1. Summary of deliverable

   The four capabilities focused on in this deliverable are the todos, projects, categories and interoperability. Each of these capabilities include session notes on exploratory testing, unit test suites for both JSON and XML formats and a summary of session findings.

   Each session of exploratory testing consists of a 45 minute session done by one or two testers. Furthermore, each of the testing sessions is based on the API documentation of said capability. For example, the category feature of the Todo Manager will be explored by trying out each of the endpoints offered in the API documentation page. The exploratory testing will also include endpoints beyond this documentation page, and tests on the examples given for each endpoint.

   The unit test suite for the capabilities follow a similar flow. Each endpoint present in the documentation is tested by an unit test on said endpoint. The tests are designed to restore the state of the application after each test, and to be able to be run in any order. The way these constraints are met on each unit test will be discussed later in the report.

2. Describes findings of exploratory testings

   Todo:

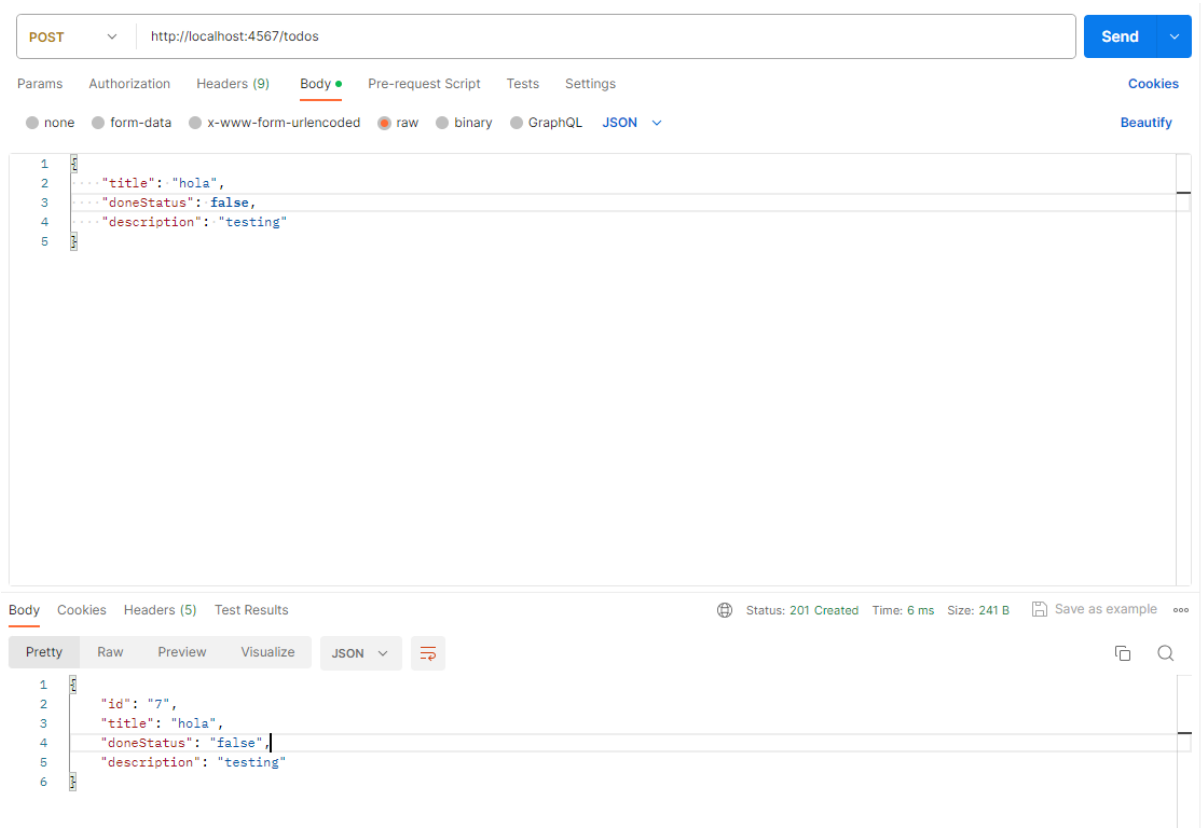
The Todo exploratory test went pretty fast, helped by the documentation both provided by Swagger and the /docs compartment of the application. There were however multiple issues we ran in while testing. The first issue we had was regarding the proposed examples for inputs using JSON format. While in the documentation we can see “doneStatus” field is surrounded by quotation marks, this leads to errors when using this as doneStatus is supposed to be a boolean, not a string.

Example JSON Input to API calls

```
{
  "title": "se cillum dolore eua",
  "doneStatus": "false",
  "description": "eserunt mollit anima"
}
```

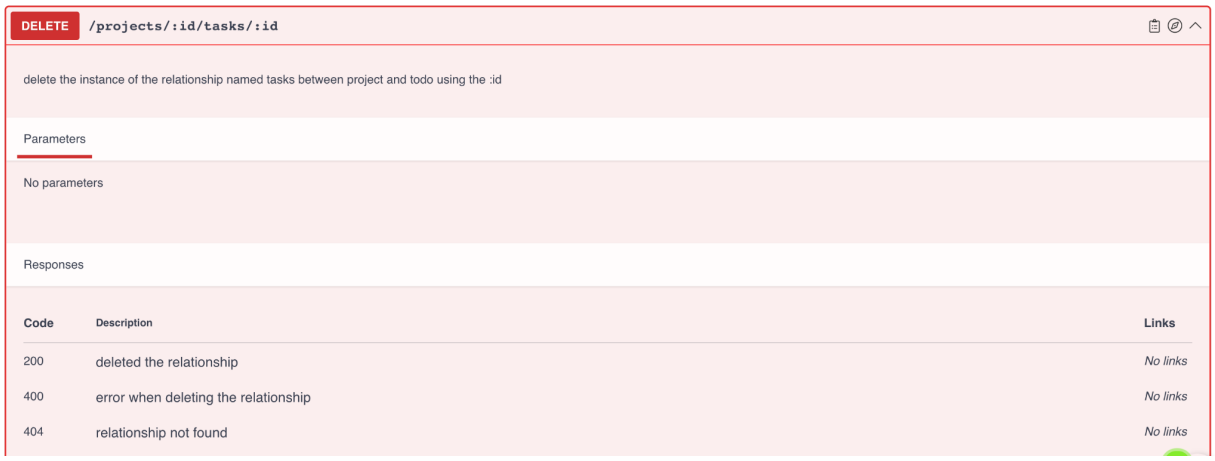
The outputs however concord with what the documentation provides as an example, so that is fine.

Testing the GET method for all the todos associated endpoints was straightforward enough, with all of them returning the right elements if they were added, modified and deleted. PUT and POST led to some extra challenges. For one, if we PUT or POST a wrong formatted todo, it will lead to an error yet the ID counter continues to increment itself. This can be seen in the test where we posted the wrong format 4 times followed by a correct one, in which the new todo was given the ID 7.



Projects:

To set up my exploratory testing, I started by forking the repository of the software to be tested. I then added a copy of the jar file to my local version, pushed it to remote, and shared the repository with my teammates. I ran the jar file on my local host and used swagger to get more in depth documentation of the API.



I systematically tested all the documented endpoints, tried submitting different inputs, as well as testing some undocumented inputs. I conducted my testing using postman to write the URLs, select the action, pass the input and read the output. I documented my findings on an .rtf file which can be found in the repository

Category:

For category, exploratory testing went extremely smoothly. Every single endpoint behaved as one would expect it to. However, there were some inconsistencies found.

The query variables seem to not be functional, despite the documentation suggesting to use in-query variables. Running the suggested request returns an error message which prompts the user to input the title as it is necessary for a category to be created.

However, even by adding the title in the in-query parameters, the exact same error message is prompted. This leaves the only way to create categories through in-body parameters.

Another inconsistency found was the two categories initialized in the software come with no description. With further testing, however, it can be seen that descriptions are not required when creating categories.

3. Describes structure of unit test suite

As discussed in the summary, the unit test suite is based on the endpoints present in the API documentation. This is done to ensure that absolutely every endpoint in every capability is tested properly. Each of these unit tests is designed to restore the state of the system after completion, which means they can also be run in any order. Doing so ensures that they are truly unit tests that can not be affected by another test being run before. The four methods being tested in this Todo Manager are GET, POST, PUT and DELETE. Each of these methods requires a different unit test to verify correctness.

A GET request is quite simple to test. The application contains default todos, projects and categories. There are two types of GET methods in the Todo manager. There is the GET method which gets every instance of the capability, i.e. every category. To test those, the unit test ensures that the default todos, projects or categories are included in the response of the GET request. Similarly, for the GET requests that specify the id, the test assures that the id of the response is the same as the one in the query.

A POST request is tested by first sending a request for a new instance of the tested capability. For example, if the capability being tested is category, the unit test creates a new description and a new title and sends those parameters in the body of the request. Then, the correctness is verified by sending a GET request with the id of the response of the POST request and verifying that the instance with the new id was indeed created. Furthermore, this application sometimes uses POST requests to update existing instances. Those endpoints are tested the same way as PUT requests. Restoration of the initial state is ensured at the end of the test by simply deleting the newly created instance

A PUT request is tested by first posting a new instance. Category will be used as an example again. A new category instance will be created with a new description and title. The POST request then returns the id of the newly created instance as well as the title and description. The test then sends a PUT request with this newly created id, but with a different title and description. The request itself returns as a response the newly updated category, but the test ensures correctness by sending a GET request with the id to ensure the category persisted is the one with the newly updated values. The state is

restored simply by deleting the new instance of the category or any other types of instances.

A DELETE request is tested in a somewhat similar fashion to a PUT request. First, a new instance is created. The unit test verifies that the instance was successfully created, takes note of the newly created id, then sends a DELETE request using the newly created id. Correctness is verified by the response code, and by simply performing another GET request using the id and verifying that the new instance no longer exists within the persistence. The state of the application is already restored when the DELETE request is sent in.

4. Describes source code repository

Our Repository was created after cloning the original eviltester/thingifier repository provided to us in the project description, which is why it may be a bit overwhelming at first glance. Our work is located in the folder named Exploratory_Testing_Group_3. Inside of it, you can find the notes taken during our 3 exploratory sessions, each concentrated on one of the 3 main parts: todos, projects and categories. All our tests are regrouped in the tests.java file located in the Exploratory_Testing_Group_3/src folder. To run the tests, please first run the runTodoManagerRestAPI.1.5.5 jar file on your machine, then go into the testing tab on VSCode and run all tests. We added a RandomOrder annotation found on StackOverflow in order to randomise the execution sequence of the test and prove they don't depend on each other in order to function. A video published here and on Github will show 2 executions in random order back to back to also demonstrate a restart of the database is not needed for them to function. For payload formatting, we used the json.org jar file, which enabled us to simply make new JSON and XML formatted inputs for the requests.

5. Describe findings of unit test suite execution

We had all conducted exploratory testing to identify the capabilities and limitations of the software in question. Following this exploratory testing, the next phase to to systematically verify, document, and prove our findings using a unit test suite. We have described the structure of each kind of of test, and we implemented each of those test for all documented endpoints. Additionally, we implemented tests for endpoints with undocumented functionality, tests

submitting malformed JSON and XML inputs, and test showing documented endpoints malfunctioning. Once all tests were written, we saw them fail when the service is not running and pass when executed in a random order when the service is running.