Intermission Report

Introduction

Intermission is a frontend for regression testing websites using Playwright. It is designed to consistent testing and thorough record keeping as easy as pressing a single button. The name comes from the playwright testing framework created by Microsoft. Intermission is meant to be a "break" from the play once it's been written so to speak.

The Intermission project is contained with the web directory in the repository. The sjuRegression framework for running tests was written prior to the project and other than any changes listed in the Statement of Ranking should be treated as an external package not associated with the in-class project.

Design Philosophy

Intermission has one main goal: Remove the tedium of regression testing as the **user** sees fit.

To follow this philosophy the backend and frontend act independently from each other. Backend api calls are not explicitly made from the frontend alone. If a user wanted to run tests and have the database save results they could make the api call from another script and the frontend would display these changes in real time.

The frontend focuses on removing the tedium by having all interactables immediately available on page load. There's no need to scroll, change pages, or expand anything to run a test. Simply open the page, run the test group, and close the tab. In accordance with this, all changes made to the database are updated live within the frontend, there's no need to ever refresh the frontend.

Frontend Features

Intermission allows the user to interact with playwright pytests in a few unique ways.

Test Groups

Test Groups hold a few details: Names, tests, cron jobs, and retults. These deatails are easily available through tiles contained on the test group dashboard.

Users are able to:

- create new tests groups
- add tests to a test group
- run the tests within a test group
- edit the name and cron job of a test group
- delete a test group.

Tests

Tests hold file names, file paths, test names, and groups. These details are available through tiles contains on the test dashboard.

Users are able to:

- Add new tests from existing test directories
- run single tests adhoc
- delete tests

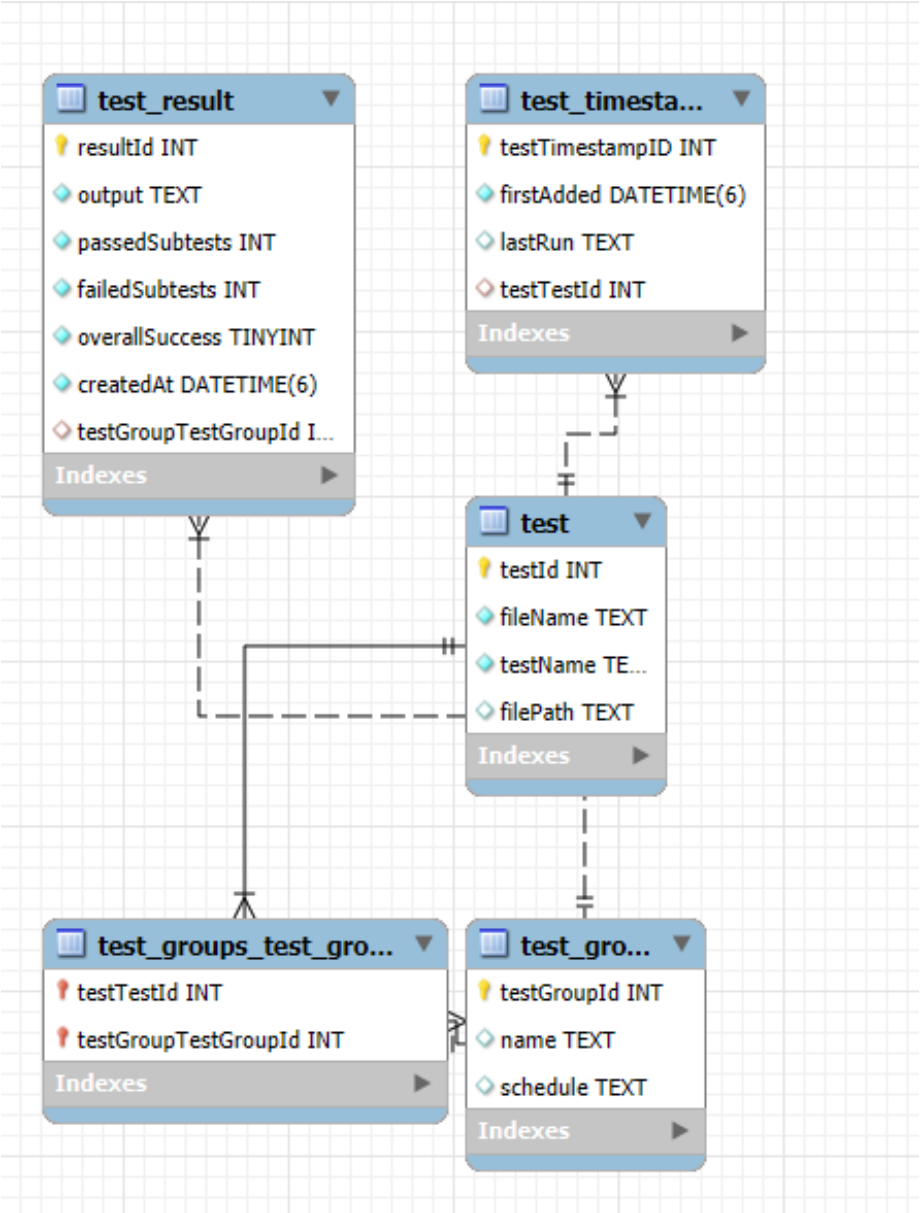
Results

Results hold the full colored pytest output from test group runs and the amount of failed and successes in those runs.

Users are able to:

- Tell at a glance the date and result of tests without expanding details
- View full color pytest outputs
- View amount of successes and failures in each result

ER Diagram


test_result
resultId INT
output TEXT
passedSubtests INT
failedSubtests INT
overallSuccess TINYINT
createdAt DATETIME(6)
testGroupTestGroupId I...
Indexes
test_timesta...
testTimestampID INT
firstAdded DATETIME(6)
lastRun TEXT
testTestId INT
Indexes
test
testId INT
fileName TEXT
testName TE...
filePath TEXT
Indexes
test_groups_test_gro...
testTestId INT
testGroupTestGroupId INT
Indexes
test_gro...
testGroupId INT
name TEXT
schedule TEXT
Indexes

Statement Of Ranking

Antonio J. Cima:

My Teammates and I agreed that I handled 20% of the project

Tasks:

- **Task 1:** I was responsible for researching and implementing axios into the projects (axios is a promise-based HTTP Client for node.js)
- **Task 2:** I was in charge of handling front end functionality, specifically in testListComponents directory (this directory was for the use of the single test tab)
- **Task 3:** That included test functionality (OnUpdateTest), and removing tests (OnRemoveTest)
- **Task 4:** I also assisted with testGroupComponents (reviewing and discussing functionality)

Ryan Smith:

My Teammates and I agreed that I handled 10% of the project

Tasks:

- **Task 1:** I reviewed front end code to ensure proper functionality
- **Task 2:** I presented on the test Group code

Andrew Kantner:

My Teammates and I agree that I handled 70% of the project Tasks:

- **Task 1:** Created the original concept and skeleton for Intermission
- **Task 2:** Handled github creation and maintenence
- **Task 3:** Researched and implemented TypeORM as a backend ORM to handle database transactions
- **Task 4:** Created the ER diagram and structure for the database
- **Task 5:** Created matching entities for use in typescript for code completion which follows the ER diagram
- **Task 6:** Created GET functionality for Test and TestGroup entities through api
- **Task 7:** Researched and implemented Axios to send requests from the frontend to the backend to interact with the database.
- **Task 8:** Created POST for creating Tests through api

- **Task 9:** Created PUT for updating Tests through api
- **Task 10:** Created DELETE for Tests through api
- **Task 11:** Created POST for creating TestGroups through api
- **Task 12:** Changed existing python code to allow for the running of specific tests from python files.
- **Task 13:** Created TestResult Entity which stores results of TestGroups after running
- **Task 14:** Created scheduler in the backend to create batches of cron jobs on server launch and an api call to update specific cron jobs when TestGroups are updated
- **Task 15:** Created POST and PUT for updating TestGroups through api
- **Task 16:** Created a function to populate the DB with example data for testing on first launch
- **Task 17:** Created Types file in frontend so typescript compiler has matching datatypes for our database
- **Task 18:** Created design of TestGroupTiles which are used for holding the data of TestGroups and functionality like adding and removing tests, or running tests
- **Task 19:** Implemented button with matching api calls to add tests to a TestGroup from the frontend
- **Task 20:** Created TestGroupDashboard which holds the TestGroupTiles inside. This allows the dashboard to control which Tiles are showing in accordance with the databse in real time.
- **Task 21:** Created TestGrouptileDetails which are an extension of TestGroupTile which show more details information about the group, like it's recent results and it's tests within the group.
- **Task 22:** Created function for validating cron data on the frontend so the user cannot insert improper cron data into the api.
- **Task 23:** Created two modals for TestGroupTiles: Edit and Set tests. Edit modal allows for the editing of a testGroup's data with checks for junk information and sane defaults if there are no inputs. Set tests allows for the updating of tests within a test group using a list. Both of these update live without refresh.
- **Task 24:** Implemented backend api to create a run button that runs all of the tests listed within the testGroup from it's related Tile.
- **Task 25:** Created a Layout component which holds dashboards. This allows for the main single page layout design so we could swap dashboards

instantly. Layouts hold Dashboards which are controlled by buttons in a side bar, and a header with the matching dashboard title.

- **Task 26:** Designed and implemented TestDashboard and TestTile. TestDashboard is for holding and controlling TestTiles. TestTile holds information for specific tests which can be added to TestGroups.
- **Task 27:** Created modal for creating a new test group. It creates and adds a new TestGroup and allows for empty names safely if the user wants to create multiple then sort them.
- **Task 28:** Designed and Implemented TestResultDashboard which displays all of the results for previous TestGroup runs and allows for easy viewing.
- **Task 29:** Created delete button for TestGroups which removes one from the database and instantly updates the frontend ui with the removal.
- **Task 30:** Created css styling for the TestResultsDashboard and implemented function which parses and displays pytest output in color
- **Task 31:** Implemented regex to parse how many subtests were passed and failed from pytest output so TestGroups recieve correct results after running.
- **Task 32:** Implemented backend api call to run a single test which is not part of a group. Like an adhoc test call.
- **Task 33:** Created css and styling changes for TestGroupDashboard to have animations and have scrolling and clean design.
- **Task 34:** Created css for modals, the layout (dashboard holder), and buttons for creating tests and test groups.

Extra Credit

Is client-side used for validating form data?

Each of the checks for validating data are confirmed on the React frontend before being sent to the api. The only calls which can send back failing codes are if the data is not found within the database. This was handled by only asking for user text input when absolutely necessary (like cron jobs or names) and doing checks to verify that information within the modals before presenting the user with the ability to send data.

Is session tracking implemented?

No