . 10
- 4.2 Backend Testing . . . 10
- 4.3 Github Actions and workflows . . . 10

1 Quality Requirements Testing

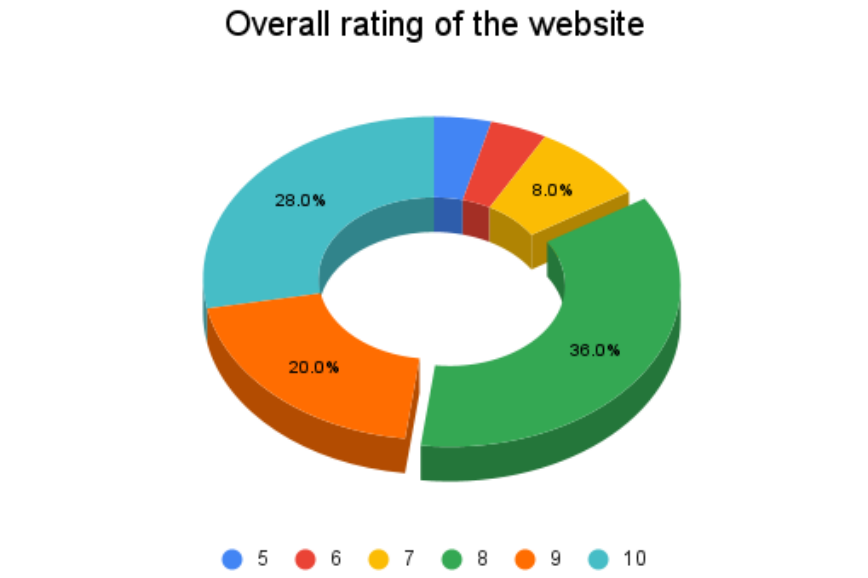
1.1 Usability

Process of Usability Testing:

- **Planning** - The first step in usability testing is to plan what you want to test and how you are going to test it.
- **Recruiting** - The next step is to recruit participants to test the application. The participants should be representative of the target audience of the application but should also be diverse enough to get a wide range of feedback.
- **Testing** - The next step is the actual testing of the application, the way in which we facilitated the testing of the application was as follows:
  - We sent all participants of our usability testing a pdf explaining all aspects of navigating to the app and a list of tasks in which they could try. The pdf can be found HERE.
  - The participant will access the application in their own time in the environment of their liking and use the application to perform tasks that were suggested on the pdf.
  - Once the user has completed the testing of the app, a questionaire was then to be filled out by all participants the questionaire can be found HERE.
- **Feedback implementation** - We collect all data from the questionaire that the testing participants filled out and use this to modify and fix issues accordingly.
- **Metrics for usability testing** - The metrics that we used to measure the usability of our application are as follows:
  - How insghtful the application is which is computed as an average with 10 being the most insightful and 1 being the least insightful. As we can see from the image below the majority of the users we tested rated the insightfulness of the applications statsitics as an 8 or higher.



– The overall rating of the application which is computed as an average with 10 being the best rating and 1 being the worst rating. As we can see from the image below the majority os users rated the application on an overall scale of 8 or higher.



– The amount of responses regarding how easyc the website is to use and navigate ranging from very easy to very diffucult.
– The overall sentiment of the responses regarding the application.

It is understood that the environment in which the users completed the testing in were all up to the users choice which allows for skewed results in terms of the environment variable not being kept constant and things like noise pollution etc can all have effects on how one would use the app but we belive that this bias of an

inconstant environment is neutralised by the adaptability of the app to be used in any environment. NEED TO ADD ONTO REPORT ONCE WE HAVE RESULTS FROM QUESTIONAIRE

1.2 Availability

Availability refers to the probability of a system functioning as intended when desired. Availability is important within our system due to the necessity of the live review service to be available whenever a user requires it.

1.2.1 Quantification

Domain Pulse uses Microsoft Azure App Service for hosting and therefore availability is highly reliant on the availability of the Azure App Service. As well as this, Domain Pulse, hosts a PostgreSQL database and several MongoDB databases on private servers. The availability of these databases is highly improved by redundancy across several servers, with quick switch-over time, allowing for little to no downtime should a database fail. A total Availability of 99.5% per month will be the minimum requirement for the system to be considered acceptably available, whereby availability is calculated as follows:

$$Availability = \frac{Uptime}{Uptime + Downtime} * 100 \tag{1}$$

1.2.2 Results

Microsoft Azure App Service calculates the Uptime Percentage, which is practically the same metric as Availability, using the equation

$$\frac{MaximumAvailableMinutes - Downtime}{MaximumAvailableMinutes} * 100 \tag{2}$$

, where Maximum Available Minutes is the total minutes in a billing month, and Maximum Available Minutes-Downtime is the total number of minutes that the app was unavailable during the billing month. Within the SLA (Service Level Agreement), an Uptime Percentage of less than 99.95% results in a credit refund of 10%, an uptime of less than 99%, results in a credit refund of 25%, and an uptime of less than 95%, resulting in a full refund. This promise of credit refund, shows a high level of confidence in the availability of the Azure App Service, and therefore the availability of Domain Pulse, being higher than 99.95%.

1.3 Security

Security is a very important quality requirement for any application as peoples information is at risk if the application is not secure. The security of our application is of utmost importance as we are dealing with peoples personal information and we need to ensure that this information is kept safe and secure. The security of our application in relation to the OWASP top 10 security vunerabilities is as follows:

- **Broken access control** - We are using Djangos support for user authentication which has been thoroughly tested and is highly secure, with this being said access control is very unlikely to be broken unless one of Djangos authentication libraries had a flaw which is unlikely. Our application only have 1 level of access which means that incorrect access control is unlikely to occur.
- **Cryptographic failures** - To protect our system against cryptographic failures we are using HTTPS and for all our password storage we are using state of the art hashing with salting.
- **Injection** - To protect our system against injection attacks we are using prepared statements provided by Djangos built in ORM (Object Relational Mapper) which is highly secure and has been thoroughly tested. We also use built in functions to execute our queries to ensure that no injection attacks can occur.
- **Insecure design** - We consistently update the architecture of our system which allowed us to identify and fix any insecure design flaws that may have been present in our system.
- **Security misconfiguration** - We have tested our application extensively and have ensured that the way our security configurations work is correct and secure.
- **Vunerable and outdated components** - We have a bot that updates our dependancies automatically and we have no known vunerabilites in our dependancies, we also make use of gitguardian which alerts us if any secrets within the repo somehow get leaked.
- **Identification and Authentication failure** - Once again we use Djangos buite lt in authentication libraries which have been thouroughly tested and are highly secure.
- **Software and data integrity failure** - All libraries and plugins that are used within our system are from trusted sources and hence we can be sure that there are no integrity failures that are known to us.
- **Security logging and monitoring failure** - We use software(Fail2ban) that bans an IP address from accessing our VM if more than 3 attemps to access the Vm are made and failed.
- **Server-side request forgery** - We only allow users to choose from sources that the application trusts and hence dont allow the user to make requests to any other sources which restricts the range of allowed URLs that can be used as input by the user.

1.4 Performance

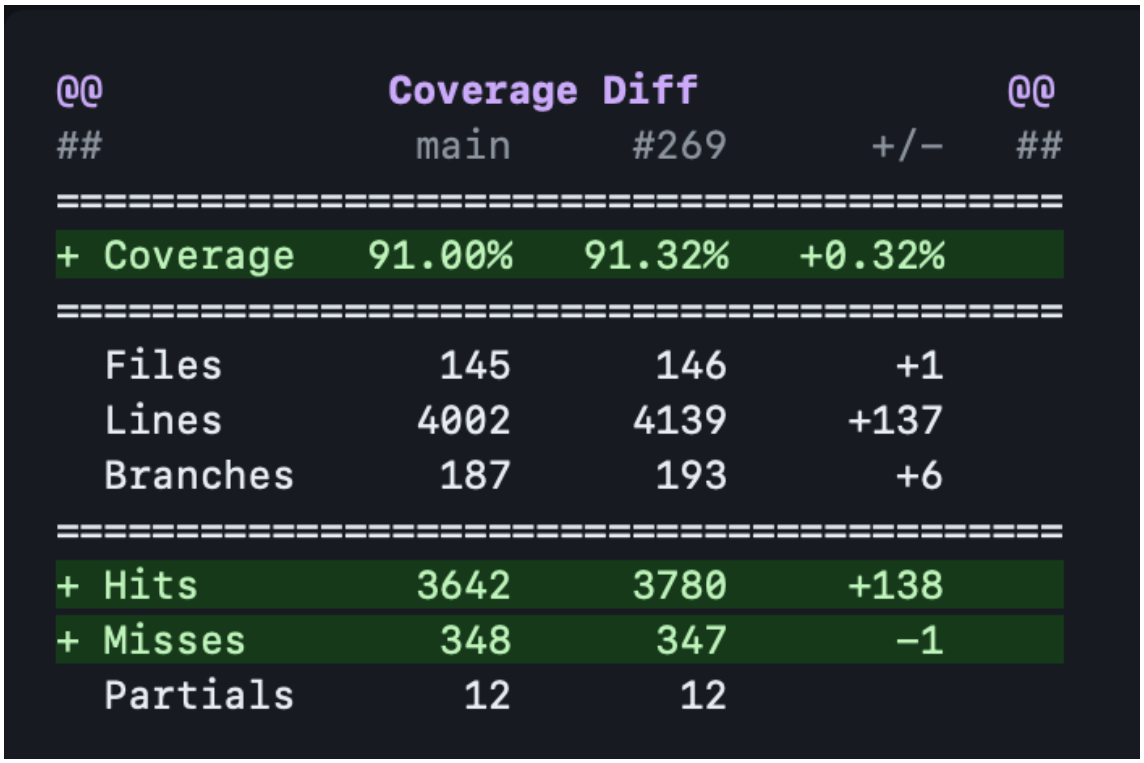
1.5 Scalability

1.6 Modifiability

2 Code Coverage

Any commit made to a branch causes automated tests to be run on the codebase of that branch, thereafter the code coverage of said branch is calculated. Any branch being merged via pull request into the development branch (dev) needs to have the coverage of changes to the codebase to match or better the coverage of the development branch. Furthermore, the coverage of the newly commited code (ie: patch) must match or exceed the coverage percentage of the project. This ensures that the coverage of the codebase is never decreasing, and that sufficient testing is being done on the codebase. Furthermore any time the development branch is merged into the master branch, the coverage of the development branch must match or be higher than that of the master branch, ensuring an increasing coverage and sufficient testing.

- A code coverage report is included in the below image:
- A link to our code cov profile can be found HERE.



@@	Coverage Diff			@@
##	main	#269	+/-	##
+ Coverage	91.00%	91.32%	+0.32%	
Files	145	146	+1	
Lines	4002	4139	+137	
Branches	187	193	+6	
+ Hits	3642	3780	+138	
+ Misses	348	347	-1	
Partials	12	12		

3 Types of testing

- **Unit testing** - Unit testing is when the smallest parts of the application which are known as units are tested to ensure that the correct operation occurs. An example of Unit tests and integration tests can be found in any file ending in "spec.ts" for frontend and an example is given HERE and a backend example can be found HERE.

- **Integration Testing** - Integration testing is when separate components are tested together to ensure the correct operation of these components occur. Integration tests can be mocked or unmocked, mocked tests are when the data used in the tests are not pulled in from an external API or database but rather mocked data is used to simulate the data that would be pulled in from an external API or database. Unmocked tests are when the data used in the tests are pulled in from an external API or database.

- **E2E Testing** - End to end testing is when the application is tested as a whole to ensure that the application is working as expected from the user interface level all the way through the application to the database level and checks all the integration between these componets work as expected. Examples of E2E tests can be found in any file ending in a ".cy.ts" for frontend and an example is given HERE.

4 Choice of testing tools/frameworks

4.1 Frontend Testing

For our frontend testing frameworks and tools we decided to use the following:

- **Karma and Jasmine** - Jasmine is the testing framework that is used to write actual tests and are typed in Javascript, Karma is the test runner that executes the tests. Karma is run from a CLI(Command Line Interface) and it will open up a browser window and run the tests in that browser. Karma will then report the results of the tests back to the CLI and can be used to generate a coverage report. Karma and Jasmine are recommended by Angular which is what our frontend is primarily built upon and they are the most popular testing frameworks for Angular applications. The advantages of using Karma and Jasmine over other testing frameworks is that they are easy to set up and use, they are well documented and they are popular amongst Angular developers which means there is a extensive amount of resources available online for help if need be.

- **Cypress** - Cypress is a testing framework that is used to write end to end tests which are tests that try and simulate a user using the application. End to end tests are needed to ensure that the application is working exactly as expected from the user interface level all the way through the application to the database level and checks all the integration between these componets work as expected. Cypress runs in a browser which makes it easy to setup and follow the tests as they excute in the browser. Cypress is documented well with a thriving community which allows for easy access to information if any problems arise.

4.2 Backend Testing

For our backend testing framework and tools we decided to use the following:

- **Django built in testing module** - Django has a built in testing module which allows for the testing of Django applications, this is yet another reason why we decided to use django as it has amazing functionality out of the box. The django testing module allows for extensive testing of the application. The advantages of using django testing framework over an external framework is that one is already used to the syntax of django since our backend is primarily built on django and hence saves valuable time trying to learn syntax of another backend testing framework. Django allows for fast pace development which is much needed in certain situations such as in ours when following an agile development strategy. Django testing framework is also well documented and has a large community which allows for easy access to information if any problems arise and help is needed.

4.3 Github Actions and workflows

Github actions were used to aid in the automatic testing and generating of code coverage reports for our application. Github actions are workflows that are activated

by certain events such as a merge, push or pull to certain branches of the repository. The workflows are defined in a yaml file and are run on a virtual machine hosted by github, Github actions ensure that all tests pass before a branch is merged into main/master. The workflows that we have defined are as follows:

- backend build
- frontend build
- coverage report
- automatic deployment
- automatic testing