Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика, искусственный интеллект и системы управления

КАФЕДРА Системы обработки информации и управления

Методические указания к лабораторным работам по курсу «Технологии разработки программного обеспечения»

Лабораторная работа №7-8

«Тестирование и оценка качества программного проекта»

Виноградова М.В., Волков А.С., Авдеев Ю.В.., Ваняшкин Ю.Ю., Мурашко И.А., Оганесян Р.Р., Маркова А.В.

Под редакцией к.т.н. доц. Виноградовой М.В.

Москва, 2024 г.

ОГЛАВЛЕНИЕ

1. ЗАДАНИЕ

Лабораторная работа №7-8 «Тестирование и оценка качества программного проекта» по курсу «Технологии разработки программного обеспечения».

1.1. Цель работы

- Изучить методы подготовки и проведения тестирования.
- Получить навыки создания и выполнения тестов для приложений и их компонентов.

1.2. Средства выполнения

- MS Visual Studio (по варианту) или фреймворк Jest (по варианту),
- Раздел Performance в Chrome DevTools,
- Сервис PageSpeed Insights,
- Инструмент SnakeViz,
- Утилита wrk,
- Утилита Locust,
- Библиотека NightWatch.

1.3. Продолжительность работы

Время выполнения лабораторной работы 8 часов.

1.4. Пункты задания для выполнения

1.4.1. Базовое задание

1. Открыть исходный код тестируемого приложения (собственное или выданный преподавателем). Добавить Unit-тест для одной из функций. Запустить тест и просмотреть результаты. Создать несколько разных тестов для проверки значений и перехвата исключений.
2. Установить параметры сбора статистики покрытия кода. Повторить модульные тесты и просмотреть данные о покрытии кода. Сделать выводы о результатах покрытия кода.
3. Провести профайлинг для произвольного веб-сайта (оценку производительности). Помимо стандартных инструментов разработчика

выполнить профайлинг в сервисе PageSpeed Insights. Выполнить тест, продемонстрировать и проанализировать результаты. Сделать выводы о результатах профайлинга.

1.4.2. Расширенное задание

4. Для произвольного **локального** проекта приложения провести профайлинг. Продемонстрировать и проанализировать результаты (время вызова каждой функции, количество вызовов и иерархия вызовов функций). Сделать выводы о результатах профайлинга.
5. Выполнить нагрузочное тестирование произвольного веб-сайта с помощью инструментов wrk и/или Locust. Настроить параметры нагрузки (частота запросов и т.д.) Выполнить тест, продемонстрировать и проанализировать результаты.

1.4.3. Дополнительное задание

6. Создать тестовый проект закодированного тестирования пользовательского интерфейса (например, для калькулятора). Наполнить тест действиями по вводу данных и проверки полученного результата. Выполнить тест и просмотреть результаты.

1.5. Содержание отчета

- Титульный лист;
- Цель работы;
- Задание;
- Описание проделанных шагов;
- Скриншоты с результатами выполнения;
- Вывод;
- Список используемой литературы.

2. ТЕОРЕТИКО-ПРАКТИЧЕСКИЙ МАТЕРИАЛ

2.1. Тестирование

Тестирование программного обеспечения — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, тестирование — это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

2.1.1. Краткий список терминов и определений

Качество программного обеспечения — это совокупность характеристик программного обеспечения, относящихся к его способности удовлетворять установленные и предполагаемые потребности.

Верификация — это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа. Т.е. выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.

Валидация — это оценка соответствия продукта ожиданиям и требованиям пользователей.

План тестирования — это документ, описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Отвечает на вопросы:

- Что надо тестировать?
- Что будете тестировать?
- Как будете тестировать?
- Когда будете тестировать?
- Критерии начала тестирования.
- Критерии окончания тестирования.

Разработка тестов – это этап процесса тестирования ПО, на котором проектируются и создаются тестовые сценарии (тест кейсы), в соответствии с определёнными ранее критериями качества и целями тестирования.

Error — ошибка пользователя, то есть он пытается использовать программу иным способом. Пример: вводит буквы в поля, где требуется вводить цифры (возраст, количество товара и т.п.). В качественной программе предусмотрены такие ситуации и выдаются сообщение об ошибке (error message).

Bug (defect) — ошибка программиста (или дизайнера или ещё кого, кто принимает участие в разработке), то есть, когда в программе, что-то идёт не так как планировалось и программа выходит из-под контроля. Например, когда никак не контролируется ввод пользователя, в результате неверные данные вызывают краш программы. Либо внутри программа построена так, что изначально не соответствует тому, что от неё ожидается.

Failure — сбой (причём не обязательно аппаратный) в работе компонента, всей программы или системы. То есть, существуют такие дефекты, которые приводят к сбоям и существуют такие, которые не приводят. UI-дефекты, например. Но аппаратный сбой, никак не связанный с software, тоже является failure.

Отчет об ошибках (Bug Report) — это документ, описывающий ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Отчёт об ошибке содержит:

- Шапка
- Короткое описание (Summary) Короткое описание проблемы, явно указывающее на причину и тип ошибочной ситуации.
- Проект (Project) Название тестируемого проекта
- Компонент приложения (Component) Название части или функции тестируемого продукта
- Номер версии (Version) Версия на которой была найдена ошибка
- Серьезность (Severity) Наиболее распространена пятиуровневая система градации серьезности дефекта
- Приоритет (Priority) Приоритет дефекта
- Статус (Status) Статус ошибки. Зависит от используемой процедуры и жизненного цикла бага
- Автор (Author) Создатель отчета

- Назначен на (Assigned To) Имя сотрудника, назначенного на решение проблемы
- Окружение
    - ОС / Сервис Пак и т.д. / Браузера + версия /… Информация об окружении, на котором был найден баг: операционная система, сервис пак, для WEB тестирования — имя и версия браузера и т.д.
- Описание
- Шаги воспроизведения (Steps to Reproduce) Шаги, по которым можно легко воспроизвести ситуацию, приведшую к ошибке.
- Фактический Результат (Result) Результат, полученный после прохождения шагов к воспроизведению
- Ожидаемый результат (Expected Result) Ожидаемый правильный результат
- Дополнения
    - Прикрепленный файл (Attachment) Файл с логами, скриншот или любой другой документ, который может помочь прояснить причину ошибки или указать на способ решения проблемы

2.1.2. Этапы тестирования:

Процесс тестирования состоит из следующих этапов:

1. Анализ продукта
2. Работа с требованиями
3. Разработка стратегии тестирования и планирование процедур контроля качества
4. Создание тестовой документации
5. Тестирование прототипа
6. Основное тестирование
7. Стабилизация
8. Эксплуатация

2.1.3. Методы тестирования

Эквивалентное Разделение (Equivalence Partitioning — EP). Как пример, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала — 0.

Анализ Граничных Значений (Boundary Value Analysis — BVA). Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам, или к любого рода сущностям, имеющим ограничения.

Причина / Следствие (Cause/Effect — CE). Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как «Имя», «Адрес», «Номер Телефона» а затем, нажать кнопку «Добавить» — это «Причина». После нажатия кнопки «Добавить», система добавляет клиента в базу данных и показывает его номер на экране — это «Следствие».

Предугадывание ошибки (Error Guessing — EG). Это когда тестировщик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы «предугадать» при каких входных условиях система может выдать ошибку. Например, спецификация говорит: «пользователь должен ввести код». Тестировщик будет думать: «Что, если я не введу код?», «Что, если я введу неправильный код?», и так далее.

Исчерпывающее тестирование (Exhaustive Testing — ET) — это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

Попарное тестирование (Pairwise Testing) — это техника формирования наборов тестовых данных. Сформулировать суть можно, например, вот так: формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров.

Матрица соответствия требований (Traceability matrix) — это двумерная таблица, содержащая соответствие функциональных требований (functional

requirements) продукта и подготовленных тестовых сценариев (test cases). В заголовках колонок таблицы расположены требования, а в заголовках строк — тестовые сценарии. На пересечении — отметка, означающая, что требование текущей колонки покрыто тестовым сценарием текущей строки. Матрица соответствия требований используется QA-инженерами для валидации покрытия продукта тестами. МСТ является неотъемлемой частью тест-плана.

Тестовый сценарий, тестовый вариант, тест (Test Case) — это артефакт, содержащий исходные данные и ожидаемый результат при тестировании. Он также описывает совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Каждый тестовый сценарий должен иметь 3 части:

- Предусловие - PreConditions - Список действий, которые приводят систему к состоянию пригодному для проведения основной проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния.
- Описание тестового сценария - Test Case Description - Список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод о удовлетворении реализации, поставленным требованиям.
- Постусловия - PostConditions - Список действий, переводящих систему в первоначальное состояние (состояние до проведения теста — initial state)

Виды Тестовых Сценариев. Тесты разделяются по ожидаемому результату на положительные и отрицательные:

- Положительный тест использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.
- Отрицательный тест оперирует как корректными, так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций (срабатывание валидаторов), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора.

Чек-лист (check list) — это документ, описывающий что должно быть протестировано. При этом он может быть абсолютно разного уровня детализации. На сколько детальным он будет зависит от требований к отчетности, уровня знания продукта сотрудниками и сложности продукта.

Как правило, список содержит только действия (шаги), без ожидаемого результата. Он менее формализован, чем тестовый сценарий. Его уместно использовать тогда, когда тестовые сценарии будут избыточны. Также чек-лист ассоциируются с гибкими подходами в тестировании.

Дефект (баг) – это несоответствие фактического результата выполнения программы ожидаемому результату. Дефекты обнаруживаются на этапе тестирования программного обеспечения (ПО), когда тестировщик проводит сравнение полученных результатов работы программы (компонента или дизайна) с ожидаемым результатом, описанным в спецификации требований.

2.1.4. Виды и типы тестирования

Тестирование программных продуктов можно разбить на виды (см. Рисунок 1)

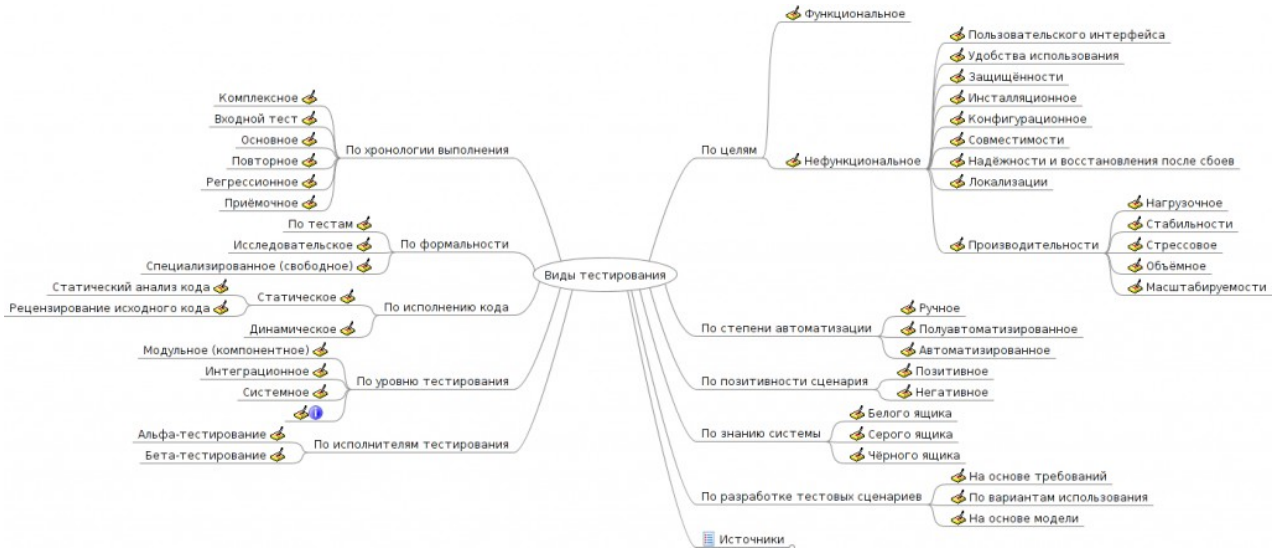


Рисунок 1. Виды тестирования

2.1.4.1. Виды Тестирования (уровни)

Модульное тестирование (Unit Testing). Компонентное (модульное) тестирование проверяет функциональность и ищет ошибки в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.).

Интеграционное тестирование (Integration Testing). Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

Системное тестирование (System Testing). Основной задачей системного тестирования является проверка как функциональных, так и не функциональных

требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д.

Операционное тестирование (Release Testing). Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес моделе системы. Следует учесть, что и бизнес-модель может содержать ошибки. Поэтому так важно провести операционное тестирование как финальный шаг валидации. Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения — критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО.

Приемочное тестирование (Acceptance Testing). Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью: определения удовлетворяет ли система приемочным критериям и вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

2.1.4.2. Типы тестирования

Функциональные виды тестирования.

- Функциональное тестирование (Functional testing). Проверка корректности и полноты выполнемых системой функций в соответствии с требованиеми к ней. Рассматривает заранее указанное функцинное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.
- Тестирование пользовательского интерфейса (GUI Testing). Функциональная проверка интерфейса на соответствие требованиям — размер, шрифт, цвет, consistent behavior.
- Тестирование безопасности (Security and Access Control Testing). Это стратегия тестирования, используемая для проверки безопасности системы, а

также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

- Тестирование взаимодействия (Interoperability Testing). Это тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в себя тестирование совместимости (compatibility testing), и интеграционное тестирование.

Нефункциональные виды тестирования.

- Все виды тестирования производительности:
    - нагрузочное тестирование (Performance and Load Testing). Это автоматизированное тестирование, имитирующее работу определенного количества бизнес-пользователей на каком-либо общем (разделяемом ими) ресурсе.
    - стрессовое тестирование (Stress Testing). Оно позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера.
    - тестирование стабильности или надежности (Stability / Reliability Testing). Задачей этого вида тестирования является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки.
    - объемное тестирование (Volume Testing) Задачей является получение оценки производительности при увеличении объемов данных в базе данных приложения.
- Тестирование установки (Installation testing) Оно направленно на проверку успешной инсталляции и настройки, а также обновления или удаления программного обеспечения.
- Тестирование удобства пользования (Usability Testing) Это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

- Тестирование на отказ и восстановление (Failover and Recovery Testing) Оно проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети). Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.
- Конфигурационное тестирование (Configuration Testing) Специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.).

Связанные с изменениями виды тестирования

- Дымовое тестирование (Smoke Testing) Оно рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение стартует и выполняет основные функции.
- Регрессионное тестирование (Regression Testing) Это вид тестирования направленный на проверку изменений, сделанных в приложении или окружающей среде (устранение дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде. Регрессионными могут быть как функциональные, так и нефункциональные тесты.
- Повторное тестирование (Re-testing) Тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.
- Тестирование сборки (Build Verification Test). Тестирование, направленное на определение соответствия, выпущенной версии, критериям качества для начала тестирования. По своим целям является аналогом Дымового Тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию. Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии.

- Санитарное тестирование или проверка согласованности/исправности (Sanity Testing) Узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Обычно выполняется вручную.

2.1.5. Дефекты

Дефекту или багу, выявленному в ходе тестирования, присваивается степень серьезности в зависимости от того, насколько баг влияет на работу программного продукта. Всего их пять:

- S1 Блокирующая (Blocker). Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.
- S2 Критическая (Critical). Критическая ошибка, неправильно работающая ключевая бизнес-логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, без возможности решения проблемы, используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системой.
- S3 Значительная (Major). Значительная ошибка, часть основной бизнес логики работает некорректно. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки.
- S4 Незначительная (Minor). Незначительная ошибка, не нарушающая бизнес логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.
- S5 Тривиальная (Trivial). Тривиальная ошибка, не касающаяся бизнес-логики приложения, плохо воспроизводимая проблема, малозаметная посредствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Также ошибке выставляют приоритет. Он нужен для того, чтобы разработчику было проще определить с исправление какого бага ему начать. Приоритеты бывают:

- P1 Высокий (High). Ошибка должна быть исправлена как можно быстрее, т.к. ее наличие является критической для проекта.
- P2 Средний (Medium). Ошибка должна быть исправлена, ее наличие не является критичной, но требует обязательного решения.
- P3 Низкий (Low). Ошибка должна быть исправлена, ее наличие не является критичной, и не требует срочного решения.

2.1.6. Принципы тестирования

Принцип 1. Тестирование демонстрирует наличие дефектов. Тестирование может показать, что дефекты присутствуют, но не может доказать, что их нет. Тестирование снижает вероятность наличия дефектов, находящихся в программном обеспечении, но, даже если дефекты не были обнаружены, это не доказывает его корректности.

Принцип 2. Исчерпывающее тестирование недостижимо. Полное тестирование с использованием всех комбинаций вводов и предусловий физически невыполнимо, за исключением тривиальных случаев. Вместо исчерпывающего тестирования должны использоваться анализ рисков и расстановка приоритетов, чтобы более точно сфокусировать усилия по тестированию.

Принцип 3. Раннее тестирование. Чтобы найти дефекты как можно раньше, активности по тестированию должны быть начаты как можно раньше в жизненном цикле разработки программного обеспечения или системы, и должны быть сфокусированы на определенных целях.

Принцип 4. Скопление дефектов. Усилия тестирования должны быть сосредоточены пропорционально ожидаемой, а позже реальной плотности дефектов по модулям. Как правило, большая часть дефектов, обнаруженных при тестировании или повлекших за собой основное количество сбоев системы, содержится в небольшом количестве модулей.

Принцип 5. Парадокс пестицида. Если одни и те же тесты будут прогоняться много раз, в конечном счете этот набор тестовых сценариев больше не будет находить новых дефектов. Чтобы преодолеть этот “парадокс пестицида”, тестовые сценарии должны регулярно рецензироваться и корректироваться, новые тесты должны быть разносторонними, чтобы охватить все компоненты программного обеспечения, или системы, и найти как можно больше дефектов.

Принцип 6. Тестирование зависит от контекста. Тестирование выполняется по-разному в зависимости от контекста. Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем сайт электронной коммерции.

Принцип 7. Заблуждение об отсутствии ошибок. Обнаружение и исправление дефектов не помогут, если созданная система не подходит пользователю и не удовлетворяет его ожиданиям и потребностям.

2.2. Модульное тестирование

Модульное тестирование получило такое название, так как функции программы разбиваются на отдельные тестируемые участки поведения, которые можно протестировать в качестве отдельных модулей. Обозреватель тестов Visual Studio предоставляет гибкий и эффективный способ запуска модульных тестов и просмотра результатов в Visual Studio. Visual Studio устанавливает платформы модульного тестирования Microsoft для управляемого и машинного кода. Платформа модульного тестирования используется для создания модульных тестов, их запуска и создания отчетов о результатах таких тестов.

Модульное тестирование максимально влияет на качество кода, когда оно является неотъемлемой частью рабочего процесса разработки ПО. После написания функции или другого блока кода приложения создаются модульные тесты, которые проверяют поведение кода в ответ на стандартные, граничные и некорректные случаи ввода данных; также проверяются любые явные или предполагаемые допущения, сделанные кодом. При разработке, управляемой тестами, модульные тесты создаются перед написанием кода, поэтому модульные тесты используются в качестве технической документации и спецификации функциональности.

2.2.1. Покрытие кода тестами

Чтобы определить, какая часть кода проекта в действительности тестируется закодированными тестами, такими как модульные тесты, можно воспользоваться возможностью покрытия кода в Visual Studio. Для обеспечения эффективной защиты от ошибок тесты должны выполнять ("покрывать") большую часть кода.

Покрытие кода возможно при выполнении методов тестов с помощью обозревателя тестов. В таблице результатов отображается процент кода, который был выполнен в каждой сборке, классе и методе. Кроме того, редактор исходного кода

показывает, какой код был протестирован. Пример отчета о покрытии исходного кода изображен на Рисунке 2.

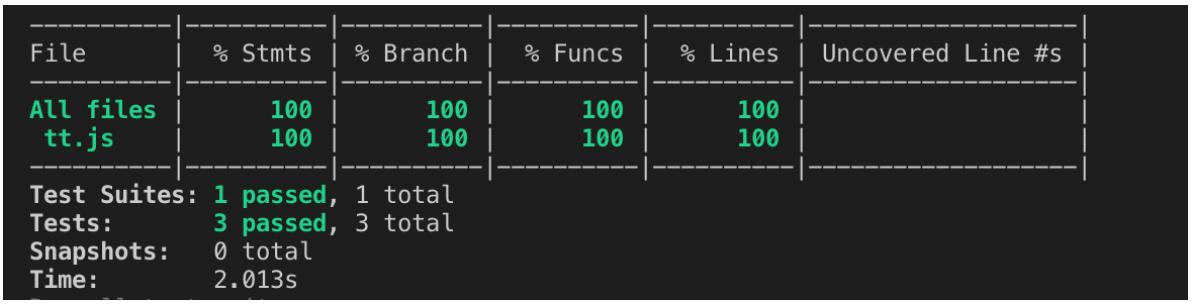


Рисунок 2. Покрытие кода

Данный вид теста запускается с помощью команды `npm run test -collectCoverage` в командной строке.

Во время работы каждого тестового примера выполняется некоторый участок программного кода системы, при выполнении всей системы тестов выполняются все участки программного кода, которые задействует эта система тестов. В случае, если существуют участки программного кода, не выполненные при выполнении системы тестов, система тестов потенциально неполна (т.е. не проверяет всю функциональность системы), либо система содержит участки защитного кода или неиспользуемый код (например, "закладки" или задел на будущее использование системы). Таким образом, отсутствие покрытия каких-либо участков кода является сигналом к переработке тестов или кода (а иногда – и требований).

К анализу покрытия программного кода можно приступать только после полного покрытия требований. Полное покрытие программного кода не гарантирует того, что тесты проверяют все требования к системе. Одна из типичных ошибок начинающего тестировщика – начинать с покрытия кода, забывая про покрытие требований.

2.2.2. Фреймворк Jest

Для покрытия кода тестами предлагается использовать Jest (https://jestjs.io/ru/).

Jest — это фреймворк для тестирования, разработанный Facebook. Он основан на Jasmine. По состоянию на сегодняшний день компания Facebook переработала большую часть его функционала и создала на его основе множество новых возможностей.

Среди особенностей Jest можно отметить следующие:

- Производительность. Фреймворк Jest признан самым быстрым в применении к большим проектам со множеством файлов тестов благодаря реализации интеллектуального механизма параллельного тестирования.
- Пользовательский интерфейс. Интерфейс Jest отличается понятностью и удобством.
- Наличие всего необходимого для начала работы. Jest поставляется со средствами создания утверждений, функций-шпионов и имитаций, которые эквивалентны отдельным библиотекам, вроде Sinon, выполняющим те же функции.
- Глобальные переменные. Как и в случае с Jasmine, Jest, по умолчанию, создаёт глобальные переменные тестирования, поэтому их не нужно явно подключать. Эту особенность можно воспринимать и как недостаток, так подобный подход вредит гибкости тестов и возможностям по управлению ими, но в большинстве случаев это всего лишь упрощает работу:

```
// "describe" уже в глобальной области видимости
// поэтому данные команды импорта не требуются:
// import { describe } from 'jest'
// import { describe } from 'jasmine'

describe('calculator', function() {
  ...
})
```

- Анализ покрытия кода тестами. В Jest имеются мощные и быстрые встроенные средства для анализа покрытия кода тестами.

2.2.3. Модульные тесты для функции multiplay

Имеется функция multiplay (см. Рисунок 3). Написана на языке JavaScript. Данная функция принимает 2 числа на вход и выводит их произведение. Тест написан с использованием библиотеки Jest. Jest можно установить, выполнив команду:

```
npm i jest
```

Тест запускается с помощь команды `npm run test` в командной строке.

```
function multiplay(a, b) {
    return a * b;
};

module.exports = { multiplay };
```

Рисунок 3. Функция multiplay

В начале каждого теста вызывается функция `test` принимающая на вход комментарий к тесту и другую функцию. В теле другой функции необходимо вызвать функцию `expect`, которой в качестве параметров мы передадим тестируемую функцию и ее параметры (входные данные для тестирования). После вызывается метод `toBe` в который передается ожидаемый результат выполнения тестируемой функции.

В данном примере (см. Рисунок 4) описано 3 unit теста каждый из которых принимает на вход пары чисел: (1,3), (2,3), (4,10) соответственно. После вызывается метод `toBe` в который передается значение, которое мы ожидаем после выполнения данной функции 3, 6, 40 соответственно.

```
const { multiplay } = require("./multiplay");

test('multiplay: adds 1 * 3 to equal 3', () => {
    expect(multiplay(1, 3)).toBe(3);
});
test('multiplay: adds 2 * 3 to equal 3', () => {
    expect(multiplay(2, 3)).toBe(6);
});
test('multiplay: adds 4 * 10 to equal 3', () => {
    expect(multiplay(4, 10)).toBe(40);
});
```

Рисунок 4. Unit-тесты

Функция `test` возвращает либо успешное сведения о результатах выполнения тестов. Если тесты выполнены успешно и полученный результат равен ожидаемому, то возвращается `PASS`, в противно случае возвращается `ERROR`.

Результаты тестирование приведены на Рисунке 5. По ним понятно, что проверялась одна функция (`Test Suites`), по 3 тестам (`Tests`). Все тесты прошли успешно, время занятое тестирование 1.228sec (`Time`)

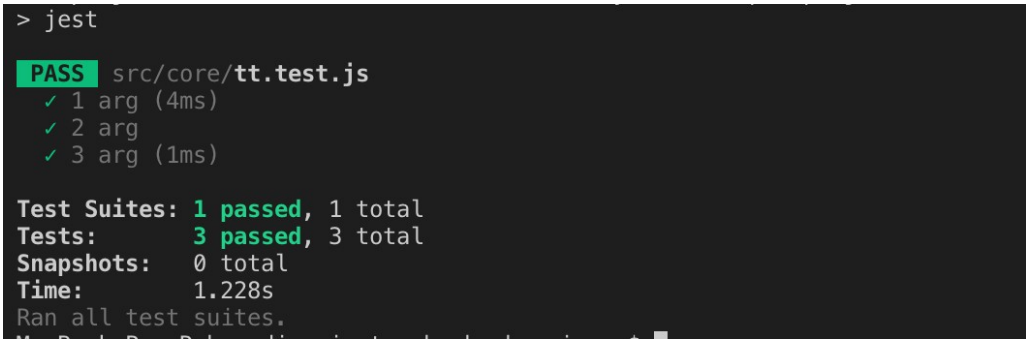


Рисунок 5. Результат тестирования

2.2.4. Анализ покрытия

Целью анализа полноты покрытия кода является выявление участков кода, которые не выполняются при выполнении тестовых примеров. Тестовые примеры, основанные на требованиях, могут не обеспечивать полного выполнения всей структуры кода. Поэтому для улучшения покрытия проводится анализ полноты покрытия кода тестами и, при необходимости, дополнительные проверки, направленные на выяснение причины недостаточного покрытия, а также определение необходимых действий по его устранению. Обычно анализ покрытия выполняется с учетом следующих соглашений.

Анализ должен подтвердить, что полнота покрытия тестами структуры кода соответствует требуемому виду покрытия и заданному минимально допустимому проценту покрытия.

Анализ полноты покрытия тестами может выявить часть исходного кода, которая не исполнялась в ходе тестирования. Для разрешения этого обстоятельства могут потребоваться дополнительные действия в процессе проверки программного обеспечения.

3. Применение инструментов тестирования

3.1. Профайлинг веб-сайта

3.1.1. Веб-тесты производительности

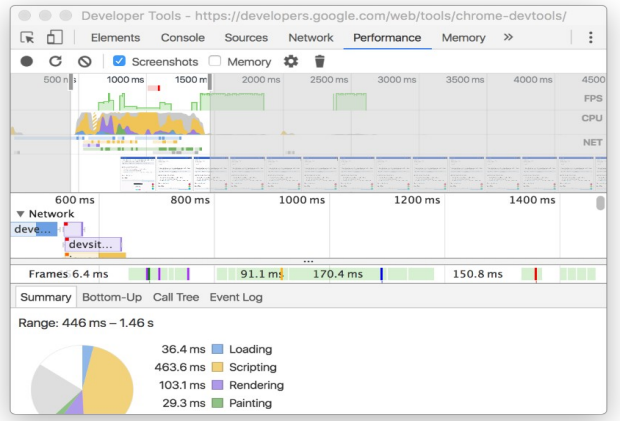
Для веб-тестов производительности предлагается использовать вкладку Performance в Chrome DevTools.

Панель отображает линию времени использования сети, выполнения JavaScript кода и уровень загрузки памяти. После первоначального построения графиков линии времени, будут доступны подробные данные о выполнение кода и всем жизненном цикле страницы. Будет возможность ознакомится с временем исполнения отдельных частей кода, появится возможность выбрать отдельный промежуток на временной шкале и ознакомится с тем какие процессы происходили в этот момент.

Инструмент применяется для улучшения производительности работы Вашей страницы в целом.

Ключевые возможности:

- Возможность сделать запись чтобы проанализировать каждое событие, которое произошло после загрузки страницы или взаимодействия с пользователем.
- Возможность просмотреть FPS, загрузку CPU и сетевые запросы в области Overview. Щелкните по событию в диаграмме, чтобы посмотреть детали об этом.
- Возможность изменить масштаб линию времени, чтобы сделать анализ проще.



Performance вкладка в DevTools

3.1.2. Панель Performance (Скорость работы)

Панель отображает линию времени использования сети, выполнения JavaScript кода и загрузки памяти. После первоначального построения графиков линии времени, будут доступны подробные данные о выполнение кода и всем жизненном цикле страницы. Будет возможно ознакомится с временем исполнения отдельных частей кода, появится возможность выбрать отдельный промежуток на временной шкале и ознакомится с тем какие процессы происходили в этот момент.

Инструмент применяется для улучшения производительности работы Вашей страницы в целом.

3.1.3. Ключевые возможности:

- Возможность сделать запись чтобы проанализировать каждое событие, которое произошло после загрузки страницы или взаимодействия с пользователем.
- Возможность просмотреть FPS (Количество кадров в секунду), загрузку CPU (Процессора) и сетевые запросы в области Overview.
- Щелкните по событию в диаграмме, чтобы посмотреть детали об этом.
- Возможность изменить масштаб временную линию, чтобы сделать анализ проще.

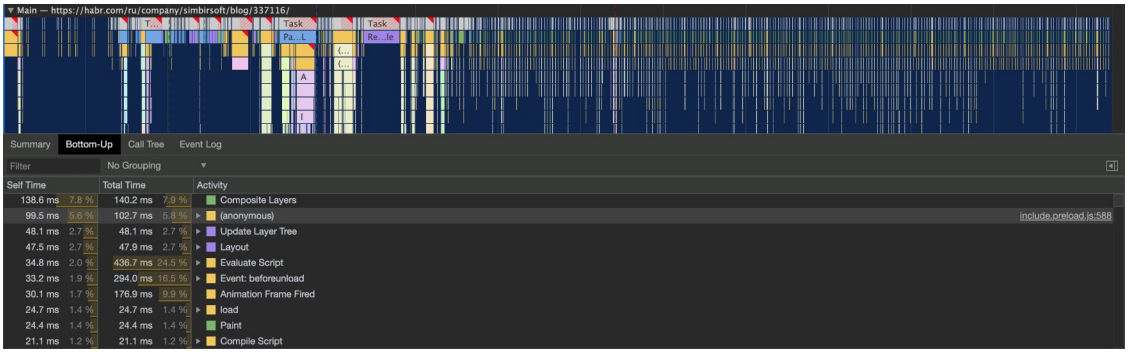


Рисунок 17. Вкладка Bottom-up

Перейдя на кладку «Bottom-up» или «Call Tree» (см. рисунок 17) можно увидеть дерево вызова функций (call tree (дерево вызовов)). Каждый из узлов можно раскрыть и увидеть более подробную информацию. Также сверху отображается графическое дерево вызовов, в котором можно легко увидеть «бутылочное горлышко» вашей системы.

Красные «треугольники» в углах узлов означают неоптимизированное время выполнение на которые и нужно обращать внимание. Во вкладке «Event log» (см. рисунок 18) можно увидеть подробное описание, всех событий, которые описаны ниже во вкладке «Summary», а именно Loading, Scipting, Rendering, Painting, System и idle (отрисовка страницы, выполнения кода Javascript, прорисовка стилей страницы, системные браузерные события и время простоя). Также можно раскрыть узлы и увидеть более подробную информацию.

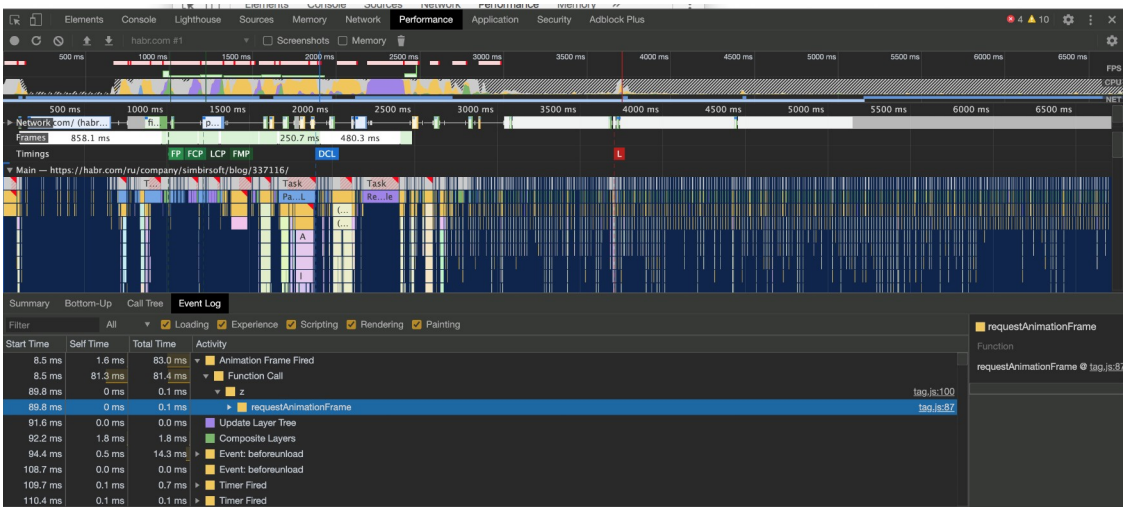


Рисунок 18. Вкладка Event log

3.1.4. Подробнее про графическое отображение

Отображение «Network» иллюстрирует работу с сетью - запрос/ответ, время запроса и размер данных. (см. рисунок 19)

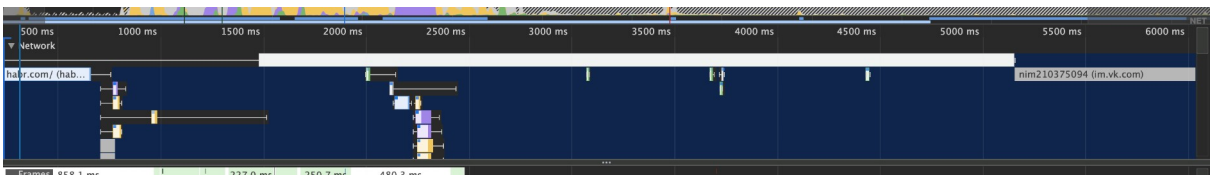


Рисунок 19. Отображение Network

Отображение «Main» иллюстрирует работу главного потока исполнения(интерпритации) Javascript. Отображение является деревом, в каждом узле которого можно увидеть более подробный список вызванных функций или проведенных операций.Также при наведении можно увидеть время выполнения конкретного узла в миллисекундах. (см. рисунок 20)

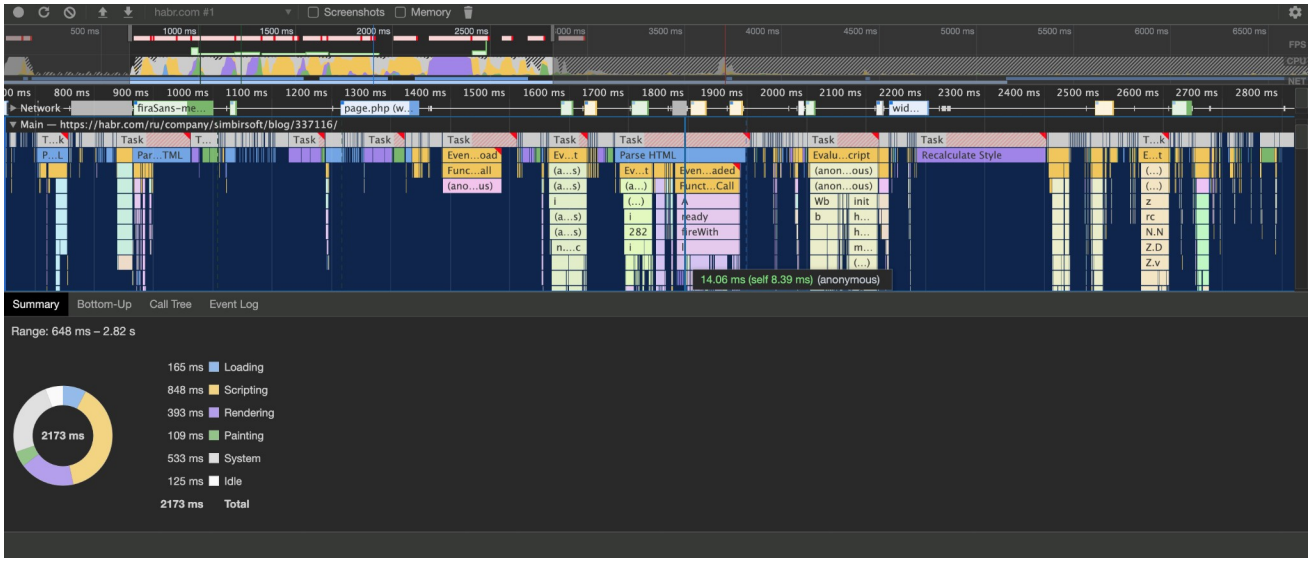


Рисунок 20. Отображение Main и общий график

Отображение «GPU» иллюстрирует загрузку графического процессора для отрисовки страницы. (см. рисунок 21)

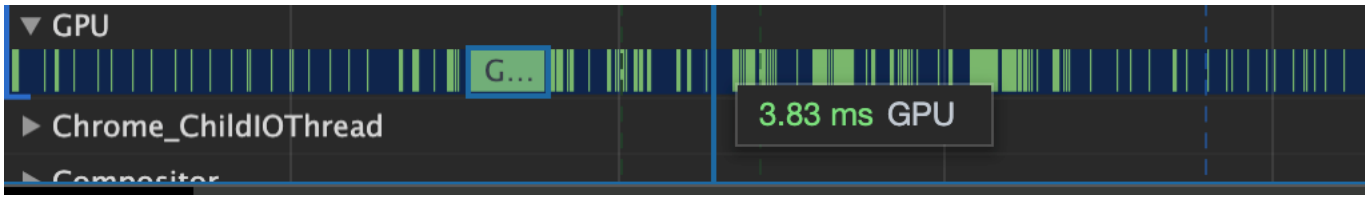


Рисунок 21. Отображение GPU

На общем графике (см. рисунок 20) можно выделить интересующий интервал, для подробного исследования показателей. Каждый цвет на графике соответствует цвету и описанию в диаграмме «Summary». В зависимости от цветовой темы браузера они могут отличаться. Например, на рисунке выше видно, что желтая линия на общем графике соответствует метрике Scripting (время выполнение Javascript).

3.1.5. Запуск тестов

Откроем необходимый сайт. Откроем DevTools а именно инструменты разработчика в вашем браузере. Сверху обнаружите вкладку Preformance и кнопку Record Button. Нажмите на эту кнопку и запустите тестирование. (см. рисунок 22)

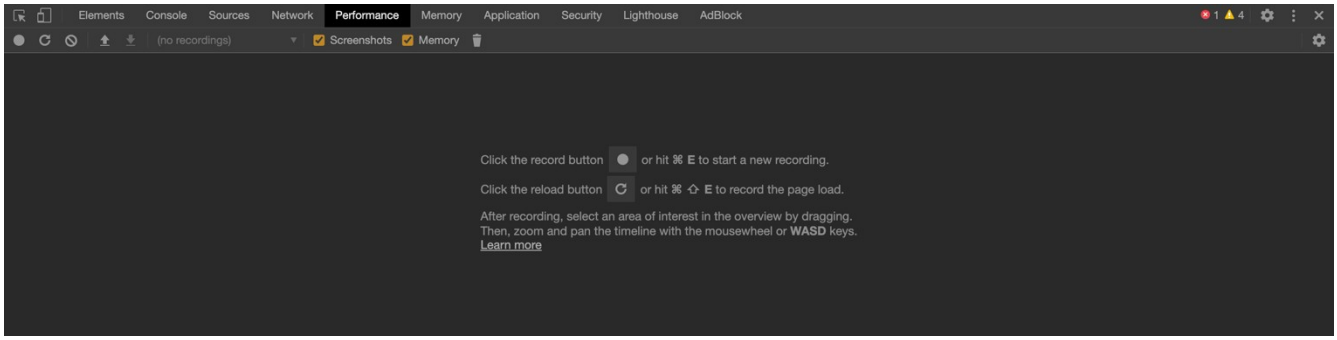


Рисунок 22. Запуск тестирования

Результаты тестирования будут представлены в виде графиков. (см. рисунок 23)

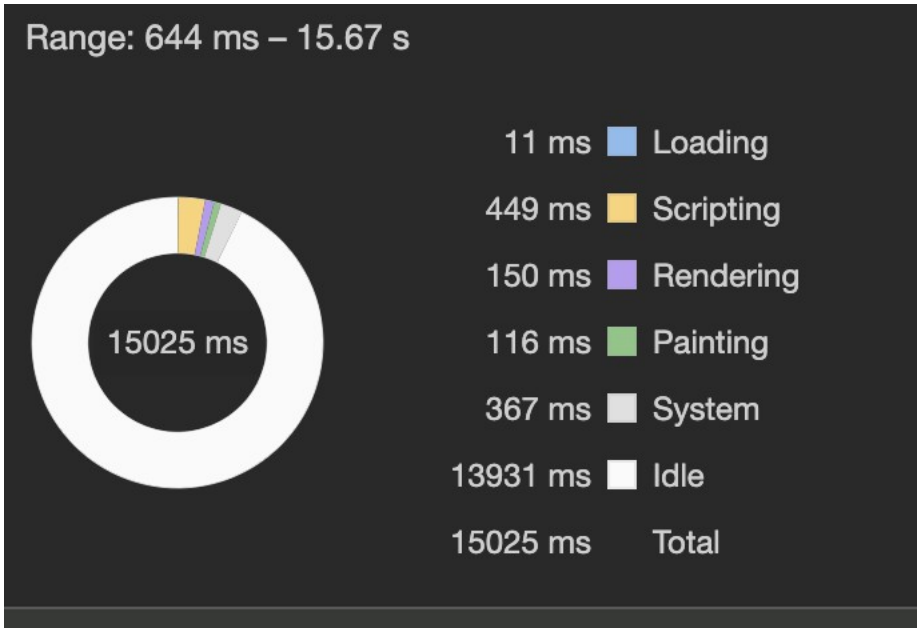


Рисунок 23. Результаты тестирования

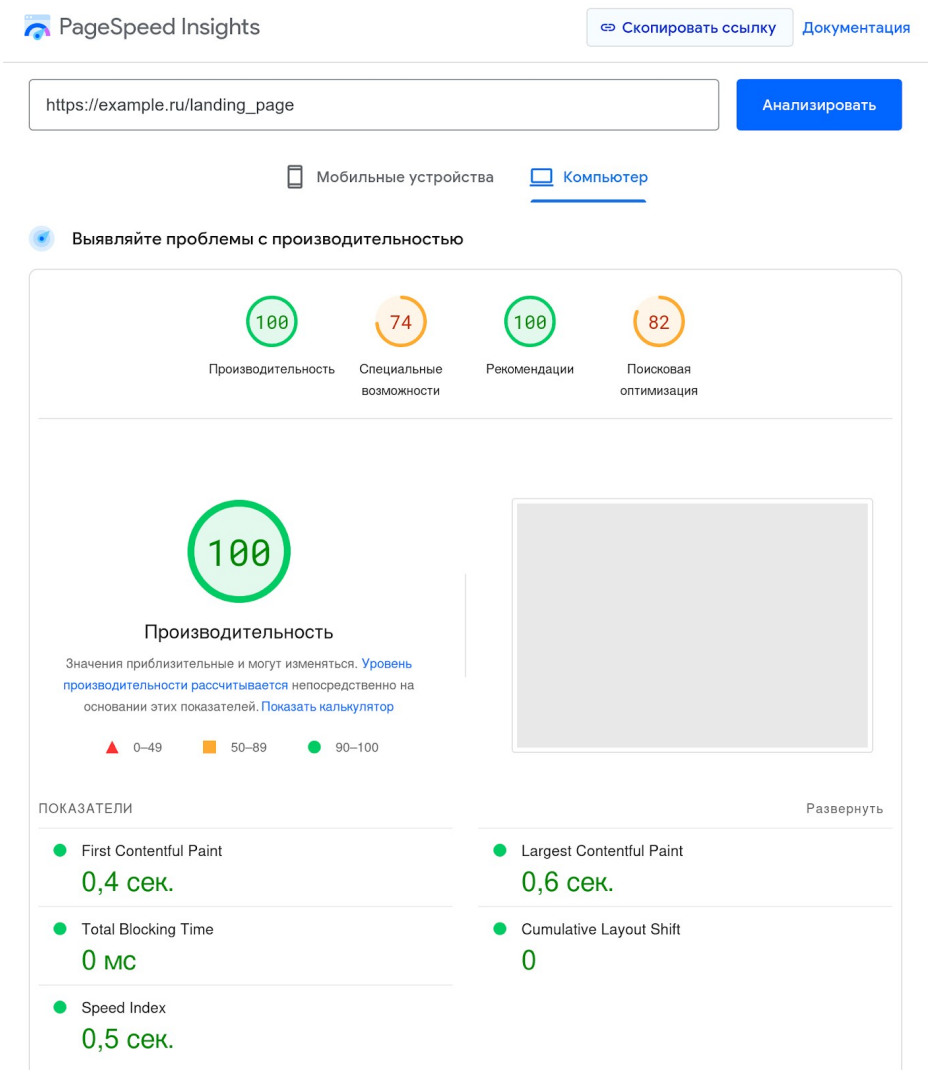
Рассмотрим полученные результаты.

- Загрузка (Loading) страницы заняла 11 мс
- Выполнение и загрузка скрипта (Scripting) страницы заняло 449 мс
- Рендеринг, а именно отрисовка страницы (Rendering) страницы занял 150 мс
- Погрузка css - стилей страницы (Painting) страницы занял 116 мс
- Погрузка системных браузерных настроек (System) страницы занял 367 мс
- Время простоя системы после выполнения всех процессов до окончания нагрузочного тестирования приложения (Idle) страницы занял 13931 мс

Исходя из результатов, полученных нами, мы можем судить, что приложение довольно быстрое.

3.1.6. Сервис PageSpeed Instights

Кроме «инструментов разработчика», для оценки параметров сайта можно использовать онлайн-сервисы, такие как PageSpeed Insights (PSI) [7]:



Сервис [8] проводит аудит скорости загрузки веб-страниц и даёт рекомендации по ускорению и оптимизации. Главный показатель в отчёте, предоставляемом PSI, — производительность в баллах. Результат от 90 и выше считается хорошим, от 50 до 90 — средним, а если ваша страница набрала менее 50 баллов, рекомендуется срочно заняться поиском узких мест в архитектуре сайта.

Рассмотрим другие важные показатели скорости загрузки:

- First Contentful Paint (FCP, первая отрисовка контента): время, необходимое браузеру для отображения первой части контента DOM (Document Object Model, объектная модель документа) после того, как пользователь перешел на страницу.

- Total Blocking Time (TBT, общее время блокировки): время после первой отрисовки контента (FCP), на которое был заблокирован основной поток. Блокировка может быть вызвана длинными (более 50 мс) задачами. Негативно влияет на использование страницы.
- Speed Index (индекс скорости загрузки). Показывает, насколько быстро контент отображается визуально во время загрузки страницы.
- Largest Contentful Paint (LCP, время отрисовки самого большого элемента). Таким элементом может быть текст, изображение или видео. Учитывается только видимая пользователю область элемента.
- Cumulative Layout Shift (CLS, совокупный сдвиг макета). Позволяет оценить общую сумму всех сдвигов вёрстки за всё время использования страницы. Сдвиги макета обычно происходят из-за асинхронной загрузки ресурсов или динамического добавления видимых элементов. Такие сдвиги происходят неожиданно для пользователя и значительно снижают удобство использования страницы.

Как видно из результатов аудита, скорость загрузки подопытной страницы находится в «зелёной» зоне и не требует оптимизации.

Кроме измерения производительности, инструмент PSI даст вам несколько подсказок по оптимизации вашего сайта:

- Специальные возможности. Разметка HTML позволяет браузерам и программам чтения с экрана понять, как организована структура страницы. В нашем примере на исследуемом сайте определённо стоит уделить больше внимания семантической вёрстке.
- Рекомендации. В нашем примере у PSI нет предложений по улучшению скорости загрузки страницы.
- Поисковая оптимизация (Search Engine Optimization, SEO). Включает в себя целый комплекс внутренних и внешних мер по улучшению позиции сайта в поисковой выдаче с целью увеличения целевого интернет-трафика. Важно отметить, что скорость загрузки сайта также влияет на показатель SEO.

Рекомендуем периодически проверять скорость работы и отзывчивость вашего сайта, особенно на этапе выбора библиотек и принятия архитектурных решений, чтобы

избежать финансовых потерь в будущем. Такой подход получил название «Разработка, ориентированная на производительность» (Performance-Driven Development). Задумайтесь о том, что вашим клиентам совершенно неважно, насколько модные библиотеки вы используете, пользователям нужны удобство и качественное содержание. В конце концов, интернет ведь создан для людей, не так ли? И, конечно, не стоит забывать, что время отклика и скорость загрузки сайта влияет на его позицию в поисковой выдаче.

3.2. Профайлинг локального приложения

3.2.1. Что такое профилировщик

Профилировщик [12] (profiler) — это инструмент, запускающий код и собирающий информацию о времени, необходимом для вызова каждой функции, количестве вызовов и иерархии вызовов функций.

Проанализировав результаты работы инструмента, вы сможете понять, исполнение какой части кода требует больше всего времени (это называется «узким местом» или «бутылочным горлышком», bottleneck), а также понять, как решить эту проблему. Вам нужно находить и устранять узкие места — это сильно повышает скорость программ.

3.2.1.1. Пример проблемы

Допустим, у нас есть большой текстовый файл, и нам нужно найти в нём множество вхождений по определённому паттерну. Сначала давайте сгенерируем большой файл со случайными алфавитно-цифровыми символами (построчно):

```
import random
import string

def generate_random_string(length):
    """Generate a random string of lowercase letters and digits."""
    letters_and_digits = string.ascii_letters + string.digits
    return ''.join(random.choice(letters_and_digits) for _ in
range(length))

def generate_random_file(filename, num_lines, line_length):
    """Generate a file of random lines with the given number and
length."""
    with open(filename, 'w') as f:
        for i in range(num_lines):
            random_line = generate_random_string(line_length) + '\n'
            f.write(random_line)
```

```
if __name__ == '__main__':
    # Генерируем 1000000 строк по 1000 символов каждая
    generate_random_file('random.txt', 1_000_000, 1000)
```

Теперь давайте зададим нашу первоначальную функцию, от которой будем отталкиваться — она считывает файл строка за строкой, а затем считает количество вхождений в них слов bob и rob, которым предшествует цифра:

```
import re

def baseline():
    num_total_matches = 0
    pattern1 = r"[0-9]{1}bob"
    pattern2 = r"[0-9]{1}rob"

    with open('random.txt', 'rt') as f:
        for line in f:
            line = line.lower()  # преобразуем весь текст в нижний регистр
            for pattern in [pattern1, pattern2]:
                # Находим все вхождения паттерна в строке
                matches = re.findall(pattern, line)

                # Считаем количество совпадений
                num_matches = len(matches)
                num_total_matches += num_matches

    return num_total_matches
```

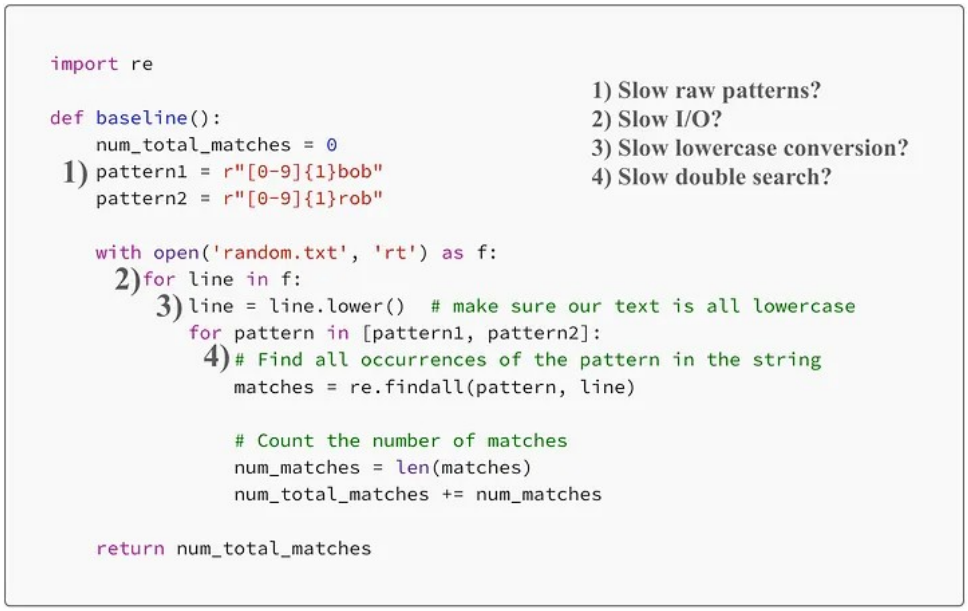
Например, в строке abc1robdef02bob есть два вхождения.

Мы выполняем функцию baseline, вычисляем количество вхождений (в данном случае 10861) и замеряем время исполнения —получилось 32 с. Как ускорить работу кода?

3.2.1.2. Потенциальные цели для совершенствования

Код потенциально могут замедлять четыре части. Вот они:

- Использование сырых строк вместо скомпилированных объектов регулярных выражений.
- Использование двух отдельных операций поиска вместо одного объединённого регулярного выражения.
- Преобразование каждой строки в нижний регистр вместо использования флага нечувствительности к регистру.
- Считывание файла построчно, а не крупными блоками.



Причины замедления программы:

1) Медленные сырые паттерны

2) Медленный ввод-вывод

3) Медленное преобразование в нижний регистр

4) Медленный двойной поиск

Как узнать, что из этого оказалось узким местом? Нужно использовать профилировщик.

3.2.2. Профилировщики Python

В Python есть *два встроенных модуля профилирования* — cProfile и profile. Они выполняют одну задачу, но cProfile написан на C, а profile — на чистом PythonТакже можно воспользоваться несколькими внешними инструментами, это иногда удобнее и проще.

Допустим, необходим быстрый и удобный способ профилирования части кода (например, функции) и сохранения результата в файл. Модуль под названием profilehooks предоставляет *простой декоратор*, которым мы можем обернуть интересующую нас функцию:

```
from profilehooks import profile

# stdout=False -> не выводить ничего в терминал
# filename -> путь к файлу с результатами профилирования
@profile(stdout=False, filename='baseline.prof')
```

```
def baseline():
    ...
```

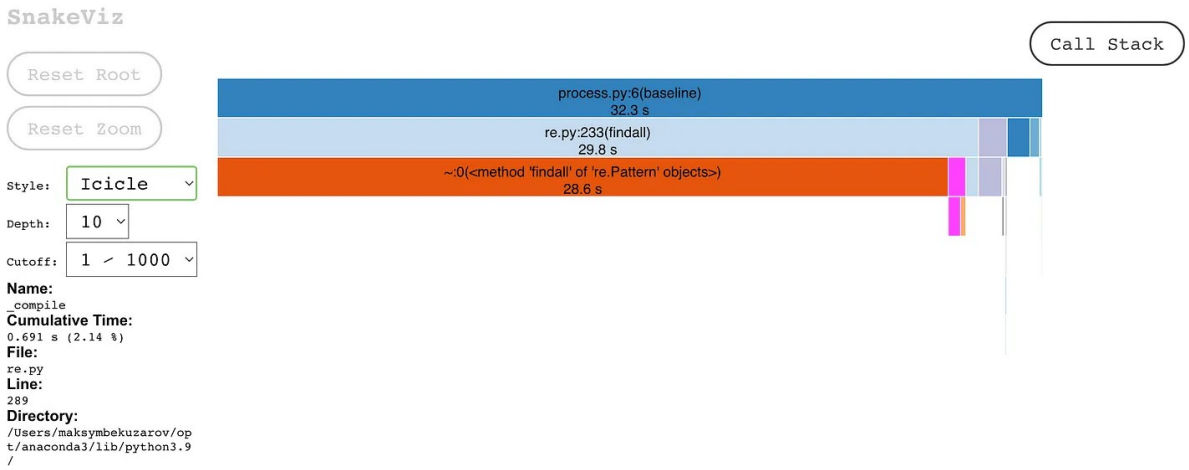
Его можно установить простой командой pip install profilehooks.

И можно визуализировать этот файл удобным для человека образом. Для этого удобно использовать два инструмента.

3.2.2.1. Инструмент SnakeViz

Snakeviz — это работающий в браузере интерактивный визуализатор результатов профилирования Python. Его легко установить (pip install snakeviz) и использовать (snakeviz <path-to-profiling-output>). Давайте взглянем на результаты профилирования нашей функции baseline.

Интерактивные результаты Snakeviz — графики icicle (слева) и sunburst (правый, менее читаемый). Можно наводить мышью и щёлкать на любую функцию, чтобы просматривать подробности. Сверху вниз идёт вложенная иерархия вызовов, а длина линии — это относительное время, потраченное кодом на их исполнение.



Кроме того, Snakeviz демонстрирует интерактивную таблицу со статистикой времени исполнения функций.

Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2000000	28.63	1.432e-05	28.63	1.432e-05	~:0(<method 'findall' of 're.Pattern' objects>)
1000001	0.9303	9.303e-07	1.128	1.128e-06	std.py:1174(__iter__)
1	0.8952	0.8952	32.35	32.35	process.py:6(baseline)
2000000	0.4892	2.446e-07	29.81	1.491e-05	re.py:233(findall)
2000001	0.481	2.405e-07	0.6907	3.453e-07	re.py:289(_compile)
1000024	0.3697	3.696e-07	0.3697	3.696e-07	~:0(<method 'lower' of 'str' objects>)
2002132	0.2092	1.045e-07	0.2092	1.045e-07	~:0(<built-in method builtins.isinstance>)
2001260/2001234	0.08624	4.309e-08	0.08625	4.31e-08	~:0(<built-in method builtins.len>)
122194	0.07442	6.09e-07	0.07442	6.09e-07	~:0(<built-in method _codecs.utf_8_decode>)
122194	0.05082	4.159e-07	0.1252	1.025e-06	codecs.py:319(decode)
15	0.02959	0.001973	0.02959	0.001973	~:0(<built-in method _imp.create_dynamic>)
94266	0.01365	1.448e-07	0.02071	2.197e-07	utils.py:330(<genexpr>)

Reload this page

Мы видим, что больше всего времени исполнения тратится внутри функции findall, то есть на выполнение поиска регулярными выражениями. Значит, если мы хотим ускорить код, то нужно сосредоточиться на ускорении этой функции, так как узким местом является она, а не другие части кода.

Давайте проверим результаты во втором инструменте.

3.2.2.2. Инструмент gprof2dot

Gprof2dot предоставляет более читаемую визуализацию в виде блок-схемы, выполняет сохранение в файлы изображений, поэтому в нём легко делиться результатами (а по необходимости и автоматизировать это). Однако он не интерактивен и требует установки в систему Graphviz.

Для установки gprof2dot достаточно выполнить команду pip install gprof2dot.

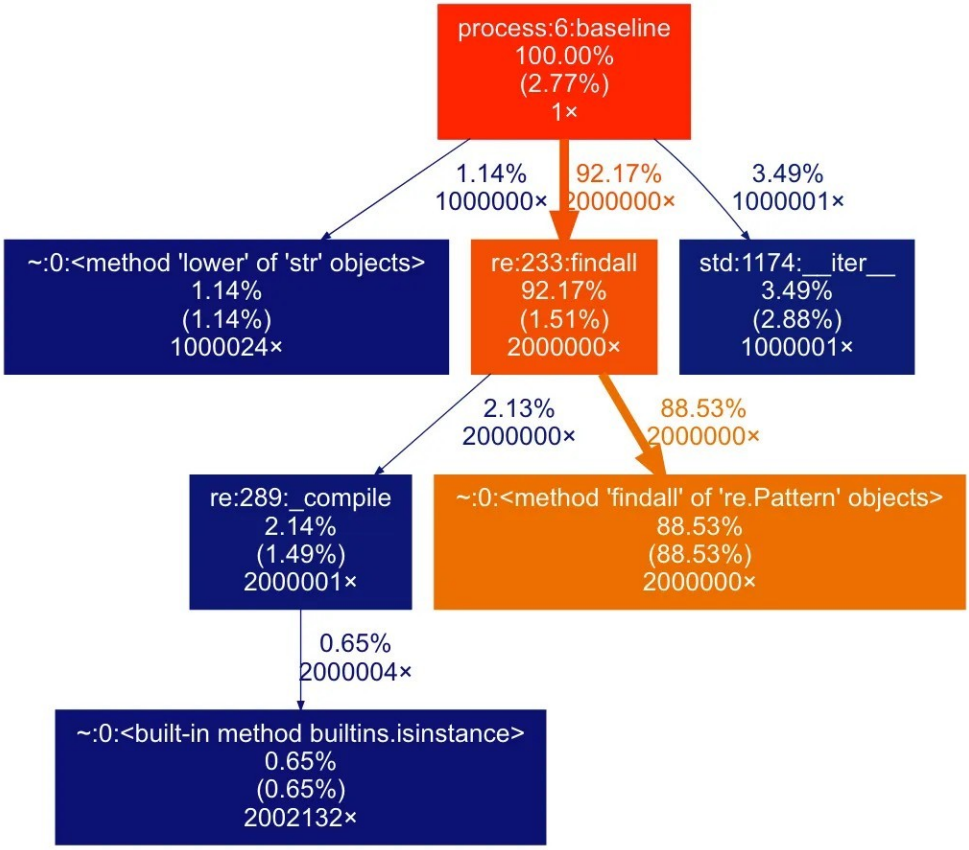
Для генерации выходного изображения с результатом профилирования нужно использовать следующую команду:

```
python -m gprof2dot -f pstats <profiling-results-file> | dot -Tpng -o
output.png
```

Сначала мы представляем иерархию вызовов функций в виде графа в формате dot, а затем генерируем изображение — визуализацию этого графа. Команда dot поддерживает различные форматы вывода, в том числе .jpg и .svg, а результаты gprof2dot также можно гибко настраивать.

Давайте посмотрим, как они выглядят. Ниже представлена иерархия исполнения функций нашего кода в виде графа. В процентах указана общая доля времени исполнения, проведённая внутри функции, а также (%), но сюда включён только

собственный код функции. Последнее значение — это количество вызовов функции в коде.



Теперь картина выглядит гораздо более читаемой, граф показывает нам, что поиск регулярными выражениями занимает 88% от общего времени исполнения, поэтому нам нужно сделать его быстрее.

Если у вас нет доступа к Graphviz (команда dot), то можно использовать *соответствующий пакет Python для Graphviz* (pip install graphviz), а также простой скрипт на Python для генерации результатов.

Сохраним результаты gprof2dot в файл .dot:

```
python -m gprof2dot -f pstats file.prof > file.dot
```

Сгенерируем изображение из этого файла .dot при помощи следующего кода:

```
import graphviz

def make_png(input_file_name, output_file_name):
    dot = graphviz.Source.from_file(input_file_name)
    dot.render(outfile=output_file_name)

if __name__ == '__main__':
```

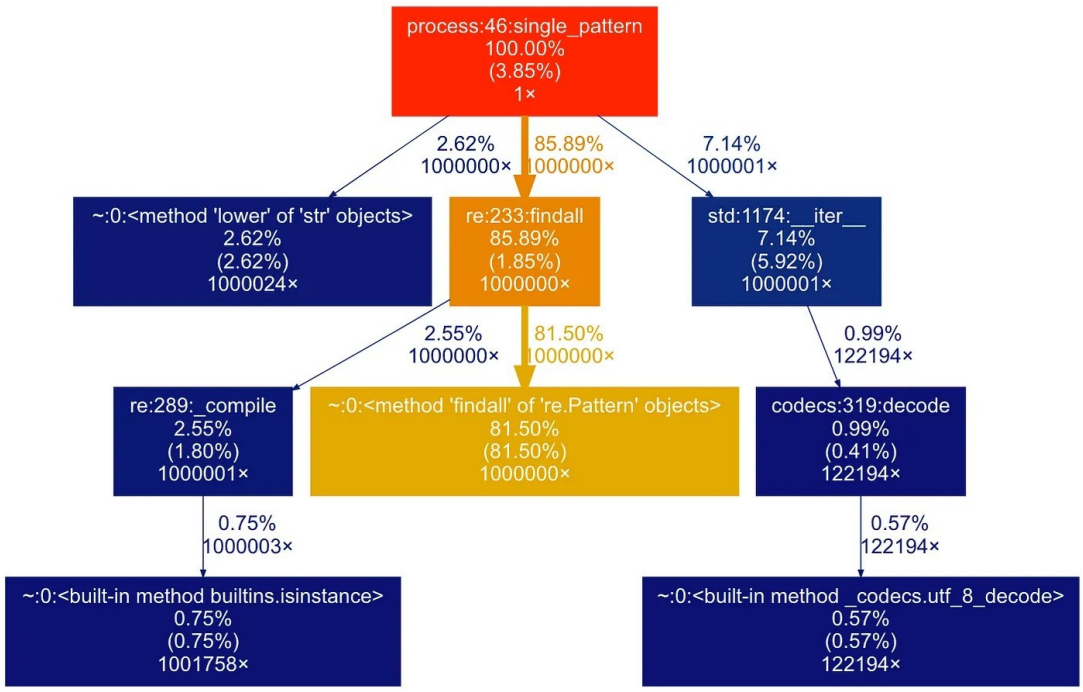
```
    make_png('file.dot', 'file.svg')  # также поддерживает .png, .jpg и
так далее.
```

3.2.3. Улучшения кода

Давайте ускорим код, скомбинировав два регулярных выражения в одно:

```
def single_pattern():
    num_total_matches = 0
    pattern = r"[0-9][rb]ob"  # Теперь у нас только одна операция поиска
вместо двух

    with open('random.txt', 'rt') as f:
        for line in tqdm(f, total=1_000_000):
            line = line.lower()
            # Находим все вхождения паттерна в строке
            matches = re.findall(pattern, line)

            # Считаем количество совпадений
            num_matches = len(matches)
            num_total_matches += num_matches

    return num_total_matches
```

Запустим и получим 15 с — скорость повысилась примерно в два раза! Обратите внимание, что % занимаемого findall времени снизился (88% -> 81.5%).



Профилирование нового кода подтвердило, что был сделалан правильный выбор — доля времени, занимаемого основной функцией findall снизилась, а для всех других функций увеличилась (тоже приблизительно в два раза).

Если мы реализуем все три остальные идеи, а не сосредоточимся на узком месте, то время исполнения всё равно останется равным 32 с, как и в изначальном коде! Мы бы потратили столько времени и усилий, не добившись никаких улучшений! Именно поэтому важно сосредоточиться на узких местах.

```
def wasted_efforts():
    num_total_matches = 0
    # Используем скомпилированные регулярные выражения
    pattern1 = re.compile(r"[0-9]{1}rob", flags=re.IGNORECASE)
    # Также используем нечувствительность к регистру вместо изменения
строк
    pattern2 = re.compile(r"[0-9]{1}bob", flags=re.IGNORECASE)

    with open('random.txt', 'rt') as f:
        # Загружаем строки в блок и обрабатываем их за один проход
        chunk = []
        for line in tqdm(f, total=1_000_000):
            chunk.append(line)

            if len(chunk) == 1000:
                chunk_str = ''.join(chunk)
                for pattern in [pattern1, pattern2]:
                    # Находим все вхождения паттерна в строке
                    matches = re.findall(pattern, chunk_str)

                    # Считаем количество совпадений
                    num_matches = len(matches)
                    num_total_matches += num_matches

                chunk = []

    # Но несмотря на наши усилия, код всё равно такой же медленный, как и
оригинальный
    return num_total_matches
```

3.2.3.1. Multiprocessing

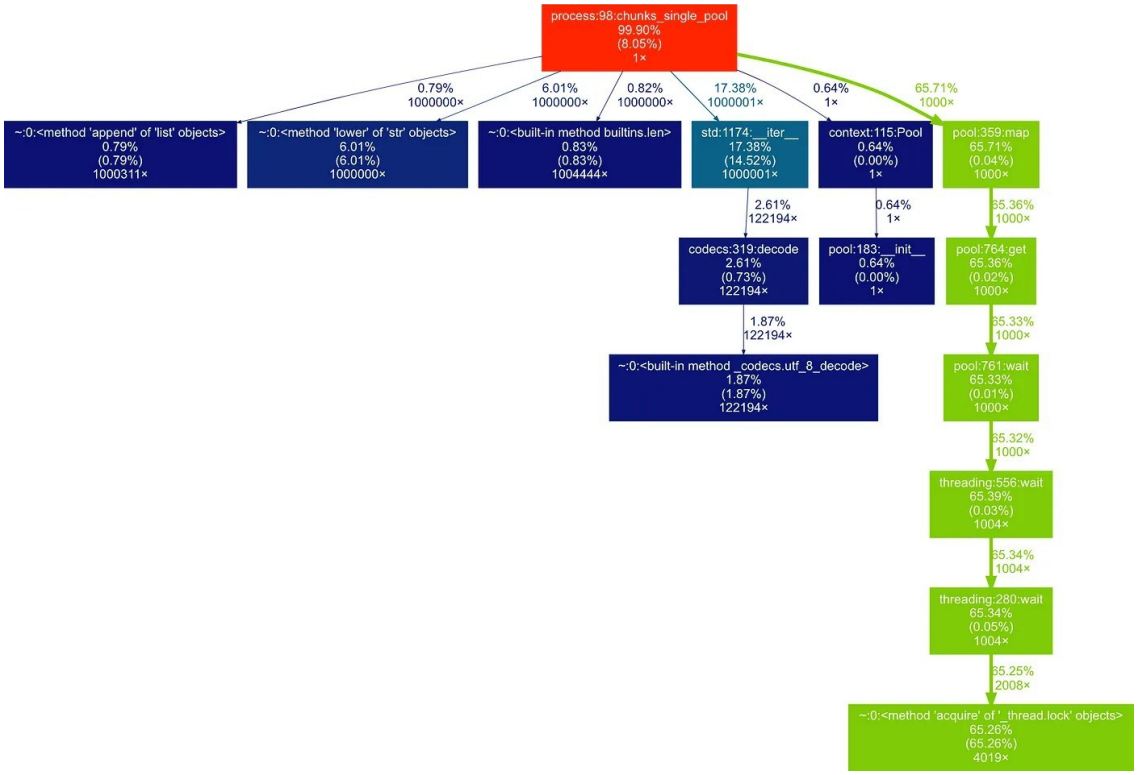
Функция findall по-прежнему остаётся узким местом. Однако в нашем придуманном примере для её дальнейшего улучшения можно сделать только одну простую вещь: распараллелить код.

Совпадения в разных строках независимы друг от друга, поэтому мы можем выполнять поиск в них параллельно. Как нам это сделать?

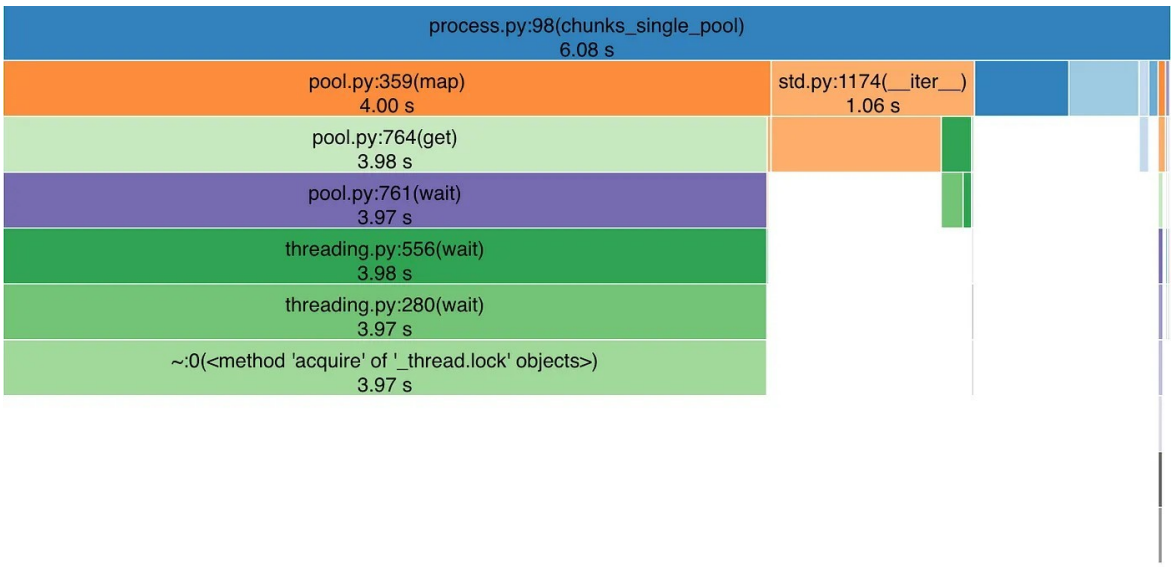
Проще всего создать *Process Pool* — объект, управляющий множеством параллельных процессов Python, работающих одновременно. Если у нас есть список значений и функция, применяемая к каждому из них, то мы просто можем вызывать метод пула map, и он будет исполнять функцию параллельно для всех значений.

```
from multiprocessing import Pool


def calc_num_matches(string):
    # Создаём отдельную функцию для объекта Pool
    # Она будет параллельно применяться к каждой строки в файле
    pattern = r"[0-9]{1}[rb]ob"
    return len(re.findall(pattern, string))

def chunks_single_pool():
    num_total_matches = 0

    pool = Pool(8)  # количество процессов должно быть <= количества ядер
вашего CPU

    with open('random.txt', 'rt') as f:
        chunk = []
        # Загружаем строки в список
        for line in tqdm(f, total=1_000_000):
            line = line.lower()
            chunk.append(line)

            if len(chunk) == 1000:
                # Затем мы параллельно применяем к каждой строке функцию
`calc_num_matches`
                num_ind_matches = pool.map(calc_num_matches, chunk)
                # И складываем все независимые совпадения
                num_matches = sum(num_ind_matches)
                num_total_matches += num_matches

                chunk = []

    return num_total_matches
```

Посмотрим… 6,1 с! Это новый рекорд. В 2,2 раза быстрее, чем оптимизированное решение, и примерно в 5 раз быстрее, чем изначальный код! Давайте посмотрим на результаты профилирования. Обратите внимание, что доля regex-поиска продолжает уменьшаться (зелёные блоки).



Snakeviz демонстрирует при многопроцессорной обработке ту же картину:



Оба результата подтверждают, что regex-поиск теперь занимает примерно 2/3 от общего времени — гораздо меньше, чем в исходной версии.

3.2.4. Особенности

3.2.4.1. Multiprocessing

Важная особенность заключается в том, что профилировщик больше не знает, что происходит в подпроцессах, выполняющих regex-поиск. Он просто показывает, что

на их выполнение потрачено 4 секунды времени, но он не имеет доступа к исполняемому в подпроцессах коду, потому что они, по сути, являются отдельными программами, слабо связанными с нашим основным процессом Python.

Если вы хотите профилировать то, что происходит в подпроцессе вашей программы, то необходимо поместить внутрь исполняемой подпроцессом функции/кода декоратор @profile.

3.2.4.2. Multithreading

Обычно не нужно писать многопоточный код непосредственно на Python, потому что его применение ограничено *Global Interpreter Lock (GIL)*, который за раз позволяет выполнять только один поток кода на Python (в одном процессе Python).

Однако если вы работаете со многими библиотеками не для Python (например, NumPy, PyTorch, scipy) или с активным вводом-выводом (веб-коммуникации), то основная часть времени будет тратиться за пределами интерпретатора Python на исполнение кода, написанного на C, C++, Fortran и так далее.

В этих двух случаях использование нескольких потоков может быть практичным, поэтому нужно иметь в виду, что оба встроенных модуля профилирования Python — *profile и cProfile* — профилируют только основной поток приложения. Если вы хотите профилировать код, исполняемые в других потоках, то можно попробовать запустить профилировщик в функции, которую будут исполнять потоки, или воспользоваться сторонними инструментами профилирования наподобие Yappi или VizTracer.

3.2.4.3. Вычисления в GPU

Если вы используете для вычислений GPU, то учтите, что это отдельное устройство в системе, работающее асинхронно с CPU. Поэтому при исполнении кода в GPU нужно быть очень внимательным с измерением времени исполнения, ведь ваш код на Python не знает, что происходит в GPU. Он может только приказать сделать что-то и ждать, пока GPU завершит выполнение этой задачи.

Давайте рассмотрим следующий код:

```
import torch
from profilehooks import profile

@profile(filename='gpu_test.prof', stdout=False)
```

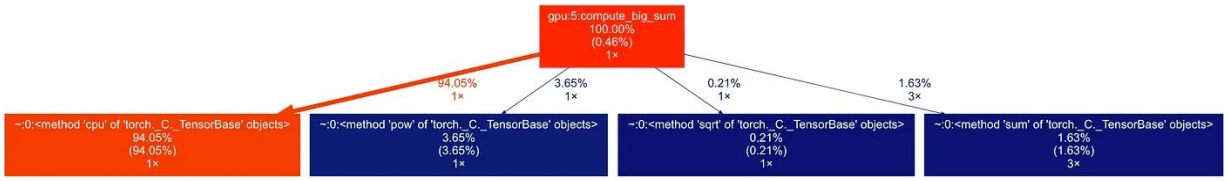
```
def compute_big_sum(tensor: torch.Tensor):
    # выполняем много затратных вычислений, результатом которых становится
одно число
    a = tensor.sum()
    b = tensor.pow(2).sum()
    c = tensor.sqrt().sum()

    sum_all = a + b + c
    sum_all_cpu = sum_all.cpu()  # передаём значение в память CPU

    # возвращаем сумму всех операций (скалярную)
    return sum_all_cpu

if __name__ == '__main__':
    # создаём большую матрицу случайных чисел, уже находящуюся в памяти
GPU
    X = torch.rand((10000, 10000), dtype=torch.float64, device='cuda:0')
    res = compute_big_sum(X)

    print(float(res))
```

Мы создаём большую матрицу 10000x10000 на PyTorch (в GPU), а затем профилируем функцию, вычисляющую следующее:

- Сумму всех элементов.
- Сумму квадратов всех элементов.
- Сумму квадратных корней всех элементов.
- Сумму всех предыдущих сумм.

На выходе эта функция возвращает одно число.

Как вы думаете, какая строка кода займёт больше всего времени? Результаты профилирования функции compute_big_sum:



Как ни удивительно, это метод .cpu(), передающий данные из памяти GPU в память CPU! Но это не имеет никакого смысла. Мы передаём всего 8 байтов данных! Неужели передача в памяти настолько медленная?

Нет.

Так как внутри PyTorch вычисляются *CUDA* в GPU, который является отдельным от CPU устройством, то при использовании функции

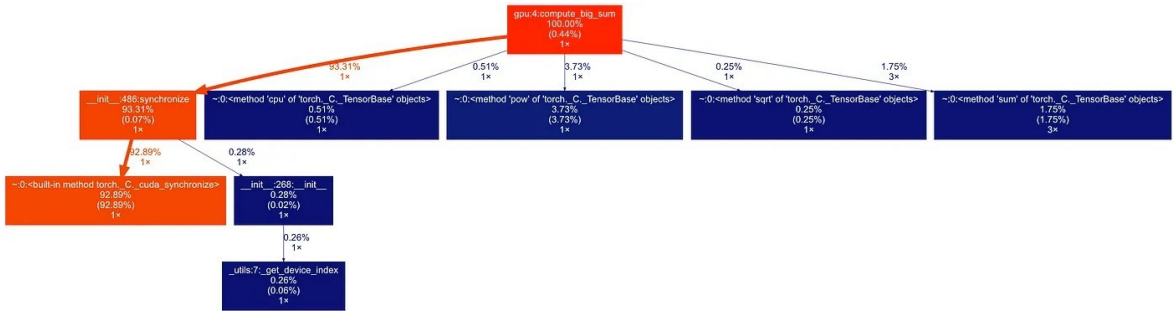
наподобие .sum() или torch.pow(). Python просто приказывает GPU начать вычисления, но не ждёт, пока они закончатся! То есть интерпретатор Python запускает вычисления и сразу же переходит к следующей строке кода.

Ожидание происходит в методе .cpu(), передающем данные из памяти GPU в память CPU. Он вынужден ждать, пока результаты всех предыдущих вычислений окажутся доступными в памяти GPU, то есть завершения исполнения всех функций, которые мы применили к нашей большой матрице.

Но как нам это профилировать?

- Использовать *встроенный профилировщик PyTorch*, имеющий поддержку профилирования операций нескольких устройств.
- Если вас не интересует, сколько времени занимает каждая отдельная операция в GPU (или она только одна), то можно добавить в код сразу после начала всех вычислений в GPU torch.cuda.synchronize(), что заставит PyTorch ждать, пока завершатся все операции в GPU.

Давайте посмотрим, как результаты профилирования будут выглядеть при использовании torch.cuda.synchronize(). Теперь .cpu() занимает меньше 1% общего времени, а почти всё время исполнения тратится на ожидание завершения вычислений в GPU, как и ожидалось.



Код нашей изменённой функции compute_big_sum будет выглядеть так:

```
def compute_big_sum(tensor: torch.Tensor):
    a = tensor.sum()
    b = tensor.pow(2).sum()
    c = tensor.sqrt().sum()

    sum_all = a + b + c
    torch.cuda.synchronize()  # добавляем это после всех операций с GPU
                              # чтобы обеспечить корректность измерений
времени
    sum_all_cpu = sum_all.cpu()
```

```
    return sum_all_cpu
```

3.2.5. Другие инструменты профайлинга

Профилирование [16] приложений поддерживает большинство современных средств разработки, таких как Visual Studio [17], VS Code [18], PyCharm [19] и другие.

3.3. Нагрузочное тестирование

3.3.1. Применение инструмента wrk

Для проведения нагрузочного тестирования API предлагается использовать wrk [4].

Нагрузочное тестирование API выполняется с помощью команды

```
wrk -t12 -c400 -d30s http://127.0.0.1:8080/index.html
```

wrk - это современный инструмент нагрузочного тестирования HTTP, способный генерировать значительную нагрузку при работе на одном многоядерном процессоре.

Опишем команды:

- t - количество используемых потоков для проведения теста
- c – количество соединений, используемых для проведения теста. Количество соединений по протоколу HTTP к вашему API
- d – время нагрузочного тестирования.

Нагрузочное тестирование (англ. load testing) — вид тестирования производительности, сбор показателей и определение производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

Для исследования времени отклика системы на высоких или пиковых нагрузках производится стресс-тестирование, при котором создаваемая на систему нагрузка превышает нормальные сценарии её использования. Не существует чёткой границы между нагрузочным и стресс-тестированием, однако эти понятия не стоит смешивать, так как эти виды тестирования отвечают на разные бизнес-вопросы и используют различную методологию.

```
Running 30s test @ https://velox-server-usa.herokuapp.com/getuser
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   216.14ms  142.53ms   1.60s    94.81%
    Req/Sec   112.96     71.76   320.00     57.87%
  32101 requests in 30.10s, 9.58MB read
  Socket errors: connect 167, read 0, write 0, timeout 4
  Non-2xx or 3xx responses: 32101
Requests/sec:   1066.47
Transfer/sec:    325.98KB
```

Рисунок 16. Проведение нагрузочного тестирования с помощью wrk

Мы провели нагрузочное тестирование (см. рисунок 16) с данными параметрами wrk -t12 -c400 -d30s и ниже описаны наши результаты.

Мы видим, что AVG Latency (средняя задержка ответа) у нас составляет 216.14 мс. Среднее количество запросов в секунду 112.96 (AVG Req/Sec). Максимальная задержка ответа составляет 1.6 сек (Max Latency), а Количество запросов 320 запросов/сек (Max Req/sec). По данным результатам можно сделать вывод о том, что наша система пригодна для довольно большого количества пользователей, и отвечает довольно быстро.

Показатель Запросов в секунду = 1066.47 (Requests/sec)

В течении теста было проведено 32101 (Non-2xx or 3xx responses) запрос в течении 30.1 сек (Requests in). И прочитано 9.58 Мбайт данных.

Среднее количество переданных данных в секунду = 325.98Кбайт (Transfer/sec)

3.3.2. Применение инструмента Locust

Согласно официальной документации [9], Locust — это простой в использовании распределенный инструмент, предназначенный для нагрузочного тестирования веб-сайтов (или других систем) и определения количества пользователей, с которыми одновременно может работать система.

Идея заключается в том, что во время тестирования сайт подвергается «атаке стаи саранчи» (от англ. locust — саранча). Поведение каждой саранчи (тестового пользователя) определяется вами, а процесс атаки отслеживается с помощью веб-интерфейса в режиме реального времени. Это помогает провести боевые испытания и выявить узкие места в коде, прежде чем допускать реальных пользователей.

Locust полностью основан на событиях, что позволяет поддерживать одновременно тысячи пользователей на одной машине. В отличие от многих других событийно-ориентированных приложений, в нем не используются callback-функции. Вместо этого он использует легковесные процессы через gevent. Каждая саранча,

роящаяся на вашем сайте, фактически работает внутри собственного процесса (или гринлета, если быть точным). Это позволяет писать очень выразительные сценарии на Python, не усложняя код callback-функциями.

3.3.2.1. Установка

Установка [10] довольно проста, поскольку Locust поддерживается на 3.6, 3.7 и 3.8 (начиная с версии 0.14). Для установки достаточно выполнить соответствующую команду pip в зависимости от версии Python. Сначала настоятельно рекомендуется настроить виртуальную среду.

Обратите внимание, что locust был переименован из locustio в просто locust.

Python 3:

```
pip install locust
```

Последняя master ветка из git:

```
pip install -e git://github.com/locustio/locust.git@master#egg=locust
```

Приведенные выше шаги должны работать в Linux и Windows. Если что-то не работает, попробуйте следующий способ.

Дополнительный фикс для Windows

Во-первых, загрузите предварительно собранные бинарные пакеты для:

- pyzmq
- gevent
- greenlet

Для установки файла wheel (whl) выполните команду:

```
pip install name-of-the-package.whl
```

После этого просто запустите команду pip install.

```
pip install locust
```

MacOS

Для MacOS перед установкой pip нужно установить gevent. Самый простой способ — установить Homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Затем установите libev:

```
brew install libev
```

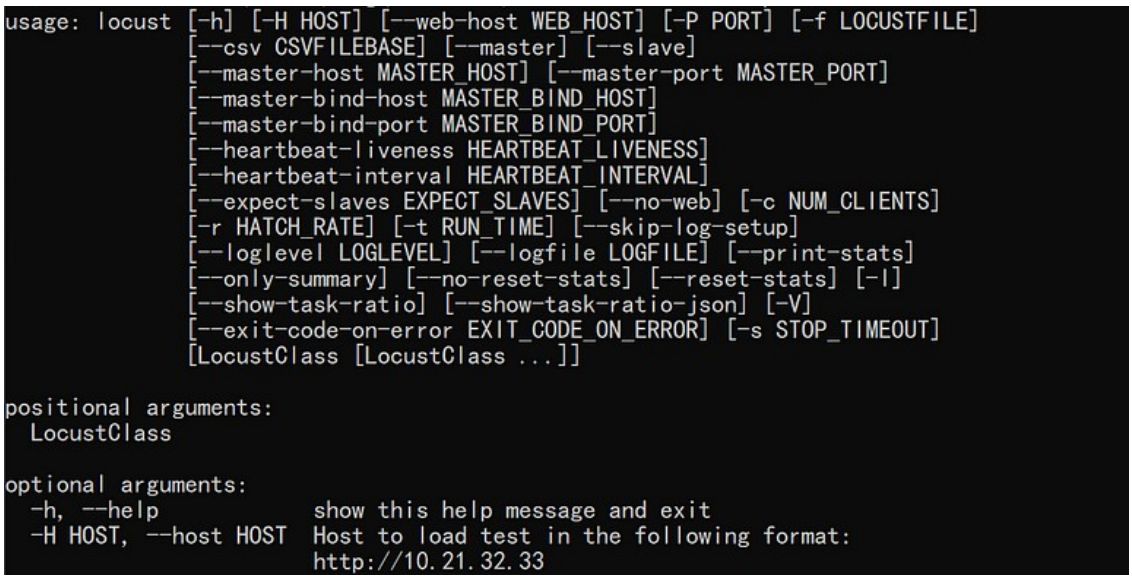
И, наконец, установите Locust с помощью pip install.

Проверка установки Locust

После завершения установки можно его протестировать, запустив в командной строке:

```
locust --help
```

Вы увидите следующий результат:



В следующем разделе мы изучим основные функциональные возможности этого модуля и рассмотрим, как написать тестовый скрипт для нагрузочного тестирования собственного сервера.

3.3.2.2. Базовое использование

Сервер Flask

Допустим, у вас есть файл с Flask сервером, myapp.py, готовый к нагрузочному тестированию:

```
from flask import Flask

server_port = 5000

app = Flask(__name__)
```

```
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == "__main__":
    app.run('0.0.0.0',port=server_port)
```

Просто запустите сервер с помощью кода в командной строке:

```
python myapp.py
```

Вы получите доступ к серверу через localhost:5000 и увидите вывод Hello World.

Скрипт Locust

Создайте новый файл с именем locustfile.py. Добавьте в него оператор import:

```
from locust import HttpUser, task, between
```

Продолжите написание класса, который наследует класс HttpUser. Класс можно назвать в соответствии со своими предпочтениями.

```
class WebsiteTestUser(HttpUser):
```

Внутри класса можно определить собственные функции, которые будут служить заданием для locust. По умолчанию он предоставляет два базовых класса, которые вызываются при старте и остановке вызова Locust:

- on_start
- on_stop

Если поверх функции добавить декоратор @task, то она будет рассматриваться как задача для Locust. В следующем примере показана простая задача, тестирующая API localhost:5000:

```
@task(1)
def hello_world(self):
    self.client.get("http://localhost:5000")
```

Для последующих задач необходимо изменить номер, как показано ниже:

```
@task(1)
def hello_world(self):
    self.client.get("http://localhost:5000")
@task(2)
```

```
def index(self):

    self.client.get("http://localhost:5000/index")
```

Если нужно выполнить POST-запрос, используйте следующий код:

```
self.client.post("/login", {"имя пользователя": "admin", "пароль":
"admin"}).
```

По завершении можно добавить внутрь класса дополнительную фичу, такую как wait_time:

- wait_time — обозначает время ожидания между выполнением задачи. В данный момент будем использовать between(0.5, 3.0), которая указывает на время ожидания между 0,5 и 3 секундами. Доступны и другие встроенные функции, такие как constant и constant_pacing. Позже мы изменим это значение на 0 для нагрузочного тестирования.

```
wait_time = between(0.5, 3.0)
```

Финальный код будет выглядеть примерно так:

```
from locust import HttpUser, task, between


class WebsiteTestUser(HttpUser):

    wait_time = between(0.5, 3.0)


    def on_start(self):

        """ on_start is called when a Locust start before any task is
scheduled """

        pass


    def on_stop(self):

        """ on_stop is called when the TaskSet is stopping """

        pass


    @task(1)

    def hello_world(self):

        self.client.get("http://localhost:5000")
```

Функции on_start и on_stop специально оставлены пустыми. Одно из основных вариантов их использования — это добавление в них вызовов login и logout для проверки аутентификации сайта.

Тест

Давайте протестируем его, открыв другую командную строку и указав в ней каталог, в котором находится locustfile.py. Если вы используете точно такое же имя, то просто выполните команду:

```
locust
```

Если же используется другое имя, пропишите такую команду (соответственно изменив имя файла):

```
locust -f personal_directory/your_file.py
```

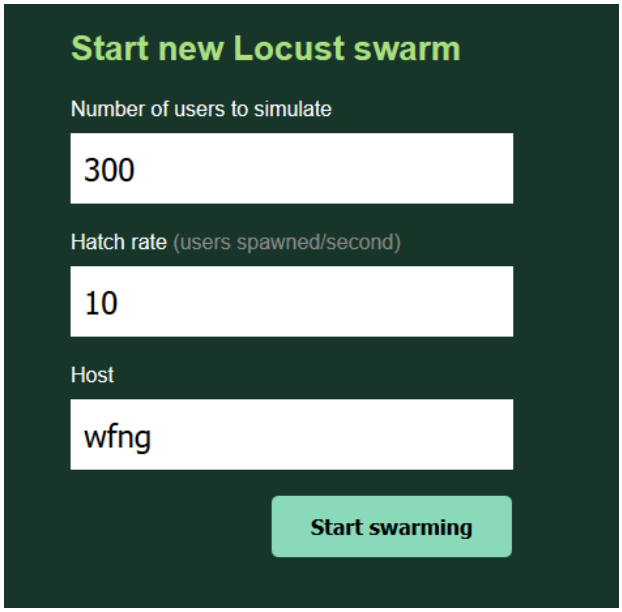
Вы получите следующий результат:

```
[2020-01-03 12:58:11,277] LAPTOP-44CV7055/INFO/locust.main: Starting web monitor at *:8089
[2020-01-03 12:58:11,278] LAPTOP-44CV7055/INFO/locust.main: Starting Locust 0.13.2
```

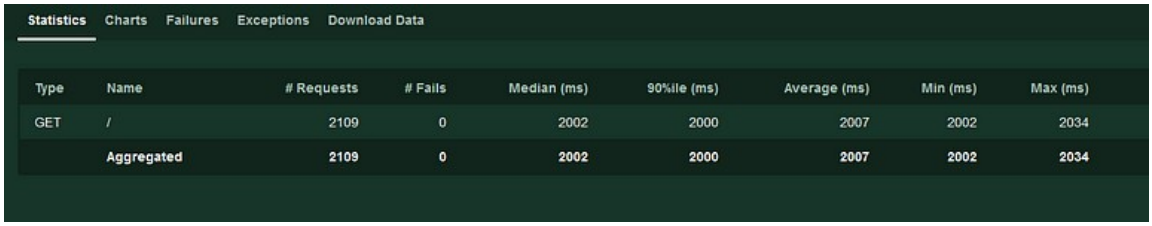
Откройте браузер и введите следующий URL:

http://localhost:8089/

Появится форма, как на картинке ниже. Лучше использовать действительный IP-адрес или имя хоста. На данный момент мы не будем проводить распределенное тестирование, а просто заполним все, что вам нравится.

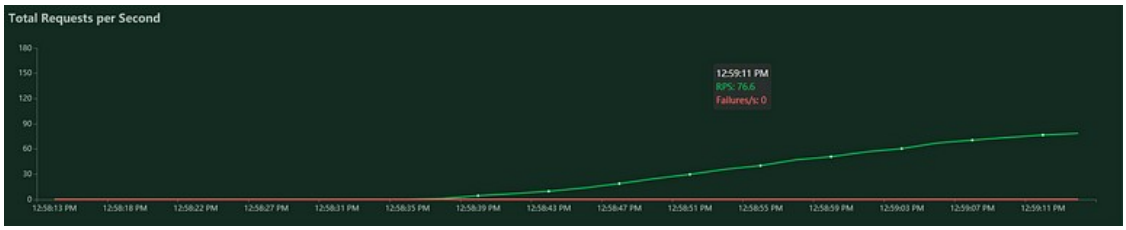


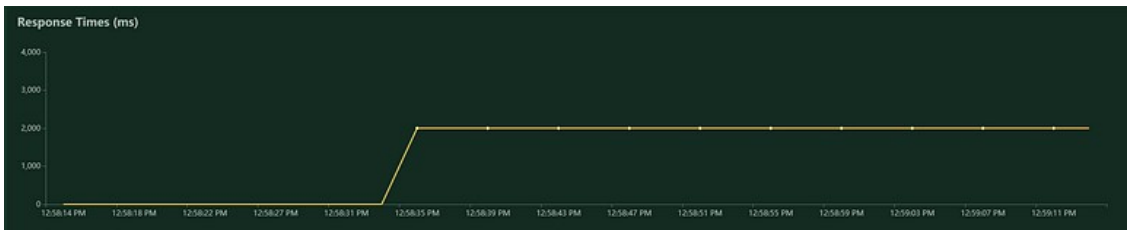
Нажмите кнопку Start swarming (Начать роение), когда все будет готово. Процесс запустится, и вы увидите изменения:

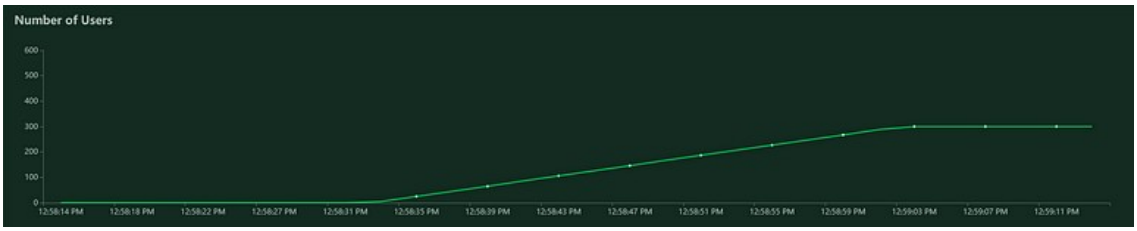


- Request — Общее количество запросов, сделанных на данный момент
- Fails — Количество запросов, которые не были выполнены
- Median — Скорость ответа для 50-го процентиля в мс
- 90%ile — Скорость ответа для 90% перцентиля/мс
- Average — Средняя скорость ответа в мс
- Min — Минимальная скорость ответа в мс
- Max — Максимальная скорость ответа в мс
- Average bytes — Средний размер ответа в байтах
- Current RPS — Текущее количество запросов в секунду
- Current Failure/s — Общее количество отказов в секунду

Перейдите на вкладку "Графики", и вы увидите следующие графики:







Остальное меню предназначено для работы с ошибками и исключениями. На вкладке Download Data можно даже скачать данные в формате csv. Вы можете заметить, что RPS в данный момент довольно низкий. В основном это связано с тем, что мы установили время ожидания (wait_time) в диапазоне от 0,5 до 3 секунд. Для реального нагрузочного тестирования измените оба значения на 0. Также можно

использовать функции constant(0) в зависимости от ваших предпочтений. Перезапустите тест, и вы увидите максимальное значение RPS, которое может выдержать ваш сервер. Попробуйте поэкспериментировать, используя различное количество пользователей (number of users) и скорость загрузки (hatch rate).

Следующий раздел посвящен моделированию той же функциональности не через веб-интерфейс, а непосредственно через командную строку.

3.3.3. Интерфейс командной строки (CLI)

Модуль также дает возможность запускать нагрузочное тестирование через CLI, что позволяет легко реализовать автоматизированный процесс. Просто добавьте no-web параметры при запуске locust. Также необходимо указать хост, количество пользователей и скорость загрузки.

```
locust -f locustfile.py --no-web --host wfng -c 1000 -r 100
```

- f — путь к файлу
- no-web — Запуск моделирования без веб-интерфейса
- c — Количество пользователей
- r — Скорость загрузки (Hatch rate)

Запустите программу на 30 секунд и остановите ее с помощью команды Ctrl-C. Вы увидите следующие выходные данные:

```
Name                                                          # reqs      # fails    Avg     Min     Max  |  Median   req/s failures/s
--------------------------------------------------------------------------------------------------------------------------------------------
GET /                                                          14589     0(0.00%)   2612    2002    6700  |    2200  321.89    0.00
--------------------------------------------------------------------------------------------------------------------------------------------
Aggregated                                                     14589     0(0.00%)   2612    2002    6700  |    2200  321.89    0.00

Percentage of the requests completed within given times
 Name                                                         # reqs    50%    66%    75%    80%    90%    95%    98%    99%  99.9% 99.99% 100%
--------------------------------------------------------------------------------------------------------------------------------------------
 GET /                                                         14589   2200   2600   2900   3200   3700   4400   5000   5300   6000   6400  6700
--------------------------------------------------------------------------------------------------------------------------------------------
 Aggregated                                                    14589   2200   2600   2900   3200   3700   4400   5000   5300   6000   6400  6700
```

Есть и другие полезные параметры, например:

- t — Остановка через определенное время. 30 с, 5 м, 1 ч, 1 ч 30 м.
- csv — Сохранить результат в файлы в формате CSV.

Ознакомиться со всеми доступными параметрами можно с помощью команды:

```
locust --help
```

3.4. UI тестирование

Для проведения тестирования UI предлагается использовать NightWatch [5].

Пример настройки NightWatch представлен в [11].

Также популярным средством тестирования веб-интерфейсов является Selenium [13, 14].

Ниже вы видите пример теста написанного на языке JavaScript с использованием библиотеки NightWatch.js .

Для установки данной библиотеки выполните в командной строке команду

```
$ npm install nightwatch --save-dev
```

Тест вы можете запустить с помощью команды

```
$ npm test
```

Репозиторий данной библиотеки находится на GitHub [15].

Рассмотрим написанный (см. рисунок 24) тест более подробно. В данном тесте исследуется приложение reactjs.org

Ключевым словом browser мы говорим системе начать эмуляцию браузера и далее с помощью команды .url() показываем на какую страницу ей необходимо перейти.

Далее с помощью команды waitForElementVisible() мы дожидаемся загрузки интересующего нас элемента. Путь к данному элементы необходимо указать с помощью css-селекторов, вторым аргументом передается время ожидания данного элемента.

После чего с помощью команды clickLinkContaingText() мы говорим браузеру нажать на элемент сайта с текстом ‘Get Started’. Далее с помощью уже известной нам команды waitForElementVisible() ожидаем загрузки интересующего нас элемента. После чего с помощью функции assert.ContainsText() проверяем текст указанный в интересующем нас элементе. Адресация к элементу идет с помощью css-селекторов.

Далее показываем что необходимо закончить данный тест с помощью команды end()

Такой тест можно написать довольно большой, и содержать в себе множество вариаций и сценариев взаимодействия с веб приложением.

```
module.exports = {
    'https://reactjs.org/': function (browser) {
        browser
            .url('https://reactjs.org/')
            .waitForElementVisible('.css-1053yfl', 1000)
            .clickLinkContainingText('Get Started')
            .waitForElementPresent('.css-1rwyxsf', 1000)
            .assert.containsText('.css-1rwyxsf', 'Getting Started')
            .end();
    }
};
```

Рисунок 24. Код теста

Результаты выполнения данного теста вы можете наблюдать ниже. (см. рисунок 25) Здесь описана информация о времени которое занял тест, об успешном или неуспешном результате выполнения. А также обо всех этапах, выполненных в результате данного теста.

```
Starting selenium server... started - PID:  3418

[First Test] Test Suite
===========================
Executing the global `beforeEach`

[Fourth Test] Test Suite
============================
Executing the global `beforeEach`

Running:  https://reactjs.org/
  Warn: WaitForElement found 2 elements for selector ".css-1053yfl". Only the first one will be checked.
 ✔ Element <.css-1053yfl> was visible after 64 milliseconds.
 ✔ Element <.css-1rwyxsf> was present after 536 milliseconds.
 ✔ Testing if element <.css-1rwyxsf> contains text: "Getting Started".

OK. 3 assertions passed. (4.182s)

[Second Test] Test Suite
============================
Executing the global `beforeEach`

[Third Test] Test Suite
===========================
Executing the global `beforeEach`

OK. 3  total assertions passed. (4.299s)
```

Рисунок 25. Выполнение теста

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое тестирование, как его выполняют? Какие виды тестирования бывают?
2. Что содержит тестовый вариант?
3. Что такое модульное тестирование? Его цели и выявляемые ошибки.
4. Каков алгоритм модульного тестирования в общем виде? Возможности и средства проведения модульного тестирования?
5. Зачем и как проводят автоматизацию тестирования? Что такое тестовый драйвер и заглушка?
6. Что такое нагрузочное тестирование? Его цели?
7. Каковы параметры нагрузочного тестирования? Какую информацию можно получить по его результатам?
8. Исходные данные и результаты веб-тестов производительности?
9. Что такое профайлинг? Его цели и результаты?
10. Какими утилитами можно выполнять профайлинг?
11. Что такое тестирование пользовательского интерфейса? Его цели?
12. Что такое функциональное тестирование? Его цели и выявляемые ошибки. Основные методы.
13. Что такое структурное тестирование? Его цели и выявляемые ошибки. Основные методы.
14. Как определить покрытие кода? Зачем оно нужно?
15. Основные типы и виды тестирования?
16. Что такое регрессионное, повторное, системное и стрессовое тестирование? Какие виды тестирования еще бывают?
17. Каковы принципы составления тестовых вариантов?

5. СПИСОК ИСТОЧНИКОВ

1. Takeuchi H., Nonaka I. The New New Product Development Game. – Текст. Изображение : электронные // Harvard Business Review : [сайт]. – URL: https://hbr.org/1986/01/the-new-new-product-development-game (дата обращения: 29.04.2022)
2. Разработка нового продукта. Новые правила игры. – The New New Product Development Game. – Текст. Изображение : электронные // Agile Russia : [сайт]. – URL: http://agilerussia.ru/methodologies/разработка-нового-продукта-новые-пра/ (дата обращения: 29.04.2022)
3. Static Code Analysis of .NET Core Projects with SonarCloud (раздел про сонар). – Текст : электронные : [сайт] – URL: https://dotnetthoughts.net/static-code-analysis-of-netcore-projects (дата обращения: 29.04.2022)
4. wrk — репозиторий проекта на Github — интернет ресурс : электронные : [сайт] — URL: https://github.com/wg/wrk (дата обращения: 29.04.2022)
5. NightWatch — основная страница проекта — интернет-ресурс : электронные : [сайт] — URL: https://nightwatchjs.org/ (дата обращения: 29.04.2022)
6. Как настроить простую систему автотестов без Java и Selenium — habr.com, Сергей Жирков — статья : электронные : [сайт] — URL: https://habr.com/ru/post/329660/ (дата обращения 29.04.2022)
7. Как провести нагрузочное тестирование – FirstVDS – статья : электронные : [сайт] — URL: https://firstvds.ru/technology/kak-provesti-nagruzochnoe-testirovanie (дата обращения 29.04.2022)
8. PageSpeed Insights — основная страница проекта — интернет-ресурс : электронные : [сайт] — URL: https://pagespeed.web.dev/?hl=ru (дата обращения: 29.04.2022)
9. Locust — Документация проекта — интернет-ресурс : электронные : [сайт] — URL: https://docs.locust.io/en/stable/ (дата обращения: 29.04.2022)
10. Введение в Locust: open source инструмент для нагрузочного тестирования на языке Python – Habr – статья : электронные : [сайт] — URL: https://habr.com/ru/companies/otus/articles/750820/ (дата обращения 29.04.2022)

11. Как настроить простую систему автотестов без Java и Selenium – Habr – статья : электронные : [сайт] — URL: https://habr.com/ru/articles/329660/ (дата обращения 29.04.2022)
12. Профилирование Python — почему и где тормозит ваш код – Habr – статья : электронные : [сайт] — URL: https://habr.com/ru/companies/ruvds/articles/757336/ (дата обращения 29.04.2022)
13. Автоматизация тестирования веб-приложения с использованием Selenium WebDriver, Python, и Behave – Habr – статья : электронные : [сайт] — URL: https://habr.com/ru/articles/262929/ (дата обращения 29.04.2022)
14. Что такое Selenium? – Habr – статья : электронные : [сайт] — URL: https://habr.com/ru/articles/152653/ (дата обращения 29.04.2022)
15. Nightwatch.js— репозиторий проекта на Github — интернет ресурс : электронные : [сайт] — URL: https://github.com/nightwatchjs/nightwatch (дата обращения: 29.04.2022)
16. Профилирование и отладка Python — Habr — статья: электронные : [сайт] — URL: https://habr.com/ru/companies/vk/articles/201594/ (дата обращения: 29.04.2022)
17. Знакомство со средствами профилирования (C#, Visual Basic, C++, F#) — Документация Microsoft — интернет ресурс : электронные : [сайт] — URL: https://learn.microsoft.com/ru-ru/visualstudio/profiling/profiling-feature-tour?view=vs-2022 (дата обращения: 29.04.2022)
18. Performance Profiling JavaScript — Документация VS Code — интернет ресурс : электронные : [сайт] — URL: https://code.visualstudio.com/docs/nodejs/profiling (дата обращения: 29.04.2022)
19. Optimize your code using profilers — Документация PyCharm — интернет ресурс : электронные : [сайт] — URL: https://www.jetbrains.com/help/pycharm/profiler.html (дата обращения: 29.04.2022)